tightcenter

panikzettel.htwr-aachen.de

Advanced Automata Theory Panikzettel

Caspar Zecha, Tobias Polock,
Philipp Schröer, Julian Schakib-Ekbatan

Version 6 — 19.08.2020

Contents

1 Introduction

This Panikzettel is about the lecture Advanced Automata Theory by Prof. Löding held in the summer semester 2018.

This Panikzettel is Open Source. We appreciate comments and suggestions at https://git.rwth-aachen.de/philipp.schroer/panikzettel.

2 Notation and Automata

We use the following notation in this lecture and define the basic automata:

- $\Sigma, \Gamma, \dots$ for alphabets,
- $a, b, \dots$ for letters,
- ε for the empty word,
- $u, v, w, \dots$ for words,
- Σ^* (Σ^+) for the set of words (of non-empty words) over Σ,
- $L, K, \dots$ for languages (subsets of Σ^*),
- $\mathcal{A}, \mathcal{B}, \dots$ for automata.

Definition: Nondeterministic Finite Automaton (NFA)

$\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ with

- finite state set Q, initial state $q_0 \in Q$,
- transition relation $\Delta \subseteq Q \times \Sigma \times Q$,
- set $F \subseteq Q$ of final states.

$\mathcal{A}$ *accepts* $w = a_1 \dots a_m$ if there is a *run* $\varrho = \varrho(0) \dots \varrho(m)$ with $\varrho(m) \in F$ and $\varrho(0) = q_0$, $(\varrho(i-1), a_i, \varrho(i)) \in \Delta$.

The language recognised by $\mathcal{A}$ is $L(\mathcal{A}) := \{w \in \Sigma^* \mid \mathcal{A} \text{ accepts } w\}$.

Definition: Deterministic Finite Automaton (DFA)

$\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ with

- finite state set Q, initial state $q_0 \in Q$,
- transition function $\delta : Q \times \Sigma \to Q$,
- set $F \subseteq Q$ of final states.

The language accepted by $\mathcal{A}$ is $L(\mathcal{A}) := \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}$. $\delta^* : Q \times \Sigma^* \to Q$ is defined as:

$$\delta^*(q, \varepsilon) := q,$$
$$\delta^*(q, wa) := \delta(\delta^*(q, w), a).$$

We can create a DFA from an NFA recognising the same language by using subset construction. Here the idea is to collect the NFA's non-deterministic possibilities as subsets, consisting of all reachable states from the previous subset. The initial subset consists of only the initial state. Subset construction on an NFA with n states may in worst case lead to a DFA with 2^n states.

3 Minimisation of NFAs

DFAs can be minimised by merging equivalent states. The result is a unique minimal DFA that is equivalent to the given DFA. This minimisation is efficient (polynomial time).

We want to analyse the problem of state merging on NFAs, i.e. taking the quotient of an automaton. Merging states can change the accepted language.

3.1 Quotient Automaton

Let $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ be an NFA and let $\sim$ be some *equivalence relation* (reflexive, symmetric and transitive) on Q. For a state $q \in Q$, the equivalence class is defined as $[q]_\sim = \{ p \in Q \mid p \sim q \}$.

Now the *quotient automaton* $\mathcal{A}_{/\sim}$ of an NFA with respect to some equivalence relation $\sim$ is easy. Merge states into equivalence classes (single states) and connect two classes if there exists a transition from a state of one class to a state in the other class. A state representing an equivalence class is *initial* or *accepting* if at least one of the states it contains is.

By induction one can show that the quotient automaton accepts all words in $L(\mathcal{A})$ and an example can be given where the equivalence relation leads to $L(\mathcal{A}) \subset L(\mathcal{A}_{/\sim})$. Thus in general $L(\mathcal{A}) \subseteq L(\mathcal{A}_{/\sim})$. To assure that the language is not changed consider only relations which merge language equivalent states.

3.1.1 Quotient on DFAs

Viewing a DFA as an NFA and then applying the quotient operation on the NFA, the resulting quotient $\mathcal{A}_{/\sim}$ is a DFA again exactly if $\sim$ is a *congruence relation* w.r.t. the transition function δ: $p \sim q \implies \delta(p,a) \sim \delta(q,a) \,\forall\, a \in \Sigma$.

For DFA $\mathcal{A}$ and $q \in Q$, let $\mathcal{A}_q$ be the DFA $\mathcal{A}$ with q as initial state. We then define $\sim_\mathcal{A}$ by $q \sim_\mathcal{A} p$ iff $L(\mathcal{A}_q) = L(\mathcal{A}_p)$ and call states q and p *language equivalent*.

Using this, the quotient $\mathcal{A}_{/\sim_\mathcal{A}}$ is the *minimal DFA accepting the same language as* $\mathcal{A}$. The minimal DFA can be computed efficiently in $\mathcal{O}(|\Sigma||Q| \cdot \log |Q|)$.

3.2 Reduction of NFAs

Minimal NFAs can be exponentially smaller than minimal DFAs w.r.t. the number of states. However, there can be several non-isomorphic minimal NFAs for a language.

Additionally, deciding whether an NFA is minimal is computationally hard (PSPACE-complete), as are the two other decision problems on the right. Remember: P $\subseteq$ NP $\subseteq$ PSPACE.

Definition: NFA Decision Problems

Given NFAs $\mathcal{A}$, $\mathcal{B}$ over the same alphabet.

1. Is $\mathcal{B}$ an NFA equivalent to $\mathcal{A}$ with the least possible number of states?
2. Are $\mathcal{A}$ and $\mathcal{B}$ equivalent?
3. Is $L(A) \neq \Sigma^*$? (minimal NFA has more than one state)

3.3 Bisimulation

Bisimulation provides an equivalence relation on the states of an NFA. It is similar to the $\sim_{\mathcal{A}}$ relation for DFAs.

The bisimulation relation $\approx_{\mathcal{A}}$ is the biggest relation (w.r.t. $\subseteq$) that satisfies the following three conditions: If $p \approx_{\mathcal{A}} q$, then

- $p \in F \Leftrightarrow q \in F$,
- $\forall (p, a, p') \in \Delta \, \exists (q, a, q') \in \Delta : p' \approx_{\mathcal{A}} q'$,
- $\forall (q, a, q') \in \Delta \, \exists (p, a, p') \in \Delta : p' \approx_{\mathcal{A}} q'$.

This relation can also be characterised as the *bisimulation game* (see right). If the Duplicator has a winning strategy in $\mathrm{BG}(\mathcal{A}, p, \mathcal{B}, q)$, we call $(\mathcal{A}, p)$ and $(\mathcal{B}, q)$ bisimilar. $p \approx_{\mathcal{A}} q \iff (\mathcal{A}, p) \approx (\mathcal{A}, q)$

Definition: Bisimulation Game

Given two automata $\mathcal{A}, \mathcal{B}$ with $p \in Q_{\mathcal{A}}$ and $q \in Q_{\mathcal{B}}$, we denote the bisimulation game between the *spoiler* and the *duplicator* as

$$\mathrm{BG}(\mathcal{A}, p, \mathcal{B}, q).$$

From a configuration (p', q') the spoiler chooses a transition $(p', a, p'') \in \Delta_{\mathcal{A}}$ or $(q', a, q'') \in \Delta_{\mathcal{B}}$.

Then the duplicator reacts accordingly from the other state.

Unless the spoiler creates a configuration $(\tilde{p}, \tilde{q})$ such that $\tilde{p} \in F_{\mathcal{A}} \Leftrightarrow \tilde{q} \notin F_{\mathcal{B}}$ or the duplicator doesn't have a transition over the chosen letter, the duplicator wins.

