# Algorithms for Tree Automata with Constraints

*Random Generation of Hard Instances of the Emptiness Problem for Tree Automata With Global Equality Constraints*

Pierre-Cyrille Héam, Vincent Hugot, Olga Kouchnarenko

{pcheam,vhugot,okouchnarenko}@lifc.univ-fcomte.fr

Université de Franche-Comté
LIFC-INRIA/CASSIS, project ACCESS

October 10, 2010

# Plan of the talk

1. **Introduction and motivation**
2. **(short) Preliminaries:**
   1. Vanilla Tree Automata
   2. Tree Automata with Constraints: TAGEDs
   3. The Emptiness problem
3. **Objectives and strategy**
4. **The random generation**
   1. **Cutting dead branches: the cleanup**
   2. **Initial random generation**
5. **Experimental results and conclusion.**

- **Tree automata**: powerful theoretical tools useful for
  - automated theorem proving
  - program verification
  - XML schema and query languages
  - ...
- **Extensions**: developed to expand expressiveness (*eg.* TAGEDs add global equality and disequality constraints.).
- **Drawback**: decidability and complexity of decision problems.

- **Long-term goal**: finding algorithms efficient enough for practical use. (for now, Emptiness for positive TAGEDs)
- **Problem**: without "real-world" testbeds, how to evaluate efficiency of our algorithms?
- **Solution**: random generation of TAGEDs.

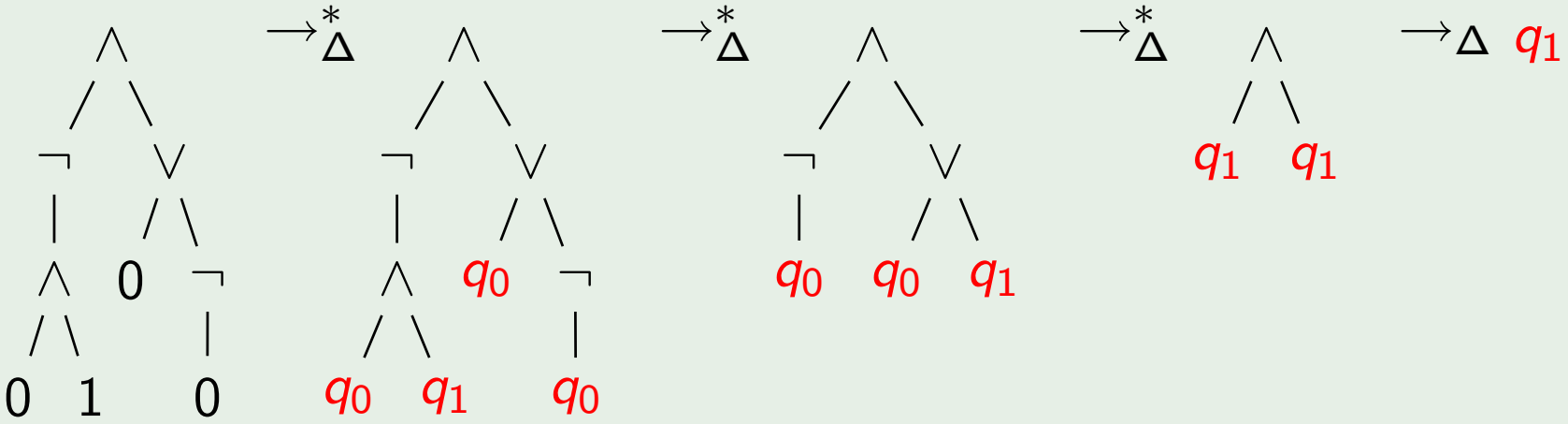
## Tree automaton for True propositional formulæ

$$\mathcal{A} \stackrel{\text{def}}{=} (\Sigma = \{\wedge, \vee/_2, \neg/_1, 0, 1/_0\},\ Q = \{q_0, q_1\}, F = \{q_1\}, \Delta)$$

$$\begin{aligned}\Delta = \{ & b \to q_b, \\ & \wedge(q_b, q_{b'}) \to q_{b \wedge b'}, \\ & \vee(q_b, q_{b'}) \to q_{b \vee b'}, \\ & \neg(q_b) \to q_{\neg b} \\ & \mid\ b, b' \in 0, 1\}\end{aligned}$$

# Bottom-Up Tree automata
## Definition through an example

$0 \to q_0, 1 \to q_1 \in \Delta \; \wedge(q_0, q_1) \to q_0, \neg(q_0) \to q_1 \in \Delta$
$\neg(q_0) \to q_1, \vee(q_0, q_1) \to q_1 \in \Delta \; \wedge(q_1, q_1) \to q_1 \in \Delta$

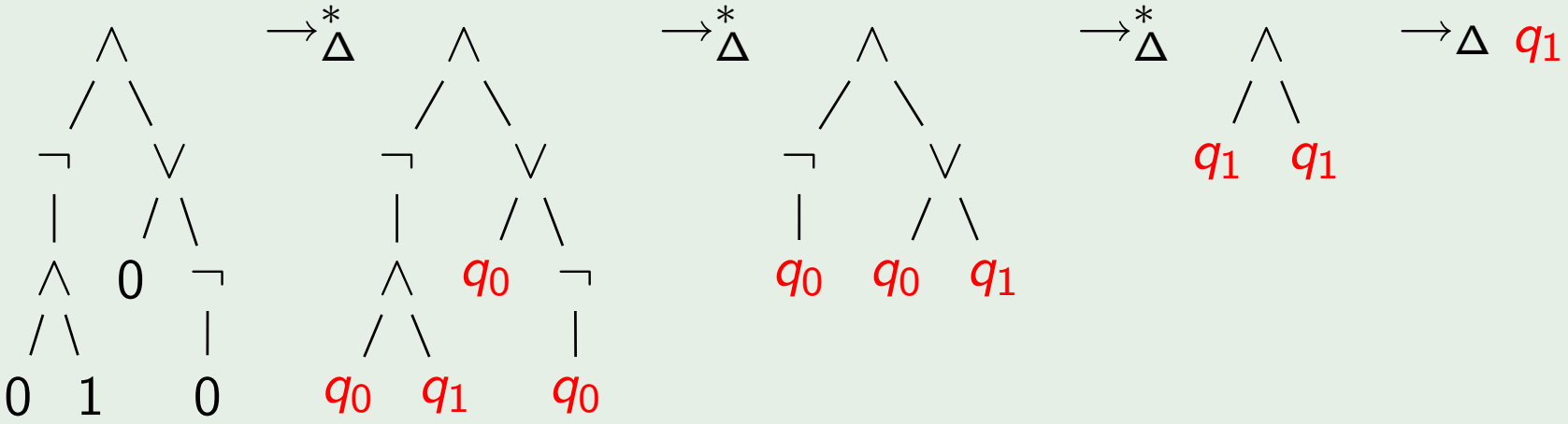



### Definition: run of $\mathcal{A}$ on a term $t \in \mathcal{T}(\Sigma)$

A run $\rho$ is a mapping from $\mathcal{P}os(t)$ to $Q$ compatible with the transition rules.

# Bottom-Up Tree automata

## Definition through an example

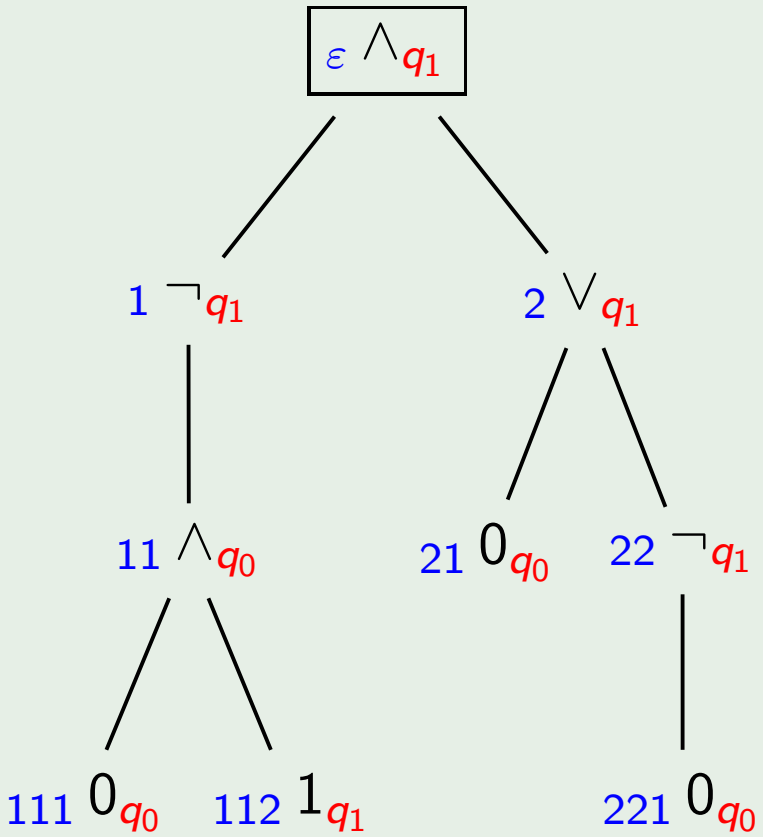



### Definition: run of $\mathcal{A}$ on a term $t \in \mathcal{T}(\Sigma)$

A run $\rho$ is a mapping from $\mathcal{P}os(t)$ to $Q$ compatible with the transition rules.

# Bottom-Up Tree automata
## Definition through an example

$\rho =$



Introduced in Emmanuel Filiot's PhD thesis on XML query languages. See [Filiot et al., 2008].

A TAGED is a tuple $\mathcal{A} = (\Sigma, Q, F, \Delta, =_{\mathcal{A}}, \neq_{\mathcal{A}})$, where

- $(\Sigma, Q, F, \Delta)$ is a tree automaton
- $=_{\mathcal{A}}$ is a reflexive symmetric binary relation on a subset of $Q$
- $\neq_{\mathcal{A}}$ is an irreflexive and symmetric binary relation on $Q$. Note that in our work, we have dealt with a slightly more general case, where $\neq_{\mathcal{A}}$ is not necessarily irreflexive.

