Algorithms for Tree Automata with Constraints

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

Student:
Vincent Hugot

Supervisors:
Pierre-Cyrille Héam
Olga Kouchnarenko

July 4, 2010

Contents

List of Tables

List of Figures

Part I

Some Preliminaries

Chapter 1

Introduction and Motivation

Tree automata, created in the fifties in the context of circuit verification, are powerful theoretical tools which have since found many useful applications in varied areas of Computer Science, such as automated theorem proving and program verification, term rewriting and XML schema languages, to name but a few. As new applications were found for them, new needs arose which prompted the development of several extensions of tree automata. A short but quite instructive survey of some of those extensions can be found in [FTT08a].

While these extensions have proved to be extremely useful from a theoretical point of view, there are practical drawbacks to their improved expressiveness, which lie mainly in the complexity and decidability of the associated decision problems (membership, emptiness, universality etc. . .). Extended expressiveness comes with a cost: in some cases those problems become undecidable, and those which are decidable fall for the most part in prohibitive classes of algorithmic complexity (*NP*-complete or worse). This makes it difficult to implement any sufficiently efficient tool based on these formalisms.

The focus of my Master's short project [(a)] and research internship [(b)] was on the elaboration and study of efficient algorithms and approaches to two decision problems associated with a recent extension of tree automata, called tree automata with global equality and disequality constraints (`TAGED`s for short).

The short project and the internship concerned the uniform membership problem for `TAGED`s and the emptiness problem for positive `TAGED`s, respectively. As announced , those are two difficult decision problems to tackle with any efficiency, for they are respectively *NP*-complete and *EXPTIME*-complete.

[(a)] First semester of 2009 – 2010.

[(b)] Second semester 2009 – 2010; kindly supported by INRIA project ACCESS.

The document you are now reading is the report for both the project and the internship. It is organised into four main parts.

In the first you will find – besides the present introduction – some reminders about tree automata, a quick survey of some of their extensions, as well as some information about the SAT problem and its practical uses. The purpose of this first part is both to give a context to this work, and to set the vocabulary, notations and concepts which will be used throughout this document.

The second part reports on the first semester's project: We propose a SAT encoding for the uniform membership problem, and discuss our preliminary experimental results.

The third part discusses several methods for generating random `TAGED`s to be used for experimental evaluation of new algorithms. This work was done as part of the internship, as the need for a systematic experimental approach arose.

The fourth part discusses the approaches and heuristics which I have developed to decide emptiness and reduce the size of `TAGED`s as much as possible. This was the main theme of the internship.

Chapter 2

Preliminaries

In this chapter we present the necessary vocabulary, notations and concepts which will be used throughout the document. References for this section include notably [CDG+07, FTT08a].

2.1 Tree Automata and extensions

Before we begin, let us state that there are two kinds of "vanilla" tree automata: bottom-up and top-down. Intuitively the first kind computes from the leaves up and the second from the root down. In the non-deterministic case both kinds have the same expressive power, however deterministic top-down tree automata are strictly less powerful than their bottom-up counterparts. And similarly to Finite State Machines, deterministic bottom-up tree automata are just as expressive as their non-deterministic brothers.

In this document, we shall only deal with the bottom-up variety, in the most general, non-deterministic, case. So whenever we write "tree automata", we mean "bottom-up non-deterministic tree automata".

2.1.1 Bottom-up Nondeterministic Finite Tree Automata

Symbols, trees, terms and subterms

Before speaking of "tree automata", let us start by defining the notions of "tree", and that of "term". We shall see that those two notions can be considered equivalent in the context which interests us, and so we will confuse them in the remainder of the document. Let Σ be a finite set of symbols, and

let $arity : \Sigma \to \mathbb{N}$ be the arity function. Intuitively, this function associates to a symbol $f \in \Sigma$ the number of "arguments" which it may take. We denote $\Sigma_n = \{ f \in \Sigma \mid arity(f) = n \}$ the set of all symbols of arity n, called "n-ary symbols", and $\mathfrak{Ar}_\Sigma = \{ k \in \mathbb{N} \mid \Sigma_k \neq \emptyset \}$ the set of all arities for which there exists at least a symbol in Σ. Whenever it is convenient, we shall denote $f/_n$ a symbol $f \in \Sigma_n$. It is assumed that the set of "constants" Σ_0 is non-empty. The couple $(\Sigma, arity)$ forms a "ranked alphabet". We will most often refer simply to the "ranked alphabet Σ" and omit the explicit mention of $arity$. We denote by $\mathcal{T}(\Sigma)$ the set of "ground terms" or more simply "terms", over the ranked alphabet Σ. It is defined as the smallest set such that

1. $\Sigma_0 \subseteq \mathcal{T}(\Sigma)$ and
2. for any $n \geqslant 1$ if $t_1, \ldots, t_n \in \mathcal{T}(\Sigma)$ then $f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma)$, for any $f \in \Sigma_n$.

A set of words S is said to be "prefix-closed" if it is such that for any $w \in S$, all prefixes of w are also in S. A tree over a set of labels L is a mapping from a prefix-closed set $S \subseteq \mathbb{N}^*$ into L. Let $t \in \mathcal{T}(\Sigma)$; it can be seen as a tree by defining the set of "positions" $\mathcal{P}os(t)$ inductively as follows:

1. $\mathcal{P}os(t) = \{ \varepsilon \}$ if $t \in \Sigma_0$
2. $\mathcal{P}os\Big(f(t_1, \ldots, t_n)\Big) = \Big\{ \varepsilon \Big\} \cup \Big\{ i.\alpha \mid i \in [\![1, n]\!] \text{ and } \alpha \in \mathcal{P}os(t_i) \Big\}$ otherwise.

Then t is a mapping from $\mathcal{P}os(t)$ to Σ, such that leaves map to Σ_0 and nodes map to symbols of corresponding arity. As announced above, we will from now on confuse those two notions: for instance for any term $t \in \mathcal{T}(\Sigma)$ and any $\alpha \in \mathcal{P}os(t)$, $t(\alpha)$ is the symbol at position α in the term t. Note that then $\mathcal{P}os(t)$ and $\mathrm{dom}(t)$ are two different notations for the same object, but in the context of trees we will systematically prefer the former notation. There remains to define the notion of "subterm" (or "subtree"). Let $t \in \mathcal{T}(\Sigma)$ and $\alpha \in \mathcal{P}os(t)$, we denote $t|_\alpha$ the subterm of t at position α, which is defined as follows:

1. $\mathcal{P}os(t|_\alpha) = \{ \beta \mid \alpha.\beta \in \mathcal{P}os(t) \}$
2. for any $\beta \in \mathcal{P}os(t|_\alpha)$, $t|_\alpha(\beta) = t(\alpha.\beta)$.

We denote by $u \trianglelefteq t$ the fact that u is a subterm of t,*ie.* there exists $\alpha \in \mathcal{P}os(t)$ such that $u = t|_\alpha$. The relation $\trianglelefteq$ is a partial order on $\mathcal{T}(\Sigma)$. The same notation is used between positions: for two positions $\alpha, \beta \in \mathcal{P}os(t)$, we say that "$\alpha$ is under

β" and note $\alpha \trianglelefteq \beta$ the fact that β is a prefix[(a)] of α. We define the induced strict order in the usual way, *ie.* $x \triangleleft y \iff x \trianglelefteq y$ and $x \neq y$, regardless of whether x and y are terms or positions. On some limited occasions, we will need a more precise evaluation of the degree to which one term is below another, and to this purpose we will denote $\alpha \trianglelefteq_n \beta$ (*resp.* $\alpha \triangleleft_n \beta$) the fact that $\alpha \trianglelefteq \beta$ (*resp.* $\alpha \triangleleft \beta$) and $|\alpha| - |\beta| = n$, for $n \in \mathbb{N}$ (*resp.* $n \in \mathbb{N}^*$). Note that $\alpha \trianglelefteq_0 \beta \iff \alpha = \beta$ and for all $n \geqslant 1$, $\alpha \triangleleft_n \beta \iff \alpha \trianglelefteq_n \beta$. In the vernacular, $\alpha \triangleleft_1 \beta$ means that α is a direct child of β, $\alpha \triangleleft_2 \beta$ means that α is a grand-child of β and so on. We have obviously $\trianglelefteq = \bigcup_{n\in\mathbb{N}} \trianglelefteq_n$ and $\triangleleft = \bigcup_{n\in\mathbb{N}^*} \triangleleft_n$.

Terms are quite often represented graphically. For instance

$$t \stackrel{\text{def}}{=} \begin{array}{l} f \\ \\ \\ \\ \end{array} \begin{array}{l} \diagup \\ \diagdown \\ \diagup \\ \diagdown \end{array} \begin{array}{l} f \\ \\ g \\ \\ \end{array} \begin{array}{l} \diagup\!\diagdown \\ \\ \diagup\!\diagdown \\ \\ \end{array} \begin{array}{l} a \\ a \\ a \\ b \end{array}$$

represents the term $t = f(f(a,a), g(a,b))$, with $a \in \Sigma_0$ and $f \in \Sigma_2$. If we add the positions of $\mathcal{P}os(t)$ as subscripts we get

$$t \stackrel{\text{def}}{=} \begin{array}{l} f_\varepsilon \\ \\ \\ \\ \end{array} \begin{array}{l} \diagup \\ \diagdown \\ \diagup \\ \diagdown \end{array} \begin{array}{l} f_1 \\ \\ g_2 \\ \\ \end{array} \begin{array}{l} \diagup\!\diagdown \\ \\ \diagup\!\diagdown \\ \\ \end{array} \begin{array}{l} a_{11} \\ a_{12} \\ a_{21} \\ b_{22} \end{array}.$$

Then we see that we have clearly, for instance $t(1) = f$ and $t|_2 = g \begin{array}{l} \diagup\!\diagdown \\ \\ \end{array} \begin{array}{l} a \\ b \end{array}$.

Tree automata

Before introducing tree automata formally, let us just state that they can be seen as an extension of finite state machines, which the reader is assumed to be familiar with. Indeed, a word can be seen as a term: let us take for instance the word $w = abc$. Then one can take the ranked alphabet $\Sigma = \{a/_1, b/_1, c/_1, \#/_0\}$ and build the corresponding unary term $t_w = a(b(c(\#)))$. Tree automata have the same expressiveness over unary terms as finite state machines have over words. The difference is that they recognise tree languages instead of just word languages, which makes them much more general.

A "non-deterministic finite tree automaton" (NFTA) over a ranked alphabet Σ is a tuple $\mathcal{A} = (\Sigma, Q, F, \Delta)$ where

[(a)] Note that this is the reverse notation from that taken in [FTT08b], which may be confusing to some readers. The advantage of choosing this notation the way we have done it is that the symbol "$\trianglelefteq$" remains consistent with respect to terms and positions, *ie.* for all $\alpha, \beta \in \mathcal{P}os(t)$, we have $\alpha \trianglelefteq \beta \implies t|_\alpha \trianglelefteq t|_\beta$.

- Σ is the ranked alphabet
- Q is a set of states, which we will see as constant symbols. Of course, states and standard symbols must not mix: $\Sigma \cap Q = \varnothing$. The set $\mathcal{T}(\Sigma \cup Q)$ is called the "set of configurations".
- $F \subseteq Q$ is the subset of "final" states
- Δ is a set of transition rules.

The rules of Δ define a "ground rewrite system" on $\mathcal{T}(\Sigma \cup Q)$. They are of the form

$$f(q_1, \ldots, q_n) \to q \quad \text{with } q, q_1, \ldots, q_n \in Q \text{ and } f \in \Sigma_n.$$

Thus, a tree automaton over Σ runs on ground terms over Σ, starting with the leaves. Indeed, rules for leaves are of the form $a \to q$, and can be considered "initial". The reader will have noticed that no set of initial states has been defined... We denote $\to_\Delta$ the rewriting relation, called "move relation" induced by Δ over $\mathcal{T}(\Sigma \cup Q)$, and $\to_\Delta^*$ its transitive reflexive closure. A term $t \in \mathcal{T}(\Sigma)$ is "accepted" by $\mathcal{A}$ if and only if there exists a final state $q_f \in F$ such that $t \to_\Delta^* q_f$. The "recognised tree language $\mathcal{L}\mathrm{ng}(\mathcal{A})$" of $\mathcal{A}$ is the set of all accepted terms:

$$\mathcal{L}\mathrm{ng}(\mathcal{A}) \stackrel{\text{def}}{=} \left\{ t \in \mathcal{T}(\Sigma) \mid \exists q_f \in F : t \to_\Delta^* q_f \right\}.$$

Or one can equivalently use the alternative definitions

$$\mathcal{L}\mathrm{ng}(\mathcal{A}, q) \stackrel{\text{def}}{=} \left\{ t \in \mathcal{T}(\Sigma) \mid t \to_\Delta^* q \right\} \quad \text{and} \quad \mathcal{L}\mathrm{ng}(\mathcal{A}) \stackrel{\text{def}}{=} \bigcup_{q_f \in F} \mathcal{L}\mathrm{ng}\left(\mathcal{A}, q_f\right).$$

Note that the move relation, such as we have defined it, destroys the term t until only one state is left. In order to keep track of the moves which led to this result, that is to say, of the states the different subterms evaluated to, another notion is needed. We call "run of $\mathcal{A}$ on t" a mapping $\rho : \mathcal{P}os(t) \to Q$ (in other words, a tree) compatible with the rules of Δ. That is to say, for every position $\alpha \in \mathcal{P}os(t)$, if $t(\alpha) = f \in \Sigma_n$, $\rho(\alpha) = q$ and $\forall i \in [\![1, n]\!] : \rho(\alpha.i) = q_i$, then there must exist some rule $f(q_1, \ldots, q_n) \to q \in \Delta$. A run ρ is said to be "accepting" or "successful" if $\rho(\varepsilon) \in F$. It follows that a term $t \in \mathcal{T}(\Sigma)$ is accepted by $\mathcal{A}$ if and only if there exists a successful run ρ of $\mathcal{A}$ on t. This is the definition which we will use most throughout this document.

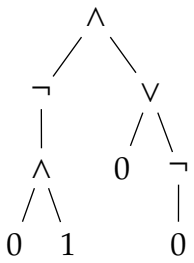
Let us take a very classical example:

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

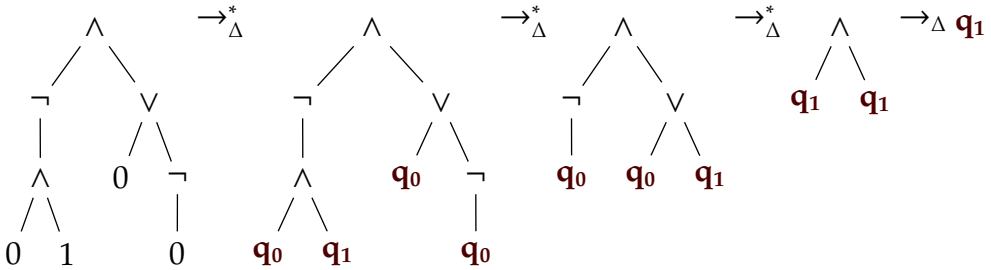
where the transition rules correspond closely to the usual rules of propositional logic:

$$\Delta = \left\{ b \to q_b,\ \wedge(q_b, q_{b'}) \to q_{b \wedge b'},\ \vee(q_b, q_{b'}) \to q_{b \vee b'},\ \neg(q_b) \to q_{\neg b} \mid\ b, b' \in \{0, 1\} \right\}.$$

For instance, $\wedge(q_0, q_1) \to q_0 \in \Delta$. Then if we consider the following term t



it can be rewritten with the help of the transition rules:

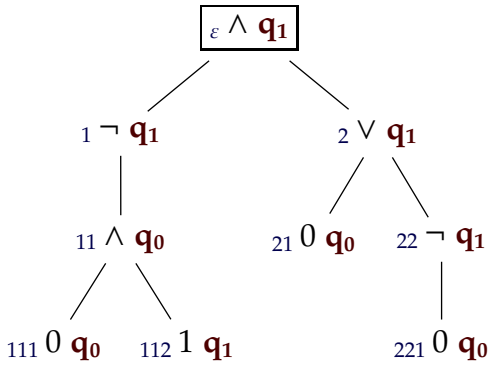


Note that each of the three first transformations above make use of several rules at once, as it would have been somewhat tedious to separate each and every step. Here is a breakdown of the rules which where used at each step of the transformation:

1. $0 \to q_0, 1 \to q_1 \in \Delta$
2. $\wedge(q_0, q_1) \to q_0, \neg(q_0) \to q_1 \in \Delta$
3. $\neg(q_0) \to q_1, \vee(q_0, q_1) \to q_1 \in \Delta$
4. $\wedge(q_1, q_1) \to q_1 \in \Delta$

This lengthy transformation $t \to^*_\Delta q_1$ can also be summarised more conveniently by the run ρ:

$\rho =$



Since $\rho(\varepsilon) = q_1 \in F$, the term t is accepted by $\mathcal{A}$. Of course, $\mathcal{A}$ recognises all true propositional formulæ, coded as trees.

Before moving on to extensions of tree automata, let us give in table 2.1[p15] a survey of common decision problems and their complexity, in both non-deterministic and deterministic cases, and state some closure properties.

Decision Problem	NFTA	DFTA
Emptiness	Linear time	Linear time
Equivalence	*EXPTIME*-complete	$O(\|\|\mathcal{A}_1\|\| \times \|\|\mathcal{A}_2\|\|)$
Finiteness	Polynomial	Polynomial
Inclusion	*EXPTIME*-complete	*EXPTIME*-complete
Intersection non-emptiness	*EXPTIME*-complete	*EXPTIME*-complete
Membership	*ALOGTIME*-complete	$O(\|\|t\|\|)$
Singleton set	Polynomial	Polynomial
Uniform Membership	$O(\|\|t\|\| \times \|\|\mathcal{A}\|\|)$	$O(\|\|t\|\| + \|\|\mathcal{A}\|\|)$
Universality	*EXPTIME*-complete	Polynomial

Table 2.1 — Decision problems for tree automata

Theorem 1 (Closure properties of tree automata)**.** *The class of recognisable tree languages is closed under union, under complementation, and under intersection.*

2.1.2 Some extensions

An aspect which is lacking in vanilla tree automata is testing equality and enforcing difference of some subterms. For instance, the language $\{ f(t,t) \mid f \in \Sigma, t \in \mathcal{T}(\Sigma) \}$ is non-regular, meaning that there is no tree automaton which recognises it. The same goes for $\{ f(t,t') \mid f \in \Sigma, t, t' \in \mathcal{T}(\Sigma) : t \neq t' \}$. Many extensions to tree automata have been proposed over the years to deal with such

constraints, the challenge being to add enough expressiveness to solve whatever problem was at hand, while preserving reasonable decidability and complexity, good closure properties etc. . . Two different approaches have been taken: either by considering local constraints, comparing only "relative" subterms, or by using global constraints, which allow comparison of arbitrary subterms.

In this section, which is very strongly inspired by [FTT08a], we will quickly go over some of these extensions.

The class AWEDC (1981)

The class *AWEDC* (short for *Tree Automata With Equality and Difference Constraints*) was first introduced by Max Dauchet and Jocelyne Mongy in Mongy's PhD thesis. Equality and difference constraints can be specified between any subterms. Transition rules are of the form:

$$f(q_1, q_2, q_3) \to_{1.3=2,\ 1.2\neq 1.3} q.$$

Then the rule only applies when, letting α be the position of the current term t, we have $t|_{\alpha.1.3} = t|_{\alpha.2}$ and $t|_{\alpha.1.2} \neq t|_{\alpha.1.3}$. This class is quite expressive and has good closure properties, unfortunately emptiness is undecidable.

Subclasses of AWEDC ($\approx$ 1990)

About ten years later, driven by new motivation in fields such as term rewriting, new subclasses of AWEDC were introduced. One of them is very simple: Automata With Constraints Between Brothers restricts AWEDC to constraints between immediate siblings. So for instance this rule

$$f(q_1, q_2, q_3) \to_{1.3=2,\ 1.2\neq 1.3} q.$$

is not allowed anymore while this one

$$f(q_1, q_2, q_3) \to_{1=2,\ 1\neq 3} q.$$

would still be. Thanks to this restriction emptiness becomes decidable, although it remains *EXPTIME*-complete in the case of non-deterministic automata.

The second class, called "reduction automata", involves ordering the states of Q and applying a transition rule

$$f(q_1, \ldots, q_n) \to_\varphi q \quad \text{where } \varphi \text{ is an AWEDC constraint}$$

only if q is strictly smaller than each q_i. This time emptiness is decidable (efficiently) in the deterministic case, but not in the non-deterministic one.

TAGED (2006)

Until now, the extensions which we have mentioned only used local constraints, that is to say, constraints between related subterms. There exist other classes which use *global* constraints, which apply between arbitrary subterms. The most recent addition to this family is the class TAGED: Tree Automata with Global Equality and Disequality Constraints, which were introduced in Emmanuel Filiot's PhD thesis [Fil08] and in the article [FTT08b].

Definition 2 (TAGED)**.** A TAGED is a tuple $\mathcal{A} = (\Sigma, Q, F, \Delta, =_{\mathcal{A}}, \neq_{\mathcal{A}})$, where

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

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

There is also a special subclass of TAGEDs which will interest us when we study the emptiness problem:

Definition 3 (Identity relation)**.** For any set S, we let $\mathfrak{id}_S$ be the identity relation on S, that is to say: $\mathfrak{id}_S = \{(x, x) \mid x \in S\}$.

Definition 4 (Diagonal TAGED)**.** A TAGED $\mathcal{A} = (\Sigma, Q, F, \Delta, =_{\mathcal{A}}, \neq_{\mathcal{A}})$ is said to be *diagonal* if $=_{\mathcal{A}} \subseteq \mathfrak{id}_Q$.

The notion of run is subject to a new constraint: it must be compatible with the equality and disequality constraints:

Definition 5 (Compatibility with the global constraints)**.** A run ρ is compatible with the equality constraint $=_{\mathcal{A}}$ if

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

In the same way, ρ is compatible with the disequality constraint $\neq_{\mathcal{A}}$ if

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

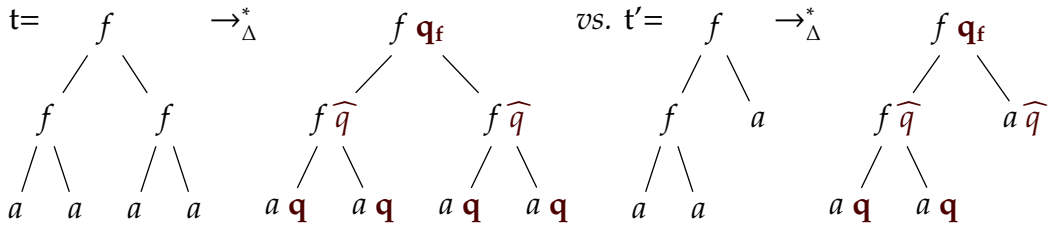
If $\neq_{\mathcal{A}}$ is not assumed to be irreflexive, this last definition must be extended into

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

Every other notion remains unchanged compared to vanilla tree automata. It is clear that TAGEDs are at least as expressive as the later, since they coincide exactly when $=_{\mathcal{A}}$ and $\neq_{\mathcal{A}}$ are both empty. They are in fact strictly more expressive, since they can recognise languages which vanilla tree automata cannot, such as the aforementioned $\{ f(t,t) \mid f \in \Sigma, t \in \mathcal{T}(\Sigma) \}$ and $\{ f(t,t') \mid f \in \Sigma, t, t' \in \mathcal{T}(\Sigma) : t \neq t' \}$. Let us convince ourselves that this is the case with the following, very classical example: the very simple positive TAGED $\mathcal{A}$ below recognises the language $\{ f(t,t) \mid f \in \Sigma, t \in \mathcal{T}(\Sigma) \}$. Note that this is a running example which the reader will meet again in section 4.1[p30], in the slides and probably in most works discussing TAGEDs.

$$\mathcal{A} \stackrel{\text{def}}{=} \left(\Sigma = \{a, f\},\ Q = \left\{q, \widehat{q}, q_f\right\},\ F = \left\{q_f\right\},\ \Delta,\ \widehat{q} =_{\mathcal{A}} \widehat{q},\ \widehat{q} \neq_{\mathcal{A}} q_f\right),$$
$$\text{where } \Delta \stackrel{\text{def}}{=} \left\{ f(\widehat{q}, \widehat{q}) \to q_f,\ f(q,q) \to q,\ f(q,q) \to \widehat{q},\ a \to q,\ a \to \widehat{q}, \right\}$$

To see how this works, consider the two terms below, and their runs for the underlying tree automaton. Both have accepting runs, since q_f if a final state, and are therefore in the language of the underlying tree automaton. The run over the first term has two instances of $\widehat{q}$ at positions 1 and 2, both over subterms structurally equal to $f(a,a)$. In other words we have $t|_1 = t|_2 = f(a,a)$. This term t is therefore compatible with the equality constraint $\widehat{q} =_{\mathcal{A}} \widehat{q}$ and it follows that the first term is accepted by the $\mathcal{A}$. On the other hand, in the case of the second term t' we also have two instances of $\widehat{q}$, but this time, one is over the subterm $t'|_1 = f(a,a)$ while the other is over $t'|_2 = a$. And because of this, we have $t'|_1 \neq t'|_2$ which violates the equality constraint $\widehat{q} =_{\mathcal{A}} \widehat{q}$, and t' is rejected by the TAGED $\mathcal{A}$.



Decidability and complexity of several decision problems are given in table 2.2[p19]. In the table, TAGED+ and TAGED- stand for positive and negative TAGEDs, respectively.

Theorem 6 (Closure properties of TAGEDs)**.** *The class of languages recognisable by TAGEDs is closed under union, and under intersection. It is not closed under complementation.*

Decision Problem	TAGED complexity
Emptiness, TAGED+	*EXPTIME*-complete
Emptiness, TAGED-	*NEXPTIME*
Finiteness, TAGED+, $=_{\mathcal{A}} \subseteq \mathfrak{id}_Q$	$O(\|\|A\|\| \times \|Q\|^2)$
Finiteness, TAGED+	*EXPTIME*
Membership	*NP*-complete
Universality	Undecidable

Table 2.2 — Decision problems for TAGED

2.2 Propositional logic and the SAT problem

In this section we give some very basic elements of propositional logic and present the SAT problem. References for this section include [GL07].

Classical propositional (or boolean) logic is the simplest of all logics. We let $\mathfrak{A}$ be an infinite set of "atoms", or "propositional variables" and build "formulæ" using only the basic logical connectors $\wedge$(and), $\vee$(or), $\neg$(not), $\Longrightarrow$ (implication). The symbols $\top$(or 1) and $\bot$(or 0) are also used to denote *true* and *false*, respectively. For instance, if we let $A, B, C, D \in \mathfrak{A}$, then $\varphi = \neg D \wedge (\bot \wedge A \wedge \top) \wedge (\bot \wedge B \vee \neg\bot) \vee C$ is a propositional formula. A truth value is associated to formulæ in the obvious way. A "literal" is either an atom A or its negation $\neg A$.

In this document, or more precisely in the proofs of our formulæ, we will mainly make use of three notions:

Definition 7 (Free Variables)**.** Le φ be a propositional formula. Then we call "free variables of φ", and denote FreeVars (φ), the propositional variables which appear in the formula φ. More formally,

$$\begin{aligned} \text{FreeVars}\,(\top/\bot) &= \varnothing \\ \text{FreeVars}\,(A) &= \{A\}, \text{ for } A \in \mathfrak{A} \\ \text{FreeVars}\,(\neg\varphi) &= \text{FreeVars}\,(\varphi) \\ \text{FreeVars}\,(\varphi[\wedge/\vee/\Rightarrow]\psi) &= \text{FreeVars}\,(\varphi) \cup \text{FreeVars}\,(\psi). \end{aligned}$$

Definition 8 (Conjunctive Normal Form)**.** A formula is in Conjunctive Normal Form (CNF) if it is a conjunction of disjunctions of literals and contains only $\neg$, $\wedge$ or $\vee$. Any formula can be put into CNF (that is to say, transformed into an equivalent formula which is in CNF) by applying basic logic rules.

Definition 9 (Valuation, Interpretation, Environment)**.** We call Valuation, Interpretation or Environment a mapping $I : \mathfrak{A} \to \{0, 1\}$, which associates a truth

value (0 or 1) to each atom. An interpretation I satisfies, or models, a propositional formula φ if φ is true for the interpretation I of the variables. We denote $I \models \varphi$ the fact that I models φ, and $I \not\models \varphi$ the fact that it does not.

Since the truth value of a propositional formula only depends on its free variables, a common abuse of this notation is to denote $J \models \varphi$ when J is only a *partial* function whose domain includes $\mathrm{FreeVars}(\varphi)$, and such that any extension I of J to the whole of $\mathfrak{V}$ models φ.

The **SAT Problem** (or Boolean satisfiability problem) consists in determining whether, for a given formula φ, there exists a valuation I which satisfies it. It is the first known *NP*-complete decision problem. Before it was proven to be so by Cook in 1971, the notion of *NP*-completeness did not even exist. Since then, a tremendous amount of research went into creating highly optimised heuristics for solving this problem, and into implementing them efficiently in so-called "SAT solvers". Let us just mention two among them, which we used in our experiments: `picoSAT` and `MiniSAT2`. Those efforts were successful enough that modern SAT solvers are generally capable of dealing with huge formulæ in reasonable time.

Since any *NP*-complete decision problem can be (polynomially) encoded into an instance of the SAT problem, encoding a new *NP*-complete problem into SAT proved to be a viable means of solving it efficiently. Instead of spending much time determining and implementing specific heuristics for each new problem, this method leverages all the work done on SAT solvers in almost forty years. This idea was first introduced in [CBRZ01], where it was used in the context of bounded model checking.

2.3 Other concepts and notations

In this last section we introduce the odd notations and concepts which do not fit well anywhere else.

Definition 10 (Size of a `TAGED` or Tree Automaton)**.** Let $\mathcal{A} = (\Sigma, Q, F, \Delta, =_{\mathcal{A}}, \neq_{\mathcal{A}})$ be a `TAGED`; in the case $\mathcal{A}$ is a vanilla tree automaton we will treat it as a `TAGED` all the same in the obvious way: $\mathcal{A} = (\Sigma, Q, F, \Delta, \varnothing, \varnothing)$. The *size* of $\mathcal{A}$ is denoted $\|\mathcal{A}\|$ and defined as:

$$\|\mathcal{A}\| \stackrel{\text{def}}{=} |Q| + 2 \cdot \left(|=_{\mathcal{A}}| + |\neq_{\mathcal{A}}| \right) + \sum_{f(p_1,\ldots,p_n)\to q \in \Delta} \left(\mathit{arity}(f) + 2 \right).$$

Definition 11 (Domain of a relation)**.** Let $\lessdot \subseteq S^2$ be a relation over a set S; then its domain $\mathrm{dom}(\lessdot)$ is defined as follows:

$$\mathrm{dom}(\lessdot) \stackrel{\text{def}}{=} \{ x \in S \mid \exists y \in S : x \lessdot y \text{ or } y \lessdot x \}$$

Definition 12 (Strict partial order)**.** Let $\lessdot \subseteq S^2$ be a relation over a set S; then it is a *strict partial order* if it satisfies the following properties, for any $x, y, z \in S$:

1. *Asymmetry*: for any $x, y \in S$, if $x \lessdot y$ then $y \not\lessdot x$.
2. *Irreflexivity*: for any $x \in S$, $x \not\lessdot x$.
3. *Transitivity*: for any $x, y, z \in S$: if $x \lessdot y$ and $y \lessdot z$ then $x \lessdot z$.

Notational conventions: I have strived to keep more or less uniform notations throughout this document. This paragraph gives the general guidelines which have been followed more or less closely.

- ◇ Automata are generally denoted $\mathcal{A}$, $\mathcal{A}'$, $\mathcal{B}$,...
- ◇ Closures of a relation R are denoted R^+ for the transitive closure of R and R^* for its reflexive and transitive closure.
- ◇ Definitions use the symbol $\stackrel{\text{def}}{=}$ whenever appropriate
- ◇ Positions in a term t (*ie.* elements of $\mathcal{P}os(t)$) are denoted α and β
- ◇ Propositional formulæ are denoted φ and ψ. The specific formulæ which we introduce use capital Greek letters
- ◇ Quotient sets are denoted $S/_\sim$, where S is a set and $\sim$ is an equivalence relation on S (or on a subset of S, in which case we silently use that subset in place of S). We denote $[x]_\sim$, or $[x]$ for short, the $\sim$-equivalence class of $x \in S$.
- ◇ References to distant elements include the page number as a subscript: for instance "section 5.1[p38]" refers to "section 5.1, page 38"
- ◇ Runs are denoted ρ and romps (used principally in chapter 15[p131]) are denoted ϱ
- ◇ States are denoted p and q

- ◇ Symbols (elements of Σ) are denoted $a, b, c, \ldots$ when they are constant, $f, g, \ldots$ when they are not, and σ when arity is completely unknown [(b)].
- ◇ Terms are denoted t, t' and subterms $u, v, \ldots$
- ◇ Valuations are denoted I and J (and in one occasion, $\mathcal{V}$).
- ◇ We shall sometimes speak of the "arity of a rule", and write $arity(r)$, for $r = f(p_1, \ldots, p_n) \to q$, which we shall take to mean $arity(f)$. For any tree t (whatever its labels), we shall also occasionally speak of its arity to mean that of its root symbol,*ie.* $arity(t) = arity(t(\varepsilon))$.

`OCaml` code: The sections concerning algorithms and implementations quote some `OCaml` code. I have written for the occasion a small program which allows a form of *literate programming*, mixing standard LaTeX and `OCaml` program code. Besides standard lexical highlighting, the program also uses special symbols as a replacement for the ASCII equivalent, for instance real arrows instead of `->` and λ instead of the `fun` of anonymous functions. The example code which follows shows this at work.

```
1  (* This is a LaTeX comment inside the .ml file *)
2  (*(*$ This is a \LaTeX\ comment inside the \texttt{.ml} file *)*)
3
4  (* The little snippets of code below show most of the few standard OCaml constructions
   which are rendered with special symbols (built-in aliases; see below) *)
5
6  (*let lambda_expression x = fun n -> 2*n + n , x*.x +. x*)
7  let lambda_expression x = λn → 2×n + n , x×.x +. x
8
9  (*let (a_match : 'a list list * 'b -> 'a list * 'b) =
10     function a::l,x -> a,x | _,x -> [],x*)
11 let (a_match : α list list × β → α list × β) =
12   λ a::l,x → a,x | ⊥,x → ∅,x
13
14 (*let quantifiers x f lst set set' = lst <> [] && x <= 10 && x >= 0
15   && List.mem x lst && List.for_all f lst
16   && List.exists f lst && Set.subset set set'
17   || Set.equal (Set.union set (Set.inter set' set)) Set.empty
18   || not (Set.is_empty Set.empty)*)
```

[(b)] Though in many cases I have also used $f, g, \ldots$ in that manner.

```
let quantifiers x f lst set set' = lst ≠ ∅ ∧ x ⩽ 10 ∧ x ⩾ 0
  ∧ ℒ.∈ x lst ∧ ℒ.∀ f lst
  ∧ ℒ.∃ f lst ∧ Set.⊆ set set'
  ∨ Set.= (Set.∪ set (Set.∩ set' set)) Set.∅
  ∨ ¬(Set.∅? Set.∅)

(* There are also a few non-standard constructions which I use, defined below *)

(* Function composition operator: Standard f ∘ g notation. *)
(*let (%) f g x = f (g x)*)
let (∘) f g x = f (g x)

(* Arguments to the right: this low-precedence operator is a syntactic short-
cut to avoid LISP-like parentheses creep *)
(*let (@@) f x = f x*)
let (⟦) f x = f x

let example f g h x y = f ⟦ g x ⟦ h y (* = f (g x (h y)) *)

(* List union: this concatenates two lists without any respect for the order of
their elements. It is faster than standard concatenation, and typically used
on lists which represent sets, hence the notation *)
(*let (@<) = List.rev_append*)
let (∪) = ℒ.rev_append

(* Indexes: Some identifiers will automatically be displayed with subscripts
*)
(*let x_1 = x_10 + b_i + k_A*)
let x₁ = x₁₀ + bᵢ + k_A

(* Aliases: The program makes is possible to declare aliases on the fly for a
given Caml identifier. This has been used in the algorithms to obtain a degree
of coherence between program notations and mathematical notations. Some
aliases have been mentionned in the previous paragraphs, but it would be
tedious to list all of them. Here we present user-defined aliases. *)
(*(*#alias:ta:alsTA*) (* note: \alsTA = LaTeX command *)
   (*#alias:qi:alsQI*) (* idem *)
 let a_tree_automaton = ta
 let f_of_qis = List.map f qi*)
let a_tree_automaton = 𝒜
```

```
let f_of_qis = ℒ.map f [q_1,...,q_n]

(* Masking and sectioning: There is some code which is necessary in practice
but utterly uninteresting to read: debug code, for instance. This code can
be masked by special markup and will not appear at all in the listings. This
supplements the markup which generates TeX files from sections of code
files, and helps bridge the gap between terse pseudo-code and practical
programming. *)
(*let some_function rho =
  (*$*) debug_out "Entering some_function..." (*$*)
  let pi = 3.1415 in 2*pi*rho *)
let some_function ρ =
  let π = 3.1415 in 2×π×ρ


(* OCaml code: XSet functor: During development of the prototypes, I have written
and used an extension of OCaml's standard library's Set.Make functor, called
XSet. The documentation for XSet-specific operations follows. Note that the
ASCII identifier is used in the signature line even if there is a LaTeX alias for
it, since those aliases are "module-aliases": they are only replaced when they
qualify an upper-case identifier.

    val of_list: elt list → t
    (* "Mod.of_list [x_1,...,x_n]" yields the set {x_1,...,x_n}. *)

    val map_to_list: (elt → α) → t → α list
    (* "Mod.map_ℒ f {x_1,...,x_n}" yields a list ℓ = [y_1,...,y_n] such that for all k ∈
    ⟦1,n⟧ : y_k = f(x_k). *)

    val push : elt → t ref → unit
    (* "Mod.↩ x S^r" adds the element x to the set reference: S^r := S^r ∪ {x}. *)

    val push_ : elt list → t ref → unit
    (* "Mod.↩⁺ [x_1,...,x_n] S^r" adds the elements of the list [x_1,...,x_n] to the set
    reference: S^r := S^r ∪ {x_1,...,x_n}. *)

    val pop : elt → t ref → unit
    (* "Mod.pop x S^r" removes the element x from the set reference: S^r := S^r \ {x}. *)

```

```
val inter_: t list → t
```
(* "`Mod.∩⁺ [`$S_1, \ldots, S_n$`]`" yields the set $\cap_{k=1}^{n} S_k$. *)

```
val union_: t list → t
```
(* "`Mod.∪⁺ [`$S_1, \ldots, S_n$`]`" yields the set $\cup_{k=1}^{n} S_k$. *)

```
val mem_ : elt list → t → bool
```
(* "`Mod.∈⁺ [`$x_1, \ldots, x_n$`] S`" returns **`true`** if and only if for all $k \in [\![1, n]\!] : \ x_k \in S$. *)

```
val add_ : elt list → t → t
```
(* "`Mod.add⁺ [`$x_1, \ldots, x_n$`] S`" yields the set $S \cup \{ x_1, \ldots, x_n \}$. *)

```
val iteri : (int → elt → unit) → t → unit
```
(* "`Mod.iteri f s`" applies f in turn to all elements of s. The elements of s are presented to f in increasing order with respect to the ordering over the type of the elements. Furthermore, f accepts the rank of each element as its first argument, starting from zero. *)

```
val iter2 : (elt → elt → unit) → t → t → unit
```
(* "`Mod.iter2 f s s'`" applies f in turn to all elements of s and s': equivalent to $f\ x_1\ x'_1; \ldots; f\ x_n\ x'_n$. The elements of the sets are presented to f in increasing order with respect to the ordering over the type of the elements. *)

```
val endo: (elt → elt) → t → t
```
(* "`Mod.endo f {`$x_1, \ldots, x_n$`}`" yields the set $\{ f(x_1), \ldots, f(x_n) \}$. *)

```
val filter_to_list : (elt → bool) → t → elt list
```
(* "`Mod.filter_to_list P {`$x_1, \ldots, x_n$`}`" yields the sub-list ℓ of $[x_1, \ldots, x_n]$ such that for any $x \in \ell : \ P(x)$ is true. *)

```
val remove_: elt list → t → t
```
(* "`Mod.remove_ [`$x_1, \ldots, x_n$`] S`" yields the set $S \setminus \{ x_1, \ldots, x_n \}$. *)

```
val product_to_list : t list → elt list list
```
(* "`Mod.`$\prod_{\mathcal{L}}$ $[S_1, \ldots, S_n]$`"` yields a list representation of the cartesian product $\prod = \prod_{k=1}^{n} S_k$ where any $\pi \in \prod$ is represented as the list $[x_1, \ldots, x_n]$ where $x_k \in S_k$ for any $k \in [\![1, n]\!]$. So it yields a list of $\prod_{k=1}^{n} |S_k|$ lists of n elements each. *)

Part II

TAGEDs and the Membership Problem

Chapter 3

Introduction

This part reports on the first semester's short project.

Update: As part of the second's semester internship, I have presented this material at the workshop *CSTVA 2010 - 2nd Workshop on Constraints in Software Testing, Verification and Analysis*, held the 10th of April 2010, in Paris, France. The slides are available online, both on the workshop's home page `http://www.st.cs.uni-saarland.de/cstva10` and on my own website.

