Implementierung einer Entscheidungsprozedur für MSO-Logik über Bäumen

Martin Kretzschmar
Matrikelnummer 2728626

Technische Universität Dresden
Studiengang Informatik
Institut für Theoretische Informatik
Lehrstuhl Grundlagen der Programmierung

Verantwortlicher Hochschullehrer: Prof. Dr.-Ing. habil. Heiko Vogler
Betreuer: Dipl.-Inf. Enrico Bormann

Dezember 2005

Contents

Chapter 1

Introduction

In this project "Implementation of a Decision Procedure for MSO Logic over Trees" I implement a well-known [1], [2] translation from formulas of monadic second order logic over trees (*MSO logic*) to bottom-up tree automata in the programming language Haskell. What might sound like an exercise in automata theory actually has practical uses. In [3] a system for generating a syntax-based editor from a formal specification is introduced. The specification language that is used there is based on a combination of the specification paradigms of macro attribute grammars and MSO logic. Both paradigms are extended to actually enable their combination.

The generator system translates a specification into an executable editor. One part of this generator builds (extended) bottom-up tree automata for each (extended) MSO formula, such that each automaton recognizes exactly those trees that satisfy the formula for which the automaton was built.

Although the formulas in that system are a superset of the formulas with which I worked in this project, parts of my implementation correspond closely to constructions that are needed in the generator system and hopefully provide code or ideas that can be reused in the generator system.

This report will start with an overview of the definitions MSO logic and tree automata and related concepts as well as the connection between MSO formulas and tree automata. Then I will show how the relationship can be implemented naïvly. Later I will show data structures with which the relationship can be implemented more efficiently. Finally I will describe some details of the Haskell implementation.

Chapter 2

Preliminaries

In this chapter we will define the notions of formulas of monadic second order logic and bottom-up tree automata as well as the connection between the two. These are built upon the more fundamental concepts of *sorted sets*, *ranked alphabets*, *sorted terms* and *sorted trees*. Then we recall Büchi's Result for tree languages and the power set construction for bottom-up tree automata

2.1 Sorted Sets

Let K be a non-empty finite set of *sorts*. A *K-sorted set* is a pair (Ω, sort) where Ω is a set (*alphabet*) and $\mathit{sort} : \Omega \to K$ is a mapping which assigns a sort to every element (*symbol*) of Ω.

We write ω^κ for $\omega \in \Omega$ whose $\mathit{sort}(\omega)$ is κ. Similarly, we write $\Omega^\kappa = \{\omega^\kappa \in \Omega\}$ to denote the subset of all elements of Ω with sort κ. When *sort* can be inferred from the context, we abbreviate (Ω, sort) to Ω.

2.2 Sorted Ranked Alphabets

Let K be a non-empty finite set of sorts. A *K-sorted ranked alphabet* is a $(K^* \times K)$-sorted set (Σ, sort).

We define $\mathit{rank} : \Sigma \to \mathbb{N}$ to map every $\sigma^{(\kappa_1 \cdots \kappa_k, \kappa_0)}$ with $k \in \mathbb{N}$ and $\kappa_0, \ldots, \kappa_k \in K$ to $\mathit{rank}(\sigma) = k$.

If $\mathit{sort}(\sigma) = (\kappa_1 \cdots \kappa_k, \kappa_0)$, then we call κ_0 σ's *result sort*.

For a sorted ranked alphabet Σ, the maximum rank is denoted by $\mathit{maxrk}(\Sigma) = \mathit{max}\{\mathit{rank}(\sigma) \mid \sigma \in \Sigma\}$, the subset of *k-ary* symbols by $\Sigma^{(k)} = \{\sigma \in \Sigma \mid \mathit{rank}(\sigma) = k\}$.

2.3 Sorted Terms

Let Σ be a K-sorted ranked alphabet and M be a K-sorted set. The set of *K-sorted terms over* Σ *and* M, $T_\Sigma(M)$, is defined as the smallest set such that:

- for all sorts $\kappa \in K$, $M^\kappa \subseteq T_\Sigma(M)^\kappa$
- for all sorts $\kappa \in K$, $\Sigma^{(\varepsilon,\kappa)} \subseteq T_\Sigma(M)^\kappa$
- if $t_1^{\kappa_1}, \ldots, t_k^{\kappa_k} \in T_\Sigma(M)$ with $k > 0$ and $\sigma \in \Sigma^{(\kappa_1 \cdots \kappa_k, \kappa_0)}$, then $\sigma(t_1, \ldots, t_k) \in T_\Sigma(M)^{\kappa_0}$.

We write T_Σ as abbreviation for $T_\Sigma(\emptyset)$

2.4 Graphs and Sorted Trees

A *directed labelled graph over alphabets* Σ *and* Γ is a triple $g = (V_g, E_g, lab_g)$. V_g is a finite set of *nodes*, $E_g \subseteq V_g \times \Gamma \times V_g$ is a finite set of labelled *edges*, and $lab : V_g \to \Sigma$ is the *node-labelling function*. We use $GR(\Sigma, \Gamma)$ to denote the *set of graphs over* Σ *and* Γ.

Let Σ be a K-sorted ranked alphabet and $\Gamma_\Sigma = \{1, \ldots, maxrk(\Sigma)\}$. A *tree over* Σ is a graph $t = (V_t, E_t, lab_t)$ over Σ and Γ_Σ such that:

- there is exactly one node which does not have incoming edges (the *root*, denoted by $root(t)$)
- for every node $n \in V_t$ with incoming edges there is exactly one sequence $n_0, \ldots, n_h \in V_t$ such that $h > 0$, $n_0 = root(t)$, and $n_h = n$, and for every $j \in \{1, \ldots, h\}, (n_{j-1}, i_j, n_j) \in E_t$ with $i_j > 0$
- for every node $n \in V_t$: if $lab_t(n) = \sigma$, $\sigma \in \Sigma^{(k)}$, then n has k outgoing edges and for every $j \in \{1, \ldots, k\}$ there is exactly one $n_j \in V_t$ such that $(n, j, n_j) \in E_t$.

The sort of a node $n \in V_t$ is defined as the result sort of its label: $sort(n) = \kappa_0 \iff lab_t(n) = \sigma$ and $sort(\sigma) = (\kappa_1 \cdots \kappa_k, \kappa_0)$.

The *set of trees over* Σ is denoted by $TR(\Sigma)$.

2.5 Monadic Second Order Logic – Syntax

Given a K-sorted ranked alphabet Σ, a $\mathcal{P}(K)$-sorted set of free *node variables* (*first order variables*) $\mathcal{V}_1$, a $\mathcal{P}(K)$-sorted set of free *node set variables* (*second order variables*) $\mathcal{V}_2$, we can define the set $MSOL(\Sigma, \mathcal{V}_1, \mathcal{V}_2)$ of *MSO-Formulas* as the smallest set where:

- *true* is an *atomic formula*.
- $label_\sigma(x)$ is an atomic formula for every $x \in \mathcal{V}_1$ and $\sigma \in \Sigma^{(\kappa_1 \cdots \kappa_l, \kappa_0)}$ where $\kappa_0 \in sort(x)$
- $type_\kappa(x)$ is an atomic formula for every $x \in \mathcal{V}_1$ and $\kappa \in sort(x)$

- $x \in X$ is an atomic formula for every $x \in \mathcal{V}_1$ and $X \in \mathcal{V}_2$ where $sort(x) = sort(X)$
- $x == y$ is an atomic formula for every $x, y \in \mathcal{V}_1$ where $sort(x) = sort(y)$
- $edge_i(x, y)$ is an atomic formula for every $x, y \in \mathcal{V}_1$ and $1 \le i \le maxrk(\Sigma)$
- x over y is an atomic formula for every $x, y \in \mathcal{V}_1$
- $\neg\psi$ is a formula for every $\psi \in MSOL(\Sigma, \mathcal{V}_1, \mathcal{V}_2)$
- $\psi_1 \wedge \psi_2$ is a formula for every $\psi_1, \psi_2 \in MSOL(\Sigma, \mathcal{V}_1, \mathcal{V}_2)$
- $\exists x : \tilde{\kappa}.\psi$ is a formula for every $x \notin \mathcal{V}_1$, $\tilde{\kappa} \in \mathcal{P}(K)$, and $\psi \in MSOL(\Sigma, \mathcal{V}_1 \dot{\cup} \{x^{\tilde{\kappa}}\}, \mathcal{V}_2)$
- $\exists X : \tilde{\kappa}.\psi$ is a formula for every $X \notin \mathcal{V}_2$, $\tilde{\kappa} \in \mathcal{P}(K)$, and $\psi \in MSOL(\Sigma, \mathcal{V}_1, \mathcal{V}_2 \dot{\cup} \{X^{\tilde{\kappa}}\})$.

2.6 Monadic Second Order Logic – Semantics

Before we can define the semantics of Monadic Second Order Logic, we must define trees whose nodes have flags that represent the variables which occur in MSO formulas.

Let $\mathcal{V}_1$ and $\mathcal{V}_2$ be finite sets of first order and second order variables, respectively, and Σ a K-sorted ranked alphabet as above. The *alphabet extended by $\mathcal{V}_1$ and $\mathcal{V}_2$ (extended alphabet)* is the K-sorted ranked alphabet

$$\Sigma_{\mathcal{V}_1,\mathcal{V}_2} = \{(\sigma, U_1, U_2)^{(\kappa_1 \cdots \kappa_n, \kappa_0)} \mid \sigma \in \Sigma^{(\kappa_1 \cdots \kappa_n, \kappa_0)}, U_1 \subseteq \bigcup_{\tilde{\kappa} \ni \kappa_0} \mathcal{V}_1^{\tilde{\kappa}}, U_2 \subseteq \bigcup_{\tilde{\kappa} \ni \kappa_0} \mathcal{V}_2^{\tilde{\kappa}}\}$$

The set of *well-marked trees* is the set of all trees over the extended alphabet where each first order variable occurs exactly once:

$$WTR(\Sigma, \mathcal{V}_1, \mathcal{V}_2) = \{t \in TR(\Sigma_{\mathcal{V}_1,\mathcal{V}_2}) \mid \forall x \in \mathcal{V}_1 \exists! n \in V_t : x \in lab_t^{1st}(n)\}$$

where $lab_t^{1st}(n) = U_1$ if and only if $lab_t(n) = (\sigma, U_1, U_2)$ for some σ, U_1, and U_2. Similarly we define $lab_t^{\Sigma}(n) = \sigma$ and $lab_t^{2nd}(n) = U_2$.

With these definitions we can define how to interpret a formula over a well-marked tree. We define the *model operator* as a relation:

$$\models \subseteq WTR(\Sigma, \mathcal{V}_1, \mathcal{V}_2) \times MSOL(\Sigma, \mathcal{V}_1, \mathcal{V}_2)$$

inductively over the structure of MSO formulas: Let $t \in WTR(\Sigma, \mathcal{V}_1, \mathcal{V}_2)$, then

- $t \models true$
- $t \models label_\sigma(x)$ iff there is a node $n \in V_t$, such that $lab_t^{\Sigma}(n) = \sigma$ and $x \in lab_t^{1st}(n)$.