3.4 Block Refinement

We can compute the bisimulation equivalence using *block refinement*. This algorithm iteratively tries to split *blocks* of states until they cannot be separated by words anymore.

When the blocks can be split, they're called *not compatible*: If there are $p, q \in B$ with

- $\exists (p, a, p') \in \Delta$ with $p' \in B'$
- $\neg \exists (q, a, q') \in \Delta$ with $q' \in B'$

The algorithm's complexity is $O(|\Delta| \cdot |Q|)$.

Algorithm: Block Refinement

Input: NFA $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$.

Output: Partition π.

```
1. $\pi = \{ Q \setminus F, F \}$.
2. While $\exists B, B' \in \pi, a \in \Sigma$ such that
   $B$ is not compatible with $(B', a)$:
   • split $B$ with respect to $(B', a)$ into
     $\{ p \in B \mid \exists p' \in B' : (p, a, p') \in \Delta \}$,
     $\{ p \in B \mid \neg \exists p' \in B' : (p, a, p') \in \Delta \}$.
```

3.5 Comparison of Reductions

Thus the $\approx_{\mathcal{A}}$-quotient can be bigger than the $\sim_{\mathcal{A}}$-quotient. The latter can be bigger than the *optimal quotient* (the smallest quotient NFA that accepts the same language as $\mathcal{A}$). But the optimal quotient is not necessarily the smallest equivalent NFA, imagine for example an NFA that does not use its non-determinism: its quotient automaton can not be smaller than the minimal DFA.

4 Learning Algorithms for DFAs

We want to develop an algorithm (*learner*) that constructs a *deterministic* finite automaton for an unknown language L. The learner has some information about L. Different settings are possible:

- *Passive*: The learner is given a finite set of words (not) in L.
- *Active*: The learner can query the language. They can ask whether a word w is in L or can construct an automaton $\mathcal{A}$ and ask whether $L = L(\mathcal{A})$.

4.1 Passive: Construction of Finite Automata from Examples

We first consider the passive learning where we want learners as computable functions f that take a sample S and return a DFA $\mathcal{A} = f(S)$.

The learner should be *consistent*, that is $f(S)$ should be consistent with every sample S. But this is not enough, as a simple consistent learner may not generalise well – e.g. accepting precisely the given samples, but not more.

Therefore we define f *learning in the limit* if there is an n_0 such that for all $n \geq n_0 : L(f(S^{L,n})) = L$.

The set of positive (negative) samples contains all words that are lexicographically smaller than the nth word and (not) in the language L.

Definition: Sample

Samples are of the form $S = (S_+, S_-)$ where S_+ (S_-) is a finite set of positive (negative) examples where $S_+ \cap S_- = \emptyset$.

A DFA $\mathcal{A}$ is *consistent* with S if $S_+ \subseteq L(\mathcal{A})$ and $S_- \cap L(\mathcal{A}) = \emptyset$.

$$S_+^{L,n} = \{ w_i \in L \mid i \leq n \}$$
$$S_-^{L,n} = \{ w_i \notin L \mid i \leq n \}$$
$$S^{L,n} = (S_+^{L,n}, S_-^{L,n})$$

Let f be a learner such that $f(S)$ is a smallest DFA that s consistent with S, for each sample S. Then, f learns all regular languages in the limit. There exists a learner that constructs a smallest DFA that is consistent for each sample S. The learner enumerates all DFAs by increasing size and checks for consistency with S.

Unfortunately, checking whether there exists a DFA with at most m states that is consistent with a sample S is NP-complete. This can be shown using a reduction from the 3-Coloring decision problem which translates to the question "Is there a consistent DFA with at most 4 states?", encoding adjacency constraints of the coloring problem in the negative samples and using positive samples to simulate coloring.

4.2 Regular Positive Negative Inference (RPNI)

The *RPNI-algorithm* is a learner that computes a DFA $f(S)$ for a sample S in polynomial time and learns every regular language in the limit.

The algorithm first constructs a simple "prefix tree acceptor" DFA $\mathcal{A}(S_+)$ which accepts exactly all words from S_+. Then the automaton is simplified using quotients with regard to congruence relations. The quotients are tried in canonical order, and only those that are consistent with the sample are used. Thus the automaton is simplified and its language is generalised, i.e. S_+ is still accepted, and S_- is still rejected.

Algorithm: Regular Positive Negative Inference (RPNI)

Input: Sample $S = (S_+, S_-)$.

Output: (partial) DFA $\mathcal{A}(S_+)_{/\sim}$.

```
1. Construct $\mathcal{A}(S_+)$ accepting precisely $S_+$. (States are prefixes of words in $S_+$)
2. Sort states $Q_+ = \{ u_0, \dots, u_n \}$ from $\mathcal{A}(S_+)$ in canonical order: $u < v \iff |u| < |v| \lor (|u| = |v| \land u$ is before $v$ in lexicographic ordering).
3. Initialise $\sim = \{ (u,u) \mid u \in Q_+ \}$.
   $\mathcal{A}(S_+)_{/\sim}$ looks just like $\mathcal{A}(S_+)$.
4. For each $u_i \in Q_+ \setminus \{ u_0 \}$:
   • If $u_i \not\sim u_j$ for all $u_j \in \{ u_0, \dots, u_{i-1} \}$:
     Skip words that have already been merged with smaller words (in canonical order)
     – Try next $u_k \in Q_+$ with $k < i$, until $L(\mathcal{A}(S_+)_{/\sim'}) \cap S_- \neq \emptyset$.
       Try to merge with the next (smaller) state (in canonical order) such that no negative example is accepted.
       ∗ $\sim'$ = extend $\sim$ to the smallest congruence with $(u_i, u_k)$.
         Merge states containing $u_i$ and $u_k$ and preserve congruence, i.e. if merging creates transitions to different states in the quotient, recursively merge those too.
     – Commit $\sim = \sim'$.
```

We consider the question "When is a sample complete enough for RPNI to produce a DFA for L?"

First we need *minimal representatives* and *minimal transition representatives* for L:

$$\mathrm{MR}(L) := \{ w \in \Sigma^* \mid w \in prf(L) \land \forall u \sim_L w : w \leq u \}$$

$$\mathrm{MTR}(L) := \left\{ w \in \Sigma^* \;\middle|\; \begin{array}{c} w \in prf(L) \land w = ua \\ u \in \mathrm{MR}(L) \land a \in \Sigma \end{array} \right\}$$

We define a sample S *RPNI-complete for* L if

- For each $q \in F_L$ (F_L being the final states in the minimal automaton for L), there is a $u \in S_+$ such that u reaches q in $\mathcal{A}_L$,
- $\forall w \in \mathrm{MTR}(L)$, there is a $v \in \Sigma^*$ such that $wv \in S_+$,
- $\forall u \in \mathrm{MR}(L)$ and $\forall v \in \mathrm{MTR}(L)$ with $u \not\sim_L v$, there exists a w such that $uw, vw \in S$ and $uw \in S_+ \iff vw \in S_-$.

If a sample is RPNI-complete, then RPNI will compute the minimal DFA for the language. For each regular language L, there is a RPNI-complete sample of polynomial size in the size of $\mathcal{A}_L$.

4.3 Active Learning with L^*

In active learning the learner can ask specific questions about the target language. We consider the *Minimally Adequate Teacher (MAT) model* which has the following two types of questions:

- Membership queries: "Is $w \in L$?" for a chosen word w.
- Equivalence queries: "Does $\mathcal{A}$ accept L?" for a DFA $\mathcal{A}$.
  If the answer is "no", then the teacher provides an arbitrary counter-example w.

The L^* algorithm asks these queries to build a minimal DFA for the target language. It does so indirectly by inferring $\sim_L$-classes, which correspond to states in $\mathcal{A}_L$.

