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 07: 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.

Exam Oral, Bonus for Homework!
• ≥ 50% of homework ⇒ 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.
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!

- ≥ 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, ...
- 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)
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)
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

1 Introduction

2 Basics

3 Alternative Representations of Regular Languages

4 Model-Checking concurrent Systems
Tree Automata

- Finite automata recognize words, e.g.:

\[
q_0 \xrightarrow{a} q_F \\
q_F \xrightarrow{b} q_0
\]
Tree Automata

- Finite automata recognize words, e.g.:

\[ q_0 ightarrow a(q_F) \]
\[ q_F ightarrow b(q_0) \]
Tree Automata

- Finite automata recognize words, e.g.:

  \[
  q_0 \rightarrow a(q_F) \quad \text{and} \quad q_F \rightarrow b(q_0)
  \]

- Words of alternating as and bs, ending with a, e.g., \textit{aba} or \textit{abababa}
Tree Automata

- Finite automata recognize words, e.g.:

  \[ q_0 \rightarrow a(q_F) \quad q_F \rightarrow b(q_0) \]

- Words of alternating as and bs, ending with a, e.g., \textit{aba} or \textit{abababa}

- Generalize to trees

  \[ q_0 \rightarrow a(q_1, q_1) \quad q_1 \rightarrow b(q_0, q_0) \quad q_1 \rightarrow L() \]
Tree Automata

- Finite automata recognize words, e.g.:

\[ q_0 \rightarrow a(q_F) \quad q_F \rightarrow b(q_0) \]

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

- Generalize to trees

\[ q_0 \rightarrow a(q_1, q_1) \quad q_1 \rightarrow b(q_0, q_0) \quad q_1 \rightarrow L() \]

- Trees with alternating „layers” of \(a\) nodes and \(b\) nodes.
Tree Automata

- Finite automata recognize words, e.g.:

\[ q_0 \rightarrow a(q_F) \quad \text{and} \quad q_F \rightarrow b(q_0) \]

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

- Generalize to trees

\[ q_0 \rightarrow a(q_1, q_1) \quad q_1 \rightarrow b(q_0, q_0) \quad q_1 \rightarrow L() \]

- Trees with alternating "layers" of \(a\) nodes and \(b\) nodes.
  - Leafs are \(L\)-nodes, as node labels will have fixed arity.
Tree Automata

- Finite automata recognize words, e.g.:

\[ q_0 \rightarrow a(q_F) \quad q_F \rightarrow b(q_0) \]

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

- Generalize to trees

\[ q_0 \rightarrow a(q_1, q_1) \quad q_1 \rightarrow b(q_0, q_0) \quad q_1 \rightarrow L() \]

- Trees with alternating „layers“ of \(a\) nodes and \(b\) nodes.
  - Leafs are \(L\)-nodes, as node labels will have fixed arity.
Tree Automata

- Finite automata recognize words, e.g.:

\[
\begin{align*}
q_0 &\rightarrow a(q_F) \\
q_F &\rightarrow b(q_0)
\end{align*}
\]

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

- Generalize to trees

\[
\begin{align*}
q_0 &\rightarrow a(q_1, q_1) \\
q_1 &\rightarrow b(q_0, q_0) \\
q_1 &\rightarrow L()
\end{align*}
\]

- Trees with alternating „layers” of \(a\) nodes and \(b\) nodes.
  - Leaf nodes are \(L\)-nodes, as node labels will have fixed arity.
Tree Automata

- Finite automata recognize words, e.g.:

  \[
  q_0 \rightarrow a(q_F) \quad q_F \rightarrow b(q_0)
  \]

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

- Generalize to trees

  \[
  q_0 \rightarrow a(q_1, q_1) \quad q_1 \rightarrow b(q_0, q_0) \quad q_1 \rightarrow L()
  \]

- Trees with alternating „layers” of \(a\) nodes and \(b\) nodes.
  - Leaf nodes are \(L\)-nodes, as node labels will have fixed arity.

- 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

1 Introduction

2 Basics

3 Alternative Representations of Regular Languages

4 Model-Checking concurrent Systems
Table of Contents

1 Introduction

2 Basics
   - Nondeterministic Finite Tree Automata
   - Epsilon Rules
   - Deterministic Finite Tree Automata
   - Pumping Lemma
   - Closure Properties
   - Tree Homomorphisms
   - Minimizing Tree Automata
   - Top-Down Tree Automata

3 Alternative Representations of Regular Languages

4 Model-Checking concurrent Systems
Terms and Trees

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

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

Ground terms: $T(\mathcal{F}) := T(\mathcal{F}, \emptyset)$. Terms without variables.
Terms and Trees

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

Ground terms: $\mathcal{T}(\mathcal{F}) := \mathcal{T}(\mathcal{F}, \emptyset)$. Terms without variables.
Terms and Trees

- Let $\mathcal{F}$ be a finite set of symbols, and $\text{arity} : \mathcal{F} \to \mathbb{N}$ 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 $\mathcal{X} \cap \mathcal{F}_0 = \emptyset$. 

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

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

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

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

Ground terms: $T(\mathcal{F}) := T(\mathcal{F}, \emptyset)$. Terms without variables.
Terms and Trees

- Let \( \mathcal{F} \) be a finite set of symbols, and \( \text{arity} : \mathcal{F} \to \mathbb{N} \) 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

\[
T(\mathcal{F}, \mathcal{X}) \supseteq \mathcal{F}_0 \\
T(\mathcal{F}, \mathcal{X}) \supseteq \mathcal{X} \\
p \geq 1, f \in F_p, \text{ and } t_1, \ldots, t_p \in T(\mathcal{F}, \mathcal{X}) \implies f(t_1, \ldots, t_n) \in T(\mathcal{F}, \mathcal{X})
\]

- 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 $\text{arity}: \mathcal{F} \rightarrow \mathbb{N}$ 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
  \[
  T(\mathcal{F}, \mathcal{X}) \supseteq \mathcal{F}_0 \\
  T(\mathcal{F}, \mathcal{X}) \supseteq \mathcal{X} \\
  p \geq 1, f \in F_p, \text{ and } t_1, \ldots, t_p \in T(\mathcal{F}, \mathcal{X}) \implies f(t_1, \ldots, t_n) \in T(\mathcal{F}, \mathcal{X})
  \]
  - 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 $\mathcal{F} = (\{f_1, \ldots, f_n\}, (f_1 \mapsto a_1, \ldots, f_n \mapsto a_n))$
- $\mathcal{F} = \text{true}/0, \text{false}/0, \text{and}/2, \text{not}/1$ - Syntax trees of boolean expressions

- $\text{Suc}(0) + (\text{Suc}(\text{Suc}(0)) \ast x) \in T(F, \{x\})$

- We will use infix-notation for terms when appropriate
Examples

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

• $\mathcal{F} = true/0, false/0, and/2, not/1$ - Syntax trees of boolean expressions
  • $\text{and}(true, \text{not}(x)) \in T(\mathcal{F}, \{x\})$
Examples

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

- $\mathcal{F} = true/0, false/0, and/2, not/1$ - Syntax trees of boolean expressions
  - $\text{and}(true, not(x)) \in T(\mathcal{F}, \{x\})$

- $\mathcal{F} = 0/0, Suc/1, +/2, */2$ - 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 $\mathcal{F} = (\{f_1, \ldots, f_n\}, (f_1 \mapsto a_1, \ldots, f_n \mapsto a_n))$

• $\mathcal{F} = true/0, false/0, and/2, not/1$ - Syntax trees of boolean expressions
  • $\text{and}(\text{true}, \text{not}(x)) \in T(\mathcal{F}, \{x\})$

• $\mathcal{F} = 0/0, \text{Suc}/1, +/2, */2$ - Arithmetic expressions over naturals (using unary representation)
  • $\text{Suc}(0) + (\text{Suc}(\text{Suc}(0)) \times x) \in T(\mathcal{F}, \{x\})$
Examples

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

- $\mathcal{F} = true/0, false/0, and/2, not/1$ - Syntax trees of boolean expressions
  - $\text{and}(true, not(x)) \in T(\mathcal{F}, \{x\})$

- $\mathcal{F} = 0/0, Suc/1, +/2, */2$ - Arithmetic expressions over naturals (using unary representation)
  - $\text{Suc}(0) + (\text{Suc}(\text{Suc}(0)) \times x) \in T(\mathcal{F}, \{x\})$
    - We will use infix-notation for terms when appropriate
• Terms can be identified by trees: Nodes with $p$ successors labeled with symbol from $\mathcal{F}_p$. 
Terms can be identified by trees: Nodes with $p$ successors labeled with symbol from $\mathcal{F}_p$.

$and(true, not(x)) \in T(\mathcal{F}, \{x\})$

```
      /
     /  
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
  ```
- $Suc(0) + (Suc(Suc(0)) \ast x)$
  