A TAGED $\mathcal{A}$ is said to be *positive* if $\neq_{\mathcal{A}}$ is empty and *negative* if $=_{\mathcal{A}}$ is empty.

Runs must be compatible with equality and disequality constraints.

Introduced in Emmanuel Filiot's PhD thesis on XML query languages. See [Filiot et al., 2008].

A TAGED is a tuple $\mathcal{A} = (\Sigma, Q, F, \Delta, =_{\mathcal{A}}, \neq_{\mathcal{A}})$, where

- $(\Sigma, Q, F, \Delta)$ is a tree automaton
- $=_{\mathcal{A}}$ is a reflexive symmetric binary relation on a subset of $Q$
- $\neq_{\mathcal{A}}$ is an irreflexive and symmetric binary relation on $Q$. Note that in our work, we have dealt with a slightly more general case, where $\neq_{\mathcal{A}}$ is not necessarily irreflexive.

A TAGED $\mathcal{A}$ is said to be *positive* if $\neq_{\mathcal{A}}$ is empty and *negative* if $=_{\mathcal{A}}$ is empty.

Runs must be compatible with equality and disequality constraints.

Let $\rho$ be a run of the TAGED $\mathcal{A}$ on a tree $t$:

**Compatibility with the equality constraint $=_{\mathcal{A}}$**

$$\forall \alpha, \beta \in \mathcal{P}os(t) : \rho(\alpha) =_{\mathcal{A}} \rho(\beta) \implies t|_\alpha = t|_\beta .$$

**Compatibility with the disequality constraint $\neq_{\mathcal{A}}$ (irreflexive)**

$$\forall \alpha, \beta \in \mathcal{P}os(t) : \rho(\alpha) \neq_{\mathcal{A}} \rho(\beta) \implies t|_\alpha \neq t|_\beta .$$

**Compatibility with the disequality constraint $\neq_{\mathcal{A}}$ (non irreflexive)**

$$\forall \alpha, \beta \in \mathcal{P}os(t) : \alpha \neq \beta \wedge \rho(\alpha) \neq_{\mathcal{A}} \rho(\beta) \implies t|_\alpha \neq t|_\beta .$$

Let $\rho$ be a run of the TAGED $\mathcal{A}$ on a tree $t$:

**Compatibility with the equality constraint $=_{\mathcal{A}}$**

$$\forall \alpha, \beta \in \mathcal{P}os(t) : \rho(\alpha) =_{\mathcal{A}} \rho(\beta) \implies t|_\alpha = t|_\beta .$$

**Compatibility with the disequality constraint $\neq_{\mathcal{A}}$ (irreflexive)**

$$\forall \alpha, \beta \in \mathcal{P}os(t) : \rho(\alpha) \neq_{\mathcal{A}} \rho(\beta) \implies t|_\alpha \neq t|_\beta .$$

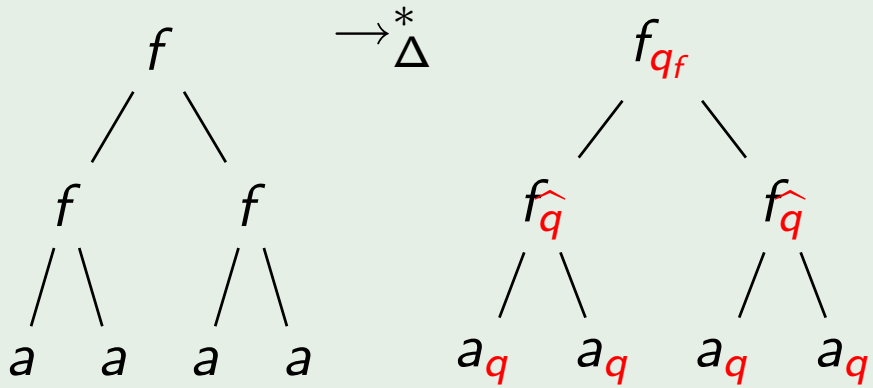
**Compatibility with the disequality constraint $\neq_{\mathcal{A}}$ (non irreflexive)**

$$\forall \alpha, \beta \in \mathcal{P}os(t) : \alpha \neq \beta \wedge \rho(\alpha) \neq_{\mathcal{A}} \rho(\beta) \implies t|_\alpha \neq t|_\beta .$$

## TAGED for $\{ f(t,t) \mid f \in \Sigma, t \in \mathcal{T}(\Sigma) \}$ [Filiot et al., 2008]

$$\mathcal{A} \stackrel{\text{def}}{=} (\Sigma = \{ a/_0, f/_2 \},\ Q = \{ q, \widehat{q}, q_f \},\ F = \{ q_f \},\ \Delta,\ \widehat{q} =_{\mathcal{A}} \widehat{q})$$

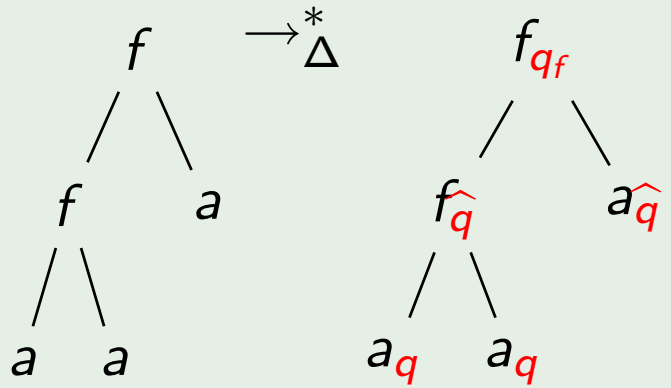
$$\text{where } \Delta \stackrel{\text{def}}{=} \{ f(\widehat{q},\widehat{q}) \to q_f,\ f(q,q) \to q,\ f(q,q) \to \widehat{q},\ a \to q,\ a \to \widehat{q}, \}$$



## TAGED for $\{ f(t,t) \mid f \in \Sigma, t \in \mathcal{T}(\Sigma) \}$ [Filiot et al., 2008]

$$\mathcal{A} \stackrel{\text{def}}{=} (\Sigma = \{ a/_0, f/_2 \},\ Q = \{ q, \widehat{q}, q_f \},\ F = \{ q_f \},\ \Delta,\ \widehat{q} =_{\mathcal{A}} \widehat{q})$$

$$\text{where } \Delta \stackrel{\text{def}}{=} \{ f(\widehat{q},\widehat{q}) \to q_f,\ f(q,q) \to q,\ f(q,q) \to \widehat{q},\ a \to q,\ a \to \widehat{q}, \}$$



# TAGED emptiness

## Emptiness Problem

**INPUT:** $\mathcal{A}$ a positive TAGED.
**OUTPUT:** $\mathcal{L}\mathrm{ng}(\mathcal{A}) = \varnothing$ ?

## Applications

- XML query languages
- model-checking, *eg.* cryptographic protocol verification, ...

## Theorem [Godoy et al., ]

The Emptiness Problem for TAGEDs is decidable.

## Theorem [Filiot et al., 2008]

The Emptiness Problem for *positive* TAGEDs is *EXPTIME*-complete.

# Objectives and Strategy

# Constraints of generation
## Fleshing out our objectives

### Long-term objective

Develop reasonably efficient approaches for deciding the Emptiness problem for positive TAGEDs

### Role of the random generation scheme

Experimental protocol to *discriminate* between *efficient* and *inefficient* approaches, as replacement of a real-world testbed.

The generated instances must be

- **Difficult:** Else we cannot discriminate between algorithms.
- **Realistic:** failing that, the results bear little relevance to expected practical performance.

# Constraints of generation
## Fleshing out our objectives

### Long-term objective

Develop reasonably efficient approaches for deciding the Emptiness problem for positive TAGEDs

### Role of the random generation scheme

Experimental protocol to *discriminate* between *efficient* and *inefficient* approaches, as replacement of a real-world testbed.

The generated instances must be

- **Difficult:** Else we cannot discriminate between algorithms.
- **Realistic:** failing that, the results bear little relevance to expected practical performance.

# What is “difficult” ?

**not difficult** =

- Deeply flawed generation scheme (*eg.* always empty)
- Often falls into special trivial case
  - *eg.* empty underlying vanilla TA
  - *eg.* diagonal positive TAGEDs [Filiot et al., 2008]
- Trivial by brute-force (*eg.* “leaf languages”)
- All final states in dead branches

# What is “difficult” ?

**not difficult** =

- Deeply flawed generation scheme (*eg.* always empty)
- Often falls into special trivial case
  - *eg.* empty underlying vanilla TA
  - *eg.* diagonal positive TAGEDs [Filiot et al., 2008]
- Trivial by brute-force (*eg.* “leaf languages”)
- All final states in dead branches

# What is “difficult” ?

**not difficult** =

- Deeply flawed generation scheme (*eg.* always empty)
- Often falls into special trivial case
  - *eg.* empty underlying vanilla TA
  - *eg.* diagonal positive TAGEDs [Filiot et al., 2008]
- Trivial by brute-force (*eg.* “leaf languages”)
- All final states in dead branches

# What is “difficult” ?

**not difficult** =

- Deeply flawed generation scheme (*eg.* always empty)
- Often falls into special trivial case
  - *eg.* empty underlying vanilla TA
  - *eg.* diagonal positive TAGEDs [Filiot et al., 2008]
- Trivial by brute-force (*eg.* “leaf languages”)
- All final states in dead branches

# What is “realistic” ?

**not realistic** =

- Enormous or tiny. . .
- *“soup blender”* or *“waffle iron”*
  - *eg.* languages almost entirely composed of “leaves”
  - *eg.* languages where all trees are isomorphic
- “Frankenstein” automaton
  - *eg.* unreachable states
  - *eg.* states that are never used
  - *eg.* rules that immediately violate the constraints
  - everything which we will call “dead branches” in general.

# What is “realistic” ?

**not realistic** =

- Enormous or tiny...
- *“soup blender”* or *“waffle iron”*
  - *eg.* languages almost entirely composed of “leaves”
  - *eg.* languages where all trees are isomorphic
- “Frankenstein” automaton
  - *eg.* unreachable states
  - *eg.* states that are never used
  - *eg.* rules that immediately violate the constraints
  - everything which we will call “dead branches” in general.

