Non-determinism, explorable automata and cyclic proofs

Émile Hazard


Numéro National de thèse : 2022ENSL0022

THÈSE

en vue de l'obtention du grade de Docteur, délivré par

l'École Normale Supérieure de Lyon

Ecole Doctorale N°512

École Doctorale en Informatique et Mathématiques de Lyon

Discipline : Informatique

Soutenue publiquement le 30/09/2022, par :

Émile Hazard

Non-determinism, explorable automata and cyclic proofs

Non-déterminisme, automates explorables et preuves cycliques

Devant le jury composé de :

Nathalie Bertrand	Directrice de recherche, Université de Rennes	Rapporteure
Luigi Santocanale	Professeur des universités, Université d'Aix-Marseille	Rapporteur
Olivier Carton	Professeur des universités, Université Paris Cité	Examinateur
Damien Pous	Directeur de Recherche, ENS de Lyon	Directeur de thèse
Denis Kuperberg	Chargé de Recherche, ENS de Lyon	Co-encadrant de thèse

Abstract

This thesis is divided into two main parts, although a common denominator can be found in the notion of non-determinism in automata, and its resolution.

The first part focuses on a notion of limited non-determinism, which is explorable automata. These are automata for which the non-determinism can be resolved by building a finite number of simultaneous runs. This generalizes the notion of Good-For-Games (GFG) automata, which correspond to the case where a single run is enough. We give another link between these two notions, namely the fact that GFGness is decidable in PTime for explorable automata. We prove that deciding whether an automaton is explorable is an ExpTime-complete problem for finite and Büchi automata. We also consider the problem of ω-explorability, in which we allow for a countable number of simultaneous runs. This problem is non-trivial for automata of infinite words, and we prove the ExpTime-completeness of deciding the ω-explorability of a co-Büchi automaton.

In the second part, we describe a cyclic proof system designed to provide certificates of inclusions between languages of infinite words. These languages are represented by ω-regular expressions, or in the final version by a generalization of these expressions, allowing for arbitrary nesting of the ω operator. We prove the soundness and completeness of that system, along with some algorithmic results: the decidability of the validity of a cyclic proof, and a PSpace proof search algorithm to decide the inclusion of languages.

Keywords: Non-determinism, automata, complexity, cyclic proofs, transfinite words

Résumé

Cette thèse est divisée en deux parties, qui ont en commun le problème du non-déterminisme chez différents modèles d'automates, et la résolution de celui-ci.

La première partie s'intéresse à une forme de limitation du non-déterminisme, que l'on nomme l'explorabilité. Un automate est explorable s'il est possible de résoudre le non-déterminisme en créant un nombre finis d'exécutions en parallèle. Il s'agit d'une généralisation de la notion d'automate Good-For-Games, *i.e.* Bon-Pour-les-Jeux (GFG), qui correspond au cas où une seule exécution suffit. Nous commençons par fournir un lien supplémentaire entre ces deux notions : le fait que l'on peut décider en temps polynomial si un automate est GFG, à condition de savoir qu'il est explorable. Nous montrons ensuite que la reconnaissance des automates GFG est un problème PSpace-complet dans le cas des automates de mots finis ou de Büchi. Nous nous intéressons aussi au problème de l'ω-explorabilité, qui correspond au cas où l'on s'autorise un nombre dénombrable d'exécutions. Celui-ci est non trivial dans le cas des mots infinis, et nous montrons que décider l'ω-explorabilité d'un automate de co-Büchi est ExpTime-complet.

Dans la seconde partie, nous présentons un système de preuves cycliques permettant de fournir des certificats d'inclusions entre langages de mots infinis. Ces langages sont représentés par des expressions ω-régulières dans un premier temps, puis par une généralisation de ces expressions, qui permet notamment des imbrications de l'opérateur ω. Nous montrons la correction et la complétude de ce système, et fournissons deux résultats algorithmiques : la décidabilité de la validité d'une preuve cyclique, ainsi qu'un algorithme PSpace pour décider l'inclusion de deux langages par la recherche de preuve.

Mots-clefs : Non-déterminisme, automates, complexité, preuves cycliques, mots transfinis

Résumé long en français

Une façon de présenter cette thèse est de commencer par introduire la notion de non-déterminisme, qui y occupe une position centrale, et fait le lien entre les deux parties présentées ici. Étant donné un problème de décision (*i.e.* auquel on répond par oui ou non à toute entrée), une machine non-déterministe résolvant ce problème est une machine qui peut se comporter de différentes façons face à une même entrée, avec la contrainte suivante : il existe un comportement aboutissant à un résultat positif si et seulement si l'entrée est une instance positive du problème. À l'inverse, une machine déterministe ne fait jamais de choix, et il y a donc une unique exécution possible face à une entrée donnée (et la réponse de la machine est donc celle obtenue en fin d'exécution).

Les machines auxquelles nous nous intéressons ici sont des automates, mais la relation entre modèle déterministe et non-déterministe est également étudiée dans d'autres contextes, comme celui des machines de Turing. Chaque modèle présente ses avantages et inconvénients : dans le cas des automates par exemple, un modèle déterministe sera plus simple à manipuler, mais parfois exponentiellement plus gros que son équivalent non-déterministe. C'est pourquoi il est intéressant de chercher des automates "intermédiaires", ce qui correspond à trouver des automates non-déterministes mais possédant de bonnes propriétés. Un tel ensemble est celui des automates Good-For-Games (GFG), *i.e.* Bon-Pour-les-Jeux, pour lesquels les choix non-déterministes peuvent être faits en connaissant les calculs faits jusqu'à présent. Comme l'indique leur nom, ces automates ont un lien privilégié avec la théorie des jeux : ils présentent l'avantage de se comporter comme des automates déterministes lorsque l'on fait leur produit avec un jeu.

Dans la première partie, nous nous intéressons à une généralisation des automates GFG, que l'on appelle les automates explorables. Il s'agit d'automates pour lesquels on peut construire un nombre fini d'exécutions, avec la garantie qu'au moins l'une d'entre elles sera acceptante si l'entrée est acceptée. Plus formellement, cela correspond à un jeu entre un joueur choisissant les lettres d'entrée, et un autre qui choisit les transitions prises par chaque exécution. Les automates GFG correspondent au cas où une unique exécution suffit. Un intérêt de cette notion d'explorabilité est le fait qu'elle facilite la reconnaissance des automates GFG. Nous montrons en effet que décider si un automate explorable est GFG peut se faire en temps polynomial. Cependant, les résultats obtenus ne permettent pas d'améliorer les algorithmes actuels de reconnaissance d'automates GFG, puisque nous montrons que décider l'explorabilité est un problème ExpTime-difficile en général, avec un algorithme ExpTime dans le cas des automates de mots finis ou des automates de Büchi.

Dans un second temps, nous explorons le problème de l'ω-explorabilité, qui correspond à autoriser un nombre infini dénombrable d'exécutions parallèles. L'intuition est que dans ces conditions, l'adversaire choisissant les lettres doit être capable de cibler une exécution à "éliminer", là où précédemment toute élimination avait la même valeur. Malgré cette difficulté supplémentaire pour l'adversaire, il s'avère que même une infinité dénombrable de tentatives peut être insuffisante pour trouver une exécution acceptante dans le cas de mots infinis. Nous montrons que le problème de décision associé (déterminer s'il est possible de trouver une exécution en en lançant une infinité dénombrable) est ExpTime-difficile ici aussi, dès lors que le modèle considéré contient les automates de sûreté (safety),

ce qui est le cas des modèles les plus usuels. Nous donnons également la borne supérieure correspondante en ExpTime pour ce problème, cette fois-ci dans le cas des automates de co-Büchi, en réduisant le jeu associé à un jeu à arène finie.

La seconde partie s'intéresse à un autre problème : celui du test d'inclusion de langages, que l'on approche à l'aide de la théorie de la preuve. Une instance du problème correspond donc à deux langages, représentés par des expressions, et l'on souhaite savoir si le premier est inclus dans le second. Le cas des langages réguliers (de mots finis) a déjà été traité à l'aide de preuves cycliques, qui permettent de traduire des raisonnements infinitaires, tels que la descente infinie. Nous étendons ce résultat en un système adapté aux expressions ω-régulières, produisant également des arbres de preuves cycliques. Nous montrons la correction (les preuves sont correctes) et la complétude (tout résultat correct est prouvable) de ce système. Dans un second temps, nous cherchons à étendre ce système à une généralisation des expressions ω-régulières, pour laquelle l'opérateur $\cdot^\omega$ (concaténation infinie) n'a plus de contraintes sur son positionnement dans l'expression. Ceci correspond à des langages de mots transfinis, pouvant être indexés par tous les ordinaux inférieurs à ω^ω. Notre incapacité à fournir un système d'arbres de preuve identique au précédent nous a mené à une version modifiée. Dans celle-ci, une preuve n'est plus un arbre cyclique, mais une forêt d'arbres cycliques, dans laquelle de nouveaux comportements apparaissent, avec notamment la possibilité de passer d'un arbre à l'autre en suivant certaines règles. Outre un résultat de correction similaire à celui du précédent système, le résultat de complétude obtenu permet de se restreindre aux preuves contenant un nombre fini d'arbres, ce qui engendre une représentation finie et donc la possibilité d'une exploitation algorithmique. Nous prouvons ainsi qu'un algorithme de recherche de preuve PSpace est possible pour exploiter ce système en vue de décider des inclusions de langages, ce qui constitue un nouveau résultat dans le cas des langages de mots transfinis. Cet algorithme est optimal puisque la borne inférieure PSpace est connue dès le cas des mots finis. Nous fournissons également un algorithme PSpace de vérification de preuve.

Un lien entre ces deux parties (automates explorables et preuves cycliques d'inclusions) peut être trouvé en considérant la place prépondérante qu'occupe la notion de non-déterminisme dans ces deux sujets. Les automates explorables constituant une forme de déterminisme partiel, que la première partie étudie dans le but d'améliorer notre compréhension du non-déterminisme. Son importance dans la première partie est donc claire. En revanche, lorsque l'on s'intéresse aux preuves cycliques dans la seconde partie, cette relation est moins évidente. Elle est pourtant bien présente, mais il est nécessaire pour la voir d'entrer davantage dans les détails d'une preuve. Intuitivement, prouver l'inclusion de langages $L_1 \subseteq L_2$ revient à fournir, pour chaque mot de L_1, une méthode pour certifier son appartenance à L_2. Pour ce faire, les choix non-déterministes venant potentiellement avec la définition de L_2 doivent être résolus, ce qui constitue une des difficultés principales d'un tel système. On peut ainsi voir une preuve de $L_1 \subseteq L_2$ comme un outil fournissant une partielle résolution du non-déterminisme de L_2. Un autre point à noter ici est le fait que le problème de l'inclusion $L_1 \subseteq L_2$ devient significativement plus simple lorsque l'on dispose d'un automate GFG pour L_2, ou à défaut d'un automate explorable avec k exécutions.

Remerciements

Voici venue la partie la plus délicate de la rédaction, dans laquelle je vais essayer de n'oublier personne. Il n'y aura pas de plan très défini dans cette partie, donc l'ordre importe peu (*i.e.* merci de ne pas me taper si vous venez après quelqu'un de clairement moins important).

Tout d'abord, merci à toi, Denis, pour m'avoir accompagné pendant ces trois ans. Au cours de cette période, la recherche et l'excellence académique ont connu des hauts et des bas, mais j'espère que tu as apprécié le voyage. Nos discussions scientifiques m'ont souvent évité des écueils, ou ont débloqué des situations, mais je suis également reconnaissant pour les divers autres échanges que nous avons pu avoir, qu'il s'agisse de politique, de jonglage, ou de n'importe quoi d'autre (moins intéressant que les deux premiers). Merci aussi à toi Damien, nous t'avons moins souvent sollicité, mais nos interactions étaient fructueuses, notamment au sujet des arbres de preuves.

J'aimerais ensuite remercier les membres du jury de ma soutenance de thèse : Nathalie Bertrand, Olivier Carton et Luigi Santocanale. Merci à vous trois pour l'intérêt que vous avez porté à mon travail, ainsi que pour les échanges que nous avons eus. Je remercie tout particulièrement les deux rapporteurs, Nathalie et Luigi, pour le temps que vous avez consacré à la lecture de mon manuscrit.

Merci également aux autres membres de l'ENS et aux chercheurs extérieurs avec lesquels j'ai échangé au cours de cette thèse, notamment tous les (ex-)membres de l'équipe Plume, et les co-bureaux successifs en particulier : Alexi, Christophe, Enguerrand, Laureline, Rémi. Nos échanges n'étaient pas systématiquement consacrés à l'avancement de la recherche (j'espère qu'elle s'en remettra), mais toujours intéressants. Une pensée particulière pour l'agent d'entretien à l'origine du corbeau en sac-poubelle qui trôna sur notre balcon pendant un moment, et qui, à défaut d'effrayer les pigeons, nous a fourni un sujet de conversation à de multiples reprises.

J'ai aussi pu profiter de ces trois années pour enseigner face à des élèves de l'ENS, et j'aimerais remercier les enseignants et TDmen avec lesquels j'ai partagé cette expérience : Pascal, Natacha, Daniel, Rémi, Alexi et Hugues. Je ne sais pas si tout ça m'a préparé au lycée, mais c'était en tout cas intéressant et apprécié. Merci également à tous les élèves que j'ai vu passer sur cette période (sauf une personne qui se reconnaîtra).

Je souhaiterais également avoir quelques mots pour l'islamo-gauchisme (et ses porteurs et porteuses), qui a gangréné mon travail de recherche pendant toutes ces années, et auquel je ne pouvais me soustraire, tant il est profondément enraciné dans la structure universitaire d'aujourd'hui. Je suppose que j'ai à son égard développé une forme de syndrome de Stockholm, et j'ai bien peur que ma situation actuelle d'enseignant dans le lycée le plus gréviste depuis la rentrée ne permette pas une rémission complète. Merci au passage aux collègues enseignant avec lesquels j'ai déjà tissé des liens, basés sur des luttes communes ou sur des vols de notes de cours.

Dans le prolongement de la veine islamo-gauchisme, je voudrais remercier les deux équipes d'élu·es étudiant·es avec lesquels j'ai travaillé pendant les mandats 2019-2021. J'ai beaucoup appris à vos côtés, sur le monde universitaire notamment, mais aussi sur la vie associative en général.

Ah et merci Monsieur Macron pour les 100 euros.

Les copains de Lokifer, je ne vous oublie pas : je sais que vous êtes les plus susceptibles de parcourir cette partie. Merci pour ce groupe où on a pu partager nos (més)aventures de la thèse ou de la vraie vie (est-ce exclusif ? Cette question est laissée en exercice au lecteur...). Je suis vraiment heureux d'avoir eu un tel groupe à Lyon pendant la thèse, qui ne se serait sans doute pas déroulée de la même façon sans vous. Nos diverses aventures furent souvent des distractions bienvenues. Merci plus globalement aux copains lyonnais, dont certains ne sont maintenant plus si lyonnais, pour tous les bons moments depuis mon arrivée à l'école.

Merci également à ma famille, qui a su me soutenir quand c'était nécessaire, notamment pendant cette dernière ligne droite que j'ai passée plus proche de vous et qui a été un moment compliqué pour moi. J'inclus aussi la future belle famille qui m'a déjà bien accueilli. Je suis heureux de vous présenter mon travail à tous.

Ce dernier mois précédant la soutenance a été difficile, et les membres de ma famille ne sont pas les seuls à avoir été présent pour m'aider à le traverser. Merci aux copains avec lesquels j'ai pu parler : Jodie-Lou, Angèle, Hugo, Rédouane, Colin, et les autres que j'ai moins vu, mais qui étaient quand même là régulièrement.

Les plus habitués à l'exercice des remerciements se doutent de ce que je réserve pour la fin : merci, donc, à ~~Frédérique Vidal~~ Alice, pour m'avoir supporté (double sens tavu) pendant tout ce temps. La confiance que tu as en moi a souvent compensé celle qu'il me manquait, et je ne crois pas connaître de personne plus attentionnée que toi. La vie ça fait parfois peur, mais moins avec toi.

Contents

Chapter 1

Introduction

In this part, we will first introduce a few notations, then present some notions that will be needed to understand this thesis. This corresponds to Sections 1.2 to 1.6, with Sections 1.5 and 1.6 specific respectively to the contents of Chapter 2 and Chapter 3. After that, we will give some context in Section 1.7, followed by a summary of the contributions presented in this thesis.

1.1 A word on notations

In order to make this work as easy to read as possible, we will try to adhere to the following guidelines concerning notations. Upper case letters ($L, Q \ldots$) will be used to name sets, while lower case letters ($w, q \ldots$) will instead correspond to elements of these sets. For bigger objects (automaton, tree, *etc.*), we will use upper case cursive letters ($\mathcal{A}, \mathcal{T} \ldots$). We might however forgo these rules in favor of more standard ones in some cases.

If S is a set, we call $|S|$ the cardinal of S, and $\mathcal{P}(S)$ the powerset of S, *i.e.* the set of all subsets of S. We use standard notations for sets, such as $\in$, $\subseteq$, $\cup$, $\cap$ and $\setminus$ for membership, inclusion, union, intersection and difference respectively. We will also call $[i, j]$ the set $\{i, i+1, \ldots, j\}$ for two integers $i, j \in \mathbb{N}$ (by convention, $[i, j]$ is the empty set $\emptyset$ if $j < i$).

The name Σ will be used for a non-empty finite set, which we call *alphabet*. Its elements are called *letters*.

We use the common notations for the main complexity classes: PTime, PSpace, ExpTime, 2-ExpTime (doubly exponential time), *etc.* which we will not redefine formally here. We also use the hardness notion: ExpTime-hard, *etc.*

1.2 Languages, automata and where to find them

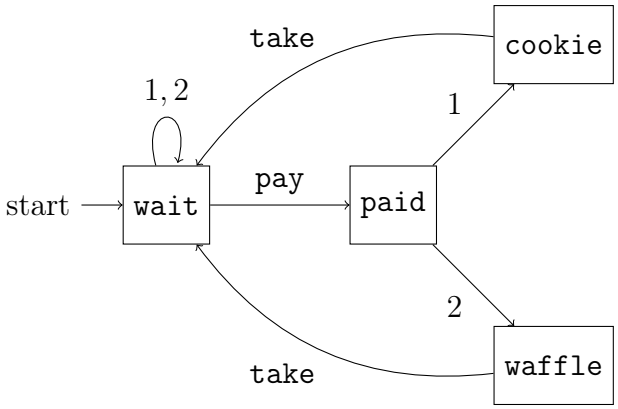
This part aims at defining the objects around which the rest of this work will revolve: languages of words, and ways to represent them (namely automata and expressions).

Automata are one possible answer to the question: "how can we model a computing system?" There are several other answers to this question, such as Turing machines or boolean circuits, but the one we want to describe here is the notion of automaton. This

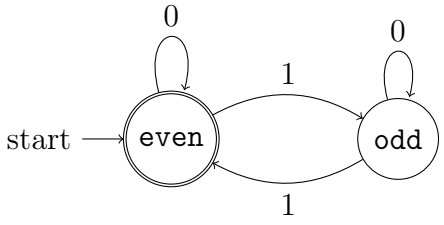
corresponds intuitively to a machine with very limited memory, as opposed to a computer that would never need all of its memory for a computation, and for which the model of Turing machines is therefore usually preferred.