  ```
  +
  \ /
  Suc  *
   \   
    \ Suc
     \ 0  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 $A = (Q, \mathcal{F}, Q_f, \Delta)$ where
  - $Q$ is a finite set of states. $Q \cap \mathcal{F}_0 = \emptyset$
Tree Automata

- A (nondeterministic) finite tree automaton (NFTA) over alphabet $\mathcal{F}$ is a tuple $A = (Q, \mathcal{F}, Q_f, \Delta)$ 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 $A = (Q, \mathcal{F}, Q_f, \Delta)$ where
  - $Q$ is a finite set of states. $Q \cap F_0 = \emptyset$
  - $Q_f \subseteq Q$ is a set of final states
  - $\Delta$ is a set of rules of the form
    
    \[ f(q_1, \ldots, q_n) \rightarrow q \]

    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
  - $\Delta$ is a set of rules of the form
    
    $$f(q_1, \ldots, q_n) \rightarrow q$$

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

- Intuition: Use the rules from $\Delta$ 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 $A = (Q, \mathcal{F}, Q_f, \Delta)$ where
  - $Q$ is a finite set of states. $Q \cap F_0 = \emptyset$
  - $Q_f \subseteq Q$ is a set of final states
  - $\Delta$ is a set of rules of the form
    
    $$ f(q_1, \ldots, q_n) \rightarrow q $$

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

- Intuition: Use the rules from $\Delta$ to re-write a given tree to a final state

- For a tree $t \in T(\mathcal{F})$ and a state $q$, we define $t \rightarrow_A q$ as the least relation that satisfies

  $$ f(q_1, \ldots, q_n) \rightarrow q \in \Delta, \forall 1 \leq i \leq n. \ t_i \rightarrow_A q_i \implies f(t_1, \ldots, t_n) \rightarrow_A q $$
Tree Automata

- A (nondeterministic) finite tree automaton (NFTA) over alphabet $\mathcal{F}$ is a tuple $A = (Q, \mathcal{F}, Q_f, \Delta)$ where
  - $Q$ is a finite set of states. $Q \cap F_0 = \emptyset$
  - $Q_f \subseteq Q$ is a set of final states
  - $\Delta$ is a set of rules of the form $f(q_1, \ldots, q_n) \rightarrow q$

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

- Intuition: Use the rules from $\Delta$ to re-write a given tree to a final state

- For a tree $t \in T(\mathcal{F})$ and a state $q$, we define $t \rightarrow_A q$ as the least relation that satisfies

  $$f(q_1, \ldots, q_n) \rightarrow q \in \Delta, \forall 1 \leq i \leq n. \ t_i \rightarrow_A q_i \implies f(t_1, \ldots, t_n) \rightarrow_A q$$

- $t \rightarrow_A q$: Tree $t$ is accepted in state $q$
Tree Automata

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

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

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

- Intuition: Use the rules from $\Delta$ to re-write a given tree to a final state
- For a tree $t \in T(\mathcal{F})$ and a state $q$, we define $t \rightarrow_A q$ as the least relation that satisfies

$$f(q_1, \ldots, q_n) \rightarrow q \in \Delta, \forall 1 \leq i \leq n. t_i \rightarrow_A q_i \implies f(t_1, \ldots, t_n) \rightarrow_A q$$

- $t \rightarrow_A q$: Tree $t$ is accepted in state $q$
- The language $L(A)$ of $A$ are all trees accepted in final states

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

- Tree automaton accepting arithmetic expressions that evaluate to even numbers

\[ \mathcal{F} = 0/0, \text{Suc}/1, +/2 \]
\[ Q := \{e, o\} \]
\[ Q_f = \{e\} \]
\[ 0 \rightarrow e \]
\[ \text{Suc}(e) \rightarrow o \]
\[ \text{Suc}(o) \rightarrow e \]
\[ e + e \rightarrow e \]
\[ e + o \rightarrow o \]
\[ o + e \rightarrow o \]
\[ o + o \rightarrow e \]
Example

• Tree automaton accepting arithmetic expressions that evaluate to even numbers

\[ F = 0/0, \ Suc/1, +/2 \]

\[ Q := \{ e, o \} \quad \quad Q_f = \{ e \} \]

\[ 0 \rightarrow e \quad \quad Suc(e) \rightarrow o \quad \quad Suc(o) \rightarrow e \]

\[ e + e \rightarrow e \quad \quad e + o \rightarrow o \quad \quad o + e \rightarrow o \quad \quad o + o \rightarrow e \]

• 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. $t \xrightarrow{A} t'$ 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. $t \xrightarrow{\mathcal{A}} t'$ 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

1 Introduction

2 Basics
   - Nondeterministic Finite Tree Automata
   - Epsilon Rules
   - Deterministic Finite Tree Automata
   - Pumping Lemma
   - Closure Properties
   - Tree Homomorphisms
   - Minimizing Tree Automata
   - Top-Down Tree Automata

3 Alternative Representations of Regular Languages

4 Model-Checking concurrent Systems
Epsilon rules

- As for word automata, we may add $\epsilon$-rules of the form

$$q \rightarrow q' \text{ for } q, q' \in Q$$
Epsilon rules

- As for word automata, we may add $\epsilon$-rules of the form

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

- The acceptance relation is extended accordingly

$$f(q_1, \ldots, q_n) \rightarrow q \in \Delta, \forall 1 \leq i \leq n. \ t_i \rightarrow_A q_i \implies f(t_1, \ldots, t_n) \rightarrow_A q$$

$$q \rightarrow q' \in \Delta, t \rightarrow_A q \implies t \rightarrow_A q'$$
Epsilon rules

- As for word automata, we may add $\epsilon$-rules of the form

  \[ q \rightarrow q' \text{ for } q, q' \in Q \]

- The acceptance relation is extended accordingly

  \[ f(q_1, \ldots, q_n) \rightarrow q \in \Delta, \forall 1 \leq i \leq n. \; t_i \rightarrow_{\mathcal{A}} q_i \implies f(t_1, \ldots, t_n) \rightarrow_{\mathcal{A}} q \]

  \[ q \rightarrow q' \in \Delta, \; t \rightarrow_{\mathcal{A}} q \implies t \rightarrow_{\mathcal{A}} q' \]

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

  \begin{align*}
    0 & \rightarrow q_n \\
    \text{nil} & \rightarrow q_l \\
    q' & \rightarrow q_l \\
    Suc(q_n) & \rightarrow q_n \\
    cons(q_n, q_l) & \rightarrow q'_l
  \end{align*}
Epsilon rules

- As for word automata, we may add $\epsilon$-rules of the form
  \[ q \rightarrow q' \text{ for } q, q' \in Q \]

- The acceptance relation is extended accordingly
  \[
  f(q_1, \ldots, q_n) \rightarrow q \in \Delta, \forall 1 \leq i \leq n. \ t_i \rightarrow_A q_i \implies f(t_1, \ldots, t_n) \rightarrow_A q \\
  q \rightarrow q' \in \Delta, t \rightarrow_A q \implies t \rightarrow_A q'
  \]

- Example: (Non-empty) lists of natural numbers
  
  \[
  0 \rightarrow q_n \\
  \text{Suc}(q_n) \rightarrow q_n \\
  \text{nil} \rightarrow q_i \\
  \text{cons}(q_n, q_i) \rightarrow q'_i \\
  q'_i \rightarrow q_i
  \]

- Last rule converts non-empty list ($q'_i$) to list ($q_i$)
Epsilon rules

- As for word automata, we may add $\epsilon$-rules of the form
  \[ q \rightarrow q' \text{ for } q, q' \in Q \]

- The acceptance relation is extended accordingly
  \[ f(q_1, \ldots, q_n) \rightarrow q \in \Delta, \forall 1 \leq i \leq n. \; t_i \rightarrow_A q_i \implies f(t_1, \ldots, t_n) \rightarrow_A q \]
  \[ q \rightarrow q' \in \Delta, t \rightarrow_A q \implies t \rightarrow_A q' \]

- Example: (Non-empty) lists of natural numbers
  \[ 0 \rightarrow q_n \]
  \[ Suc(q_n) \rightarrow q_n \]
  \[ nil \rightarrow q_l \]
  \[ cons(q_n, q_l) \rightarrow q'_l \]
  \[ q'_l \rightarrow q_l \]

- 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 $\epsilon$ - rules

**Theorem**

*For a NFTA $A$ with $\epsilon$-rules, there is a NFTA without $\epsilon$-rules that recognizes the same language*

- Proof sketch:
Equivalence of NFTAs with and without $\epsilon$ - rules

Theorem

For a NFTA $A$ with $\epsilon$-rules, there is a NFTA without $\epsilon$-rules that recognizes the same language

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

$$q \in cl(q) \quad q' \in cl(q), q' \rightarrow q'' \implies q'' \in cl(q)$$
Equivalence of NFTAs with and without $\epsilon$ - rules

**Theorem**

*For a NFTA $A$ with $\epsilon$-rules, there is a NFTA without $\epsilon$-rules that recognizes the same language.*

- **Proof sketch:**
  - Let $cl(q)$ denote the $\epsilon$-closure of $q$
    \[
    q \in cl(q), \quad q' \in cl(q), \quad q' \rightarrow q'' \quad \implies \quad q'' \in cl(q)
    \]
  - Define $\Delta' := \{ f(q_1, \ldots, q_n) \rightarrow q' \mid f(q_1, \ldots, q_n) \rightarrow q \in \Delta \wedge q' \in cl(q) \}$
Theorem

For a NFTA $A$ with $\epsilon$-rules, there is a NFTA without $\epsilon$-rules that recognizes the same language.

- Proof sketch:
  - Let $cl(q)$ denote the $\epsilon$-closure of $q$
    \[
    q \in cl(q) \quad q' \in cl(q), q' \rightarrow q'' \implies q'' \in cl(q)
    \]
  - Define $\Delta' := \{ f(q_1, \ldots, q_n) \rightarrow q' \mid f(q_1, \ldots, q_n) \rightarrow q \in \Delta \land q' \in cl(q) \}$
  - Define $A' := (Q, \mathcal{F}, Q_f, \Delta')$
Equivalence of NFTAs with and without \( \epsilon \) - rules

**Theorem**

*For a NFTA \( A \) with \( \epsilon \)-rules, there is a NFTA without \( \epsilon \)-rules that recognizes the same language.*

- **Proof sketch:**
  - Let \( cl(q) \) denote the \( \epsilon \)-closure of \( q \)

\[
q \in cl(q) \quad q' \in cl(q), q' \rightarrow q'' \implies q'' \in cl(q)
\]

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

From now on, we assume tree automata without \( \epsilon \)-rules, unless noted otherwise.
Equivalence of NFTAs with and without $\epsilon$ - rules

Theorem

For a NFTA $A$ with $\epsilon$-rules, there is a NFTA without $\epsilon$-rules that recognizes the same language

Proof sketch:

- Let $cl(q)$ denote the $\epsilon$-closure of $q$

\[ q \in cl(q) \quad q' \in cl(q), q' \rightarrow q'' \implies q'' \in cl(q) \]

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

- Define $A' := (Q, \mathcal{F}, Q_f, \Delta')$

- Show: $t \rightarrow_A q$ iff $t \rightarrow_{A'} q$

  - on board
Equivalence of NFTAs with and without $\epsilon$ - rules

**Theorem**

For a NFTA $A$ with $\epsilon$-rules, there is a NFTA without $\epsilon$-rules that recognizes the same language.

- **Proof sketch:**
  - Let $cl(q)$ denote the $\epsilon$-closure of $q$

  $q \in cl(q)$ \quad $q' \in cl(q), q' \rightarrow q'' \implies q'' \in cl(q)$

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

- From now on, we assume tree automata without $\epsilon$-rules, unless noted otherwise.
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

1 Introduction

2 Basics
   Nondeterministic Finite Tree Automata
   Epsilon Rules
   Deterministic Finite Tree Automata
   Pumping Lemma
   Closure Properties
   Tree Homomorphisms
   Minimizing Tree Automata
   Top-Down Tree Automata

3 Alternative Representations of Regular Languages

4 Model-Checking concurrent Systems
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 $\epsilon$-rules), i.e.

\[ l \rightarrow q_1 \in \Delta \land l \rightarrow q_2 \in \Delta \implies q_1 = q_2 \]
Let $A = (Q, F, Q_f, \Delta)$ be a finite tree automaton.

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

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

- For a DFTA, every tree is accepted in at most one state
Deterministic Finite Tree Automata

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

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

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

- For a DFTA, every tree is accepted in at most one state
- $A$ is complete, if for every $f \in F_n, q_1, \ldots, q_n \in Q$, there is a rule $f(q_1, \ldots, q_n) \rightarrow q$
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 $\epsilon$-rules), i.e.

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

- 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
Deterministic Finite Tree Automata

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

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

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

- 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
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 $\epsilon$-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
  - 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 $t \to_{\mathcal{A}} q$. 
Deterministic Finite Tree Automata

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

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

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

- For a DFTA, every tree is accepted in at most one state
- $A$ 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 $t \rightarrow_A q$.
- $A$ is **reduced**, if all states in $Q$ are accessible.
Membership Test for DFTA

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

\[
\text{acc } (f(t_1, \ldots, t_n)) = \\
\text{let } q_1 = \text{acc } t_1; \ldots; q_n = \text{acc } t_n \\
\text{in } \\
\text{the } q \text{ with } f(q_1, \ldots, q_n) \in \Delta
\]
Membership Test for DFTA

- Complete DFTAs have a simple (and efficient) membership test
  \[
  \text{acc} \left( f(t_1, \ldots, t_n) \right) = \\
  \text{let} \\
  q_1 = \text{acc} \ t_1; \ldots; q_n = \text{acc} \ t_n \\
  \text{in} \\
  \text{the } q \text{ with } f(q_1, \ldots, q_n) \in \Delta
  \]

- 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.

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

\[
\begin{align*}
A & := \emptyset \\
\text{repeat} & \\
A & := a \cup \{q\} \text{ for } q \text{ with } \\
f(q_1, \ldots, q_n) & \rightarrow q \in \Delta, q_1, \ldots, q_n \in A \\
\text{until} & \text{ no more states can be added to } A
\end{align*}
\]
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

\[
A := \emptyset \\
\text{repeat} \\
\quad A := a \cup \{q\} \text{ for } q \text{ with } \\
\quad \quad f(q_1, \ldots, q_n) \rightarrow q \in \Delta, q_1, \ldots, q_n \in A \\
\text{until no more states can be added to } A
\]

- 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

\[
A := \emptyset \\
\text{repeat} \\
\quad A := a \cup \{q\} \text{ for } q \text{ with } f(q_1, \ldots, q_n) \rightarrow q \in \Delta, q_1, \ldots, q_n \in A \\
\text{until no more states can be added to } A
\]

- 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

\[
A := \emptyset \\
\textbf{repeat} \\
\textbf{A := } a \cup \{q\} \textbf{ for } q \textbf{ with} \\
f(q_1, \ldots, q_n) \rightarrow q \in \Delta, q_1, \ldots, q_n \in A \\
\textbf{until } \text{ no more states can be added to A}
\]

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

\[
A := \emptyset \\
\text{repeat} \\
A := a \cup \{q\} \text{ for } q \text{ with } f(q_1, \ldots, q_n) \rightarrow q \in \Delta, q_1, \ldots, q_n \in A \\
\text{until} \text{ no more states can be added to } A
\]

- Proof sketch
  - Invariant: All states in A are accessible.
  - If there is an accessible state not in A, saturation is not complete
    - Induction on \( t \rightarrow_A q \)
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 $Q_{df} := \{ s \in Q_d | s \cap Q_f \neq \emptyset \}$

$$A_d := (Q_d, F, Q_{df}, \Delta_d)$$

Idea: $A_d$ accepts tree $t$ in the set of all states in that $A$ accepts $t$ (maybe the empty set).

Formally: $t \rightarrow A_d s \iff s = \{ q \in Q | t \rightarrow A q \}$

Lemma: The automaton $A_d$ is a complete DFTA, and we have $L(A) = L(A_d)$.

Theorem follows from this.
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 $s = \{q \in Q \mid \exists q_1 \in s_1, \ldots, q_n \in s_n \mid f(q_1, \ldots, q_n) \rightarrow q \in \Delta\}$. 

Lemma: The automaton $A_d$ is a complete DFTA, and we have $L(A) = L(A_d)$.

Theorem follows from this.
Determinization (Powerset construction)

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

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

$s = \{ q \in Q \mid \exists q_1 \in s_1, \ldots, q_n \in s_n \mid f(q_1, \ldots, q_n) \rightarrow q \in \Delta \}$

Define $A_d := (Q_d, \mathcal{F}, Q_{df}, \Delta_d)$

Idea: $A_d$ accepts tree $t$ in the set of all states in that $A$ accepts $t$ (maybe the empty set)

- Formally: $t \rightarrow_{A_d} s$ iff $s = \{ q \in Q \mid t \rightarrow_A q \}$

Lemma: The automaton $A_d$ is a complete DFTA, and we have $L(A) = L(A_d)$. (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 $f(s_1, \ldots, s_n) \rightarrow s \in \Delta_d$ iff
  $s = \{ q \in Q \mid \exists q_1 \in s_1, \ldots, q_n \in s_n \mid f(q_1, \ldots, q_n) \rightarrow q \in \Delta \}$
- Define $A_d := (Q_d, \mathcal{F}, Q_{df}, \Delta_d)$
- Idea: $A_d$ accepts tree $t$ in the set of all states in that $A$ accepts $t$ (maybe the empty set)
  - Formally: $t \rightarrow_{A_d} s$ iff $s = \{ q \in Q \mid t \rightarrow_A q \}$
- Lemma: The automaton $A_d$ is a complete DFTA, and we have $L(A) = L(A_d)$. (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 $A_d$
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 $A_d$

\[
Q_d := \emptyset \\
\Delta_d := \emptyset \\
\text{repeat} \\
\quad Q_d := Q_d \cup \{s\} \\
\quad \Delta_d := \Delta_d \cup \{f(s_1, \ldots, s_n) \rightarrow s\} \\
\quad \text{where} \\
\quad \quad f \in F_n, s_1 \ldots, s_n \in Q_d \\
\quad \quad s = \{q \in Q \mid \exists q_1 \in s_1, \ldots, q_n \in s_n. \ f(q_1, \ldots, q_n) \rightarrow q \in \Delta\} \\
\text{until} \quad \text{No more rules can be added to } \Delta_d \\
Q_{df} := \{s \in Q_d \mid s \cap Q_f \neq \emptyset\} \\
A_d := (Q_d, 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 $F = f/1, g/1, a/0$
- Consider the language $L_n := \{ t \in T(F) \mid \text{The } n\text{th symbol of } t \text{ is } f \}$
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 $\Delta$

\[
\begin{align*}
a &\rightarrow q \\
f(q) &\rightarrow q \\
f(q) &\rightarrow q_1 \\
f(q_i) &\rightarrow q_{i+1} \\
g(q_i) &\rightarrow q_{i+1}
\end{align*}
\]

for $i < n$

- 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

$L = (a + b)^* a (a + b)^n$
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 $\Delta$
    - $a \rightarrow q$
    - $f(q) \rightarrow q$
    - $g(q) \rightarrow q$
    - $f(q) \rightarrow q_1$
    - $f(q_i) \rightarrow q_{i+1}$
    - $g(q_i) \rightarrow q_{i+1}$ for $i < n$
- 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 $\Delta$

\[
\begin{align*}
a &\rightarrow q \\
f(q) &\rightarrow q \\
Q_f &\rightarrow q_1 \\
f(q_i) &\rightarrow q_{i+1} \\
g(q) &\rightarrow q \\
g(q_i) &\rightarrow q_{i+1} \quad \text{for } i < n
\end{align*}
\]

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

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

  \[
  a \rightarrow q \quad f(q) \rightarrow q \quad g(q) \rightarrow q \\
  f(q) \rightarrow q_1 \\
  f(q_i) \rightarrow q_{i+1} \quad g(q_i) \rightarrow q_{i+1} \quad \text{for } i < n
  \]

  • 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 $\Delta$

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

- 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 $L_n := \{ t \in T(\mathcal{F}) | \text{The } n\text{th symbol of } t \text{ is } f \}$
  - Automaton $Q = \{q, q_1, \ldots, q_n\}$, $Q_f = \{q_n\}$ and $\Delta$
    
    $a \rightarrow q$  
    $f(q) \rightarrow q$  
    $g(q) \rightarrow q$  
    $f(q) \rightarrow q_1$  
    $f(q_i) \rightarrow q_{i+1}$  
    $g(q_i) \rightarrow q_{i+1}$ for $i < n$

- 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
  - $L = (a + b)^* a(a + b)^n$
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 $A$ with $L(A) = L$ 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
  - $g^i(a) \rightarrow_{\mathcal{A}} q$ and $g^j(a) \rightarrow_{\mathcal{A}} q$ for $i \neq j$
Example

- Consider the language \( L := \{ f(g^i(a), g^{i+1}(a)) \mid i \in \mathbb{N} \} \)
- Not recognizable by an FTA.
- Assume we have \( A \) with \( L(A) = L \) and \( |Q| = n \)
- During recognizing \( g^{n+1}(a) \), the same state must occur twice, say
  - \( g^i(a) \to_A q \) and \( g^j(a) \to_A q \) for \( i \neq j \)
- As \( f(g^i(a), g^j(a)) \in L(A) \), we also have \( f(g^i(a), g^j(a)) \in L(A) \)
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
  - $g^i(a) \xrightarrow{\mathcal{A}} q$ and $g^j(a) \xrightarrow{\mathcal{A}} q$ for $i \neq j$
- As $f(g^i(a), g^i(a)) \in L(\mathcal{A})$, we also have $f(g^i(a), g^i(a)) \in L(\mathcal{A})$
- 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 $\text{Height}(t) > k$, there is a context $C$, a non-trivial context $C'$, and a term $u$ such that

$$ t = C[C'[u]] $$

$$ \forall n \geq 0. \ C[C'^n[u]] \in L $$

- 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

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

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

Theorem

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

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

Proof sketch:

- Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ with $L = L(\mathcal{A})$, and $t \xrightarrow{\mathcal{A}} q$, $q \in Q_f$
- 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 $\text{Height}(t) > k$, there is a context $C$, a non-trivial context $C'$, and a term $u$ such that

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

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

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

$$t = C[C'[u]]$$

$$\forall n \geq 0. \ C[C'^n[u]] \in L$$

Proof sketch:

- Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ with $L = L(\mathcal{A})$, and $t \xrightarrow{\mathcal{A}} q, q \in Q_f$
- 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 $\text{height}(t) > k$
Example

- Consider $\mathcal{F} = f/2, a/0$, and $L := \{ t \in T(\mathcal{F}) | |t| \text{ is prime}\}$
  - $|t|$ 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 $\text{height}(t) > k$
  - 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 $\text{height}(t) > k$
  - 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 \( \exists t \in L(\mathcal{A}). \text{height}(t) \leq |Q| \)
Corollaries

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

Proof ideas:
1. Remove duplicate states of accepting run repeatedly
2. Take $t \in L(\mathcal{A})$ high enough. Remove duplicate states repeatedly, until longest path has exactly one duplication.
3. $\Rightarrow$: Pump with infinitely many
Corollaries

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

- Proof ideas:
Corollaries

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

- 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 $\exists t \in L(\mathcal{A}). \text{height}(t) \leq |Q|$
  2. $L(\mathcal{A})$ is infinite, iff $\exists t \in L(\mathcal{A}). |Q| < \text{height}(t) \leq 2|Q|$

• Proof ideas:
  1. Remove duplicate states of accepting run repeatedly
  2. $\implies$: 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 $\exists t \in L(\mathcal{A}). \text{height}(t) \leq |Q|$
  2. $L(\mathcal{A})$ is infinite, iff $\exists t \in L(\mathcal{A}). |Q| < \text{height}(t) \leq 2|Q|$

- 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

1 Introduction

2 Basics
   Nondeterministic Finite Tree Automata
   Epsilon Rules
   Deterministic Finite Tree Automata
   Pumping Lemma
   Closure Properties
   Tree Homomorphisms
   Minimizing Tree Automata
   Top-Down Tree Automata

3 Alternative Representations of Regular Languages

4 Model-Checking concurrent Systems
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 $A_1 = (Q_1, F, Q_{f_1}, \Delta_1)$ and $A_2 = (Q_2, F, Q_{f_2}, \Delta_2)$. 
  - Let $A = (Q_1 \cup Q_2, F, Q_{f_1} \cup Q_{f_2}, \Delta_1 \cup \Delta_2)$.
  - Straightforward: $L(A) = L(A_1) \cup L(A_2)$.
  - However: $A$ may be nondeterministic and not complete, even if $A_1$ and $A_2$ were.
  - Let $A_1$, $A_2$ be deterministic and complete. Let $A = (Q, F, Q_{f}, \Delta)$ with $Q = Q_1 \times Q_2$, $Q_f = Q_{f_1} \times Q_2 \cup Q_1 \times Q_{f_2}$, and $\Delta = \Delta_1 \times \Delta_2$ where $\Delta_1 \times \Delta_2 : = \{ f((q_1, q_{1}'),..., (q_n, q_{n}')) \rightarrow (q, q_{}) | f(q_1, ..., q_n) \rightarrow q \in \Delta_1 \land f(q_{1}', ..., q_{n}') \rightarrow q' \in \Delta_2 \}$.
  - Then $L(A) = L(A_1) \cup L(A_2)$ and $A$ is deterministic and complete.

Intuition: Recognize with both automata in parallel.
Union

- Given automata $A_1 = (Q_1, F, Q_{f1}, \Delta_1)$ and $A_2 = (Q_2, F, Q_{f2}, \Delta_2)$.
- Assume, wlog, $Q_1 \cap Q_2 = \emptyset$
Union

- Given automata $A_1 = (Q_1, \mathcal{F}, Q_{f_1}, \Delta_1)$ and $A_2 = (Q_2, \mathcal{F}, Q_{f_2}, \Delta_2)$.
  - Assume, wlog, $Q_1 \cap Q_2 = \emptyset$
  - Let $A = (Q_1 \cup Q_2, \mathcal{F}, Q_{f_1} \cup Q_{f_2}, \Delta_1 \cup \Delta_2)$
Union

- Given automata $A_1 = (Q_1, \mathcal{F}, Q_{f_1}, \Delta_1)$ and $A_2 = (Q_2, \mathcal{F}, Q_{f_2}, \Delta_2)$.
  - Assume, wlog, $Q_1 \cap Q_2 = \emptyset$
  - Let $A = (Q_1 \cup Q_2, \mathcal{F}, Q_{f_1} \cup Q_{f_2}, \Delta_1 \cup \Delta_2)$
  - Straightforward: $L(A) = L(A_1) \cup L(A_2)$

- However: $A$ may be nondeterministic and not complete, even if $A_1$ and $A_2$ were.

- Let $A_1$, $A_2$ be deterministic and complete. Let $A = (Q, \mathcal{F}, Q_{f}, \Delta)$ with
  - $Q = Q_1 \times Q_2$
  - $Q_{f} = Q_{f_1} \times Q_{f_2} \cup Q_1 \times Q_{f_2}$
  - $\Delta = \Delta_1 \times \Delta_2$

  - Then $L(A) = L(A_1) \cup L(A_2)$ and $A$ is deterministic and complete.

- Intuition: Recognize with both automata in parallel.
Union

- Given automata \( A_1 = (Q_1, \mathcal{F}, Q_{f_1}, \Delta_1) \) and \( A_2 = (Q_2, \mathcal{F}, Q_{f_2}, \Delta_2) \).
  - Assume, wlog, \( Q_1 \cap Q_2 = \emptyset \)
  - Let \( A = (Q_1 \cup Q_2, \mathcal{F}, Q_{f_1} \cup Q_{f_2}, \Delta_1 \cup \Delta_2) \)
  - Straightforward: \( L(A) = L(A_1) \cup L(A_2) \)
- However: \( A \) may be nondeterministic and not complete, even if \( A_1 \) and \( A_2 \) were.
Given automata $A_1 = (Q_1, \mathcal{F}, Q_{f_1}, \Delta_1)$ and $A_2 = (Q_2, \mathcal{F}, Q_{f_2}, \Delta_2)$.

- Assume, wlog, $Q_1 \cap Q_2 = \emptyset$
- Let $A = (Q_1 \cup Q_2, \mathcal{F}, Q_{f_1} \cup Q_{f_2}, \Delta_1 \cup \Delta_2)$
- Straightforward: $L(A) = L(A_1) \cup L(A_2)$

However: $A$ may be nondeterministic and not complete, even if $A_1$ and $A_2$ were.

Let $A_1, A_2$ be deterministic and complete. Let $A = (Q, \mathcal{F}, Q_f, \Delta)$ with
Union

- Given automata $A_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $A_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$.
  - Assume, wlog, $Q_1 \cap Q_2 = \emptyset$
  - Let $A = (Q_1 \cup Q_2, \mathcal{F}, Q_{f1} \cup Q_{f2}, \Delta_1 \cup \Delta_2)$
  - Straightforward: $L(A) = L(A_1) \cup L(A_2)$

- However: $A$ may be nondeterministic and not complete, even if $A_1$ and $A_2$ were.

- Let $A_1, A_2$ be deterministic and complete. Let $A = (Q, \mathcal{F}, Q_f, \Delta)$ 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

\[
\Delta_1 \times \Delta_2 := \{ f(((q_1, q'_1), \ldots, (q_n, q'_n))) \rightarrow (q, q') \mid f(q_1, \ldots, q_n) \rightarrow q \in \Delta_1 \land f(q'_1, \ldots, q'_n) \rightarrow q' \in \Delta_2 \}
\]
Union

- Given automata $A_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $A_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$.
  - Assume, wlog, $Q_1 \cap Q_2 = \emptyset$
  - Let $A = (Q_1 \cup Q_2, \mathcal{F}, Q_{f1} \cup Q_{f2}, \Delta_1 \cup \Delta_2)$
  - Straightforward: $L(A) = L(A_1) \cup L(A_2)$
- However: $A$ may be nondeterministic and not complete, even if $A_1$ and $A_2$ were.
- Let $A_1, A_2$ be deterministic and complete. Let $A = (Q, \mathcal{F}, Q_f, \Delta)$ 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

  $\Delta_1 \times \Delta_2 := \{ f((q_1, q'_1), \ldots, (q_n, q'_n)) \rightarrow (q, q') \mid f(q_1, \ldots, q_n) \rightarrow q \in \Delta_1 \wedge f(q'_1, \ldots, q'_n) \rightarrow q' \in \Delta_2 \}$

  - Then $L(A) = L(A_1) \cup L(A_2)$ and $A$ is deterministic and complete.
Union

- Given automata $A_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $A_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$.
  - Assume, wlog, $Q_1 \cap Q_2 = \emptyset$
  - Let $A = (Q_1 \cup Q_2, \mathcal{F}, Q_{f1} \cup Q_{f2}, \Delta_1 \cup \Delta_2)$
  - Straightforward: $L(A) = L(A_1) \cup L(A_2)$

- However: $A$ may be nondeterministic and not complete, even if $A_1$ and $A_2$ were.

- Let $A_1, A_2$ be deterministic and complete. Let $A = (Q, \mathcal{F}, Q_f, \Delta)$ 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

  \[
  \Delta_1 \times \Delta_2 := \{ f((q_1, q'_1), \ldots, (q_n, q'_n)) \rightarrow (q, q') \mid \\
  f(q_1, \ldots, q_n) \rightarrow q \in \Delta_1 \land f(q'_1, \ldots, q'_n) \rightarrow q' \in \Delta_2 \}
  \]

  - Then $L(A) = L(A_1) \cup L(A_2)$ and $A$ 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: $L_1 \cap L_2 = \overline{L_1} \cup \overline{L_2}$
Intersection

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

- Product construction: Given automata \( A_1 = (Q_1, F, Q_f, \Delta_1) \) and \( A_2 = (Q_2, F, Q_f, \Delta_2) \).
- Define \( A = (Q_1 \times Q_2, F, Q_f \times Q_f, \Delta_1 \times \Delta_2) \).
- \( L(A) = L(A_1) \cap L(A_2) \)

Intuition: Automata run in parallel. Accept if both accept.

- \( A \) is deterministic/complete if \( A_1 \) and \( A_2 \) are.

Product construction can also be combined with reduction algorithm, to avoid construction of inaccessible states.
Intersection

- The easy way: $L_1 \cap L_2 = \overline{L_1} \cup \overline{L_2}$
- Exponential blowup for NFTA.
- Product construction: Given automata $A_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $A_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$. 

Intuition: Automata run in parallel. Accept if both accept.

$A$ is deterministic/complete if $A_1$ and $A_2$ are.

Product construction can also be combined with reduction algorithm, to avoid construction of inaccessible states.
Intersection

- The easy way: \( L_1 \cap L_2 = \overline{L_1} \cup \overline{L_2} \)
  - Exponential blowup for NFTA.
- Product construction: Given automata \( A_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1) \) and \( A_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2) \).
  - Define \( A = (Q_1 \times Q_2, \mathcal{F}, Q_{f1} \times Q_{f2}, \Delta_1 \times \Delta_2) \)
Intersection

- The easy way: $L_1 \cap L_2 = \overline{L_1} \cup \overline{L_2}$
  - Exponential blowup for NFTA.
- Product construction: Given automata $A_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $A_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$.
  - Define $A = (Q_1 \times Q_2, \mathcal{F}, Q_{f1} \times Q_{f2}, \Delta_1 \times \Delta_2)$
  - $L(A) = L(A_1) \cap L(A_2)$

Intuition: Automata run in parallel. Accept if both accept.

- $A$ is deterministic/complete if $A_1$ and $A_2$ are.
- Product construction can also be combined with reduction algorithm, to avoid construction of inaccessible states.
Intersection

- The easy way: $L_1 \cap L_2 = \overline{L_1} \cup \overline{L_2}$
  - Exponential blowup for NFTA.
- Product construction: Given automata $\mathcal{A}_1 = (Q_1, \mathcal{F}, Q_{f_1}, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, \mathcal{F}, Q_{f_2}, \Delta_2)$.
  - Define $\mathcal{A} = (Q_1 \times Q_2, \mathcal{F}, Q_{f_1} \times Q_{f_2}, \Delta_1 \times \Delta_2)$
  - $L(\mathcal{A}) = L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$
    - Intuition: Automata run in parallel. Accept if both accept.
Intersection

- The easy way: $L_1 \cap L_2 = \overline{L_1} \cup \overline{L_2}$
  - Exponential blowup for NFTA.
- Product construction: Given automata $A_1 = (Q_1, \mathcal{F}, Q_{f_1}, \Delta_1)$ and $A_2 = (Q_2, \mathcal{F}, Q_{f_2}, \Delta_2)$.
  - Define $A = (Q_1 \times Q_2, \mathcal{F}, Q_{f_1} \times Q_{f_2}, \Delta_1 \times \Delta_2)$
  - $L(A) = L(A_1) \cap L(A_2)$
    - Intuition: Automata run in parallel. Accept if both accept.
  - $A$ is deterministic/complete if $A_1$ and $A_2$ are.
Intersection

- The easy way: \( L_1 \cap L_2 = \overline{L_1} \cup \overline{L_2} \)
  - Exponential blowup for NFTA.
- Product construction: Given automata \( A_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1) \) and \( A_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2) \).
  - Define \( A = (Q_1 \times Q_2, \mathcal{F}, Q_{f1} \times Q_{f2}, \Delta_1 \times \Delta_2) \)
  - \( L(A) = L(A_1) \cap L(A_2) \)
    - Intuition: Automata run in parallel. Accept if both accept.
    - \( A \) is deterministic/complete if \( A_1 \) and \( A_2 \) 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 $O(|t| \times |A|)$ On-the-fly determinization.
More Algorithms on FTA

- Membership for NFTA. In time $O(|t| \times |A|)$ On-the-fly determinization.
- Emptiness check: Time $O(|A|)$. Exercise!
Table of Contents

1 Introduction

2 Basics
   Nondeterministic Finite Tree Automata
   Epsilon Rules
   Deterministic Finite Tree Automata
   Pumping Lemma
   Closure Properties
   Tree Homomorphisms
   Minimizing Tree Automata
   Top-Down Tree Automata

3 Alternative Representations of Regular Languages

4 Model-Checking concurrent Systems
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
  - \( x_1 \land x_2 \mapsto \neg(\neg x_1 \lor \neg x_2) \)
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.

- 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 $h_{\mathcal{F}}[q_1, \ldots, q_n]$. 
Tree homomorphisms do not preserve regularity in general

- Let \( L = \{ f(g^i(a)) \mid i \in \mathbb{N} \} \). Obviously regular.
- Let \( h_F : f(x) \mapsto f(x, x) \)
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 \( h_F : f(x) \mapsto f(x, x) \)
  - \( h(L) = \{ f(g^i(a), g^i(a)) \mid i \in \mathbb{N} \} \). 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 $h_{F}: f(x) \mapsto f(x, x)$
  - $h(L) = \{ f(g^i(a), g^i(a)) \mid i \in \mathbb{N} \}$. Not regular.

- But:
Preservation of Regularity

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

- But:
  - A tree homomorphism determined by $h_F$ is linear, iff for all $f \in \mathcal{F}$, the term $h_F(f)$ 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 $h_F[q_1, \ldots, q_n]$
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:

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

- $\text{Pos}(t)$ 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 $A = (Q, F, Q_f, \Delta)$.
- Construct NFTA $A' = (Q', F', Q'_f, \Delta')$:
Construction (Preservation of regularity)

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

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

- Assume $L$ is accepted by reduced DFTA $A = (Q, \mathcal{F}, Q_f, \Delta)$.
- Construct NFTA $A' = (Q', \mathcal{F}', Q'_f, \Delta')$:
  - With $Q \subseteq Q'$ and $Q'_f = Q_f$
  - For each rule $r = f(q_1, \ldots, q_n) \rightarrow q, t_f = h\mathcal{F}(t)$, and position $p \in Pos(t_f)$:
    - States $q'_p \in Q'$
    - If $t_f(p) = g(\ldots) \in \mathcal{F}_k: g(q'_{p_1}, \ldots, q'_{p_k}) \rightarrow q' \in \Delta'$
Construction (Preservation of regularity)

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

- Assume $L$ is accepted by reduced DFTA $A = (Q, \mathcal{F}, Q_f, \Delta)$.
- Construct NFTA $A' = (Q', \mathcal{F}', Q_f', \Delta')$:
  - With $Q \subseteq Q'$ and $Q_f' = Q_f$
  - For each rule $r = f(q_1, \ldots, q_n) \rightarrow q$, $t_f = h_\mathcal{F}(t)$, and position $p \in Pos(t_f)$:
    - States $q_p^r \in Q'$
    - If $t_f(p) = g(\ldots) \in \mathcal{F}_k$: $g(q_{p_1}^r, \ldots, q_{p_k}^r) \rightarrow q^r \in \Delta'$
    - If $t_f(p) = x_i$: $q_i \rightarrow q_p^r \in \Delta'$
    - $q_{e}^r \rightarrow q \in \Delta'$
Proof sketch

- Prove $h(L) \subseteq L(A')$. Straightforward.
Proof sketch

- Prove $h(L) \subseteq L(A')$. Straightforward.
- Prove $L(A') \subseteq h(L)$ (Sketch on board).
Proof sketch

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

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

- Prove $h(L) \subseteq L(A')$. Straightforward.
- Prove $L(A') \subseteq h(L)$ (Sketch on board).
  - Idea: Split derivation of $t \rightarrow_{A'} q \in Q$ at rules of the form $q'_r \rightarrow q$.
  - 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 $q_i \rightarrow q'_r \in \Delta'$, $q_i \in Q$ (use IH)
Proof sketch

• Prove \( h(L) \subseteq L(A') \). Straightforward.

• Prove \( L(A') \subseteq h(L) \) (Sketch on board).
  - Idea: Split derivation of \( t \rightarrow_{A'} q \in Q \) at rules of the form \( q \rightarrow q' \in \Delta' \).
  - 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 \( q_i \rightarrow q'_i \in \Delta' \), \( q_i \in Q \) (use IH)
  - Formally: Induction on size of derivation \( t \rightarrow_{A'} q \)
Last lecture

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

- Motivation: Reconsider elimination of $\land$ in Boolean formulas
Inverse Homomorphism

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

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

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

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

**Theorem**

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

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

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

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

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

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

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
Inverse Homomorphism

- Motivation: Reconsider elimination of $\land$ in Boolean formulas
  - Homomorphism: Given automaton that recognizes true formulas, construct automaton for true formulas without $\land$.
    - Not really useful
  - Inverse homomorphism: Given automaton for formulas without $\land$, construct automaton for formulas with $\land$.
    - 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 $\land$ in Boolean formulas
  - Homomorphism: Given automaton that recognizes true formulas, construct automaton for true formulas without $\land$.
    - Not really useful
  - Inverse homomorphism: Given automaton for formulas without $\land$, construct automaton for formulas with $\land$.
    - 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 $\land$ in Boolean formulas
  - Homomorphism: Given automaton that recognizes true formulas, construct automaton for true formulas without $\land$.
    - Not really useful
  - Inverse homomorphism: Given automaton for formulas without $\land$, construct automaton for formulas with $\land$.
    - 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 $\land$ in Boolean formulas
  - Homomorphism: Given automaton that recognizes true formulas, construct automaton for true formulas without $\land$.
    - Not really useful
  - Inverse homomorphism: Given automaton for formulas without $\land$, construct automaton for formulas with $\land$.
    - 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 $\land$ in Boolean formulas
  - Homomorphism: Given automaton that recognizes true formulas, construct automaton for true formulas without $\land$.
    - Not really useful
  - Inverse homomorphism: Given automaton for formulas without $\land$, construct automaton for formulas with $\land$.
    - 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 $A = (Q, \mathcal{F}, Q_f, \Delta)$ and $t \in T(\mathcal{F} \cup Q)$. 

This is obviously a generalization of the acceptance relation we defined earlier.
Generalized Acceptance Relation

- Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ and $t \in T(\mathcal{F} \cup Q)$.
- We define $t \rightarrow q$ as the least relation that satisfies
  
  $q \rightarrow q$
  $f(q_1, \ldots, q_n) \rightarrow q \in \Delta, \forall i \leq n. \ t_i \rightarrow q_i \implies f(t_1, \ldots, t_n) \rightarrow q$
Generalized Acceptance Relation

- Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ and $t \in T(\mathcal{F} \cup Q)$.
- We define $t \rightarrow_{\mathcal{A}} q$ as the least relation that satisfies
  
  $q \rightarrow_{\mathcal{A}} q$
  
  $f(q_1, \ldots, q_n) \rightarrow q \in \Delta, \forall i \leq n. \ t_i \rightarrow_{\mathcal{A}} q_i \implies f(t_1, \ldots, t_n) \rightarrow_{\mathcal{A}} q$

- This is obviously a generalization of the acceptance relation we defined earlier
Inverse Homomorphism, construction

• Let \( h : T(\mathcal{F}) \rightarrow T(\mathcal{F}') \) be a tree homomorphism determined by \( h_{\mathcal{F}} \)
Inverse Homomorphism, construction

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

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

  $f(q_1, \ldots, q_n) \to q \in \Delta$ if $f \in F_n$, $h_F(f)[p_1, \ldots, p_n] \to A' q$

  where $q_i = p_i$ if $x_i$ occurs in $h_F(f)$, and $q_i = s$ otherwise

  $a \to s \in \Delta$, $f(s, \ldots, s) \to s \in \Delta$
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 \( \mathcal{A} = (Q' \cup \{s\}, \mathcal{F}, Q'_f, \Delta) \), with the rules

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

  \[
  a \to s \in \Delta, \ f(s, \ldots, s) \to s \in \Delta
  \]

- Intuition: Accept node \( f \), if its image is accepted by \( \mathcal{A}' \)
Inverse Homomorphism, construction