- $t \models type_\kappa(x)$ iff there is a node $n \in V_t$, such that $x \in lab_t^{1st}(n)$ and $sort(n) = \kappa$.
- $t \models x \in X$ iff there is a node $n \in V_t$, such that $x \in lab_t^{1st}(n)$ and $X \in lab_t^{2nd}(n)$.
- $t \models x == y$ iff there is a node $n \in V_t$, such that $x \in lab_t^{1st}(n)$ and $y \in lab_t^{1st}(n)$.
- $t \models edge_i(x, y)$ iff there are nodes $n_1, n_2 \in V_t$, such that $x \in lab_t^{1st}(n_1)$, $y \in lab_t^{1st}(n_2)$, and there is an edge $(n_1, i, n_2) \in E_t$.
- $t \models x \,\mathrm{over}\, y$ iff there are nodes $n_1, n_2 \in V_t$, such that $x \in lab_t^{1st}(n_1)$, $y \in lab_t^{1st}(n_2)$, and there is a sequence $a_1, \ldots, a_n$ of nodes with $n > 1$ such that $(a_i, j_i, a_{i+1}) \in E_t$ for every $i \in \{1, \ldots, n-1\}$ and some j_i with $a_1 = n_1$ and $a_n = n_2$.
- $t \models \neg\varphi$ iff $t \models \varphi$ is not true.
- $t \models \psi_1 \wedge \psi_2$ iff $t \models \psi_1$ and $t \models \psi_2$.
- $t \models \exists x : \tilde{\kappa}.\psi$ iff there is a node $n \in V_t$, $sort(n) \in \tilde{\kappa}$ such that $t' \models \psi$ where $t' \in WTR(\Sigma, \mathcal{V}_1 \dot{\cup} \{x^{\tilde{\kappa}}\}, \mathcal{V}_2)$ is derived from t by defining $lab_{t'}^{1st}(n) = lab_t^{1st}(n) \cup \{x^{\tilde{\kappa}}\}$ and keeping lab_t unchanged for all other nodes in t.
- $t \models \exists X : \tilde{\kappa}.\psi$ iff there is a set of nodes $\{n_1, \ldots, n_k\} \subseteq V_t$, $sort(n_i) \in \tilde{\kappa}$ for all $i \in \{1, \ldots, k\}$ such that $t' \models \psi$ where $t' \in WTR(\Sigma, \mathcal{V}_1, \mathcal{V}_2 \dot{\cup} \{X^{\tilde{\kappa}}\})$ is derived from t by defining $lab_{t'}^{2nd}(n_i) = lab_t^{2nd}(n_i) \cup \{X^{\tilde{\kappa}}\}$ for all $i \in \{1, \ldots, k\}$ and keeping lab_t unchanged for all other nodes in t.

We usually write the model operator in infix notation, as we did in the preceding definition.

Let $\varphi \in MSOL(\Sigma, \mathcal{V}_1, \mathcal{V}_2)$. The *tree language defined by* φ is the set of trees that are models for φ:

$$L_\varphi = \{t \in WTR(\Sigma, \mathcal{V}_1, \mathcal{V}_2) \mid t \models \varphi\}$$

Later sections will be easier to understand if we identify the elements of T_Σ with the elements of $TR(\Sigma_{\emptyset,\emptyset})$, and more generally those of $T_{\Sigma_{\mathcal{V}_1,\mathcal{V}_2}}$ with the elements of $TR(\Sigma_{\mathcal{V}_1,\mathcal{V}_2})$.

The term corresponding to a tree can be obtained by traversing the tree depth-first (observing the edge labels), starting at the root, and at each node writing down the node label, an opening parenthesis, traversing the child nodes, and writing a closing parenthesis. In this process, nodes as such disappear and only their labels remain.

For computing the tree corresponding to a term, nodes must be synthesized. One possible way is to use the paths (in Dewey-Notation) to all subterms of the term as the set of nodes. Then the label of a node is the leftmost symbol of the subterm that the node (which is a path to a subterm) points to.

From now on we will use $T_{\Sigma_{\mathcal{V}_1,\mathcal{V}_2}}$ and $TR(\Sigma_{\mathcal{V}_1,\mathcal{V}_2})$ interchangeably.

We define the *class of MSO-definable tree languages* as:

$$MSOL_\Sigma = \{L \subseteq T_\Sigma \mid \exists\varphi \in MSOL(\Sigma, \emptyset, \emptyset) : L = L_\varphi\}$$

2.7 Bottom-up Tree Automata

The model operator provides the declarative semantics for Monadic Second Order Logic. Operational semantics are specified using bottom-up tree automata. Here we define bottom-up tree automata.

A *deterministic bottom-up tree automaton* is a tuple

$$M = (Q, \Sigma, \delta, F)$$

with the following elements:

- Q is a finite set of *states*
- Σ is a ranked alphabet
- $\delta = \left\{\delta_\sigma^k\right\}_{k \geq 0, \sigma \in \Sigma^{(k)}}$ is a family of transition functions where $\delta_\sigma^k : Q^k \to Q$.
- $F \subseteq Q$ is a set of *final states*

Now we extend δ to $\hat{\delta} : T_\Sigma \to Q$, the function that computes the *run* of the automaton M over a term.

$\hat{\delta}$ is defined by structural induction on $t \in T_\Sigma$:

Let $t = \sigma(t_1, \dots, t_k)$, with $t_1^{\kappa_1}, \dots, t_k^{\kappa_k} \in T_\Sigma$, $k \geq 0$, $\sigma \in \Sigma^{(\kappa_1 \cdots \kappa_k, \kappa_0)}$, then

$$\hat{\delta}(t) = \delta_\sigma^k(\hat{\delta}(t_1), \dots, \hat{\delta}(t_k))$$

With this function $\hat{\delta}$ we can define the *language accepted by a deterministic bottom-up tree automaton* $M = (Q, \Sigma, \delta, F)$, $L(M)$, as:

$$L(M) = \{t \in T_\Sigma \mid \hat{\delta}(t) \in F\}$$

Again, we define a class of tree languages:

$$RECOG_\Sigma = \{L \subseteq T_\Sigma \mid \exists M : L = L(M)\},$$

the *class of recognizable tree languages*.

As in the case of word automata, there exists the notion of a nondeterministic bottom-up tree automaton.

A *nondeterministic bottom-up tree automaton* is a tuple

$$M = (Q, \Sigma, \delta, F)$$

where:

- Q is a finite set of states
- Σ is a ranked alphabet
- $\delta = \left\{\delta_\sigma^k\right\}_{\sigma \in \Sigma^{(k)}}$ is a family of transition functions where $\delta_\sigma^k : Q^k \to \mathcal{P}(Q)$

- $F \subseteq Q$ is a set of final states.

For nondeterministic bottom-up tree automata, the definition of $\hat{\delta} : T_\Sigma \to \mathcal{P}(Q)$ to compute the run on a tree $t \in T_\Sigma$ by structural induction is as follows:

- $t = \sigma^{(\varepsilon,\kappa_0)}$
$$\hat{\delta}(t) = \delta^0_\sigma$$

- $t = \sigma(t_1, \ldots, t_k)$, with $t_1^{\kappa_1}, \ldots, t_k^{\kappa_k} \in T_\Sigma$, $k > 0$, $\sigma \in \Sigma^{(\kappa_1 \cdots \kappa_k, \kappa_0)}$
$$\hat{\delta}(t) = \bigcup \left\{ \delta^k_\sigma(q_1, \ldots, q_k) \mid q_1 \in \hat{\delta}(t_1), \ldots, q_k \in \hat{\delta}(t_k) \right\}$$

The *language accepted by a nondeterministic bottom-up tree automaton* $M = (Q, \Sigma, \delta, F)$, $L(M)$, is defined as

$$L(M) = \{t \in T_\Sigma \mid \hat{\delta}(t) \cap F \neq \emptyset\}$$

2.8 Power Set Construction

In our work we will compose tree automata to build more complex tree automata. All of our compositions require deterministic tree automata as input. The constructions for quantified formulas produce nondeterministic tree automata. In order to use these automata in later compositions, we want to transform them into deterministic tree automata.

It has been shown that deterministic bottom-up tree automata are as powerful as nondeterministic bottom-up tree automata: For every nondeterministic bottom-up tree automaton M, there exists a deterministic bottom-up tree automaton N, such that $L(N) = L(M)$.

The proof for this proposition is constructive and uses the following idea: Using the *power set construction*, we construct the deterministic bottom-up tree automaton $N = (P, \Sigma, \nu, G)$ from the nondeterministic bottom-up tree automaton $M = (Q, \Sigma, \delta, F)$ in the following way:

- $P = \mathcal{P}(Q)$

- $\nu = \left\{ \nu^k_\sigma \right\}_{\sigma \in \Sigma^{(k)}}$ where ν^k_σ is defined for every $k \geq 0$, $\sigma \in \Sigma^{(k)}$ and $p_1, \ldots, p_k \in P$ as:
$$\nu^k_\sigma(p_1, \ldots, p_k) = \bigcup \left\{ \delta^k_\sigma(q_1, \ldots, q_k) \mid q_1 \in p_1, \ldots, q_k \in p_k \right\}$$

- $G = \{p \in P \mid p \cap F \neq \emptyset\}$

It is possible to prove by induction over the structure of $t \in T_\Sigma$, that for every such t, the run $\hat{\nu}$ of the deterministic automaton has the same result has the run $\hat{\delta}$ of the nondeterministic automaton. With this fact it can be shown that $t \in L(M) \iff \{t \in T_\Sigma \mid \hat{\delta}(t) \cap F \neq \emptyset\} \iff \{t \in T_\Sigma \mid \hat{\nu}(t) \in G\} \iff t \in L(N)$, that is M and N recognize the same language.

2.9 Büchi's Result

Büchi's Theorem, as applied to tree languages by Thatcher and Wright[2] is the theoretical background for operational semantics of MSO formulas.

The theorem states, that for every ranked alphabet Σ

$$RECOG_\Sigma = MSOL_\Sigma$$

Note that this does only make sense, because we defined $MSOL_\Sigma \subseteq T_\Sigma$ and not $MSOL_\Sigma \subseteq TR_\Sigma$, which would have been more straightforward in section 2.6.

To define an operational semantics we will use the technique that is used for the proof of $MSOL_\Sigma \subseteq RECOG_\Sigma$: For every MSO formula φ, there is a bottom-up tree automaton M, such that $L_\varphi = L(M)$. The automaton M will be built inductively according to the structure of φ. During the construction we will assume that the trees on which M runs are well-marked trees.

To define the actual construction, we will first describe it intuitively, and then formally. This section is a simplified form of section 6.2.2 in [4]. On the following pages, let $\varphi \in MSOL(\Sigma, \mathcal{V}_1, \mathcal{V}_2)$. In the inuitive description we will speak about nodes (as used in the graph representation) even though tree automata are defined for terms. Additionally the descriptions suggest that states are assigned to nodes in a particular sequence. These are only tools to help imagine how the automata work.

$\varphi = true$ (atomic formula)

Intuition Construct an automaton that accepts all of $T_{\Sigma_{\mathcal{V}_1,\mathcal{V}_2}}$. No distinction between final and non-final states is necessary and one state suffices. This state is a final state.

Formal Construction Construct a deterministic bottom-up tree automaton $M = (Q, \Sigma_{\mathcal{V}_1,\mathcal{V}_2}, \delta, F)$ where:

- $Q = \{1\}$
- for every $k \geq 0$, $(\gamma, U_1, U_2) \in \Sigma^{(k)}_{\mathcal{V}_1,\mathcal{V}_2}$, $q_1 = \ldots = q_k = 1$

$$\delta^k_{(\gamma,U_1,U_2)}(q_1, \ldots, q_k) = 1$$

- $F = \{1\}$

$\varphi = label_\sigma(x)$ (atomic formula)

Intuition Construct an automaton with two states, 0 and 1. Start at the leaves and run upwards. As soon as a node $n \in V_t$ is reached where $lab^\Sigma_t(n) = \sigma$ and $lab^{1st}_t(n) \ni x$, go to state 1, otherwise stay in state 0. Propagate 1 to the root of the tree. Accept in state 1.

Formal Construction Construct a deterministic bottom-up tree automaton $M = (Q, \Sigma_{\mathcal{V}_1,\mathcal{V}_2}, \delta, F)$ where:

- $Q = \{0, 1\}$
- for every $k \geq 0, (\gamma, U_1, U_2) \in \Sigma^{(k)}_{\mathcal{V}_1,\mathcal{V}_2}, q_1, \ldots, q_k \in Q$

```
  if $(\gamma = \sigma \wedge x \in U_1) \vee (\exists i \in \{1, \ldots, k\} : q_i = 1)$ then
    $\delta^k_{(\gamma,U_1,U_2)}(q_1, \ldots, q_k) = 1$
  else
    $\delta^k_{(\gamma,U_1,U_2)}(q_1, \ldots, q_k) = 0$
  end if
```

- $F = \{1\}$

$\varphi = type_\kappa(x)$ (atomic formula)

Intuition Similar to the previous case, construct an automaton with two states, 0 and 1. Start at the leaves and run upwards. As soon as a node $n \in V_t$ is reached where $sort(n) = \kappa$ and $lab^{1st}_t(n) \ni x$, go to state 1, otherwise stay in state 0. Propagate 1 to the root of the tree. Accept in state 1.