Let us state the project's objective in some detail:

Objective: Find a reasonably efficient algorithm to solve the `TAGED` membership decision problem. As indicated in table 2.2[p19], this problem is *NP*-complete.

Suggested approach: Use a SAT encoding. See section 2.2[p19] for more information. As stated in the introduction, this suggestion has been followed.

Not-so optional objective: Implement the proposed solution and determine through experimental results whether it is viable. Since the objective is to find a *practical* approach to tackle this computationally hard problem, practical tests seem mandatory.

More formally, our main objective is, given an arbitrary `TAGED`

$$\mathcal{A} = (\Sigma, \Delta, Q, F, =_{\mathcal{A}}, \neq_{\mathcal{A}})$$

and a term $t \in \mathcal{T}(\Sigma)$, to build a propositional logic formula φ satisfiable if and only if $t \in \mathcal{L}\text{ng}(\mathcal{A})$.

The reader can find two different but complementary accounts of the suggested

SAT encoding: the "article version" in section 4.1[p29], and the "report/proof/full version" in section 5.1[p38]. The former is the most concise and up-to-date version, and arguably the most readable, but gives absolutely no proof. The latter aims at being complete, and offers proofs of correctness for each formula.

As for the experimental results, a concise account is given in section 4.2[p34], and additional remarks are made in section 5.2[p47].

Note that a complete formula example is given in figure 5.2[p50].

Chapter 4

Article version

This chapter presents the "article version" of my report. As part of my project, I co-wrote an article with my supervisors, the subject of which was essentially the same as that of the project itself. The parts which I wrote – plus some minor changes by my supervisors – are reproduced in sections 4.1[p29] and 4.2[p34].

The reader should note two practical details:

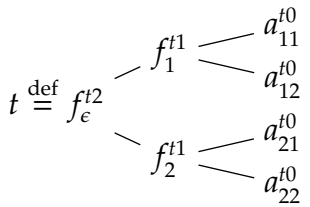
- Section 4.1 was, for the most part, written *after* section 5.1[p38]. It is therefore more up-to-date, and its explanations are likely to be clearer. Still, there are some parts which are common to both versions.
- There are some minor differences of exposition between the "article version" in this report and the "article version" in the article. Since I could not decide which way was clearer I chose to include both: whenever the article differs from what is given here, the "article's article version" has been included as a footnote.
- Section 5.2[p47] gives some experimental results and implementation details which were not covered in the article (section 4.2), mainly for want of space, or because they were out of its scope.

4.1 Propositional Encoding

This section presents our propositional encoding of the membership problem, and we informally justify it step by step. We shall also illustrate our sub-formulæ as we go along by instantiating them on a small example. For this purpose we

will use the following TAGED $\mathcal{A}$ and term t:

$$\mathcal{A} \stackrel{\text{def}}{=} \left(\Sigma = \{a, f\},\ Q = \left\{q, \widehat{q}, q_f\right\},\ F = \left\{q_f\right\},\ \Delta,\ \widehat{q} =_{\mathcal{A}} \widehat{q},\ \widehat{q} \neq_{\mathcal{A}} q_f\right),$$
$$\text{where } \Delta \stackrel{\text{def}}{=} \left\{ f(\widehat{q}, \widehat{q}) \to q_f,\ f(q,q) \to q,\ f(q,q) \to \widehat{q},\ a \to q,\ a \to \widehat{q}, \right\}$$



This small TAGED accepts $\{ f(t,t) \mid \ f \in \Sigma, t \in \mathcal{T}(\Sigma) \}$, which is a classical non-regular language. Here $\neq_{\mathcal{A}}$ is redundant and used purely for illustrative purposes. In the term, subscripts are positions and superscripts are unique references to the structure of subterms. For instance $t1$ corresponds to $f(a,a)$, which appears at positions 1 and 2.

Let us go back quickly over the conditions which must be satisfied in order for our term t to be accepted by $\mathcal{A}$: First, we need a run ρ for the underlying tree automaton $\mathcal{A}' = (\Sigma, \Delta, Q, F)$. Second, we need ρ to be *successful* for $\mathcal{A}'$. Third, it must satisfy the global constraints of $\mathcal{A}$. We shall encode each of these constraints separately, starting with the least restrictive: we must have a run ρ. So what is a run? First off, it is a *relation* between a position and a state. So the building blocks of our formula will be variables of the form, say, X_q^α, which will have the intuitive meaning that at a position $\alpha \in \mathcal{P}os(t)$, we end up in the state $q \in Q$. This corresponds to the statement "ρ exists and $\rho(\alpha) = q$". More precisely, we need ρ to be a function; let us encode, using the above variables, the fact that ρ is a partial function, that is to say, given $\alpha \in \mathcal{P}os(t)$ and $p \neq q \in Q$, we cannot have X_p^α and X_q^α at the same time: [(a)]

[(a)] *Article version:* Let us enumerate the conditions which must be satisfied in order for our term t to be accepted by $\mathcal{A}$ through a run ρ, and break them down in sub-conditions until we can encode them.

1. The run ρ is a *successful* run for the underlying tree automaton $\mathcal{A}' = (\Sigma, \Delta, Q, F)$.

- **(a)** The run ρ is a function mapping positions of t to states of $\mathcal{A}$.
  - **i.** $\rho \subseteq \mathcal{P}os(t) \times Q$
  - **ii.** $\forall \alpha \in \mathcal{P}os(t), p \neq q \in Q,\ (\alpha, p) \in \rho \implies (\alpha, q) \notin \rho$
  - **iii.** $\forall \alpha \in \mathcal{P}os(t), \exists q \in Q,\ (\alpha, q) \in \rho$
- **(b)** The run ρ must be compatible with the transition rules of Δ.
- **(c)** The run ρ must be *accepting*, *ie.* $\rho(\varepsilon) \in F$.

Definition 13 (Partial function constraint $\Theta_{\nrightarrow}$)**.**

$$\Theta_{\nrightarrow} \stackrel{\text{def}}{=} \bigwedge_{\substack{\alpha\in\mathcal{P}os(t)\\ q\in Q}} \left[X_q^\alpha \implies \bigwedge_{\substack{p\in Q\\ p\neq q}} \neg X_p^\alpha \right]$$

Applied to our minimalist example this yields $\{X_q^\epsilon \Rightarrow [\neg X_{\widehat{q}}^\epsilon \wedge \neg X_{q_f}^\epsilon]\} \wedge \{X_{\widehat{q}}^\epsilon \Rightarrow [\neg X_q^\epsilon \wedge \neg X_{q_f}^\epsilon]\} \wedge \cdots \wedge \{X_{q_f}^{22} \Rightarrow [\neg X_q^{22} \wedge \neg X_{\widehat{q}}^{22}]\}$. We also need ρ to be compatible with the transition rules of $\mathcal{A}'$ [(b)]. Let us translate the fact that a transition rule applies at a given position α by:

Definition 14 (Rule application constraint $\Psi^\alpha(r)$)**.** We define, for any $\alpha \in \mathcal{P}os(t)$, and any transition rule $f(q_1, \ldots, q_n) \to q \in \Delta$,

$$\Psi^\alpha\Big(f(q_1, \ldots, q_n) \to q\Big) \stackrel{\text{def}}{=} X_q^\alpha \wedge \bigwedge_{k=1}^{n} X_{q_k}^{\alpha.k}.$$

This is fairly straightforward: we are stating that the rule $f(q_1, \ldots, q_n) \to q \in \Delta$ applies at position α. Therefore we have "$\rho(\alpha) = q$" as a result of the application of the rule, and the k^{th} direct subterm is accepted by the state q_k, as the transition rule requires. Now, in order to express the notion of compatibility with the transition rules, we assert that, at each position in the term, a transition rule applies.

Definition 15. For any $f \in \Sigma$, we denote by $\Delta_f = \{ f(\ldots) \to \cdots \in \Delta \}$ the set of transition rules which apply to f.

Definition 16 (Rules compatibility constraint $\Phi^\varepsilon(t)$)**.**

$$\Phi^\varepsilon(t) \stackrel{\text{def}}{=} \bigwedge_{\alpha\in\mathcal{P}os(t)} \left[\bigvee_{r\in\Delta_{t(\alpha)}} \Psi^\alpha(r) \right].$$

---

2. It must satisfy the global equality constraints in $=_{\mathcal{A}}$.

3. It must satisfy the global disequality constraints in $\neq_{\mathcal{A}}$.

Condition (**1(a)i**) guides the choice of the building blocks of our formula: they will be variables of the form, say, X_q^α, which will have the intuitive meaning that at a position $\alpha \in \mathcal{P}os(t)$, we end up in the state $q \in Q$. This corresponds to the statement "ρ exists and $\rho(\alpha) = q$". Let us now encode, using the above variables, the fact that ρ is a partial function (**1(a)ii**), that is to say, given $\alpha \in \mathcal{P}os(t)$ and $p \neq q \in Q$, we cannot have X_p^α and X_q^α at the same time:

[(b)] *Article version:* (**1b**)

For instance, on our small example this would be $([X^{\epsilon}_{q_f} \wedge X^1_{\widehat{q}} \wedge X^2_{\widehat{q}}] \vee [X^{\epsilon}_{q} \wedge X^1_{q} \wedge X^2_{q}] \vee [X^{\epsilon}_{\widehat{q}} \wedge X^1_{q} \wedge X^2_{q}]) \wedge \cdots \wedge (X^{22}_{q} \vee X^{22}_{\widehat{q}})$. Note that if $\Phi^{\epsilon}(t)$ satisfies [(c)], then clearly ρ must be a total function [(d)], since at every position $\alpha \in \mathcal{P}os(t)$, we must be in some state q resulting from the application of some transition rule. Note also that if both $\Theta_{\twoheadrightarrow}$ and $\Phi^{\epsilon}(t)$ are satisfied simultaneously, then exactly one rule applies at each position. The last thing we need to encode an accepting run for a tree automaton, is to specify that the run must end up in a final state at the root of the term [(e)]; this is directly translated into $\bigvee_{q \in F} X^{\epsilon}_{q}$. Now we must add further restrictions to ensure compatibility with the global equality and disequality constraints [(f)]. The variables we have already defined are not sufficient to translate statements of the form "such subtree does (or does not) evaluate to such state"; therefore we need to introduce new variables to link states and subterms by a relation. Let us use T^q_u to denote "the subterm u evaluates to q", for any $u \trianglelefteq t$ and $q \in Q$. Of course, we need to "glue" these new variables to the old ones: if we are in a certain state q at a position α, then it follows that the subterm $t|_\alpha$ evaluates to q: this is straightforwardly translated into the next formula.

Definition 17 (Structural glue: $\Theta_{\leftrightarrows}$)**.**

$$\Theta_{\leftrightarrows} \stackrel{\text{def}}{=} \bigwedge_{\substack{\alpha \in \mathcal{P}os(t) \\ q \in Q}} \left[X^{\alpha}_{q} \implies T^{q}_{t|_\alpha} \right].$$

On our example, we have: $\{X^{\epsilon}_{q} \Rightarrow T^{q}_{2}\} \wedge \{X^{\epsilon}_{\widehat{q}} \Rightarrow T^{\widehat{q}}_{2}\} \wedge \{X^{\epsilon}_{q_f} \Rightarrow T^{q_f}_{2}\} \wedge \cdots \wedge \{X^{22}_{q_f} \Rightarrow T^{q_f}_{0}\}$, where the subscript "2" of T^q_2 designates the subtree $f\Big(f(a,a), f(a,a)\Big)$, as given in the definition of t. Now different kinds of variables being linked, let us encode the equality constraint. Supposing again that $\rho(\alpha) = q$, for the run to be compatible with the equality constraint, it must be such that no subterm different from $t|_\alpha$ can evaluate to p, where $p =_{\mathcal{A}} q$. Note that $=_{\mathcal{A}}$ is reflexive by definition, so this includes q itself.

Definition 18 (Compatibility with $=_{\mathcal{A}}$: $\Theta_{=_{\mathcal{A}}}$)**.**

$$\Theta_{=_{\mathcal{A}}} \stackrel{\text{def}}{=} \bigwedge_{\substack{\alpha \in \mathcal{P}os(t) \\ q \in Q}} \left[X^{\alpha}_{q} \implies \bigwedge_{\substack{p \in Q \\ p =_{\mathcal{A}} q}} \bigwedge_{\substack{u \trianglelefteq t \\ u \neq t|_\alpha}} \neg T^{p}_{u} \right]$$

[(c)] *Article version:* (**1b**)
[(d)] *Article version:* (**1(a)iii**)
[(e)] *Article version:* (**1c**)
[(f)] *Article version:* (**2** and **3**)

For instance: $\{X^{\epsilon}_{\widehat{q}} \Rightarrow [\neg T^{\widehat{q}}_1 \wedge \neg T^{\widehat{q}}_0]\} \wedge \{X^{11}_{\widehat{q}} \Rightarrow [\neg T^{\widehat{q}}_2 \wedge \neg T^{\widehat{q}}_1]\} \wedge \cdots \wedge \{X^{22}_{\widehat{q}} \Rightarrow [\neg T^{\widehat{q}}_2 \wedge \neg T^{\widehat{q}}_1]\}$. There remains to encode the compatibility with the disequality constraint. Let us deal with the case where either $\neq_{\mathcal{A}}$ is assumed to be irreflexive (as in [FTT08b]), or the states involved are different. Suppose that we are at position α, and that $\rho(\alpha) = q$; then we cannot have any subterm $t|_\alpha$ evaluate to any p, when $p \neq_{\mathcal{A}} q$.

Definition 19 (Compatibility with $\neq_{\mathcal{A}}$ ($p \neq q$): $\Theta_{\neq_{\mathcal{A}}}$)**.**

$$\Theta_{\neq_{\mathcal{A}}} \stackrel{\text{def}}{=} \bigwedge_{\substack{\alpha \in \mathcal{P}os(t) \\ q \in Q}} \left[X^{\alpha}_{q} \implies \bigwedge_{\substack{p \in Q \\ p \neq_{\mathcal{A}} q \\ p \neq q}} \neg T^{p}_{t|_\alpha} \right]$$

For instance: $\{X^{\epsilon}_{\widehat{q}} \Rightarrow \neg T^{q_f}_2\} \wedge \{X^{\epsilon}_{q_f} \Rightarrow \neg T^{\widehat{q}}_2\} \wedge \cdots \wedge \{X^{22}_{q_f} \Rightarrow \neg T^{\widehat{q}}_0\}$. However, for the needs of our test examples, we chose to alter the definition of $\neq_{\mathcal{A}}$ by removing its irreflexivity. The idea is to be able to write statements such as $p \neq_{\mathcal{A}} p$, with the meaning that no two *distinct* subtrees which evaluate to p may be structurally identical. Formally, ρ satisfies $\neq_{\mathcal{A}}$ iff $\forall \alpha, \beta \in \mathcal{P}os(t), (\alpha \neq \beta \wedge \rho(\alpha) \neq_{\mathcal{A}} \rho(\beta)) \implies t|_\alpha \neq t|_\beta$. This cannot be done solely in $\Theta_{\neq_{\mathcal{A}}}$, because the formula will not differentiate between two distinct subterms and the same subterm, taken twice, which is why the case where $q \neq_{\mathcal{A}} q$ must be dealt with separately. Indeed, as we do not yet have any means for linking subterms with positions, a new kind of variables is needed, of the form S^{α}_{u}, which encodes the statement "the subterm u is rooted in α". The above property is then encoded using this variable, as follows:

Definition 20 (Compatibility with $\neq_{\mathcal{A}}$ (non-irreflexive; $q \neq_{\mathcal{A}} q$): $\Omega_{\neq_{\mathcal{A}}}$)**.**

$$\Omega_{\neq_{\mathcal{A}}} \stackrel{\text{def}}{=} \bigwedge_{\alpha \in \mathcal{P}os(t)} S^{\alpha}_{t|_\alpha} \wedge \bigwedge_{\substack{\alpha \neq \beta \in \mathcal{P}os(t) \\ q \neq_{\mathcal{A}} q}} \left[X^{\alpha}_{q} \wedge X^{\beta}_{q} \implies \neg S^{\alpha}_{t|_\beta} \right]$$

We can now state our main result:

Definition 21 (SAT encoding of TAGED membership problem $\Delta_{\mathcal{A}}(t)$)**.** Let $\mathcal{A} = (\Sigma, \Delta, Q, F, =_{\mathcal{A}}, \neq_{\mathcal{A}})$ be a TAGED and $t \in \mathcal{T}(\Sigma)$; then we define

$$\Delta_{\mathcal{A}}(t) \stackrel{\text{def}}{=} \Theta_{\twoheadrightarrow} \wedge \Phi^{\epsilon}(t) \wedge \bigvee_{q \in F} X^{\epsilon}_{q} \wedge \Theta_{=_{\mathcal{A}}} \wedge \Theta_{\neq_{\mathcal{A}}} \wedge \Omega_{\neq_{\mathcal{A}}}.$$

Theorem 22 (TAGED membership, correctness and soundness)**.** *There exists a successful run ρ of the TAGED $\mathcal{A}$ on a term t iff $\Delta_{\mathcal{A}}(t)$ is satisfiable. Moreover, if $I \models \Delta_{\mathcal{A}}(t)$, then for any $\alpha \in \mathcal{P}os(t)$ we have $\rho(\alpha) = q \iff I \models X^{\alpha}_{q}$.*

The above encoding has been simplified, implemented and tested. This is the matter of the next section.

4.2 Complexity, implementation and experiments

In the first part of this section we will quickly go over some ways in which the formula can be lightened through simple observations, before discussing some of our experimentations in the second part.

The above SAT encoding, though sizeable, remains polynomial in the size of our input automaton $\mathcal{A}$ and the term t: the size of $\Delta_{\mathcal{A}}(t)$ (as number of literals) is a $O(|t|^2 |Q|^2)$. In practice however, this can often be trimmed down considerably. Let ρ be a successful run of the underlying tree automaton $\mathcal{A}$ on t, and consider for instance the structural glue: $\Theta_{\leftrightarrows} = \bigwedge_{\alpha \in \mathcal{P}os(t), q \in Q} [X_q^\alpha \implies T^q_{t|_\alpha}]$. The formula considers all possible couples (α, q), but in general this is unnecessary because not all states are obtainable at any given position. In order to ever have X_q^α, that is to say, $\rho(\alpha) = q$, there must be some transition rule of the form $t(\alpha)(\dots) \to q$ in Δ, at least. Thus we let $\sigma(\alpha)$ be the set of *possibly obtainable states at position* α: $\sigma(\alpha) \stackrel{\text{def}}{=} \{ q \in Q / \exists t(\alpha)(\dots) \to q \in \Delta \}$ and, given a position α, we only need to deal with $q \in \sigma(\alpha)$. Another observation which can be made *a priori* is that the only occurrences of negations of the form $\neg T^q_u$ occur in $\Theta_{=_{\mathcal{A}}}$ and $\Theta_{\neq_{\mathcal{A}}}$, when q is in the domain of either $\neq_{\mathcal{A}}$ or $=_{\mathcal{A}}$. It follows that literals of the form T^q_u can only alter the satisfiability of $\Delta_{\mathcal{A}}(t)$ when q is in $\mathrm{dom}(\neq_{\mathcal{A}}) \cup \mathrm{dom}(=_{\mathcal{A}})$. Thus we can reduce the formula to $\Theta_{\leftrightarrows} = \bigwedge_{\alpha \in \mathcal{P}os(t), q \in \sigma(\alpha) \cap (\mathrm{dom}(\neq_{\mathcal{A}}) \cup \mathrm{dom}(=_{\mathcal{A}}))} [X_q^\alpha \implies T^q_{t|_\alpha}]$. The same observations can be made in $\Theta_{\neq_{\mathcal{A}}}$, $\Omega_{\neq_{\mathcal{A}}}$ and $\Theta_{=_{\mathcal{A}}}$. In the case of $\Theta_{=_{\mathcal{A}}}$, we can also argue that in the subformula $\bigwedge_{u \trianglelefteq t, u \neq t|_\alpha} \neg T^p_u$ it is unnecessary to write $\neg T^p_u$ when we know that the subtree u cannot possibly evaluate to the state p. This is clearly the case if the root symbol $u(\varepsilon)$ is not used in any transition rule leading to p. Thus we let $\tau(q) \stackrel{\text{def}}{=} \{ f \in \Sigma \mid \exists f(\dots) \to q \in \Delta \}$ be the set of symbols which a subterm may be rooted in, given that it evaluates to the state q, and we lighten the above subformula into $\bigwedge_{u \trianglelefteq t, u \neq t|_\alpha, u(\varepsilon) \in \tau(p)} \neg T^p_u$. Lastly, in the revised formula $\Omega_{\neq_{\mathcal{A}}}$, it is clear that the variables $S^\alpha_{t|_\alpha}$ serve no purpose whatsoever when the subtree in α cannot evaluate to a state q such that $q \neq_{\mathcal{A}} q$. Thus we let $\mu(q) \stackrel{\text{def}}{=} \{ \alpha \in \mathcal{P}os(t) \mid t(\alpha) \in \tau(q) \}$ be the set of positions at which the subtree may evaluate to the state q, and reduce the first part of the subformula to $\bigwedge_{\alpha \in \bigcup_{q \neq_{\mathcal{A}} q} \mu(q)} S^\alpha_{t|_\alpha}$. In its second part, we arbitrarily order positions and regroup couples of implications with the same premises: $\bigwedge_{\alpha < \beta \in \mu(q), q \neq_{\mathcal{A}} q} [X^\alpha_q \wedge X^\beta_q \implies \neg S^\alpha_{t|_\beta} \wedge \neg S^\beta_{t|_\alpha}]$. Note that reducing $\Theta_{\nrightarrow}$ is much more problematic, but it is possible to simply do away with this part of the formula altogether if one replaces $\bigvee_{q \in F} X^\varepsilon_q$ by $\bigwedge_{q \notin F} \neg X^\varepsilon_q$, provided that the term is accepted by the underlying tree automaton. This can be checked separately by other, less expensive means, since the membership problem for tree automata is polynomial. Of course in that case the second result of theorem 22 does not

apply anymore. While computationally inexpensive, these simplifications can yield significant savings on TAGEDs with low density and where few states are involved in the global constraints, which are fairly reasonable assumptions in the context of XML documents processing. Note that one could find more drastic simplifications by examining the tree automaton more closely; for instance one could remove, at each position, any state which cannot appear in a successful run. Simplifications of this kind would certainly yield better results on sizeable and complex TAGEDs, but it is not certain that the overhead of implementing and computing them would be compensated by the SAT solving performance gains. For our tests we implemented the static simplifications described above, which divided the size of the generated formula by 36 in the case of our Laboratory example automaton.

In order to test our encoding, we have been developing a tool which takes as input a TAGED (in a syntax close to that of Timbuk [FGT04]) and a term, and generates the corresponding formula $\Delta_{\mathcal{A}}(t)$. However, most modern SAT solvers take input in the DIMACS CNF format, and naive conversion to Conjunctive Normal Form (using De Morgan's laws, distributivity and removal of double negations) could lead to an explosion of the size of the formula. In order to avoid running into this problem we used an existing tool to handle linear-size conversion to CNF and generation of DIMACS CNF files: the BAT[g] [MSV07], which implements an efficient CNF conversion algorithm [MV09]. Experiments were run on an 2.53GHz Intel Core2 Duo machine with 2Gb of RAM running Linux. Figure 4.1 shows the respective running times of the two SAT solvers picoSAT and MiniSAT2 on an implementation of our Laboratory example. Accepted trees of varying sizes have been generated with random member names of random length. In the figure the size of the generated trees is given in terms of the number of teams in the university; the size in terms of the number of nodes is proportional to these data. The test shows that while both solvers perform very well on this query, MiniSAT2 tends to outperform picoSAT as the terms grow, which suggests that the heuristic used for SAT solving may significantly impact the overall efficiency of our queries. Figure 4.2 shows the same experiment, this time with the small TAGED accepting $\{ f(t,t) \mid f \in \Sigma, t \in \mathcal{T}(\Sigma) \}$ (introduced at the beginning of section 4.1), and for both accepted and rejected terms. The size of the terms designates the number of nodes of the tree. Both solvers display similar performances for this experiment, with MiniSAT2 being about twice as fast as picoSAT on accepted terms. On rejected terms however both solvers show roughly the same performances, and take less time than on accepted terms, by

[g] Bit-level Analysis Tool, version 0.2

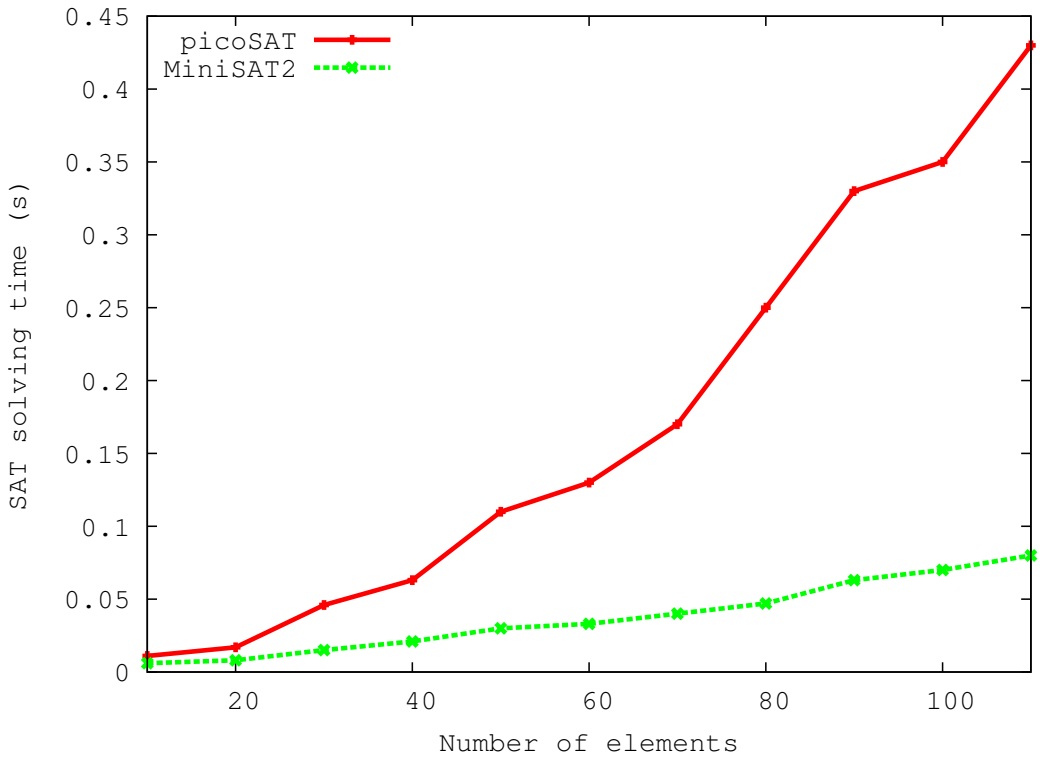


Figure 4.1 — CNF solving time, Laboratory example

a factor of 3 (`picoSAT`) and 5 (`MiniSAT2`) on large terms.
It would have been interesting to increase the size of our terms until both solvers timed out, but we were unfortunately limited by the software we used. Our own tool is not optimised for speed, and CNF conversion with BAT took about 4.5 times as much time as formula generation. Moreover, BAT fails with a stack overflow when the input formula becomes too large. Despite these practical setbacks, the results remain fairly encouraging, as the current bottleneck lies on the least computationally expensive parts of the process: both the generation of the formula and the conversion to CNF are quadratic in the worst case. On the other hand, SAT solving proves quite efficient, even on fairly large formulæ: the order of magnitude of the largest tested formulæ is of approximately 70'000 variables, 120'000 clauses and 250'000 literals (in CNF), for a solving time well under one second.

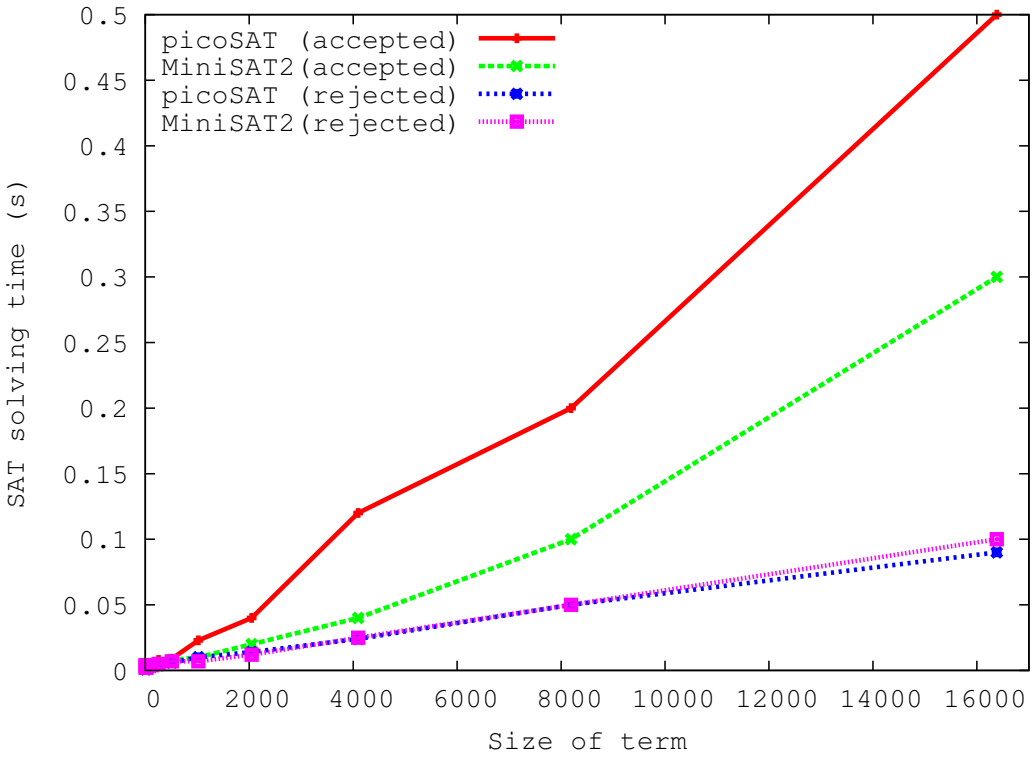


Figure 4.2 — CNF solving time, $\{ f(t,t) \mid f \in \Sigma, t \in \mathcal{T}(\Sigma) \}$

Chapter 5

Full version

This chapter presents in section 5.1 the full version, with proof, of our proposed propositional encoding. In section 5.2[p47] the reader will find some experimental results and implementation details which were not covered in the article.

The reader may refer to the introduction to chapter 4[p29] for more information concerning the practical relationship between sections 5.1 and 4.1.

5.1 Propositional encoding and proof

Let us go back quickly over the conditions which must be satisfied in order for our term t to be accepted by $\mathcal{A}$: First, we need a run ρ for the underlying tree automaton $\mathcal{A}' = (\Sigma, \Delta, Q, F)$. Second, we need ρ to be *successful* for $\mathcal{A}'$. Third, it must satisfy the global constraints of $\mathcal{A}$.

But how can we express those fairly complex conditions using only propositional variables? The directing idea which we will follow is to focus on the notion of run, and to translate the above constraints into propositional logic, step by step. We will start from the simplest, least restrictive notion, which is that of a run on $\mathcal{A}'$, to the full notion of `TAGED`-accepting run, progressively introducing new restrictions in the form of *constraint formulæ* which we append to (that is to say, put in conjunction with) our propositional formula.

Let us start at the very beginning, then: *what is a run ?* First off, it is a *relation* between a position and a state. So the building blocks of our formula will be variables of the form, say, X_q^α, which will have the intuitive meaning that at a position $\alpha \in \mathcal{P}os(t)$, we end up in the state $q \in Q$. This corresponds loosely to the statement "ρ exists and $\rho(\alpha) = q$". So in everything that follows, we will think

of the propositional formula which we are building in terms of the *inferred run relation* ϱ, which translates this idea. Note that nearly every definition from that point forward depends on both our given TAGED $\mathcal{A}$ and the term t mentioned above, which are omitted from the notations so as to keep things somewhat legible.

Definition 23 (State variables)**.** For any $\alpha \in \mathcal{P}os(t)$ and $q \in Q$, we let X_q^α be a propositional variable. We call these variables *state variables*, and we denote

$$X \stackrel{\text{def}}{=} \left\{ X_q^\alpha \mid \alpha \in \mathcal{P}os(t) \text{ and } q \in Q \right\}$$

the set of all state variables.

Definition 24 (Inferred run relation from I on φ)**.** Let φ be a satisfiable propositional formula, and I an arbitrary model of φ. Then we call *inferred run relation from I on φ* the relation $[I]\varrho \subseteq \mathcal{P}os(t) \times Q$ such that $(\alpha, q) \in [I]\varrho \iff I \models X_q^\alpha$.

Definition 25 (Set of inferred run relations on φ)**.** Let φ be a propositional formula, then we denote by $\mathcal{R}(\varphi) = \{ [I]\varrho \mid I \models \varphi \}$ the set of all inferred run relations on φ.

In everything that follows we will often slightly abuse this notation in statements such as "let $\rho = [I]\varrho \in \mathcal{R}(\varphi)$". This is to be understood as shorthand for "let $\rho \in \mathcal{R}(\varphi)$, and let I a model of φ such that ρ is the inferred run from I on φ". Such an I exists by definition; there is even an infinity of them, since only their restriction to X influences the inferred run.

The following results are merely rather straightforward consequences of the above definitions, but they will be the very backbone of our reasoning in this section, since they provide the formal glue which justifies the intuitive notion we mentioned earlier of building the formula as a conjunction of constraints.

Lemma 26 (X-Restriction lemma)**.** *Let I and J be two valuations of all propositional variables; then $[I]\varrho = [J]\varrho$ iff $I|_X = J|_X$.*

Proof. Suppose $I|_X = J|_X$; then for all $(\alpha, q) \in \mathcal{P}os(t) \times Q$, $(\alpha, q) \in [I]\varrho \iff I \models X_q^\alpha$ and since $X_q^\alpha \in X$, $I \models X_q^\alpha \iff J \models X_q^\alpha \iff (\alpha, q) \in [J]\varrho$. Thus $[I]\varrho = [J]\varrho$. Conversely, assume that $I|_X \neq J|_X$. Then there exists a $X_q^\alpha \in X$ such that $I \models X_q^\alpha$ but $J \not\models X_q^\alpha$ (resp. $I \not\models X_q^\alpha$ but $J \models X_q^\alpha$) and it follows that $(\alpha, q) \in [I]\varrho$ and $(\alpha, q) \notin [J]\varrho$ (resp. $(\alpha, q) \notin [I]\varrho$ and $(\alpha, q) \in [J]\varrho$) and so $[I]\varrho \neq [J]\varrho$. □

Lemma 27 (Chain lemma)**.** *Let φ and ψ be propositional formulæ. Then the following statements are true:*

1. $\mathcal{R}(\varphi \wedge \psi) \subseteq \mathcal{R}(\varphi) \cap \mathcal{R}(\psi) \subseteq \mathcal{R}(\varphi)$
2. $\mathcal{R}(\varphi) \subseteq \mathcal{R}(\varphi) \cup \mathcal{R}(\psi) \subseteq \mathcal{R}(\varphi \vee \psi)$
3. *If* $\text{FreeVars}(\varphi) \cup \text{FreeVars}(\psi) \subseteq X$ *then* $\mathcal{R}(\varphi) \cap \mathcal{R}(\psi) = \mathcal{R}(\varphi \wedge \psi)$
4. $\mathcal{R}(\varphi) \cup \mathcal{R}(\psi) = \mathcal{R}(\varphi \vee \psi)$

Proof. Let us prove the four points separately:

1. If $\varphi \wedge \psi$ is not satisfiable then $\mathcal{R}(\varphi \wedge \psi) = \varnothing$ and the inclusions hold trivially. Assuming that it is satisfiable, let $\rho = [I]\varrho \in \mathcal{R}(\varphi \wedge \psi)$; then $I \models \varphi \wedge \psi$ and necessarily $I \models \varphi$ and $I \models \psi$, so $\rho \in \mathcal{R}(\varphi)$ and $\rho \in \mathcal{R}(\psi)$, that is to say $\rho \in \mathcal{R}(\varphi) \cap \mathcal{R}(\psi)$.
2. If φ is not satisfiable then trivially $\mathcal{R}(\varphi) = \varnothing \subseteq \mathcal{R}(\varphi \vee \psi)$; else let $\rho = [I]\varrho \in \mathcal{R}(\varphi)$; then, since $I \models \varphi$, we also have $I \models \varphi \vee \psi$ and it follows that $\rho \in \mathcal{R}(\varphi \vee \psi)$. By the same arguments, we have also $\mathcal{R}(\psi) \subseteq \mathcal{R}(\varphi \vee \psi)$. Thus $\mathcal{R}(\varphi) \cup \mathcal{R}(\psi) \subseteq \mathcal{R}(\varphi \vee \psi)$.
3. Let φ and ψ be propositional formulæ such that $\text{FreeVars}(\varphi) \subseteq X$ and $\text{FreeVars}(\psi) \subseteq X$. We only need to prove the inclusion $\mathcal{R}(\varphi) \cap \mathcal{R}(\psi) \subseteq \mathcal{R}(\varphi \wedge \psi)$. If either φ or ψ is not satisfiable we have trivially $\mathcal{R}(\varphi) \cap \mathcal{R}(\psi) = \varnothing \subseteq \mathcal{R}(\varphi \wedge \psi)$. Assuming that both are satisfiable, if $\mathcal{R}(\varphi) \cap \mathcal{R}(\psi) = \varnothing$ then we have the trivial inclusion again, else let I and J be two valuations such that $I \models \varphi$ and $J \models \psi$ and ρ such that $\rho = [I]\varrho \in \mathcal{R}(\varphi)$ and $\rho = [J]\varrho \in \mathcal{R}(\psi)$. By lemma 26 we can define $\widetilde{I} = I|_X = J|_X$. Since $\text{FreeVars}\,\varphi \subseteq X$ and $I \models \varphi$, any extension of $\widetilde{I} = I|_X$ models φ. The same goes for ψ. So any extension of $\widetilde{I}$ models both φ and ψ. So we have $\rho = [\widetilde{I}]\varrho \in \mathcal{R}(\varphi \wedge \psi)$. Therefore the inclusion $\mathcal{R}(\varphi) \cap \mathcal{R}(\psi) \subseteq \mathcal{R}(\varphi \wedge \psi)$ holds.
4. We only need to prove the inclusion $\mathcal{R}(\varphi \vee \psi) \subseteq \mathcal{R}(\varphi) \cup \mathcal{R}(\psi)$. If neither φ nor ψ are satisfiable, then the inclusion holds trivially; else let us take any $\rho = [I]\varrho \in \mathcal{R}(\varphi \vee \psi)$. Then we have, say, $I \models \varphi$; therefore $\rho \in \mathcal{R}(\varphi)$ and *a fortiori* $\rho \in \mathcal{R}(\varphi) \cup \mathcal{R}(\psi)$. The same holds if $I \models \psi$.

□

Lemma 28 (Partition lemma)**.** *Let φ be a propositional formula. Then the following statements hold:*

1. *If* $\text{FreeVars}(\varphi) \subseteq X$ *then* $\mathcal{R}(\varphi) \cap \mathcal{R}(\neg\varphi) = \varnothing$

2. $\mathcal{R}(\varphi) \cup \mathcal{R}(\neg\varphi) = \mathcal{P}os(t) \times Q$

Proof. This is a direct corollary of the chain lemma:

1. If $\text{FreeVars}(\varphi) \subseteq X$ then $\mathcal{R}(\varphi) \cap \mathcal{R}(\neg\varphi) = \mathcal{R}(\varphi \wedge \neg\varphi) = \mathcal{R}(\bot) = \varnothing$.
2. $\mathcal{R}(\varphi) \cup \mathcal{R}(\neg\varphi) = \mathcal{R}(\varphi \vee \neg\varphi) = \mathcal{R}(\top) = \mathcal{P}os(t) \times Q$.