# What is “realistic” ?

**not realistic** =

- Enormous or tiny...
- *“soup blender”* or *“waffle iron”*
  - *eg.* languages almost entirely composed of “leaves”
  - *eg.* languages where all trees are isomorphic
- “Frankenstein” automaton
  - *eg.* unreachable states
  - *eg.* states that are never used
  - *eg.* rules that immediately violate the constraints
  - everything which we will call “dead branches” in general.

# Our generation strategy
## . . . and plan of the next two sections

### Generation mechanism

1. Generate a raw TAGED $\mathcal{A}$, as "interesting" as possible.
2. Detect whether $\mathcal{A}$ is clearly easy. Throw it away if it is.
3. Remove dead branches from $\mathcal{A}$.
4. $\mathcal{A}$ is good, ship it!

### Detect easy cases, remove dead branches

Done at the same time. We call this the **cleanup**. ↬ *next section*.

### Generate "quite" interesting TAGEDs

Generating rules with the desired **structure** of the automaton and its accepted language as guide. ↬ *next*$^2$ *section*.

# Our generation strategy
## . . . and plan of the next two sections

### Generation mechanism

1. Generate a raw TAGED $\mathcal{A}$, as "interesting" as possible.
2. Detect whether $\mathcal{A}$ is clearly easy. Throw it away if it is.
3. Remove dead branches from $\mathcal{A}$.
4. $\mathcal{A}$ is good, ship it!

### Detect easy cases, remove dead branches

Done at the same time. We call this the **cleanup**. ↬ *next section*.

### Generate "quite" interesting TAGEDs

Generating rules with the desired **structure** of the automaton and its accepted language as guide. ↬ *next*$^2$ *section*.

# Our generation strategy
## . . . and plan of the next two sections

### Generation mechanism

1. Generate a raw TAGED $\mathcal{A}$, as "interesting" as possible.
2. Detect whether $\mathcal{A}$ is clearly easy. Throw it away if it is.
3. Remove dead branches from $\mathcal{A}$.
4. $\mathcal{A}$ is good, ship it!

### Detect easy cases, remove dead branches

Done at the same time. We call this the **cleanup**. ↬ *next section*.

### Generate "quite" interesting TAGEDs

Generating rules with the desired **structure** of the automaton and its accepted language as guide. ↬ *next*$^2$ *section*.

# Our generation strategy
## . . . and plan of the next two sections

### Generation mechanism

1. Generate a raw TAGED $\mathcal{A}$, as "interesting" as possible.
2. Detect whether $\mathcal{A}$ is clearly easy. Throw it away if it is.
3. Remove dead branches from $\mathcal{A}$.
4. $\mathcal{A}$ is good, ship it!

### Detect easy cases, remove dead branches

Done at the same time. We call this the **cleanup**. ↬ *next section*.

### Generate "quite" interesting TAGEDs

Generating rules with the desired **structure** of the automaton and its accepted language as guide. ↬ *next*$^2$ *section*.

# Our generation strategy
## . . . and plan of the next two sections

### Generation mechanism

1. Generate a raw TAGED $\mathcal{A}$, as "interesting" as possible.
2. **Detect whether $\mathcal{A}$ is clearly easy. Throw it away if it is.**
3. **Remove dead branches from $\mathcal{A}$.**
4. $\mathcal{A}$ is good, ship it!

### Detect easy cases, remove dead branches

Done at the same time. We call this the **cleanup**. $\looparrowright$ *next section*.

### Generate "quite" interesting TAGEDs

Generating rules with the desired **structure** of the automaton and its accepted language as guide. $\looparrowright$ *next*$^2$ *section*.

# Our generation strategy
## . . . and plan of the next two sections

### Generation mechanism

1. **Generate a raw TAGED $\mathcal{A}$, as "interesting" as possible.**
2. Detect whether $\mathcal{A}$ is clearly easy. Throw it away if it is.
3. Remove dead branches from $\mathcal{A}$.
4. $\mathcal{A}$ is good, ship it!

### Detect easy cases, remove dead branches

Done at the same time. We call this the **cleanup**. $\looparrowright$ *next section*.

### Generate "quite" interesting TAGEDs

Generating rules with the desired **structure** of the automaton and its accepted language as guide. $\looparrowright$ *next*$^2$ *section*.

# Cleanup

Improved version of standard reduction (reachability) algorithm for TAs. Takes advantage of equality constraints to remove useless rules and states.
***ie. remove dead branches***.

***Not enough time: in annex***

# Raw TAGED Generation

## Rough outline of random generation of TA

1. Build a *pool* of head states from skeleton-driven generation. Keep track of minimum accepted height.
2. Store the rules in $\Delta$.
3. while *requested minimum height* not reached, do
   1. purge too old states from pool
   2. let $q$ be a fresh state
   3. let $\delta$ be a random number (of rules), then do $\delta$ times
      1. let $n$ be a random number (arity)
      2. let $\sigma$ be a random symbol of $\Sigma_n$
      3. let $p_1, \ldots, p_n$ be random states from pool
      4. add rule $\sigma(p_1, \ldots, p_n) \longrightarrow q$ to $\Delta$
      5. add $q$ to pool
4. $F =$ some random final states from pool

1. Build a *pool* of head states from skeleton-driven generation. **Keep track of minimum accepted height**.
2. Store the rules in $\Delta$.
3. while *requested minimum height* not reached, do
   1. purge too old states from pool
   2. let $q$ be a fresh state
   3. let $\delta$ be a random number (of rules), then do $\delta$ times
      1. let $n$ be a random number (arity)
      2. let $\sigma$ be a random symbol of $\Sigma_n$
      3. let $p_1, \ldots, p_n$ be random states from pool
      4. add rule $\sigma(p_1, \ldots, p_n) \to q$ to $\Delta$
      5. add $q$ to pool
4. $F =$ some random final states from pool

$q$ in pool: $m(q)$ is "height of the smallest term $t \in \mathcal{L}\mathrm{ng}(\mathcal{A}, q)$"

1. Build a *pool* of head states from skeleton-driven generation. Keep track of minimum accepted height.
2. **Store the rules in $\Delta$.**
3. while *requested minimum height* not reached, do
    1. purge too old states from pool
    2. let $q$ be a fresh state
    3. let $\delta$ be a random number (of rules), then do $\delta$ times
        1. let $n$ be a random number (arity)
        2. let $\sigma$ be a random symbol of $\Sigma_n$
        3. let $p_1, \ldots, p_n$ be random states from pool
        4. add rule $\sigma(p_1, \ldots, p_n) \to q$ to $\Delta$
        5. add $q$ to pool
4. $F =$ some random final states from pool

Initial (skeleton generation) rules. Other rules will be added later.

1. Build a *pool* of head states from skeleton-driven generation. Keep track of minimum accepted height.
2. Store the rules in $\Delta$.
3. while *requested minimum height* not reached, do
    1. **purge too old states from pool**
    2. let $q$ be a fresh state
    3. let $\delta$ be a random number (of rules), then do $\delta$ times
        1. let $n$ be a random number (arity)
        2. let $\sigma$ be a random symbol of $\Sigma_n$
        3. let $p_1, \ldots, p_n$ be random states from pool
        4. add rule $\sigma(p_1, \ldots, p_n) \to q$ to $\Delta$
        5. add $q$ to pool
4. $F =$ some random final states from pool

Here $q$ is "too old" if $m(q)$ is too small compared to

$$\max_{p \in \text{pool}} m(p).$$

1. Build a *pool* of head states from skeleton-driven generation. Keep track of minimum accepted height.
2. Store the rules in $\Delta$.
3. while *requested minimum height* not reached, do
   1. purge too old states from pool
   2. let $q$ be a fresh state
   3. **let $\delta$ be a random number** (of rules), then do $\delta$ times
      1. **let $n$ be a random number** (arity)
      2. let $\sigma$ be a random symbol of $\Sigma_n$
      3. let $p_1, \ldots, p_n$ be random states from pool
      4. add rule $\sigma(p_1, \ldots, p_n) \to q$ to $\Delta$
      5. add $q$ to pool
4. $F =$ some random final states from pool

Selected according to discrete probability distributions.

1. Build a *pool* of head states from skeleton-driven generation. Keep track of minimum accepted height.
2. Store the rules in $\Delta$.
3. while *requested minimum height* not reached, do
   1. purge too old states from pool
   2. let $q$ be a fresh state
   3. let $\delta$ be a random number (of rules), then do $\delta$ times
      1. let $n$ be a random number (arity)
      2. **let $\sigma$ be a random symbol of $\Sigma_n$**
      3. let $p_1, \ldots, p_n$ be random states from pool
      4. add rule $\sigma(p_1, \ldots, p_n) \longrightarrow q$ to $\Delta$
      5. add $q$ to pool
4. $F =$ some random final states from pool

Random symbols in $\Sigma_n$ are selected uniformly.

1. Build a *pool* of head states from skeleton-driven generation. Keep track of minimum accepted height.
2. Store the rules in $\Delta$.
3. while *requested minimum height* not reached, do
   1. purge too old states from pool
   2. let $q$ be a fresh state
   3. let $\delta$ be a random number (of rules), then do $\delta$ times
      1. let $n$ be a random number (arity)
      2. let $\sigma$ be a random symbol of $\Sigma_n$
      3. **let $p_1, \ldots, p_n$ be random states from pool**
      4. add rule $\sigma(p_1, \ldots, p_n) \to q$ to $\Delta$
      5. add $q$ to pool
4. $F =$ some random final states from pool

DPD biased towards states with higher min height.

1. Build a *pool* of head states from skeleton-driven generation. Keep track of minimum accepted height.
2. Store the rules in $\Delta$.
3. while *requested minimum height* not reached, do
    1. purge too old states from pool
    2. let $q$ be a fresh state
    3. let $\delta$ be a random number (of rules), then do $\delta$ times
        1. let $n$ be a random number (arity)
        2. let $\sigma$ be a random symbol of $\Sigma_n$
        3. let $p_1, \ldots, p_n$ be random states from pool
        4. add rule $\sigma(p_1, \ldots, p_n) \to q$ to $\Delta$
        5. **add $q$ to pool**
4. $F =$ some random final states from pool

The first time, $q \notin$ pool: reachable. Afterwards, just update $m(q)$.