The general idea is as follows: We have a set of *representatives* $R \subseteq \Sigma^*$ for our equivalence classes and a set of words, called *experiments* $E \subseteq \Sigma^*$. We've found a new representative $u \in R \cdot \Sigma$ if u is *escaping*: For all other representatives $v \in R$ there is an experiment $w \in E$, so that $uw \in L \Leftrightarrow vw \notin L$. One can think of u and the vs as states in the automaton: If starting from state u we get a different result reading w than starting from all vs, u must be a distinct state.

If u is not escaping, there is a unique representative $v \in R$ with $uw \in L \iff vw \in L$ for all experiments $w \in E$. Then u and v are called *compatible*.

All our measurements are saved in an *observation table* $B = (R, E, f)$ with representatives R, experiments E and the data provided by a function $f : (R \cdot E \cup R \cdot \Sigma \cdot E) \to \{0, 1\}$. B is an observation table for L if $f(w) = 1 \Leftrightarrow w \in L$.

In the top part, we have the $R \cdot E$. Throughout the algorithm, rows in the top are all distinct – each row corresponds to one equivalence class. The bottom part contains all other measurements. Iff a row in the bottom is unique among all others, then the corresponding $w \in R \cdot \Sigma$ is escaping and we make it a representative (move it to the top). B is *closed* if no escaping $w \in R \cdot \Sigma$ exists.

B	ε	a
ε	0	1
a	1	1
ab	0	0
b	1	1
aa	1	1
aba	0	0
abb	1	1

After adding escaping words from $R \cdot \Sigma$ until B is closed, we construct the *hypothesis* $\mathcal{A}_B$ and ask the teacher the equivalence query. The construction of the automaton is simple: Each of the representatives becomes a state, and each transition from a state $u \in R$ with a goes to the unique $v \in R$ compatible with the word ua. Representative ε becomes the initial state. Final states are all states with their representative $u \in R$ evaluating to 1, i.e $f(u \cdot \varepsilon) = 1$ in the observation table.

If the hypothesis is correct, we're done. Otherwise, we extend the experiments E based on the counter-example $w = a_1 \cdots a_m$ the teacher provided. First, define $r_i = \delta_B^*(q_0^B, a_1 \cdots a_i)$ to be the state reached in $\mathcal{A}_B$ after reading the prefix of length i of w. Per construction of $\mathcal{A}_B$, this is always a representative: $r_i \in R$.

Now we search for a *breakpoint* $i \in \{1, \ldots, m\}$ where our hypothesis automaton becomes wrong: $r_{i-1}a_i \cdots a_m \in L \iff r_i a_{i+1} \cdots a_m \notin L$. We then add the remaining suffix $a_{i+1} \cdots a_m$ to our experiments E. This makes $r_{i-1}a_i$ an escaping word and we must add it to R, introducing a new state in our hypothesis.

Algorithm: L^*

Input: Teacher T for a regular language $L \in \Sigma^*$.

Output: DFA $\mathcal{A}_B$.

```
• Initialise $B = (R, E, f)$ with $R = \{ \varepsilon \}$, $E = \{ \varepsilon \}$ and $f(w) = T(w) \quad \forall w \in \{ \varepsilon \} \cup \Sigma$.
• Repeat:
  1. Add escaping words from $R\Sigma$ to $R$ until $B$ is closed, and fill the table.
  2. Ask an equivalence query for hypothesis $\mathcal{A}_B = (Q_B, \Sigma, q_0^B, \delta_B, F_B)$ with: $Q_B = R$,
     $q_0^B = \varepsilon$, $F_B = \{ u \mid f(u) = 1 \}$, and
     $\delta_B(u, a) = v$ for the (unique) $v \in R$ compatible with $ua$.
  3. If the hypothesis is correct, return $\mathcal{A}_B$.
  4. For a counter-example $W = a_1 \cdots a_m$:
     a) Find a breakpoint $i$.
     b) Add $a_{i+1} \cdots a_m$ to $E$ and fill the table.
```

The algorithm L^* terminates in polynomial time and returns a DFA $\mathcal{A}_B$ that is isomorphic to $\mathcal{A}_L$.

Each iteration of the loop increases $|R|$, which is bounded by the index of L, since the representatives are all in different $\sim_L$-classes, separated by the experiments.

5 Automata and Logic

We want to decide whether all executions of a given transition system (automaton) satisfy a certain specification. Therefore, we want a logic that is equivalent to automata in expressive power.

5.1 MSO Logic on Words

Definition: Word Structure

A non-empty word $w = b_1 \dots b_m$ over $\Sigma = \{ a_1, \dots, a_n \}$ defines the *word structure*

$$\underline{w} = (dom(w), S^w, <^w, min^w, max^w, P^w_{a_1}, \dots, P^w_{a_n}),$$

where

- $dom(w) = \{ 1, \dots, m \}$ are positions,
- S^w is the successor relation,
- $<^w$ the less relation on $dom(w)$,
- $min^w = 1$ and $max^w = m$ and
- $P^w_{a_i} := \{ j \in dom(w) \mid b_j = a_i \}$ for $i = 1, \dots, n$.

Note that we can eliminate min, max and S in FO and MSO, and in MSO $<$ instead of S.

Definition: Syntax of MSO Formulae

Variables

x, y, z for positions,
X, Y, Z for sets of positions.

Constants

min, max

Atomic formulae

$x = y$	*equality*
$S(x, y)$	*successor*
$x < y$	*before*
$P_a(x)$	*a at position x*
$X(y)$	$y \in X$

Complex formulae

with $\neg$, $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$ and $\exists$, $\forall$.

We use *monadic second-order* (MSO) logic. This logic is a second-order logic with quantification over unary predicates. We are allowed to quantify over sets! Given an alphabet Σ, we have $MSO_\Sigma[S,<]$ *formulae*. Such formulae which only use quantifiers over first-order variables (position variables) are called $FO_\Sigma[S,<]$ *formulae*. If there are free variables of a formula φ among $x_1,\dots,x_m,X_1,\dots,X_n$, then this is indicated by $\varphi(x_1,\dots,x_m,X_1,\dots,X_n)$. A *sentence* is a formula without free variables.

Definition: Interpretation of Formulae

If φ evaluates to true in $\underline{w}$ when interpreting x_i by k_i and X_i by K_i:

$$(\underline{w},k_1,\dots,k_m,K_1,\dots,K_n) \models \varphi(x_1,\dots,x_m,X_1,\dots,X_n)$$

$$\text{or} \qquad \underline{w} \models \varphi[k_1,\dots,k_m,K_1,\dots,K_n]$$

- $\underline{w}$ a word structure,
- $k_1,\dots,k_m$ positions w,
- $K_1,\dots,K_n$ sets of positions in w.

The *language defined by* φ is

$$L(\varphi) := \{\, w \in \Sigma^+ \mid \underline{w} \models \varphi \,\}.$$

If φ is an $MSO_\Sigma[S,<]$-sentence, we call the language $MSO_\Sigma[S,<]$*-definable* (MSO-definable for short). If φ does not use set quantifiers, then the language is called $FO_\Sigma[S,<]$*-definable* (FO-definable for short).

5.2 The Equivalence Theorem

We show the equivalence of automata and formulae. In the direction from automata to formulae we build a formula that expresses the existence of an accepting run in the automaton. In the other direction from formulae to automata we use induction over the structure of the formula.

Theorem: Equivalence Theorem

A language $L \in \Sigma^*$ is regular iff it is MSO-definable.

5.2.1 From Automata to Formulae

We want to construct a sentence $\varphi_\mathcal{A}$ with $\underline{w} \models \varphi_\mathcal{A}$ iff $w \in L(\mathcal{A})$. $\varphi_\mathcal{A}$ has to express that there is an accepting run of the NFA $\mathcal{A}$ on w.