- Let $h : T(\mathcal{F}) \rightarrow T(\mathcal{F}')$ 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 $\mathcal{A} = (Q' \cup \{s\}, \mathcal{F}, Q'_f, \Delta)$, with the rules

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

- 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 $t \rightarrow_{\mathcal{A}} q$ iff $h(t) \rightarrow_{\mathcal{A}'} q$
Inverse Homomorphism, proof

- Show $t \to_A q$ iff $h(t) \to_{A'} q$
- On board
Table of Contents

1 Introduction

2 Basics
   Nondeterministic Finite Tree Automata
   Epsilon Rules
   Deterministic Finite Tree Automata
   Pumping Lemma
   Closure Properties
   Tree Homomorphisms
   Minimizing Tree Automata
   Top-Down Tree Automata

3 Alternative Representations of Regular Languages

4 Model-Checking concurrent Systems
Last Lecture

- Inverse homomorphisms preserve regularity
- Started Myhill-Nerode Theorem
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

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

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

• Note: \( \equiv \) is congruence, iff closed under (1-hole) contexts, i.e.

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

• For a language \( L \), we define the congruence \( \equiv_L \) by

\[
\forall C. C[u] \in L \iff C[v] \in L
\]

• Obviously an equivalence relation. Obviously a congruence.

• Intuition: \( L \) does not distinguish between \( u \) and \( v \).
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

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

• Obviously an equivalence relation. Obviously a congruence.

• Intuition: $L$ does not distinguish between $u$ and $v$.
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

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

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

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

• 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

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

• 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

\[
u \equiv_L v \iff \forall C. \ C[u] \in L \iff C[v] \in L
\]

- Obviously an equivalence relation. Obviously a congruence.
- Intuition: \(L\) does not distinguish between \(u\) and \(v\)
Myhill-Nerode Theorem

The following statements are equivalent

1. \( L \) is a regular tree language
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

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 $\Delta$ via
    \[
    f(q_1, \ldots, q_n) \to q \text{ iff } \delta(f, q_1, \ldots, q_n) = q
    \]
- Naturally extended to trees
  \[
  \delta(f(t_1, \ldots, t_n) = \delta(f, \delta(t_1), \ldots, \delta(t_n))
  \]
- Compatible with $\to_A$, i.e.
  \[
  t \to_A q \text{ iff } \delta(t) = q
  \]
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 \( A = (Q, F, Q_f, \delta) \) with \( L = L(A) \).
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

\[ \frac{1 \rightarrow 2}{\text{• Take complete DFTA } A = (Q, \mathcal{F}, Q_f, \delta) \text{ with } L = L(A).} \]
\[ \text{• Let } u \equiv v \text{ iff } \delta(u) = \delta(v) \text{ (Obviously a congruence)} \]
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 $A = (Q, F, Q_f, \delta)$ with $L = L(A)$.
- 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 $\implies$ 2
- Take complete DFTA $A = (Q, F, Q_f, \delta)$ with $L = L(A)$.
- 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

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

2 $\rightarrow$ 3
- Let $R$ be the finite-index congruence. Assume $uRv$.
- Then, $C[u]R[C[v]$ 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 → 2
- Take complete DFTA $A = (Q, F, Q_f, \delta)$ with $L = L(A)$.
- Let $u \equiv v$ iff $\delta(u) = \delta(v)$ (Obviously a congruence)
- $\equiv$ has finite index (at most $|Q|$ equivalence classes)
- We have $L = \bigcup \{ [u] | \delta(u) \in Q_f \}$

2 → 3
- Let $R$ be the finite-index congruence. Assume $uRv$.
- Then, $C[u]R[C[v]$ 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 $\implies$ 2
- Take complete DFTA $A = (Q, \mathcal{F}, Q_f, \delta)$ with $L = L(A)$.
- 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 $\implies$ 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 $\rightarrow$ 2
- Take complete DFTA $A = (Q, \mathcal{F}, Q_f, \delta)$ with $L = L(A)$.
- 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 $R$
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 $A = (Q, F, Q_f, \delta)$ with $L = L(A)$.
- 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 $\equiv_L$
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 $A = (Q, \mathcal{F}, Q_f, \delta)$ with $L = L(A)$.
- Let $u \equiv v$ iff $\delta(u) = \delta(v)$ (Obviously a congruence)
- $\equiv$ has finite index (at most $|Q|$ equivalence classes)
- We have $L = \bigcup\{[u] | \delta(u) \in Q_f\}$

2 $\rightarrow$ 3

- Let $R$ be the finite-index congruence. Assume $uRv$.
- Then, $C[u]R[C[v]$ 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 $\Delta_{min} := \{f([u_1]_{\equiv_L}, \ldots, [u_n]_{\equiv_L}) \rightarrow [f(u_1, \ldots, u_n)]_{\equiv_L} | f \in \mathcal{F}_n, u_1, \ldots, u_n \in T(\mathcal{F})\}$
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 $\rightarrow$ 2
- Take complete DFTA $A = (Q, \mathcal{F}, Q_f, \delta)$ with $L = L(A)$.
- Let $u \equiv v$ iff $\delta(u) = \delta(v)$ (Obviously a congruence)
- $\equiv$ has finite index (at most $|Q|$ equivalence classes)
- We have $L = \bigcup \{[u] | \delta(u) \in Q_f\}$

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 $\Delta_{\min} := \{f([u_1]_{\equiv_L}, \ldots, [u_n]_{\equiv_L}) \rightarrow [f(u_1, \ldots, u_n)]_{\equiv_L} | f \in \mathcal{F}_n, u_1, \ldots, u_n \in T(\mathcal{F})\}$
- Note that $\Delta_{\min}$ 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 $A = (Q, \mathcal{F}, Q_f, \delta)$ with $L = L(A)$.
• 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 $Q_{\text{min}}$ be the set of eq-classes of $\equiv_L$
• Let $\Delta_{\text{min}} := \{ f([u_1]_{\equiv L}, \ldots, [u_n]_{\equiv L}) \rightarrow [f(u_1, \ldots, u_n)]_{\equiv L} \mid f \in \mathcal{F}, u_1, \ldots, u_n \in T(\mathcal{F}) \}$
• Note that $\Delta_{\text{min}}$ is deterministic, as $\equiv_L$ is a congruence
• Let $Q_{\text{min}_f} := \{ [u] \mid u \in L \}$
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 \rightarrow 2
- Take complete DFTA \( A = (Q, \mathcal{F}, Q_f, \delta) \) with \( L = L(A) \).
- Let \( u \equiv v \) iff \( \delta(u) = \delta(v) \) (Obviously a congruence)
- \( \equiv \) has finite index (at most \(|Q|\) equivalence classes)
- We have \( L = \bigcup \{ [u] | \delta(u) \in Q_f \} \)

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 \( \Delta_{min} := \{ f([u_1]_{\equiv_L}, \ldots, [u_n]_{\equiv_L}) \rightarrow [f(u_1, \ldots, u_n)]_{\equiv_L} | f \in \mathcal{F}_n, u_1, \ldots, u_n \in T(\mathcal{F}) \} \)
- Note that \( \Delta_{min} \) is deterministic, as \( \equiv_L \) is a congruence
- Let \( Q_{min_f} := \{ [u] | u \in L \} \)
- The DFTA \( A_{min} := (Q_{min}, \mathcal{F}, Q_{min_f}, \Delta_{min}) \) 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 $A_{\text{min}}$ 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 $A_{min}$ 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 $A_{\text{min}}$ from the proof of Myhill-Nerode
- Proof sketch (more details on board):
  - Assume $L$ is recognized by complete DFTA $A = (Q, F, Q_f, \delta)$
Unique minimal DFTA

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

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

- Proof sketch (more details on board):
  - Assume $L$ is recognized by complete DFTA $A = (Q, F, Q_f, \delta)$
  - The relation $\equiv_A$ is refinement of $\equiv_L$
    - $\equiv_A \subseteq \equiv_L$
  - Thus $|Q| \geq |Q_{\text{min}}|$ (proves existence of minimal DFTA)
  - Now assume $|Q| = |Q_{\text{min}}|$:
    - 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 $A_{\text{min}}$ from the proof of Myhill-Nerode
- Proof sketch (more details on board):
  - Assume $L$ is recognized by complete DFTA $A = (Q, F, Q_f, \delta)$
  - The relation $\equiv_A$ is *refinement* of $\equiv_L$
    - $\equiv_A \subseteq \equiv_L$
  - Thus $|Q| \geq |Q_{\text{min}}|$ (proves existence of minimal DFTA)
  - Now assume $|Q| = |Q_{\text{min}}|$:
    - 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 $A_{min}$ from the proof of Myhill-Nerode

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

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

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

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
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
Minimization algorithm

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

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.
     $$\forall f \in \mathcal{F}, q_1, \ldots, q_{i-1}, q_{i+1}, \ldots q_n.
     \delta(f, q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_n)P\delta(f, q_1, \ldots, q_{i-1}, q', q_{i+1}, \ldots, q_n)$$
3. Repeat until no more refinement possible
4. Define $A_{\text{min}} := (Q_{\text{min}}, \mathcal{F}, Q_{\text{minf}}, \delta)$, where
   - $Q_{\text{min}} :=$ Equivalence classes of $P$
Minimization algorithm

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

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 \( A_{\text{min}} := (Q_{\text{min}}, \mathcal{F}, Q_{\text{minf}}, \delta) \), where
   - \( Q_{\text{min}} := \) Equivalence classes of \( P \)
   - \( Q_{\text{minf}} := \{[q] \mid q \in Q_f\} \)
Minimization algorithm

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

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.
     \[
     \forall f \in F_n, q_1, \ldots, q_{i-1}, q_{i+1}, \ldots q_n.
     \delta(f, q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_n)P \delta(f, q_1, \ldots, q_{i-1}, q', q_{i+1}, \ldots, q_n)
     \]
3. Repeat until no more refinement possible
4. Define $A_{\text{min}} := (Q_{\text{min}}, F, Q_{\text{minf}}, \delta)$, where
   - $Q_{\text{min}} :=$ Equivalence classes of $P$
   - $Q_{\text{minf}} := \{[q] \mid q \in Q_f\}$
   - $\delta_{\text{min}}(f, [q_1], \ldots, [q_n]) = [\delta(f, q_1, \ldots, q_n)]$
Minimization algorithm

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

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 $A_{\min} := (Q_{\min}, \mathcal{F}, Q_{\min f}, \delta)$, where
   - $Q_{\min} :=$ Equivalence classes of $P$
   - $Q_{\min f} := \{[q] \mid q \in Q_f\}$
   - $\delta_{\min}(f, [q_1], \ldots, [q_n]) = [\delta(f, q_1, \ldots, q_n)]$

- $L(A_{\min}) = L(A)$. Proof on board.
Last Lecture

- Myhill-Nerode Theorem
- Minimization of tree automata
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
  - $f(q_1, \ldots, q_n) \rightarrow q$
  - 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
  - \( f(q_1, \ldots, q_n) \rightarrow q \)
  - Intuition: Assign state to a given tree, consume tree

- Now: Rewrite state to a tree
  - Starting at a single root state
  - \( q \rightarrow f(q_1, \ldots, q_n) \)
Top-Down Tree Automata

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

• Now: Rewrite state to a tree
  • Starting at a single root state
  • \( q \rightarrow f(q_1, \ldots, q_n) \)
  • Intuition: Assign tree to given state, produce tree.
Top-Down Tree Automata

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

- \( F \) is a ranked alphabet
- \( Q \) is a finite set of states, with \( Q \cap F = \emptyset \)
- \( I \subseteq Q \) is a set of initial states
- \( \Delta \) is a set of rules of the form

\[
q \rightarrow f(q_1, \ldots, q_n) \text{ for } f \in F_n, q, q_1, \ldots, q_n \in Q
\]
Top-Down Tree Automata

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

\[
q \rightarrow f(q_1, \ldots, q_n) \text{ for } f \in F^n, q, q_1, \ldots, q_n \in Q
\]

- We define the production relation \( q \rightarrow_A t \) as the least relation that satisfies

\[
q \rightarrow f(q_1, \ldots, q_n) \in \Delta, q_1 \rightarrow_A t_1, \ldots, q_n \rightarrow_A t_n \implies q \rightarrow_A f(t_1, \ldots, t_n)
\]
Top-Down Tree Automata

• A tuple $A = (Q, \mathcal{F}, I, \Delta)$ 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
  • $\Delta$ is a set of rules of the form

    $$q \rightarrow f(q_1, \ldots, q_n) \text{ for } f \in \mathcal{F}^n, q, q_1, \ldots, q_n \in Q$$

• We define the *production relation* $q \rightarrow_A t$ as the least relation that satisfies

    $$q \rightarrow f(q_1, \ldots, q_n) \in \Delta, q_1 \rightarrow_A t_1, \ldots, q_n \rightarrow_A t_n \implies q \rightarrow_A f(t_1, \ldots, t_n)$$

• The language of $A$ is $L(A) := \{ t \mid \exists q \in I. q \rightarrow_A t \}$
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 Qₛ)
  - Exercise
Deterministic Top-Down Tree Automata

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

\begin{align*}
&\lvert I \rvert = 1 \\
&\text{If } q \rightarrow f(q_1, \ldots, q_n) \in \Delta \land q' \rightarrow f(q'_1, \ldots, q'_n) \in \Delta \implies q_1 = q'_1 \land \ldots \land q_n = q'_n
\end{align*}

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 $f(a, a)$.
Deterministic Top-Down Tree Automata

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

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 $f(a, a)$.
Deterministic Top-Down Tree Automata

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

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 $f(a, a)$. 

69 / 161
A top-down tree-automaton $A = (Q, F, I, \Delta)$ is deterministic, iff

- $|I| = 1$
- $q \rightarrow f(q_1, \ldots, q_n) \in \Delta \land q \rightarrow f(q'_1, \ldots, q'_n) \in \Delta \implies q_1 = q'_1 \land \ldots \land q_n = q'_n$

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

- $|I| = 1$
- $q \rightarrow f(q_1, \ldots, q_n) \in \Delta \land q \rightarrow f(q_1', \ldots, q_n') \in \Delta \implies q_1 = q_1' \land \ldots \land q_n = q_n'$

Unfortunately: There are regular languages not accepted by any deterministic top-down FTA

- $L = \{f(a, b), f(b, a)\}$. Obviously regular. Even finite.
A top-down tree-automaton $\mathcal{A} = (Q, \mathcal{F}, I, \Delta)$ is deterministic, iff

- $|I| = 1$
- $q \rightarrow f(q_1, \ldots, q_n) \in \Delta \land q \rightarrow f(q'_1, \ldots, q'_n) \in \Delta \implies q_1 = q'_1 \land \ldots \land q_n = q'_n$

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 $f(a, a)$. 

Deterministic Top-Down Tree Automata
Table of Contents

1. Introduction

2. Basics

3. Alternative Representations of Regular Languages

4. Model-Checking concurrent Systems
Table of Contents

1 Introduction

2 Basics

3 Alternative Representations of Regular Languages
   Regular Tree Grammars
   Tree Regular Expressions

4 Model-Checking concurrent Systems
Regular Tree Grammars

- Extend grammars to trees
Regular Tree Grammars

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

A regular tree grammar (RTG) is a tuple $G = (S, N, F, R)$, 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 $G = (S, N, F, R)$, where
  - $S \in N$ is a start symbol
  - $N$ is a finite set of nonterminals with arity zero, and $N \cap F = \emptyset$
  - $F$ is a ranked alphabet
  - $R$ is a set of production rules of the form $n \rightarrow \beta$, where $n \in N$ and $\beta \in T(F \cup N)$
  - These are almost top-down tree automata
  - But rules are a bit more complicated
Regular Tree Grammars

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

$$\beta \Rightarrow_G \beta' \iff \exists C \ u \ n. \ \beta = C[n] \land n \rightarrow u \in R \land \beta' = C[u]$$
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 $G = (S, N, F, R)$, we define a derivation step $\beta \Rightarrow_G \beta'$ for $\beta, \beta' \in T(F \cup N)$ 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 $\beta \arrow{G} t'$, iff $t' \in T(F)$ and $\beta \Rightarrow^*_G t'$
- For $n \in N$, we define $L(G, n) := \{ t \in T(F) \mid n \rightarrow_G t \}$
• 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

$$\beta \Rightarrow G \beta' \iff \exists C u n. \beta = C[n] \land n \rightarrow u \in R \land \beta' = C[u]$$

• 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 \textit{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 \textit{productive}, iff a tree without nonterminals can be derived from it:

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

- An RTG is \textit{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:
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.

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

2. Remove unreachable non terminals
   - Again saturation: $S$ is reachable, $n$ is reachable if there is a rule $\hat{n} \rightarrow C[n]$ 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 $n \rightarrow f(n_1, \ldots, n_n)$ for $n, n_1, \ldots, n_n \in N$
- Every RTG can be transformed into an equivalent normal one
  - Iterate: Replace a rule $n \rightarrow f(s_1, \ldots, s_n)$ by $n \rightarrow f(n_1, \ldots, n_n)$
Normalized Regular Tree Grammars

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

- RTG is normalized, iff all productions have the form $n \rightarrow f(n_1, \ldots, n_n)$ for $n, n_1, \ldots, n_n \in N$
- Every RTG can be transformed into an equivalent normal one
  - Iterate: Replace a rule $n \rightarrow f(s_1, \ldots, s_n)$ by $n \rightarrow f(n_1, \ldots, n_n)$
    - where $n_i = s_i$ if $s_i \in N$
    - $n_i \in N$ fresh otherwise. In this case, add rule $n_i \rightarrow s_i$
  - After iteration, all rules have form $n \rightarrow f(n_1, \ldots, n_n)$ or $n_1 \rightarrow n_2$
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 $n \rightarrow f(n_1, \ldots, n_n)$ for $n, n_1, \ldots, n_n \in N$
- Every RTG can be transformed into an equivalent normal one
  - Iterate: Replace a rule $n \rightarrow f(s_1, \ldots, s_n)$ by $n \rightarrow f(n_1, \ldots, n_n)$
    - where $n_i = s_i$ if $s_i \in N$
    - $n_i \in N$ fresh otherwise. In this case, add rule $n_i \rightarrow s_i$
  - After iteration, all rules have form $n \rightarrow f(n_1, \ldots, n_n)$ or $n_1 \rightarrow n_2$
  - Eliminate the latter rules by replacing $s_1 \rightarrow s_2$ by rules $s_1 \rightarrow t$ for all $t \not\in N$
    with $s_2 \rightarrow^* n \rightarrow t$
    - Cf.: Elimination of epsilon rules
Normalized Regular Tree Grammars

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

- RTG is normalized, iff all productions have the form \( n \rightarrow f(n_1, \ldots, n_n) \) for \( n, n_1, \ldots, n_n \in N \)
- Every RTG can be transformed into an equivalent normal one
  - Iterate: Replace a rule \( n \rightarrow f(s_1, \ldots, s_n) \) by \( n \rightarrow f(n_1, \ldots, n_n) \)
    - where \( n_i = s_i \) if \( s_i \in N \)
    - \( n_i \in N \) fresh otherwise. In this case, add rule \( n_i \rightarrow s_i \)
  - After iteration, all rules have form \( n \rightarrow f(n_1, \ldots, n_n) \) or \( n_1 \rightarrow n_2 \)
  - Eliminate the latter rules by replacing \( s_1 \rightarrow s_2 \) by rules \( s_1 \rightarrow t \) for all \( t \notin N \) with \( s_2 \rightarrow^* n \rightarrow t \)
    - 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 \( n \rightarrow f(n_1, \ldots, n_n) \) for \( n, n_1, \ldots, n_n \in N \)
- Every RTG can be transformed into an equivalent normal one
  - Iterate: Replace a rule \( n \rightarrow f(s_1, \ldots, s_n) \) by \( n \rightarrow f(n_1, \ldots, n_n) \)
    - where \( n_i = s_i \) if \( s_i \in N \)
    - \( n_i \in N \) fresh otherwise. In this case, add rule \( n_i \rightarrow s_i \)
  - After iteration, all rules have form \( n \rightarrow f(n_1, \ldots, n_n) \) or \( n_1 \rightarrow n_2 \)
  - Eliminate the latter rules by replacing \( s_1 \rightarrow s_2 \) by rules \( s_1 \rightarrow t \) for all \( t \not\in N \) with \( s_2 \rightarrow^* n \rightarrow t \)
    - 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

1. Introduction

2. Basics

3. Alternative Representations of Regular Languages
   - Regular Tree Grammars
   - Tree Regular Expressions

4. Model-Checking concurrent Systems
Recall: Word regular expressions

- $e ::= \varepsilon \mid \emptyset \mid a$ for $a \in \Sigma \mid e \cdot e \mid e + e \mid e^*$
Recall: Word regular expressions

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

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