1. Build a *pool* of head states from skeleton-driven generation. Keep track of minimum accepted height.
2. Store the rules in $\Delta$.
3. while *requested minimum height* not reached, do
   1. purge too old states from pool
   2. let $q$ be a fresh state
   3. let $\delta$ be a random number (of rules), then do $\delta$ times
      1. let $n$ be a random number (arity)
      2. let $\sigma$ be a random symbol of $\Sigma_n$
      3. let $p_1, \ldots, p_n$ be random states from pool
      4. add rule $\sigma(p_1, \ldots, p_n) \longrightarrow q$ to $\Delta$
      5. add $q$ to pool
4. **$F$ = some random final states from pool**

DPD, strongly biased towards higher min heights.

- Number of constraints $p =_{\mathcal{A}} q$ logarithmic in $|Q|$.
- Bias towards diagonal constraints.

| Height | $\|Q\|$ | $\|\|A\|\|$ | $\|\|A\|\| / \|Q\|$ | $\|\Delta\|$ | $\|\Delta\| / \|Q\|$ |
|---|---|---|---|---|---|
| 4 | 6.89 | 43.49 | 6.31 | 11.30 | 1.64 |
| 10 | 18.14 | 119.84 | 6.61 | 27.12 | 1.50 |
| 16 | 29.58 | 196.94 | 6.66 | 43.13 | 1.46 |
| 22 | 41.31 | 276.70 | 6.70 | 59.67 | 1.44 |
| 28 | 52.58 | 353.26 | 6.72 | 75.47 | 1.44 |
| 34 | 64.47 | 434.65 | 6.74 | 92.36 | 1.43 |
| 40 | 75.38 | 507.81 | 6.74 | 107.55 | 1.43 |
| 46 | 87.00 | 588.54 | 6.76 | 124.14 | 1.43 |
| 52 | 99.45 | 672.86 | 6.77 | 141.87 | 1.43 |
| 58 | 110.41 | 745.74 | 6.75 | 156.70 | 1.42 |
| 64 | 122.41 | 826.10 | 6.75 | 173.27 | 1.42 |
| 70 | 133.68 | 903.50 | 6.76 | 189.26 | 1.42 |
| 76 | 145.09 | 981.29 | 6.76 | 205.39 | 1.42 |

Table: Generation 4: size statistics

| $\lvert Q \rvert$ | Run $\rho$ | $\mathcal{L}\mathrm{ng}(A) \neq \varnothing$ | $\mathcal{L}\mathrm{ng}(A) = \varnothing$ | Failure |
|---|---|---|---|---|
| 4. | 26.8% | 73.2% | 0.0% | 0.0% |
| 7. | 43.6% | 55.6% | 0.8% | 0.0% |
| 10. | 48.8% | 50.8% | 0.4% | 0.0% |
| 13. | 49.2% | 50.8% | 0.0% | 0.0% |
| 16. | 50.0% | 50.0% | 0.0% | 0.0% |
| 19. | 42.4% | 57.6% | 0.0% | 0.0% |
| 22. | 41.2% | 58.4% | 0.4% | 0.0% |
| 25. | 34.8% | 65.2% | 0.0% | 0.0% |
| 28. | 30.4% | 69.6% | 0.0% | 0.0% |
| 31. | 36.4% | 63.6% | 0.0% | 0.0% |
| 34. | 38.8% | 61.2% | 0.0% | 0.0% |
| 37. | 35.6% | 64.4% | 0.0% | 0.0% |
| 40. | 28.0% | 72.0% | 0.0% | 0.0% |

Table: "Soup blender" typical results

| min H | Run $\rho$ | $A \neq \varnothing$ | $A = \varnothing$ | Failure | $\prec$ |
|---|---|---|---|---|---|
| 6 | 0.4% | 69.6% | 28.8% | 1.2% | 2.8% |
| 9 | 0.4% | 69.2% | 25.6% | 4.8% | 6.4% |
| 12 | 0.0% | 55.6% | 36.4% | 8.0% | 9.2% |
| 15 | 0.0% | 61.2% | 26.4% | 12.4% | 7.6% |
| 18 | 0.0% | 53.2% | 30.0% | 16.8% | 6.4% |
| 21 | 0.0% | 50.8% | 30.0% | 19.2% | 8.8% |
| 24 | 0.0% | 46.8% | 35.6% | 17.6% | 7.2% |
| 27 | 0.0% | 49.2% | 28.8% | 22.0% | 8.8% |
| | | | | | |
| 27 | 0.0% | 45.6% | 31.2% | 23.2% | 5.6% |
| 30 | 0.0% | 45.2% | 31.2% | 23.6% | 6.8% |
| 31 | 0.0% | 50.8% | 25.2% | 24.0% | 6.0% |
| 34 | 0.0% | 50.8% | 26.8% | 22.4% | 6.4% |
| 37 | 0.0% | 43.6% | 26.8% | 29.6% | 7.2% |

Table: Latest generation: results

# Conclusion

- This scheme avoids the experimental pitfalls of previous attempts.
  - Structured language
  - Coherent automaton
  - Sane size and density
- A better experimental protocol than hand-written automata
- Many parameters can be modelled on statistics for more realism
- Made for the Emptiness problem, but useful for other problems *eg.* Membership (with a term generation scheme)
- Forthcoming research report, more exhaustive than the slides.

# Some references

[Comon et al., 2007, Filiot et al., 2008, Tabakov and Vardi, 2005, Godoy et al., ]


Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., and Tommasi, M. (2007).

*Tree Automata Techniques and Applications*.

release October, 12th 2007.

Filiot, E., Talbot, J.-M., and Tison, S. (2008).

Tree Automata with Global Constraints.

In *12th International Conference on Developments in Language Theory (DLT)*, pages 314–326, Kyoto Japon.

Godoy, L., Jacquemard, F., and Vacher, C.

The Emptiness Problem for Tree Automata with Global Constraints.

Tabakov, D. and Vardi, M. (2005).

Experimental evaluation of classical automata constructions.

In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 396–411. Springer.

# Related generation schemes

## A successful scheme for NFAs [Tabakov and Vardi, 2005]

To generate a NFA $(\Sigma, Q, Q_0, F, \delta)$, fix $|Q|$, and $\Sigma = \{0, 1\}$, generate transitions and final states according to ratios:

$$r = r_\sigma = \frac{|\{(p, \sigma, q) \in \delta\}|}{|Q|}, \forall \sigma \in \Sigma \quad \text{and} \quad f = \frac{|F|}{|Q|}.$$

# Related generation schemes

## A successful scheme for NFAs [Tabakov and Vardi, 2005]

To generate a NFA $(\Sigma, Q, Q_0, F, \delta)$, fix $|Q|$, and $\Sigma = \{0, 1\}$, generate transitions and final states according to ratios:

$$r = r_\sigma = \frac{|\{(p, \sigma, q) \in \delta\}|}{|Q|}, \forall \sigma \in \Sigma \quad \text{and} \quad f = \frac{|F|}{|Q|}.$$

- Successful scheme for word automata
- . . . adaptation to Tree Automata?

# Related generation schemes

## A successful scheme for NFAs [Tabakov and Vardi, 2005]

To generate a NFA $(\Sigma, Q, Q_0, F, \delta)$, fix $|Q|$, and $\Sigma = \{0, 1\}$, generate transitions and final states according to ratios:

$$r = r_\sigma = \frac{|\{(p, \sigma, q) \in \delta\}|}{|Q|}, \forall \sigma \in \Sigma \quad \text{and} \quad f = \frac{|F|}{|Q|}.$$

## An adaptation to NTAs

To generate a NTA $(\Sigma, Q, F, \Delta)$, fix $|Q|$ and $\Sigma$, generate rules according to ratios:

$$r = \frac{|\Delta|}{|\{f(q_1, \ldots, q_n) \mid f(q_1, \ldots, q_n) \to q \in \Delta\}|} \quad \text{and} \quad f = \frac{|F|}{|Q|}.$$

# Related generation schemes

## An adaptation to NTAs

To generate a NTA $(\Sigma, Q, F, \Delta)$, fix $|Q|$ and $\Sigma$, generate rules according to ratios:

$$r = \frac{|\Delta|}{|\{\, f(q_1, \ldots, q_n) \mid f(q_1, \ldots, q_n) \to q \in \Delta \,\}|} \quad \text{and} \quad f = \frac{|F|}{|Q|}.$$

- Used for Universality
- Experimental protocol not fully explained

## Dense generation

Fix alphabet $\Sigma = \{ a, b, c/_0, f, g, h/_2 \}$, $|Q|$, and probas $p_\Delta$and $p_F$. Build

$$\Delta \subseteq \overline{\Delta} \qquad \text{where} \qquad \overline{\Delta} \stackrel{\text{def}}{=} \biguplus_{k \in \mathbb{N}} \Sigma_k \times Q^{k+1},$$

by choosing each rule in $\overline{\Delta}$ with proba $p_\Delta$. Build $F \subseteq Q$ by choosing each state with proba $p_F$.

- Generates automata that are very dense. *Real-world automata* are mostly sparse.
- Rules for symbols of high arity are overly represented. *eg.* try with symbol $\sigma \in \Sigma_{10}$
- *soup blender*: "leaf language", mostly dead branches. *ie.* cleanup kills everything.

## Dense generation

Fix alphabet $\Sigma = \{ a, b, c/_0, f, g, h/_2 \}$, $|Q|$, and probas $p_\Delta$ and $p_F$. Build

$$\Delta \subseteq \overline{\Delta} \qquad \text{where} \qquad \overline{\Delta} \stackrel{\text{def}}{=} \biguplus_{k \in \mathbb{N}} \Sigma_k \times Q^{k+1},$$

by choosing each rule in $\overline{\Delta}$ with proba $p_\Delta$. Build $F \subseteq Q$ by choosing each state with proba $p_F$.