The idea is to ask for i sets X_i, one for each state, where X_i is the set of positions in which $\mathcal{A}$ is in state i.

Let $\mathcal{A} = (\{1,\dots,m\},\Sigma,1,\Delta,F)$ be an NFA. $\mathcal{A}$ accepts w iff $\underline{w} \models \varphi_\mathcal{A}$.

$$\begin{aligned}
&\varphi_\mathcal{A} := \exists X_1,\dots \exists X_m[\\
&\quad \forall x(X_1(x) \vee \dots \vee X_m(x)) \wedge \textstyle\bigwedge_{i\neq j} \neg\exists x(X_i(x) \wedge X_j(x)) && \text{($X_1,\dots,X_m$ form a partition)} \\
&\quad \wedge X_1(min) && \text{(on min, the initial state is used)} \\
&\quad \wedge \forall x \forall y(S(x,y) \to \textstyle\bigvee_{(i,a,j)\in\Delta}(X_i(x) \wedge P_a(x) \wedge X_j(y))) && \text{(transitions are used everywhere)} \\
&\quad \wedge \bigvee (i,a,j) \in \Delta, j \in F(X_i(max) \wedge P_a(max)) && \text{(last transition goes to an accepting state)} \\
&]
\end{aligned}$$

5.2.2 From Formulae to Automata

We construct the automaton inductively over the structure over the formulae. Since automata are closed over negation, intersection and union, translation of $\neg$, $\wedge$ and $\vee$ is easy. However, variables are more difficult.

We first translate the formulae to MSO$_0$, which is a subset of MSO-formulae where all variables are set variables (the usual first-order variables are replaced). The idea here is to use singleton sets $\{ x \}$ for all first-order variables x.

Theorem: MSO$_0$ Equivalence

Each MSO sentence φ over words is equivalent to an MSO$_0$ sentence.

Definition: Syntax of MSO$_0$

Like MSO, but only with set variables and:

Atomic formulae

$X \subseteq Y$	*subset*
$X \subseteq P_a$	*only positions of a*
$Sing(X)$	$\lvert X \rvert = 1$
$S(X, Y)$	$X = \{ x \}, Y = \{ y \}, S(x, y)$
$X < Y$	$X = \{ x \}, Y = \{ y \}, x < y$

One can specify an automaton for each MSO$_0$ formula inductively over the syntax of such formulae. For simplicity, we restrict formulae to operators $\wedge, \neg$ and existential quantifiers. For MSO$_0$ formulae over Σ with n free variables, the automaton will recognise a language over the alphabet $\Sigma \times \{ 0, 1 \}^n$. The $\{ 0, 1 \}^n$ part encodes membership of positions of the free (set-)variables $X_1, \ldots, X_n$. Note that while words are defined over sentences (no free variables), an inductive definition requires handling formulae with free variables that arise in sub-formulae of quantors.

For the translation, assume prenex normal form, that is $Q_1 X_1 \ldots Q_n X_n \; \eta(X_1, \ldots, X_n)$ with $Q_i \in \{ \forall, \exists \}$, η is quantifier-free. In the following we denote bitvectors in $\Sigma \times \{ 0, 1 \}^n$ as $(a, x_1, \ldots, x_n)$.

MSO$_0$	**Automaton**
$X_i \subseteq X_j$	Reject iff $(a, x_1, \ldots, x_n)$ with $x_i > x_j$ occurs anywhere in the word.
$X_i \subseteq P_a$	Reject iff $(a', x_1, \ldots, x_n)$ with $x_i = 1$ and $a \neq a'$ occurs anywhere in the word.
$Sing(X_i)$	Check that $(a, x_1, \ldots, x_n)$ with $x_i = 1$ occurs exactly once.
$S(X_i, X_j)$	Check that the unique occurrence of $(a, x_1, \ldots, x_n)$ with $x_i = 1$ is directly followed by the unique occurrence of $(a', x'_1, \ldots, x'_n)$ with $x'_j = 1$.
$X_i < X_j$	Check that the unique occurrence of $(a, x_1, \ldots, x_n)$ with $x_i = 1$ is somewhere followed by the unique occurrence of $(a', x'_1, \ldots, x'_n)$ with $x'_j = 1$.
$\neg\varphi$	Accept the complement of the automaton for φ.
$\varphi \wedge \psi$	Accept the product of the automata for φ and ψ.
$\exists X_i \; \varphi$	A *projection* of the one for φ: For each transition rule $(p, (a, x_1, \ldots, x_n), q)$ in the automaton for φ, the new automaton contains a transition rule $(p, (a, x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n), q)$

Definition: Projection

Alphabets Σ_1 and Σ_2, function $f : \Sigma_1 \to \Sigma_2$.

- For $w \in \Sigma_1^*$ with $w = a_1 \ldots a_n$, $f(w) = f(a_1) \ldots f(a_n)$.
- For $L \subseteq \Sigma_1^*$, $f(L) = \{ f(w) \mid w \in L \}$

Theorem:
Regularity of Projections

If $L \subseteq \Sigma_1$ is regular, so is the projection $f(L)$.

This translation yields an automaton with at most $\left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\}|\varphi|$ states. The reason for that being that computing the complement of an automaton may require determinization of it first. There is no translation that is bounded by a tower of the form $\left.2^{2^{\cdot^{\cdot^{2^{|\varphi|}}}}}\right\}k$ for a fixed k.

5.2.3 Consequences of the equivalence theorem

Theorem: Reduction of MSO to EMSO

For every MSO formula φ on words, there is an *existential MSO* (EMSO) formula

$$\psi = \exists X_1 \exists X_2 \ldots \exists X_n \vartheta$$

with an FO formula ϑ such that ψ is equivalent to φ.

Proof. Given a formula φ, construct the automaton $\mathcal{A}$ that accepts the language defined by φ. Then translate it back into an MSO formula. This construction yields an existential MSO formula. □

Theorem: Satisfiability and Equivalence of MSO Formulae

Satisfiability and equivalence of MSO formulae over words is decidable.

Proof. The corresponding properties (non-emptiness and equality of accepted languages) on the corresponding automata are decidable. □

5.3 FO Definability

Now, we will look at FO formulae. The important difference to MSO formulae is that modulo counting is not possible in FO.

In the following we will characterise FO-definable languages by different equivalent conditions.

Theorem: Characterisation of FO-definable Languages

Let L be a regular language. The following conditions are equivalent:

- L is FO-definable.
- L is non-counting. (**??**)
- The syntactic monoid of L is aperiodic (group-free). (**??**)
- L is definable by a star-free expression. (**??**)
- L is definable in linear temporal logic (LTL). (**??**)

5.3.1 Counting Languages

Definition: Counting Languages

A language L is called *counting* if there are $u, v \in \Sigma^*$, such that for all $n \geq 1$: $uv^n \not\sim_L uv^{n+1}$

Correspondingly L is called *non-counting* if for all $u, v \in \Sigma^*$ there is $n \geq 1$: $uv^n \sim_L uv^{n+1}$

The counting condition exactly defines the difference between FO-definable and MSO-definable formulae.

5.3.2 Language Recognition by Monoids

A DFA defines a function from finite words to states: $\delta^* : \Sigma^* \to Q$. This computation of $\delta^*(wa)$ is based on the computation of $\delta^*(w)$ and a. The function δ^* is not *compositional*: $\delta^*(ww')$ cannot be computed from $\delta^*(w)$ and $\delta^*(w')$. Thus we introduce the language accepted by monoids, which can be seen as a form of automata with fully compositional behaviour.