Formal Construction Construct a deterministic bottom-up tree automaton $M = (Q, \Sigma_{\mathcal{V}_1,\mathcal{V}_2}, \delta, F)$ where:

- $Q = \{0, 1\}$
- for every $k \geq 0, (\gamma, U_1, U_2) \in \Sigma^{(\kappa_1\cdots\kappa_k,\kappa_0)}_{\mathcal{V}_1,\mathcal{V}_2}, \kappa_0, \ldots, \kappa_k \in K, q_1, \ldots, q_k \in Q$

```
  if $(\kappa_0 = \kappa \wedge x \in U_1) \vee (\exists i \in \{1, \ldots, k\} : q_i = 1)$ then
    $\delta^k_{(\gamma,U_1,U_2)}(q_1, \ldots, q_k) = 1$
  else
    $\delta^k_{(\gamma,U_1,U_2)}(q_1, \ldots, q_k) = 0$
  end if
```

- $F = \{1\}$

$\varphi = x \in X$ (atomic formula)

Intuition Similar to the previous cases, construct an automaton with two states, 0 and 1. Start at the leaves and run upwards. As soon as a node $n \in V_t$ is reached where $lab^{1st}_t(n) \ni x$ and $lab^{2nd}_t(n) \ni X$, go to state 1, otherwise stay in state 0. Propagate 1 to the root of the tree. Accept in state 1.

Formal Construction Construct a deterministic bottom-up tree automaton $M = (Q, \Sigma_{\mathcal{V}_1,\mathcal{V}_2}, \delta, F)$ where:

- $Q = \{0, 1\}$
- for every $k \geq 0, (\gamma, U_1, U_2) \in \Sigma^{(k)}_{\mathcal{V}_1,\mathcal{V}_2}, q_1, \ldots, q_k \in Q$

```
  if $(x \in U_1 \wedge X \in U_2) \vee (\exists i \in \{1, \ldots, k\} : q_i = 1)$ then
    $\delta^k_{(\gamma,U_1,U_2)}(q_1, \ldots, q_k) = 1$
  else
    $\delta^k_{(\gamma,U_1,U_2)}(q_1, \ldots, q_k) = 0$
    end if
```

- $F = \{1\}$

$\varphi = x == y$ (atomic formula)

Intuition Similar to the previous cases, construct an automaton with two states, 0 and 1. Start at the leaves and run upwards. As soon as a node $n \in V_t$ is reached where $lab_t^{1st}(n) \ni x$ and $lab_t^{1st}(n) \ni y$, go to state 1, otherwise stay in state 0. Propagate 1 to the root of the tree. Accept in state 1.

Formal Construction Construct a nondeterministic bottom-up tree automaton $M = (Q, \Sigma_{\mathcal{V}_1,\mathcal{V}_2}, \delta, F)$ where:

- $Q = \{0, 1\}$
- for every $k \geq 0, (\gamma, U_1, U_2) \in \Sigma^{(k)}_{\mathcal{V}_1,\mathcal{V}_2}, q_1, \ldots, q_k \in Q$

```
    if $(x \in U_1 \wedge y \in U_1) \vee (\exists i \in \{1, \ldots, k\} : q_i = 1)$ then
      $\delta^k_{(\gamma,U_1,U_2)}(q_1, \ldots, q_k) = 1$
    else
      $\delta^k_{(\gamma,U_1,U_2)}(q_1, \ldots, q_k) = 0$
    end if
```

- $F = \{1\}$

$\varphi = edge_i(x, y)$ (atomic formula)

Intuition Construct an automaton with three states, 0, 1, and 2. Start at the leaves and run upwards. As soon as a node $n \in V_t$ is reached where $lab_t^{1st}(n) \ni y$, go to state 1. Go to state 2 if, at node $m \in V_t$, $lab_t^{1st}(n) \ni x$ and the ith component of the tuple of input states is 1. Propagate 2 up to the root. All other nodes are in state 0. Accept in state 2

Formal Construction Construct a deterministic bottom-up tree automaton $M = (Q, \Sigma_{\mathcal{V}_1,\mathcal{V}_2}, \delta, F)$ where:

- $Q = \{0, 1, 2\}$
- for every $k \geq 0, (\gamma, U_1, U_2) \in \Sigma^{(k)}_{\mathcal{V}_1,\mathcal{V}_2}, q_1, \ldots, q_k \in Q$

```
    if $i \leq k$ then
      if $(x \in U_1 \wedge q_i = 1) \vee (\exists j \in \{1, \ldots, k\} : q_j = 2)$ then
        $\delta^k_{(\gamma,U_1,U_2)}(q_1, \ldots, q_k) = 2$
      else if $y \in U_1$ then
        $\delta^k_{(\gamma,U_1,U_2)}(q_1, \ldots, q_k) = 1$
      else
        $\delta^k_{(\gamma,U_1,U_2)}(q_1, \ldots, q_k) = 0$
      end if
    else
      if $\exists j \in \{1, \ldots, k\} : q_j = 2$ then
        $\delta^k_{(\gamma,U_1,U_2)}(q_1, \ldots, q_k) = 2$
        else if $y \in U_1$ then
            $\delta^k_{(\gamma,U_1,U_2)}(q_1,\ldots,q_k) = 1$
        else
            $\delta^k_{(\gamma,U_1,U_2)}(q_1,\ldots,q_k) = 0$
        end if
    end if
```

- $F = \{2\}$

$\varphi = x \,\mathrm{over}\, y$ (atomic formula)

Intuition Similar to the previous case, construct an automaton with three states, 0, 1, and 2. Start at the leaves and run upwards. As soon as a node $n \in V_t$ is reached where $lab_t^{1st}(n) \ni y$, go to state 1. Propagate state 1 up. Go to state 2 if, at node $m \in V_t$, $lab_t^{1st}(n) \ni x$ and any component of the tuple of input states is 1. Propagate 2 up to the root. All other nodes are in state 0. Accept in state 2.

Formal Construction Construct a deterministic bottom-up tree automaton $M = (Q, \Sigma_{\mathcal{V}_1,\mathcal{V}_2}, \delta, F)$ where:

- $Q = \{0, 1, 2\}$
- for every $k \geq 0$, $(\gamma, U_1, U_2) \in \Sigma^{(k)}_{\mathcal{V}_1,\mathcal{V}_2}$, $q_1, \ldots, q_k \in Q$

```
    if $(x \in U_1 \wedge \exists i \in \{1,\ldots,k\} : q_i = 1) \vee (\exists i \in \{1,\ldots,k\} : q_i = 2)$
    then
        $\delta^k_{(\gamma,U_1,U_2)}(q_1,\ldots,q_k) = 2$
    else if $x \notin U_1 \wedge \exists i \in \{1,\ldots,k\} : q_i = 1$ then
        $\delta^k_{(\gamma,U_1,U_2)}(q_1,\ldots,q_k) = 1$
    else if $y \in U_1$ then
        $\delta^k_{(\gamma,U_1,U_2)}(q_1,\ldots,q_k) = 1$
    else
        $\delta^k_{(\gamma,U_1,U_2)}(q_1,\ldots,q_k) = 0$
    end if
```

- $F = \{1\}$

$\varphi = \neg\psi$ ($\psi \in MSOL(\Sigma, \mathcal{V}_1, \mathcal{V}_2)$)

Intuition Assume that there is a deterministic automaton M' for the formula ψ. M' accepts exactly the trees that M should not accept. The automaton M for $\neg\psi$ then should have the same transition function as M', but the non-final states of M' will become the final states of M.

Formal Construction Let $M' = (Q', \Sigma_{\mathcal{V}_1,\mathcal{V}_2}, \delta', F')$ be a deterministic bottom-up tree automaton such that $L(M') = L_\psi$. Construct a deterministic bottom-up tree automaton $M = (Q, \Sigma_{\mathcal{V}_1,\mathcal{V}_2}, \delta, F)$ where:

- $Q = Q'$
- $\delta = \delta'$

- $F = Q \setminus F'$

$\varphi = \psi_1 \wedge \psi_2$ $(\psi_1, \psi_2 \in MSOL(\Sigma, \mathcal{V}_1, \mathcal{V}_2))$

Intuition Assume that there are deterministic automata M_1 and M_2 such that $L(M_1) = L_{\psi_1}$ and $L(M_2) = L_{\psi_2}$. The automaton M for φ should accept exactly those trees that are accepted by both of M_1 and M_2. The automaton M can be constructed in such a way, that its run simulates the parallel runs of M_1 and M_2 on any well-marked tree: As states of M, use pairs of states, the first component being a state of M_1, the second one of M_2. To compute the transition function, apply the transition function of M_1 to the first components of the tuple of input states, and the transition function of M_2 to the second component of the tuple of input states. The resulting pair is the new state. Accept in those states, where the first component is a final state of M_1, and the second component is a final state of M_2.

Formal Construction Let $M_1 = (Q_1, \Sigma_{\mathcal{V}_1,\mathcal{V}_2}, \delta_1, F_1)$ and $M_2 = (Q_2, \Sigma_{\mathcal{V}_1,\mathcal{V}_2}, \delta_2, F_2)$ be deterministic automata such that $L(M_1) = L_{\psi_1}$ and $L(M_2) = L_{\psi_2}$. Construct a deterministic bottom-up tree automaton $M = (Q, \Sigma_{\mathcal{V}_1,\mathcal{V}_2}, \delta, F)$ where:

- $Q = Q_1 \times Q_2$
- for every $k \geq 0, (\gamma, U_1, U_2) \in \Sigma^{(k)}_{\mathcal{V}_1,\mathcal{V}_2}, q_{11}, \ldots, q_{1k} \in Q_1, q_{21}, \ldots, q_{2k} \in Q_2$

  $\delta^k_{(\gamma,U_1,U_2)}((q_{11}, q_{21}), \ldots, (q_{1k}, q_{2k})) = (\delta^k_{1(\gamma,U_1,U_2)}(q_{11}, \ldots, q_{1k}), \delta^k_{2(\gamma,U_1,U_2)}(q_{21}, \ldots, q_{2k}))$
- $F = F_1 \times F_2$

$\varphi = \exists x : \tilde{\kappa}.\psi$ $(\psi \in MSOL(\Sigma, \mathcal{V}_1 \dot{\cup} \{x^{\tilde{\kappa}}\}, \mathcal{V}_2))$

Intuition Asssume that we have a deterministic automaton M' such that for every $t' \in WTR(\Sigma, \mathcal{V}_1 \dot{\cup} \{x^{\tilde{\kappa}}\}, \mathcal{V}_2)$, $t' \in L(M') \iff t' \in L_\psi$. Given a tree $t \in WTR(\Sigma, \mathcal{V}_1, \mathcal{V}_2)$, we guess a node for x in t. M' can only run on a well-marked tree. Thus we have to guess x in such a way, that there is exactly one node with x. We can further restrict where to guess x: it can only occur at a node whose sort is in $\tilde{\kappa}$. Now construct a nondeterministic bottom-up tree automaton M. At each node, choose nondeterministically if x is at this node (but respect the restrictions on the positioning of x) and compute the successor state according to the transition function of M'. To respect the condition that x may occur exactly once in this tree, extend the states of M' with a second component that shows whether x has been guessed or not. Accept in states whose first component is a final state of M', and whose second component shows that x has been seen exactly once.