□

Now we have the building blocks of our formula, and have linked them to a notion of relation; but a run is more than just a relation, it is a *function* from $\mathcal{P}os(t)$ to Q, which forbids situations in which we get in more than one state at one given position. So let us write the formula which formalises this constraint, making our relation a partial function:

Definition 29 (Partial function constraint $\Theta_{\twoheadrightarrow}$)**.**

$$\Theta_{\twoheadrightarrow} \overset{\text{def}}{=} \bigwedge_{\substack{\alpha \in \mathcal{P}os(t) \\ q \in Q}} \left[X_q^\alpha \implies \bigwedge_{\substack{p \in Q \\ p \neq q}} \neg X_p^\alpha \right]$$

Lemma 30 (Partial functions)**.** *The set $\mathcal{P}os(t) \twoheadrightarrow Q$ of all partial functions from $\mathcal{P}os(t)$ to Q is equal to the set of all inferred run relations on the partial function constraint: $\mathcal{P}os(t) \twoheadrightarrow Q = \mathcal{R}(\Theta_{\twoheadrightarrow})$.*

Proof. Let $\rho = [I]\varrho \in \mathcal{R}(\Theta_{\twoheadrightarrow})$. Let us assume that there exists $p, q \in Q, p \neq q$ and $\alpha \in \mathcal{P}os(t)$ such that $(\alpha, q) \in \rho$ and $(\alpha, p) \in \rho$, that is to say ρ is not a partial function. Then $I \models X_q^\alpha, X_p^\alpha$. But by definition $I \models \Theta_{\twoheadrightarrow}$, and it follows that $I \models \bigwedge_{q' \neq q} \neg X_{q'}^\alpha$, and consequently $I \models X_p^\alpha \wedge \neg X_p^\alpha$, which is absurd. Therefore ρ is a partial function. Conversely, let $\rho \notin \mathcal{R}(\Theta_{\twoheadrightarrow})$; then by the partition lemma 28, given that $\text{FreeVars}(\Theta_{\twoheadrightarrow}) \subseteq X$, it follows that $\rho = [I]\varrho \in \mathcal{R}(\neg\Theta_{\twoheadrightarrow})$. Since we have

$$\neg\Theta_{\twoheadrightarrow} = \bigvee_{\substack{\alpha \in \mathcal{P}os(t) \\ q \in Q}} \left[X_q^\alpha \wedge \bigvee_{\substack{p \in Q \\ p \neq q}} X_p^\alpha \right]$$

there exist $\alpha \in \mathcal{P}os(t), q \in Q$ and $p \in Q, p \neq q$ such that $I \models X_p^\alpha \wedge X_q^\alpha$, and so ρ is not a partial function. Finally we have $\rho \in \mathcal{R}(\Theta_{\twoheadrightarrow}) \iff \rho \in \mathcal{P}os(t) \twoheadrightarrow Q$. □

We need more work in order to make this partial function into a run: it must be compatible with the transition rules of $\mathcal{A}'$. Let us translate into propositional logic the fact that a transition rule applies at a given position $\alpha \ldots$

Definition 31 (Rule application constraint $\Psi^\alpha(r)$)**.** We define, for any $\alpha \in \mathcal{P}os(t)$, and any transition rule $f(q_1, \dots, q_n) \to q \in \Delta$

$$\Psi^\alpha\Big(f(q_1, \dots, q_n) \to q\Big) \stackrel{\text{def}}{=} X^\alpha_q \wedge \bigwedge_{k=1}^{n} X^{\alpha.k}_{q_k}.$$

This is fairly straightforward: we are simply stating that the rule $f(q_1, \dots, q_n) \to q \in \Delta$ applies at position α. Therefore we have $\varrho(\alpha) = q$ as a result of the application of the rule and the direct subterms are in the states q_k, as a requirement of said rule. Now, in order to express the notion of compatibility with the transition rules, we will recursively apply this function on the tree t, stating that at each position, one of the rules which can apply does so, unless of course there is no run over that term.

Definition 32. For any $f \in \Sigma$, we denote by $\Delta_f = \{ f(\dots) \to \cdots \in \Delta \}$ the set of transition rules which apply on a symbol f.

Definition 33 (Rules compatibility constraint $\Phi^\alpha(u)$)**.** We define, for any subterm $f(u_1, \dots, u_n) \trianglelefteq t$

$$\Phi^\alpha\Big(f(u_1, \dots, u_n)\Big) \stackrel{\text{def}}{=} \bigvee_{r \in \Delta_f} \Psi^\alpha(r) \wedge \bigwedge_{k=1}^{n} \Phi^{\alpha.k}(u_k)\,.$$

Now let us see how we can use these new constraints to restrain ourselves to accepting runs for the underlying tree automaton $\mathcal{A}'$. We need $\varrho \in \mathcal{R}(\Theta_{\twoheadrightarrow})$ to satisfy three more properties:

1. ϱ must be a total function. We shall see in the next lemma that if ϱ is compatible with the transition rules, it must be total.
2. ϱ must be compatible with the transition rules.
3. ϱ must associate a final state to the root ε.

We shall deal with those three requirements in the three next lemmas.

Lemma 34. *The following set inclusion holds:* $\mathcal{R}\Big(\Theta_{\twoheadrightarrow} \wedge \Phi^\varepsilon(t)\Big) \subseteq Q^{\mathcal{P}os(t)}$.

Proof. Let $\rho = [I]\varrho \in \mathcal{R}\Big(\Theta_{\twoheadrightarrow} \wedge \Phi^\varepsilon(t)\Big)$. Then by lemma 30 and the chain lemma 27 we have $\rho \in \mathcal{P}os(t) \twoheadrightarrow Q$. Let us prove that its domain is in fact the whole of $\mathcal{P}os(t)$. We have by definition

$$\Phi^\varepsilon(t) = \bigwedge_{\alpha \in \mathcal{P}os(t)} \left[\bigvee_{r \in \Delta_{t(\alpha)}} \Psi^\alpha(r) \right].$$

Let $\alpha \in \mathcal{P}os(t)$ and let us note $f = t(\alpha)$; then $I \models \bigvee_{r \in \Delta_f} \Psi^\alpha(r)$ and therefore there exists a transition rule $r = f(q_1, \ldots, q_n) \to q \in \Delta$ such that $I \models \Psi^\alpha(r)$, which in turn implies that $I \models X_q^\alpha$, that is to say ρ is defined on α and $\rho(\alpha) = q$. □

Lemma 35 (Tree automaton runs)**.** *The set of all runs over the term t for the underlying tree automaton $\mathcal{A}'$ is $\mathcal{R}\Big(\Theta_{\twoheadrightarrow} \wedge \Phi^\varepsilon(t)\Big)$.*

Proof. Let $\rho = [I]\varrho \in \mathcal{R}\Big(\Theta_{\twoheadrightarrow} \wedge \Phi^\varepsilon(t)\Big)$. Then $\rho \in Q^{\mathcal{P}os(t)}$ by lemma 34. Let us suppose that ρ is not a run, that is to say that there exists a position $\alpha \in \mathcal{P}os(t)$ where ρ is incompatible with the transition rules: in other words, for all $f(q_1, \ldots, q_n) \to q \in \Delta$ we have $f = t(\alpha) \implies q \neq \rho(\alpha) \vee \exists k \in [\![1, n]\!] \mid q_k \neq \rho(\alpha.k)$, and so there is no $r \in \Delta_{t(\alpha)}$ such that $I \models \Psi^\alpha(r)$. Therefore $I \not\models \Phi^\varepsilon(t)$ and *a fortiori* $I \not\models \Theta_{\twoheadrightarrow} \wedge \Phi^\varepsilon(t)$ which is absurd. This proves that ρ is a run. Conversely, let ρ be a run. Then $\rho \in Q^{\mathcal{P}os(t)}$, but $Q^{\mathcal{P}os(t)} \subseteq \mathcal{R}(\Theta_{\twoheadrightarrow})$ so there exists I such that $\rho = [I]\varrho \in \mathcal{R}(\Theta_{\twoheadrightarrow})$. Let us show that $\rho \in \mathcal{R}\Big(\Phi^\varepsilon(t)\Big)$. By definition of a run, for all $\alpha \in \mathcal{P}os(t)$, there exists a rule $t(\alpha)(q_1, \ldots, q_n) \to q \in \Delta$ such that $\rho(\alpha) = q$ (*ie.* $I \models X_q^\alpha$) and $\forall k \in [\![1, n]\!], \rho(\alpha.k) = q_k$ (*ie.* $\forall k \in [\![1, n]\!], I \models X_{q_k}^{\alpha.k}$). So for all α there is a rule r such that $I \models \Psi^\alpha(r)$, and thus $I \models \Phi^\varepsilon(t)$. Finally, we have $\rho \in \mathcal{R}(\Theta_{\twoheadrightarrow}) \cap \mathcal{R}\Big(\Phi^\varepsilon(t)\Big)$, and we conclude by applying the chain lemma that $\rho \in \mathcal{R}\Big(\Theta_{\twoheadrightarrow} \wedge \Phi^\varepsilon(t)\Big)$. □

Lemma 36. *The set of all successful runs over the term t for the underlying tree automaton $\mathcal{A}'$ is $\mathcal{R}\Big(\Theta_{\twoheadrightarrow} \wedge \Phi^\varepsilon(t) \wedge \bigvee_{q \in F} X_q^\varepsilon\Big)$.*

Proof. Let $\rho \in \mathcal{R}(\Theta_{\twoheadrightarrow} \wedge \Phi^\varepsilon(t) \wedge \bigvee_{q \in F} X_q^\varepsilon)$. Then by the chain lemma ρ is a run, and $\rho \in \mathcal{R}\Big(\bigvee_{q \in F} X_q^\varepsilon\Big)$, thus there exists a $q \in F$ such that $\rho(\varepsilon) = q$. This make ρ and accepting run. Conversely, let ρ be a successful run. Since it is a run, we have $\rho = [I]\varrho \in \mathcal{R}\Big(\Theta_{\twoheadrightarrow} \wedge \Phi^\varepsilon(t)\Big)$. And since it is accepting, $\rho(\varepsilon) \in F$, thus there exists a final state $q \in F$ such that $\rho(\varepsilon) = q$ (*ie.* $I \models X_q^\varepsilon$). It follows that $I \models \bigvee_{q \in F} X_q^\varepsilon$ and so $\rho \in \mathcal{R}\Big(\bigvee_{q \in F} X_q^\varepsilon\Big)$. We conclude by applying the chain lemma. Again. □

Definition 37 (Accepting tree automaton run constraint: $\Theta_\star$)**.** For short, we define $\Theta_\star \stackrel{\text{def}}{=} \Theta_{\twoheadrightarrow} \wedge \Phi^\varepsilon(t) \wedge \bigvee_{q \in F} X_q^\varepsilon$

So in terms of satisfiability, this means that a term t is accepted by $\mathcal{A}'$ if and only if $\Theta_\star$ is satisfiable. The perceptive reader will have noticed that until now, all we have done is to "reduce" a relatively simple problem – membership for tree automaton is *ALOGTIME* [(a)]-complete – to a relatively hard one: despite

[(a)] Logarithmic time alternating RAM. *ALOGTIME* is the class of languages decidable in logarithmic time by a random access alternating Turing machine.

the existence of highly optimised solvers, the *SAT* problem remains a canonical *NP*-complete problem. So let us remind ourselves that our aim was (and still is) to find a formula for `TAGED`, and not merely for traditional tree automata. An accepting run for a `TAGED` is an accepting run for the underlying tree automaton which is compatible with the global state equality and difference constraints $=_{\mathcal{A}}$ and $\neq_{\mathcal{A}}$. Therefore we need additional propositional formulæ to enforce these conditions. However we see that we will need to express constraints concerning terms (or rather, subterms) and states, regardless of position, which is new since until now we had just needed to express constraints on positions and states. Thus it appears that we need new building blocks for our propositional formula.

Definition 38 (Terms variables)**.** For any $q \in Q$ and $u \trianglelefteq t$, we let T^q_u be a propositional variable. We call these variables *term variables*, and we denote

$$T \stackrel{\text{def}}{=} \left\{ T^q_u \mid q \in Q \text{ and } u \trianglelefteq t \right\}$$

the set of all term variables.

The idea here is that if we have two subtrees u and v whose roots are in the same state q, T^q_u and T^q_v will be the same variable if and only if u and v are, structurally, the same tree, for instance $u = v = f(a, g(b))$. But of course, this can only be of use if we link, somehow, the notion of run to the structure of the term, by putting the subtrees in relation with the states of the run. This is the object of the next definitions and lemmas.

Definition 39 (Structural glue: $\Theta_{\leftrightarrows}$)**.**

$$\Theta_{\leftrightarrows} \stackrel{\text{def}}{=} \bigwedge_{\substack{\alpha \in \mathcal{P}os(t) \\ q \in Q}} \left[X^\alpha_q \implies T^q_{t|_\alpha} \right] \quad \text{and} \quad \Theta^\star \stackrel{\text{def}}{=} \Theta_\star \wedge \Theta_{\leftrightarrows}$$

The idea which this formula formalises is simply this: let us suppose that at a certain position α in our term t, we are – by an accepting run for the underlying tree automaton – in a certain state q. Then we simply assert that we have seen the current subtree $u = t|_\alpha$ in state q, which the term variable $T^q_{t|_\alpha} = T^q_u$ symbolises. Keeping this in mind, we will from now on assume that this formula is satisfied (in the informal explanations), and translate $\neq_{\mathcal{A}}$ and $=_{\mathcal{A}}$ into propositional logic. We decide to start with $\neq_{\mathcal{A}}$ since the formula is simpler.

Definition 40 (Compatibility with $\neq_{\mathcal{A}}$: $\Theta_{\neq_{\mathcal{A}}}$)**.**

$$\Theta_{\neq_{\mathcal{A}}} \stackrel{\text{def}}{=} \bigwedge_{\substack{\alpha \in \mathcal{P}os(t) \\ q \in Q}} \left[X^\alpha_q \implies \bigwedge_{\substack{p \in Q \\ p \neq_{\mathcal{A}} q}} \neg T^p_{t|_\alpha} \right]$$

Through this formula, we assert that none of the states p which must be different from q – as defined in $\neq_{\mathcal{A}}$ – can have the same subtree as q. We are assuming the structural glue here, so at each position we have already stated which subtree we have using the appropriate term variable. Then if, at another position β in one of those states $p \neq_{\mathcal{A}} q$, we ever run across a subterm $t|_\beta = t|_\alpha = u$, we will immediately have the contradiction $T_u^p \wedge \neg T_u^p$, and we will reject t. This informal argument outlines the proof of lemma 45.

We will need the same sort of gymnastic to obtain compatibility with $=_{\mathcal{A}}$, but it will be slightly more costly since we need to translate the idea that "if we are in state q at a certain subtree u, then, for all states $p =_{\mathcal{A}} q$ we must have this same subtree and no other". How is this more costly? Because this time around we need to enumerate not only a group of states, but for each state we need to specify all the terms which we *cannot* have. Let us be reminded that, since the property which interests us is satisfiability, the only tool we have is the introduction of contradictions. It is not enough to say for instance "for all p in the equivalence class of q, T_u^p", because even if we got, say, T_v^p, for $v \neq u$, then the formula would stay satisfiable. It is therefore necessary to state preemptively that for any $v \neq u$, $\neg T_v^p$. With this in mind, let us introduce the last formula (for this section, at least).

Definition 41 (Compatibility with $=_{\mathcal{A}}$: $\Theta_{=_{\mathcal{A}}}$)**.**

$$\Theta_{=_{\mathcal{A}}} \stackrel{\text{def}}{=} \bigwedge_{\substack{\alpha \in \mathcal{P}\!bs(t) \\ q \in Q}} \left[X_q^\alpha \implies \bigwedge_{\substack{p \in Q \\ p =_{\mathcal{A}} q}} \bigwedge_{\substack{u \trianglelefteq t \\ u \neq t|_\alpha}} \neg T_u^p \right]$$

We shall now prove our informal claims that these formulæ translate the notion of compatibility with the global constraints.

Definition 42 (Canonical valuation)**.** Let $\rho \subseteq \mathcal{P}\!bs(t) \times Q$; then its canonical valuation $\mathcal{V}(\rho)$ is a valuation such that $\rho = [\mathcal{V}(\rho)]\varrho$ and for any $T_u^q \in T$, $\mathcal{V}(\rho) \models T_u^q \iff \exists \alpha \in \mathcal{P}\!bs(t) \mid (\alpha, q) \in \rho$ and $u = t|_\alpha$.

Lemma 43 (Structural glue)**.** *Let φ be any propositional formula such that* FreeVars$(\varphi) \subseteq X$*; then for any $\rho \in \mathcal{R}(\varphi)$ we have $\mathcal{V}(\rho) \models \varphi \wedge \Theta_{\leftrightarrows}$, and furthermore $\mathcal{R}(\varphi) = \mathcal{R}(\varphi \wedge \Theta_{\leftrightarrows})$.*

Proof. Let $\rho = [I]\varrho \in \mathcal{R}(\varphi)$. By the X-restriction lemma we have $I|_X = \mathcal{V}(\rho)\big|_X$. But since FreeVars$(\varphi) \subseteq X$ it follows that $\mathcal{V}(\rho) \models \varphi$ and thus we can write

$\rho = [\mathcal{V}(\rho)]\varrho \in \mathcal{R}(\varphi)$. Let us show that $\mathcal{V}(\rho) \models \Theta_{\leftrightarrows}$. Suppose that $\mathcal{V}(\rho) \models \neg\Theta_{\leftrightarrows}$; then

$$\mathcal{V}(\rho) \models \bigvee_{\substack{\alpha \in \mathcal{P}os(t) \\ q \in Q}} \left[X_q^\alpha \wedge \neg T^q_{t|_\alpha} \right]$$

and there exist α, q such that $\mathcal{V}(\rho) \models X_q^\alpha \wedge \neg T^q_{t|_\alpha}$. But this is in contradiction with the way in which $\mathcal{V}(\rho)$ has been defined. Thus $\rho = [\mathcal{V}(\rho)]\varrho \in \mathcal{R}(\varphi \wedge \Theta_{\leftrightarrows})$. The reciprocal inclusion is a direct application of the chain lemma. □

This might seem strange at first glance: we are adding a new constraint yet we still get exactly the same set in the end... What then is the point of doing such a thing? The point is simply to have a memory of the structure of the term; we get exactly the same set of inferred run relations, yes, but we do know something new about them: we know that the valuations they are inferred from are models to $\Theta_{\leftrightarrows}$, and therefore we have a "memory" of visited subterms. What we are doing in this lemma is simply state that having such a memory does not change anything so long as we did not have any preexisting structural constraint, that is to say so long as our formula did not use any term variable. Incidentally, this is the case for $\Theta_\star$.

Corollary 44 (Glued tree automaton accepting runs)**.** *The set $\mathcal{R}(\Theta_\star)$ of all successful runs over the term t for the underlying tree automaton $\mathcal{A}'$ is equal to the set $\mathcal{R}(\Theta_\star \wedge \Theta_{\leftrightarrows}) = \mathcal{R}(\Theta^\star)$.*

Proof. Consequence of lemmas 36 and 43. The latter applies because $\mathrm{FreeVars}(\Theta_\star) \subseteq X$. □

Lemma 45. *The set of all successful runs over the term t for the underlying tree automaton $\mathcal{A}'$ which are also compatible with the global disequality constraint $\neq_{\mathcal{A}}$ is equal to $\mathcal{R}(\Theta^\star \wedge \Theta_{\neq_{\mathcal{A}}})$.*

Proof. Let $\rho = [I]\varrho \in \mathcal{R}(\Theta^\star \wedge \Theta_{\neq_{\mathcal{A}}})$. Then by the chain lemma and the above corollary ρ is an accepting run for $\mathcal{A}'$ and $\rho \in \mathcal{R}(\Theta_{\leftrightarrows} \wedge \Theta_{\neq_{\mathcal{A}}})$, which means that $I \models \Theta_{\leftrightarrows} \wedge \Theta_{\neq_{\mathcal{A}}}$. Let us now assume that it is not compatible with $\neq_{\mathcal{A}}$. Therefore there exists $\alpha, \beta \in \mathcal{P}os(t)$ such that $\rho(\alpha) \neq_{\mathcal{A}} \rho(\beta)$ and $t|_\alpha = t|_\beta$. Let us take the following notations: $p = \rho(\alpha), q = \rho(\beta), u = t|_\alpha = t|_\beta$. Since $I \models \Theta_{\leftrightarrows} \wedge \Theta_{\neq_{\mathcal{A}}}$ we have

$$I \models T_u^p \wedge \bigwedge_{p' \neq_{\mathcal{A}} p} \neg T_u^{p'} \quad \text{and} \quad I \models T_u^q \wedge \bigwedge_{q' \neq_{\mathcal{A}} q} \neg T_u^{q'}.$$

And because $p \neq_{\mathcal{A}} q$, it follows that

$$I \models T_u^p \wedge \neg T_u^q \quad \text{and} \quad I \models T_u^q \wedge \neg T_u^p,$$

which is contradictory in both cases. So ρ must be compatible with $\neq_{\mathcal{A}}$. Conversely, let ρ be a successful run over the term t for the underlying tree automaton $\mathcal{A}'$ which is also compatible with the global disequality constraint $\neq_{\mathcal{A}}$. Let us show that $\rho \in \mathcal{R}(\Theta^\star \wedge \Theta_{\neq_{\mathcal{A}}})$, that is to say that there exists a valuation I such that $I \models \Theta^\star \wedge \Theta_{\neq_{\mathcal{A}}}$ and $\rho = [I]\varrho$. Let us show that the canonical valuation $\mathcal{V}(\rho)$ is suitable. By its definition, we have $\rho = [\mathcal{V}(\rho)]\varrho$, so it only remains to show that $\mathcal{V}(\rho) \models \Theta^\star \wedge \Theta_{\neq_{\mathcal{A}}}$. By lemma 43 and its corollary 44 we have directly $\mathcal{V}(\rho) \models \Theta^\star$, hence there only remains to show that $\mathcal{V}(\rho) \models \Theta_{\neq_{\mathcal{A}}}$. Assume that this is not the case. We have

$$\mathcal{V}(\rho) \models \neg\Theta_{\neq_{\mathcal{A}}} = \bigvee_{\substack{\alpha \in \mathcal{P}os(t) \\ q \in Q}} \left[X_q^\alpha \wedge \bigvee_{\substack{p \in Q \\ p \neq_{\mathcal{A}} q}} T^p_{t|_\alpha} \right]$$

and so there exist $\alpha \in \mathcal{P}os(t), q = \rho(\alpha) \in Q$ and $p \neq_{\mathcal{A}} q$ such that $\mathcal{V}(\rho) \models X_q^\alpha \wedge T^p_{t|_\alpha}$. Since $\rho(\alpha) = q$, and the irreflexivity of $\neq_{\mathcal{A}}$ implies that $p \neq q$, by our definition of $\mathcal{V}(\rho)\big|_T$ there must exist $\beta \neq \alpha \in \mathcal{P}os(t)$ such that $p = \rho(\beta)$ and $t|_\alpha = t|_\beta$. But if this is the case, then ρ is not compatible with $\neq_{\mathcal{A}}$; this is absurd. It follows that $\mathcal{V}(\rho) \models \Theta^\star \wedge \Theta_{\neq_{\mathcal{A}}}$, which concludes the proof. □

[TODO] similar proof for $\Theta_{=_{\mathcal{A}}}$
[TODO] Glue all that and conclude

5.2 Some implementation details

This section lists some experimental details, which were too lengthy or distracting from the main ideas to be put elsewhere.

5.2.1 External tools and file formats

As mentioned in the previous sections, we made use of external tools, namely

picoSAT: http://fmv.jku.at/picosat/
: A SAT solver. It takes input in the DIMACS CNF file format.

MiniSAT2: http://minisat.se/
: Another SAT solver, which proved to be generally faster than the former in our experiments. It also takes input in the DIMACS CNF file format. This is the case for most modern SAT solvers.

The BAT: http://www.ccs.neu.edu/home/pete/bat/index.html
A converter to CNF, which generates output in the DIMACS CNF file format, from an input in S-expression-like syntax. The CNF conversion algorithm and the tool are described in [MV09, MSV07].

The DIMACS CNF format is a *de-facto* standard for representing propositional formulæ in CNF. Its syntax is extremely compact and basic. Comment lines are indicated by a leading `'c'`. Free variables of the formula are numbered, starting from 1, and the first (significative) line, which starts with `'p'`, indicates the number of variables and the number of clauses in the formula, in this order. The following lines each represent a clause, where a literal is denoted by the corresponding positive number if it is an atom, and by the corresponding negative number if it is a negation of the atom. Each line is terminated by a zero. Let for instance φ be the following CNF boolean formula:

$$\varphi = (X \vee \neg Z) \wedge (Y \vee Z \vee \neg X).$$

Then it is encoded by the following DIMACS CNF file:

```
c DIMACS CNF for φ
p cnf 3 2
1 -3 0
2 3 -1 0
```

5.2.2 Notes about the implemented tool

Our tool has been mentioned in section 4.2[p34]. Here we give some more elements about it.

We chose to implement it in the `Objective Caml` programming language, which is quite efficient and allows us to stay quite close to mathematical syntax. We favoured readability of the code over time and space optimisations. SAT solving is the hard part of the process, not formula generation and conversion, which are both polynomial (more precisely, quadratic in the worst case). Having an optimised implementation of those steps is a different problem, but one which remains – at least theoretically – easy. So we focused on SAT solving time in our experiments.

The tool takes as input a `TAGED` in a syntax close to that of `Timbuk` [FGT04], and a term, and generates the corresponding formula in the BAT's input format. Example inputs of our tools are given in figure 5.1.

```
(** TAGED Automaton for {f(x,x)} *)
  Taged fxxA
  Alphabet f a b
  States q qq qf
  Final qf
  Rules
    f qq qq : qf
    f q q   : q
    f q q   : qq
    a:q a:qq
    b:q b:qq
  Equal
    qq qq
  Different
    qq qf
```

```
f(f(a,a), f(a,a))
// in a_fxx
```

Figure 5.1 — Input syntax of the tool (see 5.2[p50])

The default mode of operation is of course to generate BAT input and to run BAT and a SAT solver immediately. But there is also the possibility, introduced mainly for my own benefit while I was implementing the formulæ, of outputting a LaTeX file detailing, in a user-friendly format, the input automaton, term and the corresponding generated formula. Figure 5.2[p50] gives the LaTeX output corresponding to the input of figure 5.1. Note that the tool indexes both subterms and and positions, in an effort to keep the formula somewhat readable even on high trees.

The tool is also capable of automatically generating (randomised or not) terms recognised by some of our test TAGEDs. This has been used for the experiments described in section 4.2[p34].

An experiment which was omitted in the paper is presented in figure 5.3[p51]. It shows that, in some cases, with full static optimisations as described in section 4.2, the generated CNF formula can grow linearly in the size of the input term. The reason why this was not mentioned in the article is that experiments are not needed to show that kind of result. It is certainly possible to determine, in theory, the class of inputs which satisfy this property. We have not yet gotten around to doing that, though.

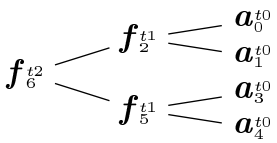
Other experiments were done, but they bring nothing new compared to those

Automaton: **fxxA**
Alphabet: $\{\boldsymbol{f}, \boldsymbol{a}, \boldsymbol{b}\}$
States: $\{\mathsf{q}, \mathsf{qq}, \mathsf{qf}\}$
Final States: $\{\mathsf{qf}\}$
Transition Rules: {

$\boldsymbol{f}(\mathsf{qq}, \mathsf{qq}) \to \mathsf{qf}$,
$\boldsymbol{f}(\mathsf{q}, \mathsf{q}) \to \mathsf{q}$,
$\boldsymbol{f}(\mathsf{q}, \mathsf{q}) \to \mathsf{qq}$,
$\boldsymbol{a} \to \mathsf{q}$,
$\boldsymbol{a} \to \mathsf{qq}$,
$\boldsymbol{b} \to \mathsf{q}$,
$\boldsymbol{b} \to \mathsf{qq}$

}
Global State Equality: $\{\mathsf{qq} = \mathsf{qq}\}$
Global State Disequality: $\{\mathsf{qq} \neq \mathsf{qf}\}$
End Automaton.

Term as expression: $\boldsymbol{f}\{\boldsymbol{f}[\boldsymbol{a}, \boldsymbol{a}], \boldsymbol{f}[\boldsymbol{a}, \boldsymbol{a}]\}$
Term as tree:



Membership formula = $[[(X^3_{\mathsf{q}} \vee X^3_{\mathsf{qq}}) \wedge ([X^5_{\mathsf{qf}} \wedge X^3_{\mathsf{qq}} \wedge X^4_{\mathsf{qq}}] \vee [X^5_{\mathsf{q}} \wedge X^3_{\mathsf{q}} \wedge X^4_{\mathsf{q}}] \vee [X^5_{\mathsf{qq}} \wedge X^3_{\mathsf{q}} \wedge X^4_{\mathsf{q}}]) \wedge ([X^2_{\mathsf{qf}} \wedge X^0_{\mathsf{qq}} \wedge X^1_{\mathsf{qq}}] \vee [X^2_{\mathsf{q}} \wedge X^0_{\mathsf{q}} \wedge X^1_{\mathsf{q}}] \vee [X^2_{\mathsf{qq}} \wedge X^0_{\mathsf{q}} \wedge X^1_{\mathsf{q}}]) \wedge (X^1_{\mathsf{q}} \vee X^1_{\mathsf{qq}}) \wedge ([X^6_{\mathsf{qf}} \wedge X^2_{\mathsf{qq}} \wedge X^5_{\mathsf{qq}}] \vee [X^6_{\mathsf{q}} \wedge X^2_{\mathsf{q}} \wedge X^5_{\mathsf{q}}] \vee [X^6_{\mathsf{qq}} \wedge X^2_{\mathsf{q}} \wedge X^5_{\mathsf{q}}]) \wedge (X^0_{\mathsf{q}} \vee X^0_{\mathsf{qq}}) \wedge (X^4_{\mathsf{q}} \vee X^4_{\mathsf{qq}})] \wedge [\neg X^6_{\mathsf{q}} \wedge \neg X^6_{\mathsf{qq}}] \wedge [\{X^3_{\mathsf{qq}} \Rightarrow T_0^{\mathsf{qq}}\} \wedge \{X^5_{\mathsf{qq}} \Rightarrow T_1^{\mathsf{qq}}\} \wedge \{X^2_{\mathsf{qq}} \Rightarrow T_1^{\mathsf{qq}}\} \wedge \{X^1_{\mathsf{qq}} \Rightarrow T_0^{\mathsf{qq}}\} \wedge \{X^6_{\mathsf{qq}} \Rightarrow T_2^{\mathsf{qq}}\} \wedge \{X^0_{\mathsf{qq}} \Rightarrow T_0^{\mathsf{qq}}\} \wedge \{X^4_{\mathsf{qq}} \Rightarrow T_0^{\mathsf{qq}}\}] \wedge [\{X^5_{\mathsf{qq}} \Rightarrow \neg T_1^{\mathsf{qf}}\} \wedge \{X^2_{\mathsf{qq}} \Rightarrow \neg T_1^{\mathsf{qf}}\} \wedge \{X^6_{\mathsf{qq}} \Rightarrow \neg T_2^{\mathsf{qf}}\}] \wedge [\{X^3_{\mathsf{qq}} \Rightarrow [\neg T_2^{\mathsf{qq}} \wedge \neg T_1^{\mathsf{qq}}]\} \wedge \{X^5_{\mathsf{qq}} \Rightarrow [\neg T_2^{\mathsf{qq}} \wedge \neg T_0^{\mathsf{qq}}]\} \wedge \{X^2_{\mathsf{qq}} \Rightarrow [\neg T_2^{\mathsf{qq}} \wedge \neg T_0^{\mathsf{qq}}]\} \wedge \{X^1_{\mathsf{qq}} \Rightarrow [\neg T_2^{\mathsf{qq}} \wedge \neg T_1^{\mathsf{qq}}]\} \wedge \{X^6_{\mathsf{qq}} \Rightarrow [\neg T_1^{\mathsf{qq}} \wedge \neg T_0^{\mathsf{qq}}]\} \wedge \{X^0_{\mathsf{qq}} \Rightarrow [\neg T_2^{\mathsf{qq}} \wedge \neg T_1^{\mathsf{qq}}]\} \wedge \{X^4_{\mathsf{qq}} \Rightarrow [\neg T_2^{\mathsf{qq}} \wedge \neg T_1^{\mathsf{qq}}]\}]]$

Figure 5.2 — Example LaTeX output of the tool (see 5.1[p49])

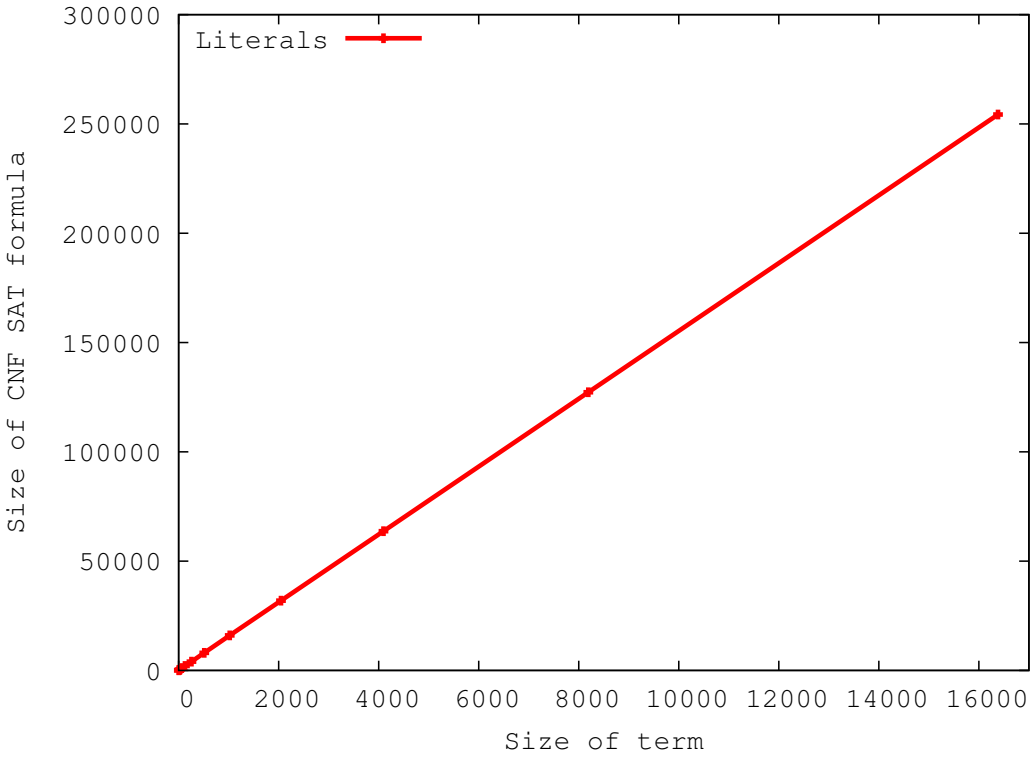


Figure 5.3 — Size of the formula for $\{ f(t,t) \mid f \in \Sigma, t \in \mathcal{T}(\Sigma) \}$

which have already been mentioned.

Chapter 6

Conclusion

As part of this project we have proposed a SAT encoding for the *NP*-complete `TAGED` membership problem, which we have proved, trimmed down, and implemented. Though experiments were limited, mainly by factors outside of our control and of the scope of this project, such as CNF generation, they remain fairly encouraging (see section 4.2[p34]). Indeed, despite the respectable size of the generated formulæ, SAT solving remains surprisingly fast.

There remain, of course, many paths to explore, notably to quicken formula generation in practice – which is for now the weak point of the process. As stated in sections 4.2 and 5.2.2[p48], given that those weak points are in theory by far the least computationally expensive, we feel confident that these steps *can* be implemented quite efficiently. An obvious practical optimisation would be to do away with the lengthy process

$$\text{input} \xrightarrow[\text{formula generation}]{} \text{BAT} \xrightarrow[\text{CNF generation}]{} \text{SAT} \xrightarrow[\text{solving}]{} \text{y/n}$$

and generate the CNF formula on-the-fly, while interfacing with a SAT solver. This would remove the need for the two intermediary file formats, and allow quicker detection of contradictions.

Another observation which could have a practical impact is that the formula, such as we have defined it, is already *mostly* in CNF, the exception being the subformula $\Phi^{\epsilon}(t)$. If this could be recoded directly in (reasonably sized) CNF, this would remove the need for the supplementary conversion step.

Overall we feel that our objectives have been reached, and that the fundamental difficulty of the problem (its *NP*-completeness) has been overcome – inasmuch as we could hope – by the chosen approach.

Part III

Generating Interesting TAGEDs Randomly

Chapter 7

Introduction and related work

Our aim during the internship was to develop efficient algorithms and heuristics to render the emptiness problem for TAGEDs tractable in as many cases as possible, in spite of its inherent complexity (*EXPTIME*-complete). However, while creating new algorithms in purely theoretical spheres is all well and good, at some point one has to confront one's ideas to experimental realities, especially when the goal one sets out to achieve is efficiency.

Unfortunately we could not find in the literature any well-established experimental protocol or test suite for tree automata with constraints; or even for vanilla tree automata. In an ideal world, we would have found and used a large database of "real-world" TAGEDs on which to evaluate our algorithms' efficiency. Since no such database exists, we had to generate our test TAGEDs randomly. While work has been done on generating non-deterministic finite automata [TV05] and deterministic top-down tree automata [HNS09], random generation of non-deterministic bottom-up tree automata did not seem to have been studied at all. Therefore, we had to come up with our own approaches to this problem.

Random generation of automata is not a trivial subject; there are many arbitrary probabilistic models which can be chosen; how relevant they are depends on what is expected of "real-life" applications. For us, it was not simply a question of generating *any* TAGEDs randomly; we also had to generate automata which were actually *interesting*, in the sense that it was necessary that they should allow us to discriminate between different algorithms for testing emptiness. Generating thousands of automata, all trivially empty or non empty, would be a complete waste of time. As we will see in the following sections, meeting this requirement proved to be a tad more tricky than anticipated. It was a mostly

iterative process where we would define a probabilistic model or a generation algorithm, then implement and test it against our emptiness algorithms and heuristics. Most of the time, it turned out that the generated automata were either not representative of real-world applications at all, or much too easily tractable to be of any value in our experiments. So we would go back to the drawing board and repeat the cycle with another approach. Fortunately, we managed to find an approach which was both efficient and satisfying enough, and a bit of an hybrid of our earlier attempts.

Organisation of this part: There are two logical steps to generating a `TAGED`: first, we must generate a random tree automaton. This is the object of the first chapter, while the second deals with the random generation of global constraints to turn our random tree automata into `TAGED`s. Note that we only concern ourselves with global equality constraints, as our algorithms deal exclusively with positive `TAGED`s.

Note that this part and the next (which concerns our algorithms for deciding the emptiness problem) are heavily cross-referenced. It is probably best to skim through this part at first, and come back to it as needed in order to make sense of the experimental result of the next part.

Implementation note: Our `OCaml` implementation of these generation methods used, unless otherwise specified, `OCaml`'s standard Pseudo-Random Number Generator (PRNG), which is described as a *Linear feedback shift register* [(a)]

[(a)] (References: Robert Sedgewick, "Algorithms", Addison-Wesley). It is seeded by a MD5-based PRNG.

Chapter 8

Generating random Tree Automata

In this chapter we propose and discuss four different approaches for generating random vanilla tree automata.

8.1 First model: dense generation

Our first approach was to try and adapt existing work; in the article [BHH+08a], the authors generate random TAGEDs in a way close to that described in [TV05] for non-deterministic finite automata. Let us first summarise their respective contributions:

- The article [TV05] introduced a probabilistic model for random generation of Nondeterministic Finite Automata (NFA), focused on the universality problem. Roughly, in order to generate a NFA $(\Sigma, Q, Q_0, F, \delta)$, they choose the alphabet Σ fixed to $\Sigma = \{0, 1\}$, one initial state, an arbitrary number of states $|Q| = 30$ – which can be considered a parameter of the model – and generate the transitions and final states according to the two metrics

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

  The value r can be thought of as the expected out-degree of each node of the associated graph, for each symbol σ. They argue that those two metrics r and f, called respectively *transition density* and *final state density*, cover interesting behaviours as they vary. This model has been used for instance in [DWDHR06], also for the universality problem.

◇ In [BHH+08a], experiments on random Tree Automata were performed in a manner which was a straightforward translation of [TV05][(a)] to the context of Tree Automata, with $|Q| = 20$. In this context, they defined the transition and final state densities as follows:

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

In other words, the transition density is defined as the average number of different right-hand side states for any given left-hand side of a transition rule.

This being said, the *exact* experimental protocol which they followed is not detailed in the article, and those metrics do not cover all possible axes; for instance, it is unknown what alphabet Σ they used, and more importantly, the number of transition rules $|\Delta|$ is unspecified. Given that their definition of the density of transitions does not depend directly upon $|Q|$, there is no way to deduce it from the other parameters; it could be pretty much anything.

For the reasons given above, we will not follow exactly the same experimental protocol as in [BHH+08a] – in particular because, as mentioned, some aspects of it are undefined in the paper – but we will still retain the ideas of transition and final state densities introduced in [TV05], which we will adapt to the context of tree automata.

In this first probabilistic model, hereafter referred to as "generation 1" or "Gene1" for brevity, we generate a random Tree Automaton by first taking a fixed alphabet $\Sigma = \{a, b, c/_0, f, g, h/_2\}$, and a number of states $|Q|$, which is a parameter of our model. Considering that a rule $f(q_1, \ldots, q_{arity(f)}) \to q \in \Delta$ is nothing more than a tuple $(f, q_1, \ldots, q_{arity(f)}, q) \in \Sigma_{arity(f)} \times Q^{arity(f)+1}$, we have

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

and we shall determine Δ by choosing each rule in the space of all possible rules $\overline{\Delta}$ with probability p_Δ, another parameter of the model. Lastly, the final states of F are chosen in the same way: each state $q \in Q$ becomes final with probability p_F.

Note that our choice of using probabilities instead of densities was made purely

[(a)] Though, strangely, they do not cite it.

to simplify implementation a bit, and does not change anything when a large enough number of automata are generated. For instance, in the case of the final states, if we take X to be the random variable corresponding to the number of final states chosen, we have $X \sim \mathcal{B}(|Q|, p_F)$ (where $\mathcal{B}$ is the Binomial distribution), and so the expected value of X (as well as the median, in fact) is $p_F |Q|$. So on average we have $|F| \approx p_F |Q|$ and thus $p_F \approx {}^{|F|}/_{|Q|}$. Similarly, we have on average $|\Delta| \approx p_\Delta \left|\overline{\Delta}\right|$,*ie.* $p_\Delta \approx {}^{|\Delta|}/_{|\overline{\Delta}|}$. In other words, for a sufficient number of generated automata, using probabilities or densities amounts to the same results. The reader will have noticed that our notion of transition density is different from that of [BHH+08a]; in our work, it corresponds to the proportion of rules chosen, out of all possible rules, and not to the average number of right-hand states for a given left-hand side as in the aforementioned article.