Let $\mathcal{M} = (M, \cdot, 1)$ be a monoid, $P \subseteq M$, and $h : \Sigma^* \to M$ be a homomorphism from the free monoid $((\Sigma^*, \cdot, \varepsilon))$ into $\mathcal{M}$.

The language $L(\mathcal{M}, P, h)$ is the set of all words that are mapped to P under h, that is

$$L(\mathcal{M}, P, h) = \{w \in \Sigma^* \mid h(w) \in P\}$$

We say that L can be recognised by $\mathcal{M}$ if there exists P and h such that $L = L(\mathcal{M}, P, h)$.

Definition: Monoids

A *monoid* is an algebraic structure $(M, \cdot, 1)$ with

- an associative operation $\cdot : M \times M \to M$

$$\forall x, y, z \in M : (x \cdot y) \cdot z = x \cdot (y \cdot z)$$

- a neutral element $1 \in M$

$$\forall x \in M : 1 \cdot x = x \cdot 1 = x$$

Definition: Homomorphisms

Let $\mathcal{M} = (M, \cdot_M, 1_M)$ and $\mathcal{N} = (N, \cdot_N, 1_N)$ be monoids.

A *monoid homomorphism* is a mapping $h : M \to N$ such that

1. $\forall x, y \in M : h(x \cdot_M y) = h(x) \cdot_N h(y)$
2. $h(1_M) = 1_N$

Theorem: Monoid Recognisability and Regularity

A language L is regular iff it can be recognised by a finite monoid.

Monoids → Automata

Let $L = L(\mathcal{M}, P, h)$ with $\mathcal{M} = (M, \cdot, 1)$. View M as the states of an automaton, P as final states, the neutral element 1 as the initial state and use h and $\cdot$ to define the transitions: $\delta(m, a) = m \cdot h(a)$, where $a \in \Sigma$ and $m \in M$.

For the resulting DFA $\mathcal{A} = (M, \Sigma, 1, \delta, P)$ we have $\delta^*(w) = h(w)$ and thus $L(\mathcal{A}) = L(\mathcal{M}, P, h)$.

Automata → Monoids

Definition: Transition Monoid

Let $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ be a DFA for L. Then the *transition monoid* is $\mathcal{M}(\mathcal{A}) = (\{\, u^{\mathcal{A}} \mid u \in \Sigma^* \,\}, \circ, \mathrm{id}_Q)$.

For $u \in \Sigma^*$, the transformation on Q by u is $u^{\mathcal{A}} : Q \to Q$, $u^{\mathcal{A}}(q) \mapsto \delta^*(q, u)$.

Theorem: Syntactic Monoid vs. Transition Monoid

The syntactic monoid $\mathcal{M}(L)$ is isomorphic to the transition monoid $\mathcal{M}(\mathcal{A}_L)$ of the minimal DFA $\mathcal{A}_L$ for L. It is the (unique) smallest monoid that can recognise L.

We can also define a monoid for a language without reference to automata, using the *syntactic congruence* $\approx_L$: Let $L \subseteq \Sigma^*$ and $u, v \in \Sigma^*$. Then $u \approx_L v$ iff $\forall x, y \in \Sigma^* : xuy \in L \Leftrightarrow xvy \in L$

Because of the isomorphism $[u]_{\approx_L} \mapsto u^{\mathcal{A}_L}$, we also know $\mathcal{M}(L)$ can be computed from an automaton for the language L.

Definition: Syntactic Monoid

The *syntactic monoid* $\mathcal{M}(L)$ is $\mathcal{M}(L) = (\Sigma^*_{/\approx_L}, \cdot, [\varepsilon]_{\approx_L})$, where

$$\Sigma^*_{/\approx_L} = \{[u]_{\approx_L} \mid u \in \Sigma^*\}$$

$$[u]_{\approx_L} \cdot [v]_{\approx_L} = [uv]_{\approx_L}$$

Definition: Group-free Monoids

A monoid $(M, \cdot, 1)$ is called *group-free* if there is no subset G of M with $|G| \geq 2$ and $1_G \in G$ such that $(G, \cdot, 1_G)$ forms a group.

5.3.3 Star-free Expressions

The *star-free expressions over* Σ are generalised regular expressions without the Kleene star.

For such an expression r let $L(r)$ be the language defined by r. A language is called star-free if it can be defined by a star-free expression.

Each star-free language $L \subseteq \Sigma^+$ is FO-definable: One can inductively give a formula $\varphi_r(x, y)$ for each star-free expression r such that $\underline{w} \models \varphi_r(a, b)$ iff the subword from a to b of w is in the language defined by r.

The inductive translation goes as following:

Definition: Star-free Expression Syntax

Atomic expressions

$\emptyset$, ε and
letters of Σ

Operators

$+$	$\cdot$	$\cap$	$\sim$
choice	*sequence*	*intersection*	*complement*

Star-free expression r	**FO formula** $\varphi_r(x, y)$
a	$P_a(x) \wedge x = y$
$\emptyset$ or ε	`false`
$s + t$	$\varphi_s(x, y) \vee \varphi_t(x, y)$
$s \cap t$	$\varphi_s(x, y) \wedge \varphi_t(x, y)$
$\sim s$	$\neg \varphi_s(x, y)$
$s \cdot t$	$\exists z \exists z' (x \leq z \wedge S(z, z') \wedge z' \leq y \wedge \varphi_s(x, z) \wedge \varphi_t(z', y))$ $(\vee \varphi_s(x, y)[$ **if** $\varepsilon \in L(t)])$ $(\vee \varphi_t(x, y)[$ **if** $\varepsilon \in L(s)])$

5.3.4 LTL-definability

Linear temporal logic (LTL) is a logic that allows referring to "time". Here, we imagine characters of a word $w \in \Sigma^+$ being read in time, so the "next time" will refer to the state where the next character is read.

- $w \models \top$ always.
- $w \models P_a$ iff $w = av$, $v \in \Sigma^*$.
- $w \models X\varphi$ iff $w = av$ for $a \in \Sigma$, $v \in \Sigma^+$ and $v \models \varphi$.
- $w \models \varphi U \psi$ iff $w = a_1 \dots a_n v$, $a_i \in \Sigma, v \in \Sigma^+$ with $v \models \psi$ and for all $1 \leq i \leq n$, $a_i \dots a_n v \models \varphi$.

Definition: Syntax of LTL

Atomic Propositions

$\top$ and P_a for $a \in \Sigma$

Logical Operators

$\wedge, \vee, \neg, \rightarrow, \leftrightarrow$

Temporal Modal Operators

$\varphi \; U \; \psi$, $X \; \varphi$
until *next*

Shorthands

$F \; \varphi := 1 \; U \; \varphi$ (*finally*),
$G \; \varphi := \neg F \; \neg\varphi$ (*globally*)

A language L is *LTL-definable* if there is an LTL formula φ such that $L = L(\varphi) := \{ w \in \Sigma^+ \mid w \models \varphi \}$. Every LTL-definable language is also FO-definable, as there is a simple translation of LTL operators to FO.

6 Automata for Finite Trees

6.1 Ranked Tree Automata

Definition: Ranked Alphabet

A *ranked alphabet* Σ is a finite alphabet where for each $a \in \Sigma$ a finite set $\mathrm{rk}(a) \subseteq \mathbb{N}$ of arities is fixed.

$$\Sigma_i = \{ a \in \Sigma \mid i \in \mathrm{rk}(a) \}$$
$$\Sigma = \Sigma_0 \cup \dots \cup \Sigma_m$$

Tree automata are a generalisation of word automata. They recognise trees over ranked alphabets (or unranked alphabets, see below).

We have two definitions for trees: The definition by induction and the labelled trees definition. They are equivalent, and either is used when convenient.