Formal Construction Let $M' = (Q', \Sigma_{\mathcal{V}_1 \dot{\cup} \{x\},\mathcal{V}_2}, \delta', F')$ be a deterministic bottom-up tree automaton such that for every $t' \in WTR(\Sigma, \mathcal{V}_1 \dot{\cup} \{x^{\tilde{\kappa}}\}, \mathcal{V}_2)$, $t' \in L(M') \iff t' \in L_\psi$. Construct a nondeterministic bottom-up tree automaton $M = (Q, \Sigma_{\mathcal{V}_1,\mathcal{V}_2}, \delta, F)$ where

- $Q = Q' \times \{0,1\}$
- for every $k \geq 0, (\gamma, U_1, U_2) \in \Sigma^{(\kappa_1 \cdots \kappa_k, \kappa_0)}_{\mathcal{V}_1 \dot\cup \{x\}, \mathcal{V}_2}, q_1, \ldots, q_k \in Q'$

```
    **if** $\delta'^k_{(\gamma,U_1,U_2)}(q_1,\ldots,q_k) = q$ and $x \notin U_1$ **then**
      $\delta^k_{(\gamma,U_1,U_2)}((q_1,0),\ldots,(q_k,0)) \ni (q,0)$
      **for every** $i \in \{1,\ldots,k\}$ **do**
        $\delta^k_{(\gamma,U_1,U_2)}((q_1,0),\ldots,(q_{i-1},0),(q_i,1),(q_{i+1},0),\ldots,(q_k,0)) \ni (q,1)$
      **end for**
    **else if** $\delta'^k_{(\gamma,U_1,U_2)}(q_1,\ldots,q_k) = q$ and $x \in U_1$ **then**
      $\delta_{(\gamma,U_1 \setminus \{x\},U_2)}((q_1,0),\ldots,(q_k,0)) \ni (q,1)$ {The condition for this
      case is written in such a way, that $x$ can only be guessed at a
      node whose sort is one of $x$'s sorts. $x \in U_1$ can only happen
      at exactly such nodes.}
    **end if**
```

- $F = F' \times \{1\}$

$\varphi = \exists X : \tilde{\kappa}.\psi \;\; (\psi \in MSOL(\Sigma, \mathcal{V}_1, \mathcal{V}_2 \dot\cup \{X^{\tilde{\kappa}}\}))$

Intuition Asssume that we have a deterministic automaton M' such that for every $t' \in WTR(\Sigma, \mathcal{V}_1, \mathcal{V}_2 \dot\cup \{X^{\tilde{\kappa}}\})$, $t' \in L(M') \iff t' \in L_\psi$. Given a tree $t \in WTR(\Sigma, \mathcal{V}_1, \mathcal{V}_2)$, we guess a set of nodes in t for X. We can restrict where to guess X: an X can only occur at a node whose sort is in $\tilde{\kappa}$. Now construct a nondeterministic bottom-up tree automaton M. At each node of a correct sort, choose nondeterministically if X is at this node and compute the successor state according to the transition function of M'. Accept in any final state of the automaton M'.

Formal Construction Let $M' = (Q', \Sigma_{\mathcal{V}_1 \dot\cup \{x\}, \mathcal{V}_2}, \delta', F')$ be a deterministic bottom-up tree automaton such that for every $t' \in WTR(\Sigma, \mathcal{V}_1, \mathcal{V}_2 \dot\cup \{X^{\tilde{\kappa}}\})$, $t' \in L(M') \iff t' \in L_\psi$. Construct a nondeterministic bottom-up tree automaton $M = (Q, \Sigma_{\mathcal{V}_1, \mathcal{V}_2}, \delta, F)$ where

- $Q = Q'$
- for every $k \geq 0, (\gamma, U_1, U_2) \in \Sigma^{(\kappa_1 \cdots \kappa_k, \kappa_0)}_{\mathcal{V}_1, \mathcal{V}_2 \dot\cup \{X\}}, q_1, \ldots, q_k \in Q'$

```
    **if** $\delta'^k_{(\gamma,U_1,U_2)}(q_1,\ldots,q_k) = q$ **then**
      $\delta^k_{(\gamma,U_1,U_2 \setminus \{X\})}(q_1,\ldots,q_k) \ni q$ {Again, this is written in such a
      way, that $X$ is only guessed on nodes of a correct sort. $X$ is
      guessed on nodes where $X \in U_2$. This can only happen for
      nodes $n$ with $sort(n) = \kappa_0 \in \tilde{\kappa}$.}
    **end if**
```

- $F = F'$

Chapter 3

Constructive Algorithms

The relationship between MSO formulas and bottom-up tree automata was stated in a compact, declarative way in section 2.9. A way that is not easily implementable in a program. We're going to reformulate them in a less declarative way which translates more easily to a real implementation. Data structures that represent sets and transition functions will be discussed in chapter 4. The *empty_transitions* function returns the family of transition functions (the "δ" of a bottom-up tree automaton) that is undefined for all symbols of the extended alphabet and tuples of states.

3.1 Atomic Formulas

The automaton construction for the formula $\varphi = true$ is very simple:

```
$Q := \{1\}$
$F := \{1\}$
$\delta := empty\_transitions(\Sigma_{\mathcal{V}_1,\mathcal{V}_2})$
**for every** $k \in \{rank(\sigma) \mid \sigma \in \Sigma\}$ **do**
  $tuple := (1, \ldots, 1) \in Q^k$
  **for every** $\sigma \in \Sigma^{(k)}$ **do**
    $(\kappa_1 \cdots \kappa_k, \kappa_0) := sort(\sigma)$
    **for every** $U_1 \subseteq \{v \in \mathcal{V}_1 \mid \kappa_0 \in sort(v)\}$ **do**
      **for every** $U_2 \subseteq \{V \in \mathcal{V}_2 \mid \kappa_0 \in sort(V)\}$ **do**
        $\delta^k_{(\sigma,U_1,U_2)}(tuple) := 1$
      **end for**
    **end for**
  **end for**
**end for**
```

We know the sets of states Q and final states F for atomic formulas in advance (this is true for all atomic formulas). We build the transition function pointwise and start with an empty transition function. We loop over all possible ranks, tuples of input symbols and symbols of the extended alphabet $\Sigma_{\mathcal{V}_1,\mathcal{V}_2}$ and define δ at each point. In this case there is exactly one tuple of input symbols for each rank.

The automaton for the formula $\varphi = label_\sigma(x)$ is slightly more interesting.

```
$Q := \{0,1\}$
$F := \{1\}$
$\delta := empty\_transitions(\Sigma_{\mathcal{V}_1,\mathcal{V}_2})$
**for every** $k \in \{rank(\sigma) \mid \sigma \in \Sigma\}$ **do**
  $tuple := (0,\ldots,0) \in Q^k$
  $tuples' := \{(1,0,\ldots,0),(0,1,0,\ldots,0),\ldots,(0,\ldots,0,1)\} \subseteq Q^k$
  **for every** $\gamma \in \Sigma^{(k)}$ **do**
    $(\kappa_1 \cdots \kappa_k, \kappa_0) := sort(\sigma)$
    **for every** $U_1 \subseteq \{v \in \mathcal{V}_1 \mid \kappa_0 \in sort(v)\}$ **do**
      **for every** $U_2 \subseteq \{V \in \mathcal{V}_2 \mid \kappa_0 \in sort(V)\}$ **do**
        **if** $\gamma = \sigma$ **then**
          $\delta^k_{(\gamma,U_1,U_2)}(tuple) := 1$
        **else**
          $\delta^k_{(\gamma,U_1,U_2)}(tuple) := 0$
        **end if**
        **for every** $tuple' \in tuples'$ **do**
          $\delta^k_{(\gamma,U_1,U_2)}(tuple') := 1$
        **end for**
      **end for**
    **end for**
  **end for**
**end for**
```

Here we see that often not all tuples from Q^k have to be generated. In this case we know that there can be at most one 1 in any tuple of input states. More than one 1 implies more than one x in the input tree. This is not possible in well-marked trees which was the assumption in section 2.9.

As a last example of an atomic formula, we show the construction for the formula $x \in X$.

```
$Q := \{0,1\}$
$F := \{1\}$
$\delta := empty\_transitions(\Sigma_{\mathcal{V}_1,\mathcal{V}_2})$
**for every** $k \in \{rank(\sigma) \mid \sigma \in \Sigma\}$ **do**
  $tuple := (0,\ldots,0) \in Q^k$
  $tuples' := \{(1,0,\ldots,0),(0,1,0,\ldots,0),\ldots,(0,\ldots,0,1)\} \subseteq Q^k$
  **for every** $\gamma \in \Sigma^{(k)}$ **do**
    $(\kappa_1 \cdots \kappa_k, \kappa_0) := sort(\gamma)$
    **if** $\kappa_0 \in sort(x) \wedge \kappa_0 \in sort(X)$ **then**
      **for every** $U_1 \subseteq \{v \in \mathcal{V}_1 \mid \kappa_0 \in sort(v)\}$ **do**
        **for every** $U_2 \subseteq \{V \in \mathcal{V}_2 \mid \kappa_0 \in sort(V)\}$ **do**
          **if** $x \in U_1 \wedge X \in U_2$ **then**
```

```
            $\delta^k_{(\gamma,U_1,U_2)}(tuple) := 1$
          else
            $\delta^k_{(\gamma,U_1,U_2)}(tuple) := 0$
          end if
          for every $tuple' \in tuples'$ do
            $\delta^k_{(\gamma,U_1,U_2)}(tuple') := 1$
          end for
        end for
      end for
    else
      for every $U_1 \subseteq \{v \in \mathcal{V}_1 \mid \kappa_0 \in sort(v)\}$ do
        for every $U_2 \subseteq \{V \in \mathcal{V}_2 \mid \kappa_0 \in sort(V)\}$ do
          $\delta^k_{(\gamma,U_1,U_2}(tuple) := 0$
          for every $tuple' \in tuples'$ do
            $\delta^k_{(\gamma,U_1,U_2)}(tuple') := 1$
          end for
        end for
      end for
    end if
  end for
end for
```

What might seem redundant at this point is "**if** $\kappa_0 \in sort(x) \wedge \kappa_0 \in sort(X)$." The algorithm would behave the same if this test was removed along with its **else** branch (U_1 and U_2 can contain x and X, resp., only if the sort of γ matches). Here the extra test causes the test in the innermost loop to be skipped in cases where we know its result without looking at U_1 and U_2. In chapter 4 we will see that the extra test allows bigger optimisations to be made.

3.2 Negation

There is nothing to be made more constructive than what is already in section 2.9 for negation.

3.3 Identifying Unreachable States

Let $M = (Q, \Sigma, \delta, F)$ be a deterministic bottom-up tree automaton. A state $q \in Q$ is *reachable*, iff there is a $t \in T_\Sigma$ such that $\hat{\delta}(t) = q$. The automaton M is *connected* iff every state $q \in Q$ is reachable.

The constructions for formulas involving conjunction or quantification produce automata which may contain states that are not reachable. To save space in the representation of the resulting automata, and to save runtime during later constructions that build upon these automata, we remove unreachable states.

We implement this by using a generalisation of the algorithm for removing non-reachable states from word automata. The input for this algorithm is a deterministic bottom-up tree automaton $M = (Q, \Sigma_{\mathcal{V}_1,\mathcal{V}_2}, \delta, F)$.

```
$Q_{reach} := \emptyset$ {states known to be reachable}
$Q_{new} := \emptyset$ {states newly discovered to be reachable}
$Q_{recent} := \emptyset$ {states discovered to be reachable in the previous iteration}
**for every** $(\alpha, U_1, U_2) \in \Sigma^{(0)}_{\mathcal{V}_1,\mathcal{V}_2}$ **do**
  $Q_{new} := Q_{new} \cup \delta^0_{(\alpha,U_1,U_2)}$ {$\star$}
**end for**
**while** $Q_{new} \neq \emptyset$ **do**
  $Q_{recent} := Q_{new}$
  $Q_{reach} := Q_{reach} \cup Q_{new}$
  $Q_{new} := \emptyset$
  **for every** $k \in \{rank(\sigma) \mid \sigma \in \Sigma\} \setminus \{0\}$ **do**
    **for every** $tuple \in (Q_{reach})^k$ **do**
      **if** $tuple \notin (Q_{reach} \setminus Q_{recent})^k$ **then**
        **for every** $(\sigma, U_1, U_2) \in \Sigma^{(k)}_{\mathcal{V}_1,\mathcal{V}_2}$ **do**
          $q := \delta^k_{(\sigma,U_1,U_2)}(tuple)$ {$\star\star$}
          **if** $q \notin Q_{reach}$ **then**
            $Q_{new} := Q_{new} \cup \{q\}$
          **end if**
        **end for**
      **end if**
    **end for**
  **end for**
**end while**
```

This algorithm terminates because the range of δ is finite. In each iteration at least one different state from the image is added to Q_{reach}. Thus there will be an iteration in which Q_{new} must stay empty, and the **while** loop terminates.