- Generates automata that are very dense. *Real-world automata* are mostly sparse.
- Rules for symbols of high arity are overly represented. *eg.* try with symbol $\sigma \in \Sigma_{10}$
- *soup blender*: "leaf language", mostly dead branches. *ie.* cleanup kills everything.

## Sparse generation

As in dense generation, but fix *expected in-degree* $\delta$,

$$\forall k \in \mathbb{N}, \quad p_\Delta(k) = \begin{cases} \dfrac{\delta}{|\mathfrak{Arr}_\Sigma| \cdot |\Sigma_k| \cdot |Q|^k} & \text{if } \Sigma_k \neq \varnothing \\ 0 & \text{if } \Sigma_k = \varnothing \end{cases}.$$

- More sparse automata: avg. $|\Delta| = \delta\,|Q|$
- No high arity explosion
- . . . but still lots of dead branches (cleanup ratio $1/30$)
- . . . and still "leaf language".

## Sparse generation

As in dense generation, but fix *expected in-degree* $\delta$,

$$\forall k \in \mathbb{N}, \quad p_\Delta(k) = \begin{cases} \dfrac{\delta}{|\mathfrak{Ar}_\Sigma| \cdot |\Sigma_k| \cdot |Q|^k} & \text{if } \Sigma_k \neq \varnothing \\ 0 & \text{if } \Sigma_k = \varnothing \end{cases}.$$

- More sparse automata: avg. $|\Delta| = \delta\,|Q|$
- No high arity explosion
- . . . but still lots of dead branches (cleanup ratio $1/30$)
- . . . and still "leaf language".

- More sparse automata: avg. $|\Delta| = \delta\,|Q|$
- No high arity explosion
- . . . but still lots of dead branches (cleanup ratio $1/30$)
- . . . **and still "leaf language".**

## Probability of final leaf

$$P = 1 - (1 - p_F)^L = 1 - (1 - p_F)^{\frac{\delta|Q|}{|\mathfrak{Ar}_\Sigma|}} \cong 1 - \left(\frac{4}{5}\right)^{|Q|}.$$

| $P$ | 0.5 | 0.75 | 0.9 | 0.99 | 0.999 |
|---|---|---|---|---|---|
| $\lvert Q\rvert$ | 3 | 6 | 10 | 20 | 30 |

# Schemes which did not work well

## Sparse generation

- More sparse automata: avg. $|\Delta| = \delta\,|Q|$
- No high arity explosion
- . . . but still lots of dead branches (cleanup ratio $1/30$)
- . . . and still "leaf language".

### Probability of final leaf

$$P = 1 - (1 - p_F)^L = 1 - (1 - p_F)^{\frac{\delta|Q|}{|\mathfrak{Ar}_\Sigma|}} \cong 1 - \left(\frac{4}{5}\right)^{|Q|}.$$

| $P$ | 0.5 | 0.75 | 0.9 | 0.99 | 0.999 |
|---|---|---|---|---|---|
| $\lvert Q\rvert$ | 3 | 6 | 10 | 20 | 30 |

**This is a pervasive problem with unstructured generation!**

## Lessons learned from previous attempts

- We want sparse automata: keep number of rules small
- Avoid high arity rules explosion
- Avoid "leaf languages": too easy for brute force.
  $\Longrightarrow$ reason in terms of the *minimal height of accepted terms*

## Preliminary Idea

Fix alphabet to say, $\Sigma^5$ with
$\Sigma^n \stackrel{\text{def}}{=} \{ a_1, \ldots, a_n/_0, f_1, \ldots, f_n/_1, g_1, \ldots, g_n/_2, h_1, \ldots, h_n/_3 \}$.

1. Generate skeletons $s_1, \ldots, s_n$, within constraints of height and width and arity $\leqslant 3$.
2. Then generate rules sets $\Delta_1, \ldots, \Delta_n$ to accept terms isomorphic to these skeletons.
3. Topmost states $q_k$ in each $\Delta_k$ = final states

## Lessons learned from previous attempts

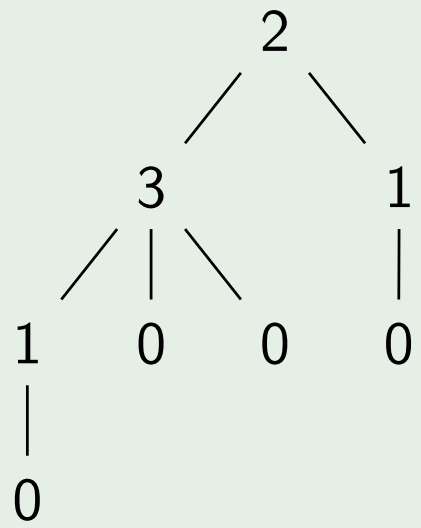
- We want sparse automata: keep number of rules small
- Avoid high arity rules explosion
- Avoid "leaf languages": too easy for brute force.
  $\Longrightarrow$ reason in terms of the *minimal height of accepted terms*

## Preliminary Idea

Fix alphabet to say, $\Sigma^5$ with
$\Sigma^n \stackrel{\text{def}}{=} \{ a_1, \ldots, a_n/_0, f_1, \ldots, f_n/_1, g_1, \ldots, g_n/_2, h_1, \ldots, h_n/_3 \}$.

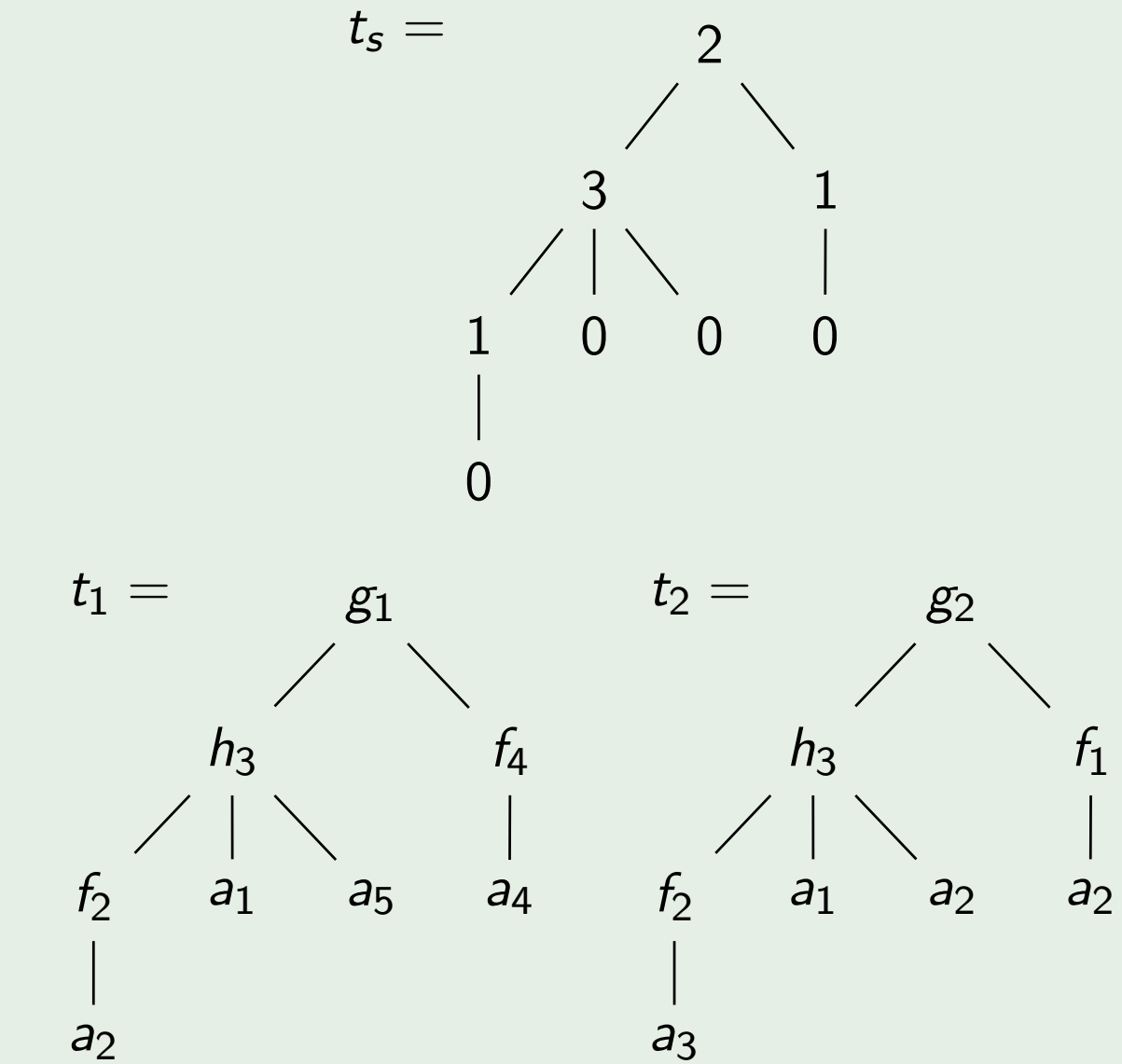
1. Generate skeletons $s_1, \ldots, s_n$, within constraints of height and width and arity $\leqslant 3$.
2. Then generate rules sets $\Delta_1, \ldots, \Delta_n$ to accept terms isomorphic to these skeletons.
3. Topmost states $q_k$ in each $\Delta_k$ = final states