To approximate the probability $p_\varnothing$ that a tree automaton recognises the empty language as a function of the parameters p_Δ and p_F, we test emptiness for 200 such generated automata. Experimental results are presented as a collection of contour plots in figure 8.1, where the different plots correspond to different values of $|Q|$, and where darker colours indicate a lesser probability of being empty. In each graph, $p_\varnothing$ takes values between 1 (white) and 0 (deepest blue). Let us examine the influence of each parameter of our model – and justify the choice which we have made of freezing other axes.

- ◇ Transition probability p_Δ: This is the parameter which exerts the most influence on the results: $p_\varnothing$ diminishes extremely fast as this parameter grows between 0 and approximately $\frac{2}{10}$, and stagnates afterwards.
- ◇ Final state probability p_F: This parameter has almost as much influence as p_Δ, acting in pretty much the same way, and also seems to stagnate after approximately $\frac{2}{10}$.
- ◇ Size of the states space $|Q|$: As this grows, the effects of the two previous parameters grow even more visible. It seems that the problem converges to a very sharp dichotomy, for $|Q|$ large enough: an automaton with p_Δ and p_F below a certain threshold – approximately $\frac{3}{20}$ – are almost certainly empty, and beyond that, almost certainly non-empty.
- ◇ The ranked alphabet Σ: This was not a parameter of our model, chosen fixed to $\Sigma = \{a, b, c/_0, f, g, h/_2\}$. However, we have conducted similar tests for the simpler $\Sigma = \{a/_0, f/_2\}$, and observed that this does not change the results in any fundamental way; this parameter seems to have pretty much

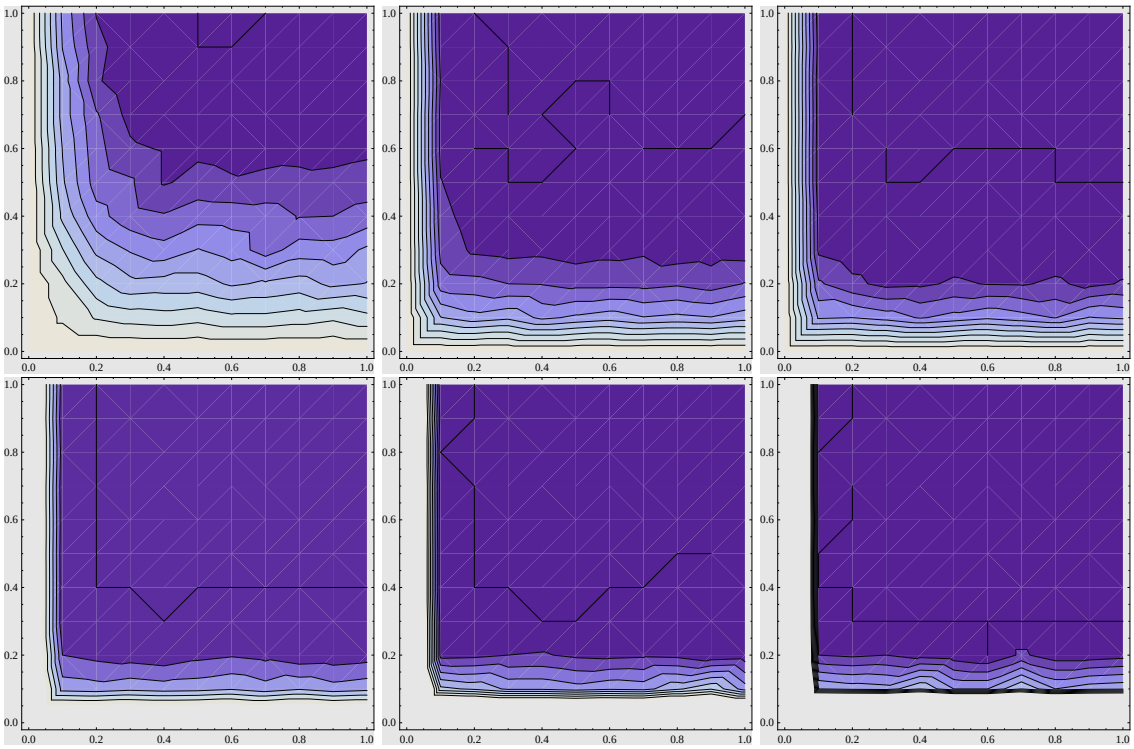


Figure 8.1 — $x = p_\Delta, y = p_F, z = p_\varnothing, |Q| = \{3, 8, 10, 13, 15, 20\}$

the same effects as $|Q|$. Overall, it seems sane to fix this and deal only with other parameters.

For information, given our choice of Σ, the total possible number of transition rules $\left|\overline{\Delta}\right|$ in the automata is given by the formula $\left|\overline{\Delta}\right| = 3\left(|Q| + |Q|^3\right)$, which yields $\{90, 1560, 3030, 6630, 10170, 24060\}$ for our selection of values of $|Q|$; so even for the relatively low values of $|Q|$ which we have used here, our automata can end up having lots of transitions. This is the main weakness of this first model: it generates automata which are very dense, and therefore are not very good representatives of real-world tree automata, which are rarely so. Another related weakness lies in the fact that rules for symbols of high arity are overly represented, simply because there are many more possible rules dealing with high arity symbols than with low arity ones. Given our choice of Σ, this simply means that we will have comparatively few leaf rules, which is not too bad. Another choice of Σ, for instance by adding a symbol σ of arity 10, would generate a set of rules where the numbers of σ-rules completely dwarfs the rest. Again, this is probably not really representative of real-world automata.

8.2 Second model: sparse generation

We shall try and address some of the weaknesses of the first model in this second approach: every notion defined in the previous section still applies unless explicitly specified otherwise.

The main problems with the high-density model were the high density of generated automata, and the overwhelming high-arity rules. Our intent is therefore to alleviate those problems by generating far less rules, and favouring small-arity rules among them. We shall still select our rules in $\overline{\Delta}$, but this time instead of using a fixed probability p_Δ, we shall make p_Δ a function of the arity of the rules. Furthermore, we shall express the density of the automaton in terms of the *expected in-degree* δ, which we define as the expected number of rules which yield q, for any state $q \in Q$. Thus we define[(b)]

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

[(b)] The definition of $\mathfrak{Nr}_\Sigma$ can be found at section 2.1.1[p11], should you need it.

We build Δ in the same way as before, with this new definition: we select each rule $f/_n(p_1, \ldots, p_n) \to q \in \overline{\Delta}$ with probability $p_\Delta(n)$. We compute the expected number of transitions :

$$|\Delta| = \sum_{k\in\mathbb{N}} p_\Delta(k) \cdot |\Sigma_k| \cdot |Q|^{k+1} = \sum_{k\in\mathfrak{Ar}_\Sigma} p_\Delta(k) \cdot |\Sigma_k| \cdot |Q|^{k+1} = |Q| \sum_{k\in\mathfrak{Ar}_\Sigma} p_\Delta(k) \cdot |\Sigma_k| \cdot |Q|^k$$
$$= |Q| \sum_{k\in\mathfrak{Ar}_\Sigma} \frac{\delta \cdot |\Sigma_k|\,|Q|^k}{|\mathfrak{Ar}_\Sigma| \cdot |\Sigma_k|\,|Q|^k} = |Q| \sum_{k\in\mathfrak{Ar}_\Sigma} \frac{\delta}{|\mathfrak{Ar}_\Sigma|} = \delta\,|Q|\,,$$

which is coherent with our definition of δ as the expected in-degree of each state, since we obtain on average δ rules for each state of Q. The in-degrees are nicely

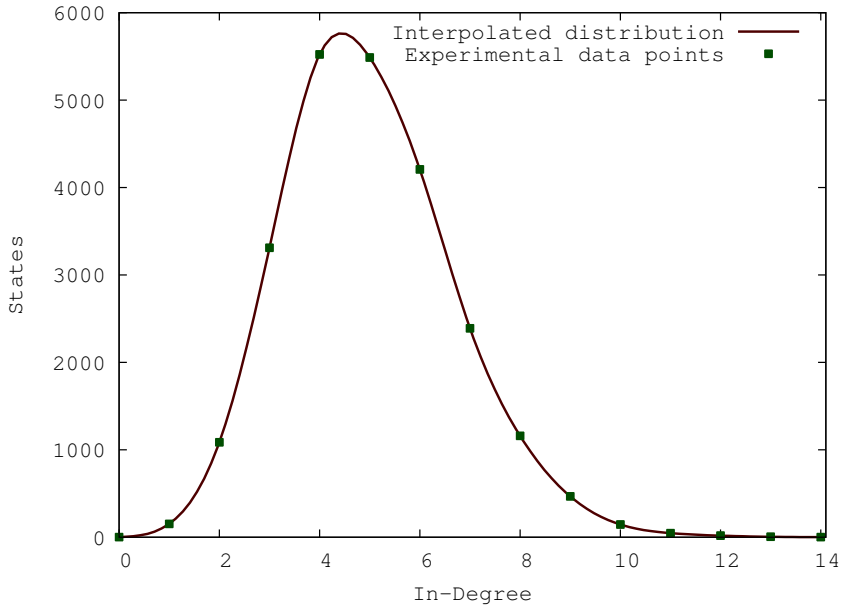


Figure 8.2 — In-degree distribution, $\delta = 5$

distributed[(c)], as is shown in figure 8.2[p62]. Each data point (x, y) corresponds to the number y of states which have an in-degree of x, for 1200 generated Tree Automata of 20 states each, and for $\delta = 5$. A spline-interpolated curve joining

[(c)] The exact distribution is difficult to compute in practice, as it is a sum of binomial random variables. This would be easy to compute if the probabilities were the same, as it is well-known that if $X_k \sim \mathcal{B}(n_k, p)$, then $\sum_k X_k \sim \mathcal{B}(\sum_k n_k, p)$, however in this case they have differing probabilities and this does not apply. Exact and approximate methods for computing this distribution are discussed in [BSS93]. We have not done it since the result is not important to this study.

the data points has been added as a visual guide. Figure 8.3[p63] shows the probability that a generated tree automaton be empty, as a function of δ, for various values of $|Q|$. Each datapoint was obtained through computation of 500

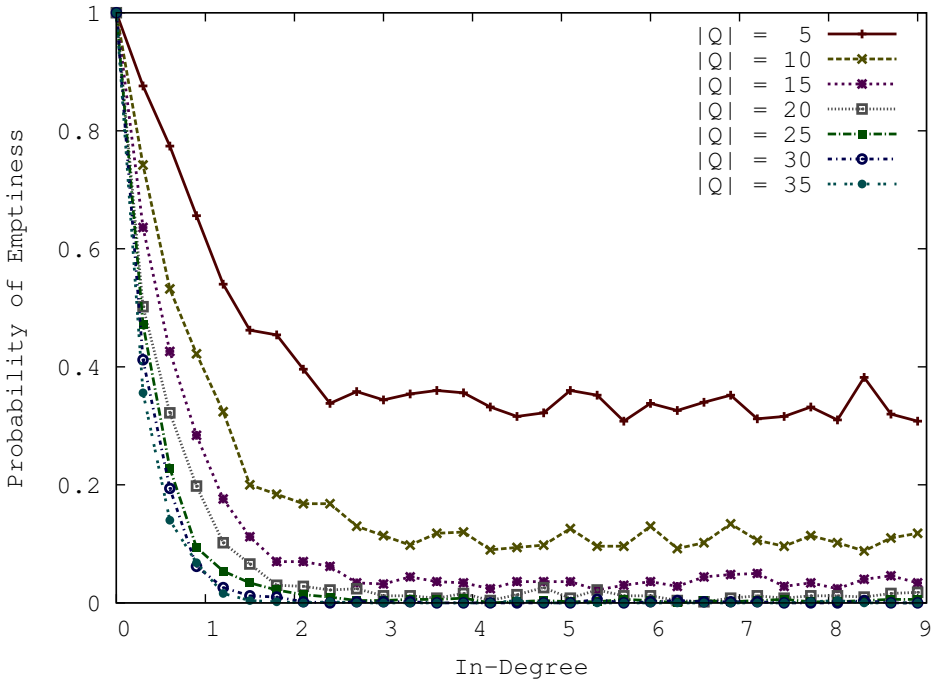


Figure 8.3 — Probability of emptiness as a function of δ

generated automata, for the indicated value of δ and $|Q|$ and a fixed value of $p_F = \frac{1}{5}$.

While more faithful to real-life tree automata in the sense that generated automata have reasonable density, and balanced rules with respect to arities, this model proved to be unsatisfactory for our purposes, for two main reasons.

1. **Dead branches:** Let us anticipate a little bit and jump forward to figure 12.2[p103], ignoring for now its abscissa, which represents a certain density of global equality constraints. Since we are only interested in vanilla tree automata for now, we need only read the values for a "corrected base probability" of zero, that is to say, when no constraints are generated. The figure shows the ratio

$$R = \frac{\text{size of generated automata}}{\text{size of the same, } \textit{reduced}},$$

where "reduced" means here that the standard reduction algorithm defined in [CDG+07] has been applied, and that useless states have been removed as per theorem 53[p91]. Note that these two operations are the only components of the "cleanup" operation which apply on vanilla tree automata. What this ratio R tells us is in essence how much of the generated automata is "dead", that is, unreachable or unconnected to any final state. $R = 30$ means that only $1/R = 1/30$ of all the states and rules we generate is actually useful. Needless to say, it is in our best interests to keep R as close to 1 as possible. But, as we see looking at figure 12.2, depending on the parameters of the model, R can become quite large, which greatly biases the results. Indeed, if the size of the underlying tree automata is cut down considerably by our simplest reduction methods, without even needing to look at the global constraints, then the efficiency of the `TAGED`-specific algorithms risks being grossly overestimated. To alleviate this concerns somewhat, we attempted generating raw tree automata R as large as possible, reducing them, and selecting the first one such that $|Q|$ comes – after reduction – within acceptable range of our initial aim. However, this method does nothing against the second criticism we have against this model:

2. **Way too easy...** As we mentioned in the introduction, our aim is to generate *interesting* automata, that is to say *difficult* cases for our algorithms. We will see later on that one variable which influences this perceived difficulty very heavily is the height of the smallest accepted term of the automaton, if any such term exists. It is clear that if any "leaf-term" is accepted by the tree automaton, then it is very easy to find (it is a matter of finding a rule $a \to q_f$, with $q_f \in F$, which is trivially done in linear time) and it is also accepted by any `TAGED` which extends the tree automaton (since there is just one node, none of the constraints applies). Therefore, the probability that a random `TAGED` be non-empty is at least as great as the probability that the underlying tree automaton accepts a leaf-term. So, let us ask ourselves the following question: with this model, what is the probability that a random tree automaton accepts a leaf-term? Let us denote L the expected number of generated leaf-rules – that is to say, rules of arity zero: since the rules are distributed evenly across all arities, we have
$$L = \frac{\delta\,|Q|}{|\mathfrak{Rr}_\Sigma|}.$$
By definition, the probability that a given leaf-rule is final if p_F. We are

looking for the probability that "there exists a final leaf-rule", *ie.* "*not* all leaf-rules are *non*-final". So the probability P which we are looking for is

$$P = 1 - (1 - p_F)^L = 1 - (1 - p_F)^{\frac{\delta|Q|}{|\mathfrak{Nr}_\Sigma|}}.$$

For the value of Σ which we have chosen, we have $|\mathfrak{Nr}_\Sigma| = |\{0, 1\}| = 2$ and let us take, for the sake of example, $\delta = 2$, and as usual $p_F = \frac{1}{5}$. Then we have $P = 1 - (\frac{4}{5})^{|Q|}$. In table 8.1[p65] we compute the values of $|Q|$ required to achieve certain key values of P: We see that, even for relatively low

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

Table 8.1 — Probability of final leaf-rules

values of $|Q|$, the chances that the generated automaton accepts some leaf is overwhelming. This is a symptom of a more general problem with the chosen approach: the generated automata tend to accept very small terms, if any. We made a doomed attempt to sidestep the problem by removing all final leaf-rules from the generated automaton, but this was clearly not enough, as the tests showed that even the most trivial emptiness algorithms were highly successful on test bases thus generated. In fact, the failure rate of the brutal algorithm was nearing 0%, regardless of the size of $|Q|$.

For the two reasons explained above, the second generation was deemed unsatisfactory.

8.3 Third model: skeleton-driven generation

For our third approach to random generation of tree automata, we tried to avoid the shortcomings observed in the previous generation; it occurred to us that we had, perhaps, attempted to address the question from the wrong angle. In the previous approaches, we had focused on defining properties of the random *automata* which we were building while in fact, what we were really interested in were properties of the *terms* which they accepted. Recall that the most important of those properties was the height of the smallest accepted term.

And so, instead of trying to generate automata based on criteria such as $|Q|$, δ, etc, we started by randomly generating "shapes", of "skeletons", and then

only did we generate automata whose recognised terms fit to those skeletons (*ie.* are isomorphic to the skeletons). Instead of having a model parametrised by quantities affecting the automata directly, we built an algorithm parametrised mainly by the quantities affecting the aspect (width, height,. . .) of the generated skeletons.

Let us explain in more detail what the skeletons are and how we generate them. Since we will present snippets of code from the prototype in this section, we need to begin by introducing the data type we use to represent trees – which is rather obvious:

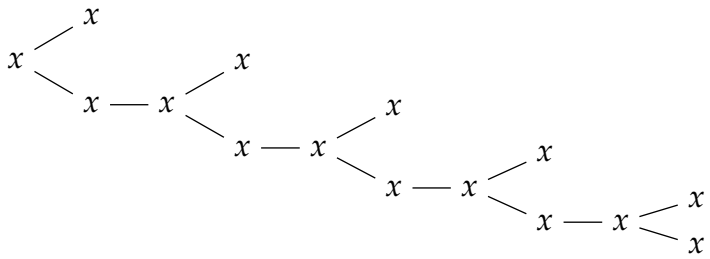
```
type α tree = Leaf of α | Node of α × α tree list

let rec 𝔥ei = λ (| Height of a tree |)
  | Leaf ⊥ → 1
  | Node (⊥,nl) → succ [[ ℒ.fold→ max (ℒ.map 𝔥ei nl) 0
```

As an example of using this data type, we have given the definition of the height of a tree, which is a direct adaptation of the definition given in [CDG+07]. Now, let us define what we meant by the "width" of a tree:

```
let rec width = λ (| Width of a tree |)
  | Leaf ⊥ → 1
  | Node (⊥,nl) → ℒ.fold→ (+) (ℒ.map width nl) 0
```

Note that unlike "height", this is not the only way we could have written this definition[(d)]. Consider for instance the following tree[(e)]



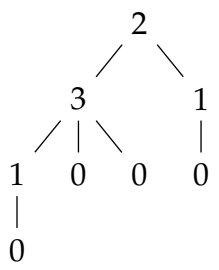
By our definition, it has a width of 6, but one could observe that at no point does any level of the tree have a width greater than 2, and this could be the basis for another definition of "width". In our experiments, we will stick with the definition which we have initially chosen though.

[(d)] Although in the case of height, one could legitimately choose a height of either 1 or 0 for leaves.

[(e)] This particular shape of trees is called a *peigne* in French, but the English equivalent escapes me for now.

With this in mind, a skeleton is a tree which we generate within certain constraints of height and width. In theory we do not need to decorate the nodes at all, but in practice we will use the nodes to store their number of children (or arity). So for instance the following tree t_s

$t_s =$



is a skeleton, of height and width equal to 4. In order to generate our skeletons, we use the simple, ad-hoc algorithm reproduced below. A few parameters and functions deserve some explanations:

- ◇ Parameter **depth**: the required height for the tree
- ◇ Function **random_breakdown** $x\ n$: function which returns a list $[x_1, \ldots, x_n]$ such that $\sum_{k=1}^{n} x_k = x$. The exact values of each x_k are chosen randomly within those parameters. Of course the function assumes the precondition $n \geqslant 0 \wedge x \leqslant n$.
- ◇ Parameter **girdle**: an upper bound on the width of the generated tree. During the generation, which is done recursively, this quantity is split at random between the children after the arity of the current node is chosen.
- ◇ Parameter **looseness**: [default=1]: determines how many branches must be protected, that is to say, cannot be cut before reaching the appointed height. Unprotected branches may be cut prematurely, if they have a girdle of 1 (that is to say they cannot branch off, and none of their children, if they have any, will branch off) and an arity 0 is selected for them.

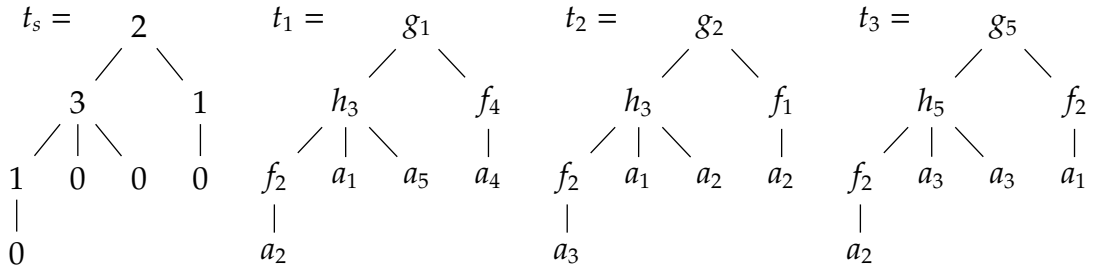
```
9  type skeleton = int tree
10 let build_skel looseness girdle depth =
11   let rec flag_protect n = λ
12     | a :: l → (a, n > 0) :: flag_protect (pred n) l
13     | ∅ → ∅
```

```
  in let rec build depth (girdle, protected) =
    assert (girdle ⩾ 1 ∧ depth ⩾ 0);
    match depth with 0 → Leaf 0 | ⊥ →
    let arity = min 3 (
      if girdle = 1 ∧ ¬protected
        then Random.int (girdle + 1)
        else 1 + Random.int girdle
    ) in
    (* split remaining girdle between children at random *)
    let girdles = flag_protect looseness (random_breakdown girdle arity) in
    if arity = 0 then Leaf 0 else
    Node (arity, ℒ.map (build(depth − 1)) girdles)
  in build depth (girdle,true)
```

Now that we can generate skeletons within our specifications, let us see how we can generate tree automata based on them. As for the previous generations, we first need to define the alphabet which we will be working with. Let us define

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

In practice, we have chosen to take five symbols for each arity, so we are working with Σ^5. With this in mind, the following algorithm generates a random set of rules, based on a skeleton. The rules which we generate, once integrated to a tree automaton, enable it to accept a number of terms isomorphic to the input skeleton. For instance, taking again our example skeleton t_s, we can generate rules to recognise the terms t_1, t_2, t_3, and potentially many others.



Again, a few explanations are warranted for some of the algorithm's parameters and functions[(f)]:

[(f)] And, in case any `OCaml` programmer wonders where the **return** keyword comes from, it is of course just syntactic sugar – a synonym to the identity function, in fact – to emphasise that we *are* returning the fresh state q_x.

- ◇ Parameter δ: this is a bit similar to the δ from generation 2, where it denoted the average in-degree of the states, in that it determines the (maximum) number of rules which we generate for each state. In practice we have chosen $\delta = 2$.
- ◇ Function **fresh_state**: generates a new state for use in the automaton, never used before – hence *fresh*.
- ◇ Function **gene_symbol** k : returns a symbol in Σ_k^5, uniformly at random.

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

Note that the algorithm yields a couple (Δ, q_h), where Δ is the set of rules we wanted, and q_h is the "head state", that is to say the state to which the accepted terms evaluate. It is intended to be used as a final state of the final automaton. So, to sum it up, we generate tree automata thusly:

1. First, we generate a number of random skeletons, within some constraints of height, maximum width etc. Let us denote $S = \{s_1, \ldots, s_n\}$ the set of our skeletons.
2. Second, we convert each skeleton into a set of transition rules using the above *conversion* function. We obtain a set of rule sets $R = \{\Delta_1, \ldots, \Delta_n\}$; we make it so that the states used each Δ_i do not appear in any Δ_j, for $i \neq j \in [\![1, n]\!]$.
3. We put all those rules together, *ie.* $\Delta = \bigcup R$, and extract the set Q of all states which appear in at least one of the rules of Δ. As for the final states, we take them to be the "head states" returned by the conversion function.

Note that, unlike previous generations, the generated automata are already completely reduced – it is indeed clear that all the states are reachable, and there are no useless states – as defined in theorem 53[p91], as all states actually serve to build a final state.

In practice we took the height h to be the sole parameter of our model, and generated three skeletons of depth taken randomly and uniformly in $[h/2, 3h/2]$, with girdles of $h/4$. Table 8.2[p70] gives statistics regarding the size of generated automata. As is clear from the table, the automata become quickly *very* size-

Height	$\|Q\|$	Size	Size/$\|Q\|$	Rules	Rules/$\|Q\|$
4	13.55	90.46	6.68	24.38	1.80
10	45.37	321.51	7.09	81.59	1.80
16	131.49	943.11	7.17	236.85	1.80
22	225.94	1620.27	7.17	406.52	1.80
28	389.99	2799.82	7.18	702.94	1.80
34	546.40	3928.71	7.19	982.75	1.80
40	835.23	6006.02	7.19	1504.01	1.80
46	1056.55	7606.87	7.20	1901.65	1.80
52	1403.94	10111.12	7.20	2526.44	1.80
58	1615.25	11633.75	7.20	2908.14	1.80
64	2145.28	15443.11	7.20	3861.08	1.80
70	2467.94	17767.11	7.20	4441.51	1.80
76	2968.86	21373.16	7.20	5344.01	1.80

Table 8.2 — Generation 3: size statistics

able, which turned out be be a major point against them. In our study of the emptiness problem, we have developed both exponential decision algorithms and polynomial methods (called "cleanup") for reducing the size of `TAGED`s prior to using the more expensive algorithms. However, with underlying tree automata as big as those, the cleanup algorithms became a bit of an overhead – the "difficult" algorithms were equipped with timeout facilities; not so for the "easy" cleanup. When generating a graph with a few dozens data-points, where each point must involve a few hundred generated automata in order to become statistically significant, this overhead quickly became a deal-breaker.

Another, more serious point against this method is that, while it completely solves the problem of accepting small terms which we suffered from when using the second generation, the generated automata have a very "rigid" structure, with each state appearing at most once in accepted terms. Furthermore, for

any state q, all the rules in $\mathfrak{Rul}(q)$ share the same signature. This last point was a major drawback, as it made those automata trivial instances for one of our emptiness algorithms. Thus this approach could not be considered to generate interesting enough automata for our purposes.

8.4 Fourth model: hybrid generation

The fourth (and last) approach that we tried was something of a hybrid of previous generations, notably 2 and 3. While we did not continue using skeletons, the idea of fixing the minimum height of recognised terms remained, as it was essential to avoid the pitfalls of generation 2. However, the aim was also to avoid having too rigid a structure, as opposed to generation 3.

So, the fourth generation is focused on two main parameters: the minimum height of terms, and the aspect of the set of rules. This time, we tried to achieve "difficult" cases by enabling the generation of all kinds of rules, for instance rules with immediate cycles ($f(\ldots, q, \ldots) \to q$), repetitions of the same state ($f(\ldots, p, \ldots, p, \ldots) \to q$), and also rules of the form $f(\ldots, p, \ldots) \to q$, where p is an "old" state, as opposed to a freshly generated state. None of these kinds of rules could be generated by the third generation: all generated rules were of the form $f(p_1, \ldots, p_n) \to q$, where the p_k were all distinct, and the terms in $\mathcal{L}\mathrm{ng}(\mathcal{A}, p_k)$ were all exactly of the same height h, and those in $\mathcal{L}\mathrm{ng}(\mathcal{A}, q)$ were of height $h+1$. As for the second generation, such rules *could* have been produced, but the odds against them actually coming into play were overwhelming, given the huge proportion of simple cases (leaf terms etc) which was characteristic of the second generation. Another requirement was that for each q, the signatures of the rules of $\mathfrak{Rul}(q)$ were sufficiently varied. As stated in the previous section, with the third generation, if you have a rule, say $f(p_1, \ldots, p_n) \to q \in \Delta$, then it follows that $\mathfrak{Rul}(q) \subseteq \{\, \sigma(p_1, \ldots, p_n) \to q \mid \sigma \in \Sigma_n \,\}$, so any rule $r \in \mathfrak{Rul}(q)$ is such that $\mathfrak{Int}(r) = \{\, p_1, \ldots, p_n \,\}$. This property was not representative of real world tree automata, and made the automata trivial instances of one of our algorithms for testing emptiness. We also wanted to avoid generating too many useless branches, which would be cut off by trivial observations; that is to say, we wanted to keep the ratio R, as defined for the second generation, to a low value. And lastly, we wished to keep the automata to a reasonable size, focusing on generating "difficult" instances, rather than "large" ones.

In short, the aim was to keep most of what we found interesting in the previous generations, without any of their shortcomings. Fortunately, the approach

which we will now discuss seems to manage that.

The fourth generation algorithm is not difficult in any way, but as it is a bit long (about two full pages), we will not give the full implemented code but instead draw and explain a very rough outline: see figure 8.4[p72]. Note that we are working with the same alphabet as the third generation, namely $\Sigma = \Sigma^5$. Of course, this outline leaves many things in the dark: for instance,

```
Data: minimum height, a number of other parameters
Result: a random tree automaton
begin
    pool ← some head states from small Gene3 terms;
    Δ ← rules from Gene3 for pool;
    while minimum height not reached do
        q ← fresh state;
        δ ← random number of rules;
        for δ times do
            n ← random arity;
            σ ← random symbol in Σ_n;
            purge too old states from pool;
            p_1, ..., p_n ← random states from pool;
            add rule σ(p_1, ..., p_n) → q to Δ;
        endfor
        add q to pool;
    endw
    F ← random states in pool;
    return tree automaton based on Δ and F;
end
```

Figure 8.4 — Rough outline of Generation 4 algorithm

each time something is selected at "random", one can wonder about the exact implementation of these random selections:

- ◇ Random number of rules and arity are selected with a certain discrete probability distribution, which is hard-coded in the algorithm using the *w_choice* function described below.
- ◇ Random symbols in Σ_n are selected uniformly.
- ◇ Random states from the pool are selected according to a discrete probability distribution which is itself a function of the minimum height of terms

which evaluate to q. The distribution is biased to favour states which recognise bigger terms. This is an important idea of the algorithm, which allows the generated automata to keep an acceptable size while allowing

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

Table 8.3 — Generation 4: size statistics

enough variety in their rules.

- ◇ Old states to be purged from the pool are states whose associated minimum height has become too low compared to the greatest associated height which has been generated. In other words, if you are currently building the top of the tree, you avoid reusing states from the bottom of the tree. The degree of tolerance for old states is called *cohesion*, and is a parameter of the procedure. Tighter cohesion means smaller, more focused automata.

Let us say a few more word about how the "pool" works. What we call pool is the set of states for which we have generated rules so far. Initially, we create a few sets of rules accepting small terms using the third generation. We store those rules in Δ, and put the head states[(g)] of those rule sets into the pool. For each q in the pool, we also keep track of the minimum height of terms which evaluate to q with respect to Δ. Let us denote it by $m(q)$. Since Gene3 is directed by the height, this quantity is known for the generated head states. Let us denote the pool by $P = \{p_1, \ldots, p_n\}$, and let $\mathbb{b} : \mathbb{N} \to \mathbb{N}$ be a function which we will call

[(g)] See previous section

"bias". Then, when we want to get states out of the pool, we select a random state X following the probability mass function:

$$\forall i \in [\![1, n]\!], \quad \mathbb{P}(X = p_i) = \frac{\mathbb{b} \circ m(p_i)}{\sum_{k=1}^{n} \mathbb{b} \circ m(p_k)}.$$

The implementation of such a choice is straightforward:

```
let w_choice wlist =
  let ⊥,weights = ℒ.split wlist in
  let totalW = ℒ.fold→ (+) weights 0 in
  let rec f dart = λ
    | (item,w) :: tl → if dart ⩽ w then item else f (dart − w) tl
  in λ()→f (1 + Random.int totalW) wlist
```

The call of **w_choice** $[(x_1, w_1), \ldots, (x_n, w_n)]$ returns a function which, when called, returns x_i with probability $w_i / \sum_{k=1}^{n} w_k$. The remaining point to discuss is the choice of the bias $\mathbb{b}$; it should be strong enough to favour states with a greater minimum height, but not so much so as to completely forbid the use of older states. We have chosen it to be

$$\mathbb{b}(w) = (w - h + d + c)^2,$$

where

- $h = \max_{p \in P} m(p)$ – in other words, it is the greatest minimum height associated with the states we have generated so far.
- c is the cohesion value, mentioned higher up. The cohesion requires the property $\forall p \in P : \ h - c \leqslant m(p) \leqslant h$ to be an invariant. Its default value in our experiments was 2.
- d is the "damping". It follows from the cohesion invariant that $\forall p \in P : \ d \leqslant m(p) - h + d + c \leqslant c + d$. The value d is chosen to be (an approximation of) the solution to the equation $(d + 2)^2 = 2d^2$. So by taking $d \approx 2(1 + \sqrt{2}) \approx 5$, we make it so that states two ranks higher than another have a twice greater chance of being selected than that other state.

Note that actually, for the choice of final states, the bias is stronger than for an ordinary choice: $\mathbb{b}(w) = (w - h + d + c)^4$.

To conclude this description of the *modus operandi* of the fourth generation, let us give the distributions used in our tests for arities and number of rules:

```
7    let new_arity = w_choice [1,2 ; 2,3 ; 3,1] in
8    let new_delta = w_choice [1,70; 2,25; 3,2; 4,1; 5,1; 6,1] in
```

As it turned out, this method generates sufficiently interesting random tree automata for our purposes – we will discuss that in the next part, and satisfies the wishes expressed at the beginning of this section. See for instance table 8.3[p73] which shows that the size of generated automata is quite reasonable – especially compared to table 8.2[p70]; and figure 12.4[p104] which shows a very satisfactory ratio $R = 1.15$, regardless of height. Note that this ratio is due solely to useless states (53[p91]), since the generated automata are reduced by design.

On a side-note, this generation proved to be useful for generating good "human-readable" examples of TAGEDs satisfying some properties: we implanted a simple procedure which, given a predicate p on TAGEDs and – optionally – a comparison function $\prec$ between TAGEDs, yields the best ($\prec$-wise) random TAGED satisfying p which it could find in reasonable time. Fourth generation random TAGEDs seem to be varied enough that this approach works quite well for pretty much anything. For instance the example automata of section 12.4[p105] were generated using this method.

A significant anecdote in that respect is that fourth generation automata were the only ones to trigger an assertion, buried deep into the code, which highlighted a lurking bug in the implementation of the reduction algorithm[(h)]. Before that time many experiments had been done with the previous generations, and that part of the code had been triggered at least several hundred thousand times, without ever covering this – rather involved – edge case.

[(h)] The previous generations were useful in finding *other* bugs though, but no more than Gene4 would have been.

Chapter 9

Generating random Constraints

In this chapter we discuss different methods for generating equality constraints and bridging the gap between the random tree automata which we generate through the four random generation approaches outlined in the previous chapter, and random positive TAGEDs.

9.1 First model: dense generation

Jointly to our first random generation model, we shall take p_Δ and p_F fixed to appropriate values $p_\Delta = \frac{11}{100}$ and $p_F = \frac{1}{5}$, which will ensure that a *small* proportion of generated underlying Tree Automata are empty for most values of $|Q|$. As for $|Q|$, it will be a parameter of our model, as it is a value which bounds the running time of the algorithms, thanks to the pumping lemma for positive TAGEDs [FTT08b, Lemma 2(Appendix)]. Another parameter which seems self-evident is the density of equality constraints, which we would think of as a value $d_= \stackrel{\text{def}}{=} |=_\mathcal{A}|/|Q|^2$. In practice, we use a probability $\mathbb{p}$. For each couple $(p, q) \in Q^2$ such that $p \leqslant q$ for an arbitrary ordering of Q, we choose $p =_\mathcal{A} q$ with probability $\mathbb{p}$, and compute the symmetric closure of this after all choices have been made. Note that because of the additional requirement that the relation be symmetric, $\mathbb{p}$ and the hypothetic density $d_=$ appear to be fairly different metrics. However, the probability that $p =_\mathcal{A} q$, for $p, q \in Q$ chosen randomly (and uniformly), is still $\mathbb{p}$, and we shall make use of that fact.

With this model it is fairly easy to evaluate explicitly the impact of trivial operations, for instance simplification by removal of spurious rules; see definition

57[p93]. We have, as an immediate consequence of the definition of $\overline{\Delta}$,

$$\left|\overline{\Delta}\right| = \sum_{k\in\mathbb{N}} |\Sigma_k| \cdot |Q|^{k+1} \text{ and } |\Delta| = p_\Delta \left|\overline{\Delta}\right| = p_\Delta \sum_{k\in\mathbb{N}} |\Sigma_k| \cdot |Q|^{k+1},$$

and bearing in mind that a rule $f/_n(p_1, \ldots, p_n) \to q$ will *not* be spurious if and only if $(p_i, q) \notin =_{\mathcal{A}}$, $\forall i \in [\![1, n]\!]$, and that those events are independent, and each happens with the same probability $(1-\mathbb{p})$, it follows that the rule is non-spurious with probability $(1-\mathbb{p})^n$. Finally, if we denote Δ_g the set of "good" (non-spurious) rules, then we have on average

$$\left|\Delta_g\right| = p_\Delta \sum_{k\in\mathbb{N}} (1-\mathbb{p})^k \cdot |\Sigma_k| \cdot |Q|^{k+1}.$$

Then if we are to randomly generate equality constraints for our first model of random tree automata, we will obtain on average the following ratio between good rules and all rules:

$$\frac{\left|\Delta_g\right|}{|\Delta|} = \frac{1 + (1-\mathbb{p})^2 |Q|^2}{1 + |Q|^2}.$$

This converges very quickly towards $(1-\mathbb{p})^2$ as $|Q|$ grows. Notice that this depends on neither p_Δ (because it was simplified away), p_F (because final states are irrelevant to a rule being spurious or not), nor $|Q|$ (because the convergence is very quick, it makes virtually no difference beyond, say, $|Q| = 10$). Figure 9.1[p78] shows the good rules ratio as a function of $\mathbb{p}$. Four curves are shown – although the naked eye may have difficulty in telling them apart: the theoretical limit ratio $(1-\mathbb{p})^2$, the theoretical ratio at $|Q| = 15$ – which is almost indistinguishable from the limit – and the experimental ratios, obtained by averaging the actual ratios for 200 generated `TAGED`s, for two different PRNGs. The first curve was obtained with the default LFSR, and the second with a Mersenne twister. As expected, both curves follow the theoretical ratio very closely [(a)].

Note that the ratio was determined by our choice of Σ in this model; with a different alphabet the limit theoretical ratio would be $(1-\mathbb{p})^N$, where $N = \max_{f\in\Sigma} arity(f)$. This is a consequence of the remark made about the high density model, that high arity rules completely overwhelm rules of lesser arity.

[(a)] The experiment was unneeded given that the ratio was easy to compute exactly. It was done nevertheless to ascertain that the implementation was not blatantly incorrect, and test the PRNGs.