We keep track of three sets of states: states that we found to be "reachable," states that we found during the current iteration, and states that we found during the previous iteration. In "iteration 0" (before the main **while** loop begins), we process the nullary symbols to get an initial set of reachable states.

In our main loop we generate tuples of reachable states for every possible rank (except 0). To save unnecessary recomputations, only those tuples are processed, that have at least one state which was only discovered in the previous iteration. Other tuples have been processed earlier. Here, "processing a tuple" means computing the successor state for this tuple and all possible input symbols of the correct rank. If this successor state is not in Q_{reach}, then it is a "new" state in this iteration.

After termination of the loop, Q_{reach} contains the set of "reachable" states.

This algorithm does not eliminate all unreachable states from an automaton if there are more sorts than one sort.

In the actual implementation, we use an extended version of the algorithm that computes the new transition function simultaneously:

- During initialization, set $\delta_{reach} := \mathit{empty\_transitions}(\Sigma_{\mathcal{V}_1,\mathcal{V}_2})$,
- in line $\star$, update δ_{reach} with $(\delta_{reach})^0_{(\alpha,U_1,U_2)} := \delta^0_{(\alpha,U_1,U_2)}$,
- in line $\star\star$, update δ_{reach} with $(\delta_{reach})^k_{(\sigma,U_1,U_2)}(\mathit{tuple}) := q$

3.4 Complex Product Construction

To build the automaton for the conjunction of two formulas, it is possible to perform the two steps

1. Compute the complex product automaton as specified in section 2.9.
2. Apply the construction to remove unreachable states.

in sequence. A more efficient way is to perform the two steps in parallel. Let $M_1 = (Q_1, \Sigma_{\mathcal{V}_1,\mathcal{V}_2}, \delta_1, F_1)$ and $M_2 = (Q_2, \Sigma_{\mathcal{V}_1,\mathcal{V}_2}, \delta_2, F_2)$ be deterministic bottom-up tree automata.

Before we can show that algorithm, we need another helper function. It is $\mathit{unzip}_k : (S \times R)^k \to S^k \times R^k$, for every $k \geq 1$ and arbitrary sets S and R. It is defined as:

$$\mathit{unzip}_k((q_{11}, q_{21}), \ldots, (q_{1k}, q_{2k})) := ((q_{11}, \ldots, q_{1k}), (q_{21}, \ldots, q_{2k}))$$

We use unzip_k to implement this part of the formal construction:

$$\delta^k_{(\gamma,U_1,U_2)}((q_{11}, q_{21}), \ldots, (q_{1k}, q_{2k})) = (\delta^k_{1(\gamma,U_1,U_2)}(q_{11}, \ldots, q_{1k}), \delta^k_{2(\gamma,U_1,U_2)}(q_{21}, \ldots, q_{2k}))$$

where we have to extract two tuples of input states for two subautomata from one tuple of pairs of input states.

In the previous section we said that the construction can be extended to build the new transition function at the same time as the set of reachable states. We introduce a macro to show this for the complex product construction: $\mathit{record\_transition}((\sigma, U_1, U_2), k, \mathit{tuple}, \mathit{successor})$:

```
if $successor \notin Q_{reach}$ then
  $Q_{new} := Q_{new} \cup \{successor\}$
end if
$(\delta_{reach})^k_{(\sigma,U_1,U_2)}(tuple) := successor$
```

Finally, the algorithm for the combined construction:

```
$Q_{reach} := \emptyset$ {states known to be reachable}
$Q_{new} := \emptyset$ {states newly discovered to be reachable}
$Q_{recent} := \emptyset$ {states discovered to be reachable in the previous iteration}
$\delta_{reach} := empty\_transitions(\Sigma_{\mathcal{V}_1,\mathcal{V}_2})$ {new family of transition functions}
for every $(\alpha, U_1, U_2) \in \Sigma^{(0)}_{\mathcal{V}_1,\mathcal{V}_2}$ do
```

```
    $record\_transition((\alpha, U_1, U_2), 0, (), ((\delta_1)^0_{(\alpha,U_1,U_2)}, (\delta_2)^0_{(\alpha,U_1,U_2)}))$
  **end for**
  **while** $Q_{new} \neq \emptyset$ **do**
    $Q_{recent} := Q_{new}$
    $Q_{reach} := Q_{reach} \cup Q_{new}$
    $Q_{new} := \emptyset$
    **for every** $k \in \{rank(\sigma) \mid \sigma \in \Sigma\} \setminus \{0\}$ **do**
      **for every** $tuple \in (Q_{reach})^k$ **do**
        **if** $tuple \notin (Q_{reach} \setminus Q_{recent})^k$ **then**
          $(tuple_1, tuple_2) = unzip_k(tuple)$
          **for every** $(\sigma, U_1, U_2) \in \Sigma^{(k)}_{\mathcal{V}_1,\mathcal{V}_2}$ **do**
            $q_1 := (\delta_1)^k_{(\sigma,U_1,U_2)}(tuple_1)$
            $q_2 := (\delta_2)^k_{(\sigma,U_1,U_2)}(tuple_2)$
            $record\_transition((\sigma, U_1, U_2), k, tuple, (q_1, q_2))$
          **end for**
        **end if**
      **end for**
    **end for**
  **end while**
```

Compared to the previous construction, the lines where δ is applied to an input symbol and a tuple of input states have changed. Here we use the definition of δ for the formula $\psi_1 \wedge \psi_2$ directly.

This construction has to be extended in the same way as the previous construction to build not only the set of states, but also the transition function for the product automaton.

To compute the set of final states for the product automaton, we start with the set Q_{reach}. Those elements where the first component is an element of F_1 and the second component is from F_2 are the new final states. In other words, compute $Q_{reach} \cap (F_1 \times F_2)$ by filtering elements.

3.5 First Order Quantification

To construct a deterministic tree automaton for a formula $\exists x : \tilde{\kappa}.\psi$, when the input is the automaton $M' = (Q', \Sigma_{\mathcal{V}_1 \dot{\cup} \{x\}, \mathcal{V}_2}, \delta', F')$ for the formula ψ, the tasks are:

1. Apply the construction from section 2.9.
2. Apply the power set construction from section 2.8.
3. Eliminate unreachable states, as explained in section 3.3.

Again, we perform all these tasks in parallel, using an algorithm of the same structure as the basic algorithm from section 3.3.

Recall that the construction for first order quantification extends the states of M' with an additional flag. Additionally the algorithm is an implementation of the power set construction. As a result, elements of Q_{reach}, Q_{new}, and Q_{recent} are subsets of $Q' \times \{0,1\}$. Then the type of the transition functions looks like this:

$$(\delta_{reach})^k_{(\sigma,U_1,U_2)} : (\mathcal{P}(Q' \times \{0,1\}))^k \to \mathcal{P}(Q' \times \{0,1\})$$

The type of the transition functions of M' however is simply

$$(\delta')^k_{(\sigma,U_1,U_2)} : (Q')^k \to Q'$$

As part of the algorithm for first order quantification we use a helper function $decompose\_tuple_k : (\mathcal{P}(Q' \times \{0,1\}))^k \to \mathcal{P}((Q')^k \times \{0,1\})$

$$\begin{aligned} decompose\_tuple_k(p_1,\dots,p_k) := \{((q'_1,\dots,q'_k), flag) \mid \\ (q'_1, flag_1) \in p_1,\dots,(q'_k, flag_k) \in p_k, flag = \sum_{i=1}^{k} flag_i, flag \le 1\} \end{aligned}$$

This combines the following ideas:

- As part of the power set construction we have to compute the union of all $\left\{\delta^k_\sigma(q_1,\dots,q_k) \mid q_1 \in p_1,\dots,q_k \in p_k\right\}$ for every input tuple $(p_1,\dots,p_k) \in Q^k_{reach}$. You can recognize a part of this construction in *decompose_tuple* if you substitute q_i with $(q'_i, flag_i)$.
- We know that our input tree is a well-marked tree. The construction for first order quantification uses the $flag$s to implement this restriction. We have to consider only those input tuples with at most one non-zero flag. This is checked with the sum.
- The construction for first order quantification in section 2.9 only needs to know if there was exactly on flag. Thus we output only the sum of all flags.

The complete algorithm for the first order quantification construction with parallel power set construction follows. The input is the automaton M' and the quantified variable x and its type $\tilde{\kappa}$

```
$Q_{reach} := \emptyset$ {states known to be reachable}
$Q_{new} := \emptyset$ {states newly discovered to be reachable}
$Q_{recent} := \emptyset$ {states discovered to be reachable in the previous iteration}
$\delta_{reach} := empty\_transitions(\Sigma_{\mathcal{V}_1,\mathcal{V}_2})$ {new family of transition functions}
**for every** $(\alpha, U_1, U_2) \in \Sigma^{(0)}_{\mathcal{V}_1,\mathcal{V}_2}$ **do**
  $(\varepsilon, \kappa_0) := sort(\alpha)$
  **if** $sort(\alpha) \in \tilde{\kappa}$ **then**
    $record\_transition((\alpha, U_1, U_2), 0, (), \{((\delta')^k_{(\alpha,U_1,U_2)}, 0), ((\delta')^k_{(\alpha,U_1\dot\cup\{x\},U_2)}, 1)\})$
  **else**
```

```
            $record\_transition((\alpha, U_1, U_2), 0, (), \{((\delta')^k_{(\alpha,U_1,U_2)}, 0)\})$
        end if
    end for
    while $Q_{new} \neq \emptyset$ do
        $Q_{recent} := Q_{new}$
        $Q_{reach} := Q_{reach} \cup Q_{new}$
        $Q_{new} := \emptyset$
        for every $k \in \{rank(\sigma) \mid \sigma \in \Sigma\} \setminus \{0\}$ do
            for every $tuple \in (Q_{reach})^k$ do
                if $tuple \notin (Q_{reach} \setminus Q_{recent})^k$ then
                    for every $(\sigma, U_1, U_2) \in \Sigma^{(k)}_{\mathcal{V}_1,\mathcal{V}_2}$ do
                        $(\kappa_1 \cdots \kappa_k, \kappa_0) := sort(\sigma)$
                        $q := \emptyset$
                        for every $(tuple', flag) \in decompose\_tuple_k(tuple)$ do
                            $q := q \cup \{((\delta')^k_{(\sigma,U_1,U_2)}(tuple'), flag)\}$
                            if $flag = 0 \wedge \kappa_0 \in \tilde{\kappa}$ then
                                $q := q \cup \{((\delta')^k_{(\sigma,U_1 \dot\cup \{x\},U_2)}(tuple'), 1)\}$
                            end if
                        end for
                        $record\_transition((\sigma, U_1, U_2), k, tuple, q)$
                    end for
                end if
            end for
        end for
    end while
```

The structure is the same as that of the previous two algorithms. The difference lies in the inner loop where we incrementally compute the successor state (which is a set) for a tuple of input states for the new automaton. Recall that the components of such a tuple are sets whose elements are pairs of states from Q' and a flag. We use *decompose_tuple* to get all tuples for which we have to compute the successor state in the automaton M'. Together with each such tuple we get a *flag* that indicates if x was already guessed.

In every case we compute the successor state with the old transition function without guessing x at the current node. If the flag indicates that x has not been guessed yet and the sort of the current node allows guessing of x, we additionally compute the successor state in M' with guessing x at the current node.

The set of final states is computed as $F := \{q \mid q \in Q_{reach}, q \cap (F' \times \{1\}) \neq \emptyset\}$. This is the straightforward combination of the construction for first order quantification from section 2.9 and the power set construction from section 2.8.

3.6 Second Order Quantification

The implementation of second order quantification is very similar to that of first order quantification. It is simpler than first order quantification because there is no need for the *flag* that were needed to check that a first order variable is guessed exactly once. Second order variables can be guessed any number of times.