A machine with a user interface can be seen as a system that changes its current state at each timestep, according to the input (*e.g.* the key(s) currently pressed by the user). This can be formally described as a set of states, which we call Q, along with the set of possible inputs Σ, and rules to change the state in reaction to an input. These rules can be described by a function δ that associates a new state to any couple of a state and an input.

Example 1.1:
Here is an automaton modeling a cookies and waffles dispenser. At the start, the machine is in the state `wait`. The user needs to first pay the price, which sends the machine to state `paid`, then type 1 for a cookie or 2 for a waffle. The machine goes to the associated state, which corresponds to serving the item. Finally, the user can take the item, which resets the machine to state `wait`.



A second, more abstract example is the following automaton, which only has two inputs: 1 and 0, corresponding for instance to pressing and not pressing a button. The state of the automaton then tells the user if he pressed it an even or odd number of timesteps.



In order to give the formal definition of an automaton, we first need to introduce the notions of word and language.

Definition (Word, language)**:**
Let us take an alphabet Σ. A *word* w over Σ is a finite sequence of letters, *i.e.* a function from $[0, n-1]$ (for some $n \in \mathbb{N}$) to Σ. n is called the length of w, sometimes denoted

$|w|$. We usually use the compact notation $w = w_0 w_1 \dots w_{n-1}$, with $w_i = w(i) \in \Sigma$ for any $i \in [0, n-1]$. The empty word, *i.e.* the only word of length 0, will be denoted ε.

A *language* is defined as a set of words. The universal language over Σ, *i.e.* the set of all words on this alphabet, is denoted Σ^*.

Definition (Deterministic finite automaton)**:**
We can now define what we call a *deterministic finite automaton* (DFA), which consists in a tuple $\mathcal{A} = (\Sigma, Q, q_0, \delta, F)$, where:

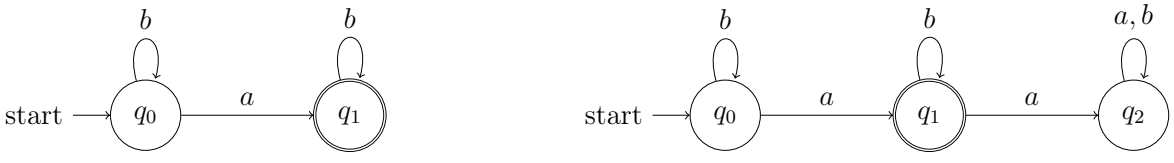
- Σ is the alphabet of the automaton, and its elements are called letters. In the previous example (second automaton from Example 1.1) it is $\{0, 1\}$.
- Q is the finite set of states of the automaton, for instance $\{\texttt{even}, \texttt{odd}\}$ above.
- q_0 is an element of Q, called the initial state. It is usually marked using a "start" arrow, like the state `even`, or simply an incoming arrow with no origin.
- δ is a function from $Q \times \Sigma$ to Q, called the transition function. It defines the evolution of states according to the input. A transition $\delta(q, a) = p$ will often be represented as $q \xrightarrow{a} p$. In the example above, $\delta(\texttt{even}, 0) = \delta(\texttt{odd}, 1) = \texttt{even}$ for instance.
- F is a subset of Q called the set of accepting states. It is usually marked by a double border in the drawing of a state, as for `even` above.

We say that a word $w = w_0 \dots w_{n-1}$ is accepted by the automaton if the sequence of states $q_0, q_1, \dots, q_n$ defined inductively by $q_{i+1} = \delta(q_i, w_i)$ (and the first state is the initial state of $\mathcal{A}$) satisfies $q_n \in F$. We call such a sequence a run, and we say that it is accepting if that final condition is satisfied. We call $\mathcal{L}(\mathcal{A})$ the language of $\mathcal{A}$, which is the set of its accepted words.

Note that although a run of a DFA is defined from a complete input word, it can be seen as a computation "on the fly" by jumping to the next state at each new letter of the input. This is the intuition behind the notion of automaton, which is meant to represent an actual machine.

Another thing to notice here is that an automaton can only answer a decision problem, *i.e.* its answers are limited to "yes" and "no", as opposed to Turing machines that can also write on their tape, which can be considered as an output.

Example 1.2:
Here are two other automata, both recognising words over $\{a, b\}$ with exactly one a.



The arrow labelled "start" on the left indicates the initial state, and the accepting states are those with a double border.

Note that the left automaton does not have transitions from q_1 labelled by a. This is a simplified version of the right automaton, which is the *complete* version. We will sometimes use this representation to save space, using the convention stating that when an automaton reads a letter that yields no transition, the word is rejected.

1.3 Non-determinism and regular expressions

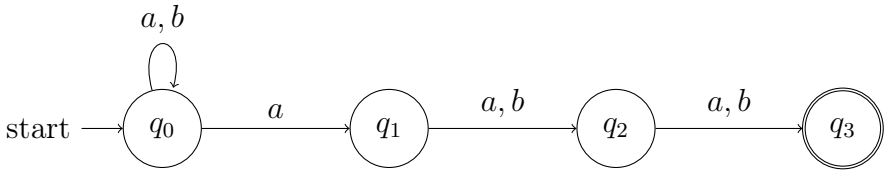
In the previous section, we defined languages along with a way to recognise them, using automata. Intuitively, the relation between these two notions is that the automaton describes the inner workings of a machine, while the language is a mathematical object describing the expected behaviour of the machine, seeing it as a black box. The first is a finite object, while the second can be infinite.

Here we will define a generalisation of the notion of automaton, that no longer corresponds to a physically realistic machine. This is the notion of non-deterministic automaton. We will then define a way to describe languages called regular expressions, and use it to illustrate how non-determinism provides a middle ground between languages and deterministic automata.

Definition (Non-deterministic finite automaton)**:**
The definition of a *non-deterministic finite automaton* (NFA) looks similar to that of a DFA: it will also be given by a tuple $\mathcal{A} = (\Sigma, Q, q_0, \Delta, F)$, with the only difference that Δ is now a set of possible transitions, with $\Delta \subseteq Q \times \Sigma \times Q$. As before, a transition $(q, a, p) \in \Delta$ can be represented as $q \xrightarrow{a} p$. This definition implies that for given state $q \in Q$ and letter $a \in \Sigma$, the number of transitions from q labelled by a might be anywhere between 0 and $|Q|$. The automaton is called *complete* if that number is never 0.

A word $w = w_0 \dots w_{n-1}$ is accepted by the automaton if there exists a run $q_0, q_1, \dots, q_n$ (where q_0 is the initial state) such that $(q_i, w_i, q_{i+1}) \in \Delta$ for any $i \in [0, n-1]$, and $q_n \in F$. We still call $\mathcal{L}(\mathcal{A})$ the language of $\mathcal{A}$.

Example 1.3:
Here is an NFA recognising words over the alphabet $\{a, b\}$ that have the letter a in the third to last position.



Upon reading a in state q_0, this automaton can either stay in state q_0 or go to q_1, and has to "guess" when the end of the word will arrive in order to provide an accepting run on the fly. In this case, the non-determinism appears as a simple way to represent the language of words with a a in third to last position.

It is worth noting at that point that NFAs are actually equivalent to DFAs, in the sense that any language recognised by an NFA is also recognised by a DFA (and the reverse is immediate). These languages are called *regular*. However, this equivalence hides a

size gap, as an NFA can be exponentially more succinct than a DFA. To illustrate that, consider the previous automaton from Example 1.3. The general version of this automaton uses $n+1$ states if we replace "third to last" with "n^{th} to last", but the smallest equivalent DFA actually needs 2^n states, as it must somehow "store" the last n letters read, which corresponds to 2^n different possibilities, each of which have a different set of accepted suffixes.

In the following, we will define a way to describe languages, called regular expressions, and detail their privileged relation with NFAs (rather than DFAs), which explains how NFAs can be seen as an intermediary point between language and DFAs.

Definition (Regular expression)**:**
A *regular expression* over an alphabet Σ is an expression generated by the following grammar:

$$e, f := a \mid e + f \mid e \cdot f \mid e^*$$

where a is any letter from Σ. In other words, letters from Σ are regular expressions, and if we take two regular expressions e and f, we can create new ones by taking $e + f$, $e \cdot f$ and e^*. These three operations are respectively called union, concatenation, and Kleene star.

Any regular expression e represents a language $\mathcal{L}(e)$ that is defined inductively using the following rules.

$$\mathcal{L}(a) = \{a\} \qquad \mathcal{L}(e + f) = \mathcal{L}(e) \cup \mathcal{L}(f) \qquad \mathcal{L}(e \cdot f) = \mathcal{L}(e) \cdot \mathcal{L}(f) \qquad \mathcal{L}(e)^* = \bigcup_{n \in \mathbb{N}} \mathcal{L}(e)^n$$

These use the concatenation of languages, which is defined, given two languages L and M, by $L \cdot M = \{uv \mid (u, v) \in L \times M\}$ (uv is created by adding the word v at the end of u). The iterated concatenation is defined inductively by $L^0 = \{\varepsilon\}$ and $L^{n+1} = L^n \cdot L$.

At that point, we can note that these expressions contain a form of symmetry that NFAs lack, in the sense that the reading order for a word is not as important from the point of view of the expression. If a language can be defined using a regular expression, then the mirror language, which contain the same words but read in the opposite order, can also be described using a similar expression, defined inductively.

One folklore result states the equivalence between NFAs and regular expressions: any language recognised by one of these can also be recognised by the other (*i.e.* regular expressions describe regular languages). Moreover, the transformations are polynomial, meaning that these two representations have the same succinctness.

Example 1.4:
The expression $(a+b)^* \cdot a \cdot (a+b) \cdot (a+b)$ corresponds to the same language as in Example 1.3, *i.e.* words over $\Sigma = \{a, b\}$ that have the letter a in the third to last position. We sometime replace the union of all letters of the alphabet with the alphabet itself, and remove the concatenation operator for simpler expressions: $\Sigma^* a \Sigma \Sigma$.

1.4 Infinite words

When we want to describe a machine that is expected to work forever, the notion of automaton as described before becomes restrictive, as it only allows for finite inputs. This is one reason to consider the notion of infinite words.

Definition (Infinite word)**:**
An *infinite word*, or *ω-word* (used to differentiate those from words over bigger ordinals, which are described in Section 3.3.1), is a labelling of $\mathbb{N}$ with letters from an alphabet Σ, *i.e.* a function $\mathbb{N} \to \Sigma$. The i^{th} letter of a word w (starting at 0 by convention) is still denoted w_i.

The language of all infinite words over Σ is denoted Σ^ω.

Example 1.5:
We often use a notation with suspension points to represent these words, for instance "*abababab*..." for the word with an infinite alternation of a and b. The general aspect of an ω-word w is $w_0 w_1 w_2 \dots$

To describe languages of such words, we use the following expressions.

Definition (ω-regular expression)**:**
To describe languages of such words, we use *ω-regular expressions*, which are defined by the grammar:

$$g, h := e^\omega \mid g + h \mid e \cdot g$$

where e stands for any regular expression. The operators $+$ and $\cdot$ are interpreted similarly as before, although in the case of $\cdot$ we now have an asymmetry, as we require that the first word be a finite one. The expression e^ω corresponds to an infinite concatenation of words from $\mathcal{L}(e)$, *i.e.* the words in $\mathcal{L}(e)$ are those that can be written $w^1 \cdot w^2 \cdot w^3 \dots$ with $w^i \in \mathcal{L}(e)$ for any $i \in \mathbb{N}$. More formally, this corresponds to functions $w : \mathbb{N} \to \Sigma$ such that there is an increasing sequence of integers $(k_i)_{i \in \mathbb{N}}$ satisfying $w_{k_i} \dots w_{k_{i+1}-1} \in \mathcal{L}(e)$ for any $i \in \mathbb{N}$.

Remark 1.1:
Note that to avoid the possibility of a finite (or even empty) word in $\mathcal{L}(e^\omega)$, one might require that $\varepsilon \notin \mathcal{L}(e)$. If we wanted to allow these instead, we would need to change the "increasing sequence" above to a condition saying that the limit of the sequence corresponds to the length of the word. To avoid this problem altogether, what we will do later in Chapter 3 is change the grammar of regular expressions to get rid of ε entirely.

With these ω-words defined, we can introduce deterministic and non-deterministic automata reading them in a similar way as the ones used for finite words.

Definition (ω-automaton)**:**
An *ω-automaton* is a tuple $\mathcal{A} = (\Sigma, Q, q_0, \Delta, F)$ where Σ, Q, q_0 and Δ play the same roles as for an NFA to create infinite runs for a given infinite word. F is still called the set of accepting states. We say that $\mathcal{A}$ is deterministic if, for any $a \in \Sigma$ and $q \in Q$, there is at most one p such that $(q, a, p) \in \Delta$. In that case, we can use the δ notation instead.

Formally, a run of $\mathcal{A}$ on $w = w_1w_2w_3\dots$ is an infinite sequence $q_0, q_1, q_2\dots$ where q_0 is the initial state of $\mathcal{A}$ and for any $i \in \mathbb{N}$, $(q_i, w_{i+1}, q_{i+1}) \in \Delta$ (or $\delta(q_i, w_{i+1}) = q_{i+1}$ in the deterministic version).

A word is accepted by $\mathcal{A}$ if there is an accepting run (which is defined below) for that word, and the language of $\mathcal{A}$ is the set $\mathcal{L}(\mathcal{A})$ of those accepted words. There are several options for the acceptance condition:

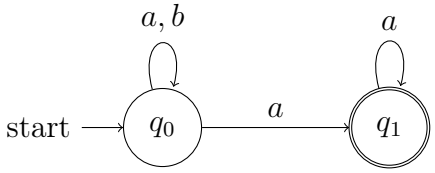
- *Reachability automaton*: A run is accepting if an accepting state is reached.
- *Safety automaton*: A run is accepting if no non-accepting state is ever reached.
- *Büchi automaton*: A run is accepting if at least one accepting state (or Büchi state) is seen infinitely often.
- *Co-Büchi automaton*: A run is accepting if every non-accepting state (or co-Büchi state) is seen finitely often.
- *Parity automaton*: Here F is replaced with a function p that associate an integer value to each state, called its priority. A run is accepting if the lowest priority of a state seen infinitely often is even.

Note that there are a few other common acceptance conditions out there, but these are the only ones that we will be using here.

The following property provides a link between the models of ω-automaton and ω-regular expression.

Prop 1.1:
Non-deterministic ω-automata (with any acceptance condition) describe the same languages as ω-regular expressions. We called those languages *ω-regular*.

Example 1.6:
Here is a non-deterministic Büchi automaton, accepting words over $\{a, b\}$ containing only the letter a after some point. A corresponding expression is given by $(a + b)^*a^\omega$.



Note that we still use the double circle to mark Büchi states here.

One can also notice that there is no equivalent deterministic Büchi automaton. This illustrates the fact that, contrarily to the case of finite words, there is not necessarily an equivalence between deterministic and non-deterministic models here. This equivalence however does hold for some models, such as parity automata.

1.5 Games

This section gives some definitions concerning games, which will hold a central role in Chapter 2. This notion is used to formalise the idea of an adversary, who often plays the role of the input when we talk about non-determinism.

Definition (Arena, play)**:**
Given two players A and B, an *arena* for those players is a set of positions $P = P_A \uplus P_B$ along with a set of possible *moves* (or transitions) $T \subseteq P \times P$, and an initial position $p_0 \in P$.

A *play* in the arena is a (possibly infinite) word over P, starting with p_0 and such that any two consecutive positions are linked by a transition from T.

Definition (Game)**:**
A *game* is given by an arena and a winning condition, which is a partition of the plays into those won by A and those won by B.

Note that this definition does not leave room for draws: any play is won by one of the players. The winning condition can for instance be expressed as a reachability objective for A, *i.e.* saying that A wins the game if any state from some given set is reached (implying that B wins otherwise).

Example 1.7:
In a game of chess, the players can be called Black and White, and the positions are given by the configuration of the board, *i.e.* the positions held by each piece. The initial position is in P_{White}, and the moves correspond to the allowed movements. Each move goes from one of $P_{\text{Black}}, P_{\text{White}}$ to the other. To simplify, we forget about the rules enforcing progression of the game, and we consider that White wins if and only if Black cannot play while his king is in check (*i.e.* every case of draw is now winning for Black, to resolve the issue mentioned above).

With this setup, the winning plays for White are those ending with a $p \in P_{\text{Black}}$ that is a starting point for no move (for any p', $(p, p') \notin T$).

Remark 1.2:
We will sometimes describe the games in a more informal manner, by explaining verbally the possible moves for each player. In some instances, we only describe the position at the beginning of a "turn", meaning that we go from a position to the next by letting A play then B answer, for instance. Such a description corresponds to an implicit arena, where we add the positions (p, t) for any position p and move t allowed from p for the first player.

Definition (Strategy)**:**
Consider a game G with positions P, including the initial position p_0, and moves T. A *strategy* for player X is a function σ from $P^* \times P_X$ to P (P^* being the set of finite words over P). Intuitively, $\sigma(w, p)$ gives a move for X when the play has visited the positions from $w \in P^*$ before reaching $p \in P_X$. It must use the transitions, meaning that $(p, \sigma(w, p)) \in T$.

A play $p_0 p_1 \dots$ is *consistent* with a strategy σ for X if, for any i such that $p_i \in P_X$,

we have $p_{i+1} = \sigma(p_0 \dots p_{i-1}, p_i)$.

A strategy for X is *winning* if all plays consistent with that strategy are won by X.

We say that a player *wins* a game if he has a winning strategy for that game.

As said before, those formal definitions will often in practice be replaced with a verbal description of the way the player should react to the moves of the adversary, instead of defining the function itself.

Several classes of games are known to be *determined*, meaning that they always have a winner (*i.e.* one of the players has a winning strategy). This is for instance the case for games with finite arena and either finite plays or an ω-regular winning condition. More generally, any game with a Borel winning condition is determined [GS16]. This includes most known games, and in particular all of those presented in this thesis.

Example 1.8 (Nim game)**:**
The Nim game with n matches is played by letting players alternatively take between 1 and 3 matches to remove them from a stack initially containing n matches. The player who removes the last one loses. We can describe this game formally, using the set of positions $[1, n] \times \{A, B\}$ with the initial position (n, A). The moves are those going from (i, A) to $(i - j, B)$ or from (i, B) to $(i - j, A)$, with $i \in [2, n]$ and $j \in [1, 3]$. The second component indicates whose turn it is, *i.e.* $P_X = [1, n] \times P_X$ for $X \in \{A, B\}$. The plays won by A are therefore those ending in $(1, B)$.

This game has a winning strategy for one of the players, depending on the initial number of matches: if $n = 4k + 1$ for some k, then B wins by ensuring the preservation of that property after each of his moves (he takes $4 - j$ matches if A took j). Otherwise, A can win using the same strategy.

Formally, the strategy for B in the $4k+1$ matches game can be written as $\sigma(w, (i, B)) = (i - j, A)$ if $i = 4k + 1 + j$ with $j \in [1, 3]$. The values on other inputs do not matter, as those will never be used.

Note that here σ does not use the memory of the play w, but only the current position (i, B). This is what we call a *memoryless* strategy, which takes only the current position as an input. A game with such winning strategies is called *positionally determined*.