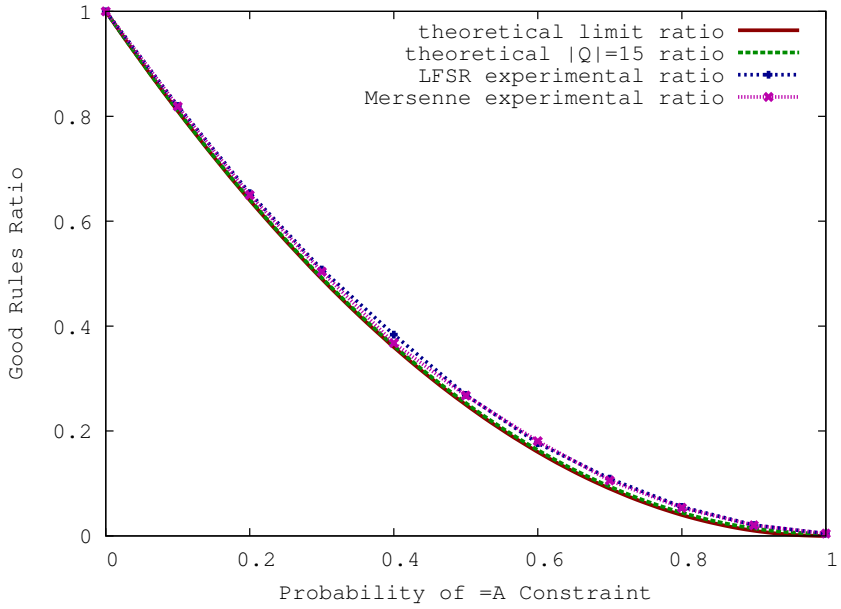


Figure 9.1 — Theoretical $|\Delta_g|/|\Delta|$ ratio and experimental data at $|Q| = 15$

9.2 Second model: sparse generation

In the first model, we had taken constraints $p =_{\mathcal{A}} q$ uniformly in Q^2 – notwithstanding symmetry. But in our experience, it seems that constraints of the form $q =_{\mathcal{A}} q$ are in fact slightly more common than the general form. It seems therefore pertinent to introduce a bias in favour of those constraints. Furthermore, we have the same problem with constraints as we had with transition rules: too high density. In practice, `TAGED`s seem to require a rather small number of constraints; thus going through the full range of $|=_{\mathcal{A}}| = 0$ to $|Q|^2$ seems like overkill. With those two remarks in mind, we devise the following random model:

Let $\mathbb{b} \in \mathbb{R}^+$ be the *bias* in favour of diagonal constraints – that is to say, constraints of the form $q =_{\mathcal{A}} q$ – and $\mathbb{p}_0 \in [0, 1/1+\mathbb{b}]$ the base probability of choosing a constraint. Then we define

$$\forall p, q \in Q : p \leqslant q, \quad \mathbb{p}(p,q) = \begin{cases} \mathbb{p}_0 & \text{if } p < q \\ (1+\mathbb{b}) \cdot \mathbb{p}_0 & \text{if } p = q \end{cases}$$

and proceed from there in the same way as before: for $(p, q) \in Q^2$ such that $p \leqslant q$, we choose $p =_{\mathcal{A}} q$ with probability $\mathbb{p}(p, q)$, and then compute the symmetric

closure. Overall we can expect on average a total number of rules of

$$\begin{aligned}|=_{\mathcal{A}}| &= \mathbb{p}_0 \cdot (|Q|^2 - |Q|) + (1 + \mathbb{b})\mathbb{p}_0 \cdot |Q| = \mathbb{p}_0 |Q| \cdot (|Q| - 1 + 1 + \mathbb{b}) \\ &= \mathbb{p}_0 |Q| \cdot (|Q| + \mathbb{b}) ,\end{aligned}$$

and the ratio of diagonal constraints among chosen constraints, which we will denote D_r for short, becomes

$$D_r = \frac{\left|\{ (q, q) \in =_{\mathcal{A}} \}\right|}{|=_{\mathcal{A}}|} = \frac{(1 + \mathbb{b})\mathbb{p}_0 \cdot |Q|}{\mathbb{p}_0 |Q| \cdot (|Q| + \mathbb{b})} = \frac{1 + \mathbb{b}}{|Q| + \mathbb{b}}$$

Note that in the case where $\mathbb{b} = 0$, that is to say there is no diagonal bias, we end up with exactly the same model as before. Of course, we didn't go to the trouble of introducing $\mathbb{b}$ not to use it; the question is which value would be appropriate for it? Suppose that we take it to be a constant; for instance $\mathbb{b} = 1$. This means that a diagonal constraint has twice as great a chance of being selected as a non-diagonal one. Is this really as great an advantage as it sounds? We have

$$\lim_{|Q| \to +\infty} D_r = \lim_{|Q| \to +\infty} \frac{1 + \mathbb{b}}{|Q| + \mathbb{b}} = 0.$$

So, regardless of how huge a bias we decide on, so long as it remains a constant, diagonal constraints are overwhelmed by non-diagonal ones, simply because there are only $|Q|$ diagonal constraints available, but $|Q|^2$ constraints in all. What we would like to do is to have a fixed ratio of diagonal rules, no matter the size of the automaton; for instance, we would like for *half* of our constraints to be diagonal. What bias do we need to get that? Let us denote $\mathbb{b}_k$ the bias such that $D_r = 1/k$, for some $k > 0$. It is solution to the equation

$$\frac{1 + \mathbb{b}_k}{|Q| + \mathbb{b}_k} = \frac{1}{k} \qquad \text{and so} \qquad \mathbb{b}_k = \frac{|Q| - k}{k - 1} \text{ and } \mathbb{p}_0 \in \left[0, \frac{k - 1}{|Q| - 1}\right].$$

Note that, if we choose $\mathbb{b} = \mathbb{b}_k$, then $|=_{\mathcal{A}}| \leqslant k\,|Q|$, the equality holding whenever $\mathbb{p}_0$ is maximal. In other words, the expected number of constraints evolves linearly with the size – which we take $|Q|$ to be the indicator of – of the generated automata. This is consistent with the spirit of the second, low-density model for tree automata, where the number of rules were also proportional to the size of the states space, and consistent with our practical observations as well. For our tests, we take

$$\mathbb{b} = \mathbb{b}_2 = |Q| - 2 \quad \text{ and } \quad \mathbb{p}_0 \in \left[0, \frac{1}{|Q| - 1}\right].$$

Now that this is settled, we must concern ourselves with trivial cases, just as we have done for the part of the model concerned with vanilla tree automata. The trivial cases were then empty automata (*ie.* accepting the empty language). Here, the trivial cases are diagonal positive `TAGED`s, which, as we have seen, can be dealt with quite easily. Figure 9.2[p80] shows the probability of generating a diagonal `TAGED`, for different values of $|Q|$. Formally, this probability – which we will denote $\mathbb{P}_d$ – can be computed as follows:

$$\mathbb{P}_d = (1 - \mathbb{p}_0)^{\frac{1}{2}|Q|\cdot(|Q|-1)}.$$

Note that, although the expression does not appear to depend on the bias, it depends on $\mathbb{p}_0$ whose range is dependent on the value of $\mathbb{b}$. Looking at the

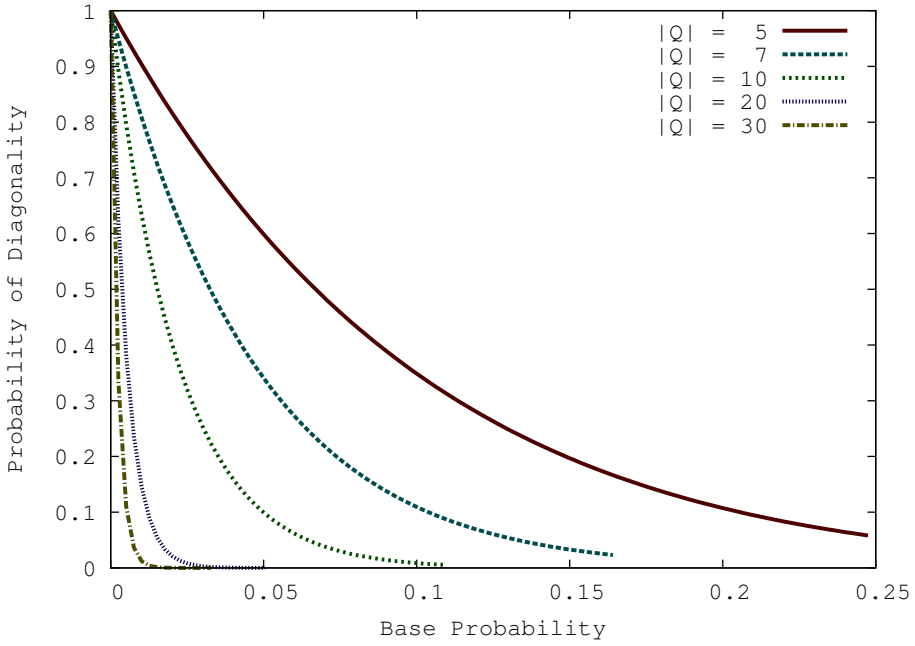


Figure 9.2 — Probability $\mathbb{P}_d$ as a function of $\mathbb{p}_0$

figure, one sees that simple configurations, that is, low values of $|Q|$ and $\mathbb{p}_0$, are very likely to generate diagonal automata, while less simple ones are very unlikely to do so. This is coherent with intuition and observations. When one writes a small, simple positive `TAGED`, it is very likely to be diagonal; in fact most of those the author had to deal with in his work so far *were* diagonal. Figure 9.2 also shows a peculiarity of the model: the changing range of the probability $\mathbb{p}_0$. Since this value depends on $\mathbb{b}$ it also depends on $|Q|$, which tends make the model a bit cumbersome to work with in practice. One would prefer to have

a parameter taking a fixed range of values. So instead of using $\mathbb{p}_0$ directly we shall use the *corrected base probability* $\mathbb{p}_\mathbb{b}$:

$$\mathbb{p}_\mathbb{b} = (1 + \mathbb{b}) \cdot \mathbb{p}_0 \quad \text{or equivalently} \quad \mathbb{p}_0 = \frac{\mathbb{p}_\mathbb{b}}{1 + \mathbb{b}}.$$

The advantage of using $\mathbb{p}_\mathbb{b}$ is that it always lives in the range $[0, 1]$. Compare

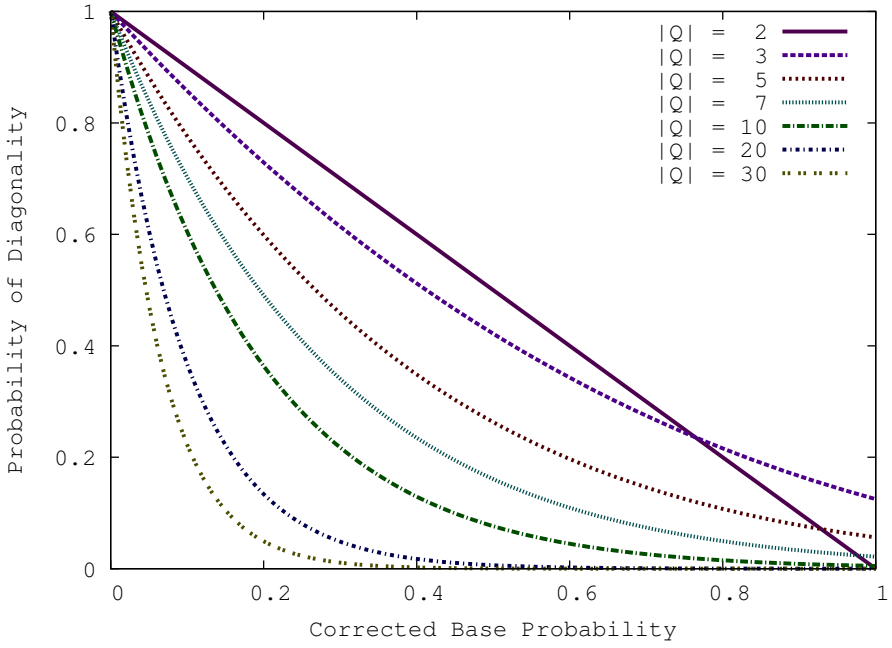


Figure 9.3 — Probability $\mathbb{P}_d$ as a function of $\mathbb{p}_\mathbb{b}$

figure 9.2 to figure 9.3[p81], which shows the same curves, but this time as functions of $\mathbb{p}_\mathbb{b}$.

In the end, there are two simple parameters to this model: $\mathbb{b}$, which we have conveniently fixed to the appropriate value $\mathbb{b}_2$ and will not need to touch again, and $\mathbb{p}_\mathbb{b} \in [0, 1]$.

Experimental results of this model with respect to the cleanup operation are presented in section 12.3[p102].

9.3 Third model: logarithmic generation

In practice, jointly with the fourth generation of random tree automata, we gave up using the second model of constraints generation because we observed that

the resulting TAGEDs did tend to become more and more frequently empty as their size grew, which we took to mean that there were too many constraints. Furthermore, it is legitimate to suppose that real-world TAGEDs would not necessarily see the numbers of constraints grow linearly in their size. For instance, if a TAGEDs is the result of transformation effected under rewriting rules, the number of constraints could remain constant, though the underlying tree automaton become increasingly large. As a compromise, we fell back on a number of constraints logarithmic in $|Q|$. We believe this choice does not affect the complexity class in which emptiness falls – *ie.* even with a logarithmic number of constraints, the problem remains *EXPTIME*-complete. However some work would be needed in order to confirm this intuition. The algorithm to generate

Data: set of states Q
Result: a set of random constraints
begin
  Csts $\leftarrow \varnothing$;
  for $\max\left(1, \log_{10}|Q|\right)$ *times* **do**
    $q, p, p' \leftarrow$ random states in Q (uniform);
    add $\{(q,q),(p,p'),(p',p)\}$ to Csts;
  endfor
  return Csts;
end

Figure 9.4 — Very simple constraints generation algorithm

the constraints is exceedingly simple, and given in the figure 9.4[p82]. Note that we keep the same bias towards diagonal constraints which we had in the second model: roughly half the constraints are diagonal.

In practice, this simple model proved to be sufficient, and used in conjunction with the fourth generation of random tree automata, kept some balance between emptiness and non-emptiness of the resulting TAGEDs, which is of course what we aimed at.

Chapter 10

Conclusion

In this part we have explored ways of generating random TAGEDs, and in so doing conceived, implemented and tested four methods for generating random tree automata, and three for generating equality constraints.

From our experiments it appears that, while there are many possible ways of generating random tree automata, making them actually *interesting* is not quite as straightforward as one might have expected. Our initial attempts to adapt established experimental methods from the world of word automata [TV05], presented in sections 8.1[p57] and 8.2[p61] have turned out to generate absolutely trivial tree automata, and therefore trivial TAGEDs, from the point of view of the emptiness problem at the very least.

The aim of this study was in no way to be either exhaustive or profound, but simply to create, without wasting too much time, an experimental protocol fit for the evaluation of our own specific algorithms on our own specific problem, and we did not push this survey any further than we strictly needed to. Nevertheless, it served to highlight the lack of coverage of this area in the literature, and the dangers of being both the designer of an algorithm and that of the home-made experimental system intended to test it. For instance, the experimental results of the brutal algorithm (see 15[p131]) with the second generation *were* excellent, and only the fact that this clashed with our expectations for it lead us to question the quality of our experimental protocol. In the absence of efficient reference algorithms, test beds and implementations – which is the case for TAGEDs and other new flavours of tree automata – and without any *a priori* notions concerning the practical efficiency of a new algorithm or heuristic, it is only too easy to confuse bad protocols and good results.

In the light of this, we think that it would be both interesting and highly useful

to continue this study of random generation of tree automata and extended tree automata in a more systematic way. Retrospectively, the time allocated to the internship could have been better spent if this part had been done earlier – instead of in the second half, as it happened. The ability to test ones algorithms proved extremely helpful both as a source of inspiration for their development, and as a means of ensuring correctness of the implementation. Several lingering bugs in the implementation, some of them rather severe, were indeed revealed by randomly generated automata.

It would also certainly be educational to revisit the experiments regarding the membership problem done during the internship (see for instance section 4.2[p34]) using random `TAGED`s.

Part IV

TAGEDs and the Emptiness Problem

Chapter 11

Introduction

The main theme of the internship was to tackle the emptiness decision problem for positive TAGEDs.

Definition 46 (Emptiness decision problem)**.**
Input: $\mathcal{A}$, an automaton (in this case a positive TAGED)
Output: $\mathcal{L}\text{ng}(\mathcal{A}) = \varnothing$?

This problem arises in many circumstances, for instance in model checking the question of whether a "bad" state is reachable translates to emptiness of the automaton accepting the intersection of the language of bad states and that of reachable states. In the context of queries over tree languages (such as TQL), for which TAGEDs have been initially created, decidability of query languages reduces to emptiness of (sub-classes of) TAGEDs [Fil08].

Decidability of the emptiness problem for TAGEDs in general remained an open problem for a while, but it recently turned out to be decidable, although the precise complexity of the problem is still unknown. These findings are still drafts though, and do not appear to have been officially published yet. The draft of the article [GJV] is available online [(a)].

More is known about the two main subclasses of TAGEDs: positive and negative TAGEDs. As shown in table 2.2[p19], the emptiness problem of in *NEXPTIME* for negative TAGEDs, and *EXPTIME*-complete for positive TAGEDs – and this is the best known lower bound for the complexity of emptiness of TAGEDs. [Fil08, FTT08b, GJV].

[(a)] On http://www.lsi.upc.es/~ggodoy/papers/globalconstraints.pdf.

As mentioned before, in this work we will exclusively be interested in positive TAGEDs, and our objective will be to find algorithms for solving it in "reasonable" time, within the boundaries of the possible, of course.

Even though the problem is *EXPTIME*-complete, there are cases where simple observations can be enough to conclude, and if they are not, might still give ways in which the automaton can be trimmed down without changing its recognised language. If those operations remain fairly inexpensive compared to a more general algorithm, using them before running more expensive tests would appear to be a good strategy.

In the next chapter, we will introduce several such inexpensive tactics to reduce the problem as much as possible (chapters 12[p88] and 13[p107]), detect and deal with easy cases (chapter 14[p118]), and one – expensive – tactic (chapter 15[p131]) to deal with the remainder. All those tactics are linked together through a global strategy which is presented in chapter 16[p139].

Note that in every definition and theorem which follows, we will assume – unless otherwise specified – that we are dealing with a reduced[(b)] TAGED $\mathcal{A} = (\Sigma, Q, F, \Delta, =_{\mathcal{A}}, \neq_{\mathcal{A}})$ (*cf.* definition 2[p17]), and we may use those notations without reminders.

[(b)] By the standard reduction algorithm for tree automata. *cf.* [CDG+07] and algorithm 12.1[p91].

Chapter 12

Cleanup: hunting for spuriousness

In this chapter we will show that the presence of a global equality constraint may render a number of transition rules and states visibly inoperative. Those rules and states will be called *spurious* if it is clear that their use would fatally be in contradiction with the equality constraint, and therefore that they can be removed without altering the language recognised by the TAGED.

This chapter formalises and justifies this notion of *spurious constructions* and presents algorithms to sanitise TAGEDs, that is to say, to remove all spurious constructions from the automaton. This operation can be seen as an extension of the reduction algorithm for vanilla tree automata to positive TAGEDs. The aim is not yet to decide emptiness – though this might happen in trivial cases where all final states are spurious, for instance – but to lighten the load of the general algorithms which will. To this end we will also introduce a simplification which applies even on vanilla tree automata.

Of course, the computational cost of this sanitising operation remains negligible compared to that of the general algorithms presented in the next sections.

The last section gives some experimental results obtained on random TAGEDs.

12.1 A theory of spuriousness

Let us begin with an observation which applies to vanilla tree automata as well as TAGEDs; the classical reduction algorithm given in [CDG+07] removes from the automaton all those states which are not reachable and thus contribute nothing. However this does not mean that every reachable state *does* contribute something; if a state has no possible use whatsoever in building an accepted

term, that is to say, if that state is neither final nor usable in a run which leads to a final state, then it is useless and can safely be removed, even if it is reachable. Of course, by "safely" we mean that the language accepted by the automaton is not affected in any way by this operation. This is the object of theorem 53[p91]; first let us introduce two simple definitions for syntactic convenience:

Definition 47 (Associated rules)**.** Let $\mathcal{A}$ be a `TAGED` and $q \in Q$. The the *associated rules* of q are defined as $\mathfrak{Rul}(q) \stackrel{\text{def}}{=} \{ r \in \Delta \mid r = f(\dots) \to q \}$.

Definition 48 (Antecedents)**.** Let $\mathcal{A}$ be a `TAGED` and $r = f(q_1, \dots, q_n) \to q \in \Delta$. We call *antecedents of* r, and denote $\mathfrak{Ant}(r)$, the set $\{ q_1, \dots, q_n \}$.

With these definitions, let us express the idea that for each state, say, q, there are a limited number of rules which could have produced it – the rules in $\mathfrak{Rul}(q)$ – and therefore in a run, q's children can only be chosen among those states which are antecedents to one one those rules. This is what we can *potential requirements of* q: the set of states which *can*, potentially, appear as direct children of q in a well-formed run.

Definition 49 (Potential requirements)**.** Let $\mathcal{A}$ be a `TAGED`, and let $q \in Q$. The *potential requirements* of state q are defined as

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

Now, we can generalise this notion: if only a few states, say, p_k, can be q's children, then only the states which can be p_k's children for some k – *ie.* are a potential requirement of p_k – can be q's grand-children... And thus we define the set of all states which can appear under q, either as direct children, grand-children, etc. We call these states "friends of q".

Definition 50 (Friend states)**.** Let $\mathcal{A}$ be a `TAGED`, and $q \in Q$. We define $\mathfrak{Frnd}(q)$ as the smallest subset of Q satisfying the two following properties:

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

The next lemma formalises and justifies what we have been saying informally: it states and proves that if a certain state q appears in a well-formed run, then we know that all the states which appear under q are its friends.

Lemma 51 ("Rely on your Friends" principle)**.** *Let $\mathcal{A}$ be a* TAGED*, $t \in \mathcal{T}(\Sigma)$ a term, and ρ a run of the underlying tree automaton $\mathfrak{ta}(\mathcal{A})$ on t. Then the following holds :* $\forall \alpha, \beta \in \mathcal{P}os(t) : \beta \lhd \alpha \implies \rho(\beta) \in \mathfrak{Frnd}(\rho(\alpha))$.

Proof. We will prove the equivalent statement $\forall \alpha \in \mathcal{P}os(t), \forall \beta \in \mathcal{P}os(t) : \exists n \geqslant 1 : \beta \lhd_n \alpha,\ \rho(\beta) \in \mathfrak{Frnd}(\rho(\alpha))$ by induction on n. Let $\alpha \in \mathcal{P}os(t)$, fixed but arbitrary.

1. (base case) let $\beta \lhd_1 \alpha$; then β is a direct child of α. It follows immediately from definition 49[p89] that we have $p \in \mathfrak{pReq}(q) \iff \exists f(\dots p \dots) \to q \in \Delta$, and thus if we were to assume that $\rho(\beta) \notin \mathfrak{pReq}(\rho(\alpha))$, it would follow that there is no rule $f(\dots \rho(\beta) \dots) \to \rho(\alpha) \in \Delta$, which would imply that ρ is not compatible with the transition rules and is therefore not a run. Since ρ is in fact a run this is absurd, and $\rho(\beta) \in \mathfrak{pReq}(\rho(\alpha)) \subseteq \mathfrak{Frnd}(\rho(\alpha))$.

2. (inductive case) let us assume that, for some n, $\forall \beta \lhd_n \alpha : \rho(\beta) \in \mathfrak{Frnd}(\rho(\alpha))$. Let $\gamma \in \mathcal{P}os(t)$ such that $\gamma \lhd_{n+1} \alpha$; then, $\mathcal{P}os(t)$ being prefix-closed, there must exists some $\beta \in \mathcal{P}os(t)$ such that $\gamma \lhd_1 \beta \lhd_n \alpha$. By the same reasoning as in the base case we have $\rho(\gamma) \in \mathfrak{pReq}(\rho(\beta))$, it follows by the definition of the friends states and our induction hypothesis that $\rho(\gamma) \in \mathfrak{Frnd}(\rho(\alpha))$.

Thus we have proved the result by induction. □

Before moving on to the announced theorem, we need to formalise what we meant by "removing a state from an automaton", which we call *restriction*. Since it is an operation which we will use quite frequently it deserves its own notation. Note that this is a straightforward adaptation of the notion used implicitly in [CDG+07], for instance when describing the reduction algorithm (*cf.* figure 12.1[p91] for the algorithm and figure 12.2[p100] for our implementation of it). We also introduce the *projection*, which consists simply in changing the set of final states.

Definition 52 (Restriction by states, projection)**.** Let $\mathcal{A} = (\Sigma, Q, F, \Delta, =_{\mathcal{A}}, \neq_{\mathcal{A}})$ be a TAGED, and let $S \subseteq Q$ be a set of states. We call *restriction of $\mathcal{A}$ to S* and denote $\mathfrak{Rst}(\mathcal{A}, S)$ the TAGED $(\Sigma, S, F \cap S, \Delta', =_{\mathcal{A}} \cap S^2, \neq_{\mathcal{A}} \cap S^2)$ where

$$\Delta' \stackrel{\text{def}}{=} \{ f(q_1, \dots, q_n) \to q \in \Delta \mid \{q, q_1, \dots, q_n\} \subseteq S \}.$$

We also call *projection of $\mathcal{A}$ on S* the TAGED $\mathfrak{Prj}(\mathcal{A}, S) \stackrel{\text{def}}{=} (\Sigma, Q, S, \Delta, =_{\mathcal{A}}, \neq_{\mathcal{A}})$.

With these new tools in hand, we can at last justify what we said at the beginning of this section: only those states which can possibly be used to build a final state – *ie.* which are friends of a final state – are of any real use for the automaton. The others can be removed without altering its recognised language.

Data: A TAGED $\mathcal{A}$
Result: A TAGED $\mathcal{A}'$ such that $\mathcal{L}\mathrm{ng}(\mathcal{A}) = \mathcal{L}\mathrm{ng}(A')$
begin
  Reach $\leftarrow \varnothing$;
  repeat
    add q to Reach where $r \in \mathfrak{Rul}(q), \mathfrak{Ant}(r) \subseteq$ Reach;
  until *no state can be added to* Reach ;
  return $\mathfrak{Rst}(\mathcal{A}, \text{Reach})$;
end

Figure 12.1 — Reduction algorithm, from [CDG+07, page 25]

Theorem 53 (Removal of useless states)**.** *Let $\mathcal{A} = (\Sigma, Q, F, \Delta)$ be a tree automaton. Then*

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

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

Proof. Let us show that some run ρ is an accepting run of $\mathcal{A}$ is and only if it is an accepting run of $\mathcal{A}'$. Since $\mathcal{A}'$ is a restriction of $\mathcal{A}$, it is clear that *any* run of $\mathcal{A}'$ is also a run of $\mathcal{A}$. It remains to show that if ρ is an *accepting* run of $\mathcal{A}$, it is also an accepting run of $\mathcal{A}'$. Suppose that this is not the case, that is to say, there exists a term $t \in \mathcal{T}(\Sigma)$ such that $\mathcal{A}$ accepts t through the run ρ, but ρ is not an accepting run of $\mathcal{A}'$. This could happen if ρ was a run for $\mathcal{A}'$, but not an accepting one; that is to say $\rho(\varepsilon) \in F_{\mathcal{A}}$ but $\rho(\varepsilon) \notin F_{\mathcal{A}'}$. However by definition of the restriction we have $F_{\mathcal{A}} = F_{\mathcal{A}'} = F$. Thus if ρ was a run for $\mathcal{A}'$, it would have to be accepting. Therefore ρ is not a run for $\mathcal{A}'$. It follows that ρ makes use of one of the rules which were removed, and by definition each removed rule makes use of a state which is neither final nor in $\bigcup_{q_f \in F} \mathfrak{Frnd}(q_f)$. So we can conclude that there exists $\alpha \in \mathcal{P}os(t), p \notin F \cup \bigcup_{q_f \in F} \mathfrak{Frnd}(q_f)$ such that $\rho(\alpha) = p$. On the other hand, we know by definition of an accepting run that $\rho(\varepsilon) \in F$, and either $\alpha = \varepsilon$, which is contradictory since $\rho(\alpha) = p \notin F$, or $\alpha \lhd \varepsilon$. But in that case lemma 51[p90] applies and $p = \rho(\alpha) \in \mathfrak{Frnd}(\rho(\varepsilon))$. that is also in contradiction with $p \notin \bigcup_{q_f \in F} \mathfrak{Frnd}(q_f)$. In all cases, we are faced with contradictions, and so our assumption is disproved, and ρ *is* an accepting run for $\mathcal{A}'$. □

The result applies only to vanilla tree automata so far. Fortunately the added complication of global equality (or even disequality) constraints does not invalidate the result.

Corollary 54 (Removal of useless states)**.** *The same result as theorem 53*$_{[p91]}$ *holds for* TAGEDs*.*

Proof. By theorem 53 the accepting runs of $\mathrm{ta}(\mathcal{A})$ are those of $\mathrm{ta}(\mathcal{A}')$ and *vice versa*. Since $\mathcal{A}'$ is a restriction of $\mathcal{A}$, its constraints are weaker and therefore the accepting runs of the former are a superset of those of the latter. There remains to show that every accepting run of $\mathcal{A}'$ is also accepting for $\mathcal{A}$. Let ρ be a successful run of $\mathcal{A}'$ on a term $t \in \mathcal{T}(\Sigma)$. So for all $\alpha \in \mathcal{P}os(t)$, we have $\rho(\alpha) \in F \cup \bigcup_{q_f \in F} \mathfrak{Frnd}(q_f)$. Let us suppose that ρ is *not* a successful run for $\mathcal{A}$. Then since it is an accepting run of $\mathrm{ta}(\mathcal{A})$, it must be incompatible with a global constraint. So there exist $p, q \in Q$ such that, say, $p =_{\mathcal{A}} q$, and two positions $\alpha, \beta \in \mathcal{P}os(t)$ such that $\rho(\alpha) = p$ and $\rho(\beta) = q$ and $t|_\alpha \neq t|_\beta$. But ρ is compatible with the constraints of $\mathcal{A}'$, therefore at least p or q must be specific to $\mathcal{A}$, that is, be in $Q \setminus F \cup \bigcup_{q_f \in F} \mathfrak{Frnd}(q_f)$. This is a contradiction. Thus ρ is also an accepting run for $\mathcal{A}$. □

Now let us examine TAGEDs, or more specifically, positive TAGEDs in more detail, and study some immediate consequences of the introduction of constraints to the influence of some rules and states. It seldom hurts to state the obvious, so let us do so in this next lemma:

Lemma 55. *Let $\mathcal{A}$ be a* TAGED*. If $\mathcal{L}ng(\mathrm{ta}(\mathcal{A})) = \varnothing$ then $\mathcal{L}ng(\mathcal{A}) = \varnothing$.*

Proof. We know that, trivially, $\mathcal{L}\mathrm{ng}(\mathcal{A}) \subseteq \mathcal{L}\mathrm{ng}(\mathrm{ta}(\mathcal{A}))$, and this result follows. □

Testing emptiness of a tree automaton is linear, so this is a very inexpensive test – which is quite fortunate as we will be using it fairly often. In the particular case when a positive TAGED is diagonal – that is to say, $=_{\mathcal{A}} \subseteq \{(q, q) \mid q \in Q\}$: all its equality constraints are of the form $q =_{\mathcal{A}} q$ – then this linear test is enough to decide emptiness.

Theorem 56 (Diagonal testing)**.** *Let $\mathcal{A}$ be a diagonal positive* TAGED*. Then $\mathcal{L}ng(\mathcal{A}) = \varnothing \iff \mathcal{L}ng(\mathrm{ta}(\mathcal{A})) = \varnothing$.*

Proof. See beginning of proof of [FTT08b, Theorem 1]. □

Now that those preliminary observations are over and done with, let us move on to what we announced earlier on: the observation of the contradictions which the introduction of global equality constraints can create in a tree automaton. We will see that some rules become absurd and some states unusable when certain conditions are met. We call those rules and states "spurious". We will

define those conditions and show that, just as was the case for useless states, spurious elements can be removed from a TAGED without altering its accepted language. We start by the most obvious observation:

Definition 57 (Spurious rule)**.** Let $\mathcal{A}$ be a TAGED. A rule $f(q_1, \ldots, q_n) \to q \in \Delta$ is *spurious* if there exists $k \in [\![1, n]\!]$ such that $q_k =_{\mathcal{A}} q$.

It is clear that no spurious rule can actually be used in any run. If that was the case then there would exist a term structurally equal to one of its strict subterms, which is absurd. So it follows that spurious rules have no influence whatsoever on the language recognised by a TAGED. This outlines the proof of the next lemma.

Lemma 58 (Removal of spurious rules)**.** *Let $\mathcal{A}$ be a TAGED, and let $S \subseteq \Delta$ be the set of all the spurious rules of Δ. Then, if we let $\mathcal{A}' \stackrel{def}{=} (\Sigma, Q, F, \Delta \setminus S, =_{\mathcal{A}}, \neq_{\mathcal{A}})$, we have $\mathcal{L}ng(\mathcal{A}) = \mathcal{L}ng(\mathcal{A}')$.*

Proof. We have trivially $\mathcal{L}\mathrm{ng}(\mathcal{A}) \supseteq \mathcal{L}\mathrm{ng}(\mathcal{A}')$. Let $t \in \mathcal{L}\mathrm{ng}(\mathcal{A})$ and let ρ be the run by which t has been accepted. Being a run, ρ is compatible with the transition rules, that is to say for any position $\alpha \in \mathcal{P}os(t)$, there exists a transition rule

$$r = t(\alpha)\Big(\rho(\alpha.1), \ldots, \rho(\alpha.arity(t(\alpha)))\Big) \to \rho(\alpha) \in \Delta.$$

Suppose that r is spurious. Then there is a k such that $\rho(\alpha.k) =_{\mathcal{A}} \rho(\alpha)$, and it follows that $t|_\alpha = t|_{\alpha.k}$. Thus t is structurally equal to its own child, which is absurd. Therefore r is not spurious: $r \in \Delta \setminus S$, and it follows that ρ is also a run for $\mathcal{A}'$. Finally we have $\mathcal{L}\mathrm{ng}(\mathcal{A}) \subseteq \mathcal{L}\mathrm{ng}(\mathcal{A}')$, which concludes the proof. □

We will now extend this notion of "spurious construction" to less direct cases, where instead of having immediate spurious rules, we have two or more rules leading to the same kind of contradictions. The watchful reader will notice an uncanny similarity between the *potential* requirements introduced at the beginning of the present chapter and the *sure* requirements which we are about to define. While the former generously encompassed all the states which *could* possibly be direct children of some state q, the latter is limited to the very closed circle of those states which *must* appear as direct children of q, for the simple reason that every single rule which builds q uses them as antecedents.

Definition 59 (Sure requirements)**.** Let $\mathcal{A}$ be a TAGED, and let $q \in Q$. The *sure*

requirements of state q are defined as[(a)]

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

We extend this notion in the same way we did before when we went from potential requirements to friends: if q is sure to need p as its direct child, and p is sure to need p' as its own direct child, then q is sure to need p' to be its grand-child. If p' does not appear in a run, neither can q. We call "needs of q" the set of states which, according to this principle, must appear in a run if q itself appears.

Definition 60 (Needs)**.** Let $\mathcal{A}$ be a TAGED, and $q \in Q$. We define $\mathfrak{Need}(q)$ as the smallest subset of Q satisfying the two following properties:

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

We now formalise and prove what we said informally: if a state appears in a run, then it is necessary that its needs should appear under it in this same run.

Lemma 61 (Needs)**.** *Let $\mathcal{A}$ be a TAGED, and let $t \in \mathcal{T}(\Sigma)$, $\beta \in \mathcal{P}os(t)$ and $q \in Q$. Let ρ be a run of $\mathcal{A}$ on t, compatible with the global constraints, such that $\rho(\beta) = q$. Then for any $p \in \mathfrak{Need}(q)$, there exists a position $\alpha_p \lhd \beta$ such that $\rho(\alpha_p) = p$.*

Proof. We prove this result by induction on $\mathfrak{Need}(q)$.

1. (base case) Suppose $p \in \mathfrak{sReq}(q)$. Since ρ is a run, it is compatible with the transition rules of Δ. We have $\rho(\beta) = q$, therefore, letting n be the arity of $t(\beta)$, there exists a rule $f(q_1, \ldots, q_n) \to q \in \mathfrak{Rul}(q)$ such that for all $k \in [\![1, n]\!]$, $q_k = \rho(\beta.k)$. By definition of $\mathfrak{sReq}(q)$, there exists $i \in [\![1, n]\!]$ such that $p = q_i = \rho(\beta.i)$. We have $\alpha_p = \beta.i \lhd \beta$.

(a) *Update:* This definition can be changed to

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

and the interest of doing so is that $\mathfrak{Need}(q)$ becomes larger in general, while retaining its properties, and so we can catch a few more cases. [TODO] integrate that.

2. (inductive case) Suppose that there exists $p' \in \mathfrak{Need}(q)$ such that $p \in \mathfrak{sReq}(p')$. By induction hypothesis there exists $\alpha_{p'} \lhd \beta$ such that $\rho(\alpha_{p'}) = p'$. We use the same arguments as in the base case. Let m be the arity of $t(\alpha_{p'})$. Then there exists a rule $f(q_1, \ldots, q_m) \to p' \in \mathfrak{Rul}(p')$ such that for all $k \in [\![1, m]\!]$, $\rho(\alpha_{p'}.k) = q_k$. By definition of $\mathfrak{sReq}(p')$, there exists $i \in [\![1, m]\!]$ such that $p = q_i = \rho(\alpha_{p'}.i)$. We have $\alpha_p = \alpha_{p'}.i \lhd \alpha_{p'} \lhd \beta$.

Thus the proof is concluded. □

In other words, in order to "build" the state q, one must first be able to build any state $p \in \mathfrak{Need}(q)$ strictly under it. Suppose that we are in the following scenario: we have some states $q_0, \ldots, q_n$ such that $q_0 =_{\mathcal{A}} q_n$, some symbols $f_0, \ldots, f_n$ (not necessarily all distinct) and the rules

$$\begin{aligned} f_0(\ldots, q_0, \ldots) &\to q_1 && \in \Delta \\ &\ldots \\ f_k(\ldots, q_k, \ldots) &\to q_{k+1} && \in \Delta \\ &\ldots \\ f_{n-1}(\ldots, q_{n-1}, \ldots) &\to q_n && \in \Delta \end{aligned}$$

If, for any $k \in [\![1, n]\!]$, we have no rule $r \in \mathfrak{Rul}(q_k)$ such that $q_{k-1} \notin \mathfrak{Int}(r)$, that is to say, if there is no way to build q_k without first building q_{k-1}, then it is impossible to build a term which evaluates to q_n. Indeed, such a term would necessarily have one of its strict children evaluate to q_0. But this is not compatible with $q_0 =_{\mathcal{A}} q_n$. Such a state q_n will be called "spurious". The following definitions and lemma characterise spurious states and formalise the intuitive notion that spurious states can be removed from a `TAGED` without altering its accepted language.

Definition 62 (Spurious states)**.** Let $\mathcal{A}$ be a `TAGED`. A state $q \in Q$ is said to be a *spurious state* if there exists $p \in \mathfrak{Need}(q)$ such that $p =_{\mathcal{A}} q$.

That definition is rather intuitive: if you *need* p to be a strict child of q, and at the same time you need p and q to be structurally equal, then you are in trouble... As announced, this generalises the notion of spurious rules to cases where the child-father structural equality is buried a little more deeply. Clearly, one can safely get rid of such states:

Lemma 63 (Removal of spurious states)**.** *Let $\mathcal{A}$ be a `TAGED`, $S \subseteq Q$ the set of all its spurious states, and $\mathcal{A}' = \mathfrak{Rst}(\mathcal{A}, Q \setminus S)$. Then $\mathcal{L}ng(\mathcal{A}) = \mathcal{L}ng(\mathcal{A}')$.*

Proof. It is clear that $\mathcal{L}\mathrm{ng}(\mathcal{A}) \supseteq \mathcal{L}\mathrm{ng}(\mathcal{A}')$. Let $t \in \mathcal{L}\mathrm{ng}(\mathcal{A})$ and let ρ be the run by which t has been accepted. Suppose that there is a position $\beta \in \mathcal{P}os(t)$ such that $q = \rho(\beta) \in S$. Then by definition of a spurious state there exists a state $p \in \mathfrak{Need}(q)$ such that $p =_{\mathcal{A}} q$, and by lemma 61[p94], there exists a position $\alpha_p \lhd \beta$ such that $\rho(\alpha_p) = p$. By the equality constraint we have $t|_\beta = t|_{\alpha_p}$, but this is impossible because no term may be structurally equal to one of its strict children. Thus for all positions $\beta \in \mathcal{P}os(t)$, $\rho(\beta) \in Q \setminus S$, and ρ is also a run for $\mathcal{A}'$. Finally we have $\mathcal{L}\mathrm{ng}(\mathcal{A}) \subseteq \mathcal{L}\mathrm{ng}(\mathcal{A}')$, which concludes the proof. □

Until now we have only focused on spurious constructions based on the obvious impossibility of building a term equal to one of its strict subterms. We will now see another impossibility, based on the symbols of Σ.

Definition 64 (Support of a state[(b)])**.** Let $\mathcal{A}$ be a `TAGED`, and let $q \in Q$ be a state. We call *support of* q and denote $\mathfrak{Sup}(q)$ the set of all symbols of Σ in which a term which evaluates to q may be rooted. $\mathfrak{Sup}(q) \stackrel{\text{def}}{=} \{ f \in \Sigma \mid \exists f(\dots) \to q \in \Delta \}$.

Short of actually testing a full structural equality, it can be useful, be very inexpensive, to at least see whether the roots of two trees evaluating to to different states *can* possibly have the same symbol. Say that you have the constraint $p =_{\mathcal{A}} q$, but all the rules of $\mathfrak{Rul}(p)$ are of the form $f(\dots) \to p$, while all the rules of $\mathfrak{Rul}(q)$ are of the form $g(\dots) \to q$. Clearly, while those states could very well appear in runs, they cannot in any way appear *together*. Suppose now that there is a state which *requires* p and q to appear together in the run; then clearly the use of this state yields a contradiction, and again, it can be safely removed. This is what we call a Σ-spurious state.

Definition 65 (Σ-spurious state)**.** Let $\mathcal{A}$ be a `TAGED`. A state $q \in Q$ is said to be a Σ-*spurious state* if there exists $p, p' \in \mathfrak{Need}(q)$ such that $p =_{\mathcal{A}} p'$ and $\mathfrak{Sup}(p) \cap \mathfrak{Sup}(p') = \emptyset$.

Lemma 66 (Removal of Σ-spurious states[(c)])**.** *Let $\mathcal{A}$ be a `TAGED`, $S \subseteq Q$ the set of all its Σ-spurious states, and $\mathcal{A}' = \mathfrak{Rst}(\mathcal{A}, Q \setminus S)$. Then $\mathcal{L}ng(\mathcal{A}) = \mathcal{L}ng(\mathcal{A}')$.*

Proof. It is clear that $\mathcal{L}\mathrm{ng}(\mathcal{A}) \supseteq \mathcal{L}\mathrm{ng}(\mathcal{A}')$. Let $t \in \mathcal{L}\mathrm{ng}(\mathcal{A})$ and let ρ be the run by which t has been accepted. Suppose that there is a position $\beta \in \mathcal{P}os(t)$ such that $q = \rho(\beta) \in S$. Then by definition of a Σ-spurious state there exist

(b) Similar to τ introduced in section 4.2[p34]

(c) Note that it would have been quite wrong to define a Σ-spurious state simply as a state such that $\exists p' \in Q : p =_{\mathcal{A}} p'$ and $\mathfrak{Sup}(p) \cap \mathfrak{Sup}(p') = \emptyset$. Such a state *can* be used in an accepting run, provided that its "opposite" p' does not appear in the same run.

two states $p, p' \in \mathfrak{Need}(q)$ such that $p =_{\mathcal{A}} p'$, and by lemma 61[p94], there exist two distinct positions $\alpha, \alpha' \lhd \beta$ such that $\rho(\alpha) = p$ and $\rho(\alpha') = p'$. Since ρ is a run, it is compatible with the transition rules, and therefore $t(\alpha) \in \mathfrak{Sup}(p)$ and $t(\alpha') \in \mathfrak{Sup}(p')$. By the equality constraint we have $t|_\alpha = t|_{\alpha'}$, and thus $t(\alpha) = t(\alpha')$ and it follows that $t(\alpha) \in \mathfrak{Sup}(p) \cap \mathfrak{Sup}(p') = \emptyset$. This is absurd. Thus for all positions $\beta \in \mathcal{P}os(t)$, $\rho(\beta) \in Q \setminus S$, and ρ is also a run for $\mathcal{A}'$. Finally we have $\mathcal{L}\mathrm{ng}(\mathcal{A}) \subseteq \mathcal{L}\mathrm{ng}(\mathcal{A}')$, which concludes the proof. □

The act of removing all the spurious constructions from the `TAGED` is called *sanitising*. It is legitimated by the following theorem, which summarises the results of this section.