A subset $T \subseteq T_\Sigma$ is a *tree language* over Σ.

Definition: Tree over Σ

Definition by induction:
The set of Σ-*trees* is defined inductively by:

- each $a \in \Sigma_0$ is a Σ-tree
- $a(t_1, \dots, t_i)$ is a Σ-tree where $t_1, \dots, t_i$ are Σ-trees and $a \in \Sigma_i$

Labeled trees definition:
Σ-*labeled tree* is $t = (\mathrm{dom}_t, \mathrm{val}_t)$ with

- $\mathrm{dom}_t \subseteq (\mathbb{N}_+)^*$ such that
  - if $wv \in \mathrm{dom}_t$, then $w \in \mathrm{dom}_t$
  - if $w.i \in \mathrm{dom}_t$ for $i \in \mathbb{N}_+$, then $w.j \in \mathrm{dom}_t$ for all $1 \leq j < i$
- labels $\mathrm{val}_t : \mathrm{dom}_t \rightarrow \Sigma$ such that
  - For $\mathrm{val}_t(w) = a$, $a \in \Sigma_n$ such that w has n successors in dom_t ($w.i \in \mathrm{dom}_t$ iff $1 \leq i < n$).

The ith successor of a node u is $u.i$ or ui.

A simple usage of the inductive definition is $\mathrm{yield}(t)$, the word obtained by reading leaf labels from left to right in the tree t: $\text{yield}(a) = a$ for $a \in \Sigma_0$, $\text{yield}(a(t_1, \ldots, t_i)) = \text{yield}(t_1) \cdots \text{yield}(t_i)$ for $a \in \Sigma_i$.

A *deterministic tree automaton (DTA)* over the ranked alphabet Σ is the simple extension of a DFA for trees.

The automaton evaluates trees from bottom to top, basically folding subtrees into states [catamorphism]. A DTA accepts an input tree if the state reached at the tree root is a final state.

A tree language T is *regular* if there is a DTA $\mathcal{A}$ with $T = L(\mathcal{A})$.

Definition: Deterministic Tree Automaton

A DTA over $\Sigma = \Sigma_0 \cup \ldots \cup \Sigma_m$ is of the form $\mathcal{A} = (Q, \Sigma, \delta, F)$ with

- finite state set Q,
- final state set $F \subseteq Q$,
- transition function $\delta : \bigcup_{i=0}^{m} (Q^i \times \Sigma_i) \to Q$.

An important result is that the derivation tree of a context-free grammar is regular, although not all regular tree languages can be represented as a CFG.

The class of regular tree languages is closed under complement, union, and intersection.

A *nondeterministic tree automaton (NTA)* is the obvious extension of DTAs with nondeterminism.

We define a *run* of $\mathcal{A}$ on t as a mapping $\rho : \text{dom}_t \to Q$, such that

- for each leaf u: $(\text{val}_t(u), \rho(u)) \in \Delta$,
- for each u with successors: $(\rho(u.1), \ldots, \rho(u.i), \text{val}_t(u), \rho(u)) \in \Delta$.

Definition: Nondeterministic Tree Automaton

An NTA is of the form $\mathcal{A} = (Q, \Sigma, \Delta, F)$ like a DTA but with $\Delta \subseteq \bigcup_{i=0}^{m} (Q^i \times \Sigma_i \times Q)$

$\mathcal{A}$ accepts t if there is a run ρ of $\mathcal{A}$ with $\rho(\varepsilon) \in F$.

One can construct a DTA $\mathcal{A}'$ from a NTA $\mathcal{A}$ with $T(\mathcal{A}) = T(\mathcal{A}')$ by using the classical subset construction.

We can compute the set of reachable states for a NTA in polynomial time.

By computing the the set R of reachable states, we can decide the emptiness problem for NTAs. Depending on whether there are final states in R, the NTA is empty: $T(\mathcal{A}) = \emptyset$ iff $R \cap F = \emptyset$

Algorithm: NTA Reachability

Input: NTA $\mathcal{A} = (Q, \Sigma, \Delta, F)$.

Output: Set of reachable states R.

```
1. $R := \{q \in Q \mid \exists a \in \Sigma_0 : (a, q) \in \Delta\}$
2. Repeat until no new states are added:
   • For each $(q_1, \ldots, q_i, a, q) \in \Delta$
     – If $q_1, \ldots, q_i \in R$, then add $q$ to $R$.
```

6.1.1 Myhill-Nerode and Minimisation

To minimise DTAs, we want to generalise Myhill-Nerode equivalence to trees. However, while words have a notion of concatenation to represent what comes after the part of the word already read, the same is not true for trees.

To fix this problem, we introduce contexts or special trees: Trees that have a hole $\circ$ in place of one leaf. The set of these is called S_Σ.

The concatenation $t_1 \cdot t_2$ of such trees inserts t_2 in place of the leaf of t_1.

Then we can define Myhill-Nerode equivalence on tree automata as follows:

$$t_1 \sim_T t_2 \iff \forall s \in S_\Sigma \ (s \cdot t_1 \in T \iff s \cdot t_2 \in T)$$

The number of $\sim_T$-equivalence classes is called the *index* of T. T is regular iff its index is finite.

As in the word-case, we can use this equivalence to minimise DTA. Unlike the word case, unreachable states have to be purged first to avoid them influencing refinement.

In the refinement step, we separate p from p' if there are previously separated states q and q', a letter $a \in \Sigma_n$, and a sequence $p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n$ such that $\delta(p_1, \ldots, p_{i-1}, p, p_{i+1}, \ldots p_n, a) = q$ and $\delta(p_1, \ldots, p_{i-1}, p', p_{i+1}, \ldots, p_n, a) = q'$. Initially separate all pairs (q, q') with $(q \in F \Leftrightarrow q' \notin F)$.

6.2 Unranked Tree Automata

Unranked tree automata accept *unranked trees*.

A *nondeterministic bottom-up automaton for unranked trees* (NUTA) has a state set Q, a transition relation Δ and a set of final states F.

Since the number of successors of a node is unbounded, one specifies transitions by giving a regular language of state-sequences for the successors instead.

Definition: NUTA

A NUTA over Σ is of the form (Q, Σ, Δ, F):

- state set Q,
- transitions $\Delta \subseteq \text{Reg}(Q) \times \Sigma \times Q$,
- final states $F \subseteq Q$.

$\text{Reg}(Q)$ is the set of regular word languages over Q.

Definition: Unranked Trees

Definition by induction:
The set T_Σ of unranked trees over Σ is defined inductively by:

- Each $a \in \Sigma$ is a Σ-tree
- $a(t_1 \cdots t_n)$ is a Σ-tree for $a \in \Sigma$ and Σ-trees $t_1, \ldots, t_n$.

Labelled trees definition:
Σ*-labelled tree* is $t = (\text{dom}_t, \text{val}_t)$ with

- $\text{dom}_t \subseteq \mathbb{N}_+^*$ such that
  - if $wv \in \text{dom}_t$ then $w \in \text{dom}_t$
  - if $w.n \in \text{dom}_t$ for $n \in \mathbb{N}_+$ then $w.i \in \text{dom}_t$ for all $1 \leq i < n$
- labels $\text{val}_t : \text{dom}_t \to \Sigma$

The ith successor of a node u is $u.i$ or ui.

Definition: Runs and Languages of NUTAs

A run of a NUTA $\mathcal{A} = (Q, \Sigma, \Delta, F)$ on a tree $t = (\text{dom}_t, \text{val}_t)$ is a mapping $\rho : \text{dom}_t \to Q$ such that

- For each node $u \in \text{dom}_t$ with i successors, there is a transition $(L, \text{val}_t(u), \rho(u))$ with $\rho(u.1) \cdots \rho(u.i) \in L$.
- For each leaf u, there is a transition rule $(L, \text{val}_t(u), \rho(u))$ with $\varepsilon \in L$.