- $e ::= \emptyset | \varepsilon | a \text{ for } a \in \Sigma | e \cdot e | e + e | e^*$
  - 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”
- \(s(\square)^* \cdot 0\)
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
Tree regular expressions

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

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

- There may be more than one iteration, over different positions
  - Number position markers: □₁, □₂, \ldots
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: \(\square_1, \square_2, \ldots\)
  - \(cons(s(\square_1)^* \cdot 1 \cdot 0, \square_2)^* \cdot 2 \ nil\)
Tree regular expressions

- Consider the set \{0, s(0), s(s(0)), \ldots\}
  - Want to represent this as „regular expression“
- \(s(□)^\ast \cdot 0\)
  - Idea: □ indicates position for concatenation
  - \(t_1 \cdot t_2\) inserts \(t_2\) at square-position in \(t_1\)
  - \(f(\ldots)^\ast = □ + f(\ldots) \cdot f(\ldots)^\ast\) iterates over position □
- There may be more than one iteration, over different positions
  - Number position markers: □_1, □_2, \ldots
  - \(\text{cons}(s(□_1)^\ast_1 \cdot 1 0, □_2)^\ast_2 \cdot 2 \text{ nil}\)
- Note: TATA notation: \(s(□_1)^\ast_1 \cdot □_1 . □_1 \text{ nil}\)
Substitution and Concatenation

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

- Let $\mathcal{K} := \square_1/0, \square_2/0, \ldots$. 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 $i \neq j \rightarrow a_i \neq a_j$:

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

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

  $f(s_1, \ldots, s_m)\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\}$

  $= \{f(t_1, \ldots, t_m) \mid t_i \in s_i\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\}\}$

- 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 $L_1 \cdot L_2 := L_1\{\square_i \leftarrow L_2\}$
Substitution and Concatenation

- Let $\mathcal{K} := \emptyset_1/0, \emptyset_2/0, \ldots$. 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
  
  f(s_1, \ldots, s_m)\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\} \\
  \quad = \{f(t_1, \ldots, t_m) \mid t_i \in s_i\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\}\}
  \]

- 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 $\mathcal{K} := □_1/0, □_2/0, \ldots$ 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
  
  f(s_1, \ldots, s_m)\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\}
  
  = \{f(t_1, \ldots, t_m) \mid t_i \in s_i\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\}\}
  
- 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
  
  \[
  L_1 \cdot_i L_2 := L_1\{□_i \leftarrow L_2\}\]
Iteration

- Iteration $L^{n,i}$

\[ L^{0,i} := □_i \]

\[ L^{n+1,i} = L^{n,i} \cup L \cdot_i L^{n,i} \]
Iteration

- Iteration $L^{n,i}$

  $L^{0,i} := \square_i$

  $L^{n+1,i} = L^{n,i} \cup L \cdot L^{n,i}$

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

- Iteration $L^{n,i}$

$\begin{align*}
L^{0,i} &:= \square_i \\
L^{n+1,i} &= L^{n,i} \cup L \cdot i \cdot L^{n,i}
\end{align*}$

- 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 $L^{n,i}$

\[
L^{0,i} := \square_i, \quad L^{n+1,i} = L^{n,i} \cup L \cdot L^{n,i}
\]

- 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 $L^{n,i}$

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

- 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: $f(f(\square, f(\square, \square)), \square) \in \{f(\square, \square)\}^3$

- 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.
  - \( G = (S, N, F, R) \) with \( L = L(G) \) and \( G_i = (S_i, N_i, F, R_i) \) with \( L_i = L(G_i) \).
- Then let \( G' = (S, N \cup N_1 \cup \ldots \cup N_n, F, R' \cup R_1 \cup \ldots \cup R_n) \) 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 \( L_i \).
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 $\Box_i$ 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: 
    - $\forall A \in N. \ A \rightarrow_{G'} s' \implies \exists s. \ A \rightarrow_{G} s \land s' \in s\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\}$
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 $\square_i$ 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:
      - $\forall A \in N. \ A \rightarrow_{G'} s' \implies \exists s. \ A \rightarrow_G s \wedge s' \in s\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\}$
    - By induction on derivation length
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:
      \[ \forall A \in N. \ A \rightarrow_{G'} s' \implies \exists s. \ A \rightarrow_G s \wedge s' \in s\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\} \]
    - 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 \( G = (S, N, \mathcal{F}, R) \)

\[ L^* \subseteq L(G') \]: Obvious by construction

\[ L(G') \subseteq L^* \]: Re-ordering derivation. Formally: Induction on derivation length.
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 \( G' = (S', N \cup \{S'\}, \mathcal{F} \cup K, R') \), such that

\( L^* \subseteq L(G') \): Obvious by construction

\( L(G') \subseteq L^* \): Re-ordering derivation. Formally: Induction on derivation length.
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 $G' = (S', N \cup \{S'\}, \mathcal{F} \cup \mathcal{K}, R')$, such that
    • $R'$ contains the rules from $R$, with $\Box$ replaced by $S'$
  • $L^* \subseteq L(G')$: Obvious by construction
  • $L(G') \subseteq L^*$: Re-ordering derivation. Formally: Induction on derivation length.
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 \( G = (S, N, F, R) \)
  - Construct \( G' = (S', N \cup \{S'\}, F \cup K, R') \), such that
    - \( R' \) contains the rules from \( R \), with \( □ \) replaced by \( S' \)
    - \( S' \to □ \in R' \) 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 \( G' = (S', N \cup \{S'\}, \mathcal{F} \cup \mathcal{K}, R') \), 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 \( G' = (S', N \cup \{S'\}, \mathcal{F} \cup \mathcal{K}, R') \), such that
    - \( R' \) contains the rules from \( R \), with \( \square \) replaced by \( S' \)
    - \( S' \rightarrow \square \in R' \) and \( S' \rightarrow S \in R' \)
  - \( L^* \subseteq L(G') \): Obvious by construction
  - \( L(G') \subseteq L^* \): Re-ordering derivation. Formally: Induction on derivation length.
Tree Regular Expressions

- Syntax

\[ e ::= \emptyset \mid f(e, \ldots, e) \text{ for } f \in \mathcal{F}_n \mid e + e \mid e \cdot i \mid e^* \]

\( n \text{ times} \)
Tree Regular Expressions

- **Syntax**

\[ e ::= \emptyset \mid f(e, \ldots, e) \text{ for } f \in \mathcal{F}_n \mid e + e \mid e \cdot_i e \mid e^* \]

- **Semantics**

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

A tree language $L$ is regular if and only if there is a regular expression $e$ with $L = \llbracket e \rrbracket$

- Proof ($\leftrightarrow$): Straightforward, by induction on $e$, using preservation of regularity by union, concatenation, and closure
Kleene Theorem for Tree Languages

A tree language $L$ is regular if and only if there is a regular expression $e$ with $L = \llbracket e \rrbracket$

- Proof ($\iff$): Straightforward, by induction on $e$, using preservation of regularity by union, concatenation, and closure
- Proof ($\implies$): Construct reg-exp inductively over increasing number of states
Kleene Theorem for Tree Languages (Proof)

• Let $A = (Q, \mathcal{F}, Q_F, \Delta)$ be bottom-up automaton.
Kleene Theorem for Tree Languages (Proof)

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

- Let $A = (Q, \mathcal{F}, Q_F, \Delta)$ 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 $A = (Q, \mathcal{F}, Q_F, \Delta)$ 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 $K \subseteq \{q_1, \ldots, q_k\}$
- $L(A) = \bigcup_{i \mid q_i \in Q_F} T(i, n, \emptyset)$
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 $A = (Q, F, Q_F, \Delta)$ be bottom-up automaton.
  - Let $Q = \{q_1, \ldots, q_n\}$
- Define $T(i, j, K)$ for $K \subseteq Q$ as those trees over $T(F \cup K)$ 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 $K \subseteq \{q_1, \ldots, q_k\}$
- $L(A) = \bigcup_{i \mid q_i \in Q_F} T(i, n, \emptyset)$
- $T(i, 0, K)$ is finite
  - Runs accepting $t \in T(i, 0, K)$ contain no internal states
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 $Q = \{q_1, \ldots, q_n\}$
- 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
Kleene Theorem for Tree Languages (Proof)

- Let $A = (Q, F, Q_F, \Delta)$ be bottom-up automaton.
  - Let $Q = \{q_1, \ldots, q_n\}$
- Define $T(i, j, K)$ for $K \subseteq Q$ as those trees over $T(F \cup K)$ 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 $K \subseteq \{q_1, \ldots, q_k\}$
- $L(A) = \bigcup_{i|q_i \in Q_F} T(i, n, \emptyset)$
- $T(i, 0, K)$ is finite
  - Runs accepting $t \in T(i, 0, K)$ contain no internal states
  - I.e., $t = a()$ or $t = f(a_1, \ldots a_m)$, for $a, a_1, \ldots a_m \in F \cup K$
  - Thus, representable by regular expression
- For $j > 0$:

$$T(i, j, K) = T(i, j - 1, K \cup \{q_j\}) \cdot_{q_j} T(j, j - 1, K \cup \{q_j\})^* \cdot_{q_j} T(j, j - 1, K)$$

Initial segment

Runs between $q_j$s

Final segment
Kleene Theorem for Tree Languages (Proof)

- Let $A = (Q, F, Q_F, \Delta)$ be bottom-up automaton.
  - Let $Q = \{q_1, \ldots, q_n\}$
- Define $T(i, j, K)$ for $K \subseteq Q$ as those trees over $T(F \cup K)$ 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 $K \subseteq \{q_1, \ldots, q_k\}$
- $L(A) = \bigcup_{i | q_i \in Q_F} T(i, n, \emptyset)$
- $T(i, 0, K)$ is finite
  - Runs accepting $t \in T(i, 0, K)$ contain no internal states
  - I.e., $t = a()$ or $t = f(a_1, \ldots a_m)$, for $a, a_1, \ldots a_m \in F \cup K$
  - Thus, representable by regular expression
- For $j > 0$:

$$T(i, j, K) = T(i, j - 1, K \cup \{q_j\}) \cdot_{q_j} T(j, j - 1, K \cup \{q_j\})^{*,q_j} \cdot_{q_j} T(j, j - 1, K)$$

  - Initial segment
  - Runs between $q_j$'s
  - Final segment

- Regular expression for $L(A)$ can be constructed
Last Lecture

- Tree regular expressions
- Kleene theorem
  - Tree regular expressions can express exactly the tree regular languages
# Table of Contents

1. Introduction
2. Basics
3. Alternative Representations of Regular Languages
4. Model-Checking concurrent Systems
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
</tr>
<tr>
<td>2</td>
<td>Basics</td>
</tr>
<tr>
<td>3</td>
<td>Alternative Representations of Regular Languages</td>
</tr>
<tr>
<td>4</td>
<td>Model-Checking concurrent Systems</td>
</tr>
<tr>
<td></td>
<td>Motivation</td>
</tr>
<tr>
<td></td>
<td>Pushdown Systems</td>
</tr>
<tr>
<td></td>
<td>Dynamic Pushdown Networks</td>
</tr>
<tr>
<td></td>
<td>Acquisition Histories</td>
</tr>
<tr>
<td></td>
<td>Acquisition Histories for DPN</td>
</tr>
</tbody>
</table>
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

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

\[1 \xrightarrow{\tau} 12\]  \[1 \xrightarrow{\tau} \varepsilon\]

\[2 \xleftrightarrow{\tau} 3\]

\[3 \xleftrightarrow{\tau} \varepsilon\]
Table of Contents

1 Introduction

2 Basics

3 Alternative Representations of Regular Languages

4 Model-Checking concurrent Systems
   Motivation
   Pushdown Systems
   Dynamic Pushdown Networks
   Acquisition Histories
   Acquisition Histories for DPN
Push-Down Systems (PDS)

- In order to model (finitely many) return values, we add state
- A push-down system (PDS) $M$ is a tuple $(P, \Gamma, \text{Act}, p_0, \gamma_0, \Delta)$ where
  - $P$ is a finite set of states
  - $\Gamma$ is a finite stack alphabet
  - $\text{Act}$ is a finite set of actions
  - $p_0 \gamma_0 \in P\Gamma$ is the initial configuration
  - $\Delta$ is a finite set of rules, of the form
    $$p \gamma \xrightarrow{a} p' w$$
    where $p, p' \in P$, $a \in \text{Act}$, $\gamma \in \Gamma$, and $w \in \Gamma^*$
• Configurations have the form $p w \in P \Gamma^*$
• The step-relation $\rightarrow \subseteq P \Gamma^* \times \text{Act} \times P \Gamma^*$ is defined by

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

• $\rightarrow^* \subseteq P \Gamma^* \times \text{Act}^* \times P \Gamma^*$ is its extension to sequences of steps
  • $p w \xrightarrow{l}^* p' w'$ iff $l = a_1 \ldots a_n$ and $p w \xleftarrow{a_1} \xrightarrow{a_n} p' w'$
Normalized PDS

- Simplifying assumptions
  - There are only three types of rules
    
    \[ p \gamma \xrightarrow{a} p' \gamma' \quad \text{for } p, p' \in P \text{ and } \gamma, \gamma' \in \Gamma \quad \text{(base)} \]
    
    \[ p \gamma \xrightarrow{a} p' \gamma_1 \gamma_2 \quad \text{for } p, p' \in P \text{ and } \gamma, \gamma_1, \gamma_2 \in \Gamma \quad \text{(call)} \]
    
    \[ p \gamma \xrightarrow{a} p' \quad \text{for } p, p' \in P \text{ and } \gamma \in \Gamma \quad \text{(return)} \]

  - Does not reduce expressiveness. Emulate rule \( p \gamma \xrightarrow{\gamma_1 \ldots \gamma_n} \) by sequence of call rules.

  - The empty stack must not be reachable
    - Does not reduce expressiveness
    - Introduce fresh \( \bot \) stack symbol, a rule \( p_0 \bot \xrightarrow{\tau} p_0 \gamma_0 \bot \), and set initial state to \( p_0 \bot \)
    - \( \tau \) 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

\[
\begin{align*}
XR & ::= \langle Base \rangle(XR) \mid \langle Call \rangle^R(XR, XR) \mid \langle Return \rangle \\
XN & ::= \langle Base \rangle(XN) \mid \langle Call \rangle^N(XN) \mid \langle Call \rangle^R(XR, XN) \mid \langle P \times \Gamma \rangle
\end{align*}
\]

- Where \textit{Base, Call, Return} are rules of respective type
- Intuition: \(XR\) – Returning execution trees, \(XN\) – non-returning execution trees
Example

\[ p_1 \xleftarrow{\tau} p_{12} \quad p_1 \xrightarrow{\tau} p \]
\[ p_2 \xleftrightarrow{x=y} p_3 \]
\[ p_3 \xrightarrow{\tau} p \]

- Example execution tree
  - \( \langle p_1 \xrightarrow{\tau} p_{12} \rangle^N (\langle p_1 \xrightarrow{\tau} p_{12} \rangle^R (\langle p_1 \xrightarrow{\tau} p \rangle, \langle p_2 \xleftrightarrow{x=y} p_3 \rangle (\langle p_3 \rangle)))) \)
Execution Trees of PDS

- Execution trees of PDS $M = (P, \Gamma, \text{Act}, p_0, \gamma_0, \Delta)$ described by tree automata $A_M = (Q, F, I, \Delta_{A_M})$
- 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

\[
\begin{align*}
p_\gamma \rightarrow \langle p_\gamma \xrightarrow{a} p' \gamma' \rangle (p' \gamma') & \quad \text{if } p_\gamma \xrightarrow{a} p' \gamma' \in \Delta \\
p_\gamma \rightarrow \langle p_\gamma \leftarrow p' \gamma_1 \gamma_2 \rangle ^N (p' \gamma_1) & \quad \text{if } p_\gamma \leftarrow p' \gamma_1 \gamma_2 \in \Delta \\
p_\gamma \rightarrow \langle p_\gamma \leftarrow p' \gamma_1 \gamma_2 \rangle ^R (p' \gamma_1 | p'', p'' \gamma_2) & \quad \text{if } p'' \in P \text{ and } p_\gamma \leftarrow p' \gamma_1 \gamma_2 \in \Delta \\
p_\gamma \rightarrow \langle p_\gamma \rangle & \\
p_\gamma | p'' \rightarrow \langle p_\gamma \leftarrow p' \gamma' \rangle (p' \gamma' | p'') & \quad \text{if } p_\gamma \leftarrow p' \gamma' \in \Delta \\
p_\gamma | p'' \rightarrow \langle p_\gamma \leftarrow p' \gamma_1 \gamma_2 \rangle ^R (p' \gamma_1 | p''', p''' \gamma_2 | p'') & \quad \text{if } p''' \in P \text{ and } p_\gamma \leftarrow p' \gamma_1 \gamma_2 \in \Delta \\
p_\gamma | p'' \rightarrow \langle p_\gamma \xrightarrow{\tau} p'' \rangle & \quad \text{if } p_\gamma \xrightarrow{\tau} p'' \in \Delta
\end{align*}
\]
Execution Trees – Intuition of rules

- \( p_\gamma \rightarrow \langle p_\gamma \xrightarrow{a} p'_\gamma' \rangle (p'_\gamma') \) (Base)
  - Make a base step, then continue execution from \( p'_\gamma' \)

- \( p_\gamma \rightarrow \langle p_\gamma \xrightarrow{a} p'\gamma_1\gamma_2 \rangle^N (p'_\gamma_1) \) (Call, no-return)
  - Continue execution from \( p'_\gamma_1 \).
  - As call does not return, \( \gamma_2 \) is never looked at again, and remaining execution does not depend on it

- \( p_\gamma \rightarrow \langle p_\gamma \xrightarrow{a} p'\gamma_1\gamma_2 \rangle^R (p'_\gamma_1 | p'', p''\gamma_2) \) (Call, return)
  - Execute procedure, it returns with state \( p'' \). Then continue execution from \( p''\gamma_2 \).

- \( p_\gamma \rightarrow \langle p_\gamma \rangle \) (Finish)
  - Non-deterministically decide that execution ends here

- \( p_\gamma | p'' \rightarrow \langle p_\gamma \xrightarrow{a} p'_\gamma' \rangle (p'_\gamma' | p'') \) (Base)
  - Base step, then continue execution

- \( p_\gamma | p'' \rightarrow \langle p_\gamma \xrightarrow{a} p'\gamma_1\gamma_2 \rangle^R (p'_\gamma_1 | p'''', p''''\gamma_2 | p'') \) (Call, return)
  - Return from called procedure in state \( p''''' \), then continue execution

- \( p_\gamma | p'' \rightarrow \langle p_\gamma \xrightarrow{\tau} p'' \rangle \) (Return)
  - Return rule returns to specified state \( p'' \)
Reached Configuration

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

\[
c(\langle p_{\gamma} \xrightarrow{a} p'_{\gamma'} \rangle(t)) = c(t)
\]

\[
c(\langle p_{\gamma} \xrightarrow{\tau} p'_{\gamma_1 \gamma_2} \rangle^R(t_1, t_2)) = c(t_2)
\]

\[
c(\langle p_{\gamma} \xrightarrow{\tau} p'_{\gamma_1 \gamma_2} \rangle^N(t)) = c(t)\gamma_2
\]

\[
c(\langle p_{\gamma} \rangle) = p_{\gamma}
\]

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

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

- 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:
  \( \exists l. p_{\gamma} \xrightarrow{l} p'w \land w \neq \varepsilon \) iff \( \exists t. p_{\gamma} \rightarrow t \land c(t) = p'w \)
- Main lemmas that are required
  - An execution can be repeated when we append some symbols to the stack:
    lemma stack-append: \( pw \xrightarrow{l} p'w \land w \neq \varepsilon \) \( \implies \) \( pwv \xrightarrow{l} p'w'v \)
  - 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:
    \( p_{\gamma}w \xrightarrow{l} p'w' \) \( \implies \)
    \( \exists p'' l_1 l_2. p_{\gamma} \xrightarrow{l_1} p'' \land p''w \xrightarrow{l_2} p'w' \land l = l_1l_2 \)  \( \text{(ret)} \)
    \( \lor \exists w''. w' = w''w \land w'' \neq \varepsilon \land p_{\gamma} \xrightarrow{l} p'w'' \)  \( \text{(no-ret)} \)
  - Corollary: On a returning execution, we can find the point where the topmost stack symbol is popped
    lemma find-return: \( p_{\gamma}w \xrightarrow{l} p' \) \( \implies \exists l_1 l_2 p''. p_{\gamma} \xrightarrow{l_1} p'' \land p''w \xrightarrow{l_2} p' \)
Proofs:

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

1 Introduction

2 Basics

3 Alternative Representations of Regular Languages

4 Model-Checking concurrent Systems
   - Motivation
   - Pushdown Systems
   - Dynamic Pushdown Networks
   - Acquisition Histories
   - Acquisition Histories for DPN
Thread Creation

- Concurrent programs may create threads
- These run in parallel
Example

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

main () {
    p();
    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 $M = (P, \Gamma, \text{Act}, p_0, \gamma_0, \Delta)$ consists of
Dynamic Pushdown Networks

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

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

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

  $p \gamma \xrightarrow{a} p' \gamma'$ \hspace{1cm} for $p, p' \in P$ and $\gamma, \gamma' \in \Gamma$ \hspace{1cm} (base)

  $p \gamma \xrightarrow{a} p' \gamma_1 \gamma_2$ \hspace{1cm} for $p, p' \in P$ and $\gamma, \gamma_1, \gamma_2 \in \Gamma$ \hspace{1cm} (call)

  $p \gamma \xrightarrow{a} p'$ \hspace{1cm} for $p, p' \in P$ and $\gamma \in \Gamma$ \hspace{1cm} (return)

  $p \gamma \xrightarrow{a} p_1 \gamma_1 \triangleright p_2 \gamma_2$ \hspace{1cm} for $p, p_1, p_2 \in P$ and $\gamma, \gamma_1, \gamma_2 \in \Gamma$ \hspace{1cm} (spawn)

Assumption: Empty stack not reachable in any spawned thread
Dynamic Pushdown Networks

- Pushdown systems
- Spawn-rules may have side-effect of creating a new PDS
- A DPN $M = (P, \Gamma, \text{Act}, p_0, \gamma_0, \Delta)$ consists of
  - A finite set of states $P$
  - A finite set of stack symbols $\Gamma$
  - A finite set of actions $\text{Act}$
  - An initial configuration $p_0\gamma_0 \in P\Gamma$
  - Rules $\Delta$ of the form
    \[
    p\gamma \xrightarrow{a} p'\gamma' \quad \text{for } p, p' \in P \text{ and } \gamma, \gamma' \in \Gamma \quad \text{(base)}
    \]
    \[