There are also *finite memory* strategies, which take as input the current position and an element m from a finite set M, called memory. The value of m is updated after each move using some function associated to the strategy.

1.6 (Cyclic) proofs

The subject of this section will no longer concern languages, automata or games. Here we will instead talk about the notion of proof. We will start from the intuitive notion, originating in the geometric reasoning that most people will know from early school, and work our way up to the abstract notion of proof, which we will then generalize to give the intuition of what a cyclic proof might look like. Note that this part only concerns Chapter 3: it is not a prerequisite to Chapter 2.

Let us first look at the kind of reasoning one might see in a proof from Greek mathematics. We can for instance consider the following sentence: “If the line A is perpendicular

to the line B, which in turn is perpendicular to C, then A must be parallel to C". This sentence contains two hypotheses: $A \perp B$ (A is perpendicular to B) and $B \perp C$, and one conclusion: $A \parallel C$ (parallel). It uses the deduction rule "Two lines perpendicular to a third one are parallel", which we call `pepepa`. The sentence can then be summarized into the following representation:

$$\frac{A \perp B \quad B \perp C}{A \parallel C}\ \texttt{pepepa}$$

where the hypotheses are written above the conclusion, meaning that the reasoning is proceeding downwards. This presents the advantage of allowing the representation of a sequence of such arguments. For instance, if we knew that $B \perp C$ because $B \perp D$ and $D \parallel C$, (rule that we call `pepape`), we get this:

$$\frac{A \perp B \qquad \dfrac{B \perp D \quad D \parallel C}{B \perp C}\ \texttt{pepape}}{A \parallel C}\ \texttt{pepepa}$$

We can keep building a proof that way, using statements that we know to be true at the top, and rules such as `pepepa` and `pepape` to get to the conclusion at the bottom. This creates a tree structure, which we can formally define as follows.

Definition (Tree)**:**
A (finite) *tree* of degree k can be defined as a (finite) non-empty prefix-closed subset of $[0, k-1]^*$, which is the set of finite words over $[0, k-1] = \{0, 1, 2, \ldots, k-1\}$. The elements of a tree will be called nodes. The empty word ε is necessarily a node of any tree, and is called the root.

Given a tree $\mathcal{T}$ of degree k and a node $v \in \mathcal{T}$, any node $vn \in \mathcal{T}$ for $n \in [0, k-1]$ is called a parent of v, and v is its child. A node with no parents is called a *leaf*.

Remark 1.3:
The notions of parent and child in a tree are sometimes the opposite version of what we define here, but this definition is more natural from a proof tree point of view, as the root will hold the conclusion, and therefore should come last.

Another thing to note is that we will mostly be interested in binary trees in this thesis, *i.e.* $k = 2$. However, general proof can use any value.

A proof tree will be defined as a tree with some additional information. To be able to talk about this information, we need to define the following elements.

Definition (Sequent, abstraction)**:**
We generally define a *sequent* as a couple of two (possibly empty) lists of expressions, usually called left side (or antecedents), and right side (or succedents, consequents). The expressions are usually provided by some grammar, and the whole sequent will be denoted $e_1, \ldots, e_n \vdash f_1, \ldots, f_m$.

The interpretation of such a sequent may vary, but the standard one will be: "If e_i is true for any i, then f_j is true for some j". This is defined using a boolean interpretation for the expressions. We say that a sequent is *sound*, or *correct*, if this interpretation holds.

We define informally an *abstraction* as a pattern that can be matched to some sequents, using typed variables that correspond to either lists or expressions. A sequent S is matched by an abstraction A if some valuation ν of the variables (respecting the types) transforms the A into S: we note $S = \nu(A)$.

Usually we will use upper case Greek letters for variables of lists, and lower case Latin letters for expressions.

Remark 1.4:
Later in Chapter 3, we will change the shape of sequents, replacing $\vdash$ with $\to$ and changing the shape of the right side. The first change is simply made to be consistence with the literature, but one justification is the Curry-Howard correspondence between proofs and functional programs. The second change is due to our need for more intricate sequents that store additional information.

We can now use these elements to define the notions of proof system and proof tree.

Definition (Proof system, proof tree)**:**
Given a grammar $\mathcal{G}$ of sequents, a (well-founded) *proof system* for these sequents is given by a set of *inference rules*. These rules are tuples that can be represented as follows:

$$\frac{H_1 \quad \cdots \quad H_n}{C}\ r$$

r is the name of the rule, $C, H_1, \ldots, H_n$ are abstractions with a shared set of variables, and C is called the conclusion of r while the others are its hypotheses. If $n = 0$, then r is called an *axiom*.

A (well-founded) *proof tree* is then given by a finite tree T, along with a labelling of each node with both a sequent and a rule name. Moreover, for each node v, if we call r the rule name at v, S_0 the sequent of v, and $S_1, \ldots, S_m$ those of its parents, then we can represent this part of the tree as:

$$\frac{S_1 \quad \cdots \quad S_m}{S_0}\ r$$

and the rule r is given by the system as described above. This rule must be observed, *i.e.* we require $n = m$, and there has to be a valuation ν such that $\nu(C) = S_0$ and $\nu(H_i) = S_i$ for any $i \in [1, n]$. Informally, this corresponds to using the rules as "bricks" to build the proof tree.

We say that a sequent is *derivable* in a system is there is a proof tree from that system with the sequent at the root.

Note that the example above can be seen as trees using such rules, where we omitted the symbol $\vdash$ since we only considered sequents written $\vdash f$. This can be extended to allow for expression on the left side, see Example 1.9 below.

Example 1.9:
We can create the following system:

$$\frac{\Gamma \vdash X \perp Y \quad \Gamma \vdash Y \perp Z}{\Gamma \vdash X \shortparallel Z}\ \texttt{pepepa} \qquad \frac{\Gamma \vdash X \perp Y \quad \Gamma \vdash Y \shortparallel Z}{\Gamma \vdash X \perp Z}\ \texttt{pepape}$$

$$\frac{\Gamma \vdash X \parallel Y \qquad \Gamma \vdash Y \parallel Z}{\Gamma \vdash X \parallel Z}\ \texttt{papapa} \qquad \frac{\Gamma \vdash X \parallel Y}{\Gamma \vdash Y \parallel X}\ \texttt{papa} \qquad \frac{\Gamma \vdash X \perp Y}{\Gamma \vdash Y \perp X}\ \texttt{pepe}$$

But then we cannot create any proof tree because there is no way to end a branch. However, if we add the following axiom rule:

$$\frac{}{\Gamma, f, \Delta \vdash f}\ \texttt{ax}$$

then we can for instance build this proof tree, where Γ is the list containing $A \parallel B$, $B \perp C$ and $C \parallel D$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash B \perp C}\ \texttt{ax}}{\Gamma \vdash C \perp B}\ \texttt{pepe} \qquad \dfrac{\dfrac{}{\Gamma \vdash A \parallel B}\ \texttt{ax}}{\Gamma \vdash B \parallel A}\ \texttt{papa}}{\Gamma \vdash C \perp A}\ \texttt{pepape}}{\Gamma \vdash A \perp C} \qquad \dfrac{}{\Gamma \vdash C \parallel D}\ \texttt{ax}}{\Gamma \vdash A \perp D}\ \texttt{pepape}$$

where each element of Γ is used at an `ax` rule.

This system from Example 1.9 gives some tools to formally prove geometrical results using some axioms. However, we only talked about the syntactic aspect so far, and said nothing concerning the semantic aspect, *i.e.* what "$A \perp B$" really means. In other words, the system would work the same way if we replaced $\parallel$ by ☺ and $\perp$ by ☹. To make a link between a proof system and mathematical truth, we now need to provide a link between the interpretation of sequents and their derivability. This is done using the following notions.

Definition (Soundness, completeness)**:**
Given a proof system $\mathcal{S}$ with an interpretation for sequents, we say that $\mathcal{S}$ is *sound* if every derivable sequent is sound. Conversely, we say that it is *complete* if every sound sequent is derivable.

In the case of finite proof trees (as opposed to the infinite ones we will describe later), the soundness is rather straightforward to prove, using the following property, which is obtained through a simple induction on the height of the tree.

Prop 1.2:
A proof system is sound if and only if every rule is sound, *i.e.* any valuation that produces sound hypotheses also gives sound conclusion.

The proof of completeness (when it does hold) is often harder in those systems. One way to go about it is to use a (complete) algebraic axiomatization of the structure described by the sequent. One can then show that every axiom can be proven, and that their consequences therefore are derivable. We will not give many details here, as it is not the method we will use later.

We will now talk about cyclic proofs, but let us first look at the following example.

Example 1.10:
We consider the types of lists and of lists of zeros, constructed using the empty list () and the concatenation symbol "::". The inductive definition is:

$\texttt{list}\ ()$, if $\texttt{list}\ l$ then $\texttt{list}\ n :: l$, $\texttt{0list}\ ()$, if $\texttt{0list}\ l$ then $\texttt{0list}\ 0 :: l$.

where "$\texttt{list}\ l$" should be interpreted as "l is a list". We can then define the following rules, in a system where sequents are written $\Gamma \vdash A$, to be interpreted as "the formula A is a consequence of the set of formulas Γ".

$$\frac{}{\Gamma \vdash \texttt{list}\ ()}\ \texttt{init list} \qquad \frac{\Gamma \vdash \texttt{list}\ l}{\Gamma \vdash \texttt{list}\ n :: l}\ \texttt{iter list}$$

$$\frac{\Gamma, l = () \vdash A \quad \Gamma, l = n :: l', \texttt{list}\ l' \vdash A}{\Gamma, \texttt{list}\ l \vdash A}\ \texttt{case list}$$

We also add similar rules `init 0list`, `iter 0list` and `case 0list`, where n is replaced with 0. With that and some reasonable deduction rules (mainly to deal with equalities), we can for instance write these two proof trees:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \texttt{list}\ ()}\ \texttt{init list}}{\Gamma \vdash \texttt{list}\ 1 :: ()}\ \texttt{iter list}}{\Gamma \vdash \texttt{list}\ 2 :: 1 :: ()}\ \texttt{iter list}$$

$$\dfrac{\dfrac{}{n :: l = () \vdash n = 0}\ \texttt{nomatch} \quad \dfrac{\dfrac{}{n = 0, l = l', \texttt{list}\ l' \vdash n = 0}\ \texttt{ax}}{n :: l = 0 :: l', \texttt{list}\ l' \vdash n = 0}\ \texttt{match}}{\texttt{0list}\ n :: l \vdash n = 0}\ \texttt{case 0list}$$

However, the proofs we can do here with simple rules are rather limited. To prove something like $\texttt{0list}\ l \vdash \texttt{list}\ l$, we would need some notion of induction, for instance the rule:

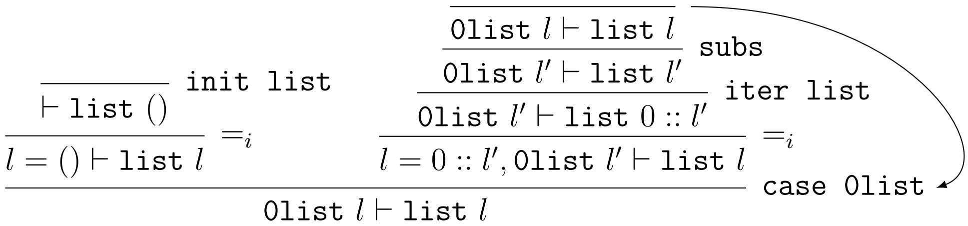
$$\frac{\texttt{0list}\ () \vdash \texttt{list}\ () \quad (\texttt{0list}\ l \vdash \texttt{list}\ l) \vdash (\texttt{0list}\ n :: l \vdash \texttt{list}\ n :: l)}{\texttt{0list}\ l \vdash \texttt{list}\ l}\ \texttt{indu}$$

Instead of adding this rule, we will look at what happens when we switch to cyclic proofs, once we have defined those.

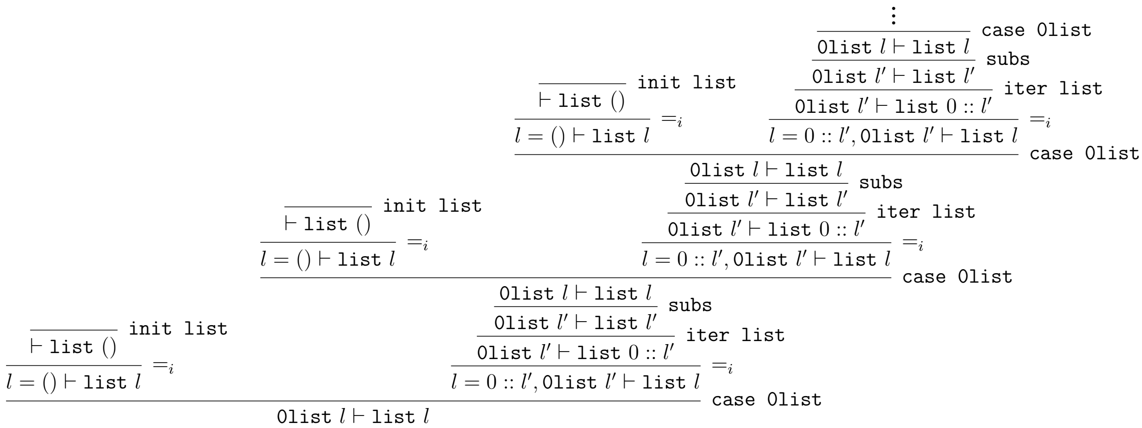
Definition (Preproof)**:**
Given a set of inference rules, a *preproof* is defined the same way as a finite proof tree, except that we now remove the finiteness condition on the tree.

We say that a preproof is *cyclic*, or *regular*, if it has finitely many different labelled subtrees. In that case we can represent it using a finite tree with back links as shown in Example 1.11 below.

Example 1.11:
We can consider the following preproof using the previous rules (the substitution rule `subs` simply lets us rename variables using unused names, and $=_i$ uses an equality to replace expressions):

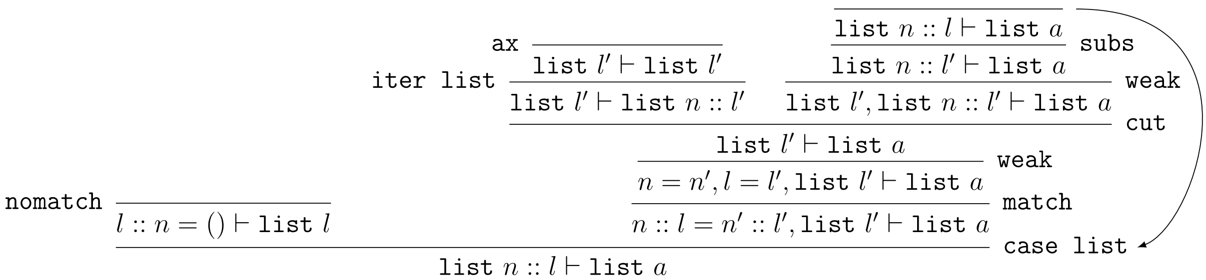


This actually represents the following infinite tree:



Intuitively, this preproof is creating a finite branch for every possible length of the list: for length n, this branch goes n times to the right at the `case list` rule, then to the left. This is another way of writing an inductive reasoning, by implementing it within the proof structure instead of as an axiom.

However, we can also use this system to write proofs such as this one:



This is obviously a behaviour that needs to be forbidden, as the root sequent is not sound. Note that this preproof uses the rules `weak` (weakening the left side of the sequent) and `cut` (corresponding to the *modus ponens*, which says $((A \Rightarrow B) \wedge A) \Rightarrow B)$, but we will later give examples of unwanted preproofs that do not even require this (in Example 1.12 for this system, or later with another system in Figure 3.2).

In order to avoid proving such a statement using a circular reasoning, cyclic proof systems generally come with a validity condition that ensures that every infinite branch behaves correctly.

Definition (Validity condition, non-well-founded proof system)**:**
A *validity condition* is a condition on infinite branches of a system.

A *non-well-founded proof system* is given by a set of rules along with a validity condition. Its proofs are every preproofs using these rules and satisfying the validity condition.

Sometimes we restrict that scope to cyclic preproofs. We can extend the notion of cyclicity (or regularity) to proofs, *i.e.* valid preproofs.

The notion of derivability and the soundness and completeness of such a system can be defined as before.

Remark 1.5:
Note that this kind of system does not exclude finite proofs, for which the validity condition is empty. We can for instance imagine a non-well-founded system that extends a well-founded one, adding some rules and a validity conditions, and thus preserving the proofs from the original system.

The validity condition is often formulated as an ω-regular condition on branches. This makes validity easier to decide, and the system can then be manipulated using algorithmic tools.

Example 1.12:
In the `list` system described in Example 1.10 and Example 1.11, we can informally take the validity condition saying that in any infinite branch, there must be a list that decreases in size infinitely many times. This is what happens to the list l in the first preproof, but it does not occur in the preproof for $\texttt{list}\ n :: l \vdash \texttt{list}\ a$, meaning that this unwanted preproof is indeed caught by the validity condition, as we would hope.

We can also extend this example by adding co-inductive notions, such as streams:

$$\frac{\Gamma \vdash \texttt{stream}\ s}{\Gamma \vdash \texttt{stream}\ n :: s}\ \texttt{iter stream} \qquad \frac{\Gamma, s = n :: s', \texttt{stream}\ s' \vdash A}{\Gamma, \texttt{stream}\ s \vdash A}\ \texttt{case stream}$$

This describes infinite lists (or ω-words), and the natural way to prove something on such a structure is to use infinite branches in a preproof. We can for instance write a similar proof to the one above for the case of streams:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\texttt{0stream}\ s \vdash \texttt{stream}\ s}}{\texttt{0stream}\ s' \vdash \texttt{stream}\ s'}\ \texttt{subs}}{\texttt{0stream}\ s' \vdash \texttt{stream}\ 0 :: s'}\ \texttt{iter stream}}{s = 0 :: s', \texttt{0stream}\ s' \vdash \texttt{stream}\ s}\ =_i}{\texttt{0stream}\ s \vdash \texttt{stream}\ s}\ \texttt{case 0stream}$$

However we can also write the following preproof:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\texttt{stream}\ s \vdash \texttt{list}\ s}}{\texttt{stream}\ s' \vdash \texttt{list}\ s'}\ \texttt{subs}}{\texttt{stream}\ s' \vdash \texttt{list}\ n :: s'}\ \texttt{iter list}}{s = n :: s', \texttt{stream}\ s' \vdash \texttt{list}\ s}\ =_i}{\texttt{stream}\ s \vdash \texttt{list}\ s}\ \texttt{case stream}$$

Thankfully we can make this last preproof invalid by taking the usual approach to the validity condition, which is more precise than the version described above. Along an infinite branch, we want to see an *inductive* object being unfolded infinitely many times on the *left* side of the sequents, or a *co-inductive* object being unfolded infinitely many times on the *right* side of the sequents.

This duality can be understood as follows. In the first case, we unfold an inductive structure on the antecedents side, and each resulting finite branch (taking the "initializing" step at some point) corresponds to a proof for one instance of the hypothesis. The infiniteness of the unfolding guaranties that each instance is considered. On the other hand, the second case is that of an unfolding of a co-inductive structure on the consequents side, so we need to make sure that the proof visits the whole object to prove the result, hence the other part of the validity condition. A more theoretical approach would be to look at the proof theory of fixed-point logics, which makes explicit the duality between the two sides of a sequent, that goes hand in hand with the duality between inductive and co-inductive objects.