A run ρ is accepting if $\rho(\varepsilon) \in F$.

The set $T(\mathcal{A})$ is the set of trees t such that there is an accepting run of $\mathcal{A}$ on t.

A language $T \subseteq T_\Sigma$ is *regular* if there is a NUTA that accepts it.

We call a NUTA *normalised* if for each $a \in \Sigma$ and $q \in Q$ there is a unique transition (L, a, q). Each NUTA can be trivially normalised, since regular languages with $\emptyset$ and $\cup$ form a monoid.

We further call a NUTA *deterministic* (also abbreviated as DUTA), if for each two transitions (L_1, a, q_1), (L_2, a, q_2) either $q_1 = q_2$ or $L_1 \cap L_2 = \emptyset$.

We use the *subset construction*. From a normalised NUTA $\mathcal{A}$, we build a DUTA $(2^Q, \Sigma, \Delta', F')$. $\Delta' = \{ (K_{a,P}, a, P) \mid a \in \Sigma,\ P \subseteq Q \}$ where $P_1 \dots P_n \in K_{a,P}$ iff the states reachable in $\mathcal{A}$ from a selection of states of $P_1 \dots P_n$ are exactly the states in P. $F' = \{ P \subseteq Q \mid P \cap F \neq \emptyset \}$.

6.3 First-Child-Next-Sibling (FCNS)

We can encode unranked trees in *First-Child-Next-Sibling* (FCNS) encoding. A tree simply becomes a binary tree where each node has a choice: Either go to a child, or go to the next sibling.

Formally, we define the FCNS with the ranked alphabet $\Gamma = \Gamma_0 \cup \Gamma_2$ with $\Gamma_0 = \{ \# \}$ and $\Gamma_2 = \Sigma$.

$$\mathrm{fcns}(t_1 \dots t_n) := \begin{cases} \# & \text{if } n = 0 \\ a\ (\mathrm{fcns}(t'_1 \dots t'_m), \mathrm{fcns}(t_2 \dots t_n)) & \text{otherwise} \end{cases}$$
$$\text{where } t_1 = a(t'_1 \dots t'_m) \text{ if } n \geq 1$$

We also define $\mathrm{fcns}(T) = \{ \mathrm{fcns}(t) \mid t \in T \}$

One can also prove that $T \subseteq T_\Sigma$ is regular iff $\mathrm{fcns}(T)$ is regular. The translation between a NUTA for T and an NTA for $\mathrm{fcns}(T)$ is polynomial.

The emptiness problem and the membership problem for NUTAs are both decidable in polynomial time. The inclusion problem for NUTAs is decidable in exponential time and the inclusion problem for complete DUTAs is decidable in polynomial time.

6.4 Logic on Trees

Definition: Logic on Ranked Trees

Logic on ranked trees over an alphabet $\Sigma = \Sigma_0 \cup \ldots \cup \Sigma_n$ uses a signature with a unary relation P_a for each $a \in \Sigma$ and binary relations $S_1, \ldots, S_n$.

Then $P_a(x)$ represents "at x, the symbol is a" and $S_i(x, y)$ represents "y is the i-th successor of x".

From these, one can additionaly define shorthands $S(x, y)$ for "x is the parent of y" and $x \sqsubseteq y$ for "x is an ancestor of y".

A tree $t = (\mathrm{dom}_t, \mathrm{val}_t)$ is then represented as a structure $\mathfrak{A}$ with $A = \mathrm{dom}_t$, $P_a^{\mathfrak{A}} = \{ x \in \mathrm{dom}_t \mid \mathrm{val}_t(x) = a \}$, $S_i^{\mathfrak{A}} = \{ v.i \mid v \in \mathbb{N}^*, v.i \in \mathrm{dom}_t \}$.

As in the word-case, the set of MSO-definable tree languages is the same as the set of regular tree languages (the construction is largely analogous).

The set of FO-definable tree languages is however not yet characterised.

Definition: Logic on Unranked Trees

For unranked trees, instead of using an infinite signature containing all possible S_i for $i \in \mathbb{N}$, in addition to the P_a, S and $\sqsubseteq$, one introduces a symbol $<$, where $x < y$ means "x and y have the same parent and x occurs to the left of y".

In this case we have yet another equivalence theorem (MSO-definable = regular).

7 Algorithms for Pushdown Systems

A *pushdown system* (PDS) has two kinds of memory: A state from P, and a stack consisting of elements from the stack alphabet Γ.

Each transition is a four-tuple of (p, a, v, q). p is the current state and a the current topmost element of the stack. v is the word to be pushed onto the stack (arbitrarily many characters at once). Lastly, q is the next state. We also write a transition as $pa \to qv$.

A valid *step* is denoted $pw \vdash p'w'$ with $w = aw_0$, $w' = vw_0$ and $pa \to p'v \in \Delta$.

Multiple steps are denoted $pw \vdash^i p'w'$ with the obvious requirements.

The *configuration graph* represents the graph of configurations reachable by valid steps.

Definition: Pushdown System (PDS)

A PDS is of the form $\mathcal{P} = (P, \Gamma, \Delta)$ with

- state set P,
- stack alphabet Γ, and
- finite transition relation $\Delta \subseteq (P \times \Gamma \times \Gamma^* \times P)$ with rules (p, a, v, q), also written as $pa \to qv$.

A *configuration* is a pair (state, stack content). We abbreviate (q, u) as qu.

Definition: Configuration Graph

The *configuration graph* $G_{\mathcal{P}}$ of $\mathcal{P} = (P, \Gamma, \Delta)$ has

- $\{ c \mid c \in P\Gamma^* \}$ as set of vertices.
- $\{ (c_1, c_2) \mid c_1 \vdash c_2 \}$ as set of edges.

7.1 Reachability Analysis

We study the *general reachability problem*: Given a pushdown system $\mathcal{P}$, an initial configuration c_1, and a regular set $C \subseteq P\Gamma^*$ of target configurations, we want to decide whether there exists a configuration $c_2 \in C$ that is reachable from c_1.

For a PDS, the *P-automaton* is an NFA where the states P of the pushdown system are the initial states in the automaton. A P-automaton can accept a configuration pw if, starting from state p, w is accepted.

A P-automaton is *normalised* if states from P have no incoming transitions.

Definition: P-Automaton

For a PDS $\mathcal{P}$, a *P-automaton* is an NFA $\mathcal{A} = (Q, \Gamma, P, \Delta_{\mathcal{A}}, F)$ with $P \subseteq Q$. It accepts configurations $C(\mathcal{A}) = \{ pw \mid w \text{ is accepted from } p \}$.

We solve the *general reachability problem* given $\mathcal{P}, c_1, C$ as follows:

The solution has three steps and everything works out in polynomial time.

1. Construct the normalised P-automaton $\mathcal{A}$ that accepts C.
2. Compute $\mathcal{A}_{\text{pre}*}$ for $\mathcal{A}$ with the saturation algorithm.
3. Iff c_1 is accepted by $\mathcal{A}_{\text{pre}*}$, then $c_1 \vdash^* c_2$ for some $c_2 \in C$ $\; C(\mathcal{A}_{\text{pre}*}) = \text{pre}^*_{\mathcal{P}}(C(\mathcal{A}))$

Since the general reachability problem is decidable, so is the *simple reachability problem* (where $C = \{ c_2 \}$).