    p\gamma \xrightarrow{a} p'\gamma_1\gamma_2 \quad \text{for } p, p' \in P \text{ and } \gamma, \gamma_1, \gamma_2 \in \Gamma \quad \text{(call)}
    \]
    \[
    p\gamma \xrightarrow{a} p' \quad \text{for } p, p' \in P \text{ and } \gamma \in \Gamma \quad \text{(return)}
    \]
    \[
    p\gamma \xrightarrow{a} p_1\gamma_1 \triangleright p_2\gamma_2 \quad \text{for } p, p_1, p_2 \in P \text{ and } \gamma, \gamma_1, \gamma_2 \in \Gamma \quad \text{(spawn)}
    \]

- Assumption: Empty stack not reachable in any spawned thread
Configurations

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

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

- Convention: We identify $c$ with the singleton list $\text{Cons}(c, \text{Nil})$, and use $l_1 l_2$ for the concatenation of $l_1$ and $l_2$.

- We may use $[c_1, ..., c_n]$ for the list $\text{Cons}(c_1, \text{Cons}(..., \text{Cons}(c_n, \text{Nil})...))$ for clarification of notation.
Configurations

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

- Configurations are trees over the alphabet $\langle pw \rangle | Cons/2 | Nil/0$
  - 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 $l$
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
  
  \[
  \text{conf} ::= \langle pw \rangle(\text{conflist}) \quad \text{conflist} ::= \text{Nil} \mid \text{Cons}(\text{conf}, \text{conflist})
  \]
- 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 $\text{Cons}(c, \text{Nil})$, 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
  $conf ::= \langle pw \rangle (conflist) \mid conflist ::= Nil \mid Cons(conf, conflist)$
- 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

\[ C[\langle p\gamma w\rangle(l)] \xrightarrow{a} C[\langle p' w' w\rangle(l)] \]

if \( p\gamma \xrightarrow{a} p' w' \in \Delta \) \hspace{10cm} \text{(no-spawn)}

\[ C[\langle p\gamma w\rangle(l)] \xrightarrow{a} C[\langle p_{1 \gamma 1} w\rangle(l\langle p_{2 \gamma 2}\rangle(\text{Nil}))] \]

if \( p\gamma \xrightarrow{a} p_{1 \gamma 1} \triangleright p_{2 \gamma 2} \in \Delta \) \hspace{10cm} \text{(spawn)}
Semantics

• A step modifies a thread’s state according to a rule

\[
C[\langle p \gamma w \rangle (l)] \xrightarrow{a} C[\langle p' w' w \rangle (l)]
\]

\[
\text{if } p \gamma \xrightarrow{a} p' w' \in \Delta \quad \text{(no-spawn)}
\]

\[
C[\langle p \gamma w \rangle (l)] \xrightarrow{a} C[\langle p_1 \gamma_1 w \rangle (l\langle p_2 \gamma_2 \rangle (Nil))]
\]

\[
\text{if } p \gamma \xrightarrow{a} p_1 \gamma_1 \triangleright p_2 \gamma_2 \in \Delta \quad \text{(spawn)}
\]

• For any context \( C \) with exactly one occurrence of \( x_1 \), such that \( C[\langle p \gamma w \rangle (l)] \in conf \) 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

\[ C[\langle p \gamma w \rangle(l)] \xrightarrow{a} C[\langle p' w' \rangle(l)] \]

\[ \text{if } p \gamma \xrightarrow{a} p' w' \in \Delta \]

\[ C[\langle p \gamma w \rangle(l)] \xrightarrow{a} C[\langle p_1 \gamma_1 w \rangle(l\langle p_2 \gamma_2 \rangle(\text{Nil}))] \]

\[ \text{if } p \gamma \xrightarrow{a} p_1 \gamma_1 \triangleright p_2 \gamma_2 \in \Delta \]

• For any context \( C \) with exactly one occurrence of \( x_1 \), such that
\( C[\langle p \gamma w \rangle(l)] \in \text{conf} \) is a configuration

• Having exactly one occurrence of \( x_1 \) ensures that exactly one thread makes a step

• Intuition:

\( \text{(no-spawn)} \)

\( \text{(spawn)} \)
Semantics

• A step modifies a thread’s state according to a rule

\[
C[\langle p\gamma w\rangle(l)] \xrightarrow{a} C[\langle p'w'w\rangle(l)]
\]

if \( p\gamma \xleftarrow{a} p'w' \in \Delta \)

\[
C[\langle p\gamma w\rangle(l)] \xrightarrow{a} C[\langle p_1\gamma_1 w\rangle(l\langle p_2\gamma_2\rangle(\text{Nil}))]
\]

if \( p\gamma \xrightarrow{a} p_1\gamma_1 \triangleright p_2\gamma_2 \in \Delta \)

• For any context \( C \) with exactly one occurrence of \( x_1 \), such that \( C[\langle p\gamma w\rangle(l)] \in \text{conf} \) 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

\[ C[\langle p \gamma w \rangle(l)] \xrightarrow{a} C[\langle p' w' w \rangle(l)] \]

\[ \text{if } p \gamma \xrightarrow{a} p' w' \in \Delta \]

\[ C[\langle p \gamma w \rangle(l)] \xrightarrow{a} C[\langle p_1 \gamma_1 w \rangle(l \langle p_2 \gamma_2 \rangle(\text{Nil}))] \]

\[ \text{if } p \gamma \xrightarrow{a} p_1 \gamma_1 \triangleright p_2 \gamma_2 \in \Delta \]

- For any context \( C \) with exactly one occurrence of \( x_1 \), such that \( C[\langle p \gamma w \rangle(l)] \in \text{conf} \) 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 \( \langle p \gamma \xrightarrow{a} p_1 \gamma_1 \triangleright p_2 \gamma_2 \rangle(t_1, t_2) \) describes execution of spawn-step
Execution Trees

- Binary node $\langle p_\gamma \xrightarrow{a} p_1 \gamma_1 \triangleright p_2 \gamma_2 \rangle(t_1, t_2)$ describes execution of spawn-step
  - $t_1$ describes remaining execution of spawning thread
Execution Trees

- Binary node $\langle p_\gamma \xleftrightarrow{a} p_1 \gamma_1 \triangleright p_2 \gamma_2 \rangle(t_1, t_2)$ describes execution of spawn-step
  - $t_1$ describes remaining execution of spawning thread
  - $t_2$ describes execution of spawned thread
Execution Trees

- Binary node $\langle p_0 \leftarrow a p_1 \mathrel{\triangleright} p_2 \rangle(t_1, t_2)$ 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 $c : XN \rightarrow conf$
Function $c : XN \rightarrow conf$

- $c(\langle Spawn \rangle(t_1, t_2)) = [c(t_2)]c(t_1)$
Configuration of Execution Tree

- Function $c : \text{XN} \rightarrow \text{conf}$
  - $c(\langle \text{Spawn} \rangle(t_1, t_2)) = [c(t_2)]c(t_1)$
    - Prepend configuration reached by spawned thread
Configuration of Execution Tree

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

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

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

Define \( c : XN \rightarrow \text{conf} \) and \( s : XR \rightarrow \text{conflist} \)

\[
c(\langle p_\gamma \xrightarrow{a} p'_{\gamma'} \rangle(t)) = c(t)
\]

\[
c(\langle p_\gamma \xrightarrow{\tau} p'_{\gamma_1 \gamma_2} \rangle^R(t_1, t_2)) = s(t_1)c(t_2)
\]

\[
c(\langle p_\gamma \xrightarrow{\tau} p'_{\gamma_1 \gamma_2} \rangle^N(t)) = c(t)\gamma_2
\]

\[
c(\langle p_\gamma \xrightarrow{a} p_1 \gamma_1 \triangleright p_2 \gamma_2 \rangle(t_1, t_2)) = [c(t_2)]c(t_1)
\]

\[
c(\langle p_\gamma \rangle) = \langle p_\gamma \rangle
\]

\[
s(\langle p_\gamma \xrightarrow{a} p'_{\gamma'} \rangle(t)) = s(t)
\]

\[
s(\langle p_\gamma \xrightarrow{\tau} p'_{\gamma_1 \gamma_2} \rangle^R(t_1, t_2)) = s(t_1)s(t_2)
\]

\[
s(\langle p_\gamma \xrightarrow{a} p_1 \gamma_1 \triangleright p_2 \gamma_2 \rangle(t_1, t_2)) = [c(t_2)]s(t_1)
\]

\[
s(\langle p_\gamma \xrightarrow{a} p' \rangle) = \text{Nil}
\]

where \( \langle pw_\gamma \rangle(l) = \langle pw_\gamma \rangle(l) \)
Execution trees of DPN

- Execution trees are regular set
Execution trees are regular set
Same idea as for PDS. New rules for $A_M$:

\[ p_{\gamma} \rightarrow \langle p_{\gamma} \xrightarrow{a} p_{1\gamma_1} \triangleright p_{2\gamma_2} \rangle(p_{1\gamma_1}, p_{2\gamma_2}) \]  
if $p_{\gamma} \xrightarrow{a} p_{1\gamma_1} \triangleright p_{2\gamma_2} \in \Delta$

\[ p_{\gamma} | p'' \rightarrow \langle p_{\gamma} \xrightarrow{a} p_{1\gamma_1} \triangleright p_{2\gamma_2} \rangle(p_{1\gamma_1} | p'', p_{2\gamma_2}) \]  
if $p_{\gamma} \xrightarrow{a} p_{1\gamma_1} \triangleright p_{2\gamma_2} \in \Delta$
Execution trees of DPN

- Execution trees are regular set
- Same idea as for PDS. New rules for $A_M$:

  \[ p\gamma \rightarrow \langle p\gamma \xrightarrow{a} p_1\gamma_1 \triangleright p_2\gamma_2 \rangle (p_1\gamma_1, p_2\gamma_2) \quad \text{if} \quad p\gamma \xrightarrow{a} p_1\gamma_1 \triangleright p_2\gamma_2 \in \Delta \]

  \[ p\gamma \mid p'' \rightarrow \langle p\gamma \xrightarrow{a} p_1\gamma_1 \triangleright p_2\gamma_2 \rangle (p_1\gamma_1 \mid p'', p_2\gamma_2) \quad \text{if} \quad p\gamma \xrightarrow{a} p_1\gamma_1 \triangleright p_2\gamma_2 \in \Delta \]

- Complete rules on next slide
Rules for execution trees

\[ p_\gamma \rightarrow \langle p_\gamma \overset{a}{\leftarrow} p_\gamma' \rangle (p_\gamma') \]
if \( p_\gamma \overset{a}{\leftarrow} p_\gamma' \in \Delta \)

\[ p_\gamma \rightarrow \langle p_\gamma \overset{a}{\leftarrow} p_\gamma' \rangle \overset{N}{\rightarrow} (p_\gamma') \]
if \( p_\gamma \overset{a}{\leftarrow} p_\gamma' \in \Delta \)

\[ p_\gamma \rightarrow \langle p_\gamma \overset{a}{\leftarrow} p_\gamma_1 \gamma_2 \rangle \overset{R}{\rightarrow} (p_\gamma_1 | p_\gamma' \gamma_2) \]
if \( p_\gamma \overset{a}{\leftarrow} p_\gamma_1 \gamma_2 \in \Delta \)

\[ p_\gamma \rightarrow \langle p_\gamma \overset{a}{\leftarrow} p_\gamma \rangle \]
if \( p_\gamma \overset{a}{\leftarrow} p_\gamma \in \Delta \)

\[ p_\gamma | p'' \rightarrow \langle p_\gamma \overset{a}{\leftarrow} p_\gamma' \rangle (p_\gamma' | p'') \]
if \( p_\gamma \overset{a}{\leftarrow} p_\gamma' \in \Delta \)

\[ p_\gamma | p'' \rightarrow \langle p_\gamma \overset{a}{\leftarrow} p_\gamma_1 \gamma_2 \rangle \overset{R}{\rightarrow} (p_\gamma_1 | p'' \gamma_2, p'' \gamma_2 | p'') \]
if \( p'' \in P \) and \( p_\gamma \overset{a}{\leftarrow} p_\gamma_1 \gamma_2 \in \Delta \)

\[ p_\gamma | p'' \rightarrow \langle p_\gamma \overset{a}{\leftarrow} p_\gamma \rangle \]
if \( p'' \in P \) and \( p_\gamma \overset{a}{\leftarrow} p_\gamma \in \Delta \)

\[ p_\gamma | p'' \rightarrow \langle p_\gamma \overset{a}{\leftarrow} p_\gamma_1 \gamma_2 \rangle \overset{R}{\rightarrow} (p_\gamma_1 | p'' \gamma_2, p'' \gamma_2 | p'') \]
if \( p'' \in P \) and \( p_\gamma \overset{a}{\leftarrow} p_\gamma_1 \gamma_2 \in \Delta \)

\[ p_\gamma | p'' \rightarrow \langle p_\gamma \overset{a}{\leftarrow} p_\gamma \rangle \]
if \( p'' \in P \) and \( p_\gamma \overset{a}{\leftarrow} p_\gamma \in \Delta \)

\[ p_\gamma | p'' \rightarrow \langle p_\gamma \overset{a}{\leftarrow} p_\gamma \rangle \]
if \( p'' \in P \) and \( p_\gamma \overset{a}{\leftarrow} p_\gamma \in \Delta \)

\[ p_\gamma | p'' \rightarrow \langle p_\gamma \overset{a}{\leftarrow} p_\gamma \rangle \]
if \( p'' \in P \) and \( p_\gamma \overset{a}{\leftarrow} p_\gamma \in \Delta \)

\[ p_\gamma | p'' \rightarrow \langle p_\gamma \overset{a}{\leftarrow} p_\gamma \rangle \]
if \( p'' \in P \) and \( p_\gamma \overset{a}{\leftarrow} p_\gamma \in \Delta \)
Relating Execution Trees and DPN Semantics

**Theorem**

Let $M$ be a DPN. Then $\exists l. \, p_0 \gamma_0 \xrightarrow{l}^* c'$ iff $\exists t. \, t \in L(A_M) \land c(t) = c'$

- Note: Relating the action sequences is more difficult
Relating Execution Trees and DPN Semantics

**Theorem**

Let $M$ be a DPN. Then $\exists l. \; p_0 \gamma_0 \xrightarrow{l}^* c' \iff \exists t. \; t \in L(A_M) \land c(t) = c'$

- Note: Relating the action sequences is more difficult
  - They are *interleavings* of the thread’s action sequences
Theorem

Let $M$ be a DPN. Then $\exists l. \ p_0 \gamma_0 \xrightarrow{\cdot} \ c'$ iff $\exists t. \ t \in L(A_M) \land c(t) = c'$

- 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 $s_2$

  $$s_1 \otimes \varepsilon = \{s_1\} \quad \varepsilon \otimes s_2 = \{s_2\}$$

  $$a_1 s_1 \otimes a_2 s_2 = a_1 (s_1 \otimes a_2 s_2) \cup a_2 (a_1 s_1 \otimes s_2)$$

- 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

\[\langle pw \rangle (c) s \rightarrow^{\ast} \langle p'w' \rangle (l') \iff \exists c' l'' s_1 s_2. l' = c' l'' \land s \in s_1 \otimes s_2 \land \langle pw \rangle (\varepsilon) s_1 \rightarrow^{\ast} \langle p'w' \rangle (l'') \land c s_2 \rightarrow^{\ast} c'\]

- Proof, by induction on number of steps:
  \[\langle p \gamma \rangle (\varepsilon) \rightarrow^{\ast} \langle p' \rangle (c') \iff \exists t. p \gamma | p' \rightarrow t \land s(t) = c' \langle p \gamma \rangle (\varepsilon) \rightarrow^{\ast} \langle p'w' \rangle (c') \land w' \neq \varepsilon \iff \exists t. p \gamma \rightarrow t \land c(t) = \langle p'w' \rangle (c')\]

- Need to prove both propositions simultaneously
  - But may separate

\[\Rightarrow\text{ and } \iff\text{ directions}\]
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:

  \[
  \langle pw \rangle([c]) \xrightarrow{s}^* \langle p' w' \rangle(l') \iff \\
  \exists c' \ l'' \ s_1 \ s_2. \ l' = c' l'' \land s \in s_1 \otimes s_2 \land \langle pw \rangle(\varepsilon) \xrightarrow{s_1}^* \langle p' w' \rangle(l'') \land c \xrightarrow{s_2}^* c'
  \]
Proof 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:

\[
\langle pw \rangle([c]) \xrightarrow{s}^* \langle p' w' \rangle(l') \iff \\
\exists c' I'' s_1 s_2. I' = c'I'' \land s \in s_1 \otimes s_2 \land \langle pw \rangle(\varepsilon) \xrightarrow{s_1}^* \langle p' w' \rangle(l'') \land c \xrightarrow{s_2}^* c'
\]

- Proof, by induction on number of steps:

\[
\langle p\gamma \rangle(\varepsilon) \rightarrow^* \langle p' \rangle(c') \iff \exists t. p\gamma | p' \rightarrow t \land s(t) = c' \\
\langle p\gamma \rangle(\varepsilon) \rightarrow^* \langle p' w' \rangle(c') \land w' \neq \varepsilon \iff \exists t. p\gamma \rightarrow t \land c(t) = \langle p' w' \rangle(c')
\]

• Need to prove both propositions simultaneously
  • But may separate \Rightarrow and \Leftarrow directions
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:
  \[ \langle p\gamma \rangle(\varepsilon) \rightarrow^* \langle p' \rangle(c') \iff \exists t. p\gamma | p' \rightarrow t \land s(t) = c' \]
  \[ \langle p\gamma \rangle(\varepsilon) \rightarrow^* \langle p'w' \rangle(c') \land w' \neq \varepsilon \iff \exists t. p\gamma \rightarrow t \land c(t) = \langle p'w' \rangle(c') \]

• 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:
  \[
  \langle pw \rangle([c]) \xrightarrow{s^*} \langle p' w' \rangle(l') \iff \\
  \exists c' l'' s_1 s_2. \ l' = c' l'' \land s \in s_1 \otimes s_2 \land \langle pw \rangle(\varepsilon) \xrightarrow{s_1^*} \langle p' w' \rangle(l'') \land c \xrightarrow{s_2^*} c'
  \]

- Proof, by induction on number of steps:
  \[
  \langle p_\gamma \rangle(\varepsilon) \to^* \langle p' \rangle(c') \iff \exists t. p_\gamma \mid p' \to t \land s(t) = c'
  \]
  \[
  \langle p_\gamma \rangle(\varepsilon) \to^* \langle p' w' \rangle(c') \land w' \neq \varepsilon \iff \exists t. p_\gamma \to t \land c(t) = \langle p' w' \rangle(c')
  \]

- Need to prove both propositions simultaneously
- But may separate \(\implies\) and \(\iff\) directions
Example Proof Step

- Example step for $\Rightarrow$-direction

$\langle p_\gamma \rangle(\varepsilon) \rightarrow^* \langle p' \rangle(l') \implies \exists t. p_\gamma | p' \rightarrow t \land s(t) = l'$

$\langle p_\gamma \rangle(\varepsilon) \rightarrow^* \langle p' w' \rangle(l') \land w' \neq \varepsilon \implies \exists t. p_\gamma \rightarrow t \land c(t) = \langle p' w' \rangle(l')$
Example Proof Step

- Example step for ⇒-direction

\[
\langle p_\gamma \rangle (\varepsilon) \rightarrow^{*} \langle p' \rangle (l') \quad \Longrightarrow \quad \exists t. p_\gamma | p' \rightarrow t \land s(t) = l' \\
\langle p_\gamma \rangle (\varepsilon) \rightarrow^{*} \langle p' w' \rangle (l') \land w' \neq \varepsilon \quad \Longrightarrow \quad \exists t. p_\gamma \rightarrow t \land c(t) = \langle p' w' \rangle (l')
\]

- Case: Returning path makes a spawn-step
Example Proof Step

- Example step for ⇒-direction

\[ \langle p_\gamma \rangle (\varepsilon) \rightarrow^* \langle p' \rangle (l') \implies \exists t. p_\gamma | p' \rightarrow t \land s(t) = l' \]

\[ \langle p_\gamma \rangle (\varepsilon) \rightarrow^* \langle p' w' \rangle (l') \land w' \neq \varepsilon \implies \exists t. p_\gamma \rightarrow t \land c(t) = \langle p' w' \rangle (l') \]

- Case: Returning path makes a spawn-step
  - We have \( r := p_\gamma \leftrightarrow \hat{p}_\gamma \triangleright p_1 \gamma_1 \in \Delta \) and \( \langle \hat{p}_\gamma \rangle (p_1 \gamma_1) \rightarrow^* \langle p' \rangle (c') \)
Example Proof Step

- Example step for ⇒-direction

\[ \langle p_\gamma \rangle (\varepsilon) \rightarrow^* \langle p' \rangle (l') \implies \exists t. p_\gamma | p' \rightarrow t \land s(t) = l' \]

\[ \langle p_\gamma \rangle (\varepsilon) \rightarrow^* \langle p' w' \rangle (l') \land w' \neq \varepsilon \implies \exists t. p_\gamma \rightarrow t \land c(t) = \langle p' w' \rangle (l') \]

- Case: Returning path makes a spawn-step
  - We have \( r := p_\gamma \leftrightarrow \hat{p}_\gamma \triangleright p_1 \gamma_1 \in \Delta \) and \( \langle \hat{p}_\gamma \rangle (p_1 \gamma_1) \rightarrow^* \langle p' \rangle (c') \)
  - Using indep-steps, to separate executions of spawned and spawning thread, we obtain \( c', l'' \) where

\[ l' = c' l'' \land \langle \hat{p}_\gamma \rangle \varepsilon \rightarrow^* \langle p' \rangle (l'') \land \langle p_1 \gamma_1 \rangle (\varepsilon) \rightarrow^* c' \]
Example Proof Step

- Example step for $\Rightarrow$-direction

$\langle p^\gamma \rangle (\varepsilon) \rightarrow^* \langle p' \rangle (l') \implies \exists t. p^\gamma \upharpoonright p' \rightarrow t \land s(t) = l'$

$\langle p^\gamma \rangle (\varepsilon) \rightarrow^* \langle p' w' \rangle (l') \land w' \neq \varepsilon \implies \exists t. p^\gamma \rightarrow t \land c(t) = \langle p' w' \rangle (l')$

- Case: Returning path makes a spawn-step
  - We have $r := p^\gamma \leftrightarrow \hat{p}^\gamma \triangleright p_1^\gamma_1 \in \Delta$ and $\langle \hat{p}^\gamma \rangle (p_1^\gamma_1) \rightarrow^* \langle p' \rangle (c')$
  - Using indep-steps, to separate executions of spawned and spawning thread, we obtain $c', l''$ where

  $l' = c' l'' \land \langle \hat{p}^\gamma \rangle \varepsilon \rightarrow^* \langle p' \rangle (l'') \land \langle p_1^\gamma_1 \rangle (\varepsilon) \rightarrow^* c'$

  - With IH, we obtain $t_1, t_2$ with

    $\hat{p}^\gamma \upharpoonright p' \rightarrow t_1 \land s(t_1) = l'' \land p_1^\gamma_1 \rightarrow t_2 \land c(t_2) = c'$
Example Proof Step

- Example step for \(\Rightarrow\)-direction

\[
\langle p_\gamma \rangle (\varepsilon) \rightarrow^* \langle p' \rangle (l') \quad \Longrightarrow \quad \exists t. p_\gamma | p' \rightarrow t \wedge s(t) = l'
\]

\[
\langle p_\gamma \rangle (\varepsilon) \rightarrow^* \langle p' w' \rangle (l') \wedge w' \neq \varepsilon \quad \Longrightarrow \quad \exists t. p_\gamma \rightarrow t \wedge c(t) = \langle p' w' \rangle (l')
\]

- Case: Returning path makes a spawn-step
  - We have \(r := p_\gamma \leftrightarrow \hat{p}_\gamma \triangleright p_1 \gamma_1 \in \Delta\) and \(\langle \hat{p}_\gamma \rangle (p_1 \gamma_1) \rightarrow^* \langle p' \rangle (c')\)
  - Using indep-steps, to separate executions of spawned and spawning thread, we obtain \(c', l''\) where

\[
l' = c' l'' \wedge \langle \hat{p}_\gamma \rangle \varepsilon \rightarrow^* \langle p' \rangle (l'') \wedge \langle p_1 \gamma_1 \rangle (\varepsilon) \rightarrow^* c'
\]

- With IH, we obtain \(t_1, t_2\) with

\[
\hat{p}_\gamma | p' \rightarrow t_1 \wedge s(t_1) = l'' \wedge p_1 \gamma_1 \rightarrow t_2 \wedge c(t_2) = c'
\]

- By definition of the rules for \(A_M\), we get

\[
p_\gamma | p' \rightarrow \langle r \rangle (\hat{p}_\gamma | p', p_1 \gamma_1) \rightarrow \langle r \rangle (t_1, t_2)
\]
Example Proof Step

- Example step for $\Rightarrow$-direction

\[
\langle p_\gamma \rangle (\varepsilon) \to^* \langle p' \rangle (l') \quad \Longrightarrow \quad \exists t. p_\gamma | p' \to t \land s(t) = l'
\]
\[
\langle p_\gamma \rangle (\varepsilon) \to^* \langle p' w' \rangle (l') \land w' \neq \varepsilon \quad \Longrightarrow \quad \exists t. p_\gamma \to t \land c(t) = \langle p' w' \rangle (l')
\]

- Case: Returning path makes a spawn-step
  - We have $r := p_\gamma \leftrightarrow \hat{p}_\gamma \triangleright p_1 \gamma_1 \in \Delta$ and $\langle \hat{p}_\gamma \rangle (p_1 \gamma_1) \to^* \langle p' \rangle (c')$
  - Using indep-steps, to separate executions of spawned and spawning thread, we obtain $c'$, $l''$ where

\[
l' = c'l'' \land \langle \hat{p}_\gamma \rangle \varepsilon \to^* \langle p' \rangle (l'') \land \langle p_1 \gamma_1 \rangle (\varepsilon) \to^* c'
\]

- With IH, we obtain $t_1, t_2$ with

\[
\hat{p}_\gamma | p' \to t_1 \land s(t_1) = l'' \land p_1 \gamma_1 \to t_2 \land c(t_2) = c'