1.7 Context and contributions

In this section, we give the main elements of context regarding each chapter, along with the main results we brought. For a more thorough introduction to each of them, please refer to the corresponding chapter introduction: Section 2.1 or Section 3.1. Note that this does not reflect the chronology of the research in this thesis: the work presented in Chapter 3 is anterior to that of Chapter 2.

Chapter 2 concerns a notion of partial non-determinism in automata that we introduce here, called explorability. This is rather similar to the notion of width introduced in [KM19]. Both can be defined using games in which a player chooses the input while the other moves tokens in the automaton to find accepting runs. However, in the case of width a token can "jump" to the position of another token, which is not possible for explorability. A consequence of that difference is that the width of an automaton is bounded by its number of states, while the number of tokens required to explore the same automaton can be infinite, and even uncountable in the case of infinite words. This gives an interest to the notion of explorable automata, which are those where a finite number of tokens is enough for exploration.

The main motive to study explorability is its link to Good-For-Games (GFG) automata. These correspond to 1-explorable automata: there is a strategy to build an accepting run on the fly. GFG automata behave like deterministic ones when composed with games, which leads to an important application for the reactive Church synthesis problem.

This problem consists in finding a reactive system (a transducer) that follows a given specification. It can be represented as a game between a player choosing the input and another answering with the output. The output player aims to respect the specification. If he has a winning strategy, then it provides a solution to the Church synthesis problem (and otherwise there is none). See *e.g.* [Tho08] for more details about this algorithm.

If we have a deterministic automaton for the specification, then we can combine this automaton with the game, to get a new game that is equivalent but easier to solve (quasi-polynomial time for a parity condition, or polynomial for fixed parity). This then gives a solution to the Church synthesis problem, if it exists. The interest of GFG automata is that they can actually be used instead of deterministic ones in this process. Since they can be exponentially more succinct, this can greatly improve the complexity of the

algorithm when we know that the specification automaton is GFG. The problem that naturally arises from here is that of recognizing GFG automata.

This new problem still contains several open questions. For instance, the only known algorithm for deciding whether a parity automaton is GFG is the naive one, which is ExpTime, with no matching lower bound. There is however a polynomial algorithm (which we will describe) solving that problem once we know that the automaton we consider is explorable. This gives the initial motivation for our interest in the problem of deciding explorability. Although this hope of improving existing algorithms for recognizing GFG automata was not satisfied, we did get results that might help better understand the notion of non-determinism.

We provide ExpTime lower and upper bounds for both the problem of explorability and the problem of ω-explorability (finitely or countably many tokens), in addition to the PTime algorithm mentioned above to recognize GFG automata when we know the input to be explorable. We do not consider the problem of k-explorability, where a finite number of tokens is fixed, as it is mostly dealt with using the same proofs as for width. Precisely, [KM19] provides an optimal ExpTime algorithm to decide whether the width of an automaton is at most k, and this can be used to get a 2-ExpTime algorithm for k-explorability (ExpTime if k is at most polynomial in the size of the automaton), along with the preserved ExpTime lower bound.

Chapter 3 tackles another problem, which is that of the inclusion of languages described by expressions. This problem is strongly correlated with non-determinism: deciding whether $A \subseteq B$ basically amounts to solving the non-determinism in the description (automaton or expression) of B. In particular, if B is given by a GFG automaton, then the problem can be reduced to a parity game in which one player chooses a word of A and builds a corresponding accepting run, while the other must build an accepting run in the GFG automaton for B. Solving such a game can then be done in quasi-polynomial time. This reasoning could also be applied to some extent to k-explorable automata, although the bound is less satisfying as the resulting game has a more complicated winning condition.

In this thesis however, instead of the usual automaton approach, we look at this problem from the point of view of proof theory. This has already been considered to some extent by [DP17] and [DP18]. These articles present proof systems that provide certificates of inclusion for regular expressions. In other words, given two expressions e and f, the language inclusion $\mathcal{L}(e) \subseteq \mathcal{L}(f)$ holds if and only if there is a proof for it in those systems ($\Rightarrow$ is the completeness of the system, and $\Leftarrow$ is its soundness). These proofs are infinite, with a conclusion that can be derived from a chain of deductions with no beginning. Among those proofs, some can be represented finally as the unfolding of a finite graph, and thus can be used for an algorithmic application.

We first generalize these systems to a version suited to ω-regular expressions. For this, we add the operator $\cdot^\omega$, which generates infinite concatenations of words. This operator can be defined as a greatest fixed-point, while the Kleene star operator $\cdot^*$ used in regular expressions is a smallest fixed-point. This provides a justification for adding $\cdot^\omega$, as it behaves in a dual way to the Kleene star, and thus is not expected to create complicated new phenomena. Going from regular to ω-regular expressions does however require some adaptations to the proof system, mainly due to the fact that it now has to solve non-

determinism on limit behaviours. For instance, the inclusion $\mathcal{L}((a+b)^\omega) \subseteq \mathcal{L}((b^*a)^\omega + (b^*a)^*b^\omega)$ intuitively requires knowing whether an infinite word over the alphabet $\{a, b\}$ contains finitely or infinitely many times the letter a. This is dealt with by adding more structure to sequents, creating what is sometimes called hypersequents. The presented proof system is shown to be sound and complete, even when restricted to its regular fragment (*i.e.* cyclic proofs are enough).

We then tried to extend these results to more general expressions, in which the operator $\cdot^\omega$ is no longer restricted to a single use at the end. We allow to keep writing after $\cdot^\omega$, and even to nest them within each other (*e.g.* to write $(a^\omega)^\omega$). Such expressions describe languages of transfinite words, indexed by ordinals smaller than ω^ω (for instance, $(a^\omega)^\omega$ has length ω^2). Our first intuition was to simply use the rules from the previous system with almost no change to get one for the transfinite case. The reason for that is the fact that we did not add any new operator when going from infinite words to transfinite ones. However, this approach turned out to be unable to deal with the amount of non-determinism in transfinite expressions. The issue there is basically the fact that an expression can yield words of very different lengths, and it is therefore hard to cut a couple of expressions into matching sub-expressions to deal with them separately. This is why we ended up with a system in which the proofs are no longer trees, but instead forests of trees, that can interact together. Those proof forests embed transfinite descent in the same way as proof tree embed infinite descent (a proof tree can be used several times in such a reasoning, just as a same sequent could be used several times in an infinite descent).

We then prove this new system to be sound and complete, even when restricted to its regular fragment, *i.e.* to finite forests of cyclic trees. We use this system to build a PSPACE proof search algorithm deciding the inclusion of languages represented by those transfinite expressions. We can also verify a proof with a similar PSPACE algorithm. Note that this seemingly bad complexity for proof check is actually matched by several common cyclic proof systems (see *e.g.* [NST19; Das18]), making it unsurprising.

Chapter 2

Explorability

I could have added some inspired quote about exploration here.
You know, those with retouched pictures in the background.
Just be thankful I did not.

Émile Hazard

In this part, we define the class of explorable automata on finite or infinite words. This is a generalization of Good-For-Games (GFG) automata, where this time non-deterministic choices can be resolved by building finitely many simultaneous runs instead of just one. We show that recognizing GFG parity automata of fixed index among explorable ones is in PTime, thereby giving a strong link between the two notions. We then show that recognizing explorable automata is ExpTime-complete, in the case of finite words or Büchi automata. Additionally, we define the notion of ω-explorable automata on infinite words, where countably many runs can be used to resolve the non-deterministic choices. We show that all reachability automata are ω-explorable, but this is not the case for safety ones. We finally show ExpTime-completeness for ω-explorability of automata on infinite words for the safety and co-Büchi acceptance conditions.

2.1 Introduction

In several fields of theoretical computer science, the tension between deterministic and non-deterministic models is a source of fundamental open questions, and has led to important lines of research. The most famous of this kind is the P vs NP question in complexity theory. This paper aims at further investigating the frontier between determinism and non-determinism in automata theory. Although Non-deterministic and Deterministic Finite Automata (NFA and DFA) are known to be equivalent, many subtle questions remain about the cost of determinism, and a deep understanding of non-determinism will be needed to solve them.

One of the approaches investigating non-determinism in automata is the study of Good-For-Games (GFG) automata, introduced in [HP06]. An automaton is GFG if, when reading input letters one by one, its non-determinism can be resolved on-the-fly

without any need to guess the future. This constitutes a model that is intermediary between non-determinism and determinism, and can sometimes bring the best of both worlds. Like deterministic automata, GFG automata on infinite words retain good properties such as their soundness with respect to composition with games, making them appropriate for use in Church synthesis algorithms [HP06]. On the other hand, like non-deterministic automata, they can be exponentially more succinct than deterministic ones [KS15]. There is a very active line of research trying to understand the various properties of GFG automata, see *e.g.* [AK21; AKL21; Bok+20; BL22; LZ20; CF19; Sch20] for latest developments. Notice that GFG automata are also called *history-deterministic*, a terminology introduced originally in the theory of regular cost functions [Col09]. The name "history-deterministic" corresponds to the above intuition of solving non-determinism on-the-fly, while "good-for-games" refers to sound composition with games. These two notions may actually differ in some quantitative frameworks, but coincide on boolean automata [BL21].

The goal of this chapter is to pursue this line of research by introducing and studying the class of explorable automata on finite and infinite words. The intuition behind explorability is to limit the amount of non-determinism required by the automaton to accept its language, in a more permissive way than GFG automata. If $k \in \mathbb{N}$, an automaton is k-explorable if when reading input letters, it suffices to keep track of k runs to build an accepting one, if it exists. An automaton is explorable if it is k-explorable for some $k \in \mathbb{N}$. This can be seen as a variation on the notion of GFG automaton, which corresponds to the case $k = 1$. The present work can be compared to [KM19], where a notion related to k-explorability (called *width*) is introduced and studied. In [KM19], the notion of simultaneous runs is different and more permissive, and does not give any meaningful notion of explorability, because n simultaneous runs always suffice for an automaton with n states. However, some results of [KM19] also apply to k-explorability, notably ExpTime-hardness of deciding k-explorability of an NFA if k is part of the input. The matching ExpTime algorithm from [KM19] can also be translated to a 2-ExpTime one for k-explorability. Surprisingly however, the techniques used in [KM19] are quite different from the ones we need here. This shows that fixing a bound k for the number of runs leads to very different problems compared to asking for the existence of such a bound.

One of the motivations to introduce the notion of explorability is to tackle one of the important open questions about GFG automata: what is the complexity of deciding whether an automaton is GFG? Recognizing GFG automata is known to be in PTime for Büchi [BK18] and co-Büchi [KS15] automata, but even for 3 parity ranks, the only known upper bound is ExpTime via the naive algorithm from [HP06]. We show how explorable automata can simplify this question: if the input automaton is explorable, then the problem becomes PTime. Therefore, the question of recognizing GFG automata can be shifted to: how hard is it to recognize explorable automata?

We then proceed to study the decidability and complexity of the explorability problem: deciding whether an input automaton on finite or infinite words is explorable. For this, we establish a connection with the population control problem studied in [Ber+19]. This problem asks, given an NFA with an arbitrary number of tokens in the initial state, whether a controller can choose input letters, thereby forcing every token to reach a

designated state, even if tokens are controlled by an opponent. It is shown in [Ber+19] that the population control problem is ExpTime-complete, and we adapt their proof to our setting to show that the explorability problem is ExpTime-complete as well, already for NFAs. We also show that a direct reduction is possible, but at an exponential cost, yielding only a 2-ExpTime algorithm for the NFA explorability problem. In the case of infinite words, we adapt the proof to the Büchi case, thereby showing that the Büchi explorability problem is in ExpTime as well. We also remark that, as in [Ber+19], the number of tokens needed to witness explorability can go as high as doubly exponential in the size of the automaton.

This ExpTime-completeness result means that we unfortunately cannot directly use the intermediate notion of explorable automata to improve on the complexity of recognizing GFG automata, as could have been the hope. We still believe however that this explorability notion is of interest towards a better understanding of non-determinism in automata theory.

Notice that interestingly, from a model-checking perspective, our approach is dual to [Ber+19]: in the population control problem, an NFA is well-behaved when we can "control" it by forcing arbitrarily many runs to simultaneously reach a designated state, via an appropriate choice of input letters. On the contrary, in our approach, the input letters form an adversarial environment, and our NFA is well-behaved when its non-determinism is limited, in the sense that it is enough to spread finitely many runs to explore all possible behaviours.

On infinite words, we push further the notion of explorability, by remarking that for some automata, even following a countable number of runs is not enough. This leads to defining the class of *ω-explorable automata*, as those automata on infinite words where non-determinism can be resolved using countably many runs. We show that ω-explorable automata form a non-trivial class even for the safety acceptance condition (but not for reachability), and give an ExpTime algorithm recognizing ω-explorable automata, encompassing the safety and co-Büchi conditions. We also show ExpTime-hardness of this problem, by adapting the ExpTime-hardness proof of [Ber+19] to the setting of ω-explorability.

Summary of the contributions. We show that given an explorable parity automaton of fixed parity index, it is in PTime to decide whether it is GFG. The algorithm used for Büchi in [BK18] is conjectured to work for any acceptance condition (this is the "G_2 conjecture"), and it is in fact this algorithm that is shown here to work on any explorable parity automaton.

We show that given an NFA or Büchi automaton, it is decidable and ExpTime-complete to check whether it is explorable. Our proof of ExpTime-completeness for NFAs uses techniques developed in [Ber+19], where ExpTime-completeness is shown for the NFA population control problem. We generalize this result to ExpTime explorability checking for Büchi automata, requiring further adaptations. We also give a black box reduction using the result from [Ber+19]. This is enough to show decidability of the NFA explorability problem, but it yields a 2-ExpTime algorithm. As in [Ber+19], the ExpTime algorithm yields a doubly exponential tight upper bound on the number of tokens needed to witness explorability.

On infinite words, we show that any reachability automaton is ω-explorable, but that

this is not the case for safety automata. We show that both the safety and co-Büchi ω-explorability problems are ExpTime-complete.

Related Works. Many works aim at quantifying the amount of non-determinism in automata. A survey by Colcombet [Col12] gives useful references on this question. Let us mention for instance the notion of ambiguity, which quantifies the number of simultaneous accepting runs. Similarly to [KM19], we can note that ambiguity is orthogonal to k-explorability. Remark however that our finite/countable/uncountable explorability hierarchy is reminiscent of the finite/polynomial/exponential ambiguity hierarchy (see *e.g.* [WS91]).

In [Hro+00], several ways of quantifying the non-determinism in automata are studied from the point of view of complexity, including notions such as the number of advice bits needed.

Another approach is studied in [PSA17], where a measure of the maximum non-deterministic branching along a run is defined and compared to other existing measures.

Following the GFG approach, a hierarchy of non-determinism and an analysis of this hierarchy via probabilistic models is given in [AKL21].

We define explorability via games with tokens, inspired by the approach in [BK18]. These games with tokens and their interplay with various quantitative acceptance conditions were recently investigated in [BL22].

2.2 Explorable automata

2.2.1 Preliminaries

We recall that, if $i \leq j$ are integers, we will denote by $[i, j]$ the integer interval $\{i, i + 1, \dots, j\}$. If S is a set, its cardinal will be denoted $|S|$, and its powerset $\mathcal{P}(S)$.

We work with a fixed finite alphabet Σ. We will use the following default notation for the components of an automaton $\mathcal{A}$: $Q_{\mathcal{A}}$ for its set of states, $q_0^{\mathcal{A}}$ for its initial state, $F_{\mathcal{A}}$ for its accepting states, $\Delta_{\mathcal{A}}$ for its set of transitions. The subscript might be omitted when clear from context. We might also specify its alphabet by $\Sigma_{\mathcal{A}}$ instead of Σ for cases where different alphabets come into play. If $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation, and $(p, a) \in Q \times \Sigma$, we will note $\Delta(p, a) = \{q \in Q, (p, a, q) \in \Delta\}$. If $X \subseteq Q$, we note $\Delta(X, a) = \bigcup_{p \in X} \Delta(p, a)$.

We will consider non-deterministic automata on finite words (NFAs). A run of such an automaton on a word $a_1 a_2 \dots a_n \in \Sigma^*$ is a sequence of states $q_0 q_1 \dots q_n \in Q^*$ (q_0 being the initial state), such that, for all $i \in [0, n-1]$, we have $q_{i+1} \in \Delta(q_i, a_{i-1})$. Such a run is accepting if $q_n \in F$, *i.e.* if the run belongs to Q^*F. As usual, the language of an automaton $\mathcal{A}$, denoted $L(\mathcal{A})$, is the set of words that admit an accepting run.

We will also deal with automata on infinite words, and we recall here some standard acceptance conditions for such automata. A run on an infinite word $w \in \Sigma^\omega$ is now an infinite sequence of states, *i.e.* an element of Q^ω, starting in q_0 and following as before transitions of the automaton according to the letters of w. Such a run of Q^ω is accepting in a safety (resp. reachability, Büchi, co-Büchi) automaton if it belongs to F^ω (resp. Q^*FQ^ω, $(Q^*F)^\omega$, Q^*F^ω). States from F will be called Büchi states in Büchi automata, and states from $Q \setminus F$ will be called co-Büchi states in co-Büchi automata.

Finally, we will also mention the parity acceptance condition: it uses a ranking function `rk` from Q to an interval of integers $[i, j]$. A run is accepting if the minimal rank appearing infinitely often is even (following the convention of [Ber+19]).

2.2.2 Explorability

We start by introducing the k-explorability game, which is the central tool allowing us to define the class of explorable automata.

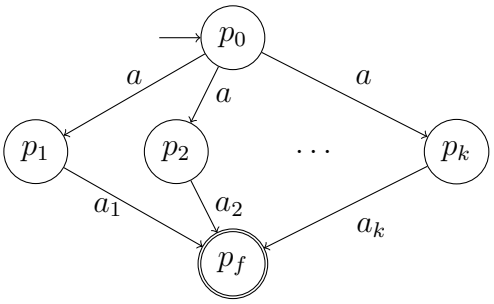
Definition (k-explorability game)**:**
Consider a non-deterministic automaton $\mathcal{A}$ on finite or infinite words, and an integer k. The *k-explorability game* on $\mathcal{A}$ is played on the arena Q^k. The two players are called Determiniser and Spoiler, and they play as follows.

- The initial position is the k-tuple $S_0 = (q_0, \dots, q_0)$.
- At step i from a position $S_{i-1} \in Q^k$, Spoiler chooses a letter $a_i \in \Sigma$, and Determiniser chooses $S_i \in Q^k$ such that for any token $l \in [0, k-1]$, $S_{i-1}(l) \xrightarrow{a} S_i(l)$ is a transition of $\mathcal{A}$ (where $S_i(l)$ stands for the l-th component in S_i).