Theorem 67 (Sanitising). *Let $\mathcal{A}$ be a `TAGED`, and let $Q_s \subseteq Q$ the set of all its spurious states, $Q_\Sigma \subseteq Q$ the set of all its Σ-spurious states, and $\Delta_s \subseteq \Delta$ the set of all its spurious rules. Then if we let $\mathcal{A}' = \mathfrak{Rst}\left((\Sigma, Q, F, \Delta \setminus \Delta_s, =_{\mathcal{A}}, \neq_{\mathcal{A}}), F \cup \bigcup_{q_f \in F} \mathfrak{Frnd}(q_f) \setminus (Q_s \cup Q_\Sigma)\right)$ we have $\mathcal{L}ng(\mathcal{A}) = \mathcal{L}ng(\mathcal{A}')$.*

Proof. Immediate consequence of corollary 54[p92] and lemmas 58[p93], 63[p95] and 66[p96]. □

Note that the resulting automaton is not necessarily the smallest one can obtain using this method: in some edge cases, removal of useless states might render some other states spurious and *vice versa*. So, in practice, we shall use this theorem repeatedly until a fixed point is reached. This operation will be referred to as "cleanup".

12.2 Algorithms and implementation

In this section we present the algorithms corresponding to the theory which we have just introduced. Rather than presenting pseudo-code, we give extracts of the actual implementation of the (second) `OCaml` prototype which we have developed to validate our results. The code is automatically converted to LaTeX, highlighted and prettified through a program which we have written for this purpose, about which some information is given at the end of section 2.3[p22]. Note that part of the code has been masked – for instance the module declarations are incomplete, and the assertions are invisible – to avoid cluttering the listings overmuch with programming details and save space overall.

Before writing any algorithm, we need to define our data types – in particular, we need to have a full representation for a `TAGED` – and a few obvious and easy

functions to manipulate those types, which were introduced in the previous section. The reader will notice that those definitions are little more than the mathematical definitions of the previous section, rewritten in Caml's own syntax. Inasmuch as possible, the same notations have been taken in the theoretical sections and in the implementation.

```
(* Here we define the basic type of TAGED. *)

(* A symbol is represented by a string. The arity of the symbols will be kept
separately *)
type symbol = string

(* A state q ∈ Q is represented by a string *)
type state = string

(* A transition rule r = f(q_1,...,q_n) → q is represented by the triplet
(f,[q_1,...,q_n],q) *)
type transition_rule = symbol × state list × state

(* We use the Set structure extensively. It has the advantage of having the
behaviour the closest to the theory (we use sets in the theory, after all) while
having a reasonnably efficient implementation in OCaml, based on balanced
binary trees. Here we use XSet, which is merely a trivial extension of the
Set.Make functor of the standard library, with a few added operations. More
information on the operations of XSet can be found at the end of section
2.3[p22]. *)

module Q = XSet (struct type t = state end)
module Σ = XSet (struct type t = symbol end)
module Δ = XSet (struct type t = transition_rule end)

(* We represent a relation R ⊆ Q^2 simply by the corresponding set. *)
module Q² = XSet (struct type t = state × state end)

(* Tree automaton with global equality and disequality constraints: TAGED. *)
type taged = {
    (* The alphabet Σ, an the associated arity: Σ → ℕ function. *)
    Σ : Σ.t; arity : Σ.elt → int;
```

```
25
26      (* The transition rules Δ, states Q and final states F ⊆ Q *)
27      Δ : Δ.t; Q : Q.t; F : Q.t;
28
29      (* And lastly, the TAGED-specific part: the global constraints =_A, ≠_A ⊆ Q². Note
        that the symmetry of both relations is garanteed by the parser *)
30      =_A : Q².t; ≠_A : Q².t;
31    }
32
33  (* Standard functions for manipulating TAGEDs *)
34
35  (* Compute restriction by states ℜst (A, set). *)
36  let ℜst A set =
37    let in_set_square (p,q) = Q.∈ p set ∧ Q.∈ q set in
38    let r_in_set (⊥,[q1,...,qn],q) =
39      L.∀ (λqk→Q.∈ qk set) (q::[q1,...,qn])
40    in { A with
41      Q = set;
42      F = Q.∩ A.F set;
43      Δ = Δ.filter r_in_set A.Δ;
44      =_A = Q².filter in_set_square A.=_A;
45      ≠_A = Q².filter in_set_square A.≠_A;
46    }
47
48  (* Get the domain of a relation.*)
49  let domain_of_rel ?(strict=false) r =
50    let f (x,y) set = if ¬strict ∨ x ≠ y then
51      Q.add⁺ [x;y] set else set
52    in Q².fold f r Q.∅
53
54  (* Antecedents of a rule *)
55  let 𝔄nt (⊥,[q1,...,qn],⊥) = Q.of_list [q1,...,qn]
56
57  (* Associated rules of a state *)
58  let ℜul A q = Δ.filter (λ(⊥,⊥,p)→p=q) A.Δ
59
60  (* Support of a state *)
61  let 𝔖up A q = Σ.of_list[[ Δ.map_L (λ(f,⊥,⊥)→f) (ℜul A q)
```

Now that the foundations have been laid, we can give the algorithms of this section. They are all quite simple: the first one computes the set of reachable states of a tree automaton, which is useful for testing emptiness and reducing the automaton. Then the notions of the previous section are coded, and once again, they are a rather straightforward rewriting of the theory.

```
(* Standard algorithm to compute the set of all reachable states of the underlying tree automaton of a TAGED 𝒜. This algorithm is linear and follows the
62  idea given in [CDG+07, Exercice 1.18 p47]. *)
63  let reachable_states 𝒜 =
64
      (* We associate each state q ∈ Q with the list of all rules r ∈ Δ such that
65    q ∈ 𝔄𝔫𝔱(r). For now, we create an empty association table *)
66    let state_rules = H.create (Δ.# 𝒜.Δ) in
67
      (* Here, we represent a rule r ∈ Δ by a counter initialised to Card (𝔄𝔫𝔱(r)) and
      its destination state q ∈ Q. We will in the same move convert the rules of Δ
68    to a counter, and populate state_rules. *)
69    let rule_to_count (f,[q1,...,qn],q) =
70      let 𝔄𝔫𝔱 = 𝔄𝔫𝔱 (f,[q1,...,qn],q) in
71      let counter = (q, ref⟦ Q.# 𝔄𝔫𝔱) in
72      Q.iter (λqi→H.add state_rules qi counter) 𝔄𝔫𝔱;
73      counter
74    in let rules_count = Δ.map_ℒ rule_to_count 𝒜.Δ in
75
      (* Now, we compute the set of reachable states as follows: a state q is "activated" (ie. flagged as reachable) when the counter of a rule in ℜ𝔲𝔩(q) is zero,
      that is to say, when all the states required to get into state q have been flagged
      as reachable. Thus, we will start the algorithm by activating the states in
      which the leaves (constant symbols) can evaluate: if there is a rule a → q
      then q is activated. When a state q is activated, the counters of all rules which
76    needed q are decremented. The same state is not activated twice. *)
77
78    let reachable = ref Q.∅ in
79    let rec decrement (q,c) = match !c with
80      | 1 → decr c; activate q
81      | n when n ⩾ 2 → decr c
82    and activate q =
83      if ¬⟦ Q.∈ q !reachable then (
84        Q.↩ q reachable;
```

```
85        let q_rules = H.find_all state_rules q in
86        L.iter decrement q_rules;
87      )
88    in let trigger (q,c) = if !c = 0 then activate q in
89    L.iter trigger rules_count; return !reachable
90
```

(* Is the language accepted by the tree automaton or the diagonal positive TAGED $\mathcal{A}$ empty? It is clear that $\mathcal{A}$ is empty if and only if none of its final *)
91 states is reachable. *)

```
92  let ta_empty A = Q.∅?[[ Q.∩ (reachable_states A) A.F
93  let trivial_empty A = Q.∅? A.F
94
```

95 (* Compute reduced automaton *)

```
96  let reduction A = 𝔎st A [[ reachable_states A
97
```

98 (* Remove the spurious rules of $\mathcal{A}$. *)

```
99  let rm_spurious_rules A =
100   let nspurious (⊥,[q1,...,qn],q) =
101     ¬[[L.∃ (λqi→Q².∈ (qi,q) A.=A) [q1,...,qn]
102   in reduction {A with Δ = Δ.filter nspurious A.Δ}
103
```

104 (* Sure and Potential requirements *)

```
105 let 𝔰Req A q =
106   let ante_sets = Δ.mapL 𝔄nt (𝔎ul A q) in Q.∩+ ante_sets
107 let 𝔭Req A q =
108   let ante_sets = Δ.mapL 𝔄nt (𝔎ul A q) in Q.∪+ ante_sets
109
```

110 (* Inductive refinement for both Needs and Friends *)

```
111 let refine basefun q =
112   let rec iter set =
113     let reqs = Q.∪+[[ Q.mapL basefun set in
114     if Q.⊆ reqs set then set else iter (Q.∪ set reqs)
115   in iter[[ basefun q
116
117 let 𝔑eed A = refine (𝔰Req A)
118 let 𝔉rnd A = refine (𝔭Req A)
119
```

120 (* Remove the spurious and Σ-spurious states from $\mathcal{A}$. *)

```
121 let rm_spurious_states A =
```

```
122   let spurious q = Q.∃ (λp→Q².∈ (p,q) 𝒜.=𝒜) (𝔑eed 𝒜 q) in
123   let Σ-spurious q = Q².∃ (λ(p,p')→
124       Q.∈⁺ [p;p'] (𝔑eed 𝒜 q) ∧
125       Σ.∅?[[ Σ.∩ (𝔖up 𝒜 p) (𝔖up 𝒜 p')
126     ) 𝒜.=𝒜
127   in let ok q = ¬(spurious q) ∧ ¬(Σ-spurious q)
128   in reduction [[ 𝔑st 𝒜 (Q.filter ok 𝒜.Q)
129
    (* Remove useless states – ie. states which are neither final states nor friends
130   of final states *)
131 let rm_useless_states 𝒜 = reduction [[ 𝔑st 𝒜 [[
132   Q.∪ (𝒜.F) (Q.∪⁺ (Q.map_ℒ (𝔉rnd 𝒜) 𝒜.F))
133
134 (* TAGED reduction (sanitising) *)
135 let sanitize = rm_useless_states ∘ rm_spurious_states ∘ rm_spurious_rules
136
    (* Stronger sanitize, which make sure that a fixed point is reached and the
137   automaton is minimal. *)
138 let rec cleanup 𝒜 =
139   let tac = sanitize 𝒜 in let tacc = sanitize tac in
140   if Q.# tac.Q = Q.# tacc.Q then tacc else cleanup tacc
```

12.3 Experimental results

The experimental results concerning the cleanup operation are quite good considering that it is so simple. The algorithms presented in the previous section are clearly polynomial, and yet TAGEDs (or at least our random specimens) are trimmed down by a factor which can sometimes be considerable. See for instance figure 12.2: as mentioned in section 8.2[p61], it shows the ratio

$$R = \frac{\text{size of generated automata}}{\text{size of the same, } \textit{cleaned up}},$$

as a function of the density of global constraints as defined in the second model (see section 9.2[p78]), for different values of δ (the expected in-degree – again, see section 8.2[p61]) and $|Q|$. On this figure one can see that random automata which are very spare are reduced to a small fraction of their former selves: cut down by a factor of almost 60 in the worst case, and are quite sensitive to the density of constraints. On the other hand, denser automata are less affected, both in general and by the density of constraints, which is quite logical: when $\mathfrak{Rul}(q)$

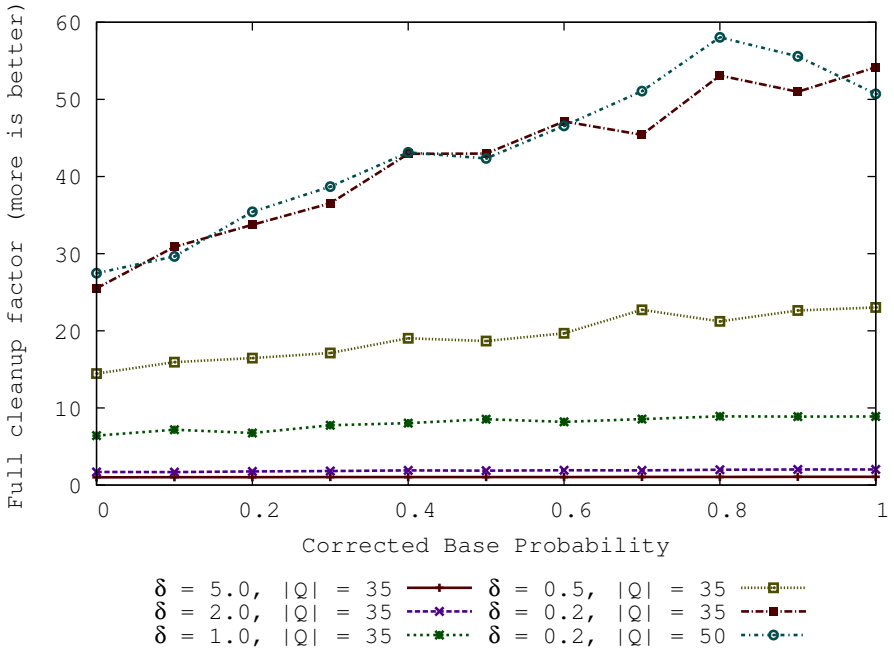


Figure 12.2 — Gene2: Full cleanup factor (raw *vs.* cleanup)

grows large, $\mathfrak{sReq}(q)$ tends to become empty, which limits the usefulness of the technique. To clarify the effect a bit, figure 12.3 shows a more pertinent ratio

$$R' = \frac{\text{size of } \textit{reduced} \text{ generated automata}}{\text{size of the same, } \textit{cleaned up}},$$

where *reduced* is taken to mean "through the standard reduction algorithm (*cf.* [CDG+07])". Recall that the standard reduction algorithm is part of the cleanup operation, but is not our own contribution. While the first graph took into account both the standard reduction algorithm and our own methods, this second graph shows the gain incurred by using our method on top of the standard algorithm. We see that[(d)], for sparse automata, we cut their size down by a factor of five, compared to what the reduction algorithm alone did.

[(d)] We also see that the gain does not appear to depend on the density of constraints, which I can't honestly explain. However these statistics had been made rather early on, and since then a small, vicious lurking bug has been detected through random generation, and corrected, in the implementation of the reduction algorithm. I am pretty sure that if we were to redo this experiment with the corrected implementation, we would see the same slant as in the first graph. Meanwhile, the numbers in these graphs can be taken as *lower bounds* to the real numbers. I apologise for the inconvenience.

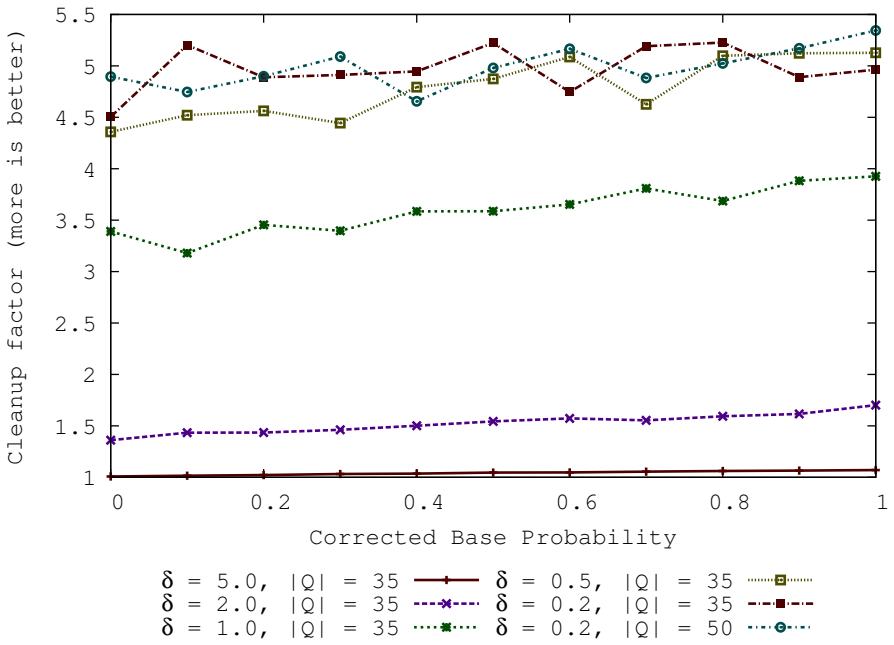


Figure 12.3 — Gene2: Cleanup factor (reduced *vs.* cleanup)

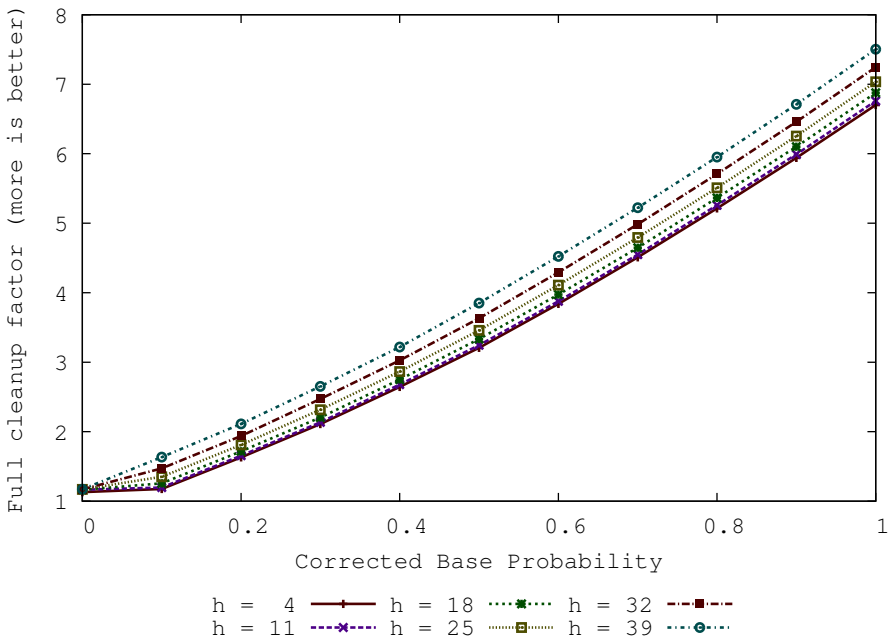


Figure 12.4 — Gene4: Cleanup factor

As discussed in section 8.2[p61], the efficiency of the cleanup operation when there are no constraints can also be taken to mean that the sample automata (from the second generation) are simply not too good: too many of their states are simply completely useless, disconnected from their final states. We have in figure 12.4 the same results, but this time with random TAGEDs of the fourth generation, taken with the second model of constraints generation. Recall that Gene4 yields TAGEDs which are already reduced, so this curve, similarly to figure 12.3, shows only the influence of methods specific to *cleanup*.

12.4 Conclusion

In this chapter, we have introduced the *cleanup* operation, which improves on the standard reduction algorithm for vanilla tree automata (*cf.* [CDG+07]), and takes advantage of the global equality constraints of a TAGED to detect and remove even more rules and states. The cleanup operation itself has a low complexity – polynomial, and is primarily intended to be used as preliminary to a more expensive algorithm, such as emptiness. Nevertheless, it should be noted that in many cases, it can be quite enough to decide: see for instance the two examples below [(e)].

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

This TAGED ('example 1') is in fact empty, and a cleanup operation suffices to show that it is. Its sole final state is q_6, which depends on q_0 and q_1. The former is not a problem (leaf state) but the latter depends on both q_3 and q_4 – they are sure requirements. We have $q_3 =_{\mathcal{A}} q_4$, but $\mathfrak{Sup}(q_3) = \{a_3\}$ and $\mathfrak{Sup}(q_4) = \{a_2, a_5\}$. Therefore q_1 is Σ-spurious. Remove q_1 from this TAGED, and q_6 becomes unreachable: without any final state, the automaton is empty.

Let us take another example:

[(e)] Kindly provided by random generation. *cf.* 8.4[p71].

```
TAGED 'example 2' [44] = {
  states = #6{q0, q1, q2, q3, q4, q5}
  final = #1{q5}
  rules = #11{
    a2()->q2, a2()->q3, a3()->q0, a3()->q3, a3()->q4, a4()->q2,
    a5()->q4, g3(q5, q0)->q5, g4(q1, q1)->q5, h1(q2, q3, q4)->q1,
    h3(q2, q3, q4)->q1
  }
  ==rel = #3{(q1,q3), (q3,q1), (q5,q5)}
}
```

Here again, we have only one final state q_5, which depends on q_1. There are two rules which generate q_1, but as it happens both of them have q_3 in their antecedents. We have $q_1 =_{\mathcal{A}} q_3$, and thus those two rules are spurious. q_1 becomes unreachable, and consequently so does q_5. The TAGED is empty.

And these are in no way isolated cases: the method used to generate those examples reports that 13.5% of random fourth generation TAGEDs of height 3 – such as those ones – 24.5% of TAGEDs of height 6 and 30.7% of TAGEDs of height 20 are non-empty when reduced but empty once cleaned up.

All in all, the *cleanup* operation is a valuable, cheap tool which can greatly simplify and speed up the operation of more expensive algorithms on TAGEDs – such as emptiness decision – and enables us to conclude immediately in a surprising number of cases.

Chapter 13

Signature quotienting

The material in this chapter is born from an observation made on some hand-written TAGEDs which we used during our experiments with the membership and emptiness problems. In those TAGEDs, we would have a state, say, q_{char}, coding for instance an alphanumeric character, and this single state would have a great number of rules, all similar except for the symbol they carry. Here, we would have

$$\mathfrak{Rul}(q_{\text{char}}) = \{\, a \to q_{\text{char}}, \ldots, z \to q_{\text{char}}, A \to q_{\text{char}}, \ldots, Z \to q_{\text{char}} \,\},$$

which is a very verbose way of saying, essentially

$$\text{“}\{a, \ldots, z, 0, \ldots, 9, A, \ldots, Z\} \to q_{\text{char}} \in \Delta\text{”},$$

with quotation marks all around because of course the type of this is not quite right. This being said, we have here an awful lot of rules which *mean* something quite simple. Having lots and lots of rules is a very detrimental thing for us because lots of rules means lots of possible choices for the brutal algorithm trying to build a successful run (*cf.* chapter 15[p131]), which will most probably timeout before going very far. In contrast, when a human being performs the equivalent of the brutal algorithm in their head, they will most probably perform subconsciously a rather obvious optimisation: It does not *immediately* matter which character is chosen for q_{char}; so rather than exploring head-first all the possible combinations, one can just leave “q_{char}” as it is for now, and worry about *other states*. Later on, for instance when asked to exhibit a term, or when trying to satisfy a global equality constraint, one can safely choose a concrete symbol for q_{char} among the many allowed by the transition rules.

In other words, and slightly more generally, when rules differ only by their symbol, the choice of this symbol can be delayed until necessary. This idea is discussed further in the two next sections.

Note that since it is a rather specialised optimisation, and one which is unlikely to be of significant use on random automata – unless we came up with a generation method designed specifically to generate edge cases like this on a regular basis – we have not insisted overmuch on it. Nevertheless, as pointed out by our example, this edge case seems to us common enough to warrant at least minimal inquiry.

13.1 A first attempt

In this section we present our first attempt at translating the basic idea that when several transition rules may apply, only their signatures matter at first, and the choice of the symbol can be deferred until the equality constraint is tested.

As noted in the introduction, the "type" of transition rules changes, as symbols become sets of symbols. Thus in order to capture the idea, we need to introduce a new data structure:

Definition 68 (signature-TAGED)**.** Let $\mathcal{A}$ be a positive TAGED. We call *signature-TAGED of* $\mathcal{A}$ the tuple[(a)] $\mathcal{A}^s = (\Sigma^s, Q, F, \Delta^s, =_{\mathcal{A}})$ where

$$\Delta^s = \left\{ \Phi(q_1, \dots, q_n) \to q \;\middle|\; \Phi = \{ f \in \Sigma : f(q_1, \dots, q_n) \to q \in \Delta \} \neq \varnothing \right\}$$
$$\Sigma^s = \left\{ \Phi \subseteq \Sigma \;\middle|\; \Phi(\dots) \to q \in \Delta^s \right\}.$$

With these definitions, the arity of $\Phi \in \Sigma^s$ is that of any symbol in Φ (all symbols in Φ have the same arity by definition). We define the *signature-equivalence* between two terms in $\mathcal{T}(\Sigma^s)$ as follows: Let $U = \Phi(U_1, \dots, U_n) \in \mathcal{T}(\Sigma^s)$ and $V = \Psi(V_1, \dots, V_m) \in \mathcal{T}(\Sigma^s)$

$$U \approx^s V \iff (n = m) \wedge (\Phi \cap \Psi \neq \varnothing) \wedge (\forall i \in [\![1, n]\!] : U_i \approx^s V_i).$$

A *signature-term* $T \in \mathcal{T}(\Sigma^s)$ is *accepted by the signature-TAGED* $\mathcal{A}^s$ if the two following conditions hold:

- ⋄ it is accepted by the tree automaton $(\Sigma^s, Q, F, \Delta^s)$ — say, by the run ρ.

(a) Note that this is not *exactly* a TAGED...

- ◇ and ρ is *signature-compatible* with the constraint $=_{\mathcal{A}}$, that is to say

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

Signature-TAGEDs and vanilla TAGEDs are similar in every other respect.

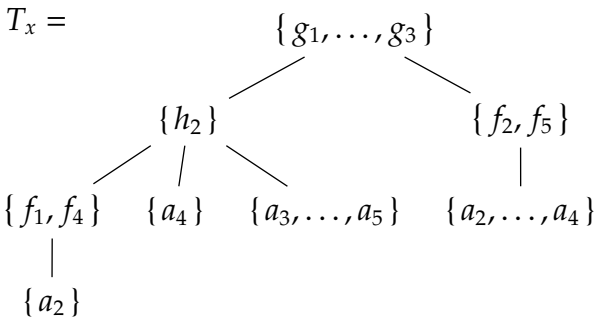
As we are manipulating terms over sets of symbols, at some point we will wish to go back to simple terms over symbols, by choosing a symbol for each node among those available:

Definition 69 (Signature-expansions)**.** Let $\mathcal{A}$ be a positive TAGED and $\mathcal{A}^s$ be its signature-TAGED. Let $T \in \mathcal{T}(\Sigma^s)$. We define its *signature-expansion*

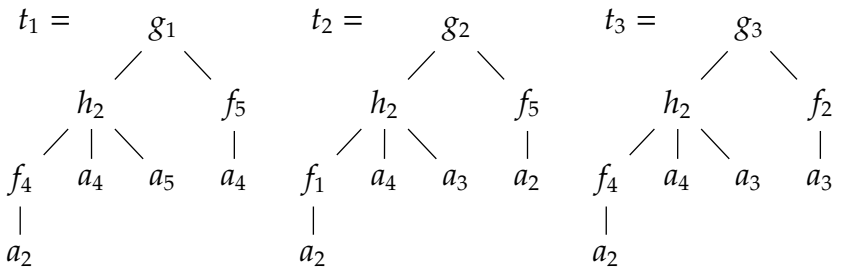
$$\mathfrak{Exp}(T) \stackrel{\text{def}}{=} \{ t \in \mathcal{T}(\Sigma) \mid \mathcal{P}os(t) = \mathcal{P}os(T) \text{ and } \forall \alpha \in \mathcal{P}os(t) : t(\alpha) \in T(\alpha) \} .$$

We note that, since any $\Phi \in \Sigma^s$ is non-empty, $\mathfrak{Exp}(T) \neq \varnothing$ for any signature-term T.

Let us illustrate this definition with an example: let T_x be a signature-term:



We can expand it into any of



so we have $t_1, t_2, t_3 \in \mathfrak{Sxp}(T_x)$, among a total of $|\mathfrak{Sxp}(T_x)| = 2 \times 2 \times 3 \times 3 \times 3 = 108$, if I count correctly. Those are as many choices which are delayed during the construction of T_x, by a brutal algorithm for instance, until checking a structural property of the term becomes necessary. Let us now link the signature-equivalence which we have defined earlier with this notion of signature-expansion:

Lemma 70. *Let $U, V \in \mathcal{T}(\Sigma^s)$; then $U \approx^s V \iff \mathfrak{Sxp}(U) \cap \mathfrak{Sxp}(V) \neq \emptyset$.*

Proof. Suppose $U \approx^s V$. It is clear that $\mathcal{P}os(U) = \mathcal{P}os(V)$. Then we take t such that $\mathcal{P}os(t) = \mathcal{P}os(U) = \mathcal{P}os(V)$ and $\forall \alpha \in \mathcal{P}os(t) : t(\alpha) \in U(\alpha) \cap V(\alpha)$. Such a t exists since by definition of $\approx^s$, we have $\forall \alpha \in \mathcal{P}os(t) : U(\alpha) \cap V(\alpha) \neq \emptyset$. It is clear that $t \in \mathfrak{Sxp}(U) \cap \mathfrak{Sxp}(V)$. Conversely, suppose that there exists a $t \in \mathfrak{Sxp}(U) \cap \mathfrak{Sxp}(V)$. Then we have $\mathcal{P}os(t) = \mathcal{P}os(U) = \mathcal{P}os(V)$ and $\forall \alpha \in \mathcal{P}os(t) : t(\alpha) \in U(\alpha) \cap V(\alpha)$ and thus $U \approx^s V$. □

Now, let us attempt to bridge the gap between those signature-TAGEDs and vanilla TAGEDs:

Lemma 71. *Let $\mathcal{A}$ be a positive TAGED and $\mathcal{A}^s$ its signature-TAGED. Then for each $t \in \mathcal{L}ng(\mathcal{A})$ there exists $T \in \mathcal{L}ng(\mathcal{A}^s)$ such that $t \in \mathfrak{Sxp}(T)$.*

Proof. Suppose that $t \in \mathcal{L}ng(\mathcal{A})$ is accepted by a run ρ. Then we build $T \in \mathcal{L}ng(\mathcal{A}^s)$ as follows: $\mathcal{P}os(T) = \mathcal{P}os(t)$ and

$$\forall \alpha \in \mathcal{P}os(T), T(\alpha) = \{ f \in \Sigma : f(\rho(\alpha.1), \ldots, \rho(\alpha.arity(t(\alpha))) \to \rho(\alpha) \in \Delta \}.$$

Since $t \in \mathcal{L}ng(\mathcal{A})$, and thus must be compatible with the transition rules, by this construction $t(\alpha) \in T(\alpha)$, and ρ is clearly a run of $\mathcal{A}^s$ on T. As for the equality constraint, we have $\forall \alpha, \beta \in \mathcal{P}os(t) : \rho(\alpha) =_{\mathcal{A}} \rho(\beta) \implies t|_\alpha = t|_\beta$. Since we have by construction $t|_\alpha \in \mathfrak{Sxp}(T|_\alpha)$ and $t|_\beta \in \mathfrak{Sxp}(T|_\beta)$, it follows that $t|_\alpha = t|_\beta \in \mathfrak{Sxp}(T|_\alpha) \cap \mathfrak{Sxp}(T|_\beta)$, and thus $T|_\alpha \approx^s T|_\beta$. It follows that $T \in \mathcal{L}ng(\mathcal{A}^s)$, which concludes the proof. □

This has an immediate consequence which is of interest to us:

Corollary 72 (Signature-TAGED as over-approximation[(b)])**.** *Let $\mathcal{A}$ be a positive TAGED and $\mathcal{A}^s$ its signature-TAGED. Then $\mathcal{L}ng(\mathcal{A}^s) = \emptyset \implies \mathcal{L}ng(\mathcal{A}) = \emptyset$.*

[(b)] Note that this is an abuse of language on our part, as there is no over-approximation in the usual sense, *ie.* with respect to inclusion.

Thanks to this last result, if we can prove emptiness of the signature TAGED, then we have proved that of the TAGED. We would certainly *like* the converse to be true, because then we could work exclusively with signature-TAGEDs to decide emptiness. But is it true? Let us be optimistic, and attempt to prove that it is indeed true.

Conjecture 73 (Signature-TAGED as under-approximation)**.** *Let $\mathcal{A}$ be a positive TAGED and $\mathcal{A}^s$ its signature-TAGED. Then for each $T \in \mathcal{L}ng(\mathcal{A}^s)$ there exists $t \in \mathcal{L}ng(\mathcal{A})$ such that $t \in \mathfrak{Sxp}(T)$.*

Proof. Or more accurately "doomed attempt at *a proof".* Suppose $T \in \mathcal{L}\mathrm{ng}(\mathcal{A}^s)$ is accepted by a run ρ. We attempt to build $t \in \mathfrak{Sxp}(T)$, such that $t \in \mathcal{L}\mathrm{ng}(\mathcal{A})$. First off, let us note and show that *any* term $t \in \mathfrak{Sxp}(T)$ is compatible with the transition rules of $\mathcal{A}$, and therefore accepted by the underlying tree automaton $\mathfrak{ta}(\mathcal{A})$. Let $\alpha \in \mathcal{P}os(T)$: since ρ is a run of $\mathcal{A}^s$, there exists a rule

$$T(\alpha)\,(\rho(\alpha.1), \ldots, \rho(\alpha.arity(T(\alpha)))) \to \rho(\alpha) \in \Delta^s,$$

and thus, by definition of t and Δ^s, there exists a rule

$$t(\alpha)\,(\rho(\alpha.1), \ldots, \rho(\alpha.arity(t(\alpha)))) \to \rho(\alpha) \in \Delta,$$

which shows that any t is indeed compatible with the transition rules of $\mathcal{A}$. The problem remains of finding a $t \in \mathfrak{Sxp}(T)$ which is also compatible with $=_{\mathcal{A}}$. We know that $\mathfrak{Sxp}(T|_\alpha) \cap \mathfrak{Sxp}(T|_\beta) \neq \varnothing$ because $T|_\alpha \cong^s T|_\beta$, and it follows that

$$\forall \alpha, \beta \in \mathcal{P}os(t) : \rho(\alpha) =_{\mathcal{A}} \rho(\beta) \implies \exists u_{\alpha,\beta} \in \mathfrak{Sxp}(T|_\alpha) \cap \mathfrak{Sxp}(T|_\beta).$$

Then why not choose $t|_\alpha = t|_\beta = u_{\alpha,\beta}$, and choose any symbol arbitrarily everywhere the constraints did not touch? The we would respect $=_{\mathcal{A}}$ by construction, right? Right? Well, not quite. So long as we consider only one constraint at a time, it works, but let us imagine that we have three distinct positions $\alpha, \beta, \gamma \in \mathcal{P}os(t)$ such that $\rho(\alpha) =_{\mathcal{A}} \rho(\beta)$ and $\rho(\beta) =_{\mathcal{A}} \rho(\gamma)$. Let also imagine that, say,

$$T(\alpha) = \{x, a, b\},\ T(\beta) = \{x, y\},\ T(\gamma) = \{y, f, g\}.$$

Let us note that this choice is coherent with our definition of a signature-TAGED and the equality constraints. Coming up with a concrete TAGED where this case happens is left as a recreational exercise to the reader. Let us sum it up: we need to have $t|_\alpha = t|_\beta = t|_\gamma$, so we need *at minima* to choose a symbol $\sigma \in \Sigma$ such that $t(\alpha) = t(\beta) = t(\gamma) = \sigma$. We have $t \in \mathfrak{Sxp}(T)$, so that means that $\sigma \in T(\alpha) \cap T(\beta) \cap T(\gamma) = \varnothing$. And so we have found a counter-example to our conjecture. □

Corollary 74 (Signature-TAGED as under-approximation)**.** *The converse of corollary 72*$_{[p110]}$ *is false.*

Proof. See above. □

This observation is something of a setback for us, as it means that we must either be content with the half-relationship we have between TAGEDs and signature TAGEDs, or change the definition of $\cong^s$ to something stronger. Considering that a full implementation of signature-TAGEDs has some overhead compared to that of plain TAGEDs, if only because we are dealing with sets of sets instead of just plain sets, and that implementing emptiness algorithms on vanilla TAGEDs was difficult and error-prone enough already, we decided not to continue with the development of signature-TAGEDs[(c)]. This is compounded with the fact that, as stated in the introduction, this optimisation *may* be of good use with *some* real-life TAGEDs, but would probably not shine with general random automata unless they were specifically designed with that in mind. So all in all this would have added overhead and complexified everything for a very modest measurable return on investment. All this considered, and rather than abandoning the idea purely and simply – which would not be satisfactory either since the basic idea is, after all, perfectly legitimate – we opted for a middle-ground solution, described in the next section.

13.2 As an over-approximation

The guiding idea here will be to find another approach to the problem, which

- ◇ keeps most of the advantages of the previous approach
- ◇ and works with the standard data structures, *ie.* TAGEDs.

We will achieve that by merging some well-chosen symbols.

Definition 75 (Signature-similarity/equivalence)**.** Let $\mathcal{A}$ be a TAGED. Then we define the *signature-similarity relation* on Σ, denoted $\sim$, as follows:

$$\forall f,g \in \Sigma : \; f \sim g \iff \exists (p_1,\dots,p_n,q) \in Q^{n+1} \; st. \; \begin{cases} f(p_1,\dots,p_n) \to q \in \Delta \\ g(p_1,\dots,p_n) \to q \in \Delta \end{cases}.$$

This relation is trivially symmetric and reflexive. The *signature-equivalence relation* on Σ, denoted $\cong^s$ is defined as the transitive closure of $\sim$. It is an equivalence relation.

(c) The necessary data structures and some algorithms using them *were* implement though.

The same remark as in the previous section applies: symbols group themselves by arity naturally:

Lemma 76 (Conservation of arity)**.** *Let $\mathcal{A}$ be a* TAGED*. Then the following holds:*

$$\forall f, g \in \Sigma : \; f \cong^s g \implies arity(f) = arity(g).$$

We call arity-preserving *a relation which satisfies this property.*

Proof. By induction over the length k of the smallest chain $f = \sigma_1 \sim \ldots \sim \sigma_k = g$:

- ◇ (base case $k = 2$): Suppose $f \cong^s g$, such that $f \sim g$. Then by definition of $\sim$ there exist $f(p_1, \ldots, p_n) \to q \in \Delta$ and $g(p_1, \ldots, p_n) \to q \in \Delta$, and it is clear – this time by definition of a tree automaton's transition rules – that $f, g \in \Sigma_n$, so we have $n = arity(f) = arity(g)$.
- ◇ (inductive case): Let $f \cong^s g$, such that the smallest chain $f = \sigma_1 \sim \ldots \sim \sigma_k = g$ is of length $k > 2$. Then we have by transitivity $f = \sigma_1 \cong^s \sigma_{k-1} \sim \sigma_k = g$, where smallest chain between f and σ_{k-1} is of length $k - 1$. Thus by induction hypothesis, $arity(f) = arity(\sigma_{k-1})$, and since we have $\sigma_{k-1} \sim \sigma_k = g$, it follows – by the same argument as in the base case – that $arity(\sigma_{k-1}) = arity(g)$. Thus $arity(f) = arity(g)$

Thus the result is proved by induction. □

Proof. (terser version). Using definition 81[p115]: We have clearly $(\cong^s) \subseteq (=^a)$; it follows that $(\cong^s)^* \subseteq (=^a)^* = (=^a)$. □

Definition 77 (Signature-quotiented TAGED)**.** Let $\mathcal{A} = (\Sigma, Q, F, \Delta, =_{\mathcal{A}}, \neq_{\mathcal{A}})$ be a TAGED. Then its *signature-quotiented* ***TAGED***, or *signature-****TAGED*** for short, is the TAGED $\mathcal{A}^s = (\Sigma^s, Q, F, \Delta^s, =_{\mathcal{A}}, \neq_{\mathcal{A}})$, where

$$\Sigma^s \stackrel{\text{def}}{=} \Sigma/_{\cong^s} \quad \text{and} \quad \Delta^s \stackrel{\text{def}}{=} \left\{ [\sigma](p_1, \ldots, p_n) \to q \;\middle|\; \sigma(p_1, \ldots, p_n) \to q \in \Delta \right\}.$$

There the reader might object and say "Wait! This is not a genuine TAGED at all: the so-called symbols of Σ^s are in fact *sets* of symbols, and we have gone back to the same kind of definition which we had in the previous section". And the reader would be right, but this time we do not merely have nondescript sets of symbols, we have equivalence classes, and moreover those classes are homogeneous for the arity. Thus there is nothing presenting us from simply taking a *representative* for each class. With that in mind, we are simply *merging* some symbols together. We lose some information compared to the previous approach, but we can keep using the same machinery as for vanilla TAGEDs. Now, let us see that a signature-quotiented TAGED is an over-approximation of the original TAGED.