For the formula $\exists X : \tilde{\kappa}.\psi$, we use an automaton $M' = (Q', \Sigma_{\mathcal{V}_1,\mathcal{V}_2\dot{\cup}\{X\}}, \delta', F')$ for the formula ψ as input.

Instead of *decompose_tuple* we can use a simpler function, $sub\_tuples_k : (\mathcal{P}(Q'))^k \to \mathcal{P}((Q')^k)$ where

$$sub\_tuples_k(p_1, \ldots, p_k) := \{(q'_1, \ldots, q'_k) \mid q'_1 \in p_1, \ldots, q'_k \in p_k\}$$

```
$Q_{reach} := \emptyset$ {states known to be reachable}
$Q_{new} := \emptyset$ {states newly discovered to be reachable}
$Q_{recent} := \emptyset$ {states discovered to be reachable in the previous iteration}
$\delta_{reach} := empty\_transitions(\Sigma_{\mathcal{V}_1,\mathcal{V}_2})$ {new family of transition functions}
for every $(\alpha, U_1, U_2) \in \Sigma^{(0)}_{\mathcal{V}_1,\mathcal{V}_2}$ do
  $(\varepsilon, \kappa_0) := sort(\alpha)$
  if $\kappa_0 \in \tilde{\kappa}$ then
    $record\_transition((\alpha, U_1, U_2), 0, (), \{(\delta')^k_{(\alpha,U_1,U_2)}, (\delta')^k_{(\alpha,U_1,U_2)}\})$
  else
    $record\_transition((\alpha, U_1, U_2), 0, (), \{(\delta')^k_{(\alpha,U_1,U_2)}\})$
  end if
end for
while $Q_{new} \neq \emptyset$ do
  $Q_{recent} := Q_{new}$
  $Q_{reach} := Q_{reach} \cup Q_{new}$
  $Q_{new} := \emptyset$
  for every $k \in \{rank(\sigma) \mid \sigma \in \Sigma\} \setminus \{0\}$ do
    for every $tuple \in (Q_{reach})^k$ do
      if $tuple \notin (Q_{reach} \setminus Q_{recent})^k$ then
        for every $(\sigma, U_1, U_2) \in \Sigma^{(k)}_{\mathcal{V}_1,\mathcal{V}_2}$ do
          $(\kappa_1 \cdots \kappa_k, \kappa_0) := sort(\sigma)$
          $q := \emptyset$
          for every $(tuple') \in sub\_tuples_k(tuple)$ do
            $q := q \cup \{(\delta')^k_{(\sigma,U_1,U_2)}(tuple')\}$
            if $\kappa_0 \in \tilde{\kappa}$ then
              $q := q \cup \{(\delta')^k_{(\sigma,U_1,U_2\dot{\cup}\{X\})}(tuple')\}$
            end if
          end for
          $record\_transition((\sigma, U_1, U_2), k, tuple, q)$
        end for
      end if
```

```
        end for
      end for
    end while
```

This is exactly the same algorithm as in the previous section, only that all traces of $flag$s were removed (most notably in the test if guessing X is allowed) and *decompose_tuple* was replaced by *sub_tuples*.

The set of final states is computed as $F := \{q \mid q \in Q_{reach}, q \cap F' \neq \emptyset\}$.

Chapter 4

Data Structures

In the previous chapter it became clear which operations on automata, especially on the transition functions, are the most common. This has an impact on the choice of data structures.

4.1 Basic Data Structures

We do not implement basic data structures like sets ourselves, but use the types available in the Haskell Hierarchical Libraries [5], namely:

- `Set` (a set of values of one type) from the module `Data.Set` and
- `Map` (a map from keys to values, sometimes called "dictionary") from the module `Data.Map`

Both types are implemented using binary trees. According to their documentation, insertion and test for membership (lookup of the value for a key) in a `Set` (`Map`) have time complexity $O(\log n)$ where n is the cardinality of the `Set` (the cardinality of the set of keys).

We use `Maps` in the implementation of the transition functions and to map free variables to their sorts.

4.2 Ranked Alphabet and Sets of Free Variables

A data type for K-sorted ranked alphabets was provided as part of the `Input` module, type `RankAlph`. It is a list of (symbol name, symbol rank, symbol sort)-triples. We use an opaque type `RankedAlphabet` that wraps `RankAlph`, so that the simple list could be replaced by a different implementation if the need arises.

There is no type for extended ranked alphabets in the `Input` module. Our program doesn't represent the extended ranked alphabet as a single Haskell object either, but we always store an extended alphabet $\Sigma_{\mathcal{V}_1,\mathcal{V}_2}$ as the K-sorted ranked alphabet Σ and the $\mathcal{P}(K)$-sorted set $\mathcal{V}_1 \cup \mathcal{V}_2$.

The reason for representing $\Sigma_{\mathcal{V}_1,\mathcal{V}_2}$ as two parts are:

- In the course of constructing a tree automaton, the ranked alphabet Σ never changes. Whenever a quantification is translated, the set of free variables of the automaton for the subformula differs from the set of free variables of the automaton for the complete formula.
- There is no need to store $\Sigma_{\mathcal{V}_1,\mathcal{V}_2}$ as e.g. one big K-sorted ranked alphabet with all its symbols. Whenever we have to iterate over all elements of the extended ranked alphabet, we can use nested loops over Σ and $\mathcal{P}(\mathcal{V}_1)$ and $\mathcal{P}(\mathcal{V}_2)$ instead. The constructions for atomic formulas in section 3.1 are an example where we use such nested loops.

We store $\mathcal{V}_1 \cup \mathcal{V}_2$ as one set and not two. There was no construction that required us to use two separate sets. By using only one set, we cannot tell if a particular free variable is a first order variable or a second order variable. Neither can we handle cases where one variable name is used for both a first order variable and a second order variable. The first issue is not a problem, because we only need the information first order/second order at the time when we construct the automaton for a quantified formula. At that time, we can get this information from the formula itself. The second issue is a special case that we accept as a limitation for our system. We do not expect that anyone constructs such an input because it cannot be represented with the formal notation for MSO formulas either.

4.3 Transition Functions

The most interesting data structure of this project is that for representing the family of transition functions (δ).

The simplest possible way of representing the family of transition functions is to use a `Map` that maps every pair of symbols of the extended alphabet and tuple of input states to an output state.

When you look at the constructions in section 2.9, you see that all constructions for atomic formulas have some state (usually 0, 1 for the *true* automaton) that is the successor state for almost all inputs. We exploit this fact by storing that special state as a *default* successor state for all transitions. Only for those inputs where the successor state is different from the default value do we store an *explicit* key-value pair in the `Map`. Using default successor states is one of the ideas discussed in “Algorithms for Guided Tree Automata” [6].

We define a *map with default* as a pair $f = (f_{default}, f_{explicit})$ where $f_{default} \in B$ is a constant, the default value of f, and $f_{explicit} : A - \rightarrow B$ is a non-total function. Evaluating such a map with default f for input a, compute:

$$f(a) = \begin{cases} f_{explicit}(a) & \text{if it is defined,} \\ f_{default} & \text{otherwise} \end{cases}$$

In our pseudocode notation we'll write $f(default) := y$ to mean $f_{default} := y$ and $f(x) := z$ for $x \neq default$ to mean $f_{explicit}(x) := z$.

The other idea for optimization that we applied is divide-and-conquer. We are going to store δ as two halves. The first half maps a tuple of input states and an input symbol (without the sets U_1 and U_2 of variables) to an *intermediate result*, the second half maps the sets of first order and second order variables (U_1 and U_2) and an intermediate result to a successor state.

Formally, our internal representation of an automaton is different from the original definition of a bottom-up tree automaton. The original definition was $M = (Q, \Sigma_{\mathcal{V}_1,\mathcal{V}_2}, \delta, F)$. In our internal representation we use:

$$N = (Q, R, \Sigma_{\mathcal{V}_1,\mathcal{V}_2}, \mu, \nu, F)$$

where

- Q is the set of states, as before
- R is a finite set of intermediate results
- $\Sigma_{\mathcal{V}_1,\mathcal{V}_2}$ is a K-sorted extended ranked alphabet, as before
- $\mu = \left\{\mu^k_\sigma\right\}_{k\geq 0, \sigma \in \Sigma^{(k)}}$ is a family of functions $\mu^k_\sigma : Q^k \to R$, what we called "first half of δ", represented as maps with default
- $\nu = \{\nu_r\}_{r \in R}$ is a family of functions $\nu_r : \mathcal{P}(\mathcal{V}_1) \times \mathcal{P}(\mathcal{V}_2) \to Q$, the "second half of δ", represented as maps with default

We define $L(N) = L(N')$ where $N' = (Q, \Sigma_{\mathcal{V}_1,\mathcal{V}_2}, \delta', F)$ is a traditional bottom-up tree automaton where

$$(\delta')^k_{(\sigma,U_1,U_2)}(q_1, \ldots, q_k) = \nu_{\mu^k(\sigma,q_1,\ldots,q_k)}(U_1, U_2)$$

Note that the set of states and the set of final states are the same in both representations N and N' of the automaton.

When we are constructing automata for an MSO formula, our goal will be to choose R such that the representations of μ and ν as `Maps` with defaults are as small as possible.

4.3.1 Atomic Formulas

For atomic formulas we can find good sets R of intermediate results if we restate the constructions from section 2.9 in a very generalized informal way:

- $label_\sigma(x)$: if the node's label is (σ, U_1, U_2) for some U_1, U_2, then inspect U_1, else ignore U_1 and U_2.
- $type_\kappa(x)$: if the node's sort is κ, then inspect U_1, else ignore U_1 and U_2

- $x \in X$: if the node's sort matches one of x and X's sorts, then inspect U_1 and U_2, else ignore U_1 and U_2
- $x == y$: if the node's sort matches one of x and y's sorts, then inspect U_1, else ignore U_1 and U_2
- $edge_i(x.y)$: if one of the input states is 2: propagate this state; if the ith input state is 1: if the node's sort matches one of x's sorts, then inspect U_1; otherwise: if the node's sort matches one of y's sorts, then inspect U_1.

In all these cases, the function μ implements the "if" part of the description, while the function ν implements the "then" and "else" branches.

Finally here is the refined construction for the atomic formula $label_\sigma(x)$:

```
$Q := \{0, 1\}$
$R := \{0, 1, 2\}$
$\mu := empty\_\mu$
$\nu := empty\_\nu$
**for every** $k \in \{rank(\sigma) \mid \sigma \in \Sigma\}$ **do**
  $tuples' := \{(1, 0, \ldots, 0), (0, 1, 0, \ldots, 0), \ldots, (0, \ldots, 0, 1)\} \subseteq Q^k$
  **for every** $\gamma \in \Sigma^{(k)}$ **do**
    **if** $\gamma = \sigma$ **then**
      $\mu^k_\gamma(default) := 2$
    **else**
      $\mu^k_\gamma(default) := 0$
    **end if**
    **for every** $tuple' \in tuples'$ **do**
      $\mu^k_\gamma(tuple') := 1$
    **end for**
  **end for**
**end for**
$\nu_0(default) := 0$
$\nu_1(default) := 1$
$\nu_2(default) := 0$
**for every** $U_1 \in \mathcal{V}_1$ **do**
  **for every** $U_2 \in \mathcal{V}_2$ **do**
    **if** $x \in U_1$ **then**
      $\nu_2(U_1, U_2) := 1$
    **end if**
  **end for**
**end for**
```

Compare this to the construction from section 3.1. We are using the same set of states with the same intuitive idea.

For nodes where the symbol is not σ, we go to intermediate result 0. For this case, we know that for all possible values of U_1 and U_2, the successor state will be 0. Thus all we have to store for ν_0 is the default result 0. For any symbol and tuple of input states where one input state is 1, we know that the successor state will be 1. Again, all we have to store for ν_1 is the default result 1. Finally, for nodes with symbol σ we store 2 as the default value of μ_σ^k. In ν_2 we have to test if x is in the set of first order variables U_1 and in those cases explicitly go to state 1, otherwise we go to 0.