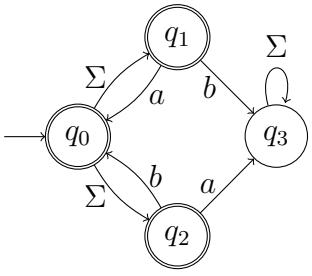
The play is won by Determiniser if for any $\beta \leq \omega$ such that the word $(a_i)_{1 \leq i < \beta}$ is in $\mathcal{L}(\mathcal{A})$, there is a token $l < k$ being accepted by $\mathcal{A}$, meaning that the sequence $(S_i(l))_{i<\beta}$ is an accepting run[1]. Otherwise, the winner is Spoiler.
We will say that $\mathcal{A}$ is *k-explorable* if Determiniser wins the k-explorability game.
We will say that $\mathcal{A}$ is *explorable* if it is k-explorable for some $k \in \mathbb{N}$.

Example 2.1:
The NFA $\mathcal{A}_k$ on alphabet $\{a, a_1, \dots, a_k\}$ depicted below is k-explorable, but not $(k-1)$-explorable. It can easily be adapted to a binary alphabet, by replacing in the automaton $a_1, \dots, a_k$ by distinct words of the same length.

On the other hand, the NFA $\mathcal{C}$ is a non-explorable NFA accepting all words on alphabet $\Sigma = \{a, b\}$. Indeed, Spoiler can win the k-explorability game for any k, by eliminating tokens one by one, choosing at each step the letter b if q_1 is occupied by at least one token, and the letter a otherwise.



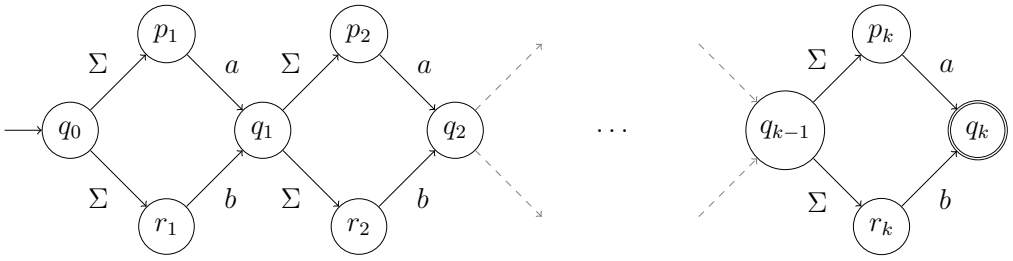
Explorable $\mathcal{A}_k$



Non-explorable $\mathcal{C}$

[1]This condition $\beta \leq \omega$ is actually accounting separately for the two cases of finite and infinite words, corresponding respectively to $\beta < \omega$ and $\beta = \omega$.

Example 2.2:
The following NFA $\mathcal{B}_k$ with $3k+1$ states on alphabet $\Sigma = \{a, b\}$ is explorable, but requires 2^k tokens. Indeed, since when choosing the $2i^{\text{th}}$ letter Spoiler can always pick the state p_i or r_i containing the least amount of tokens to decide whether to play a or b, the best strategy for Determiniser is to split his tokens evenly at each q_i. This means he needs to start with 2^k tokens to end up with at least one token in q_k after a word of Σ^{2k}.



Automaton $\mathcal{B}_k$

Let us mention a few facts that follow from the definition of explorability:

Lemma 2.1:

- Any automaton with a finite language is explorable.
- If $\mathcal{A}$ is k-explorable, then it is n-explorable for all $n \geq k$.
- If $\mathcal{A}$ is k-explorable and $\mathcal{B}$ is n-explorable on the same alphabet, then
  - $\mathcal{A} \cup \mathcal{B}$ (with states $Q = \{q_0\} \cup Q_\mathcal{A} \cup Q_\mathcal{B}$) is $(k+n)$-explorable,
  - the union product $\mathcal{A} \times \mathcal{B}$ (with $F = (F_\mathcal{A} \times Q_\mathcal{B}) \cup (Q_\mathcal{A} \times F_\mathcal{B})$) is $\max(k, n)$-explorable,
  - the intersection product $\mathcal{A} \times \mathcal{B}$ (with $F = F_\mathcal{A} \times F_\mathcal{B}$) is (kn)-explorable.

Proof. If $L(\mathcal{A})$ is finite, it is enough to take $k = |L(\mathcal{A})|$ tokens to witness explorability: for each $u \in L(\mathcal{A})$, the token t_u assumes that the input word is u and follows an accepting run of $\mathcal{A}$ over u as long as input letters are compatible with u. As soon as an input letter is not compatible with u, the token t_u is discarded and behaves arbitrarily for the rest of the play.

If $\mathcal{A}$ is k-explorable and $n \geq k$, then Determiniser can win the n-explorability game by using the same strategy with the first k tokens and making arbitrary choices with the $n - k$ remaining tokens.

If $\mathcal{A}$ and $\mathcal{B}$ are k- and n-explorable respectively, then Determiniser can use both strategies simultaneously with $k + n$ tokens in $\mathcal{A} \cup \mathcal{B}$, using k tokens in $\mathcal{A}$ and n tokens in $\mathcal{B}$. If the input word is in $\mathcal{A}$ (resp. $\mathcal{B}$), then the tokens playing in $\mathcal{A}$ (resp. $\mathcal{B}$) will win the play.

In the union product $\mathcal{A} \times \mathcal{B}$, it is enough to take $\max(k, n)$ tokens: if $0 \leq i < \min(k, n)$, the token number i follows the strategy of the token i in $\mathcal{A}$ on the first coordinate, and the strategy of the token i in $\mathcal{B}$ in the second one. If $\min(k, n) \leq i < \max(k, n)$, say wlog $k \leq i < n$, the token i follows an arbitrary strategy on the $\mathcal{A}$-component and the strategy of token i on the $\mathcal{B}$-component.

However, Determiniser may need up to kn tokens to play in $\mathcal{A} \times \mathcal{B}$ when the accepting set is $F_\mathcal{A} \times F_\mathcal{B}$: the token (i, j) will use the strategy of the token i in the k-explorability game of $\mathcal{A}$ together with the strategy of the token j in the n-explorability game of $\mathcal{B}$. This lower bound of kn cannot be improved: consider for instance $\mathcal{A}_k \times \mathcal{A}_n$, where $\mathcal{A}_k, \mathcal{A}_n$ are from Example 2.1. □

Notice that a similar notion was introduced in [KM19] under the name *width*. In [KM19], the emphasis is put on another version of the explorability game, where tokens can be duplicated, and $|Q|$ is an upper bound for the number of necessary tokens. In this work, we will on the contrary focus on non-duplicable tokens, for which some results of [KM19] already apply. In particular, the following holds:

Theorem 2.1 ([KM19, Rem. 6.9])**:**
Given an NFA $\mathcal{A}$ and an integer k, it is ExpTime-complete to decide whether $\mathcal{A}$ is k-explorable (even if we fix $k = |Q_\mathcal{A}|/2$).

We aim here at answering a different question:

Definition (Explorability problem)**:**
The *explorability* problem is the question, given a non-deterministic automaton $\mathcal{A}$, of deciding whether it is explorable.

Questions : Is the explorability problem decidable ? If yes, what is its complexity ? We will first give some motivation for this problem in Section 2.2.3.

2.2.3 Link with GFG automata

An automaton $\mathcal{A}$ is *Good-for-Games* (GFG) if it is 1-explorable, *i.e.* if there is a strategy $\sigma : \Sigma^* \to Q$ resolving the non-determinism based on the word read so far, with the guarantee that the run piloted by this strategy is accepting whenever the input word is in $L(\mathcal{A})$. See *e.g.* [Bok+13] for an introduction to GFG automata.

We will give here an additional and stronger link between explorable and GFG automata. In this part, we will mainly be interested in automata on infinite words.

One of the main open problems related to GFG automata on infinite words is to decide, given a nondeterministic parity automaton, whether it is GFG. For now, the problem is only known to be in PTime for co-Büchi [KS15] and Büchi [BK18] automata. Extending this result even to 3 parity ranks is still open, and only a naive ExpTime upper bound [HP06] is known in this case. The following result shows that explorability is relevant in this context:

Theorem 2.2:
Given an explorable parity automaton $\mathcal{A}$ of fixed parity index, it is in PTime to decide whether it is GFG.

This is one of the motivations to get a better understanding of explorable automata. Indeed, if we can obtain an efficient algorithm for recognizing them, or if we are in a context guaranteeing that we are only dealing with explorable automata, this result shows that we can obtain an efficient algorithm for recognizing GFG automata.

The rest of this section will be devoted to the proof of Theorem 2.2. The proof idea is inspired by [BK18].

Let $\mathcal{A}$ be an explorable parity automaton, of fixed parity index $[i, j]$.

We briefly recall the definition of the token game $G_k(\mathcal{A})$ defined in [BK18], for an arbitrary $k \in \mathbb{N}$. At each round, Adam plays a letter $a \in \Sigma$, then Eve moves her token according to an a-transition, and finally Adam moves his k tokens according to a-transitions. Eve wins the play if her token builds an accepting run, or if all of Adam's tokens build rejecting runs.

We will prove that the game $G_2(\mathcal{A})$ is won by Eve if and only $\mathcal{A}$ is GFG. Since $G_2(\mathcal{A})$ can be solved in PTIME for fixed parity index [BK18], this is enough to conclude.

First, it is clear that if $\mathcal{A}$ is GFG, then Eve wins $G_2(\mathcal{A})$ [BK18]: Eve can simply play her GFG strategy with her token, ignoring Adam's tokens.

For the converse, assume Eve wins $G_2(\mathcal{A})$, we want to prove that $\mathcal{A}$ is GFG. We use the following lemma:

Lemma 2.2 ([BK18, Thm. 14])**:**
Eve wins $G_2(\mathcal{A})$ if and only if Eve wins $G_k(\mathcal{A})$ for all $k \geq 2$.

Since $\mathcal{A}$ is explorable, there is $k \in \mathbb{N}$ such that $\mathcal{A}$ is k-explorable. Let τ_k be a winning strategy for Determiniser in the k-explorability game of $\mathcal{A}$, and σ_k be a winning strategy for Eve in $G_k(\mathcal{A})$. We show that we can combine these two strategies to yield a GFG strategy σ for $\mathcal{A}$. This proof follows the same idea as in [BK18] where the explorability hypothesis is not available, but $\mathcal{A}$ is assumed to be Büchi.

Let us first sketch the remainder of the proof. The strategy σ will store k virtual tokens in its memory. When the automaton reads a new letter $a \in \Sigma$, these k tokens will be updated according to τ_k. Then the choice of σ will follow the strategy σ_k against these k tokens. Notice that the strategies τ_k and σ_k might use additional memory, but this is completely transparent in this proof scheme. If the input word is in $L(\mathcal{A})$, then by correctness of τ_k, one of the k virtual tokens will accept. Thus, by correctness of σ_k, the run chosen by σ will be accepting. Therefore, σ is a correct GFG strategy, witnessing that $\mathcal{A}$ is GFG. This concludes the proof sketch of Theorem 2.2.

We now write this proof more formally, by first describing the shape of strategies τ_k and σ_k.

The strategy τ_k has access to the history of the play in the k-explorability game, and must decide on a move for Determiniser. Notice that it is always enough to know the history of the opponent's moves (here the letters of Σ played so far), since this allows to compute the answer of Determiniser at each step, and therefore build a unique play. Thus we can take for τ_k a function $\Sigma^* \to Q^k$. If the word played so far is $u \in \Sigma^*$, the tuple of states reached by the k tokens moved according to τ_k is $\tau_k(u) \in Q^k$, with in particular $\tau_k(\varepsilon) = (q_0^{\mathcal{A}}, \dots, q_0^{\mathcal{A}})$.

If $w = a_1 a_2 \cdots \in \Sigma^\omega$, and $i \in \mathbb{N}$, let us note $(q^i_{w,1}, \dots, q^i_{w,k}) = \tau_k(a_1 \dots a_i)$. That is $q^i_{w,j}$ is the state reached by the j^{th} token after i steps in the run induced by τ_k and u. If $j \in [1, k]$, let us note $\rho_{u,j}$ the infinite run $q^0_{w,j} q^1_{w,j} q^2_{w,j} \dots$, followed by the j^{th} token in this play. By definition of τ_k, we have the guarantee that for all $w \in L(\mathcal{A})$, there exists $j \in [1, k]$ such that $\rho_{w,j}$ is accepting.

If $u = a_1 \dots a_n \in \Sigma^*$ is a finite word, we define $\tau'_k(u) = (\tau_k(\varepsilon), \tau_k(a_1), \tau_k(a_1 a_2) \dots, \tau_k(u))$

the list of partial runs induced by τ_k on u.

Let us now turn to the strategy σ_k of Eve in $G_k(\mathcal{A})$. The type of this strategy is $\sigma_k : \Sigma^* \times (Q^k)^* \to Q$. Indeed, this time, the history of Adam's moves must contain his choice of letters together with his choices of positions for his k tokens. So $\sigma_k(u, \gamma)$ gives the state reached by Eve's token after a history (u, γ) for the moves of Adam. Notice that at each step, Eve must move before Adam in this game $G_k(\mathcal{A})$, so γ does not contain the choice of Adam on the last letter of u. This means that, except for $u = \varepsilon$, we can always assume $|u| = |\gamma| + 1$ in a history (u, γ).

We have the guarantee that if Adam plays an infinite word w together with runs $\rho_1, \dots, \rho_k$ on w, at least one of which is accepting, then the run yielded by σ_k against $(w, (\rho_1, \dots, \rho_k))$ is accepting.

We finally define the GFG strategy σ for $\mathcal{A}$, of type $\Sigma^* \to Q$, by induction: $\sigma(\varepsilon) = q_0^{\mathcal{A}}$, and $\sigma(ua) = \sigma_k(ua, \tau'_k(u))$.

This amounts to playing the strategy σ_k in $G_k(\mathcal{A})$, against Adam playing a word w and moving his k tokens according to the strategy τ_k against w. If the infinite word $w = a_1 a_2 \dots$ chosen by Adam is in $L(\mathcal{A})$, then by correctness of τ_k one of the k runs $\rho_{w,1}, \dots, \rho_{w,k}$ yielded by τ_k is accepting. Hence, by correctness of σ_k, the run $\sigma(\varepsilon)\sigma(a_1)\sigma(a_1 a_2)$ yielded by σ (based on σ_k) is accepting. This concludes the proof that σ is a correct GFG strategy for $\mathcal{A}$, witnessing that $\mathcal{A}$ is GFG.

2.3 Decidability and complexity of explorability

In this section, we prove that the explorability problem is decidable and ExpTime-complete for NFAs and Büchi automata.

We start by showing in Section 2.3.1 decidability of the explorability problem for NFAs using the results of [Ber+19] as a black box. This yields an algorithm in 2-ExpTime. We give in Section 2.3.2 a polynomial reduction in the other direction, thereby obtaining ExpTime-hardness of the NFA explorability problem. To obtain a matching upper bound and show ExpTime-completeness, we use again [Ber+19], but this time we must "open the black box" and dig into the technicalities of their ExpTime algorithm while adapting them to our setting. We do so in Section 2.3.3, directly treating the more general case of Büchi automata.

2.3.1 2-ExpTime algorithm via a black box reduction

Let us start by recalling the population control problem (PCP) of [Ber+19].

Definition (k-population game)**:**
Given an NFA $\mathcal{B}$ with a distinguished target state $f \in Q_{\mathcal{B}}$, and an integer $k \in \mathbb{N}$, the *k-population game* is played similarly to the k-explorability game, only the winning condition differs: Spoiler wins if the game reaches a position where all tokens are in the state f.

The PCP asks, given $\mathcal{B}$ and $f \in Q_{\mathcal{B}}$, whether Spoiler wins the k-population game for all $k \in \mathbb{N}$. Notice that this convention is opposite to explorability, where positive instances

are defined via a win of Determiniser. The PCP is shown in [Ber+19] to be ExpTime-complete. We will present here a direct exponential reduction from the explorability problem to the PCP.

Let $\mathcal{A}$ be an NFA. Our goal is to build an exponential NFA $\mathcal{B}$ with a distinguished state f such that $(\mathcal{B}, f)$ is a negative instance of the PCP if and only if $\mathcal{A}$ is explorable.

We choose $Q_\mathcal{B} = (Q_\mathcal{A} \times \mathcal{P}(Q_\mathcal{A})) \uplus \{f, \bot\}$, where $f, \bot$ are fresh sink states. The alphabet of $\mathcal{B}$ will be $\Sigma_\mathcal{B} = \Sigma \uplus \{a_{\text{test}}\}$, where a_{test} is a fresh letter.

The initial state of $\mathcal{B}$ is $q_0^\mathcal{B} = (q_0^\mathcal{A}, \{q_0^\mathcal{A}\})$. Notice that we do not need to specify accepting states in $\mathcal{B}$, as acceptance plays no role in the PCP.

We finally define the transitions of $\mathcal{B}$ in the following way:

- $(p, X) \xrightarrow{a} (q, \Delta_\mathcal{A}(X, a))$ if $a \in \Sigma$ and $q \in \Delta_\mathcal{A}(p, a)$,
- $(p, X) \xrightarrow{a_{\text{test}}} f$ if $p \notin F_\mathcal{A}$ and $X \cap F_\mathcal{A} \neq \emptyset$.
- $(p, X) \xrightarrow{a_{\text{test}}} \bot$ otherwise.

We aim at proving the following Lemma:

Lemma 2.3:
For any $k \in \mathbb{N}$, $\mathcal{A}$ is k-explorable if and only if Determiniser wins the k-population game on $(\mathcal{B}, f)$.

Notice that as long as letters of Σ are played, the second component of states of $\mathcal{B}$ evolves deterministically and keeps track of the set of reachable states in $\mathcal{A}$. Moreover, the letter a_{test} also acts deterministically on $Q_\mathcal{B}$. Therefore, the only non-determinism to be resolved in $\mathcal{B}$ is how the first component evolves, which amounts to building a run in $\mathcal{A}$. Thus, strategies driving tokens in $\mathcal{A}$ and $\mathcal{B}$ are isomorphic. It now suffices to observe that Spoiler wins the k-population game on $(\mathcal{B}, f)$ if and only if he has a strategy allowing to eventually play a_{test} while all tokens are in a state of the form (q, X) with $q \notin F_\mathcal{A}$ and $X \cap F_\mathcal{A} \neq \emptyset$. This is equivalent to Spoiler winning the k-explorability game of $\mathcal{A}$, since $X \cap F_\mathcal{A} \neq \emptyset$ witnesses that the word played so far is in $L(\mathcal{A})$.

This concludes the proof that $\mathcal{A}$ is explorable if and only if $(\mathcal{B}, f)$ is a negative instance of the PCP. So given an NFA $\mathcal{A}$ that we want to test for explorability, it suffices to build $(\mathcal{B}, f)$ as above, and use the ExpTime algorithm from [Ber+19] as a black box on $(\mathcal{B}, f)$. Since $\mathcal{B}$ is of exponential size compared to $\mathcal{A}$, we obtain the following result:

Theorem 2.3:
The NFA explorability problem is decidable and in 2-ExpTime.

2.3.2 ExpTime-hardness of NFA explorability

We will perform here an encoding in the converse direction: starting from an instance $(\mathcal{B}, f)$ of the PCP, we build polynomially an NFA $\mathcal{A}$ such that $\mathcal{A}$ is explorable if and only if $(\mathcal{B}, f)$ is a negative instance of the PCP.

It is stated in [Ber+19] that, without loss of generality, we can assume that f is a sink state in $\mathcal{B}$, and we will use this assumption here.