Definition 78 (Signature-quotiented term)**.** Let $\mathcal{A}$ be a positive TAGED and $\mathcal{A}^s$ be its signature-TAGED. We call *signature-quotiented term* of t and denote t^s the tree defined recursively by $a^s = [a]$, for $a \in \Sigma_0$, and $(f(u_1, \ldots, u_n))^s = [f]\,({u_1}^s, \ldots, {u_n}^s)$ for $f \in \Sigma_n, n \geqslant 1$.

We note immediately that by this definition, $\mathcal{P}os(t) = \mathcal{P}os(t^s)$ and for any $\alpha \in \mathcal{P}os(t)$ we have the equality ${t|_\alpha}^s = t^s|_\alpha$.

Theorem 79 (Signature over-approximation)**.** *Let $\mathcal{A}$ be a positive TAGED and $\mathcal{A}^s$ be its signature-TAGED. Then for each $t \in \mathcal{L}ng(\mathcal{A})$, we have $t^s \in \mathcal{L}ng(\mathcal{A}^s)$.*

Proof. Let $t \in \mathcal{L}\mathrm{ng}(\mathcal{A})$, accepted by a certain successful run ρ of $\mathcal{A}$ on t. Let us show first that t^s is compatible with the transition rules of $\mathcal{A}^s$. Since $t \in \mathcal{L}\mathrm{ng}(\mathcal{A})$, it is compatible with the transition rules of $\mathcal{A}$, that is to say for every $\alpha \in \mathcal{P}os(t)$, there exists a rule

$$t(\alpha)\,(\rho(\alpha.1), \ldots, \rho(\alpha.arity(t(\alpha)))) \to \rho(\alpha) \in \Delta.$$

And it follows by definition that there also exists

$$[t(\alpha)]\,(\rho(\alpha.1), \ldots, \rho(\alpha.arity(t(\alpha)))) \to \rho(\alpha) \in \Delta^s,$$

and therefore t^s is compatible with the transition rules of $\mathcal{A}^s$. Let us now show that t^s satisfies the equality constraints of $\mathcal{A}^s$. We have

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

By the observation above, we have then

$$t|_\alpha = t|_\beta \implies {t|_\alpha}^s = {t|_\beta}^s \implies t^s|_\alpha = t^s|_\beta\,.$$

Thus t^s is compatible with the equality constraints $=_{\mathcal{A}}$ of $\mathcal{A}^s$, which concludes the proof. □

And thus we have exactly the same over-approximation result as we had before with regard to emptiness:

Corollary 80 (Signature-TAGED as over-approximation)**.** *Let $\mathcal{A}$ be a positive TAGED and $\mathcal{A}^s$ its signature-TAGED. Then $\mathcal{L}ng(\mathcal{A}^s) = \varnothing \implies \mathcal{L}ng(\mathcal{A}) = \varnothing$.*

Let us note that, although $\mathcal{A}^s$ has potentially lost much information compared to $\mathcal{A}$, these losses are only of a "superficial nature", meaning that the structure of the underlying trees accepted by $\mathcal{A}$ and $\mathcal{A}^s$, respectively, are the same. This remains true in fact with any arity-preserving equivalence relation.

Speaking of such relations, the reader may have noticed that corollary 80 and its lemma remain true if we substitute $\cong^s$ by any other arity-preserving equivalence relation. Here are two other candidate equivalence relations, one coarser than $\cong^s$ and the other finer, which we could redefine $\mathcal{A}^s$ with:

Definition 81 (Same-arity and signature-identity relations)**.** We define the *same-arity* relation (denoted $=^a$) and the *signature-identity* relation (denoted $\equiv^s$), such that:

$$f =^a g \iff arity(f) = arity(g),$$

and, denoting $\texttt{sigs}(\sigma) \stackrel{\text{def}}{=} \left\{ (p_1, \ldots, p_n, q) \;\middle|\; \sigma(p_1, \ldots, p_n) \to q \in \Delta \right\}$,

$$f \equiv^s g \iff \texttt{sigs}(f) = \texttt{sigs}(g).$$

Both relations are trivially arity-preserving equivalence relations – and of course $=^a$ is the equivalence kernel of *arity*.

Conjecture 82 (Friendly quotient)**.** *Let $\mathcal{A}$ be a positive* `TAGED` *and $\mathcal{A}^s_{\equiv^s}$ its signature-*`TAGED`*, using $\equiv^s$ instead of $\cong^s$. Then $\mathcal{L}ng\left(\mathcal{A}^s_{\equiv^s}\right) = \varnothing \iff \mathcal{L}ng\left(\mathcal{A}\right) = \varnothing$.*

Proof. [TODO] Coming soon. □

Note that although $\equiv^s$ is very restrictive, there are cases when it is perfectly suitable, of which the q_{char} example of this chapter's introduction is an example.

To avoid potential confusion in the remainder of this work, the equivalence relation used to build a signature-`TAGED` will appear explicitly as a subscript: for instance, the signature-`TAGED` of the `TAGED` $\mathcal{A}$ with respect to the equivalence relation $\equiv^s$ will be denoted $\mathcal{A}^s_{\equiv^s}$. Unless of course there is no ambiguity.

13.3 Implementations

As previously mentioned , the first approach was quite cumbersome to work with, both in theory and in practice. Consequently, although the data structures have been coded, not much have been done with them. As for the second approach, the code below presents an implementation (of signature-equivalence) which is both simple and efficient:

```
1  module DS = DisjointSet
2
3  let compress A =
4    let classes = DS.create 32 and sigt = H.create 127 in
```

```
  Σ.iter (λσ→DS.add classes σ) A.Σ;
  Δ.iter (λ(f,[q1,...,qn],q) → H.add sigt ([q1,...,qn],q) f) A.Δ;
  L.iter (λs→ DS.U+ classes[[ H.find_all sigt s) (hkeys sigt);
  return { A with
    Δ = Δ.endo (λ(f,[q1,...,qn],q)→
          DS.find classes f, [q1,...,qn],q) A.Δ;
  }
```

Note that the algorithm is clearly linear.

Let us take an example to illustrate this chapter a bit: here we have a cleaned-up TAGED (the constraints have been omitted as they are irrelevant here) before and after signature-quotient:

```
TAGED 'example' [58] = {
  states = #6{q0, q1, q2, q3, q4, q5}
  final = #2{q1, q5}
  rules = #16{
    a1()->q0, a1()->q3, a3()->q2, a3()->q4, a4()->q0,
    a4()->q4, a5()->q2, a5()->q3, f1(q1)->q5, f5(q1)->q5,
    g1(q1, q5)->q5, g3(q0, q5)->q5, g5(q0, q5)->q5,
    g5(q1, q5)->q5, h3(q2, q3, q4)->q1, h4(q2, q3, q4)->q1
}}

Classes = #{<g5 g3 g1>; <h4 h3>; <a5 a4 a3 a1>; <f5 f1>}#

TAGED 'sig-quotient' [34] = {
  states = #6{q0, q1, q2, q3, q4, q5}
  final = #2{q1, q5}
  rules = #8{
    a4()->q0, a4()->q2, a4()->q3, a4()->q4,
    f5(q1)->q5, g5(q0, q5)->q5, g5(q1, q5)->q5,
    h4(q2, q3, q4)->q1
}}
```

On this example we have a size ratio before/after quotient of $^{58}/_{34} \approx 1.71$, which is about as much as we can hope for with random TAGEDs of the fourth generation. Although about 80% of those randomly generated automata are reduced in size by the operation, it is most often by a very small amount.

For this reason – and also because the second approach was only formalised at the very end on the internship – this implementation was not used in our tests for deciding emptiness.

13.4 Conclusion

In this chapter we have introduced a rather specialised way of merging symbols of a TAGED and reducing it to another, smaller TAGED.

We showed that, depending on the equivalence relation used to merge the symbols, the resulting "compressed" TAGED may be either an approximation of the original with respect to the emptiness decision problem – in the sense that emptiness of the approximation implies that of the original; or equivalent to the original (again, with respect to emptiness).

This method is not very efficient against fourth generation random TAGEDs but may be of interest against special TAGEDs possessing one of several states similar to the q_{char} of our running example.

Chapter 14

Parenting relations

In the previous two chapters we have seen various ways of reducing a TAGED and in so doing, deciding emptiness in some particular cases when all final states, for one reason or another, become unreachable. Thus we have so far only covered easy cases where the TAGED is visibly empty.

In this chapter we cover cases where it is as visibly non-empty. We have seen before that in the case of diagonal positive TAGEDs, deciding emptiness is no more difficult than testing reachability of the final states (*cf.* theorem 56[p92]); here we will attempt to do almost the same thing with TAGEDs which are not diagonal, but not far from being so.

Consider for example the following handwritten positive TAGED, for which neither cleanup nor brutal algorithm yield any result (well, the latter would *eventually* yield results, but we might not be alive enough to still care by then. . .)

```
TAGED 'Heam' [146] = {
  alphab = #5{a/0, b/0, c/0, d/0, f/2}
  states = #10{q, q1, q2, q3, q4, q5, q6, q7, q8, qf}
  final = #1{qf}
  rules = #39{a()->q, a()->q1, a()->q2, b()->q, b()->q1, b()->q2,
    c()->q, c()->q1, c()->q2, d()->q, d()->q1, d()->q2,
    f(q, q)->q, f(q, q)->q1, f(q, q)->q2, f(q, q1)->q3,
    f(q, q1)->q5, f(q, q2)->q4, f(q, q2)->q6, f(q, q3)->q3,
    f(q, q3)->q5, f(q, q4)->q4, f(q, q4)->q6, f(q, q6)->q8,
    f(q, q7)->q7, f(q, qf)->qf, f(q1, q)->q3, f(q1, q)->q5,
    f(q2, q)->q4, f(q2, q)->q6, f(q3, q)->q3, f(q3, q)->q5,
    f(q4, q)->q4, f(q4, q)->q6, f(q5, q)->q7, f(q6, q)->q8,
    f(q7, q8)->qf, f(q8, q7)->qf, f(qf, q)->qf
```

```
  }
  ==rel = #2{(q1,q2), (q2,q1)}
}
```

Let us play around with the rules of this TAGED, and more specifically let us discard the symbols for now and observe the states we need to build (or reach) before building another. Let us do that as though it was a diagonal TAGED, which it is not, of course, but if it *were* diagonal, we would only need to make *one* choice for each state. Doing this we see several ways of constructing a term accepted

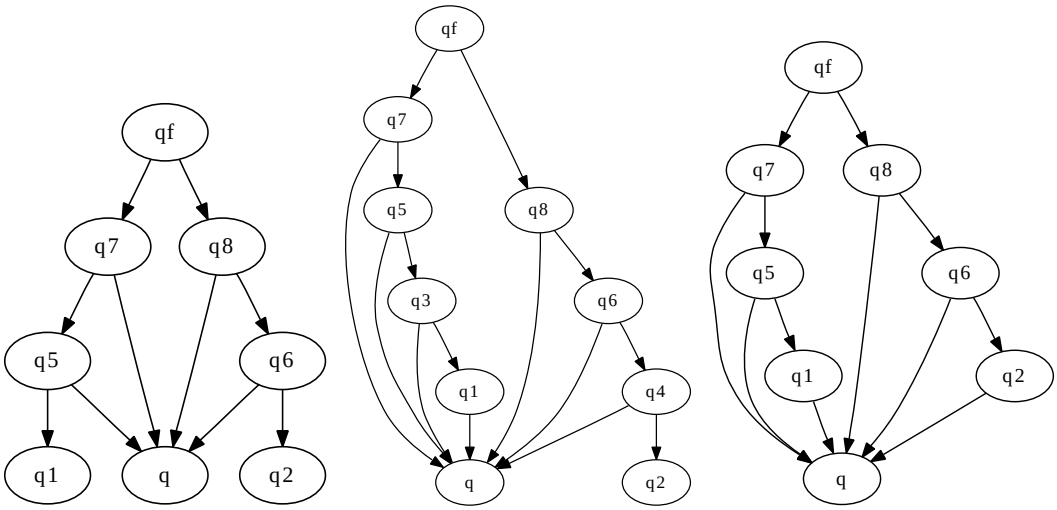


Figure 14.1 — Three of sixteen ways to build q_f

by this TAGED. For instance if we discard cycles – and we do, there is only one way of building q_f, and we first need to build both q_7 and q_8 for that. On the other hand for, say, q_5, we can use either a rule where we depend on q and q_1, or one where we depend on q and q_3, and exploring these two possibilities yields two different sets of dependency graphs. Figure 14.1[p119] shows three different possible dependency graphs which one might obtain, among a total of 16 possible[(a)].

Let us disregard for an instant the fact that we have a global equality constraint here. If this were a vanilla tree automaton, or even a diagonal TAGED, then we could take any such graph – noting that no state appears twice on any of them, start from the leaves, assign a term to each state as we go and thus build a term

[(a)] These figures were generated automatically by our prototype, using the dot tool for rendering.

and a run following these skeletons. It would work because we need only assign *one* term t_q to each state q, and we are free to make that choice for each state (once all their dependencies have made their own choices) quite independently from the choices made by other states.

Once we take the constraints into account, things change. Now q_1 and q_2 must synchronise their choices. The question is "is there a term which evaluates to both q_1 and q_2". If the answer is yes – and in that case it is, then we can affect that term to both states and move on. There are no other constraints, and thus no other trouble, and we build a term accepted by $\mathcal{A}$, which is therefore not empty. Problem solved.

What we would like to do is automata this kind of reasoning a bit. In essence, what we have done is to take a term in the intersection of the languages recognised by two tiny automata: the projection (*cf.* definition 52[p90]) of $\mathcal{A}$ on q_2 and its projection on q_2. If the intersection had turned out to be empty, then we could not have used this skeleton to build an accepted term.

The case we have shown was actually pretty trivial: in the first graph of figure 14.1[p119], q_1 and q_2 are leaves, so it might be argued that there is no need for projections, intersections etc. But what if we had been confronted to the *third* graph instead of the first? This time they are not leaves, but depend on q; so in order to respect the dependency graph we will take projections on q_1 (*resp.* q_2) *and* q. We then find that the intersection is non-empty and can conclude, again, that $\mathcal{A}$ is not empty. However, if we tried the second graph the intersection would be empty, and we would not be able to conclude as to $\mathcal{A}$. Fortunately since those kind of tests are very inexpensive, the idea is to test several – if not all – dependency graphs for a given `TAGED`.

In the next section we will formalise those ideas and extend them to much more general cases, where there can be many states below a state under constraints, and even states under constraints below other states under constraints (provided that certain restrictions are satisfied). In all cases, we will attempt to build a successful run (or romp, more exactly) by fixing as many states as we can in a dependency graph, and testing emptiness of the intersection of several languages. The idea is that this intersection will always be recognised by automata for which emptiness is computationally easy to decide, namely vanilla tree automata and diagonal positive `TAGED`s.

14.1 The theory

Prerequisites: This section relies on the notion of romp, introduced in chapter 15[p131] and on theorem 97[p135].

Let us give an intuitive notion of what we will be attempting here... Our ambition, given a TAGED $\mathcal{A}$, is to establish a kind of blueprint for building a term which it recognises. We shall attempt to find such a blueprint which does not rely too heavily on global equality constraints, and then we will test this blueprint to see whether a term can actually be built following it.

What we called "blueprint" informally will in fact be a Directed Acyclic Graph (DAG) where an edge denotes the fact that a state is built on top of another. There are in general several ways to build one state – say, if it has several rules in $\mathfrak{Rul}(q)$ with different signatures – and so there will be as many blueprints as the combination of all possible choices for all states. Of course, we will not be interested in *all* blueprints; we will simplify the problem by not accepting cycles – which is why we use a DAG – an we will not consider trivially spurious cases, for instance if two states p and q are such that $p =_{\mathcal{A}} q$, there will be no question that none of those can be built on top of the other one, and any blueprint which suggests that will be summarily rejected.

When we find a blueprint that is apparently correct, we will test whether it is simple enough for us to conclude. Roughly, we will need the equality constraints to be "at the bottom", and solve them by reducing the original problem to emptiness of an intersection of tree automata or diagonal TAGEDs, which is an easier problem.

So, if we succeed in finding a blueprint which is correct, simple enough, and for which the "sub-automata" we build are non-empty, we will have succeeded in building a term accepted by $\mathcal{A}$. On the other hand if we do not it does not necessarily mean that $\mathcal{A}$ is empty.

Note that there might be a great many possible blueprints; in practice we will not test all of them, but just a few. The more we test, the higher our chances of detecting non-emptiness

Now that we have some intuition of the problem, let us formalise all that. Instead of using DAGs directly, we will be using relations extending to strict orders by transitivity; which is pretty much the same thing. The so-called "parenting relation" introduced by the next definition is to be thought of as an edge in one of our DAGs.

Definition 83 (Parenting relation)**.** Let $\mathcal{A}$ be a positive TAGED, and $q_f \in F$ one of its final states. Then a relation on $Q \prec$ is a *parenting relation of* $\mathcal{A}$ *(for* q_f*)* if it satisfies the four following properties:

- **(q_f-domination)**:
  The ordered set $(\mathrm{dom}(\prec), \prec^+)$ has a greatest element, which is q_f.
- **(Transitionality)**:
  $\forall q \in \mathrm{dom}(\prec) : \exists r \in \mathfrak{Rul}(q)$ *st.* $\mathfrak{Ant}(r) = \left\{ p \;\middle|\; p \prec q \right\}$
- **(Strictness)**:
  $\prec^+$ is a strict partial order (*cf.* definition 12[p21]) on its domain.
- **(Aspuriousness)**:
  There are no two states $p, q \in \mathrm{dom}(\prec)$ such that $p \prec^+ q$ and $p =_{\mathcal{A}} q$.

Let us take all these properties one by one and see where they are coming from; this should help make the link with our informal explanations at the beginning of this section:

- *(q_f-domination)*: Here we are saying that this blueprint is for q_f, which is built over all other states.
- *(Transitionality)*: A blueprint would not be a blueprint if it did not follow elementary rules of construction. In that case, the transition rules. If we say that a state is built on top of those other states, then there must be a rule which allows it. Furthermore, we cannot mix rules haphazardly for a same state; one rule is chosen and that is all.
- *(Strictness)*: As mentioned before, we refuse cycles, both direct and indirect.
- *(Aspuriousness)*: We refuse any blueprint which is in obvious contradiction with the global equality constraints.

When we run into states which are involved in global equality constraints, say $p, q \in Q$ such that $p =_{\mathcal{A}} q$, we will need to test whether there exists a term which evaluates to both p and q, through two runs using exclusively the states which were allocated for p and q's constructions, respectively. This boils down to building an automaton which uses the appointed states for building q, another for p, and testing whether there exists a term accepted by both – *ie.* testing if the intersection of their accepted languages is empty. If we are lucky and manoeuvre right, then this will be much simpler than the original problem, since what we will actually need to do is decide emptiness of diagonal TAGEDs. So, let us define here what we meant by "automaton which uses the appointed states", exactly:

Definition 84 (Automaton under a state)**.** Let $\prec$ be a parenting relation of a TAGED $\mathcal{A}$, and $q \in \mathrm{dom}(\prec)$. We call *automaton under the state* q and denote $\mathfrak{Udr}(q, \prec)$, or simply $\mathfrak{Udr}(q)$, the automaton

$$\mathfrak{Udr}(q, \prec) = \mathfrak{Prj}\Big(\mathfrak{Rst}\Big(\mathcal{A}, \{\, p \mid p \preccurlyeq^{+} q \,\}\Big), q\Big).$$

Note that we deliberately used the vague term "automaton" here, as it can in fact turn out to be a vanilla tree automaton, a diagonal TAGED, a positive TAGED etc, depending on $\mathcal{A}$, q and $\prec$.

In all cases, how do we decide emptiness of the intersection of several languages defined by TAGEDs? We need to introduce the notion of *product* of tree automata, extended to TAGEDs. (The following definition and theorem are drawn from existing literature).

Definition 85 (Product of positive TAGEDs [Fil08])**.** Let

$$\mathcal{A} = (\Sigma, Q_A, F_A, \Delta_A, =_{\mathcal{A}}) \quad \text{and} \quad \mathcal{B} = (\Sigma, Q_B, F_B, \Delta_B, =_{\mathcal{B}})$$

be positive TAGEDs. We assume without loss of generality that $Q_A \cap Q_B = \varnothing$. Then we define the *product* of $\mathcal{A}$ and $\mathcal{B}$, denoted $\mathcal{A} \otimes \mathcal{B}$ to be the TAGED:

$$\mathcal{A} \otimes \mathcal{B} \stackrel{\text{def}}{=} (\Sigma,\ Q_A \times Q_B,\ F_A \times F_B,\ \Delta_A \otimes \Delta_B,\ =_{\mathcal{A}} \otimes =_{\mathcal{B}}),$$

where

$$\Delta_A \otimes \Delta_B \stackrel{\text{def}}{=} \left\{ f\Big((p_1^A, p_1^B), \ldots, (p_n^A, p_n^B)\Big) \to (q^A, q^B) \;\middle|\; \begin{array}{c} f(p_1^A, \ldots, p_n^A) \to q^A \in \Delta_A \\ f(p_1^B, \ldots, p_n^B) \to q^B \in \Delta_B \end{array} \right\}$$

and

$$(=_{\mathcal{A}} \otimes =_{\mathcal{B}}) \stackrel{\text{def}}{=} \left\{ \Big((p^A, p^B), (q^A, q^B)\Big) \;\middle|\; p^A =_{\mathcal{A}} q^A \text{ or } p^B =_{\mathcal{B}} q^B \right\}$$

Theorem 86 (Product of positive TAGEDs *cf.* [Fil08])**.** *Let $\mathcal{A}$ and $\mathcal{B}$ be two positive TAGEDs. Then $\mathcal{L}ng(\mathcal{A} \otimes \mathcal{B}) = \mathcal{L}ng(\mathcal{A}) \cap \mathcal{L}ng(\mathcal{B})$.*

Proof. See [CDG+07, page 30,31] for vanilla tree automata, and [Fil08, section 5.3, page 92] for the extension to TAGEDs and proof. □

Note that the product of positive TAGEDs does not appear to be a commutative operation: $\mathcal{A} \otimes \mathcal{B}$ and $\mathcal{B} \otimes \mathcal{A}$ are different automata, with different states. However we still have $\mathcal{L}ng(\mathcal{A} \otimes \mathcal{B}) = \mathcal{L}ng(\mathcal{B} \otimes \mathcal{A}) = \mathcal{L}ng(\mathcal{A}) \cap \mathcal{L}ng(\mathcal{B})$. So, by shameless abuse of notation we will take $\bigotimes_{\mathcal{A} \in S} \mathcal{A}$ to mean *any one of* the

possible automata resulting from taking the product of all automata in S in every possible order. What matters to us is that we have the property:

$$\mathcal{L}\mathrm{ng}\left(\bigotimes_{\mathcal{A}\in S}\mathcal{A}\right) = \bigcap_{\mathcal{A}\in S}\mathcal{L}\mathrm{ng}\left(\mathcal{A}\right).$$

Before closing the parenthesis on the notion of product of tree automata, let us confirm our intuition that the product of diagonal TAGEDs is still a diagonal TAGED itself.

Theorem 87 (Closure of diagonal positive TAGEDs under product)**.** *Let $\mathcal{A}$ and $\mathcal{B}$ be two diagonal positive TAGEDs. Then $\mathcal{A}\otimes\mathcal{B}$ is a diagonal positive TAGED.*

Proof. We have by theorem 86[p123]

$$(=_{\mathcal{A}}\otimes=_{\mathcal{B}}) = \left\{\left((p^A,p^B),(q^A,q^B)\right)\;\middle|\; p^A =_{\mathcal{A}} q^A \text{ or } p^B =_{\mathcal{B}} q^B\right\}.$$

But since both $=_{\mathcal{A}}$ and $=_{\mathcal{B}}$ only contain diagonal constraints, *ie.* constraints of the form $p =_{\mathcal{A}} p$, we have

$$(=_{\mathcal{A}}\otimes=_{\mathcal{B}}) = \left\{\left((p^A,p^B),(p^A,p^B)\right)\;\middle|\; p^A =_{\mathcal{A}} p^A \text{ or } p^B =_{\mathcal{B}} p^B\right\},$$

and therefore $=_{\mathcal{A}}\otimes=_{\mathcal{B}}$ contains only diagonal constraints: $\mathcal{A}\otimes\mathcal{B}$ is a diagonal positive TAGED. □

We now have all the tools we need to define which blueprints we will consider good, that is to say, which ones enable us to actually build an accepted term.

Definition 88 (Fruitful parenting relation)**.** Let $\prec$ be a parenting relation of the positive TAGED $\mathcal{A}$, and $(\equiv_{\mathcal{A}}) \stackrel{\text{def}}{=} (=_{\mathcal{A}}\cap\mathrm{dom}(\prec)^2)^*$ We say that $\prec$ is *fruitful* if

$$\forall\,[q]\in\mathrm{dom}(\prec)/_{\equiv_{\mathcal{A}}}, \mathrm{Card}\left([q]\right)>1: \quad \bigcap_{q\in[q]}\mathcal{L}\mathrm{ng}\left(\mathfrak{Udr}(q,\prec)\right)\neq\varnothing.$$

Theorem 89 (Fruitful positive TAGEDs)**.** *Let $\mathcal{A}$ be a positive TAGED. If there exists a fruitful parenting relation $\prec$ for one of its final states q_f, then it is non-empty.*

Proof. We let $(\equiv_{\mathcal{A}}) \stackrel{\text{def}}{=} (=_{\mathcal{A}}\cap\mathrm{dom}(\prec)^2)^*$, and take the notation $[q]$ to mean $[q]_{\equiv_{\mathcal{A}}}$. Let us build a romp ϱ_{q_f}, thus:

1. If the state $q \in \mathrm{dom}(\prec)$ is such that $\mathrm{Card}\,([q]) > 1$. Let us denote $\{p_1, \ldots, p_n\} = [q]$. By definition of a fruitful run we have $\bigcap_{q \in [q]} \mathcal{L}\mathrm{ng}\,(\mathfrak{Udr}(q, \prec)) \neq \varnothing$, so there exists ϱ, a successful romp of $\bigotimes_{q \in [q]} \mathfrak{Udr}(q, \prec)$, and we have – re-ordering if necessary – $\pi_Q(\varrho)(\varepsilon) = (p_1, \ldots, p_n)$. Let us denote ϱ_k the k-th component of ϱ for the states, while the symbol remains unchanged. For each $k \in [\![1, n]\!]$, we affect $\varrho_{p_k} := \varrho_k$. Of course we have by construction $\pi_Q\Big(\varrho_{p_k}\Big) = p_k$, and as it is a romp of a restriction of $\mathcal{A}$, it is also a raw romp of $\mathcal{A}$. Note that by construction $\pi_\Sigma\Big(\varrho_{p_k}\Big) = \pi_\Sigma\Big(\varrho_{p'_k}\Big)$, for all $k \neq k' \in [\![1, n]\!]$.

2. If the state $q \in \mathrm{dom}(\prec)$ is such that $\mathrm{Card}\,([q]) = 1$. We ignore q if ϱ_q was already defined at a previous step; if it was not, then by transitionality of $\prec$, there exists $\sigma(p_1, \ldots, p_n) \to q \in \Delta$ *st.* $\{p_1, \ldots, p_n\} = \Big\{p \;\Big|\; p \prec q\Big\}$. For each $k \in [\![1, n]\!]$, we compute ϱ_{p_k} if not already computed, and affect $\varrho_q := \langle \sigma, q \rangle \Big(\varrho_{p_1}, \ldots, \varrho_{p_n}\Big)$, which is clearly a raw romp of $\mathcal{A}$.

3. If the state $q \in \mathrm{dom}(\prec)$ is such that $q \notin \mathrm{dom}(\equiv_{\mathcal{A}})$, we do exactly as in the case $\mathrm{Card}\,([q]) = 1$.

We start the algorithm by beginning the computation of ϱ_{q_f}. Since $\prec$ is dominated by q_f, ϱ_q will be generated for all states $q \in \mathrm{dom}(\prec)$. In the end, we get a raw romp of $\mathcal{A}$ ϱ_{q_f}, rooted in q_f. Let us show that this romp is successful. By theorem 97[p135], it suffices to show that

$$\forall\, [q] \in \mathrm{dom}(\prec)/_{\equiv_{\mathcal{A}}} :\ \mathrm{Card}\left(\left\{\, \pi_\Sigma(\varrho)\big|_\alpha \;\middle|\; \pi_Q(\varrho)(\alpha) \in [q] \,\right\}\right) = 1.$$

For states q such that $\mathrm{Card}\,([q]) = 1$, there is trivially only one term in the set, and so the cardinal can hardly be anything but 1. And for classes with more than one element, we have already noted that they share the same term by construction. Thus ϱ_{q_f} is compatible with $=_{\mathcal{A}}$, and we have found a successful romp of $\mathcal{A}$. It follows that $\mathcal{A}$ is non-empty, which concludes the proof. □

All this is well and good, but so far nothing tells us that deciding emptiness for those products of automata is going to be easy. We now justifies what we have said informally before, *ie.* that we can arrange to only deal with diagonal positive TAGEDs. We are hoping to find a DAG (or, in this formal model, a parenting relation) which arranges the states in such a way as to let us reduce the problem to diagonal automata. For this, we will need to reason on the relative positions of states involved in global constraints in our blueprint. For convenience, we will reason on a relation dealing with such states only:

Definition 90 (Parenting core)**.** Let $\prec$ be a parenting relation of a TAGED $\mathcal{A}$. We call *core* of $\prec$ – and often denote $\preccurlyeq^+$ – the relation

$$(\preccurlyeq^+) \stackrel{\text{def}}{=} (\prec^+) \cap \mathrm{dom}(=_{\mathcal{A}})^2.$$

Now, we will give the formal characterisation of the good blueprints, *ie.* those for which the sub-automata we create are diagonal. We shall justify those conditions immediately after.

Definition 91 (Flat and pseudo-flat parenting relations)**.** Let $\prec$ be a parenting relation of a TAGED $\mathcal{A}$. It is called *flat* if its core $\preccurlyeq^+$ is empty, and *pseudo-flat* if

$$\forall p, q, p' \in Q : \quad p \preccurlyeq^+ q \implies (p =_{\mathcal{A}} p' \implies p = p').$$

We said before that we would need to compute the intersection of languages accepted by TAGEDs; let us see how:

Theorem 92 (flat and pseudo-flat tests)**.** *Under the conditions and notations of theorem 89*[p124]*, let* $[q]$ *such that* $\mathrm{Card}([q]) > 1$*, and let*

$$\mathcal{U} = \bigotimes_{q \in [q]} \mathfrak{Udr}(q, \prec).$$

Then the following statements hold:

1. *If* $\prec$ *is flat then* $\mathcal{U}$ *is a vanilla tree automaton or a diagonal positive* TAGED *with only one constraint, on its sole final state.*

2. *If* $\prec$ *is pseudo-flat then* $\mathcal{U}$ *is a diagonal* TAGED*.*

Proof. The proof is in two parts:

1. Let $\prec$ be pseudo-flat. To prove that $\mathcal{U}$ is diagonal, it suffices to prove that every member of the product is diagonal (*cf.* theorem 87[p124]). Let $q \in [q]$, and let us denote $\mathcal{B} = \mathfrak{Udr}(q, \prec)$. The sole final state of $\mathcal{B}$ is q, and we *may* have $q =_{\mathcal{B}} q$: we know q is involved in non-diagonal constraints in $\mathcal{A}$, but we don't know whether there are diagonal ones on top of this. As it turns out, we do not need to know. Let us show that there can be no other constraint. Suppose there was; then it would be either of the form $p =_{\mathcal{B}} q$ or $p =_{\mathcal{B}} p'$, where $p, p' \in \mathrm{dom}(=_{\mathcal{B}})$ are such that $p \neq q$ and $p' \neq q$. By definition of $\mathcal{B}$ we have $p, p' \prec^+ q$. So the constraint $p =_{\mathcal{B}} q$ is illegal by aspuriousness of $\prec$. That leaves us with $p =_{\mathcal{B}} p'$. Recall that $|Q| > 1$; this

means that $q \in \mathrm{dom}(=_{\mathcal{A}})$. Therefore we have $p, p' \lessdot^+ q$ and, if we recall the definition (*cf.* 91) of a pseudo-flat relation:

$$p \lessdot^+ q \implies (p =_{\mathcal{A}} p' \implies p = p').$$

So our constraint is of the form $p =_{\mathcal{B}} p$. Therefore $\mathcal{B}$ is diagonal.

2. If $\prec$ is flat, then we are in an even more stringent case as above (trivially if a relation is flat it is also pseudo-flat). So by the same argument we may have $q =_{\mathcal{B}} q$. But this time, we cannot have any constraint of the form $p =_{\mathcal{B}} p'$ at all since then we would have $p \lessdot^+ q$; but since $\prec$ is flat, $\lessdot^+$ is empty. Thus $\mathcal{B}$ is either a vanilla tree automaton, or $(=_{\mathcal{B}}) = \{(q, q)\}$. □

Of course, some `TAGED`s will not admit of any suitable parenting relation. Let us see a quick test which will detect some of such cases before too much time has been invested in the computation of parenting relations.

Conjecture 93 (Test). *Let $\mathcal{A}$ be a `TAGED`. Let us call* parenting need *and denote $\prec_n$ the relation*

$$p \prec_n q \iff p \in \mathfrak{Need}(q).$$

and $\lessdot^+$ its core – as if it was a parenting relation. Then if for all $q_f \in F$, $(\lessdot^+) \cap \left\{ p \;\middle|\; p \prec_n q_f \right\}^2$ is not pseudo-flat, there is no pseudo-flat parenting relation for $\mathcal{A}$.

14.2 Implementation and experiments

Our prototype implements an earlier version of the approach – the complete formalisation was written afterwards. Here is the algorithm we use to generate and test – or print, or anything else – all or some possible parenting relations (called dependency graphs in the code).

```
1
2  (* Module: set of possible antecedents of a state (set of sets of states) *)
3  module 2^Q = XSet (struct type t = Q.t end)
4
   (* Return set of possible antecedent bags (ie. sets of states q depends on);
   bags with immediate cycles (q in bag) are not returned. More formally:
5  ante_bags A q = { Ant(r) | r ∈ Rul(q) and q ∉ Ant(r) }. *)
6  let ante_bags A q =
7    let Δ = Rul A q in let antes = Δ.map_L Ant Δ in
8    2^Q.filter (λbag→¬[[ Q.∈ q bag) (2^Q.of_list antes)
```

```
9
10  (* Return a new graph with bag taken into account *)
11  let bag_graph g q bag =
12    let [q_1,...,q_n] = Q.elements bag and g' = ref g in
13    L.iter (λp→ g' := G.add_edge !g' q p) [q_1,...,q_n];
14    return !g'
15
16  (* Return the transitive closure of a graph *)
17  let trans_cl g = Ops.transitive_closure g
18
19  (* Returns true if there are no loops in the graph *)
20  let no_loop g =
21    G.fold_vertex (λv b → b ∧ ¬[[ G.mem_edge g v v) g true
22
23  (* Ensures aspuriousness of a transitive graph *)
24  let no_spurious A g =
25    Q^2.∀ (λ(p,q)→ p ⩾ q ∨ ¬[[ G.mem_edge g p q) A.=_A
26
27  (* Iterate a function f on all (or at most cap) valid dependency graphs of a
    state q *)
28  let dependencies ?(cap=+∞) A f q = let c = ref 1 in
29    let rec deps A fixed graph = λ
30      | q :: qs → 2^Q.iteri (λk bag →
31        if !c < cap ∨ k < 1 then begin
32          if k > 0 then incr c;
33          let bagged_graph = bag_graph graph q bag in
34          let trans = trans_cl bagged_graph in
35          if no_loop trans ∧ no_spurious A trans then
36            deps A (Q.add q fixed) bagged_graph
37              (qs ∪ Q.elements (Q.diff bag fixed));
38        end) (ante_bags A q);
39      | ∅ → f graph
40    in deps A Q.∅ G.∅ [q]
```

The testing function which we have implemented should correspond to the flat case (*cf.* definition 91[p126]) , though again, as this was coded before the formalisation was done, the implementation is much more complex and labyrinthine that it would need to be, and there might be subtle behavioural discrepancies. The results could probably be better with a simpler implementation covering the more general pseudo-flat cases. With that in mind, table 14.1[p130] shows some

test results for this approach. The results presented in this table are to be understood in terms of the outcomes of the strategy outlined in chapter 16[p139], with fourth generation random TAGEDs (*cf.* section 8.4[p71]) equipped with the third model of random constraints (*cf.* section 9.3[p81]). Each line is averaged from the results of 250 generated automata. Each instance of the brutal algorithm has a timeout of 0.5 second, and the number of tested parenting relations was capped to 25 before the blank line, and 1 afterwards. The last column indicates the net percentage of cases which were decided (*ie.* fell into the *Something* column) thanks to parenting relations and would otherwise have been failures.

14.3 Conclusion

In this chapter we have introduced a way of (positively) deciding emptiness in a range of special cases. Tests (limited to a weak version of the approach) have shown this method to be useful in circumstances where the brutal algorithm was too inefficient to be of any use at all.

Height	Run	Something	Nothing	Failure	< results
6	0.4%	69.6%	28.8%	1.2%	2.8%
9	0.4%	69.2%	25.6%	4.8%	6.4%
12	0.0%	55.6%	36.4%	8.0%	9.2%
15	0.0%	61.2%	26.4%	12.4%	7.6%
18	0.0%	53.2%	30.0%	16.8%	6.4%
21	0.0%	50.8%	30.0%	19.2%	8.8%
24	0.0%	46.8%	35.6%	17.6%	7.2%
27	0.0%	49.2%	28.8%	22.0%	8.8%
27	0.0%	45.6%	31.2%	23.2%	5.6%
30	0.0%	45.2%	31.2%	23.6%	6.8%
31	0.0%	50.8%	25.2%	24.0%	6.0%
34	0.0%	50.8%	26.8%	22.4%	6.4%
37	0.0%	43.6%	26.8%	29.6%	7.2%

Table 14.1 — Tests of “parenting relation” approach

Chapter 15

A brutal algorithm

So far, we have only seen inexpensive tactics which *may* enable us to decide emptiness. However the emptiness decision problem is *EXPTIME*-complete and there will always be TAGEDs for which none of theses tactics – or for that matter, any such inexpensive tactic anyone might come up with in the future – will work. In this chapter we present a "brutal" algorithm which will decide in all cases – provided of course that we are willing to wait for it to terminate.

15.1 Algorithms and implementation

As its name indicates, the algorithm which we are about to present is far from subtle. It is in fact a brute force algorithm, with a few twists. In the context of deciding TAGED emptiness, a real, pure and unalloyed brute-force approach would be that of figure 15.1[p131]. Although we are not going to proceed *exactly*

Data: A positive TAGED $\mathcal{A}$
Result: Nothing or Run ρ,t
begin
  foreach $t \in \mathcal{L}ng\,(\mathfrak{ta}\,(\mathcal{A}))$ st. $\mathfrak{Hei}(t) \leqslant \mathcal{A}.Q$ **do**
    if t *is accepted by* $\mathcal{A}$ *through run* ρ **then return** Run ρ,t ;
  endfch
  return Nothing;
end

Figure 15.1 — Pure brute-force emptiness

like that – because that would be so inefficient as to become absolutely useless,

it is still the general idea of the brutal algorithm. Note that algorithm 15.1[p131] terminates, since there are only finitely many terms $t \in \mathcal{L}\mathrm{ng}\left(\mathfrak{ta}\left(\mathcal{A}\right)\right)$ such that $\mathfrak{Hei}(t) \leqslant |Q|$, and gives the expected result, thanks to the pumping lemma for positive TAGEDs [FTT08b, Lemma 2(Appendix)].

Still, generating and testing *all* possible terms would be quite a titanesque endeavour for all but the tiniest of TAGEDs. Not only are there a *considerable* number of terms to test, but each test is in essence an instance of the membership problem, which is itself *NP*-complete. The idea here will be to cut all that down somewhat with some simple observations; it will of course not change the theoretical complexity of the algorithm, but it will make it practical to actually run it on slightly less microscopic instances of the problem.

Since the implementation of the brutal algorithm is quite lengthy, and the algorithm itself is a bit complex taken as a whole, we will give here neither the code nor even a full outline of it; instead we will iteratively refine different aspects of algorithm 15.1[p131], independently of one another.

Let us start with the generation of the terms of the underlying tree automaton; how do we proceed? First off, it seems impractical[(a)] to actually generate *terms*, because immediately afterwards we will want determine the existence of a run, and then test this run for compatibility with the global equality constraints. This is counterproductive, because we will create a term using the rules (so we know what the run would be), then discard this information (we keep the term, not the run), and then painfully recreate all possible runs to finally test them for compatibility. Rather than following this Rube Goldberg contraption, we will generate terms and runs at the same time; more specifically, we will generate "hybrids". Recall that a term t is a mapping from $\mathcal{P}os(t)$ to Σ – satisfying some properties mentioned in section 2.1.1[p11], and that a run ρ on this same term t is a mapping from $\mathcal{P}os(t)$ to Q. So, for convenience, we will manipulate the two at the same time: we will build "runs" which will be mappings from $\mathcal{P}os(t)$ to $\Sigma \times Q$. Of course those are a different kind of objects, so to allay confusion, we will call them "romps" instead of "runs", and denote them ϱ instead of ρ. When representing them explicitly, we will denote $\langle f, q\rangle(\varrho_1, \ldots, \varrho_n)$ the romp which combines a tree $f(t_1, \ldots, t_n)$ and a run $q(\rho_1, \ldots, \rho_n)$. We can untangle term and run through the two functions

$$\pi_\Sigma() : \langle f, q\rangle(\varrho_1, \ldots, \varrho_n) \mapsto f\left(\pi_\Sigma(\varrho_1), \ldots, \pi_\Sigma(\varrho_n)\right)$$
$$\text{and } \pi_Q() : \langle f, q\rangle(\varrho_1, \ldots, \varrho_n) \mapsto q\left(\pi_Q(\varrho_1), \ldots, \pi_Q(\varrho_n)\right).$$

[(a)] As a matter of fact it *is* impractical. I know. I have tried.