$t_s =$



# Towards a scheme that works better

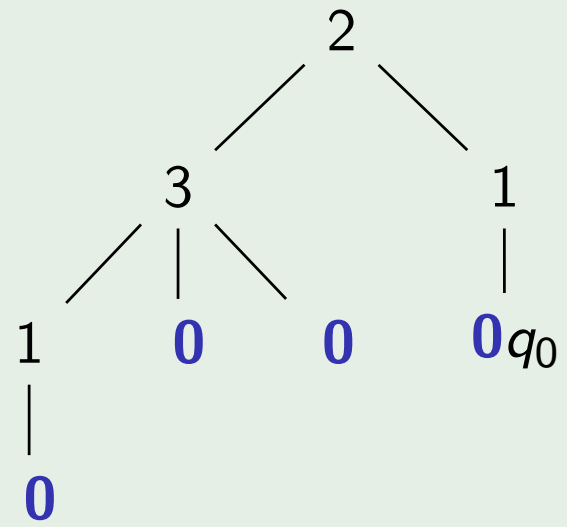
## Skeleton-driven generation



# Towards a scheme that works better

## Skeleton-driven generation

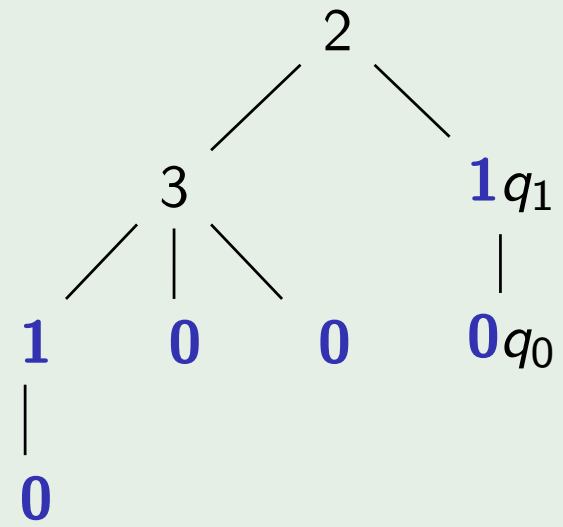
$t_s =$




**Generated rules examples (the real algorithm is recursive from top)**

new state $q_0$, $a_2 \longrightarrow q_0, a_5 \longrightarrow q_0 \in \Delta$, etc

$t_s =$

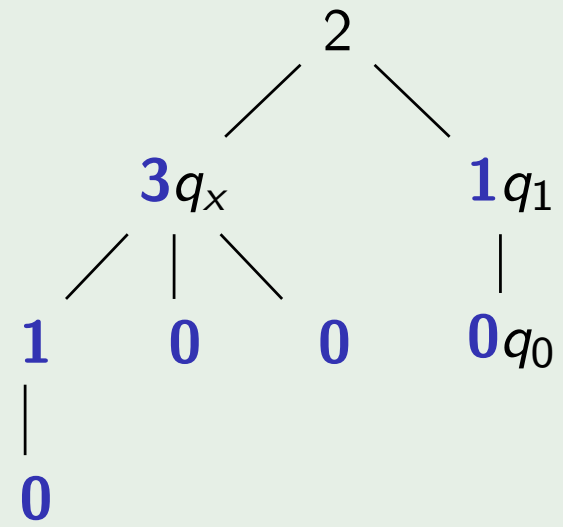



## Generated rules examples (the real algorithm is recursive from top)

new state $q_0$, $a_2 \longrightarrow q_0, a_5 \longrightarrow q_0 \in \Delta$, etc

new state $q_1$, $f_3(q_0) \longrightarrow q_1, f_2(q_0) \longrightarrow q_1, f_5(q_0) \longrightarrow q_1 \in \Delta$, etc

$t_s =$



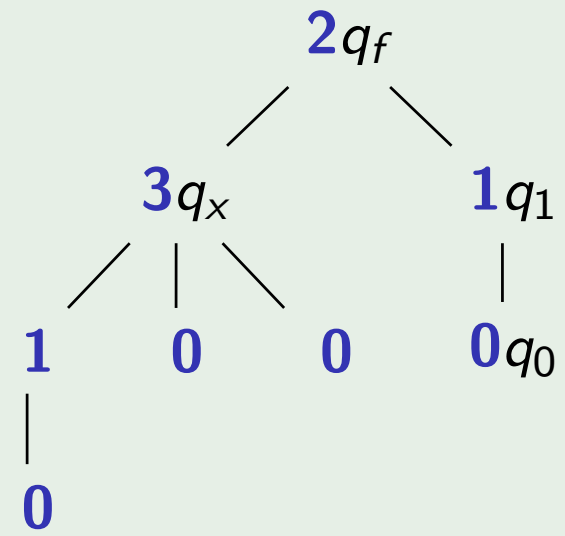

## Generated rules examples (the real algorithm is recursive from top)

new state $q_0$, $a_2 \longrightarrow q_0, a_5 \longrightarrow q_0 \in \Delta$, etc

new state $q_1$, $f_3(q_0) \longrightarrow q_1, f_2(q_0) \longrightarrow q_1, f_5(q_0) \longrightarrow q_1 \in \Delta$, etc

new state $q_x$, obtained after a few steps

$t_s =$




## Generated rules examples (the real algorithm is recursive from top)

new state $q_0$, $a_2 \longrightarrow q_0, a_5 \longrightarrow q_0 \in \Delta$, etc
new state $q_1$, $f_3(q_0) \longrightarrow q_1, f_2(q_0) \longrightarrow q_1, f_5(q_0) \longrightarrow q_1 \in \Delta$, etc
new state $q_x$, obtained after a few steps
new final state $q_f$, $g_1(q_x, q_1) \longrightarrow q_f \in \Delta$

## Getting $(\Delta_k, q_k)$ from $s_k$ (OCaml code)

```
let conversion δ skel =
  let Δ = ref Δ.∅ in
  let make_rules ar [q1,...,qn] q m = for k = 1 to m do
    let σ = gene_symbol ar in Δ.↩ (σ,[q1,...,qn],q) Δ
  done in let rec f = λ
    | Leaf 0 →
      let qx = fresh_state() in make_rules 0 ∅ qx δ; return qx
    | Node (ar, subs) →
      let qx = fresh_state() and [q1,...,qn] = ℒ.map f subs in
      make_rules ar [q1,...,qn] qx δ; return qx
  in let head = f skel in (!Δ, head)
```

## Getting a TA from $(\Delta_k, q_k)$

We have $\Sigma$ fixed, just extract all states from all $\Delta_k$ to $Q$, $F = \{ q_k \mid k = 1..n \}$, $\Delta = \cup_k \Delta_k$.

## Getting $(\Delta_k, q_k)$ from $s_k$ (OCaml code)

```
let conversion δ skel =
  let Δ = ref Δ.∅ in
  let make_rules ar [q1,...,qn] q m = for k = 1 to m do
    let σ = gene_symbol ar in Δ.↩ (σ,[q1,...,qn],q) Δ
  done in let rec f = λ
    | Leaf 0 →
      let qx = fresh_state() in make_rules 0 ∅ qx δ; return qx
    | Node (ar, subs) →
      let qx = fresh_state() and [q1,...,qn] = ℒ.map f subs in
      make_rules ar [q1,...,qn] qx δ; return qx
  in let head = f skel in (!Δ, head)
```

## Getting a TA from $(\Delta_k, q_k)$

We have $\Sigma$ fixed, just extract all states from all $\Delta_k$ to $Q$, $F = \{ q_k \mid k = 1..n \}$, $\Delta = \cup_k \Delta_k$.

# Towards a scheme that works better

## Skeleton-driven generation

- Guaranteed minimal height (difficulty?)
- No dead branches for TA

- The automata are sparse, but the number of states explodes with the height.
- *waffle iron*: all accepted terms are isomorphic to one of $n$ trees ($n$ small). This by construction. *Compromises difficulty!*
- Many kinds of transition rules are not represented
  - rules with immediate cycles *eg.* $f(\dots, q, \dots) \to q$
  - repetitions of the same state *eg.* $f(\dots, p, \dots, p, \dots) \to q$
  - reusing old states *eg.* $f(\dots, p, \dots) \to q$, with $p$ not *fresh*
  - for any $q \in Q$, all rules in $\mathfrak{Rul}(q)$ share the same signature!

# Towards a scheme that works better

## Skeleton-driven generation

- Guaranteed minimal height (difficulty?)
- No dead branches for TA

- The automata are sparse, but the number of states explodes with the height.
- *waffle iron*: all accepted terms are isomorphic to one of $n$ trees ($n$ small). This by construction. *Compromises difficulty!*
- Many kinds of transition rules are not represented
  - rules with immediate cycles *eg.* $f(\dots, q, \dots) \to q$
  - repetitions of the same state *eg.* $f(\dots, p, \dots, p, \dots) \to q$
  - reusing old states *eg.* $f(\dots, p, \dots) \to q$, with $p$ not *fresh*
  - for any $q \in Q$, all rules in $\mathfrak{Rul}(q)$ share the same signature!

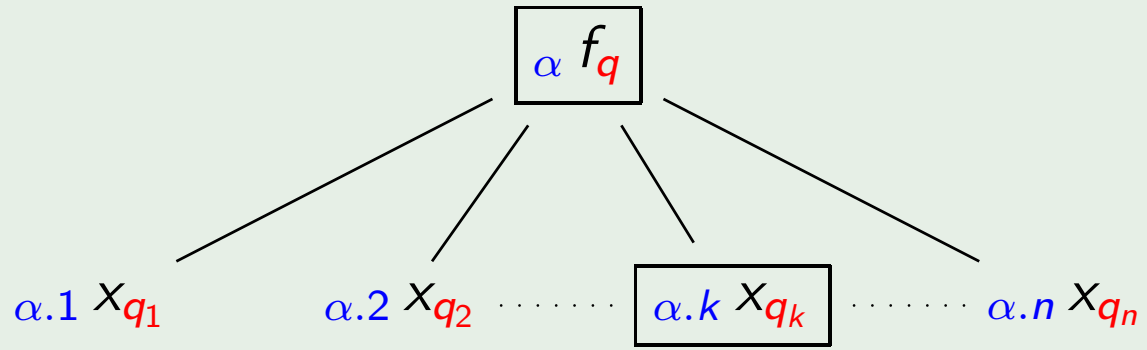
# Cleanup

Improved version of standard reduction (reachability) algorithm for tree automata, which takes advantage of equality constraints to remove useless rules and states. In other words, ***remove dead branches***.