Let $\mathcal{C}$ be the 4-state automaton of Example 2.1, that is non-explorable and accepts all words on alphabet $\Sigma_\mathcal{C} = \{a, b\}$. Recall that, as an instance of the PCP, $\mathcal{B}$ does not come with an acceptance condition. We will consider that its accepting set is $F_\mathcal{B} = Q_\mathcal{B} \setminus \{f\}$.

We will take for $\mathcal{A}$ the product automaton $\mathcal{B} \times \mathcal{C}$ on alphabet $\Sigma_A = \Sigma_\mathcal{B} \times \Sigma_\mathcal{C}$, with the union acceptance condition: $\mathcal{A}$ accepts whenever one of its components accepts. The transitions of $\mathcal{A}$ are defined as expected: $(p, p') \xrightarrow{a_1,a_2} (q, q')$ in $\mathcal{A}$ whenever $p \xrightarrow{a_1} q$ in $\mathcal{B}$ and $p' \xrightarrow{a_2} q'$ in $\mathcal{C}$.

Since $L(\mathcal{C}) = (\Sigma_\mathcal{C})^*$, we have $L(\mathcal{A}) = (\Sigma_\mathcal{A})^*$. The intuition for the role of $\mathcal{C}$ in this construction is the following: it allows us to modify $\mathcal{B}$ in order to accept all words, without interfering with its explorability status.

Lemma 2.4:
For any $k \in \mathbb{N}$, $\mathcal{A}$ is k-explorable if and only if Determiniser wins the k-population game on $(\mathcal{B}, f)$.

Proof. Assume that $\mathcal{A}$ is k-explorable, via a strategy σ. Then Determiniser can play in the k-population game on $(\mathcal{B}, f)$ using σ as a guide. In order to simulate σ, one must feed to it letters from $\Sigma_\mathcal{C}$ in addition to letters from $\Sigma_\mathcal{B}$ chosen by Spoiler. This is done by applying a winning strategy for Spoiler in the k-explorability game of $\mathcal{C}$. Assume for contradiction that, at some point, this strategy σ reaches a position where all tokens are in a state of the form (f, q) with $q \in Q_\mathcal{C}$. Since f is a sink state, when the play continues it will eventually reach a point where all tokens are in (f, q_3), where q_3 is the rejecting sink of $\mathcal{C}$. This is because we are playing letters from $\Sigma_\mathcal{C}$ according to a winning strategy for Spoiler in the k-explorability game of $\mathcal{C}$, and this strategy guarantees that all tokens eventually reach q_3 in $\mathcal{C}$. But this state (f, q_3) is rejecting in $\mathcal{A}$, and $L(\mathcal{A}) = (\Sigma_\mathcal{A})^*$, so this is a losing position for Determiniser in the k-explorability game of $\mathcal{A}$. Since we assumed σ is a winning strategy in this game, we reach a contradiction. This means that following this strategy σ together with an appropriate choice for letters from $\Sigma_\mathcal{C}$, we guarantee that at least one token never reaches the sink state f on its $\mathcal{B}$-component. This corresponds to Determiniser winning in the k-population game on $(\mathcal{B}, f)$. □

Conversely, assume that Determiniser wins in the k-population game on $(\mathcal{B}, f)$, via a strategy σ. The same strategy can be used in the k-explorability game of $\mathcal{A}$, by making arbitrary choices on the $\mathcal{C}$ component. As before, this corresponds to a winning strategy in the k-explorability game of $\mathcal{A}$, since there is always at least one token with $\mathcal{B}$-component in $F_\mathcal{B} = Q_\mathcal{B} \setminus \{f\}$. This achieves the hardness reduction, and allows us to conclude:

Theorem 2.4:
The NFA explorability problem is ExpTime-hard.

Remark 2.1:
Using standard padding arguments, it is straightforward to extend Theorem 2.4 to ExpTime-hardness of explorability for automata on infinite words, using any of the acceptance conditions defined in Section 2.2.1.

Let us give some intuition on why we can obtain a polynomial reduction in one direction, but did not manage to do so in the other direction. Intuitively, the explorability problem is "more difficult" than the PCP for the following reason. In the PCP, Spoiler is

allowed to play any letters, and the winning condition just depends on the current position. On the contrary, the winning condition of the k-explorability game mentions that the word chosen by Spoiler must belong to the language of the NFA. In order to verify this, we *a priori* need to append to the arena an exponential deterministic automaton for this language, and this is what is done in Section 2.3.1. This complicated winning condition is also the source of difficulty of recognizing GFG parity automata.

2.3.3 ExpTime algorithm for Büchi explorability

Theorem 2.5:
The explorability problem can be solved in ExpTime for Büchi automata (and all simpler conditions).

Before giving a complete proof of Theorem 2.5, we write a sketch that summarizes the main elements in this proof.

Proof sketch

The algorithm is adapted from the ExpTime algorithm for the PCP from [Ber+19]. We will recall here the main ideas of this algorithm, and describe how we adapt it to our setting.

Let $\mathcal{A}$ be an NFA, together with a target state f. The idea in [Ber+19] is to abstract the population game with arbitrary many tokens by a game called the *capacity game*. This game allows Determiniser to describe only the support of his set of tokens, *i.e.* the set of states occupied by tokens. The sequence of states obtained in a play can be analysed via a notion of *bounded capacity*, in order to detect whether it actually corresponds to a play with finitely many tokens. This notion can be approximated by the more relaxed *finite capacity*, which is a regular property that is equivalent to bounded capacity in a context where games are finite-memory determined. This property of finite capacity can be verified by a deterministic parity automaton, yielding a parity game that can be won by Spoiler if and only if $(\mathcal{A}, f)$ is a positive instance of the PCP. Since this parity game has size exponential in $\mathcal{A}$, this yields an ExpTime algorithm for the PCP.

Here, we will perform the following tweaks to this construction. We now start with a Büchi automaton $\mathcal{A}$, and want to decide whether it is explorable.

First, we need to control that the infinite word played by Spoiler is in $L(\mathcal{A})$. This requires to build a deterministic parity automaton $\mathcal{D}$ recognizing $L(\mathcal{A})$, and incorporate it into the arena. The size of $\mathcal{D}$ is exponential with respect to $\mathcal{A}$. We then follow [Ber+19] and build the capacity game augmented with $\mathcal{D}$. This time, a sequence of supports is winning if infinitely many of them contain an accepting state. We emphasize that we use here a particularity of the Büchi condition: observing the sequences of support sets of tokens is enough to decide whether one of the tokens follows an accepting run. The same particularity was used in [BK18], and was a crucial tool allowing to give a PTime algorithm for Büchi GFGness. Since this modification still allows us to manipulate supports as simple sets, we can make use of the capacity game as before. We give in Remark 2.3 (after the complete proof) an example showing that a naive adaptation of this construction to co-Büchi automata would not be correct.

Finally, we show that we can as in [Ber+19] obtain a parity game of exponential size characterizing explorability of $\mathcal{A}$, yielding the wanted ExpTime algorithm.

We also remark that, as in [Ber+19], this construction gives a doubly exponential upper bound on the number of tokens needed to witness explorability. Moreover, the proof from [Ber+19] that this is tight also stands here.

Complete proof of Theorem 2.5

In this part, $\mathcal{A} = (\Sigma, Q, q_0^{\mathcal{A}}, \Delta_{\mathcal{A}}, F_{\mathcal{A}})$ is a non-deterministic Büchi automaton. We start by computing in exponential time an equivalent deterministic parity automaton $\mathcal{D} = (\Sigma, Q_{\mathcal{D}}, q_0^{\mathcal{D}}, \delta_{\mathcal{D}}, F_{\mathcal{D}})$, via any standard method.

The algorithm described in this section is adapted from [Ber+19]. Many results from this previous work still hold in our framework. We will however need to adapt some constructions and give new arguments, both to fit our explorability framework, and to generalize from NFA to Büchi automata.

Definition (Transfer graph)**:**
A *transfer graph* G is a subset of $Q \times Q$. We say that it is compatible with a letter a if every edge in G corresponds to a transition in $\mathcal{A}$ labelled by a, *i.e.* for any $(q, r) \in G$, we have $(q, a, r) \in \Delta_{\mathcal{A}}$. In other words, G is a subgraph of the transition graph of the letter a.

Given a transfer graph G and a set of states $X \subseteq Q$, we note $G(X) = \{q \in Q \mid \exists r \in X, (q, r) \in G\}$. We call respectively $\mathrm{Dom}(G)$ and $\mathrm{Im}(G)$ the projections of G on its first and second coordinate, *i.e.* $\mathrm{Dom}(G) = \{q \in Q \mid \exists r \in Q, (q, r) \in G\}$ and $\mathrm{Im}(G) = G(Q)$.

The composition of transfer graphs is defined the natural way: $G \cdot H = \{(x, z) \mid \exists y, (x, y) \in G \wedge (y, z) \in H\}$.

Definition (Support game)**:**
The *support game* is played in the arena $\mathcal{P}(Q) \times Q_{\mathcal{D}}$, called *support arena*. It is played as follows by Determiniser and Spoiler.

- The starting support is $S_0 = (\{q_0^{\mathcal{A}}\}, q_0^{\mathcal{D}})$.
- At any given step with support (B, q), Spoiler chooses a letter $a \in \Sigma$, then Determiniser chooses a transfer graph G compatible with a, and with $\mathrm{Dom}(G) = B$. The play then moves to $(\mathrm{Im}(G), \delta_{\mathcal{D}}(q, a))$.

A play can be represented by a sequence $(B_0, q_0) \xrightarrow{a_1, G_1} (B_1, q_1) \xrightarrow{a_2, G_2} (B_2, q_2) \ldots$.

We say that Spoiler wins the play if the run $q_0 q_1 q_2 \ldots$ of $\mathcal{D}$ is parity accepting, while only finitely many B_i contain Büchi states (from $F_{\mathcal{A}}$).

Note that a winning strategy for Determiniser in the support game cannot in general be interpreted as a witness of explorability. This is illustrated by the automaton $\mathcal{C}$ from Example 2.1. For any $k \in \mathbb{N}$, the k-explorability game is won by Spoiler on that automaton, while Determiniser wins the support game. Intuitively, the support game does not account for the limits of resources for Determiniser.

On the other hand, a winning strategy for Spoiler in this support game does translate into a non-explorability witness, *i.e.* a strategy for Spoiler in the k-explorability game for

any k. The support game is therefore "too easy" for Determiniser, and this is what we try to correct in the following.

Definition (Projection of a play)**:**
Given a play $S_0 \xrightarrow{a_1} S_1 \xrightarrow{a_2} S_2 \dots$ in the k-explorability game, the *projection* of that play in the support arena is the play $(B_0, q_0) \xrightarrow{a_1, G_1} (B_1, q_1) \xrightarrow{a_2, G_2} (B_2, q_2) \dots$, where:

- B_i is the support of S_i (states occupied in S_i),
- $q_0 = q_0^{\mathcal{P}}$ and $q_{i+1} = \delta_{\mathcal{D}}(a_{i+1}, q_i)$ for all i,
- $G_{i+1} = \{(S_i(j), S_{i+1}(j)) \mid j \in [0, k-1]\}$.

This corresponds to forgetting the multiplicity of tokens and only keeping track of the transitions that are used.

Definition (Realisable play)**:**
A play in the support arena is *realisable* if it is the projection of a play in the k-explorability game for some $k \in \mathbb{N}$.

We would like to restrict plays in the support arena to realisable ones only. To do so, we define the notion of capacity as follows.

Definition (Accumulator and capacity [Ber+19])**:**
In a play $(B_0, q_0) \xrightarrow{a_1, G_1} (B_1, q_1) \xrightarrow{a_2, G_2} (B_2, q_2) \dots$, an *accumulator* is a sequence $(T_j)_{j \in \mathbb{N}}$ such that for any j, $T_j \subseteq B_j$ and $T_{j+1} \supseteq G_{i+1}(T_j)$. An edge $(q, r) \in G_{j+1}$ is an *entry* for $(T_j)_{j \in \mathbb{N}}$ at index i if $q \notin T_j$ and $r \in T_{j+1}$.

A play has *finite capacity* if every accumulator has finitely many entries, and *bounded capacity* if the number of entries of its accumulators is bounded.

This definition gives us tools to talk about realisable plays in a more practical way, as shown by the following Lemma. Note that although the explorability game is replaced by the population control game in [Ber+19], the same proof still applies here.

Lemma 2.5 ([Ber+19, Lem 3.5])**:**
A play is realisable if and only if it has bounded capacity.

Moreover, the proof of Lemma 2.5 can also be used to get the following result, which we will use later. Note that we talk about the explorability game in this Lemma, but this only concerns its arena, regardless of the winning condition. The proof holds because the arena from [Ber+19] is identical.

Lemma 2.6 ([Ber+19, Lem 3.5])**:**
If Determiniser has a strategy τ in the support arena such that any play compatible with τ has capacity bounded by c, then he has a strategy τ' in the 2^{c+1}-tokens explorability game such that any play compatible with τ' has its projection compatible with τ.

We will use the notion of capacity to define the following game, using finite capacity instead of bounded to obtain a regular winning condition.

Definition (Capacity game)**:**
The *capacity game* is played in the support arena. Given a play $(B_0, q_0) \xrightarrow{a_1,G_1} (B_1, q_1) \xrightarrow{a_2,G_2} (B_2, q_2) \ldots$, Spoiler wins if it is a winning play in the support game, or if it has infinite capacity.

Lemma 2.7 ([Ber+19, Prop 3.8])**:**
Either Spoiler or Determiniser wins the capacity game, and the winner has a winning strategy with finite memory.

Proof. Although this result talks about slightly different objects than in [Ber+19, Prop 3.8], their proof actually still holds with our definitions of capacity game and support game. The proof proceeds by building a nondeterministic Büchi automaton verifying that the capacity is infinite, determinising it into a parity automaton, and incorporating it into the arena to yield a parity game equivalent to the capacity game. The winner of this parity game has a positional strategy, which corresponds to a finite memory strategy in the capacity game. □

Lemma 2.8 (adapted from [Ber+19, Prop 3.9])**:**
If Spoiler wins the capacity game, then he wins the k-explorability game for any k.

Proof. Here Spoiler can simply apply the strategy for the capacity game to the explorability game, by remembering only the information that is relevant from the point of view of the capacity game (*i.e.* the supports and transfer graphs). This will simulate a realisable play of the capacity game, which has bounded capacity by Lemma 2.5. Since the strategy is winning in the capacity game, and this simulated play cannot have infinite capacity, Spoiler wins the underlying support game. This ensures the win for Spoiler in the explorability game: he plays a word of $L(\mathcal{A})$ as witnessed by the acceptance of $\mathcal{D}$, while finitely many Büchi states are witnessed by tokens of Determiniser. We use here the particular property of Büchi condition: one of the tokens follows an accepting run if and only if it occurs infinitely many times that the support set occupied by tokens contains a Büchi state. □

Lemma 2.9 (adapted from [Ber+19, Prop 3.10])**:**
If Determiniser wins the capacity game using finite memory M, then he wins the k-explorability game for some $k \in \mathbb{N}$.

Proof. We first prove that under these conditions, Determiniser can win the capacity game while ensuring a capacity bounded by $|M| \times |Q_{\mathcal{D}}| \times 4^{|Q|}$.

Let us consider a winning strategy τ with memory M for Determiniser in the capacity game. We take a play $(B_0, q_0) \xrightarrow{a_1,G_1} (B_1, q_1) \xrightarrow{a_2,G_2} (B_2, q_2) \ldots$ compatible with τ, and we show that its capacity is bounded by $|M| \times |Q_{\mathcal{D}}| \times 4^{|Q|}$.

Given an accumulator $\mathcal{T} = (T_i)_{i \in \mathbb{N}}$, if there are two integers $i < j$ such that $m_i = m_j$ (memory states at steps i and j), $B_i = B_j$, $q_i = q_j$ and $T_i = T_j$, then one can build a play that loops on the corresponding interval, while still being compatible with τ. This accumulator cannot have infinitely many entries, so $\mathcal{T}$ does not have any entry in the interval $[i, j]$. As a consequence, if i and j are entry times, we have $(m_i, B_i, q_i, T_i) \neq$

(m_j, B_j, q_j, T_j), which means there can be at most $|M|\times 2^{|Q|}\times|\mathcal{Q}_\mathcal{D}|\times 2^{|Q|} = |M|\times|Q_\mathcal{D}|\times 4^{|Q|}$ entries in the accumulator $\mathcal{T}$.

We now know that the capacity of any play compatible with τ is bounded by $|M| \times |Q_\mathcal{D}| \times 4^{|Q|}$. Take $k = 2^{1+|M|\times|Q_\mathcal{D}|\times 4^{|Q|}}$. Lemma 2.6 then provides a strategy for Determiniser in the k-explorability game, that ensures that the successive supports (*i.e.* the sets of states occupied by tokens) contain Büchi states infinitely often. This means that at least one token visits Büchi states infinitely often, since there are finitely many tokens. This ensures a win for Determiniser. □

These Lemmas 2.8 and 2.9 give a way to solve the explorability problem if we can efficiently find the winner of the corresponding capacity game. Note that we could use the parity game built in the proof of Lemma 2.7 to solve the problem, but this would yield a doubly exponential algorithm, since the parity automaton that we build in this proof is itself doubly exponential.

The following gives an exponential time algorithm for solving the capacity game, and therefore the explorability problem.

Definition (Leaks and separations)**:**
If G and H are two transfer graphs, we say that G *leaks* at H if there are three states q, x, y such that $(q, y) \in G \cdot H$, $(x, y) \in H$ and $(q, x) \notin G$.

We say that G *separates* states r and t if there is a q such that $(q, r) \in G$ and $(q, t) \notin G$. The separator of G, noted $\texttt{Sep}(G)$, is the set of all such (r, t).

Note that in a play denoted as before, whenever $i < j < n$, we have $\texttt{Sep}(G[i, n]) \subseteq \texttt{Sep}(G[j, n])$.

We will now define the tracking list of a play. The point of that list will be to provide an easy way to detect indices that leak infinitely often.

Definition (Tracking list)**:**
The *tracking list* $\mathcal{L}_n$ at step n is a list of transfer graphs $\{G[i_1, n], \ldots, G[i_{k_n}, n]\}$. It is defined inductively, with $\mathcal{L}_0$ the empty list, and $\mathcal{L}_n$ computed as follows.

- We update every $G[i, n-1]$ in $\mathcal{L}_{n-1}$ into $G[i, n]$ by composing with G_n.
- We then add $G[n-1, n] = G_n$ at the end of the list.
- And finally, we clean the list by removing any graph with a separator identical to the previous one.

If for some i, $G[i, n] \in \mathcal{L}_n$ for every $n > i$, we say that i is *remanent*.

To properly use these tracking lists, it suffices to know that the following result holds. For more details, we refer the reader to [Ber+19].

Lemma 2.10 ([Ber+19, Lem 4.4])**:**
A play has infinite capacity if and only if there is a remanent index that leaks infinitely often.

We now define a game $\mathcal{G}_\mathcal{A}$ associated to $\mathcal{A}$, that extends the support arena using tracking lists to detect infinite capacity plays. Once again, this is an adaptation from [Ber+19].