So, the question now becomes "how are we going to generate those romps?". It seems logical to start with the simplest possible romps (leaf-romps) and iteratively build all romps of height n on top of the already-built romps of height strictly less than n. The generation would then be something like algorithm 15.2[p133]. Of course, the algorithm such as it is written here is very inefficient,

Data: A tree automaton $\mathcal{A}$
Result: All romps ϱ such that $\mathfrak{Hei}(\varrho) \leqslant |Q|$
begin
  Romps $\leftarrow \left\{ \langle a, q\rangle () \;\middle|\; a \to q \in \Delta \right\}$;
  for $h = 1$ *to* $|Q|$ **do**
    add $\left\{ \langle f, q\rangle (\varrho_1, \ldots, \varrho_n) \;\middle|\; \begin{array}{l} f(p_1, \ldots, p_n) \to q \in \Delta \\ \forall k \in [\![1, n]\!] : \left\{ \begin{array}{l} \varrho_k \in \text{Romps} \\ \pi_Q(\varrho_k)(\varepsilon) = p_k \end{array} \right. \end{array} \right\}$ to Romps;
  endfor
  return Romps;
end

Figure 15.2 — Building romps

as the same terms will be built over and over again. In mathematical set notation it does not matter, but for a computer program's data structures, it probably does... There are several way of mitigating this problem, for instance storing romps of height $n-1$ separately and making sure that at least one ϱ_k is selected among them, thus making sure that each iteration generates romps of height n.

We will not detail this part of the algorithm any further, since such a discussion concerns vanilla tree automata in general, and is is no way specific to TAGEDs. We will now see how the introduction of the TAGED's global equality constraints helps us cut down on the numbers of romps which we need to store. The following definition should be self-evident, but let us write it anyway:

Definition 94 (Romp on an automaton)**.** Let $\mathcal{A}$ be a positive TAGED (*resp.* a tree automaton). We call the romp ϱ a "romp of $\mathcal{A}$" if $\pi_Q(\varrho)$ is a run of $\mathcal{A}$ on $\pi_\Sigma(\varrho)$, compatible with the global equality constraints (*resp.* a run of $\mathcal{A}$). We call *raw romp* of a TAGED $\mathcal{A}$ a romp of $\mathfrak{ta}(\mathcal{A})$. A romp ϱ is *successful* or *accepting* if $\pi_Q(\varrho)(\varepsilon) \in F$.

This definition being given, the following theorem should be just as obvious:

Theorem 95. *Let $\mathcal{A}$ be a positive TAGED. If ϱ is a romp of $\mathcal{A}$ (compatible with $=_{\mathcal{A}}$), then for every $\alpha \in \mathcal{P}os(\varrho)$, $\varrho|_\alpha$ is compatible with $=_{\mathcal{A}}$.*

Proof. We have $\left\{ \alpha.\beta \;\middle|\; \beta \in \mathcal{P}os(\varrho|_\alpha) \right\} \subseteq \mathcal{P}os(\varrho)$. Thus, by definition of compatibility with $=_{\mathcal{A}}$ (*cf.* definition 5[p17]), if ϱ is compatible, so is $\varrho|_\alpha$. □

What does this mean for us? Since in the end, we are only interested in romps which are compatible with $=_{\mathcal{A}}$, and since the above theorem tells us that any such romp is built exclusively on sub-romps which are themselves compatible with $=_{\mathcal{A}}$, it follows that we do not need to generate and store *every* possible romp of the underlying tree automaton, but only those which are compatible with $=_{\mathcal{A}}$; the others can be immediately discarded.

Of course, this assumes that we can test the compatibility of a romp with $=_{\mathcal{A}}$. This is not a computationally easy task, but we may be able to facilitate it somewhat by leveraging the assumption we have just made. Recall that we build new romps out of old ones, following the transition rules. Say we have $f(p_1, \dots, p_n) \to q \in \Delta$; we take n romps $\varrho_1, \dots, \varrho_n$ out of the ones we have already generated and combine them into $\varrho = \langle f, q \rangle (\varrho_1, \dots, \varrho_n)$. Then we test compatibility with $=_{\mathcal{A}}$. But consider that our new assumption is that every one of those old romps $\varrho_1, \dots, \varrho_n$ is already compatible with $=_{\mathcal{A}}$. There are still many things which can break compatibility for the new romp ϱ, but clearly not as many as in the general case.

If an incompatibility arises, it will necessarily stem either from a constraint of the form $q =_{\mathcal{A}} p_k$, for $p_k \in \mathrm{ran}\,(\pi_Q(\varrho_k))$ – clearly unsatisfiable as a tree cannot be structurally equal to one of its sub-trees – or a constraint of the form $p_k =_{\mathcal{A}} p_{k'}$, for $k \neq k'$, that is to say, a constraint between two different children. In that last case though one has to be mindful that there might be a hidden constraint $p_k =_{\mathcal{A}} q_k$ *in the same child*; to be sure to take that into account we will extend $=_{\mathcal{A}}$ to an equivalence relation[(b)] on $\mathrm{ran}\,(\pi_Q(\varrho))$. Let us write that down:

Conjecture 96. *Let ϱ be a raw romp of a positive* TAGED *$\mathcal{A}$ such that for all $k \in [\![1, arity(\varrho)]\!]$, $\varrho|_k$ is compatible with $=_{\mathcal{A}}$, and let us denote $\rho = \pi_Q(\varrho)$ and $t = \pi_\Sigma(\varrho)$. Let $T = \mathrm{ran}(\rho)$ and*

$$\widehat{T} = \{\, \rho(\varepsilon) \,\} \cup \left\{ q \in T \;\middle|\; \begin{array}{c} \exists k, k' \in [\![1, arity(t)]\!] : k \neq k', \exists \alpha, \beta \in \mathbb{N}^* : \\ \rho(k.\alpha) = p \wedge \rho(k'.\beta) = q \wedge p =_{\mathcal{A}} q \end{array} \right\},$$

Let $(\equiv_{\mathcal{A}}) \overset{def}{=} (=_{\mathcal{A}} \cap T^2)^*$. Then ϱ is compatible with $=_{\mathcal{A}}$ if and only if

$$\forall\, [q] \in {}^{T}\!/_{\equiv_{\mathcal{A}}} :\; [q] \cap \widehat{T} \neq \varnothing :\; \mathrm{Card}\left(\left\{ \, t|_\alpha \;\middle|\; \rho(\alpha) \in [q] \, \right\} \right) = 1.$$

[(b)] This annoying tendency of $=_{\mathcal{A}}$ to be almost, *but not quite*, an equivalence relation is a bottomless well of potential mistakes for the unwary.

We do not offer a full proof of this; we have given above a rationalisation of the fact that given our hypothesis that the direct children we are building a romp with are already compatible with the equality constraints, we need not test *every* constraint, but only those which put into play either the father and one of its offspring, or two different children. Constraints internal to any one of the children have already been taken care of in the past. In the conjecture the set $\widehat{T}$ is precisely the set of the states we need worry about; this takes care of the first big scary formula. As for the second, let us take a look at theorem 97[p135] below: it's pretty much the same. In the conjecture, we are simply saying that to ensure compatibility with $=_{\mathcal{A}}$, it is *sufficient* to test the states which are "relatives" of those states in $\widehat{T}$, that is to say, we test only the states which are directly or transitively in relation (via $=_{\mathcal{A}}$) with a state of $\widehat{T}$. We are taking the time to compute $\widehat{T}$ because computing many useless structural comparisons of subtrees would probably be much more expensive on large trees.

Theorem 97 (Characterisation of compatibility with $=_{\mathcal{A}}$)**.** *Let ϱ be a raw romp of a positive* TAGED *$\mathcal{A}$, and let us denote $\rho = \pi_Q(\varrho)$ and $t = \pi_\Sigma(\varrho)$. Let $T = \mathrm{ran}(\rho)$ and $(\equiv_{\mathcal{A}}) \overset{def}{=} (=_{\mathcal{A}} \cap T^2)^*$. Then ρ*[c] *is compatible with $=_{\mathcal{A}}$ if and only if*

$$\forall\, [q] \in {}^{T}\!/_{\equiv_{\mathcal{A}}} :\ \mathrm{Card}\left(\left\{\, t|_\alpha \;\middle|\; \rho(\alpha) \in [q] \,\right\}\right) = 1.$$

Proof. The proof is in two parts:

- $\Longleftarrow$: Suppose that ρ satisfies this condition (call it C). Then let $\alpha, \beta \in \mathcal{P}os(t)$ such that $\rho(\alpha) =_{\mathcal{A}} \rho(\beta)$. By definition we also have $\rho(\alpha) \equiv_{\mathcal{A}} \rho(\beta)$, and thus there exists $[q] \in T/\equiv_{\mathcal{A}}$ such that $\rho(\alpha), \rho(\beta) \in [q]$. By condition C we must have $t|_\alpha = t|_\beta$ – else the cardinal would be at least 2. Thus ρ is compatible with $=_{\mathcal{A}}$.

- $\Longrightarrow$ *(by contraposition)*: Suppose that ρ satisfies $\neg C$, that is to say

  $$\exists\, [q] \in {}^{T}\!/_{\equiv_{\mathcal{A}}} :\ \mathrm{Card}\left(\left\{\, t|_\alpha \;\middle|\; \rho(\alpha) \in [q] \,\right\}\right) > 1.$$

  (The cardinal could clearly not be zero: the equivalence classes cannot be empty…) Then this means that there exist $\alpha, \beta \in \mathcal{P}os(t)$ such that $\alpha \neq \beta$ and $\rho(\alpha), \rho(\beta) \in [q]$ and $t|_\alpha \neq t|_\beta$. We have $\rho(\alpha) \equiv_{\mathcal{A}} \rho(\beta)$ and thus either $\rho(\alpha) =_{\mathcal{A}} \rho(\beta)$ – and in that case $=_{\mathcal{A}}$ is violated – or there exist states $p_1, \ldots, p_n \in [q]$ such that $\rho(\alpha) =_{\mathcal{A}} p_1 \wedge p_1 =_{\mathcal{A}} p_2 \wedge p_2 \ldots =_{\mathcal{A}} p_n \wedge p_n =_{\mathcal{A}} \rho(\beta)$. Since all states $p_1, \ldots, p_n$ are in T, there exist $\gamma_1, \ldots, \gamma_n \in \mathcal{P}os(t)$ such that

[c] Or equivalently, ϱ.

for all $k \in [\![1, n]\!]$, $\rho(\gamma_k) = p_k$. Suppose for an instant that ρ is compatible with $=_{\mathcal{A}}$: then we have $t|_\alpha = t|_{\gamma_1}$ and $t|_{\gamma_1} = t|_{\gamma_2}$ and $\dots t|_{\gamma_n} = t|_\beta$, and by transitivity $t|_\alpha = t|_\beta$. This is in contradiction with $t|_\alpha \neq t|_\beta$, thus ρ is *not* compatible with $=_{\mathcal{A}}$. □

Let us see how we will compute $\widehat{T}$ in practice: the algorithm is given in figure 15.3[p136]. Now that we have $\widehat{T}$, let us compute the set $\widetilde{T}$ of all states which are

Data: A raw romp ϱ of a positive `TAGED` $\mathcal{A}$
Result: The set $\widehat{T}$ as in conjecture 96[p134]
begin
  $\widehat{T} \leftarrow \left\{ \pi_Q(\varrho)(\varepsilon) \right\}$;
  $n \leftarrow arity(\varrho)$;
  foreach $(p, q) \in (=_{\mathcal{A}})$ **do**
    if $p, q \notin \widehat{T}$ **then**
      for $k = 1$ *to* $n - 1$ **do**
        if $p \in \mathrm{ran}\left(\pi_Q(\varrho_k)\right)$ **then**
          for $l = k{+}1$ *to* n **do**
            if $p \in \mathrm{ran}\left(\pi_Q(\varrho_l)\right)$ **then** add p and q to $\widehat{T}$;
  return $\widehat{T}$;
end

Figure 15.3 — Building the set $\widehat{T}$ (*cf.* conjecture 96[p134])

in relation with $\widehat{T}$, either directly or by transitivity:

$$\widetilde{T} = \bigcup_{\substack{[q] \in T/\equiv_{\mathcal{A}} \\ [q] \cap \widehat{T} \neq \varnothing}} [q]\,,$$

and with this in hand we can proceed to decide compatibility as in algorithm 15.4[p137]: Note that in practice the equivalence classes of this algorithm can be replaced by representative states – through a disjoint-set data structure for instance; the exact values of the classes as sets are not needed anymore once one has computed $\widetilde{T}$.

We now have reasonably efficient ways of generating romps and of testing their compatibility with the global equality constraints. Let us expand on algorithm

Data: A raw romp ϱ of a positive TAGED $\mathcal{A}$, set $\widetilde{T}$
Result: true $\iff$ ϱ is $=_{\mathcal{A}}$-compatible
begin
  let h be a table "$[q] \mapsto$ *a set of terms*";
  foreach $\alpha \in \mathcal{P}os(\varrho)$ **do**
    if $\pi_Q(\varrho)(\alpha) \in \widetilde{T}$ **then**
      add $\left[\pi_Q(\varrho)(\alpha)\right] \mapsto \pi_\Sigma(\varrho)\big|_\alpha$ to h;
      if Card $(h([\pi_Q(\varrho)(\alpha)])) > 1$ **then**
        return false;
  return true;
end

Figure 15.4 — Deciding $=_{\mathcal{A}}$-compatibility of a romp

15.2[p133] to get an outline of the resulting global algorithm: Note that some obvious conditions which have already been discussed and/or given in algorithm 15.2[p133] have been omitted in algorithm 15.5[p138], for instance the condition that $\varrho_i \in$ Romps, for all $i \neq k$, etc.

15.2 Conclusion

In this chapter we have seen a systematic, brute force approach for deciding emptiness. Unlike all the other approaches which we have presented in this work, this algorithm will *eventually* give an answer in *all* cases, and not only in a range of special cases.

Nevertheless, in spite of the few tricks it uses to curtail the inefficiency of the brute-force technique, it is far from efficient enough to be used on its own. As such, it is meant to be used as a last resort only, when simpler, more economical *modus operandi* have failed.

```
Data: A positive TAGED 𝒜
Result: Nothing if 𝒜 is empty, Run ρ, t if run ρ accepts t
begin
    romps ← last ← { ⟨a, q⟩ () | a → q ∈ Δ };
    buff ← ∅;
    for h = 1 to |Q| do
        foreach f(p_1, ..., p_n) → q ∈ Δ do
            for k = 1 to n do
                add { ϱ = ⟨f, q⟩(ϱ_1, ..., ϱ_n) | ϱ_k ∈ last ∧ ϱ sat. =_𝒜 } to buff;
                if any of those ϱ is such that π_Q(ϱ)(ε) ∈ F then
                    immediately return Run π_Q(ϱ), π_Σ(ϱ);
        add buff to romps;
        last ← buff;
        buff ← ∅;
    return Nothing;
end
```

Figure 15.5 — Outline of the brutal algorithm

Chapter 16

Strategies and tactics

Deciding emptiness is a difficult problem – an *EXPTIME*-complete one – for which there does not exist any really efficient algorithm. We have tackled this problem in a somewhat military spirit, through strategy and tactics. So far, we have concentrated exclusively on various tactical aspects; in this chapter we will see the global strategy which will govern the application and composition of theses tactics. Our "battle" plays in three main phases, corresponding roughly to three main tactics or groups of tactics:

- ◇ Preliminary, inexpensive tactics to weed out the weak elements as soon as possible and reduce the size of the `TAGED`s before more expensive tactics come into play – just as a volley of arrows would open an epic medieval battle, before the bold and reckless charge of the heavy cavalry... It may or may not yield impressive results, but the cost of trying is quite negligible set against the humongous expenses of the whole battle. Our arrows are the detection of patently spurious rules, useless, spurious and Σ-spurious states and such, which we regrouped under the banner of the *cleanup* operation (*cf.* 12[p88]), and the detection of "interchangeable symbols", which we called *signature-quotienting* (*cf.* 13[p107]).
- ◇ Secondary tactics, much more involved and expensive, but not quite as much so as a frontal assault. The aim here is to try to find and take advantage of hidden weaknesses which render the problem much simpler that it initially appears. Who knows, the enemy might be nothing more than a giant with feet of clay... Such a tactic is the generation of a number of dependency graphs in the hope of finding a trivial one (*cf.* 14[p118]). Sometimes, it may also pay to attempt to divide the opposing army, and

engage into skirmishes and "smaller battles" on several fronts at once, for it may be that one of those fronts gives way quickly. Our corresponding tactic will be an attempt to disentangle the various final states of the `TAGED`, in case that some of them are only loosely tied to the rest of the automaton – for instance if two final states correspond to two very different kinds of accepted terms. This may provide either a quick resolution or grounds to eliminate a few states and reduce the overall size of the problem. This last tactic will be discussed in this chapter.

- ◇ Finally, a tactic for those sad days when all other tactics have failed: brutal, almost mindless frontal assault, knowing full well that barring miracles, favourable horoscopes and other divine interventions, the chances of success are laughably low. Our own "tactic of the desperate man" is the brute force algorithm of chapter 15[p131].

16.1 Outline of the algorithm

Figure 16.1[p141] presents a semi-formal outline of our global strategy – presenting the code in full would have been counterproductive. As usual with such an outline, many things are left in the dark, which we will detail and explain in this section.

- ◇ **Line 2 :** Our initial volley of arrows. First we compute $\mathcal{A}$'s friendly quotient (*cf.* conjecture 82[p115]), and apply the cleanup operation.
- ◇ **Line 3 :** The "candidates" are the final states of the automaton; notice the loop next line. What we are doing is taking each final state individually, and see whether it can be safely disentangled from the rest of the `TAGED` $\mathcal{A}$.
- ◇ **Line 4 :** Clearly, it goes without saying that if there are no final states at all, $\mathcal{A}$ is empty.
- ◇ **Line 6 :** Here we detail what we meant earlier by "disentangling" a final state from the rest of the automaton. We compute and reduce as much as possible a projection of $\mathcal{A}$ on each final state q_f; note that there may be sizeable chunks of $\mathcal{A}$ which are not connected at all to q_f (*ie.* are not *friends* of q_f, *cf.* definition 50[p89]), but which were connected to other final states of $\mathcal{A}$. However, since after the projection none of these other final states are still final anymore, and as the cleanup operation encompasses the removal of states which are not friends of a final state (*cf.* theorem 53[p91]),

```
   Data: A positive TAGED 𝒜
   Result: One of Nothing, Something, Run ρ,t, Failure
 1 begin
 2     𝒜 ← cleanup 𝒜^s_{≡_s};
 3     candidates ← 𝒜.F;
 4     if candidates = ∅ then return Nothing;
 5     foreach q_f ∈ candidates do
 6         focus ← cleanup 𝔓rj(𝒜, q_f);
 7         if focus.F = ∅ then remove q_f from candidates and continue loop;
 8         if focus is diagonal then return Something;
 9         if focus is significantly smaller than 𝒜 then
10             switch dependency graphs on focus do
11                 case Something: return Something;
12                 case Failure: do nothing;
13             switch brutal algo on focus do
14                 case Run ρ,t: return Run ρ,t;
15                 case Nothing: remove q_f from candidates;
16                 case Failure: do nothing;
17     if candidates = ∅ then return Nothing;
18     if candidates ≠ 𝒜.F then 𝒜 ← cleanup 𝔓rj(𝒜, candidates);
19     switch dependency graphs on 𝒜 do
20         case Something: return Something;
21         case Failure: do nothing;
22     switch brutal algo on 𝒜 do
23         case Run ρ,t: return Run ρ,t;
24         case Nothing: return Nothing;
25         case Failure:
26             try over and under approximations;
27             return Failure;
28 end
```

Figure 16.1 — Outline of the emptiness decision algorithm

the resulting automaton, called "focus" (on q_f) may be much smaller than $\mathcal{A}$. Note that we have

$$\mathcal{L}\mathrm{ng}\,(\mathcal{A}) = \bigcup_{q_f \in F} \mathcal{L}\mathrm{ng}\Big(\text{focus on } q_f\Big).$$

◇ **Line 7 :** We test whether our current focus is trivially empty; it might very well be that q_f turns out to be spurious or Σ-spurious, or anything else which causes the cleanup operation to remove it[(a)]. Then we know that this $\mathcal{A}$ does not have any accepting run whose root state is q_f – if it had, the focus on q_f would not be empty. If the focus turns out to be empty, we remove q_f from the candidates and try the other final states. See line 18 what happens to the candidates when we have filtered out all of those whose focus was empty.

◇ **Line 8 :** Supposing we make it this far, the focus *must* have final states – well, being a focus, it has in fact *exactly* one final state, but it is not important. What *is* important is that it has *a* final state, which is clearly reachable since the focus is the product of a cleanup operation and that each state of a cleaned-up `TAGED` is necessarily reachable. If it also turns that the focus is a diagonal `TAGED` then by theorem 56[p92] it must be non-empty. And if the focus is non-empty, then $\mathcal{A}$ itself must be non-empty, and we can conclude immediately.

◇ **Line 9 :** If we ever reach this line, it means that we could not conclude inexpensively on the question of whether or not the focus is empty. So our new question is: would it be a good idea to bring the heavy artillery to bear on the focus, or would we be better inspired to move on and look for an easier point of approach? To decide that, we examine whether the computation of the focus has resulted in a significantly smaller automaton than the original `TAGED` $\mathcal{A}$. In the worst case where there is only one final state in $\mathcal{A}$, the – one and only – focus will be the very same `TAGED` as $\mathcal{A}$. It is clear then that there is no point in spending time on a sub-problem which is as hard as the whole problem. In the best case, typically if we have a `TAGED` which has been built as a union of smaller `TAGED`s, the size of

(a) Although at the moment of writing this I am taken by a strong doubt... Given the way the cleanup operation is built, it is more probable that in that case q_f would have been caught beforehand by the initial cleanup line 2. This being said, even if the test is useless it is completely inexpensive, and it *might* become useful in the future if the cleanup operation is extended in a way which breaks the above (conjectured) property.

the focus may be a small fraction of that of $\mathcal{A}$, and as a result, *much* more tractable than $\mathcal{A}$. In that latter case, we most definitely want to sound the charge! If all goes well, we might be able to decide emptiness for the focus, thereby ending the battle if it is not empty, or getting rid of a few states if it is. In the implementation, we have taken "significantly smaller" to be "less than one half the size of the original".

- **Lines 17 and 18 :** If we reach these line, then we have examined every focus and have been unable to conclude that any of them was non-empty – because if we had, the algorithm would have ended there and then. On the other hand, we may have been able to conclude that some of the focuses, if not all of them, were empty, and eliminated the corresponding final states from the candidates. Of course, if *all* final states have been so eliminated, $\mathcal{A}$ is empty and we conclude immediately. Otherwise, if we have eliminated *some* final states but not quite all of them, we reduce $\mathcal{A}$ as much as we can before proceeding to the final battle.

- **Other lines :** The remaining lines are all applications of semi-expensive (dependency graphs, *cf.* 14[p118]) and very expensive (brutal algorithm, *cf.* 15[p131]) tactics. See their respective chapters for explanations.

- **Everywhere :** The algorithm is of course equipped with a timeout – or more specifically each tactic is equipped with a timeout – which causes it to abort and yield *Fail* if the computation threatens to take a few more millennia than one is willing to wait. In practice the timeout threshold has been kept very low in our tests (of the order of one half-second) given that we needed to test hundreds of random `TAGED`s to get statistically significant results.

16.2 Experiments

Limited experimental results have been shown in table 14.1[p130]. A great number of other experiments have been performed, but those are probably the most pertinent since they used the latest version of the (implemented) emptiness algorithms and random `TAGED`s generation.

Note that the implementation was not fully up-to-date with the techniques presented in this report, and that some of those techniques, such as signature-quotienting, were not implemented at all. As it is the results are quite good, with more than 70% of cases decided through simple approaches even with

automata accepting fairly high terms. In comparison the brutal algorithm does not perform well – nor did we expect it to – and fails to solve anything but the most trivial instances. It should be noted that its timeout was set fairly low (one half second), but other experiments have shown that increasing it, even by an order of magnitude, had little effect.

So it would appear that finding more and more specialised approaches to deal with special cases, or perhaps finding a wholly different systematic approach[(b)], is a more viable strategy than optimising variants of the brutal algorithm.

16.3 Conclusion

In this chapter we have glued together all the different tactics which we had introduced in the previous chapters. The strategy outlined here can of course be extended with other tactics.

The next (brief) chapter offers a global conclusion regarding our general approach of the emptiness problem.

[(b)] Maybe choice functions?

Chapter 17

Conclusion

In the conclusion of our efforts to come up with a suitable protocol for random evaluation of our algorithms (*cf.* chapter 10[p83]), we spoke briefly of *the dangers of being both the designer of an algorithm and that of the home-made experimental system intended to test it*. Let us emphasise this by saying that the results presented in table 14.1[p130], or any other such table for that matter, tell just as much about the TAGEDs which were used for the tests as they do about the efficiency of our algorithms. This *caveat* is of course true of pretty much *any* experiment, in a sense, but happened to be felt very acutely in our work.

TAGEDs are a fairly new kind of automata, firmly rooted in theoretical grounds, which have yet to be used in any implementation that could be taken as reference. Neither of course does there exist any library of "real-world TAGEDs". In their absence, we developed, tested and assayed our techniques and algorithms against what we intuitively perceived to be likely characteristics for hypothetical TAGEDs involved in what we imagined to be potential applications.

We honestly believe that the choices we have made in this respect are legitimate and coherent, and that our experimental protocol (in particular the fourth generation of random TAGEDs) is fairly representative of what one could expect from the real world. To that effect, our research took the form of a (thankfully *not* endless) loop where we would

- ◇ Have an idea, formalise and implement it.
- ◇ Test it against the best random TAGEDs at our disposal, and tweak the approach to get the best possible results.
- ◇ If the results were too good (not enough failures) or unbalanced (the automata turned out to be predominantly empty/non-empty), try to generate

automata for which the results *were* balanced and put our algorithms in difficulty.

Thus we were able to weed out cases such as, for instance, the second generation of random `TAGED`s, for which it turned out that almost all were non-empty, and decidable instantly and painlessly by the brutal algorithm.

Though we believe that the surviving approaches, presented in the previous chapters, fulfilled our objectives – inasmuch as we could hope in the alloted time frame – there is obviously still much which could be done, both in terms of new approaches and improvements over existing ones. The reader will have noticed that, although nearly all our results are duly justified, there are still a few conjectures and suggestions for improvement lying around in footnotes or between parentheses. Those are generally symptoms of late ideas for which there was no more time when they came up. More importantly, there are paths which we imagine to be potentially quite interesting but which remain completely or almost completely unexplored; for instance the use of choice functions (*cf.* [Fil08, FTT08b]) or the rewriting of any positive `TAGED` into a new `TAGED` for which the approach of chapter 14[p118] would decide emptiness in all cases.

Bibliography

[ABB+05] Alessandro Armando, David A. Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Paul Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The avispa tool for the automated validation of internet security protocols and applications. In Etessami and Rajamani [ER05], pages 281–285.

[AC05] Alessandro Armando and Luca Compagna. An optimized intruder model for sat-based model-checking of security protocols. In A. Armando and L. Viganò, editors, *Electronic Notes in Theoretical Computer Science*, volume 125, pages 91–108. Elsevier Science Publishers, March 2005. Presented to the IJCAR04 Workshop ARSPA, available at http://www.avispa-project.org.

[ACC07] A. Armando, R. Carbone, and L. Compagna. Ltl model checking for security protocols. In *CSF*, pages 385–396, 2007.

[AEF+05] Roy Armoni, Sergey Egorov, Ranan Fraer, Dmitry Korchemny, and Moshe Y. Vardi. Efficient ltl compilation for sat-based model checking. In *ICCAD* [DBL05], pages 877–884.

[AJMd02] P. A. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso. Regular tree model checking. In E. Brinksma and K. G. Larsen, editors, *Computer Aided Verification, CAV'02*, volume 2404 of *Lecture Notes in Computer Science*, pages 555–568. Springer-Verlag, July 27–31 2002.

[ASV09] S. Abiteboul, L. Segoufin, and V. Vianu. Modeling and verifying active xml artifacts. *IEEE Data Eng. Bull.*, 32(3):10–15, 2009.

[Baa07] Franz Baader, editor. *Term Rewriting and Applications, 18th International Conference, RTA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4533 of *Lecture Notes in Computer Science*. Springer, 2007.

[BBGM08] E. Balland, Y. Boichut, T. Genet, and P.-E. Moreau. Towards an efficient implementation of tree automata completion. In *AMAST*, pages 67–82, 2008.

[BC06] Marco Bernardo and Alessandro Cimatti, editors. *Formal Methods for Hardware Verification, 6th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2006, Bertinoro, Italy, May 22-27, 2006, Advanced Lectures*, volume 3965 of *Lecture Notes in Computer Science*. Springer, 2006.

[BCHK08] Y. Boichut, R. Courbis, P.-C. Héam, and O. Kouchnarenko. Finer is better: Abstraction refinement for rewriting approximations. In *Rewriting Techniques and Application, RTA'08*, volume 5117 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 2008.

[BGJ08] B. Boyer, T. Genet, and T. P. Jensen. Certifying a tree automata completion checker. In *IJCAR'08*, volume 5195 of *Lecture Notes in Computer Science*, pages 523–538. Springer, 2008.

[BGJR07] Y. Boichut, Th. Genet, Th. P. Jensen, and L. Le Roux. Rewriting approximations for fast prototyping of static analyzers. In Baader [Baa07], pages 48–62.

[BHH+08a] A. Bouajjani, P. Habermehl, L. Holík, T. Touili, and T. Vojnar. Antichain-based universality and inclusion testing over nondeterministic finite tree automata. *Implementation and Applications of Automata*, pages 57–67, 2008.

[BHH+08b] A. Bouajjani, P. Habermehl, L. Holík, T. Touili, and T. Vojnar. Antichain-based universality and inclusion testing over nondeterministic finite tree automata. *Lecture Notes in Computer Science*, 5148:57–67, 2008.

[BHK08] Y. Boichut, P.-C. Héam, and O. Kouchnarenko. Approximation-based tree regular model-checking. *Nordic Journal of Computing*, 14:194–219, 2008.

[BHK09] Y. Boichut, P.-C. Héam, and O. Kouchnarenko. Tree automata for detecting attacks on protocols with algebraic cryptographic primitives. *ENTCS*, 239:57–72, 2009. Infinity 2006, 2007, 2008 Best papers.

[BHRV06] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular tree model checking. In *Infinity'05*, volume 149 of *Electronic Notes in Theoretical Computer Science*, pages 37–48, 2006.

[BMSS09] M. Bojanczyk, A. Muscholl, Th. Schwentick, and L. Segoufin. Two-variable logic on data trees and xml reasoning. *J. ACM*, 56(3), 2009.

[BS69] J.L. Bell and A.B. Slomson. *Models and Ultraproducts, an introduction*. Dover Publications., 1969.

[BSS93] K. Butler, M. Stephens, and STANFORD UNIV CA DEPT OF STATISTICS. The distribution of a sum of binomial random variables. 1993.

[BT92] B. Bogaert and S. Tison. Equality and disequality constraints on direct subterms in tree automata. In *STACS*, pages 161–171, 1992.

[BT02] A. Bouajjani and T. Touili. Extrapolating tree transformations. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification, CAV'02*, volume 2404 of *Lecture Notes in Computer Science*, pages 539–554. Springer-Verlag, July 27–31 2002.

[CBRZ01] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[CC05] H. Comon and V. Cortier. Tree automata with one memory set constraints and cryptographic protocols. *Theoretical Computer Science (TCS'05)*, 331, 2005.

[CDG+07] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. 2007. release October, 12th 2007.

[CHK09] Roméo Courbis, Pierre-Cyrille Héam, and Olga Kouchnarenko. Taged approximations for temporal properties model-checking. In Maneth [Man09], pages 135–144.

[CLJP08] H. Comon-Lundh, F. Jacquemard, and N. Perrin. Visibly tree automata with memory and constraints. *Logical Methods in Computer Science*, 4(2), 2008.

[CLM07] Nathalie Caspard, Bruno Leclerc, and Bernard Monjardet. *Ensembles ordonnés finis: concepts, résultats et usages*. Springer, 2007.

[DBL05] *2005 International Conference on Computer-Aided Design (ICCAD'05), November 6-10, 2005, San Jose, CA, USA*. IEEE Computer Society, 2005.

[DCC95] M. Dauchet, A.-C. Caron, and J.-L. Coquidé. Automata for reduction properties solving. *J. Symb. Comput.*, 20(2):215–233, 1995.

[DIMV09] A. H. Dediu, A.-M. Ionescu, and C. Martín-Vide, editors. *Language and Automata Theory and Applications (LATA 2009), Tarragona, Spain, April 2-8, 2009. Proceedings*, volume 5457 of *Lecture Notes in Computer Science*. Springer, 2009.

[DWDHR06] M. De Wulf, L. Doyen, T.A. Henzinger, and J.F. Raskin. Antichains: A new algorithm for checking universality of finite automata, 2006.

[DWDR+06] M. De Wulf, L. Doyen, J.F. Raskin, et al. A lattice theory for solving games of imperfect information, 2006.

[EM07] S. Escobar and J. Meseguer. Symbolic model checking of infinite-state systems using narrowing. In *Rewriting Techniques and Applications, RTA'07*, pages 153–168, 2007.

[ER05] Kousha Etessami and Sriram K. Rajamani, editors. *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*. Springer, 2005.

[FGT04] Guillaume Feuillade, Thomas Genet, and Valérie Viet Triem Tong. Reachability analysis over term rewriting systems. *J. Autom. Reasoning*, 33(3-4):341–383, 2004.

[Fil08] Emmanuel Filiot. *Logics for n-ary queries in trees.* PhD thesis, Université des Sciences et Technologie de Lille - Lille I, 10 2008.

[FTT08a] Emmanuel Filiot, Jean-Marc Talbot, and Marseille Sophie Tison. Tree automata techniques and applications (slides). 2008.

[FTT08b] Emmanuel Filiot, Jean-Marc Talbot, and Sophie Tison. Tree Automata with Global Constraints. In *12th International Conference on Developments in Language Theory (DLT)*, pages 314–326, Kyoto Japon, 2008.

[GGW06] Aarti Gupta, Malay K. Ganai, and Chao Wang. Sat-based verification methods and applications in hardware verification. In Bernardo and Cimatti [BC06], pages 108–143.

[GJV] L.B.C.C.G. Godoy, F. Jacquemard, and C. Vacher. The Emptiness Problem for Tree Automata with Global Constraints.

[GL07] Jean Goubault-Larrecq. Logique propositionnelle, **P**, **NP**. lsv.ens-cachan.fr, december 2007.

[GT95] R. Gilleron and S. Tison. Regular tree languages and rewrite systems. *Fundamenta Informatica*, 24(1/2):157–174, 1995.

[HIRV06] P. Habermehl, R. Iosif, A. Rogalewicz, and T. Vojnar. Abstract regular tree model checking of complex dynamic data structures. In *SAS'06, 13th International Static Analysis Symposium*, volume 4134 of *Lecture Notes in Computer Science*, pages 52–70, 2006.

[HNS09] P.C. Héam, C. Nicaud, and S. Schmitz. Random Generation of Deterministic Tree (Walking) Automata. 2009.

[HU79] J. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[IYG+08] Franjo Ivancic, Zijiang Yang, Malay K. Ganai, Aarti Gupta, and Pranav Ashar. Efficient sat-based bounded model checking for software verification. *Theor. Comput. Sci.*, 404(3):256–274, 2008.

[JKV09] F. Jacquemard, F. Klay, and C. Vacher. Rigid tree automata. In Dediu et al. [DIMV09], pages 446–457.

[JR09] F. Jacquemard and M. Rusinowitch. Rewrite based verification of xml updates. *CoRR*, abs/0907.5125, 2009.

[JRV06] F. Jacquemard, M. Rusinowitch, and L. Vigneron. Tree automata with equality constraints modulo equational theories. In *IJCAR*, pages 557–571, 2006.

[Kaa08] Lisa Kaati. *Reduction Techniques for Finite (Tree) Automata*. PhD thesis, Uppsala University, Computer Systems, 2008.

[KL07] W. Karianto and Ch. Löding. Unranked tree automata with sibling equalities and disequalities. In *ICALP*, pages 875–887, 2007.

[Lam94] L. Lamport. A temporal logic of actions. *ACM Transactions On Programming Languages And Systems, TOPLAS*, 16(3):872–923, May 1994.

[Man09] Sebastian Maneth, editor. *Implementation and Application of Automata, 14th International Conference, CIAA 2009, Sydney, Australia, July 14-17, 2009. Proceedings*, volume 5642 of *Lecture Notes in Computer Science*. Springer, 2009.

[Mes92] J. Meseguer. Conditioned rewriting logic as a united model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. SV, 1992.

[MSV07] P. Manolios, S.K. Srinivasan, and D. Vroon. BAT: The bit-level analysis tool. *Lecture Notes in Computer Science*, 4590:303, 2007.

[Mur99] M. Murata. Hedge automata: a formal model for XML schemata, 1999.

[MV09] B.C.P. Manolios and D. Vroon. Faster SAT Solving with Better CNF Generation. *Design, Automation and Test in Europe, DATE*, 2009.

[Nev02] F. Neven. Automata theory for XML researchers. *ACM SIGMOD Record*, 31(3):39–46, 2002.

[OT05] H. Ohsaki and T. Takai. ACTAS: A system design for associative and commutative tree automata theory. *Electronic Notes in Theoretical Computer Science*, 124(1):97–111, 2005.

[Pnu77] A. Pnueli. The temporal logic of programs. In *FOCS'77*, pages 46–57, 1977.

[Sch04] T. Schwentick. Trees, automata and XML. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, page 222. ACM, 2004.

[Sch07] Th. Schwentick. Automata for xml—a survey. *J. Comput. Syst. Sci.*, 73(3):289–315, 2007.

[Spr09] Springer. *TAGED Approximations for Temporal Properties Model-Checking*, 2009.

[SSMH04] H. Seidl, Th. Schwentick, A. Muscholl, and P. Habermehl. Counting in trees for free. In *ICALP*, pages 1136–1149, 2004.

[TV05] D. Tabakov and M. Vardi. Experimental evaluation of classical automata constructions. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 396–411. Springer, 2005.

Index

Algorithms for Tree Automata with Constraints

Keywords: Tree automata, Tree automata with constraints, TAGED, membership problem, *NP*-completeness, SAT solvers, conversion to CNF, emptiness problem, *EXPTIME*-completeness

Abstract: Tree automata are a widely used formalism in Computer Science. Since their creation in the fifties, numerous more expressive extensions have been created, mainly for theoretical needs. Unfortunately the decision problems associated to these extensions are too often undecidable or in prohibitive classes of algorithmic complexity (*NP*-complete or worse), and not much work has been done to find efficient heuristics for them. This makes it difficult to implement any sufficiently efficient tool based on these formalisms. This Master's project and internship report proposes efficient approaches for the membership problem for TAGEDs (tree automata with global equality and disequality constraints), and the emptiness problem for positive TAGEDs , which are respectively *NP*-complete and *EXPTIME*-complete. The approach proposed for the membership problem is based on a SAT encoding of the problem, which leverages the efficient heuristics of modern SAT solvers. For the emptiness problem, we introduce several techniques aimed at reducing the size of the problem and decide as many special cases as possible before resorting to a general but much more expensive algorithm. We also introduce several methods for generating random tree automata, with or without constraints.

Mots-clefs: Automates d'arbres, Automates d'arbres avec contraintes, TAGED, problème d'appartenance, *NP*-complétude, solvers SAT, conversion vers CNF, problème du vide, *EXPTIME*-complétude

Résumé: Les automates d'arbres sont un formalisme très utilisé en informatique. Depuis leur création dans les années cinquante, de nombreuses extensions, plus expressives, ont été créées, principalement pour résoudre des problèmes théoriques. Malheureusement, les problèmes de décision associés à ces extensions sont le plus souvent indécidables ou dans des classes de complexité prohibitives (*NP*-complet ou pire), et peu de recherches ont été effectuées pour leur trouver des heuristiques efficaces. Ceci fait qu'il est difficile d'implanter des outils raisonnablement efficaces basés sur ces formalismes. Ce rapport de projet et de stage de Master propose des approches pour le problème d'appartenance au langage reconu par un TAGED (automate d'arbre avec contraintes globales d'égalité et de différence) et le problème du vide pour les TAGEDs positifs. Ces deux problèmes sont respectivement *NP* et *EXPTIME*-complets. L'approche proposée pour le problème d'appartenance est fondée sur un codage SAT du problème, qui tire profit des heuristiques efficaces des solvers SAT modernes. Pour le problème du vide, nous introduisons plusieurs techniques permettant de réduire la taille du problème de de décider autant de cas particuliers que possible, avant de recourir à un algorithme général mais nettement moins efficace. Nous introduisons également plusieurs méthodes pour générer des automates d'arbres aléatoires, avec ou sans contraintes.