\]

- By definition of the rules for $A_M$, we get

\[
p_\gamma | p' \to \langle r \rangle \langle \hat{p}_\gamma | p', p_1 \gamma_1 \rangle \to \langle r \rangle (t_1, t_2)
\]

- 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?"

- Or it may be a false positive due to over-approximation
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 $[l \text{ (acquire)}]$ and $]l \text{ (release)}$ 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

  - Thread 1 executes:
    
    - \([0? 0! 0]
    - \([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! 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

- 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
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? [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
  Thread 1: [0?]0! [0]0?[0!]0
  Thread 2: [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.
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? [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

Thread 1:

\[0 \to 1 \to l_1 \to l_1 \to l_2 \to \text{start of output}\]

Thread 2:

\[0 \to 1 \to l_2 \to l_2 \to l_1 \to \text{start of output}\]

If one thread starts before the other has finished initialization, the other will be stuck at \([l_i \to l_i]\) forever.

Thus, final states of PDSs simultaneously reachable, iff encoded CF-languages have non-empty intersection.
• 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: $[0!][1!][l_1][l_2] <\text{start of output}>$
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: $[0!][1!][l_1][l_2]$ <start of output>
  - Thread 2: $[0?][1?][l_2][l_1]$ <start of output>

If one thread starts before the other has finished initialization, the other will be stuck at $l_i$ forever
Thus, final states of PDSs simultaneously reachable, iff encoded CF-languages have non-empty intersection
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_1l_2 <\text{start of output}>
- Thread 2: $[0?1?]l_2l_1 <\text{start of output}>

If one thread starts before the other has finished initialization, the other will be stuck at $[l_i]l_i$ forever.
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: $[0!1![l_1]l_1l_2$ <start of output>
- Thread 2: $[0?1?[l_2]l_2l_1$ <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

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 $l_1 \lor l_2 \lor l_3$: Nondeterministically run one of $[l_i; l_i]$
- Enforce correct order of guessing assignment and checking: One additional lock
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 $l_1 \lor l_2 \lor l_3$: 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 $l_1 \lor l_2 \lor l_3$: 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 $\overline{x}_i$
  • Formula $\Phi = \bigwedge_{i=1}^{m} \bigvee_{j=1}^{3} l_{ij}$, 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 $\Phi = \land_{i=1}^{m} \lor_{j=1}^{3} l_{ij}$, where the $l_{ij}$ are literals
    - $\lor_{j=1}^{3} l_{ij}$ is called clause
Reduction to 3-SAT

- Reminder (3-SAT)
  - Variables $x_0, \ldots, x_n$, literal: $x_i$ or $\bar{x}_i$
  - Formula $\Phi = \bigwedge_{i=1}^{m} \bigvee_{j=1}^{3} l_{ij}$, where the $l_{ij}$ are literals
    - $\bigvee_{j=1}^{3} l_{ij}$ is called clause
  - It is NP-complete to decide whether $\Phi$ is satisfiable.
Reduction to 3-SAT

• Reminder (3-SAT)
  • Variables $x_0, \ldots, x_n$, literal: $x_i$ or $\overline{x}_i$
  • Formula $\Phi = \bigwedge_{i=1}^{m} \bigvee_{j=1}^{3} l_{ij}$, where the $l_{ij}$ are literals
    • $\bigvee_{j=1}^{3} l_{ij}$ is called clause
  • It is NP-complete to decide whether $\Phi$ is satisfiable.
    • i.e. whether there is a valuation of the variables such that $\Phi$ holds.
Reduction to 3-SAT

\[ \text{ass}(i): \]
\[
\text{if} \ldots \text{then} \{
    \text{acquire} \ x_i \ \text{ass}(i+1) \ \text{release} \ x_i
\}
\text{else} \{
    \text{acquire} \ \bar{x}_i \ \text{ass}(i+1) \ \text{release} \ \bar{x}_i
\}
\text{return}
\]

\[ \text{ass}(n+1): \]
\[
\text{acquire}(s); \ \text{release}(s); \]
\text{label1: return}

\[ \text{thread1: ass}(1) \]

\[ \text{check}(i): \]
\[
\text{if} \ldots \{
    \text{acquire} \ l_{i1}; \ \text{release} \ l_{i1};
\}\text{else if} \ldots \{
    \text{acquire} \ l_{i2}; \ \text{release} \ l_{i2};
\}\text{else} \{
    \text{acquire} \ l_{i3}; \ \text{release} \ l_{i3};
\}
\]

\[ \text{thread2:} \]
\[
\text{acquire}(s);
\text{check}(1); \ldots; \text{check}(m);
\text{label2: skip}
\text{release}(s)
\]

- 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

1 Introduction

2 Basics

3 Alternative Representations of Regular Languages

4 Model-Checking concurrent Systems
   Motivation
   Pushdown Systems
   Dynamic Pushdown Networks
   Acquisition Histories
   Acquisition Histories for DPN
2-PDS with locks

- Two PDS with locks. Both share same rules.
2-PDS with locks

- Two PDS with locks. Both share same rules.
  - $M = (P, \Gamma, \text{Act}, \mathbb{L}, p_1^0 \gamma_1^0, p_2^0 \gamma_2^0, \Delta)$

Assumption: Locks are well-nested and non-reentrant

In particular, thread does not free "foreign" locks
2-PDS with locks

- Two PDS with locks. Both share same rules.
  - $M = (P, \Gamma, \text{Act}, \mathbb{L}, p_1^0 \gamma_1^0, p_2^0 \gamma_2^0, \Delta)$
    - $P, \Gamma, \Delta$: States, stack alphabet, rules
Two PDS with locks. Both share same rules.

- \( M = (P, \Gamma, \text{Act}, \mathbb{L}, p_1^0 \gamma_1^0, p_2^0 \gamma_2^0, \Delta) \)
  - \( P, \Gamma, \Delta \): States, stack alphabet, rules
  - \( \text{Act} = \text{Act}_{nl} \cup \{x \mid x \in \mathbb{L}\} \cup \{\} \cup \{x \mid x \in \mathbb{L}\} \)
2-PDS with locks

- Two PDS with locks. Both share same rules.
  - $M = (P, \Gamma, \text{Act}, \mathbb{L}, p_1^0 \gamma_1^0, p_2^0 \gamma_2^0, \Delta)$
    - $P, \Gamma, \Delta$: States, stack alphabet, rules
    - $\text{Act} = \text{Act}_{nl} \cup \{x \mid x \in \mathbb{L}\} \cup \{x \mid x \in \mathbb{L}\}$
    - $\mathbb{L}$: Finite set of locks

Assumption: Locks are well-nested and non-reentrant

In particular, thread does not free "foreign" locks
2-PDS with locks

- Two PDS with locks. Both share same rules.
  - \( M = (P, \Gamma, Act, \mathbb{L}, p^0_1 \gamma^0_1, p^0_2 \gamma^0_2, \Delta) \)
    - \( P, \Gamma, \Delta \): States, stack alphabet, rules
    - \( Act = Act_{nl} \cup \{[[x] \mid x \in \mathbb{L}\} \cup \}x \mid x \in \mathbb{L}\} \)
    - \( \mathbb{L} \): Finite set of locks
    - \( p^0_1 \gamma^0_1, p^0_2 \gamma^0_2 \): Initial states of left and right PDS

Assumption: Locks are well-nested and non-reentrant

In particular, thread does not free "foreign" locks
2-PDS with locks

- Two PDS with locks. Both share same rules.
  - $M = (P, \Gamma, \text{Act}, \mathbb{L}, p_0^0, \gamma_1^0, p_2^0, \gamma_2^0, \Delta)$
    - $P, \Gamma, \Delta$: States, stack alphabet, rules
    - $\text{Act} = \text{Act}_{nl} \cup [x \mid x \in \mathbb{L}] \cup \{[x] \mid x \in \mathbb{L}\}$
    - $\mathbb{L}$: Finite set of locks
    - $p_0^0, \gamma_1^0, p_2^0, \gamma_2^0$: Initial states of left and right PDS
- Assumption: Locks are well-nested and non-reentrant
Two PDS with locks. Both share same rules.

- \( M = (P, \Gamma, \text{Act}, \mathbb{L}, p_1^0 \gamma_1^0, p_2^0 \gamma_2^0, \Delta) \)
  - \( P, \Gamma, \Delta \): States, stack alphabet, rules
  - \( \text{Act} = \text{Act}_{nl} \cup \{[x | x \in \mathbb{L}] \} \cup \{]x | x \in \mathbb{L}\} \)
  - \( \mathbb{L} \): 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: \((p_1 w_1, p_2 w_2, L) \in P\Gamma^* \times P\Gamma^* \times 2^L\)
Semantics

- Configurations: \((p_1w_1, p_2w_2, L) \in P\Gamma^* \times P\Gamma^* \times 2^L\)
  - \(\text{cond}([x, L]) = x \not\in L, \text{eff}([x, L]) = L \cup \{x\}\)
Semantics

- Configurations: \((p_1 w_1, p_2 w_2, L) \in P\Gamma^* \times P\Gamma^* \times 2^L\)
  - \(\text{cond}([x, L]) = x \notin L, \text{eff}([x, L]) = L \cup \{x\}\)
  - \(\text{cond}([], L) = \text{true}, \text{eff}([], L) = L \setminus \{x\}\)
Semantics

- Configurations: $(p_1 w_1, p_2 w_2, L) \in P\Gamma^* \times P\Gamma^* \times 2^L$
  - $\text{cond}([x, L]) = x \notin L$, $\text{eff}([x, L]) = L \cup \{x\}$
  - $\text{cond}([x, L]) = \text{true}$, $\text{eff}([x, L]) = L \setminus \{x\}$
  - $\text{cond}(a, L) = \text{true}$, $\text{eff}(a, L) = L$ for $a \in \text{Act}_{nl}$
• Configurations: \((p_1 w_1, p_2 w_2, L) \in P \Gamma^* \times P \Gamma^* \times 2^L\)
  
  - \(\text{cond}(\lfloor x \rfloor, L) = x \notin L, \text{eff}(\lfloor x \rfloor, L) = L \cup \{x\}\)
  
  - \(\text{cond}(\rfloor x \rfloor, L) = \text{true}, \text{eff}(\rfloor x \rfloor, L) = L \setminus \{x\}\)

  - \(\text{cond}(a, L) = \text{true}, \text{eff}(a, L) = L\) for \(a \in \text{Act}_{nl}\)

• Step

\[
\begin{align*}
(p_\gamma w_1, p_2 w_2, L) \xrightarrow{a_{\text{ls}}} (p' w' w_1, p_2 w_2, \text{eff}(a, L)) & \quad \text{if } p_\gamma \xrightarrow{\hat{a}} p' w' \in \Delta \text{ and } \text{cond}(a, L) \text{ (left)} \\
(p_1 w_1, p_\gamma w_2, L) \xrightarrow{a_{\text{ls}}} (p_1 w_1, p' w' w_2, \text{eff}(a, L)) & \quad \text{if } p_\gamma \xrightarrow{\hat{a}} p' w' \in \Delta \text{ and } \text{cond}(a, L) \text{ (right)}
\end{align*}
\]
Lock sensitive scheduling

- Idea: Abstraction from PDS

1. Check whether two execution sequences can be interleaved
2. Configurations: \((l_1, l_2, L) \in \text{Act}^* \times \text{Act}^* \times 2\)
3. Step: \((al_1, l_2, L) \xrightarrow{a} (l_1, l_2, \text{eff}(a, L))\) if \(\text{cond}(a, L)\) (left)
4. \(\xrightarrow{a} (l_1, l_2, \text{eff}(a, L))\) if \(\text{cond}(a, L)\) (right)
5. Lemma: \((p_1w_1, p_2w_2, L) \xrightarrow{l} * (p_1'w_1, p_2'w_2, L')\) iff \(\exists l_1, l_2. p_1w_1l_1 \xrightarrow{*} p_1'w_1 \land p_2w_2l_2 \xrightarrow{*} p_2'w_2 \land (l_1, l_2, L) l \rightarrow * (\epsilon, \epsilon, L')\)
6. Intuition: Schedule lock-insensitive executions of the single PDSs
7. Proof: Straightforward simulation proof
Lock sensitive scheduling

- Idea: Abstraction from PDS
  - Check whether two execution sequences can be interleaved

• Lemma \((p_1w_1, p_2w_2, L) \rightarrow^* (p'_1w'_1, p'_2w'_2, L')) \iff \exists l_1, l_2. (p_1w_1 \rightarrow^* p'_1w'_1 \land p_2w_2 \rightarrow^* p'_2w'_2 \land (l_1, l_2, L) \rightarrow^* (\varepsilon, \varepsilon, L'))

• Intuition: Schedule lock-insensitive executions of the single PDSs

• Proof: Straightforward simulation proof
Lock sensitive scheduling

- Idea: Abstraction from PDS
  - Check whether two execution sequences can be interleaved
- Configurations: \((l_1, l_2, L) \in \text{Act}^* \times \text{Act}^* \times 2^L\)
Lock sensitive scheduling

- Idea: Abstraction from PDS
  - Check whether two execution sequences can be interleaved
- Configurations: \((l_1, l_2, L) \in \text{Act}^* \times \text{Act}^* \times 2^L\)
- Step

\[
(l_1, l_2, L) \xrightarrow{a} (l_1, l_2, \text{eff}(a, L)) \quad \text{if } \text{cond}(a, L) \quad \text{(left)}
\]

\[
(l_1, al_2, L) \xrightarrow{a} (l_1, l_2, \text{eff}(a, L)) \quad \text{if } \text{cond}(a, L) \quad \text{(right)}
\]
Lock sensitive scheduling

- Idea: Abstraction from PDS
  - Check whether two execution sequences can be interleaved
- Configurations: \((l_1, l_2, L) \in \text{Act}^* \times \text{Act}^* \times 2^L\)
- Step

\[(al_1, l_2, L) \xleftrightarrow{a} (l_1, l_2, \text{eff}(a, L)) \quad \text{if } \text{cond}(a, L) \quad \text{(left)}\]

\[(l_1, al_2, L) \xleftrightarrow{a} (l_1, l_2, \text{eff}(a, L)) \quad \text{if } \text{cond}(a, L) \quad \text{(right)}\]

- Lemma

\[(p_1 w_1, p_2 w_2, L) \xrightarrow{l}^* (p_1' w_1', p_2' w_2', L') \quad \text{iff } \exists l_1, l_2. \ p_1 w_1 \xrightarrow{l_1}^* p_1' w_1' \land p_2 w_2 \xrightarrow{l_2}^* p_2' w_2' \land (l_1, l_2, L) \xrightarrow{l}^* (\varepsilon, \varepsilon, L')\]
Lock sensitive scheduling

- Idea: Abstraction from PDS
  - Check whether two execution sequences can be interleaved
- Configurations: \((l_1, l_2, L) \in \text{Act}^* \times \text{Act}^* \times 2^L\)
- Step

\[
(al_1, l_2, L) \xrightarrow{a} (l_1, l_2, \text{eff}(a, L)) \quad \text{if } \text{cond}(a, L) \quad \text{(left)}
\]

\[
(l_1, al_2, L) \xrightarrow{a} (l_1, l_2, \text{eff}(a, L)) \quad \text{if } \text{cond}(a, L) \quad \text{(right)}
\]

- Lemma

\[
(p_1 w_1, p_2 w_2, L) \xrightarrow{I}^* (p'_1 w'_1, p'_2 w'_2, L')
\]

iff \(\exists l_1, l_2 \cdot p_1 w_1 \xrightarrow{l_1}^* p'_1 w'_1 \land p_2 w_2 \xrightarrow{l_2}^* p'_2 w'_2 \land (l_1, l_2, L) \xrightarrow{I}^* (\varepsilon, \varepsilon, L')\)

- 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:** \((l_1, l_2, L) \in \text{Act}^* \times \text{Act}^* \times 2^L\)
- **Step**

\[
(al_1, l_2, L) \xrightarrow{a} (l_1, l_2, \text{eff}(a, L)) \quad \text{if} \ cond(a, L) \quad \text{(left)}
\]

\[
(l_1, al_2, L) \xrightarrow{a} (l_1, l_2, \text{eff}(a, L)) \quad \text{if} \ cond(a, L) \quad \text{(right)}
\]

- **Lemma**

\[
(p_1 w_1, p_2 w_2, L) \xrightarrow{l_1}^* (p'_1 w'_1, p'_2 w'_2, L')
\]

iff \(\exists l_1, l_2. \ p_1 w_1 \xrightarrow{l_1}^* p'_1 w'_1 \wedge p_2 w_2 \xrightarrow{l_2}^* p'_2 w'_2 \wedge (l_1, l_2, L) \xrightarrow{l}^* (\varepsilon, \varepsilon, L')\)

- **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 $\circ$.
  - $X_2 ::= \circ(XN, XN)$
Execution trees of 2-PDS

- Intuitively: Append execution trees of left and right PDS to binary root node $\circ$.
  - $X_2 ::= \circ(X_N, X_N)$
- 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 $i \rightarrow \circ(p_1^0 \gamma_1^0, p_2^0 \gamma_2^0)$
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 \( i \rightarrow \circ(p_1^0 \gamma_1^0, p_2^0 \gamma_2^0) \)
- We have (with lemma from previous slide)

\[
(p_1 w_1, p_2 w_2, L) \xrightarrow{I}^* (p_1' w_1', p_2' w_2', L')
\]
iff \( \exists t_1, t_2. i \rightarrow \circ(t_1, t_2) \land c(t_1) = p_1' w_1' \land c(t_2) = p_2' w_2' \land (a(t_1), a(t_2), L) \xrightarrow{I}^* (\varepsilon, \varepsilon, L') \)

- \( c : XN \rightarrow conf \) extracts reached configuration from execution tree
- \( a : XN \rightarrow Act^* \) extracts labeling sequence from execution tree (cf. Homework 9.2)
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 $i \rightarrow \circ(p_1^0 \gamma_1^0, p_2^0 \gamma_2^0)$
- We have (with lemma from previous slide)
  \[
  (p_1 w_1, p_2 w_2, L) \xrightarrow{I}^* (p'_1 w'_1, p'_2 w'_2, L')
  \iff \exists t_1, t_2. i \rightarrow \circ(t_1, t_2) \land c(t_1) = p'_1 w'_1 \land c(t_2) = p'_2 w'_2
  \land (a(t_1), a(t_2), L) \xrightarrow{I}^* (\varepsilon, \varepsilon, L')
  \]

- Where $c : XN \rightarrow \text{conf}$ extracts reached configuration from execution tree
  and $a : XN \rightarrow \text{Act}^*$ 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 $(l_1, l_2, \emptyset) \rightarrow^* (\varepsilon, \varepsilon, \_)$
  - Sets of which can be computed by tree automaton over execution trees
- Thus, we get a tree automaton for schedulable execution trees.
• 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

- 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: 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 \( [x \in \text{action}] \) 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

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: 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 $l \in \text{Act}^*$, we define $ah(l) := (A(l), G(l))$ where

  - $A(l) = \emptyset$
  - $A(l) = A(l')$ if $a \in \text{Act}^* l$ or $a = x$ for $x \in L$
  - $A(l) = A(l') \cup \{x\}$ if $x \not\in l$
  - $G(l) = \emptyset$
  - $G(l) = G(l')$ if $x \in l$
  - $G(l) = G(l') \cup \{(x, \text{acq}(l))\}$ if $x \not\in l$

- Lemma $(l_1, l_2, \emptyset) \rightarrow^* (\varepsilon, \varepsilon, \_)$ if $A(l_1) \cap A(l_2) = \emptyset \land \text{acyclic}(G(l_1) \cup G(l_2))$
Acquisition Histories: Definition

- Given an execution sequence \( l \in \text{Act}^* \), we define \( ah(l) := (A(l), G(l)) \) where
  - \( A(l) \subseteq \mathbb{L} \) is the set of finally acquired locks:
    \[
    \begin{align*}
    A(\varepsilon) &= \emptyset \\
    A(al) &= A(l) & \text{if } a \in \text{Act}_{nl} \text{ or } a = [x] \text{ for } x \in \mathbb{L} \\
    A([x]l) &= A(l) & \text{if } [x] \in l \\
    A([x]l) &= A(l) \cup \{x\} & \text{if } [x] \not\in l
    \end{align*}
    \]
Acquisition Histories: Definition

- Given an execution sequence \( l \in \text{Act}^* \), we define \( ah(l) := (A(l), G(l)) \) where
  - \( A(l) \subseteq \mathbb{L} \) is the set of finally acquired locks:
    \[
    \begin{align*}
    A(\varepsilon) &= \emptyset, \\
    A(al) &= A(l) \quad \text{if } a \in \text{Act}_{nl} \text{ or } a = ]_x \text{ for } x \in \mathbb{L}, \\
    A([x]l) &= A(l) \quad \text{if } ]_x \in l, \\
    A([x]l) &= A(l) \cup \{x\} \quad \text{if } ]_x \notin l.
    \end{align*}
    \]
  - \( G(l) \subseteq \mathbb{L} \times \mathbb{L} \) is the lock graph:
    \[
    \begin{align*}
    G(\varepsilon) &= \emptyset, \\
    G(al) &= G(l) \quad \text{if } a \in \text{Act}_{nl} \text{ or } a = ]_x \text{ for } x \in \mathbb{L}, \\
    G([x]l) &= G(l) \quad \text{if } ]_x \in l, \\
    G([x]l) &= G(l) \cup \{x\} \times \text{acq}(l) \quad \text{if } ]_x \notin l.
    \end{align*}
    \]
    where \( \text{acq}(l) := \{x \mid ]_x \in l\} \).
Acquisition Histories: Definition

- Given an execution sequence \( l \in \text{Act}^* \), we define \( ah(l) := (A(l), G(l)) \) where
  - \( A(l) \subseteq \mathbb{L} \) is the set of finally acquired locks:
    - \( A(\varepsilon) = \emptyset \)
    - \( A(al) = A(l) \) if \( a \in \text{Act}_{nl} \) or \( a = [x] \) for \( x \in \mathbb{L} \)
    - \( A([x]l) = A(l) \) if \( [x] \in l \)
    - \( A([x]l) = A(l) \cup \{x\} \) if \( [x] \notin l \)
  - \( G(l) \subseteq \mathbb{L} \times \mathbb{L} \) is the lock graph:
    - \( G(\varepsilon) = \emptyset \)
    - \( G(al) = G(l) \) if \( a \in \text{Act}_{nl} \) or \( a = [x] \) for \( x \in \mathbb{L} \)
    - \( G([x]l) = G(l) \) if \( [x] \in l \)
    - \( G([x]l) = G(l) \cup \{x\} \times \text{acq}(l) \) if \( [x] \notin l \)

  where \( \text{acq}(l) := \{x \mid [x] \in l\} \)

- Lemma

\[
(l_1, l_2, \emptyset) \rightarrow^* (\varepsilon, \varepsilon, \_\_ \_\_) \text{ iff } A(l_1) \cap A(l_2) = \emptyset \land \text{acyclic}(G(l_1) \cup G(l_2))
\]
Proof ideas

- \[ L, (l_1, l_2, L) \rightarrow^* (\varepsilon, \varepsilon, _-^*) = \Rightarrow A(l_1) \cap A(l_2) = \emptyset \land \text{acyclic}(G(l_1) \cup G(l_2)) \]

- 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 \(G(l_1) \cup G(l_2)\)
Proof ideas

- \[ \iff \]
  - Generalize to

\[
\forall L. (l_1, l_2, L) \rightarrow^* (\varepsilon, \varepsilon, \_ ) \iff A(l_1) \cap A(l_2) = \emptyset \land \text{acyclic}(G(l_1) \cup G(l_2))
\]
Proof ideas

• \[ \Rightarrow \]
  • Generalize to

\[ \forall L. (l_1, l_2, L) \rightarrow^* (\varepsilon, \varepsilon, _) \Rightarrow A(l_1) \cap A(l_2) = \emptyset \land \text{acyclic} (G(l_1) \cup G(l_2)) \]

• Induction on \( \rightarrow^* \)
Proof ideas

• $\Rightarrow$
  • Generalize to
  $$\forall L. (l_1, l_2, L) \rightarrow^* (\varepsilon, \varepsilon, \_ ) \implies A(l_1) \cap A(l_2) = \emptyset \land \text{acyclic}(G(l_1) \cup G(l_2))$$
  
  • Induction on $\rightarrow^*$
    • Interesting case: First step is final acquisition: $[x}$
Proof ideas

• \( \Rightarrow \)
  • Generalize to

\[
\forall L. (l_1, l_2, L) \rightarrow^* (\varepsilon, \varepsilon, \_ ) \implies A(l_1) \cap A(l_2) = \emptyset \land \text{acyclic}(G(l_1) \cup G(l_2))
\]

• Induction on \( \rightarrow^* \)
  • Interesting case: First step is final acquisition: \( [x \]
  • \([x \) will not occur in remaining execution
Proof ideas

• $\implies$
  
  • Generalize to

  $\forall L. (l_1, l_2, L) \rightarrow^* (\varepsilon, \varepsilon, _) \implies A(l_1) \cap A(l_2) = \emptyset \land \text{acyclic}(G(l_1) \cup G(l_2))$

• 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

• \[\Rightarrow\]
  • Generalize to

\[
\forall L. \ (l_1, l_2, L) \rightarrow^* (\varepsilon, \varepsilon, _) \implies A(l_1) \cap A(l_2) = \emptyset \land \text{acyclic}(G(l_1) \cup G(l_2))
\]

• 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

• \[\Leftarrow\]
Proof ideas

- $\implies$
  - Generalize to

  $\forall L. (l_1, l_2, L) \rightarrow^* (\varepsilon, \varepsilon, _) \implies A(l_1) \cap A(l_2) = \emptyset \land \text{acyclic}(G(l_1) \cup G(l_2))$