The **states** of $\mathcal{G}_\mathcal{A}$ are in $\mathcal{P}(Q) \times Q_\mathcal{D} \times \mathcal{G}^{\leq |Q|^2}$, where $\mathcal{G}^{\leq |Q|^2}$ is the set of lists of at most $|Q|^2$ transfer graphs. Each state can be written as (B, q, L) where B is a subset of Q, q is a state of $\mathcal{D}$, and L is a tracking list. The initial state is $(\{q_0^\mathcal{A}\}, q_0^\mathcal{D}, \varepsilon)$.

The **transitions** are the ones that can be written $(B, q, L) \xrightarrow{p,a,G} (B', q', L')$ with the following conditions.

- $(B, q) \xrightarrow{a,G} (B', q')$ is a transition from the support arena.
- L' is obtained by updating L with G, as detailed in the definition of tracking list.
- Take $L = \{H_1, \dots, H_k\}$ and $L' = \{H'_1, \dots, H'_{k'}\}$. Let p' be the smallest index such that $H_{p'}$ leaks at G, or $k+1$ if there is no such index. Let p'' be the smallest index such that $H'_{p''} \neq H_{p''} \cdot G$, or $k+1$ if there is none. We then take $p = \min(2p'+1, 2p'')$ (which implies that $p \in [2, 2|Q|^2 + 1]$).

To choose a transition, Spoiler first chooses a letter, then Determiniser picks a transition graph compatible with that letter. The rest is determined by the conditions above. This creates a play that can be denoted as $(B_0, q_0, L_0) \xrightarrow{a_1,G_1,p_1} (B_1, q_1, L_1) \xrightarrow{a_2,G_2,p_2} \dots$.

The winning condition for Spoiler goes as follows. Either the inferior limit of $(p_i)_{i>0}$ is odd, or the run $(q_i)_{i\geq 0}$ is accepting while there are finitely many accepting states seen in $(B_i)_{i\geq 0}$.

Lemma 2.11 (adapted from [Ber+19, Thm 4.5])**:**
Spoiler wins $\mathcal{G}_\mathcal{A}$ if and only if he wins the capacity game.

Proof. First note that strategies in the support arena can be easily translated to $\mathcal{G}_\mathcal{A}$ and conversely, since in both cases Spoiler only chooses letters while Determiniser picks transfer graphs, and the rest is determined by these data.

If Spoiler has a winning strategy in $\mathcal{G}_\mathcal{A}$, then he can play the same strategy in the capacity game. Such a play can be written as $(B_0, q_0) \xrightarrow{a_1,G_1} (B_1, q_1) \xrightarrow{a_2,G_2} \dots$, and the play of $\mathcal{G}_\mathcal{A}$ happening in the memory of Spoiler is $(B_0, q_0, L_0) \xrightarrow{a_1,G_1,p_1} (B_1, q_1, L_1) \xrightarrow{a_2,G_2,p_2} \dots$. We use the notation $L_n = \{H_n^1, \dots, H_n^{k_n}\}$.

Since Spoiler plays according to a winning strategy in the simulated game $\mathcal{G}_\mathcal{A}$, at least one of his winning conditions for that game hold in this play.

If the limit parity is $2p+1$ for some p, then for any n large enough, H_n^p is the same as H_n^{p+1} (otherwise there would be a parity less than $2p+1$ later) and leaks infinitely often, so Spoiler wins the capacity game.

If the run $(q_i)_{i\geq 0}$ is accepting while there are finitely many accepting states seen in $(B_i)_{i\geq 0}$, then this also ensures the win for Spoiler in the capacity game.

In both cases, the play is therefore won by Spoiler.

On the other hand, if Spoiler wins the capacity game, he can also use the same strategy in $\mathcal{G}_\mathcal{A}$, with the same correspondence between the winning conditions. □

We can finally conclude with the main result of this section:

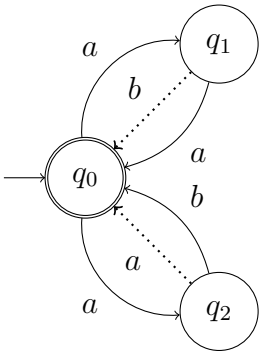


Figure 2.1: A co-Büchi automaton on which the projection of a play in not enough to determine the winner (the dotted lines represent co-Büchi transitions)

Theorem 2.6:
The Büchi explorability problem can be solved in ExpTime.

Proof. To prove this result, it is enough to prove that the game $\mathcal{G}_{\mathcal{A}}$ can be solved in exponential time in the size of $\mathcal{A}$, since the answer to that problem also answers the explorability of $\mathcal{A}$. We show that the winning condition of the game $\mathcal{G}_{\mathcal{A}}$ for Spoiler can be seen as a disjunction of parity conditions. Formally, it is of the form Parity$\vee$(Parity$\wedge$Co-Büchi). But it is straightforward to turn the conjunction into a parity condition with twice as many priorities. Thus $\mathcal{G}_{\mathcal{A}}$ can be seen as a generalized parity game. Such games are studied in [CHP07], which gives us an algorithm for solving $\mathcal{G}_{\mathcal{A}}$ in time $O(m^{4d}m^2)\frac{(2d)!}{d!^2}$, where d is the number of priorities and m the size of the game.

If we take $n = |\mathcal{A}|$, using the fact that $m = O(2^n)$, we get the complexity $O(2^{4nd+2n})\frac{(2d)!}{d!^2}$, which can be simplified into $O(2^{4n^3+2n}(2n^2)^{n^2}) = O(2^{5n^3+2n})$ using the fact that $d = O(n^2)$. This gives us an exponential bound for the time complexity of this problem. □

Remark 2.2:
We can also be interested in the number of tokens needed for Determiniser to witness explorability of an automaton. By inspecting our proof, we can see that we obtain a doubly exponential upper bound. Moreover, we can use the same construction as in [Ber+19, Prop 6.3] to show that this is tight, *i.e.* some automata require a doubly exponential number of tokens to witness explorability.

Remark 2.3:
This algorithm only works as such in the case of Büchi automata. The next step would be to adapt it to co-Büchi, with the hope that a solution for both these models might lead to one for parity automata. However, in order to use a similar method in the co-Büchi case, we would want some way to check the winning condition for a play in the explorability game using only the projection of that play in the support arena. This is not possible with the current definitions of these games: we can create plays in the explorability game with the same projection, but different winners. Take the automaton from Figure 2.1. If we play the 2-explorability game on that automaton, Determiniser has a strategy to ensure that the support are always maximal, alternating between $\{q_0\}$ and $\{q_1, q_2\}$. However,

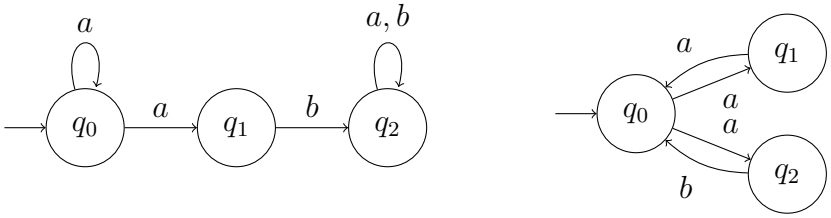


Figure 2.2: Two safety automata. Left: ω-explorable, but not explorable. Right: not ω-explorable.

Spoiler can either choose to always take the co-Büchi transition with the same token, or to alternate between tokens. He only wins in the second case.

2.4 Explorability with countably many tokens

In this section, we look at the same problem of explorability of an automaton, but we now allow for infinitely many tokens. More precisely, we will redefine the explorability game to allow an arbitrary cardinal for the number of tokens, then consider decidability problems regarding that game. This notion will mainly be interesting for automata on infinite words.

2.4.1 Definition and basic results

The following definition extends the notion of k-explorability to non-integer cardinals:

Definition (κ-explorability game)**:**
Consider an automaton $\mathcal{A}$ and a cardinal κ. The *κ-explorability game* on $\mathcal{A}$ is played on the arena $(Q_{\mathcal{A}})^{\kappa}$, between Determiniser and Spoiler. They play as follows.

- The initial position is S_0 associating q_0 to all κ tokens.
- At step i, from position S_{i-1}, Spoiler chooses a letter $a_i \in \Sigma$, and Determiniser chooses S_i such that for any token α, $S_{i-1}(\alpha) \xrightarrow{a} S_i(\alpha)$ is a transition in $\mathcal{A}$.

The play is won by Determiniser if for any $\beta \leq \omega$ such that the word $(a_i)_{1 \leq i < \beta}$ is in $\mathcal{L}(\mathcal{A})$, there is a token $\alpha \in \kappa$ building an accepting run, meaning that the sequence $(S_i(\alpha))_{i<\beta}$ is an accepting run. Otherwise, the winner is Spoiler.

We will say in particular that $\mathcal{A}$ is ω-explorable if Determiniser wins the game with ω tokens. We use here the notation ω for convenience, it should be understood as the countably infinite cardinal $\aleph_0$. We will however explicitly use the fact that such an amount of tokens can be labelled by $\mathbb{N}$, in order to describe strategies for Spoiler or Determiniser in the ω-explorability game. The following lemma gives a first few results on generalized explorability.

Lemma 2.12:

- Determiniser wins the explorability game on $\mathcal{A}$ with $|\mathcal{L}(\mathcal{A})|$ tokens.

- ω-explorability is not equivalent to explorability
- There are non ω-explorable safety automata.

Proof. For the first item, a strategy for Determiniser is to associate a token to each word of $\mathcal{L}(\mathcal{A})$ and to have it follow an accepting run for that word. Let us add a few details on the cardinality of $\mathcal{L}(\mathcal{A})$. First, a dichotomy result has been shown in [Niw91] (even in the more general case of infinite trees): if $\mathcal{L}(\mathcal{A})$ is not countable, then it has the cardinality of continuum, and this happens if and only if $\mathcal{L}(\mathcal{A})$ contains a non-regular word. In this case, we can simply associate a token with every possible run. In the other case where $\mathcal{L}(\mathcal{A})$ is countable, we have to associate an accepting run to each word, and this can be done without needing the Axiom of Countable Choice: a canonical run can be selected (*e.g.* lexicographically minimal).

We now want to prove that there are automata that are ω-explorable but not explorable. One such automaton is given in Figure 2.2 (left), where the rejecting sink state is omitted. Against any finite number of tokens, Spoiler has a strategy to eliminate them one by one, by playing a while Determiniser sends tokens to q_1, and b the first time q_1 is empty after the play of Determiniser. On the other hand, with tokens indexed by ω, Determiniser can keep the token 0 in q_0, and send the token i to q_1 at step i. Those strategies are winning, which proves both non explorability and ω explorability of the automaton.

The last item is proven by the second example from Figure 2.2. A winning strategy for Spoiler against countable tokens consists in labelling the tokens with integers, then targeting each token one by one (first token 0, then 1, 2, *etc.*). Each token is removed using the correct two-letters sequence (a, then b if the token is in q_1 or a if it is in q_2). With this strategy, every token is removed at some point, even if there might always be tokens in the game. □

The first item of Lemma 2.12 implies that the ω-explorability game only gets interesting when we look at automata over infinite words: since any language of finite words over a finite alphabet is countable, Determiniser wins the corresponding ω-explorability game. We will therefore focus on infinite words in the following.

Let us emphasize the following slightly counter-intuitive fact: in the ω-explorability game, it is always possible for Determiniser to guarantee that infinitely many tokens occupy each currently reachable state. However, even in a safety automaton, this is not enough to win the game, as it does not prevent that each individual token might be eventually "killed" at some point. As the following Lemma shows, this phenomenon does not occur in reachability automata.

Lemma 2.13:
Any reachability automaton is ω-explorable.

Proof. For every $w \in \Sigma^*$ such that there is a finite run ρ leading to an accepting state, Determiniser can use a single token following ρ. This token will accept all words of $w \cdot \Sigma^\omega$. Since Σ^* is countable, we only need countably many such tokens to cover the whole language, hence the result.

Let us give another equally simple view: a winning strategy for Determiniser in the ω-explorability game is to keep infinitely many tokens in each currently reachable state, as described above. Since acceptance in a reachability automaton is witnessed at a finite time, this strategy is winning. □

2.4.2 ExpTime algorithm for co-Büchi automata

We already know, from the example of Figure 2.2, that the result from Lemma 2.13 does not hold in the case of safety automata. However, we have the following decidability result, which talks about co-Büchi automata, and therefore still holds for safety automata as a subclass of co-Büchi.

Theorem 2.7:
The ω-explorability of co-Büchi automata is decidable in ExpTime.

To prove this result, we will use the following *elimination game*. $\mathcal{A}$ will from here on correspond to a co-Büchi (complete) automaton. We start by building a deterministic co-Büchi automaton $\mathcal{D}$ for $L(\mathcal{A})$ (*e.g.* using the breakpoint construction [MH84]).

Definition (Elimination game)**:**
The *elimination game* is played on the arena $\mathcal{P}(Q_\mathcal{A})\times Q_\mathcal{A}\times Q_\mathcal{D}$. The two players are named Protector and Eliminator, and the game proceeds as follows, starting in the position $(\{q_0^\mathcal{A}\}, q_0^\mathcal{A}, q_0^\mathcal{D})$.

- From position (B, q, p) Eliminator chooses a letter $a \in \Sigma$.
- If q is not a co-Büchi state, Protector picks a state $q' \in \Delta_\mathcal{A}(q, a)$.
- If q is a co-Büchi state, Protector picks any state $q' \in \Delta_\mathcal{A}(B, a)$. Such an event is called *elimination*.
- The play moves to position $(\Delta_\mathcal{A}(B, a), q', \delta_\mathcal{D}(p, a))$.

Such a play can be written $(B_0, q_0, p_0) \xrightarrow{a_1} (B_1, q_1, p_1) \xrightarrow{a_2} (B_2, q_2, p_2) \ldots$, and Eliminator wins if infinitely many q_i and finitely many p_i are co-Büchi states.

Intuitively, what is happening in this game is that Protector is placing a token that he wants to protect in a reachable state, and Eliminator aims at bringing that token to a co-Büchi state while playing a word of $L(\mathcal{A})$. If Protector eventually manages to preserve his token from elimination on an infinite suffix of the play, he wins.

Lemma 2.14:
The elimination game can be solved in polynomial time (in the size of the game).

Proof. To prove this result, we simply need to note that the winning condition is a parity condition of fixed index. If we label the co-Büchi states q_i with rank 1, the co-Büchi states p_i with rank 2, and the others with 3, then take the lowest rank in (B_i, q_i, p_i) (ignoring B_i), Eliminator wins if and only if the inferior limit of ranks is even. As any parity game with 3 ranks can be solved in polynomial time [Cal+17], this is enough to get the result. □

We want to prove the equivalence between this game and the ω-explorability game to obtain Theorem 2.7.

Lemma 2.15:
$\mathcal{A}$ is ω-explorable if and only if Protector wins the elimination game on $\mathcal{A}$.

Proof. First, let us suppose that Eliminator wins the elimination game on $\mathcal{A}$. To build a strategy for Spoiler in the ω-explorability game of $\mathcal{A}$, we first take a function $f : \mathbb{N} \to \mathbb{N}$ such that for any $n \in \mathbb{N}$, $|f^{-1}(n)|$ is infinite (for instance f is described by the sequence $0,\ 0,1,\ 0,1,2,\ 0,1,2,3,\ldots$). The strategy for Spoiler will focus on sending token $f(0)$, then $f(1)$, then $f(2)$, *etc.* to a co-Büchi state.

Let σ be a memoryless winning strategy for Eliminator in the elimination game (recall that parity games do not require memory [EJ91]). Spoiler will follow this strategy σ in the ω-explorability game, by keeping an imaginary play of the elimination game in his memory: $M = \mathcal{P}(Q_{\mathcal{A}}) \times Q_{\mathcal{A}} \times Q_{\mathcal{D}} \times \mathbb{N}$.

- At first, the memory holds the initial state $(\{q_0^{\mathcal{A}}\}, q_0^{\mathcal{A}}, q_0^{\mathcal{D}}, 0)$, and the current target is given by the last component: it is the token $f(0)$.
- From (B, q, p, n) Spoiler plays in both games the letter a given by σ.
- Once Determiniser has played, Spoiler sets the memory to $(\Delta_{\mathcal{A}}(B, a), q', \delta_{\mathcal{D}}(p, a), n)$ where q' is the new position of the token $f(n)$, except if q was a co-Büchi state, in which case we move to $(\Delta_{\mathcal{A}}(B, a), q', \delta_{\mathcal{D}}(p, a), n + 1)$ where q' is the new position of the token $f(n + 1)$. We then go back to the previous step.

This strategy builds a play of the elimination game in the memory, that is consistent with σ. We know that σ is winning, which implies that the word played is in $\mathcal{L}(\mathcal{A})$, and that every $n \in \mathbb{N}$ is visited (each elimination increments n, and there are infinitely many of those). An elimination happening while the target is the token $f(n)$ corresponds, on the exploration game, to that token visiting a co-Büchi state. Ultimately this means that Determiniser did not provide any accepting run, while Spoiler did play a word from $\mathcal{L}(\mathcal{A})$, and therefore won.

Let us now consider the situation where Protector wins the elimination game, using some strategy τ. We want to build a winning strategy for Determiniser in the ω-explorability game. Similarly, this strategy will keep track of a play in the elimination game in its memory. Determiniser will maintain ω tokens in any reachable state, while focusing on a particular token which follows the path of the current target in the elimination game. When that token visits a co-Büchi state, we switch to the new token specified by τ.

Since τ is winning in the elimination game, either the word played by Spoiler is not in $\mathcal{L}(\mathcal{A})$, which ensures a win for Determiniser, or there are no eliminations after some point, meaning that the target token at that point never visits another co-Büchi state, which also implies that Determiniser wins. □

With Lemmas 2.14 and 2.15 we get a proof of Theorem 2.7, since the elimination game associated to $\mathcal{A}$ is of exponential size and can be built using exponential time.

2.4.3 ExpTime-hardness of the ω-explorability problem

Theorem 2.8:
The ω-explorability problem for (any automaton model embedding) safety automata is ExpTime-hard.

We will prove the result in the case of safety automata, since a simple reduction can then generalize the result to any automaton model capable of simulating safety automata. This includes all acceptance conditions treated in this thesis: Büchi, co-Büchi, parity, *etc.* Note that a consequence of this result is the optimality of the algorithm provided in Section 2.4.2 to prove Theorem 2.7.

We will again give a proof sketch before the complete proof of this theorem.

Proof sketch

The main idea will be to reduce the acceptance problem of a PSpace alternating Turing machine (ATM) to the ω-explorability problem of some automaton that we build from the machine. This reduction is an adaptation of the one from [Ber+19] showing ExpTime-hardness of the NFA population control problem (defined in Section 2.3.1).

The computation of an ATM can be seen as a game between two players, who respectively aim for acceptance and rejection of the input. These players influence the output by choosing the transitions when facing a non-deterministic choice, that can belong to either one of them.

Let us first describe the automaton built in [Ber+19]. In that reduction, the choices made by the ATM players are translated into choices for Determiniser and Spoiler. The automaton has two main blocks: one dedicated to keeping track of the machine's configuration, which we call Config, and another focusing on the simulation of the ATM choices, which we call Choices. In Config, there is no non-determinism: the tokens move following the transitions of the machine given as input to the automaton. In Choices, Determiniser can pick a transition by sending his token to the corresponding state, while Spoiler uses letters to pick his.