4.3.2 Conjunction

We want to benefit from the efficient storage of the transition functions not only when we handle atomic formulas, but also when we handle composite formulas.

- We would like to compute the new default results for the composed transition functions from the old default results. We want to process as few key-value pairs as possible, ideally only those key-value pairs that have to be stored explicitly in the composed transition function.
- We want to compute μ and ν for the composed transition from the functions μ and ν of the input transition function(s). We do not want to first combine the two halves to get a single representation for δ then apply a from the previous chapter construction, then try to split δ again.

Before we are going to show the refinement of the previous algorithm for the complex product construction to use μ and ν, we need two new macros:

- $record\_\mu(\sigma, tuple, r)$:
```
if $r \neq \mu_\sigma^{rank(\sigma)}(default)$ then
    $\mu_\sigma^{rank(\sigma)}(tuple) := r$
end if
if $r \notin R_{reach}$ then
    $R_{new} := R_{new} \cup \{r\}$
end if
```
- $record\_\nu(r, (U_1, U_2), q)$:
```
if $q \neq \nu_r(default)$ then
    $\nu_r(U_1, U_2) := q$
end if
if $q \notin Q_{reach}$ then
    $Q_{new} := Q_{new} \cup \{r\}$
end if
```

These macros have the same role as *record_transition* in the section 3.4, but they work with μ and ν, not with δ. Additionally they make sure that no redundant explicit key-value pairs are stored (when the value equals the default).

This is the algorithm for complex product construction as we have implemented it:

```
$Q_{reach} := \emptyset$
```

```
$Q_{new} := \emptyset$
$Q_{recent} := \emptyset$
$R_{reach} := \emptyset$
$R_{new} := \emptyset$
$\mu := empty\_\mu$
$\nu := empty\_\nu$
**for every** $\sigma \in \Sigma$ **do**
  $k := rank(\sigma)$
  $record\_\mu(\sigma, default, ((\mu_1)^k_\sigma(default), (\mu_2)^k_\sigma(default))$
**end for**
**for every** $(r_1, r_2) \in R_{new}$ **do**
  $record\_\nu((r_1, r_2), default, ((\nu_1)_{r_1}(default), (\nu_2)_{r_2}(default))$
  **for every** $(U_1, U_2)$ which has an explicit entry in $(\nu_1)_{r_1}$ or in $(\nu_2)_{r_2}$ **do**
    $record\_\nu((r_1, r_2), (U_1, U_2), ((\nu_1)_{r_1}(U_1, U_2), (\nu_2)_{r_2}(U_1, U_2)))$
  **end for**
**end for**
**while** $Q_{new} \neq \emptyset$ **do**
  $Q_{recent} := Q_{new}$
  $Q_{reach} := Q_{reach} \cup Q_{new}$
  $Q_{new} := \emptyset$
  $R_{reach} := R_{reach} \cup R_{new}$
  $R_{new} := \emptyset$
  **for every** $k \in \{rank(\sigma) \mid \sigma \in \Sigma\} \setminus \{0\}$ **do**
    **for every** $tuple \in (Q_{reach})^k$ **do**
      **if** $tuple \notin (Q_{reach} \setminus Q_{recent})^k$ **then**
        $(tuple_1, tuple_2) = unzip_k(tuple)$
        **for every** $\sigma \in \Sigma^{(k)}$ **do**
          $record\_\mu(\sigma, tuple, ((\mu_1)^k_\sigma(tuple_1), (\mu_2)^k_\sigma(tuple_2)))$
        **end for**
      **end if**
    **end for**
  **end for**
  **for every** $(r_1, r_2) \in R_{new}$ **do**
    $record\_\nu((r_1, r_2), default, ((\nu_1)_{r_1}(default), (\nu_2)_{r_2}(default))$
    **for every** $(U_1, U_2)$ which has an explicit entry in $(\nu_1)_{r_1}$ or $(\nu_2)_{r_2}$ **do**
      $record\_\nu((r_1, r_2), (U_1, U_2), ((\nu_1)_{r_1}(U_1, U_2), (\nu_2)_{r_2}(U_1, U_2)))$
    **end for**
  **end for**
**end while**
```

The first difference is that we keep track of two more sets: R_{reach} and R_{new}. These are used for computing reachable intermediate results.

Then, instead of processing only nullary symbols, we are able to compute the new default intermediate results for all symbols. This is possible because we can always compute the default intermediate results even with an empty set of reachable states. For the product construction the new default intermediate state is simply the pair of the default successors for the two subautomata.

Now we have a set of reachable intermediate results. We compute the default and explicit successor states for these intermediate results. The default successor state is again a pair of the defaults from the input automata. We must store explicit successor states for those sets U_1 and U_2 where at least one of the two input automata has stored an explicit entry. The pair $((\nu_1)_{r_1}(U_1, U_2), (\nu_2)_{r_2}(U_1, U_2))$ is different from the default successor state that we have computed in the previous step exactly in that case.

We are looking only at those (U_1, U_2) which had an explicit entry in one of the subautomata in order to follow the rule of processing as few key-value pairs as possible.

Then we come to the main loop of the algorithm. We have to maintain the sets R_{reach} and R_{new} in addition to Q_{recent}, Q_{reach}, and Q_{new}. In the first half of each iteration we compute intermediate results for tuples of input states with at least one new input state and keep track of new intermediate results. Then we compute the default and explicit successor states for those new intermediate results.

4.3.3 Second Order Quantification

The construction for second order quantification is simpler than the construction for first order quantification. We show only the pseudo code for second order for that reason.

For conjunction it was possible to extend the complex product construction in a straightforward way to our representation with μ and ν. For quantification we have to do more work.

Assume that we have an automaton $N' = (Q', R', \Sigma_{\mathcal{V}_1, \mathcal{V}_2 \cup \{X\}}, \mu', \nu', F')$ for the subformula ψ and we want to compute a deterministic automaton $N = (Q, R, \Sigma_{\mathcal{V}_1, \mathcal{V}_2}, \mu, \nu, F)$ for the formula $\exists X : \tilde{\kappa}.\psi$. The states Q that we compute are sets of states of the subautomaton like in the construction from section 3.6 but the intermediate results R are not simply sets of intermediate results of the subautomaton. Remember that we have to guess at which nodes the quantified second order variable occurs. Guessing variable positions is implemented in ν. All quantifications specify a set of sorts which a node must have to allow that we guess the variable's position at that node. A node's sort is the result sort of its symbol label. But the information about the symbol is only available in μ, not in ν. We have to transfer the information "is it allowed to guess the variable at this node" from μ to ν. To do this, we extend the intermediate results of the subautomaton with a flag. Together with the power set construction this means that intermediate results of the automaton for the quantified formula are elements of $\mathcal{P}(R' \times \{0, 1\})$ when R' are the intermediate results of the subautomaton.

```
$Q_{reach} := \emptyset$
$Q_{new} := \emptyset$
$Q_{recent} := \emptyset$
```

```
$R_{reach} := \emptyset$
$R_{new} := \emptyset$
$\mu := empty\_\mu$
$\nu := empty\_\nu$
for every $\sigma \in \Sigma$ do
  $(\kappa_1 \cdots \kappa_k, \kappa_0) := sort(\sigma)$
  if $\kappa_0 \in \tilde{\kappa}$ then
    $record\_\mu(\sigma, default, \{((\mu')^k_\sigma(default), 1)\})$
  else
    $record\_\mu(\sigma, default, \{((\mu')^k_\sigma(default), 0)\})$
  end if
end for
for every $r \in R_{new}$ do
  $record\_\nu(r, default, \{\nu_{r'}(default) \mid (r', flag) \in r\}$
  $V$ := set of all $(U_1, U_2)$ which have an explicit entry in $\nu'_{r'}$ for any $r' \in \{r' \mid$
  $(r', flag) \in r\}$.
  $V' := \{(U_1, U_2 \setminus \{X\}) \mid (U_1, U_2) \in V\}$
  for every $(U_1, U_2) \in V'$ do
    $q := \{\nu'_{r'}(U_1, U_2) \mid (r', flag) \in r\}$
    $q := q \cup \{\nu'_{r'}(U_1, U_2 \cup \{X\}) \mid (r', flag) \in r, flag = 1\}$
    $record\_\nu(r, (U_1, U_2), q)$
  end for
end for
while $Q_{new} \neq \emptyset$ do
  $Q_{recent} := Q_{new}$
  $Q_{reach} := Q_{reach} \cup Q_{new}$
  $Q_{new} := \emptyset$
  $R_{reach} := R_{reach} \cup R_{new}$
  $R_{new} := \emptyset$
  for every $k \in \{rank(\sigma) \mid \sigma \in \Sigma\} \setminus \{0\}$ do
    for every $tuple \in (Q_{reach})^k$ do
      if $tuple \notin (Q_{reach} \setminus Q_{recent})^k$ then
        for every $\sigma \in \Sigma^{(k)}$ do
          $(\kappa_1 \cdots \kappa_k, \kappa_0) := sort(\sigma)$
          if $\kappa_0 \in \tilde{\kappa}$ then
            $r := \{(\mu'_\sigma(tuple'), 1) \mid tuple' \in sub\_tuples(tuple)\}$
          else
            $r := \{(\mu'_\sigma(tuple'), 0) \mid tuple' \in sub\_tuples(tuple)\}$
          end if
          $record\_\mu(\sigma, tuple', r)$
        end for
      end if
    end for
  end for
  for every $r \in R_{new}$ do
    $record\_\nu(r, default, \{\nu_{r'}(default) \mid (r', flag) \in r\}$
    $V$ := set of all $(U_1, U_2)$ which have an explicit entry in $\nu'_{r'}$ for any $r' \in \{r' \mid$
    $(r', flag) \in r\}$.
    $V' := \{(U_1, U_2 \setminus \{X\}) \mid (U_1, U_2) \in V\}$
    for every $(U_1, U_2) \in V'$ do
       $q := \{\nu'_{r'}(U_1, U_2) \mid (r', flag) \in r\}$
       $q := q \cup \{\nu'_{r'}(U_1, U_2 \cup \{X\}) \mid (r', flag) \in r, flag = 1\}$
       $record\_\nu(r, (U_1, U_2), q)$
    end for
  end for
end while
```

As in the construction for the complex product, the first part of the algorithm processes default intermediate values for all symbols σ. You can see that we added the flag to indicate if guessing of the variable is allowed for this intermediate value.

Next we compute ν_r for all intermediate results r that we have found so far. For the default successor state of r we compute the default successor state of all elements of r with the subautomaton's function ν'.

Then we have to define the explicit part of ν_r. Remember that our goal is to look at as few explicit entries for ν_r as possible. We have to look at all explicit entries of the subautomaton (and store the keys of the explicit key-value pairs as V). In the subautomaton the variable X is a free variable, in the automaton that we are constructing now, it is not a free variable, so we have to remove it where it occurs in V (and store them in V').

Now we compute the value of ν_r for each pair $(U_1, U_2) \in V'$. This is works like the standard power set construction: compute the successor state for all elements of r in the subautomaton and use the resulting set as successor state in the new automaton. At this point we implement the guessing of X: compute ν' of the subautomaton again, but use $U_2 \cup \{X\}$ in place of U_2 if the flag allows it.

Finally, the main loop. After updating the sets Q_{recent}, Q_{reach}, Q_{new}, R_{reach}, and R_{new}, we handle explicit entries for μ. This works with the helper function *sub_tuples* like in the construction from section 3.6, but adds the $flag$ that indicates if guessing X is allowed for that particular intermediate result.

The second part of each iteration of the main loop processes new intermediate results in the same way as the processing of the initial set of intermediate results before the main loop.

4.3.4 First Order Quantification

The construction for first order quantification transforms an automaton $N' = (Q', R', \Sigma_{\mathcal{V}_1,\mathcal{V}_2}, \mu', nu', F')$ for a formula ψ into an automaton $N = (Q, R, \Sigma_{\mathcal{V}_1,\mathcal{V}_2}, \mu, \nu, F)$ for the formula $\exists X : \tilde{\kappa}.\psi$.