Definition: $\text{pre}^*_{\mathcal{P}}$

For a set of configurations C of a pushdown system $\mathcal{P}$: $\text{pre}^*_{\mathcal{P}}(C) = \{ c \in P\Gamma^* \mid \exists c' \in C \; c \vdash^* c' \}$

Algorithm: Saturation Algorithm

Input: PDS $\mathcal{P} = (P, \Gamma, \Delta)$ and
a normalised P-automaton $\mathcal{A}$.

Output: $\mathcal{A}_{\text{pre}^*}$.

```
1. If $pa \to p'v \in \Delta$ and $\mathcal{A} : p' \xrightarrow{v} q$,
     • add $(p, a, q)$ to $\mathcal{A}$.
2. Repeat 1 until $\mathcal{A}$ is not changed.
```

This also implies reachability problems for sequential programs with recursion over finite domains are decidable.

The saturation algorithm always terminates in polynomial time as no states are added at any point and at most $|P \times \Gamma \times Q|$ transitions can be added. For each transition added, we need to, at most, iterate over all transition rules of the PDS.

One can also go the other way and compute $\text{post}^*_{\mathcal{P}}(C) = \{ c \in P\Gamma^* \mid \exists c' \in C \; c' \vdash^* c \}$. The algorithm for this is not discussed in the lecture.

7.2 PDS with Several Stacks

To model concurrency, one can use PDS with multiple stacks.

Definition: n-pushdown System

An n-PDS has a transition relation $\Delta \subseteq Q \times \{ 1, \ldots, n \} \times \Gamma \times \Gamma^* \times Q$.

Configurations are of the form $(p, w_1, \ldots, w_n)$ with $p \in Q$, $w_i \in \Gamma^*$.

Transitions are as follows:

$$(p, w_1, \ldots, w_{i-1}, aw, w_{i+1}, \ldots, w_n) \vdash (q, w_1, \ldots, w_{i-1}, vw, w_{i+1}, \ldots, w_n)$$

if a transition rule (p, i, a, v, q) exists.

An n-PDS is *deterministic* if for each state q there is a unique $i \in \{ 1, \ldots, n \}$ such that all transitions from q modify stack i and for each symbol on the top of the stack, there is at most one applicable transition rule.

One can simulate a Turing machine with a deterministic 2-PDS by having the two stacks hold the contents of the tape to the left/right of the reading head (with the top of the stack being the side close to the reading head).

Thus the simple reachability problem for 2-PDS is undecidable (by reduction from the halting problem).

8 Register Machines and Communicating Automata

Definition: n-register Machine

An n-register machine is a program using variables $X_1 \ldots, X_n$ for natural numbers and the instruction set

- `INC` X_i
- `DEC` X_i
- `IF` $X_i = 0$ `GOTO` ℓ
- `GOTO` ℓ

where ℓ is a line number of the program.

An n-register machine can be simulated by an n-PDS where the height of stack i encodes the contents of register i and the state holds the current line number.

Thus the simple reachability problem for 1-register machines is decidable.

The simple reachability problem for 2-register machines is however undecidable:

Proof. One can reduce the simple reachability problem for n-register machines to that for 2-register machines and the simple reachability problem for n-PDS to that for $n+1$-register machines, as illustrated below.

n-PDS $\to$ $n+1$-register:

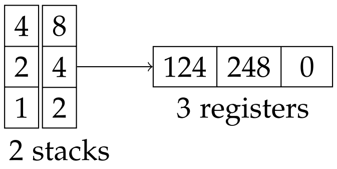

Figure 1: From 2-PDS to 3-register

Each stack is encoded as a decimal number where the top element of the stack is represent by the least significant digit. This way the top element can be easily identified using the modulo 10 operation. Its removal corresponds to a division by 10, pushing a number k to the stack corresponds to multiplication by 10 followed by an addition of k. The $n+1$ register is needed since multiplication and division require an auxiliary register. If the stack alphabet is different from $\{1, \ldots, 9\}$ a different base is needed.

n-register $\to$ 2-register:
The first register holds the encoding of the n registers of the original machine as a product of the first n prime numbers, with the register values stored as their exponents:

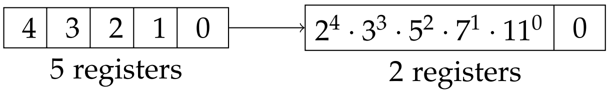

Figure 2: From 5-register to 2-register

The second register is again just a helper register for multiplication, division and storing division remainder. Checking whether register $i \in [1 \dots n]$ is equal to 0 corresponds to dividing our encoding by the ith prime number. If the division has a remainder the ith register value must have been 0. Indeed the encoding in our example $(2^4 \cdot 3^3 \cdot 5^2 \cdot 7^1 \cdot 11^0)$ is dividable by all prime numbers up to 7 but has a remainder of 1 when divided by 11. □

8.1 Data Words

Definition: Data Words

A *data word* is a word $w \in \Sigma^*$, $|w| = n$ together with *data* for each position, represented by a function $d : \{1, \dots, n\} \to \mathbb{N}$.

We extend logic to data-words in the following way:

In addition to the usual parts of FO/MSO over words, we add relations

- $d(x) = 0$
- $d(x) = d(y)$, $d(x) < d(y)$, $d(x) \leq d(y)$
- $d(x) = d(y) + 1$

for terms x, y.

Satisfiability for FO-sentences modulo data-words (given a sentence φ, does there exist a data-word w, d satisfying it?) is undecidable.

Proof. We can construct an FO-sentence describing valid, terminating computations of a 2-register machine. Thus we reduce the halting problem of 2-register machines to the satisfiability of FO-sentences over data-words.

This construction uses as an alphabet the line numbers of the register machine's program, together with a *padding symbol* #. A word describes the sequence of line numbers in the order they were visited with # following each line number. Pairs of data cells represent the two register values:

linenumber	1	#	j_1	#	j_2	#	$\cdots$
registers	0	0	m_1	n_1	m_2	n_2	$\cdots$

Figure 3: Potentially valid data-word for a 2-register machine

Our conjunction of FO-formulas asserts that we are initially in line 1 and both registers are equal to 0. Furthermore the word contains the line number of the *stop*-instruction at some point. The register machine's instructions are translated to FO-formula so that line numbers and register values of the data word change as expected from the simulated instructions: We are axiomatizing valid computations/runs of the given register machine. □

9 Communication via FIFO Channels

In an attempt to model communicating processes, we introduce the notion of *message passing automata*, wherein multiple automata can communicate messages of a finite alphabet via queues.

However, this model is too powerful to be effectively analysed: The simple reachability problem for message passing automata is undecidable even when restricted to a single automaton with a queue to itself.

Proof. One can simulate the run of a turing machine, using the queue to hold the contents of the tape. □

Definition: Message Passing Automaton

A *message passing automaton* is a structure $(CN, \Gamma, \mathcal{A}_1, \ldots, \mathcal{A}_n)$ with

- set of channels $CN \subseteq \{1, \ldots, n\}^2$

  (i, j) is the channel $\mathcal{A}_i$ *writes* and A_j *reads* from
- message alphabet Γ
- automata $\mathcal{A}_i = (Q_i, \Sigma_i, q_{0i}, \Delta_i, F_i)$ with transitions of the form
  - (p, a, q) with $p, q \in Q_i$, $a \in \Sigma_i$
  - $(p, m!j, q)$ with $p, q \in Q_i$, $m \in \Gamma$ and $(i, j) \in CN$

    (Write m to channel (i, j))

    Automata write from the left.
  - $(p, m?j, q)$ with $p, q \in Q_i$, $m \in \Gamma$ and $(j, i) \in CN$

    (If m is the first letter of channel (j, i), remove it and go to state q)

    Automata read from the right.

Configurations for message passing automata contain the states of each automaton as well as the contents of each channel.