- 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

- $\iff$
  - Generalize to

  $A(l_1) \cap A(l_2) = \emptyset \land \text{acyclic}(G(l_1) \cup G(l_2))$
  
  $\implies \forall L. L \cap (\text{acq}(l_1) \cup \text{acq}(l_2)) = \emptyset \implies (l_1, l_2, L) \rightarrow^* (\varepsilon, \varepsilon, _) \quad (1)$
Proof ideas

- \[\Rightarrow\]
  - Generalize to
  \[\forall L. (l_1, l_2, L) \rightarrow^* (\varepsilon, \varepsilon, _) \implies A(l_1) \cap A(l_2) = \emptyset \land \text{acyclic}(G(l_1) \cup G(l_2))\]
  - 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
  - \(\Leftarrow\)
  - Generalize to
  \[A(l_1) \cap A(l_2) = \emptyset \land \text{acyclic}(G(l_1) \cup G(l_2))\]
  \[\implies \forall L. L \cap (\text{acq}(l_1) \cup \text{acq}(l_2)) = \emptyset \implies (l_1, l_2, L) \rightarrow^* (\varepsilon, \varepsilon, _) \quad (1)\]
  - Induction on \(|l_1| + |l_2|\)
Proof ideas

- $\Rightarrow$
  - Generalize to
    $$\forall L. \ (l_1, l_2, L) \rightarrow^* (\varepsilon, \varepsilon, _) \implies A(l_1) \cap A(l_2) = \emptyset \land \text{acyclic}(G(l_1) \cup G(l_2))$$
  - 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

- $\Leftarrow$
  - Generalize to
    $$A(l_1) \cap A(l_2) = \emptyset \land \text{acyclic}(G(l_1) \cup G(l_2)) \implies \forall L. \ L \cap (\text{acq}(l_1) \cup \text{acq}(l_2)) = \emptyset \implies (l_1, l_2, L) \rightarrow^* (\varepsilon, \varepsilon, _) \ (1)$$
    - Induction on $|l_1| + |l_2|$
      - Schedule usages of locks first
Proof ideas

• \(\rightarrow\)
  - Generalize to
  \[
  \forall L. (l_1, l_2, L) \rightarrow^* (\varepsilon, \varepsilon, _) \implies A(l_1) \cap A(l_2) = \emptyset \land \text{acyclic}(G(l_1) \cup G(l_2))
  \]

• 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

• \(\leftarrow\)
  - Generalize to
  \[
  A(l_1) \cap A(l_2) = \emptyset \land \text{acyclic}(G(l_1) \cup G(l_2))
  \implies \forall L. L \cap (\text{acq}(l_1) \cup \text{acq}(l_2)) = \emptyset \implies (l_1, l_2, L) \rightarrow^* (\varepsilon, \varepsilon, _) \tag{1}
  \]

• 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

1. Introduction

2. Basics

3. Alternative Representations of Regular Languages

4. Model-Checking concurrent Systems
   - Motivation
   - Pushdown Systems
   - Dynamic Pushdown Networks
   - Acquisition Histories
   - Acquisition Histories for DPN
DPNs with locks

- Same ideas as for 2-PDS
DPNs with locks

- Same ideas as for 2-PDS
- \( M = (P, \Gamma, \text{Act}, L, p_0, \gamma_0, \Delta) \)
DPNs with locks

- Same ideas as for 2-PDS
- $M = (P, \Gamma, \text{Act}, L, p_0, \gamma_0, \Delta)$
  - $P, \Gamma, \Delta$: States, stack alphabet, rules (with spawns)
DPNs with locks

- Same ideas as for 2-PDS
- $M = (P, \Gamma, \text{Act}, L, p_0, \gamma_0, \Delta)$
  - $P, \Gamma, \Delta$: States, stack alphabet, rules (with spawns)
  - $\text{Act} = \text{Act}_{nl} \cup \{x | x \in L\} \cup \{]x | x \in L\}$

$L$: 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
DPNs with locks

- Same ideas as for 2-PDS
- \( M = (P, \Gamma, \text{Act}, L, p_0, \gamma_0, \Delta) \)
  - \( P, \Gamma, \Delta \): States, stack alphabet, rules (with spawns)
  - \( \text{Act} = \text{Act}_{nl} \cup \{[x \mid x \in L] \cup \{]x \mid x \in L\} \)
  - \( L \): Finite set of locks
  - Assumption: Locks are well-nested and non-reentrant
  - In particular, thread does not free "foreign" locks
DPNs with locks

- Same ideas as for 2-PDS
- \( M = (P, \Gamma, \text{Act}, \mathbb{L}, p_0, \gamma_0, \Delta) \)
  - \( P, \Gamma, \Delta \): States, stack alphabet, rules (with spawns)
  - \( \text{Act} = \text{Act}_{nl} \cup \{x | x \in \mathbb{L}\} \cup \{y | y \in \mathbb{L}\} \)
  - \( \mathbb{L} \): 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.
DPNs with locks

- Same ideas as for 2-PDS
- \( M = (P, \Gamma, \text{Act}, \mathbb{L}, p_0, \gamma_0, \Delta) \)
  - \( P, \Gamma, \Delta \): States, stack alphabet, rules (with spawns)
  - \( \text{Act} = \text{Act}_{nl} \cup \{ [x \mid x \in \mathbb{L}] \} \cup \{ ]x \mid x \in \mathbb{L} \} \)
  - \( \mathbb{L} \): 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
- \( M = (P, \Gamma, \text{Act}, \mathbb{L}, p_0, \gamma_0, \Delta) \)
  - \( P, \Gamma, \Delta \): States, stack alphabet, rules (with spawns)
  - \( \text{Act} = \text{Act}_{nl} \cup \{x \mid x \in \mathbb{L}\} \cup \{x \mid x \in \mathbb{L}\} \)
  - \( \mathbb{L} \): 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: $conf ::= \langle pw \rangle (conflist)$  $conflist ::= Nil | Cons (conf, conflist)$
Semantics

- As for 2-PDS: Add set of locks
  - Recall: \( \text{conf} ::= \langle pw \rangle (\text{conflist}) \) \( \text{conflist} ::= \text{Nil} | \text{Cons}(\text{conf}, \text{conflist}) \)
  - \( \text{conf}_{ls} ::= \text{conf} \times \mathbb{L} \)
• As for 2-PDS: Add set of locks
  • Recall: \( \text{conf} ::= \langle \text{pw} \rangle(\text{conflist}) \quad \text{conflist} ::= \text{Nil}|\text{Cons}(\text{conf}, \text{conflist}) \)
  • \( \text{conf}_\text{ls} ::= \text{conf} \times \mathbb{L} \)

• Step relation:

\[
(c, L) \xrightarrow{a} (c', \text{eff}(a, L)) \text{ iff } \text{cond}(a, L) \land c \xrightarrow{a} c'
\]
Lock-Sensitive Scheduling

- Abstract from DPN-configurations
Lock-Sensitive Scheduling

- Abstract from DPN-configurations
- Scheduling tree:

\[ BL ::= \text{Nil} \mid \text{Cons}(a, BL) \mid \text{Spawn}(a, BL, BL) \quad \text{for all} \ a \in \text{Act} \]

\[ ST ::= \langle BL \rangle(SL) \quad SL ::= \text{Nil} \mid \text{Cons}(ST, SL) \]
Lock-Sensitive Scheduling

- Abstract from DPN-configurations
- Scheduling tree:

\[
BL ::= Nil \mid Cons(a, BL) \mid Spawn(a, BL, BL) \quad \text{for all } a \in \text{Act}
\]

\[
ST ::= \langle BL \rangle(SL) \quad SL ::= Nil \mid Cons(ST, SL)
\]

- Combination of configurations and sequences of actions to be executed
Lock-Sensitive Scheduling

- Abstract from DPN-configurations
- Scheduling tree:

\[ BL ::= \text{Nil} \mid \text{Cons}(a, BL) \mid \text{Spawn}(a, BL, BL) \text{ for all } a \in \text{Act} \]

\[ ST ::= \langle BL \rangle (SL) \quad SL ::= \text{Nil} \mid \text{Cons}(ST, SL) \]

- Combination of configurations and sequences of actions to be executed
- Each thread in configuration is labeled by actions it still has to execute
Lock-Sensitive Scheduling

- Abstract from DPN-configurations
- Scheduling tree:

\[
\begin{align*}
BL &::= Nil \mid Cons(a, BL) \mid Spawn(a, BL, BL) \quad \text{for all } a \in Act \\
ST &::= \langle BL \rangle (SL) \\
SL &::= Nil \mid Cons(ST, SL)
\end{align*}
\]

- 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:

\[ BL ::= \text{Nil} \mid \text{Cons}(a, BL) \mid \text{Spawn}(a, BL, BL) \quad \text{for all } a \in \text{Act} \]

\[ ST ::= \langle BL \rangle(SL) \quad SL ::= \text{Nil} \mid \text{Cons}(ST, SL) \]

- 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

\[ (C[\langle Cons(a, l)\rangle(s)], L) \xrightarrow{a} (C[\langle l\rangle(s)], \text{eff}(a, L)) \text{ iff } \text{cond}(a, L) \quad \text{(no-spawn)} \]

\[ (C[\langle Spawn(a, l_1, l_2)\rangle(s)], L) \xrightarrow{a} (C[\langle l_1\rangle(s[\langle l_2\rangle(\text{Nil})])], \text{eff}(a, L)) \text{ iff } \text{cond}(a, L) \quad \text{(spawn)} \]

where \( C \) is a context with exactly one occurrence of \( x_1 \).
Lock-Sensitive Scheduling

- Abstract from DPN-configurations
- Scheduling tree:

\[
BL ::= \text{ Nil } \mid \text{ Cons}(a, BL) \mid \text{ Spawn}(a, BL, BL) \quad \text{for all } a \in \text{ Act}
\]

\[
ST ::= \langle BL \rangle (SL) \\
SL ::= \text{ Nil } \mid \text{ Cons}(ST, SL)
\]

- 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

\[
(C[\langle \text{Cons}(a, l)\rangle(s)], L) \xrightarrow{a} (C[\langle l\rangle(s)], \text{eff}(a, L)) \text{ iff } \text{cond}(a, L) \quad \text{(no-spawn)}
\]

\[
(C[\langle \text{Spawn}(a, l_1, l_2)\rangle(s)], L) \xrightarrow{a} (C[\langle l_1\rangle(s[\langle l_2\rangle(\text{Nil})])], \text{eff}(a, L)) \text{ iff } \text{cond}(a, L) \quad \text{(spawn)}
\]

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 \text{ Nil}

\[
ST_{\text{term}} ::= \langle \text{Nil} \rangle (SL_{\text{term}}) \\
SL_{\text{term}} ::= \text{ Nil } \mid \text{ Cons}(ST_{\text{term}}, SL_{\text{term}})
\]
Operations on Branching Lists

- Generalized concatenation

\[
\begin{align*}
(Nil) l' & := l' \\
\text{Cons}(a, l) l' & := \text{Cons}(a, ll') \\
\text{Spawn}(a, l_1, l_2) l' & := \text{Spawn}(a, l_1 l', l_2)
\end{align*}
\]
Operations on Branching Lists

- Generalized concatenation

\[(\text{Nil})l' := l'\]
\[\text{Cons}(a, l)l' := \text{Cons}(a, ll')\]
\[\text{Spawn}(a, l_1, l_2)l' := \text{Spawn}(a, l_1l', l_2)\]

- This thread’s steps: \(\text{this} : BL \rightarrow \text{Act}^*\)

\[\text{this}(\text{Nil}) := \text{Nil}\]
\[\text{this}(\text{Cons}(a, l)) := \text{Cons}(a, \text{this}(l))\]
\[\text{this}(\text{Spawn}(a, l_1, l_2)) = \text{Cons}(a, \text{this}(l_1))\]
Operations on Branching Lists

- Generalized concatenation

\[(\text{Nil})l' := l'\]
\[\text{Cons}(a, l)l' := \text{Cons}(a, ll')\]
\[\text{Spawn}(a, l_1, l_2)l' := \text{Spawn}(a, l_1l', l_2)\]

- This thread’s steps: this : BL \to \text{Act}^*

\[\text{this}(\text{Nil}) := \text{Nil}\]
\[\text{this}(\text{Cons}(a, l)) := \text{Cons}(a, \text{this}(l))\]
\[\text{this}(\text{Spawn}(a, l_1, l_2)) = \text{Cons}(a, \text{this}(l_1))\]

- Set of steps

\[x \in \text{Nil} := \text{false}\]
\[x \in \text{Cons}(a, l) := x = a \lor x \in l\]
\[x \in \text{Spawn}(a, l_1, l_2) := x = a \lor x \in l_1 \lor x \in l_2\]
Relation of execution tree and scheduling tree

- Execution trees correspond to scheduling trees: $st : XN \rightarrow ST$ and $st' : XN \rightarrow BL$ where

  $$st(t) := \langle st'(t) \rangle (Nil)$$

  $$st' (\langle p \gamma \xrightarrow{a} p' \gamma' \rangle (t)) := \text{Cons}(a, st'(t))$$

  $$st' (\langle p \gamma \xrightarrow{a} p_1 \gamma_1 \triangleright p_2 \gamma_2 \rangle (t_1, t_2)) := \text{Spawn}(a, st'(t_1), st'(t_2))$$

  $$st' (\langle p \gamma \xrightarrow{a} p' \gamma_1 \gamma_2 \rangle^N (t)) := \text{Cons}(a, st'(t))$$

  $$st' (\langle p \gamma \xrightarrow{a} p' \gamma_1 \gamma_2 \rangle^R (t_1, t_2)) := [a] st'(t_1) st'(t_2)$$

  $$st' (\langle p \gamma \rangle) := Nil$$

  $$st' (\langle p \gamma \xrightarrow{a} p' \rangle) := \text{Cons}(a, Nil)$$
Relation of execution tree and scheduling tree

- Execution trees correspond to scheduling trees: \( st : XN \rightarrow ST \) and \( st' : XN \rightarrow BL \) where

\[
\begin{align*}
st(t) & := \langle st'(t) \rangle (\text{Nil}) \\
st'((p\gamma \overset{a}{\to} p'\gamma')(t)) & := \text{Cons}(a, st'(t)) \\
st'((p\gamma \overset{a}{\to} p_1\gamma_1 \triangleright p_2\gamma_2)(t_1, t_2)) & := \text{Spawn}(a, st'(t_1), st'(t_2)) \\
st'((p\gamma \overset{a}{\to} p'\gamma_1\gamma_2)^N(t)) & := \text{Cons}(a, st'(t)) \\
st'((p\gamma \overset{a}{\to} p'\gamma_1\gamma_2)^R(t_1, t_2)) & := [a]st'(t_1)st'(t_2) \\
st'((p\gamma)) & := \text{Nil} \\
st'((p\gamma \overset{a}{\to} p')) & := \text{Cons}(a, \text{Nil})
\end{align*}
\]

- It can be proved

\[
(\langle p_0\gamma_0 \rangle (e), \emptyset) \xrightarrow{I}^* (c', L)
\]

\[
\equiv \exists t \in XN. \exists t' \in ST_{\text{term}}. t \in L(A_M) \land c(t) = c' \land \langle st(t), \emptyset \rangle \xrightarrow{I}^* (t', L)
\]
Relation of execution tree and scheduling tree

- Execution trees correspond to scheduling trees: \( st : XN \rightarrow ST \) and \( st' : XN \rightarrow BL \) where

\[
st(t) := \langle st'(t) \rangle(\text{Nil})
\]

\[
st'(\langle p \gamma \xrightarrow{a} p' \gamma' \rangle(t)) := \text{Cons}(a, st'(t))
\]

\[
st'(\langle p \gamma \xrightarrow{a} p_1 \gamma_1 \triangleright p_2 \gamma_2 \rangle(t_1, t_2)) := \text{Spawn}(a, st'(t_1), st'(t_2))
\]

\[
st'(\langle p \gamma \xrightarrow{a} p' \gamma_1 \gamma_2 \rangle^N(t)) := \text{Cons}(a, st'(t))
\]

\[
st'(\langle p \gamma \xrightarrow{a} p' \gamma_1 \gamma_2 \rangle^R(t_1, t_2)) := [a]st'(t_1)st'(t_2)
\]

\[
st'(\langle p \gamma \rangle) := \text{Nil}
\]

\[
st'(\langle p \gamma \xrightarrow{a} p' \rangle) := \text{Cons}(a, \text{Nil})
\]

- It can be proved

\[
(\langle p_0 \gamma_0 \rangle(\varepsilon), \emptyset) \xrightarrow{l}^* (c', L)
\]

\[
\iff \exists t \in XN. \exists t' \in ST_{\text{term}}. t \in L(\mathcal{A}_M) \wedge c(t) = c' \wedge (st(t), \emptyset) \xrightarrow{l}^* (t', L)
\]

- 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

\[ A(\text{Nil}) = \emptyset \]

\[ A(\text{Spawn}(a, l_1, l_2)) = A(l_1) \cup A(l_2) \]

\[ A(\text{Cons}(a, l)) = A(l) \]

\[ A(\text{Cons}(\left[x, l_1\right])) = A(l_1) \]

\[ A(\text{Cons}(\left[x, l_1\right])) = A(l_1) \cup \{x\} \times \text{acq}(l) \]

where \[ \text{acq}(l) := \{ x \mid x \in l \} \]
Acquisition Histories for Scheduling Trees

- Assumption: Acquisition and release only on base rules
- Compute set of final acquisitions

\[ A(Nil) = \emptyset \]

\[ A(Spawn(a, l_1, l_2)) = A(l_1) \cup A(l_2) \]

\[ A(Cons(a, l)) = A(l) \quad \text{if } a \in Act_{nl} \text{ or } a = ]_x \text{ for } x \in \mathbb{L} \]

\[ A(Cons([x, l])) = A(l) \quad \text{if } ]_x \in this(l) \]

\[ A(Cons([x, l])) = A(l) \cup \{x\} \quad \text{if } ]_x \notin this(l) \]
Acquisition Histories for Scheduling Trees

- Assumption: Acquisition and release only on base rules
- Compute set of final acquisitions
  \[ A(\text{Nil}) = \emptyset \]
  \[ A(\text{Spawn}(a, l_1, l_2)) = A(l_1) \cup A(l_2) \]
  \[ A(\text{Cons}(a, l)) = A(l) \quad \text{if } a \in \text{Act}_{nl} \text{ or } a = ]_x \text{ for } x \in \mathbb{L} \]
  \[ A(\text{Cons}([x, l])) = A(l) \quad \text{if }]_x \in \text{this}(l) \]
  \[ A(\text{Cons}([x, l])) = A(l) \cup \{x\} \quad \text{if }]_x \notin \text{this}(l) \]
- Check consistency of final acquisitions
  \[ \text{fac}(\text{Nil}) = true \quad \text{fac}(\text{Cons}(a, l)) = \text{fac}(l) \quad \text{fac}(\text{Spawn}(a, l_1, l_2)) = \text{fac}(l_1) \]
Acquisition Histories for Scheduling Trees

• Assumption: Acquisition and release only on base rules

• Compute set of final acquisitions

\[
A(\text{Nil}) = \emptyset
\]

\[
A(\text{Spawn}(a, I_1, I_2)) = A(I_1) \cup A(I_2)
\]

\[
A(\text{Cons}(a, I)) = A(I)
\]

if \( a \in \text{Act}_{nl} \) or \( a = ]_x \) for \( x \in \mathbb{L} \)

\[
A(\text{Cons}([x, I])) = A(I)
\]

if \( ]_x \in \text{this}(I) \)

\[
A(\text{Cons}([x, I])) = A(I) \cup \{x\}
\]

if \( ]_x \notin \text{this}(I) \)

• Check consistency of final acquisitions

\[
fac(\text{Nil}) = \text{true} \quad fac(\text{Cons}(a, I)) = fac(I) \quad fac(\text{Spawn}(a, I_1, I_2)) = fac(I_1)
\]

• Compute acquisition graph

\[
G(\text{Nil}) = \emptyset
\]

\[
G(\text{Spawn}(a, I_1, I_2)) = G(I_1) \cup G(I_2)
\]

\[
G(\text{Cons}(a, I)) = G(I)
\]

if \( a \in \text{Act}_{nl} \) or \( a = ]_x \) for \( x \in \mathbb{L} \)

\[
G(\text{Cons}([x, I])) = G(I)
\]

if \( ]_x \in \text{this}(I) \)

\[
G(\text{Cons}([x, I])) = G(I) \cup \{x\} \times \text{acq}(I)
\]

if \( ]_x \notin \text{this}(I) \)

where \( \text{acq}(I) := \{x \mid ]_x \in I\} \)
Acquisition Graphs characterize Schedulability

- For scheduling tree $\langle bl \rangle (Nil) \in ST$ and labeling sequence $l \in Act^*$, we have

$$\exists t'. (\langle bl \rangle (Nil), \emptyset) \xrightarrow{l}^* (t', L) \wedge t' \in ST_{term} \iff \text{acyclic}(G(bl)) \wedge \text{fac}(bl)$$
Acquisition Graphs characterize Schedulability

- For scheduling tree $\langle bl \rangle (Nil) \in ST$ and labeling sequence $l \in Act^*$, we have

$$\exists t'. (\langle bl \rangle (Nil), \emptyset) \xrightarrow{l}^* (t', L) \land t' \in ST_{term} \iff \text{acyclic}(G(bl)) \land \text{fac}(bl)$$

- Proof Ideas:
Acquisition Graphs characterize Schedulability

- For scheduling tree \( \langle bl \rangle(\text{Nil}) \in ST \) and labeling sequence \( l \in \text{Act}^* \), we have

\[
\exists t'. (\langle bl \rangle(\text{Nil}), \emptyset) \xrightarrow{l^*} (t', L) \land t' \in ST_{\text{term}} \iff \text{acyclic}(G(bl)) \land \text{fac}(bl)
\]

- Proof Ideas:
  - \( \rightarrow \): Scheduling strategy: Schedule usages first. Final acquisitions in topological ordering of acquisition graph.
  - \( \leftarrow \): Formally: Generalize to initial set of locks disjoint from locks that occur in scheduling tree. Generalize to arbitrary scheduling tree. Induction on scheduling tree.
Acquisition Graphs characterize Schedulability

- For scheduling tree $\langle bl \rangle (Nil) \in ST$ and labeling sequence $l \in Act^*$, we have

$$\exists t'. (\langle bl \rangle (Nil), \emptyset) \xrightarrow{l}^* (t', L) \land t' \in ST_{term} \iff \text{acyclic}(G(bl)) \land fac(bl)$$

- Proof Ideas:
  - $\implies$
    - $G(t)$ 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 \( l \in Act^* \), we have

\[
\exists t'. (\langle bl \rangle (Nil), \emptyset) \xrightarrow{l}^* (t', L) \land t' \in ST_{term} \iff \text{acyclic}(G(bl)) \land \text{fac}(bl)
\]

- Proof Ideas:
  - \( \Rightarrow \)
    - \( G(t) \) 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 \( l \in Act^* \), we have

\[
\exists t'. (\langle bl \rangle (Nil), \emptyset) \xrightarrow{l}^* (t', L) \land t' \in ST_{term} \iff \text{acyclic}(G(bl)) \land \text{fac}(bl)
\]

- Proof Ideas:
  - \( \implies \)
    - \( G(t) \) 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.
  - \( \impliedby \)
Acquisition Graphs characterize Schedulability

- For scheduling tree $\langle bl \rangle (Nil) \in ST$ and labeling sequence $l \in Act^*$, we have

$$\exists t'. (\langle bl \rangle (Nil), \emptyset) \xrightarrow{l}^* (t', L) \land t' \in ST_{term} \iff \text{acyclic}(G(bl)) \land \text{fac}(bl)$$

- Proof Ideas:
  - $\implies$
    - $G(t)$ 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.
  - $\impliedby$
    - 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 $I \in Act^*$, we have

  $$\exists t'. (\langle bl \rangle (Nil), \emptyset) \xrightarrow{I}^* (t', L) \land t' \in ST_{term} \iff \text{acyclic}(G(bl)) \land \text{fac}(bl)$$

- Proof Ideas:
  - $\implies$
    - $G(t)$ 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.
  - $\impliedby$
    - 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:

• 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
Remark on complexity

- The lock-sensitive reachability problem is in NP:
  - For a sequential run, only polynomially many acquisition graphs/final acquisition sets occur

- 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
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
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
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

<table>
<thead>
<tr>
<th></th>
<th>DPN</th>
<th>PPDS</th>
<th>2PDS</th>
<th>DFN</th>
<th>PFSM</th>
<th>nFSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{EF}(p_1 \parallel p_2)$</td>
<td>NP*?</td>
<td>NP†?</td>
<td>NP†?</td>
<td>NP*!</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>$\text{EF}(A)$</td>
<td>NP</td>
<td>NP</td>
<td>NP†?</td>
<td>NP</td>
<td>NP</td>
<td>P</td>
</tr>
<tr>
<td>$\text{EF}(p_1 \parallel p_2 \land \text{EF}(p_3 \parallel p_4))$</td>
<td>NP</td>
<td>NP</td>
<td>NP</td>
<td>NP*!</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>$\text{EF}(A_1 \land \text{EF}(A_2))$</td>
<td>NP</td>
<td>NP</td>
<td>NP</td>
<td>NP</td>
<td>NP</td>
<td>P</td>
</tr>
<tr>
<td>$\text{EF}^{\neg}$ (fixed #ops)</td>
<td>NP</td>
<td>NP</td>
<td>NP</td>
<td>NP</td>
<td>NP</td>
<td>P</td>
</tr>
<tr>
<td>$\text{EF}$ (fixed #ops)</td>
<td>$\geq \text{PSPACE}^♭$</td>
<td>$\geq \text{NP}$</td>
<td>P</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{EF}^{\neg}$</td>
<td>$\geq \text{PSPACE}^♭$</td>
<td>$\geq \text{NP}^♭$</td>
<td>P</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{EF}$</td>
<td>$\geq \text{PSPACE}^♭$</td>
<td>$\geq \text{NP}^♭$</td>
<td>P</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Requires spawn inside lock

*! Polynomial algorithm if no spawn inside lock

*? Complexity unknown if no spawn inside lock

†? Hardness proof requires deadlocks/escapable locks. Complexity without this unknown.

‡ Hardness result requires no locks

reg? Hardness requires regular APs. Complexity for double-indexed APs unknown ($\geq \text{NP}$)
The End

Thank you for listening