To implement first order quantification, we have to extend the construction from the previous section with code that ensures that a first order variable is guessed at exactly one node. As in section 3.5, we add an additional flag to states to implement this restriction. We have to add this flag to our intermediate results, too, that means $Q \subseteq \mathcal{P}(Q' \times \{0,1\})$ and $R \subseteq \mathcal{P}(R' \times \{0,1\} \times \{0,1\})$. Not that elements of an intermediate result carry two flags now, the first to indicate if the quantified variable has already been guessed, the second to indicate if the current node's sort allows guessing of the quantified variable.

For details on the construction we refer to the actual Haskell implementation of the construction.

4.4 Automata

In our implementation, an `Automaton` consists of

- a set of states, representing Q
- a set of final states, representing F
- a K-sorted ranked alphabet representing Σ
- a $\mathcal{P}(K)$-sorted set of free variables, representing $\mathcal{V}_1 \cup \mathcal{V}_2$
- a representation of the transition function, consisting of representations for μ and ν.

Chapter 5

Implementation Details

5.1 Ruby Style Loops

In chapter 3 we saw that many of the constructions involve deeply nested loops over finite sets. The usual way of implementing this in Haskell is by using recursion, list comprehension, or nested maps and folds. An earlier version of the program was using list comprehensions heavily. Although that version did not have e. g. default results and was not using the approach with two halves for the transition function, the source code was not very readable. Now the code is using a variant of map, together with monads which allows writing loops like those of the imperative programming language Ruby [7]. The name each was taken from Ruby.

```
each :: (Monad m) => [a] -> (a -> m b) -> m ()
each = flip mapM_
```

mapM_ from the Haskell 98 standard library applies a function which has one parameter and returns an action (i. e. a monadic result) to all elements of a list, then combines the actions with the monad operator >>. each is simply mapM_ with flipped parameters.

Writing loops with each looks like this:

```
eachExample :: IO ()
eachExample =
    each [0, 1, 2] $ \i -> do
        let i_squared = i*i
        each ["a", "b", "c"] $ \j -> do
            putStr (show i_squared)
            putStr " "
            putStr j
            putStr ", "
        putStrLn ""
```

The output of eachExample is:

```
0 a, 0 b, 0 c,
1 a, 1 b, 1 c,
4 a, 4 b, 4 c,
```

This is a form that makes implementing the algorithms from chapter 3 straightforward.

5.2 Building μ and ν

We said that the functions μ and ν that represent transition functions are implemented using `Maps` from the module `Data.Map`. We implement it with this simple Haskell data type:

```
data MapWithDefault key value = MapWithDefault {
    defaultValue :: Maybe value,
    explicitMap  :: Map key value
}
```

Both the data type `Map` and the data type `MapWithDefault` are "pure" functional data types, that means there is no way to destructively manipulate a value of that type.

All constructions that work with μ and ν are stated in a way that incrementally defines the maps. To use this style of programming in Haskell, we use the monad type constructor `State` from the module `Control.Monad.State` which is part of the Haskell Hierarchicall Libraries [5]. With this monad and the use of `each` and simple wrappers around `each`, the main part of the construction for $label_\sigma(x)$ looks like this:

```
transitionDefinition = execTransitionDefinition $ do
  eachRank ra $ \k -> do
      let tuples = tuplesOneDifferent k q0 q1
      eachConstructorWithRank ra k $ \con -> do
          each tuples $ \tuple -> do
              constructorExplicit con tuple r1
          constructorDefault con (if con == sigma
                                  then r2
                                  else r0)

  intermediateDefault r0 q0
  intermediateDefault r1 q1
  eachVariableSubset fv $ \varset -> do
      when (x `elem` varset) $
           intermediateExplicit r2 varset q1
  intermediateDefault iSigma q0
```

The monad provides the functions `constructorDefault`, `constructorExplicit`, `intermediateDefault`, and `intermediateExplicit` which simulate destructive updates of μ and ν by using the *State* monad.

5.3 Stateful Automata Compositions

The algorithms for composing algorithms also require that we keep global, updatable state. In those cases, the state consists of the sets Q_{recent}, Q_{reach}, Q_{new}, R_{reach}, and R_{new}, as well as the representations for the composed μ and ν. We wrote a polymorphic type that we use in all constructions that are based on the construction from section 3.3 (data type `FooState` in module `CompositeConstructions.hs`).

One of the goals of this project was to produce automata whose states have as little structure as possible. Therefore all our automata use only integers as states. Structured states exist only temporarily as long as a composition of automata runs. We achieve this with a slight extension of the sets Q_{reach} and R_{reach}. Instead of storing only structured states in these sets, we store a new integer name along with each structured state. Thus during the run of a composition algorithm we can already use the future unstructured name of a state. When the algorithm has finished, we simply drop the structured states and ouput only the integer states.

5.4 Code Generation

The task was to generate Haskell Code that represents the automaton.

We generate Haskell code using standard string operations. In our output we represent μ_σ as a `Map` for every $\sigma \in \Sigma$. We represent ν as simple Haskell functions, because the automata for which we generate code always have one free variable. That means we do not need the help of `Map`.

Final states are represented in a list in our output.

To generate code for an MSO formula which is given as `fi ::`FormulaInfo— (the data structure was part of the predefined input) and the ranked alphabet Σ is given as `ra :: RankAlph`, import the module `CodeGeneration` and use this Haskell expression:

```
formulaToHaskell fi ra
```

This expression returns a string that represents the final states and the transition function for the computed automaton.

5.5 Unit Tests

We're using the `Test.HUnit`[8] module to implement xUnit-style [9] unit tests. HUnit tests are not a complete solution for software testing, and of course they are not a substitute for verification of the implemented algorithms.

Each unit test tests a single function or a small group of functions that belong together. The general structure of such a unit test is:

1. Set up input values for the function

2. Evaluate the function with these input values
3. Compare the result with the expected value.

As an example, here is an HUnit test for the `replaceListValue` function which takes a list and two values as input. It returns all possible lists obtained from the input list by replacing one occurrance or no occurance of the first value by the second value.

```
module Main where

import ListFunctions (replaceListValue)
import Test.HUnit

testReplaceListValue :: Test
testReplaceListValue = TestCase $ do
              assertEqual "replaceListValue␣empty␣List"
                  [[]] (replaceListValue [] 1 2)
              assertEqual "replaceListValue␣multiple␣occurrences"
                  [[2,2,1,3,4], [1,2,2,3,4], [1,2,1,3,4]]
                  (replaceListValue [1,2,1,3,4] 1 2)

main = runTestTT testReplaceListValue
```

In this case, step one is not necessary because the input values for the functions are simple lists and integers. Step two are the `replaceListValue [] 1 2` and the `replaceListValue [1,2,1,3,4] 1 2` expressions. `assertEqual` is one of HUnit's functions for checking results. Its third argument is the actual result, the second argument is the expected result. The first argument is a message that will be printed if the expected and the actual result are not equal. This example shows, that it is possible to group more than one sequence of the three steps outlined above in one test. The disadvantage is that a failure in the first `assertEqual` leads to skipping the rest of this test, the second `assertEqual` will never be checked. The advantage is less (boilerplate) code.

An essential feature of these tests is that they run fully automated. No human inspection of results is necessary. This means they can be run as often as after every change to the program. That way, with good coverage, many programming errors can be found shortly after they have been introduced.

One requirement of this project was, that the program be built in a modular way, and that single modules of the code be reusable in other programs, e.g. programs dealing with the more powerful MSO* logic. HUnit tests actually encourage this code property. To be able to write tests with the three steps as above, it must be easily possible to set up input values, usually values of custom data types defined for the program, in few steps. To test a function in isolation it has to be usable in isolation.

5.6 Haddock In-Line Documentation

Another requirement of this project was, that the code is extensively documented. While this report provides information on the implemented algorithms and data structures, it does not document the modules in detail.

We use the Haddock [10] system for in-line code documentation at the level of individual functions. To use Haddock, Haskell function definitions, modules, and data structures must be annotated with comments that use a simple mark-up language. The `haddock` program extracts type declarations and these comments from the source code and generates hyperlinked HTML documentation.

This is the complete definition of `replaceListValue`, including Haddock comments:

```
-- | Takes a list and to values. Returns all possible lists that
--   you get when you replace one (or no) instance of the first
--   value by the second value.
--
--   @
--   replaceListValue [1,2,1,3,4] 1 2
--   @
--   yields [[2,2,1,3,4],[1,2,2,3,4],[1,2,1,3,4]]
replaceListValue :: (Eq a) => [a] -> a -> a -> [[a]]
replaceListValue [] _ _ = [[]]
replaceListValue (s:rest) q p
    | s==q = (p:rest):(map (s:) (replaceListValue rest q p))
    | otherwise = map (s:) (replaceListValue rest q p)
```

Haddock comments start with `-- |`. Example code is marked up with `@`.

Chapter 6

Conclusion and Future Work

With this project we implemented the algorithms to construct bottom-up tree automata for recognizing trees that are models for MSO formulas. We used a representation for the transition function that exploits redundancies in the well-known constructions to derive a space-efficient table representation for the transition function. We adapted the constructions to benefit from the special representation of the transition function.

Still, there are some shortcomings and some ideas that were not explored:

6.1 Representation of the Transition Function

Our representation of the transition function with μ and ν was first developed as part of the implementation in Haskell and not in a formal way. You can see that in a lack of formality in section 4.3. The choice of intermediate results in the implementation of atomic formulas was made intuitively by looking at the well-known construction. For composite formulas, intermediate results from subautomata were naïvely carried over to intermediate results for composite automata. It should be investigated if there are ways to compute a set of intermediate results with which more compact maps with defaults can be used. The time complexity of the resulting algorithms should be compared to the current naïve algorithms.

6.2 Respecting Sorts in During Reachability Constructions

All constructions based on the algorithm from section 3.3 ignore sort information and thus might decide that states are reachable that are in fact not reachable in any valid tree. It should be investigated how to extend the algorithms to respect sort information and how this extension interacts with our way of representing transition functions.

6.3 Minimization

There is an algorithm to minimize bottom-up tree automata, by computing a special congruence relation on the set of states and then replacing the set of states by the set of congruence classes. We did not manage to implement minimization as part of this project. Particularly because of our representation of transition functions.

6.4 Using BDDs

We believe that it is possible to implement the ν part of our transition function representation by using reduced ordered binary decision diagrams (ROBDDs or just BDDs) [11]. Using BDDs would lead to more space-efficient storage of ν.

Bibliography

[1] J. R. Büchi. Weak second-order arithmetic and finite automata. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, (6):66–92, 1960.

[2] J. W. Thatcher and J. B. Wright. Generalized finite automata theory with application to a decision problem of second-order logic. *Math. Syst. Theory*, 2(1):57–81, 1968.

[3] E. Bormann. Entwurf und implementierung eines systems zur erzeugung syntaxgesteuerter editoren. Master's thesis, Dresden University of Technology, 2000.

[4] Ari Saptawijaya. Implementation of an operational semantics for mso logic over trees. Master's thesis, Dresden University of Technology, 2004.

[5] Haskell hierarchical libaries reference documentation. `http://www.haskell.org/ghc/docs/latest/html/libraries/index.html`, 2005.

[6] Morten Biehl, Nils Klarlund, and Theis Rauhe. Algorithms for guided tree automata. In *WIA '96: Revised Papers from the First International Workshop on Implementing Automata*, pages 6–25, London, UK, 1997. Springer-Verlag.

[7] Dave Thomas with Chad Fowler and Andy Hunt. *Programming Ruby – The Pragmatic Programmers' Guide*. The Facets of Ruby Series. The Pragmatic Bookshelf, second edition, 2004.

[8] Dean Herington. Hunit 1.0 user's guide. `http://hunit.sourceforge.net/HUnit-1.0/Guide.html`, 2002.

[9] Kent Beck. *Test-Driven Development by Example*. The Addison-Wesley Signature Series. Addison-Wesley, 2003.

[10] Simon Marlow. Haddock: A haskell documentation tool. `http://www.haskell.org/haddock/`, 2005.

[11] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 8(C-35):677–691, 1986.