1. Spurious rules
2. Useless states
3. $\Sigma$-spurious states
4. Spurious states

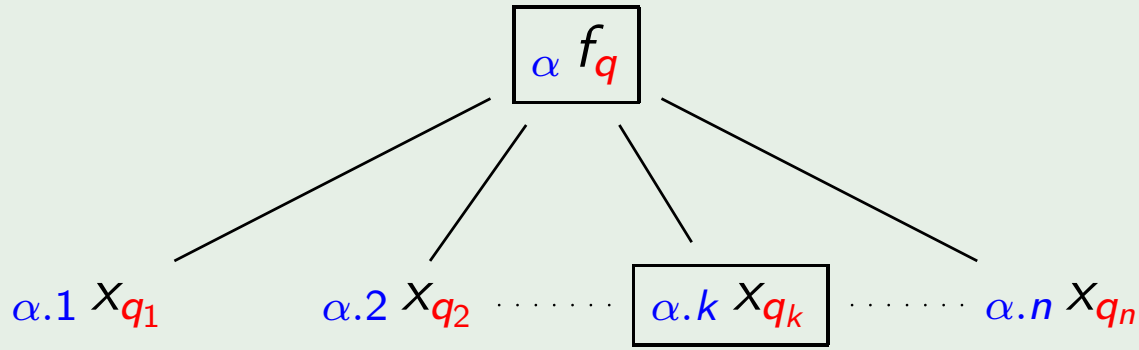
## Definition (Spurious rule)

Let $\mathcal{A}$ be a TAGED. A rule $f(q_1, \ldots, q_n) \to q \in \Delta$ is *spurious* if there exists $k \in [\![1, n]\!]$ such that $q_k =_{\mathcal{A}} q$.



## Definition (Spurious rule)

Let $\mathcal{A}$ be a TAGED. A rule $f(q_1, \ldots, q_n) \to q \in \Delta$ is *spurious* if there exists $k \in [\![1, n]\!]$ such that $q_k =_{\mathcal{A}} q$.

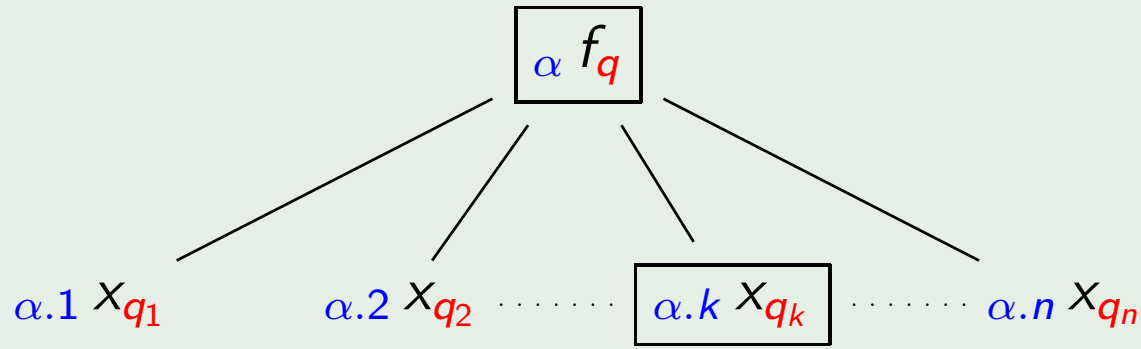



## Lemma (Removal of spurious rules)

*All spurious rules can be removed without altering the accepted language.*

## Definition (Spurious rule)

Let $\mathcal{A}$ be a TAGED. A rule $f(q_1, \ldots, q_n) \to q \in \Delta$ is *spurious* if there exists $k \in [\![1, n]\!]$ such that $q_k =_{\mathcal{A}} q$.




## Proof idea

If a spurious rule was used, a term would have to be equal with one of its strict subterms. Which is absurd. □

Let $p_y^x, p, q \in Q$, $\sigma_1, \ldots, \sigma_m \in \Sigma$, and

$$\mathfrak{Rul}(q) = \left\{ \begin{array}{c} \sigma_1(p_1^1, \ldots, p_{n_1}^1, p, p_1'^1, \ldots, p_{n_1'}'^1) \longrightarrow q \\ \vdots \\ \sigma_m(p_1^m, \ldots, p_{n_m}^m, p, p_1'^m, \ldots, p_{n_m'}'^m) \longrightarrow q \end{array} \right\}$$

## Sure requirements

$$p \in \mathfrak{sReq}(q)$$

## Potential Requirements

$$\mathfrak{pReq}(q) = \{ p \} \cup \left\{ p_y^x, p_y'^x \;\middle|\; x, y \in \ldots \right\}$$

# Cleanup: hunting for spuriousness

## Sure and Potential requirements

Let $p_y^x, p, q \in Q$, $\sigma_1, \ldots, \sigma_m \in \Sigma$, and

$$\mathfrak{Rul}(q) = \left\{ \begin{array}{c} \sigma_1(p_1^1, \ldots, p_{n_1}^1, p, p_1'^1, \ldots, p_{n_1'}'^1) \longrightarrow q \\ \vdots \\ \sigma_m(p_1^m, \ldots, p_{n_m}^m, p, p_1'^m, \ldots, p_{n_m'}'^m) \longrightarrow q \end{array} \right\}$$

### Sure requirements

$$p \in \mathfrak{sReq}(q)$$

### Potential Requirements

$$\mathfrak{pReq}(q) = \{ p \} \cup \left\{ p_y^x, p_y'^x \;\middle|\; x, y \in \ldots \right\}$$

Let $p_y^x, p, q \in Q$, $\sigma_1, \ldots, \sigma_m \in \Sigma$, and

$$\mathfrak{Rul}(q) = \left\{ \begin{array}{c} \sigma_1(p_1^1, \ldots, p_{n_1}^1, p, p_1'^1, \ldots, p_{n_1'}'^1) \to q \\ \vdots \\ \sigma_m(p_1^m, \ldots, p_{n_m}^m, p, p_1'^m, \ldots, p_{n_m'}'^m) \to q \end{array} \right\}$$

## Sure requirements

$$p \in \mathfrak{sReq}(q)$$

## Potential Requirements

$$\mathfrak{pReq}(q) = \{ p \} \cup \left\{ p_y^x, p_y'^x \;\middle|\; x, y \in \ldots \right\}$$

# Cleanup: hunting for spuriousness

## Sure and Potential requirements

Let $p_y^x, p, q \in Q$, $\sigma_1, \ldots, \sigma_m \in \Sigma$, and

$$\mathfrak{Rul}(q) = \left\{ \begin{array}{c} \sigma_1(p_1^1, \ldots, p_{n_1}^1, p, p_1'^1, \ldots, p_{n_1'}'^1) \to q \\ \vdots \\ \sigma_m(p_1^m, \ldots, p_{n_m}^m, p, p_1'^m, \ldots, p_{n_m'}'^m) \to q \end{array} \right\}$$

### Sure requirements

$$\mathfrak{sReq}(q) \stackrel{\text{def}}{=} \bigcap_{\substack{r \in \mathfrak{Rul}(q) \\ q \notin \mathfrak{Ant}(r)}} \mathfrak{Ant}(r),$$

### Potential Requirements

$$\mathfrak{pReq}(q) \stackrel{\text{def}}{=} \bigcup_{r \in \mathfrak{Rul}(q)} \mathfrak{Ant}(r).$$

$\mathfrak{Frnd}(q) =$ "transitive closure of $\mathfrak{pReq}(q)$".
$\mathfrak{Need}(q) =$ "transitive closure of $\mathfrak{sReq}(q)$".

**Definition (Friend states)**

$\mathfrak{Frnd}(q)$: the smallest subset of $Q$ satisfying

1. $\mathfrak{pReq}(q) \subseteq \mathfrak{Frnd}(q)$
2. if $p \in \mathfrak{Frnd}(q)$ then $\mathfrak{pReq}(p) \subseteq \mathfrak{Frnd}(q)$

**Definition (Needs)**

$\mathfrak{Need}(q)$: smallest subset of $Q$ satisfying

1. $\mathfrak{sReq}(q) \subseteq \mathfrak{Need}(q)$
2. if $p \in \mathfrak{Need}(q)$ then $\mathfrak{sReq}(p) \subseteq \mathfrak{Need}(q)$

# Cleanup: hunting for spuriousness
## Needs and friends

*"Only friends of q appear under q"*

**Lemma ("Rely on your Friends" principle)**

*Let $\rho$ a run:* $\forall \alpha, \beta \in \mathcal{P}os(t) : \beta \lhd \alpha \implies \rho(\beta) \in \mathfrak{Frnd}(\rho(\alpha))$.

*"Every need of q appears under q"*

**Lemma (Needs)**

*Let $\rho$ a run such that $\rho(\beta) = q$. For any $p \in \mathfrak{Need}(q)$, there exists a position $\alpha_p \lhd \beta$ such that $\rho(\alpha_p) = p$.*

*"Only friends of q appear under q"*

**Lemma ("Rely on your Friends" principle)**

*Let $\rho$ a run:* $\forall \alpha, \beta \in \mathcal{P}os(t) : \beta \lhd \alpha \implies \rho(\beta) \in \mathfrak{Frnd}(\rho(\alpha))$.

*"Every need of q appears under q"*

**Lemma (Needs)**

*Let $\rho$ a run such that $\rho(\beta) = q$. For any $p \in \mathfrak{Need}(q)$, there exists a position $\alpha_p \lhd \beta$ such that $\rho(\alpha_p) = p$.*

*"Only friends of a final state are useful"*

**Theorem (Removal of useless states)**

*Let $\mathcal{A} = (\Sigma, Q, F, \Delta)$ be a tree automaton. Then*

$$\mathcal{L}ng(\mathcal{A}) = \mathcal{L}ng(\mathcal{A}') \text{ with } \mathcal{A}' \stackrel{def}{=} \mathfrak{Rst}\left(\mathcal{A}, F \cup \bigcup_{q_f \in F} \mathfrak{Frnd}(q_f)\right).$$

*Furthermore, the accepting runs are the same for $\mathcal{A}$ and $\mathcal{A}'$.*

**Proof idea**

Every accepting run is rooted in a final state. Therefore they cannot use any state not in $F \cup \bigcup_{q_f \in F} \mathfrak{Frnd}(q_f)$. □

*"Only friends of a final state are useful"*

**Theorem (Removal of useless states)**

*Let $\mathcal{A} = (\Sigma, Q, F, \Delta)$ be a tree automaton. Then*

$$\mathcal{L}ng(\mathcal{A}) = \mathcal{L}ng(\mathcal{A}') \text{ with } \mathcal{A}' \stackrel{def}{=} \mathfrak{Rst}\left(\mathcal{A}, F \cup \bigcup_{q_f \in F} \mathfrak{Frnd}(q_f)\right).$$

*Furthermore, the accepting runs are the same for $\mathcal{A}$ and $\mathcal{A}'$.*

**Proof idea**

Every accepting run is rooted in a final state. Therefore they cannot use any state not in $F \cup \bigcup_{q_f \in F} \mathfrak{Frnd}(q_f)$. □

**Definition (Support of a state)**

*Support of $q$*: the set of all symbols of $\Sigma$ in which a term which evaluates to $q$ may be rooted.

$$\mathfrak{Sup}(q) \stackrel{\text{def}}{=} \{ f \in \Sigma \mid \exists f(\dots) \to q \in \Delta \}.$$

**Definition ($\Sigma$-spurious state)**

A state $q \in Q$ is a *$\Sigma$-spurious state* if there exist $p, p' \in \mathfrak{Need}(q)$ such that $p =_{\mathcal{A}} p'$ and $\mathfrak{Sup}(p) \cap \mathfrak{Sup}(p') = \varnothing$.

**Lemma (Removal of $\Sigma$-spurious states)**

*Let $\mathcal{A}$ be a TAGED, $S \subseteq Q$ the set of all its $\Sigma$-spurious states, and $\mathcal{A}' = \mathfrak{Rst}(\mathcal{A}, Q \setminus S)$. Then $\mathcal{L}ng(\mathcal{A}) = \mathcal{L}ng(\mathcal{A}')$.*

**Definition (Support of a state)**

*Support of $q$*: the set of all symbols of $\Sigma$ in which a term which evaluates to $q$ may be rooted.

$$\mathfrak{Sup}(q) \stackrel{\text{def}}{=} \{ f \in \Sigma \mid \exists f(\dots) \to q \in \Delta \}.$$

**Definition ($\Sigma$-spurious state)**

A state $q \in Q$ is a *$\Sigma$-spurious state* if there exist $p, p' \in \mathfrak{Need}(q)$ such that $p =_{\mathcal{A}} p'$ and $\mathfrak{Sup}(p) \cap \mathfrak{Sup}(p') = \varnothing$.

**Lemma (Removal of $\Sigma$-spurious states)**

*Let $\mathcal{A}$ be a TAGED, $S \subseteq Q$ the set of all its $\Sigma$-spurious states, and $\mathcal{A}' = \mathfrak{Rst}(\mathcal{A}, Q \setminus S)$. Then $\mathcal{L}ng(\mathcal{A}) = \mathcal{L}ng(\mathcal{A}')$.*

# Cleanup: hunting for spuriousness

## $\Sigma$-spurious states

**Definition ($\Sigma$-spurious state)**

A state $q \in Q$ is a *$\Sigma$-spurious state* if there exist $p, p' \in \mathfrak{Need}(q)$ such that $p =_{\mathcal{A}} p'$ and $\mathfrak{Sup}(p) \cap \mathfrak{Sup}(p') = \varnothing$.

**Lemma (Removal of $\Sigma$-spurious states)**

*Let $\mathcal{A}$ be a TAGED, $S \subseteq Q$ the set of all its $\Sigma$-spurious states, and $\mathcal{A}' = \mathfrak{Rst}(\mathcal{A}, Q \setminus S)$. Then $\mathcal{L}ng(\mathcal{A}) = \mathcal{L}ng(\mathcal{A}')$.*

**Proof idea**

If $q$ appears in an accepting run, then so must $p$ and $p'$. But they cannot satisfy the equality (rooted in different symbols). Contradiction. So $q$ cannot appear in any accepting run. □

**Definition (Σ-spurious state)**

A state $q \in Q$ is a *Σ-spurious state* if there exist $p, p' \in \mathfrak{Need}(q)$ such that $p =_{\mathcal{A}} p'$ and $\mathfrak{Sup}(p) \cap \mathfrak{Sup}(p') = \varnothing$.

**Lemma (Removal of Σ-spurious states)**

*Let $\mathcal{A}$ be a TAGED, $S \subseteq Q$ the set of all its Σ-spurious states, and $\mathcal{A}' = \mathfrak{Rst}(\mathcal{A}, Q \setminus S)$. Then $\mathcal{L}ng(\mathcal{A}) = \mathcal{L}ng(\mathcal{A}')$.*

**Proof idea**

If $q$ appears in an accepting run, then so must $p$ and $p'$. But they cannot satisfy the equality (rooted in different symbols). Contradiction. So $q$ cannot appear in any accepting run. □

**Definition (Spurious states)**

Let $\mathcal{A}$ be a TAGED. A state $q \in Q$ is said to be a *spurious state* if there exists $p \in \mathfrak{Need}(q)$ such that $p =_{\mathcal{A}} q$.

**Lemma (Removal of spurious states)**

*Let $\mathcal{A}$ be a TAGED, $S \subseteq Q$ the set of all its spurious states, and $\mathcal{A}' = \mathfrak{Rst}(\mathcal{A}, Q \setminus S)$. Then $\mathcal{L}ng(\mathcal{A}) = \mathcal{L}ng(\mathcal{A}')$.*

**Proof idea**

Suppose $q$ appears in an accepting run at position $\beta$, then $\exists \alpha_p \lhd \beta$ *st.* $\rho(\alpha_p) = p$. A strict subterm and its parent are equal. Contradiction. So $q$ does not appear. □

**Definition (Spurious states)**

Let $\mathcal{A}$ be a TAGED. A state $q \in Q$ is said to be a *spurious state* if there exists $p \in \mathfrak{Need}(q)$ such that $p =_{\mathcal{A}} q$.

**Lemma (Removal of spurious states)**

*Let $\mathcal{A}$ be a TAGED, $S \subseteq Q$ the set of all its spurious states, and $\mathcal{A}' = \mathfrak{Rst}(\mathcal{A}, Q \setminus S)$. Then $\mathcal{L}ng(\mathcal{A}) = \mathcal{L}ng(\mathcal{A}')$.*

**Proof idea**

Suppose $q$ appears in an accepting run at position $\beta$, then $\exists \alpha_p \lhd \beta$ *st.* $\rho(\alpha_p) = p$. A strict subterm and its parent are equal. Contradiction. So $q$ does not appear. □

**Definition (Spurious states)**

Let $\mathcal{A}$ be a TAGED. A state $q \in Q$ is said to be a *spurious state* if there exists $p \in \mathfrak{Need}(q)$ such that $p =_{\mathcal{A}} q$.

**Lemma (Removal of spurious states)**

*Let $\mathcal{A}$ be a TAGED, $S \subseteq Q$ the set of all its spurious states, and $\mathcal{A}' = \mathfrak{Rst}(\mathcal{A}, Q \setminus S)$. Then $\mathcal{L}ng(\mathcal{A}) = \mathcal{L}ng(\mathcal{A}')$.*

**Proof idea**

Suppose $q$ appears in an accepting run at position $\beta$, then $\exists \alpha_p \lhd \beta$ *st.* $\rho(\alpha_p) = p$. A strict subterm and its parent are equal. Contradiction. So $q$ does not appear. □

# Cleanup: hunting for spuriousness
## An example

```
TAGED 'example 1' [64] = {
  states = #7{q0, q1, q2, q3, q4, q5, q6}
  final = #1{q6}
  rules = #16{
a2()->q0, a2()->q2, a2()->q4, a3()->q3, a5()->q0, a5()->q2,
a5()->q4, f1(q5)->q5, f3(q1)->q5, g1(q1, q5)->q5, g3(q0, q0)->q5,
g3(q1, q5)->q5, g5(q1, q1)->q5, h2(q2, q3, q4)->q1,
h3(q0, q0, q1)->q6, h3(q2, q3, q4)->q1
  }
  ==rel = #3{(q0,q0), (q3,q4), (q4,q3)}
}
```

State $q_1$ is $\Sigma$-spurious, because it depends on $q_3$ and $q_4$ ($q_3, q_4 \in \mathfrak{Need}(q_1)$ and $\mathfrak{Sup}(q_3) \cap \mathfrak{Sup}(q_4) = \{ a_3 \} \cap \{ a_2, a_5 \} = \varnothing$). Furthermore $q_1 \in \mathfrak{Need}(q_6)$, so $q_6$ is unreachable, and $\mathcal{L}\mathrm{ng}(\mathcal{A}) = \varnothing$.

# Cleanup: hunting for spuriousness
## An example

```
TAGED 'example 1' [64] = {
  states = #7{q0, q1, q2, q3, q4, q5, q6}
  final = #1{q6}
  rules = #16{
a2()->q0, a2()->q2, a2()->q4, a3()->q3, a5()->q0, a5()->q2,
a5()->q4, f1(q5)->q5, f3(q1)->q5, g1(q1, q5)->q5, g3(q0, q0)->q5,
g3(q1, q5)->q5, g5(q1, q1)->q5, h2(q2, q3, q4)->q1,
h3(q0, q0, q1)->q6, h3(q2, q3, q4)->q1
  }
  ==rel = #3{(q0,q0), (q3,q4), (q4,q3)}
}
```

State $q_1$ is $\Sigma$-spurious, because it depends on $q_3$ and $q_4$ ($q_3, q_4 \in \mathfrak{Need}(q_1)$ and $\mathfrak{Sup}(q_3) \cap \mathfrak{Sup}(q_4) = \{ a_3 \} \cap \{ a_2, a_5 \} = \varnothing$). Furthermore $q_1 \in \mathfrak{Need}(q_6)$, so $q_6$ is unreachable, and $\mathcal{L}\mathrm{ng}(\mathcal{A}) = \varnothing$.