The automaton constructed this way will basically read a sequence of runs of the ATM. At each run, some tokens must be sent into both blocks. Reaching an accepting state of a run lets Spoiler send some tokens from Choices to his target state, specifically those whose choices for the transitions of the ATM were followed. He can then restart with the remaining tokens until all are in the target. This process will ensure a win for Spoiler if he has a winning strategy in the ATM game. If he does not, then Determiniser can use a strategy ensuring rejection in the ATM game to avoid the configurations where he loses tokens, provided he starts with enough tokens.

This equivalence between acceptance of the ATM and the automaton being a positive instance of the PCP provides the ExpTime-hardness of their problem.

In our setup, getting rid of tokens one by one is not enough: Spoiler needs to be able to target a specific token and send it to the target state (which is now the rejecting state $\bot$) in one run. If he can do that, repeating the process for every token, without omitting any, ensures his win. If he cannot, then Determiniser has a strategy to pick a specific token and preserving it from $\bot$, and therefore wins.

This is why we adapt our reduction to allow Spoiler to target a specific token, no matter where it chooses to go. To do so, we change the transitions so that winning a run lets Spoiler additionally send every token from Config into $\bot$. With that and the fact that he can already target a token in Choices, we get a winning strategy for Spoiler when the ATM is accepting.

If the ATM is rejecting, Spoiler is still able to send some tokens to $\bot$, but he no longer has that targeting ability, which is how Determiniser is able to build a strategy preserving a specific token to win. To ensure the sustainability of this method, Determiniser needs to keep ω additional tokens following his designated token, so that he always has ω tokens to spread into the gadgets every time a new run starts.

Overall, we are able to compute in polynomial time from the ATM a safety automaton that is ω-explorable if and only if the ATM rejects its input. Since acceptance of a polynomial space ATM is known to be ExpTime-hard, we obtain Theorem 2.8.

Complete proof of Theorem 2.8

We reduce from the acceptance problem of a PSpace alternating Turing machine. This is again inspired from [Ber+19].

We take an alternating Turing machine $\mathcal{M} = (\Sigma_\mathcal{M}, Q_\mathcal{M}, \Delta_\mathcal{M}, q_0^\mathcal{M}, q_f^\mathcal{M})$ with $Q_\mathcal{M} = Q_\exists \uplus Q_\forall$. It can be seen as a game between two players: existential ($\exists$) and universal ($\forall$). On a given input, the game creates a run by letting $\exists$ (resp. $\forall$) solve the non-determinism in states from $Q_\exists$ (resp. $Q_\forall$) by picking a transition from Δ. Player $\exists$ wins if the play reaches the accepting state $q_f^\mathcal{M}$, and w is accepted if and only if $\exists$ has a winning strategy. We assume that $\mathcal{M}$ uses polynomial space $P(n)$ in the size n of its input, *i.e.* the winning strategies can avoid configurations with tape longer than $P(n)$. We also fix an input word $w \in (\Sigma_\mathcal{M})^*$.

We will assume for simplicity that $\Sigma_\mathcal{M} = \{0, 1\}$ and that the machine alternates between existential and universal states, starting with an existential one (meaning that $q_0 \in Q_\exists$ and the transitions are either $Q_\exists \to Q_\forall$ or $Q_\forall \to Q_\exists$). In our reduction, this will mean that we give the choice of the transition alternatively to Spoiler (playing $\exists$) and Determiniser ($\forall$).

We create a safety automaton $\mathcal{A} = (Q, \Sigma, q_0, \Delta, \bot)$ with:

- $Q = Q_\mathcal{M} \uplus \mathrm{Pos} \uplus \mathrm{Mem} \uplus \mathrm{Trans} \uplus \{q_0, \texttt{store}, \bot, \top\}$ where:

$$\begin{aligned} \mathrm{Pos} &= [1, P(n)] \\ \mathrm{Mem} &= \{m_{b,i} \mid b \in \{0,1\}, i \in [1, P(n)]\} \\ \mathrm{Trans} &= \{E\} \cup \{A_t \mid t \in \Delta_\mathcal{M}\} \end{aligned}$$

- $\Sigma = \{a_{t,p} \mid t \in \Delta_\mathcal{M} \text{ and } p \in [1, P(n)]\} \uplus \{\texttt{init}, \texttt{end}, \texttt{restart}, \texttt{win}\} \uplus \{\texttt{check}_q \mid q \in Q_\mathcal{M}\} \uplus \{\texttt{check}_{b,i} \mid (b, i) \in \{0,1\} \times [1, P(n)]\}$.

- $\bot$ is a rejecting sink state: a run is accepting if and only if it never reaches this state.

Let us give the intuition for the role of each state of $\mathcal{A}$. First, the states in $Q_\mathcal{M}$, Pos and Mem are used to keep track of the configuration of $\mathcal{M}$, as described in Lemma

2.16. Those in Trans are used to simulate the choices of $\exists$ and $\forall$ (played by Spoiler and Determiniser respectively). The state `store` keeps tokens safe for the remaining of a run when Spoiler decides to ignore their transition choice. The sinks $\top$ and $\bot$ are respectively the one Spoiler must avoid at all cost, and the one in which he wants to send every token eventually.

We now define the transitions in Δ. The states $\top$ and $\bot$ are both sinks ($\top$ accepting and $\bot$ rejecting). We then describe all transitions labelled by the letter $a_{t,p}$ with $p \in \text{Pos}$ and $t = (q, q', b, b', d) \in \Delta_{\mathcal{M}}$, where q and q' are the starting and destination states of t, while b and b' are the letters read and written at the current head position, and $d \in \{L, R\}$ is the direction taken by the head. These transitions are:

- $q \to q'$.
- $p \to p'$ with $p' = p + 1$ if $d = R$, or $p - 1$ if $d = L$. It goes to $\top$ if $p' \notin [1, P(n)]$.
- $m_{b,p} \to m_{b',p}$, and $m_{b'',p''} \to m_{b'',p''}$ for any b'' and any $p'' \neq p$.
- $E \to A_{t'}$ for any transition t'.
- $A_t \to E$.
- $q'' \to \top$ for any $q'' \neq q$.
- $m_{1-b,p} \to \top$ ($1 - b$ is the boolean negation of b).
- $p' \to \top$ for any $p' \neq p$.
- $A_{t'} \to \texttt{store}$ for any transitions $t' \neq t$.

The first three bullet points manage the evolution of the configuration of $\mathcal{M}$. The next two deal with the alternation between players, and the next three punish Spoiler if the transition is invalid (the `check` letters will handle the case where Determiniser is the one giving an invalid transition). The last one saves the tokens that are not chosen for the transition.

The other letters give the following transitions.

- `init` goes from q_0 to the states E, $q_0^{\mathcal{M}}$, and $1 \in \text{Pos}$, and also to the states $m_{b,i}$ corresponding to the initial content of the tape, *i.e.* all $m_{b,i}$ such that b is the i-th letter of w (or 0 if $i > |w|$).
- `end` labels transitions from any non-accepting state of $\mathcal{M}$ to $\top$, from `store` to q_0, and from any other state to $\bot$.
- $\texttt{check}_q$ creates a transition from A_t to $\bot$ for any $t \in \Delta$ starting from q. It also creates a transition from q to $\top$. Any other state is sent back to q_0. Intuitively, playing that letter means that q is not the current state and that any transition starting from q is invalid.

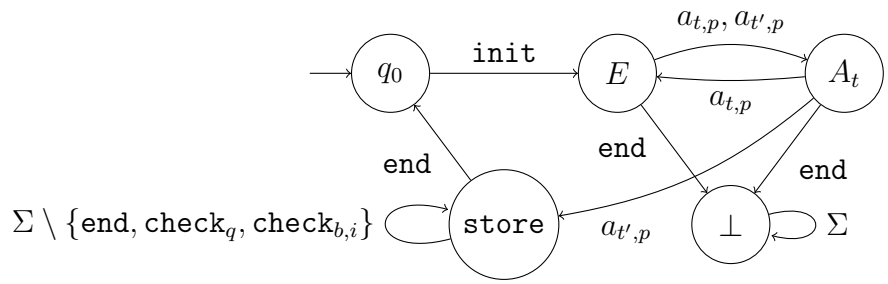


Figure 2.3: Gadget for simulating the choice of $\forall$ in the alternation (transitions labelled by `check` are not represented, and t' represents any transition different from t).

- $\texttt{check}_{b,i}$ creates a transition from A_t to $\bot$ for any $t \in \Delta$ reading b on the tape. It also creates transitions from any $j \in \text{Pos} \setminus \{i\}$ and from $m_{b,i}$ to $\top$. Any other state is sent to q_0. Intuitively, playing that letter means that the current head position is i, and that its content is not b, so any transition reading b is invalid.

To summarize, the states of $\mathcal{A}$ can be seen as two blocks, apart from q_0, $\top$ and $\bot$: those dealing with the configuration of $\mathcal{M}$ ($Q_\mathcal{M}$, Pos and Mem), and those from the gadget of Figure 2.3 which deal with the alternation and non-deterministic choices.

The following result provides tools to manipulate the relation between $\mathcal{A}$ and $\mathcal{M}$.

Lemma 2.16:
Let us consider a play of the ω-explorability game on $\mathcal{A}$, that we stop at some point. Suppose that the letters $a_{t,p}$ played since the last `init` are $a_{t_1,p_1}, \dots, a_{t_k,p_k}$. If $\top$ is not reachable from q_0 with this sequence, then we can define a run ρ of $\mathcal{M}$ on w taking the sequence of transitions $t_1, \dots, t_k$. The following implications hold:

Token present in	implies that at the end of ρ
$q \in Q_\mathcal{M}$	the current state is q
$p \in \text{Pos}$	the head is in position p
$m_{b,i} \in \text{Mem}$	the tape contains b at position i
E	it is the turn of $\exists$
A_t	it is the turn of $\forall$

Proof. These results are obtained by straightforward induction from the definitions. The unreachability of $\top$ is used to ensure that only valid transitions are played. □

We will now prove that $\mathcal{A}$ is ω-explorable if and only if the Turing machine $\mathcal{M}$ rejects the word w. Let us first assume that $w \in \mathcal{L}(\mathcal{M})$. There is a winning strategy $\sigma_\exists$ for $\exists$ in the alternating Turing machine game, and Spoiler will use that strategy in the explorability game to win against ω tokens. He will consider that the tokens are labelled by integers, and always target the smallest one that is not already in $\bot$. He proceeds as follows.

- Spoiler plays `init` from a position where every token is either in q_0 or $\bot$. We can assume from here that Determiniser sends tokens to each possible state, and just add imaginary tokens if he does not. Additionally, if the target token does not go to E, then Spoiler creates an imaginary target token in E that will play only valid

transitions (we will describe what this means later). Its purpose is to ensure that we actually reach an accepting state of $\mathcal{M}$ to destroy the real target token.

- When there are tokens in E, Spoiler plays letters according to $\sigma_\exists$. More formally, if the letters played since `init` are $a_{t_1,p_1} \dots a_{t_i,p_i}$, then Spoiler plays $a_{t_{i+1},p_{i+1}}$ where $t_{i+1} = \sigma_\exists(t_1, \dots, t_i)$ and $p_{i+1} = p_i + 1$ or $p_i - 1$ depending on the head movement in t_i.
- After such a play, Determiniser can move tokens to any state A_t. If there are more than one occupied state, Spoiler picks the one containing the current target token (possibly imaginary).
  - If that state corresponds to an invalid transition (wrong starting state or wrong tape content at the current head position), then Spoiler plays the corresponding `check` letter. Formally, if the target token (not the imaginary one, since Spoiler can avoid invalid transitions for that one) is in $\mathcal{A}_t$, Spoiler plays $\texttt{check}_q$ if the starting state q of t does not match the current state of the tape (given by Lemma 2.16), or $\texttt{check}_{b,i}$ if the current head position is i and does not contain b. In both cases, the target token is sent to $\bot$ with no other token reaching $\top$ (by Lemma 2.16). This sends us back to the first step, but with an updated target.
  - If the state instead corresponds to a valid transition, then Spoiler can play the corresponding $a_{t,p}$, where p is the current head position (again, given by Lemma 2.16), then go back to the previous step (where there are tokens in E).
- If no invalid transition is reached, the run eventually gets to an accepting state of $\mathcal{M}$ because $\sigma_\exists$ is winning. This corresponds to a stage where Spoiler can safely play `end` to get rid of the target token along with all tokens outside of `store`, by sending them to $\bot$ (the only reason not to play `end` would be the existence of tokens in non-accepting states of $Q_\mathcal{M}$). This sends us back to the first step, but with an updated target.

This strategy guarantees that after k runs, at least the first k tokens are in state $\bot$, and therefore cannot witness an accepting run. We also know that the final word is accepted by $\mathcal{A}$, because an accepting run can be created by going to the state `store` as soon as possible in each factor corresponding to a run of $\mathcal{M}$.

Conversely, if there is a winning strategy $\sigma_\forall$ for the universal player in the alternation game on $\mathcal{M}(w)$, then we can build a winning strategy for Determiniser in the ω-explorability game. This strategy is more straightforward than the previous one, as we can focus on the tokens sent to E (while still populating each state when `init` is played, but these other tokens follow a deterministic path until the next `init`).

Determiniser will initially choose a specific token, called leader. He then sends ω tokens to every reachable state when Spoiler plays `init`, with the leader going to E. Determiniser then moves the tokens in the leader's state according to $\sigma_\forall$. Spoiler cannot send the leader to $\bot$, since the only way to do that would be using the letter `end`, but this would immediately ensure the win for Spoiler, as there will always be some token in non-accepting states of $\mathcal{M}$ (because $\sigma_\forall$ is winning), and those tokens would be sent to $\top$

upon playing `end`. This means that Spoiler has no way to send the leader to $\bot$ without losing the game, and therefore that Determiniser wins.

Note that with that strategy, Spoiler can still safely send some tokens to $\bot$ by playing the wrong transition, which sends the tokens following the leader to `store`, then some well-chosen `check` letter to send the remaining ones to $\bot$. However, Determiniser will start the next run with still ω tokens, including the leader. This is why the choice of a specific leader is important, as it can never be safely sent to $\bot$.

This proves that the automaton $\mathcal{A}$ created from $\mathcal{M}$ and w (using polynomial time) is ω-explorable if and only if $\mathcal{M}$ rejects w. This completes the proof, since the acceptance problem is ExpTime-hard for alternating Turing machines using polynomial space.

2.5 Conclusion of Chapter 2

We introduced and studied the notions of explorability and ω-explorability, for automata on finite and infinite words. We showed that these problems are ExpTime-complete for Büchi condition in the first case and co-Büchi condition in the second case.

It is plausible that these results could be generalized to higher parity conditions, for instance by replacing the notion of support set by Safra trees, but this is outside the scope of this paper, and we leave this investigation for further research.

Although we showed that the original motivation of using explorability to improve the current knowledge on the complexity of the GFGness problem for all parity automata cannot be directly achieved, since deciding explorability is at least as hard as GFGness, we believe that explorability is a natural property in the study of degrees of nondeterminism, and that this notion could be used in other contexts as a middle ground between deterministic and non-deterministic automata. Moreover, a new objective from there might be to try using the hardness results from this thesis to get better lower bounds for GFG recognizability, in particular in the case of parity automata with at least three priorities (for which the upper bound is ExpTime, with no matching lower bound). The hardness result for explorability provides a closer candidate for reduction than most existing problems.

We did not focus on the decision problem of k-explorability, which already has lower and upper bounds inherited from the width problem in [KM19]. Those are respectively ExpTime and 2-ExpTime, instead of the ExpTime-completeness of width. This discrepancy is due to the lack of a bound on k in the case of explorability, which makes its size significant in the size of an instance of the problem. Some future work might be to tackle this problem in order to remove this gap between the two bounds.

Chapter 3

Cyclic proofs for transfinite expressions

You must obey the law,
because it's illegal to break the law.

Unknown

In this chapter, we introduce a cyclic proof system for proving inclusions of transfinite expressions, describing languages of words of ordinal length. We show that recognising valid cyclic proofs is decidable, that our system is sound and complete, and well-behaved with respect to cuts. Moreover, cyclic proofs can be effectively computed from expressions inclusions. We show how to use this to obtain a PSPACE algorithm for transfinite expression inclusion.

3.1 Introduction

Language inclusion. Deciding inclusion of regular languages is a fundamental problem in verification. For instance, if a program and a specification are modelled by regular languages P and S respectively, the correctness of the program is expressed by the inclusion $P \subseteq S$.

The most standard approach to deciding regular language inclusion is via automata, and this field of research is still active, see for instance [BP13] for well-performing nondeterministic automata inclusion algorithms using coinduction techniques. Language inclusion is especially important in the framework of infinite words. Indeed, the standard way to model possible behaviours of a system is via ω-regular languages. For instance, Linear Temporal Logic (LTL), which is a practical way to describe some ω-regular languages, is heavily used for expressing specifications. Inclusion of ω-regular languages is still being investigated, with recent works giving refined algorithms [Abd+10]. Finally, generalizing further, some models of automata and expressions defining languages of transfinite words (*i.e.* words of ordinal length) were studied in [Cho78; Bed96]. Transfinite expressions allow any nesting of Kleene star and ω-power. Such expressions define languages of transfinite words, for instance the expression $(a^+b^\omega)^\omega$ describes a language of words of length ω^2. This more general setting of transfinite words can be used for instance to model

phenomena with Zeno-type behaviours, such as a ball bouncing at smaller and smaller heights, and after infinitely many bounces it is considered stabilized and can perform some other action.

Proofs systems. The above algorithms give only a yes/no answer, but in some cases the user is interested in having a certificate witnessing inclusion, that he can check independently. This justifies the use of formal proof systems, where proofs can be easily communicated. On finite words, the seminal work [Koz94] gives a complete axiomatic system for regular expression inclusion. Complete axiomatisations for ω-regular expressions were given as well [CLS15].

Cyclic proofs systems. A proof is usually a finite tree with axioms as leaves, built using certain logical rules, and having the conclusion to prove as root. However, under certain conditions, we can consider that infinite trees form valid proofs. Such proofs are called non-well-founded, and can naturally express for instance reasoning by infinite descent. Many proof systems based on non-well-founded proofs were shown to be sound and complete in various frameworks, so these special proofs should be considered as a perfectly valid way of establishing a result. Such proof systems often require a validity condition on their infinite proofs, for instance of the form "on any infinite branch, such a rule must be used infinitely many times". Such a validity condition is often necessary to impose some kind of progress along the branches of the proof, in order to avoid proving false formulas by circular reasoning. These non-well-founded proofs have been studied in several contexts, such as arithmetic [Sim17], first-order logic [Bro05], modal μ-calculus [DHL06; AL17; Dou+16], LTL formulas [KS16], and others. Non-well-founded proofs are especially suited to reason about objects defined via fixed points. Since a non-well-founded proof is a priori an infinite object, it is often relevant to consider the special case of *cyclic* (or *regular*) proofs: those are the proofs obtained as the unfolding of finite graphs, so they are finitely describable.

One of the main advantages of moving to non-well-founded proofs is that in many cases it removes the need to guess invariants (or auxiliary lemmas). See for instance [BS07], where a cut-free completeness result is proved for a non-well-founded proof system. This makes the system more amenable to proof search: in most non-well-founded systems, we can prove any true formula φ using only formulas that are (in some sense) sub-formulas of φ. In a context where automated proof assistants such as Coq are becoming standard tools, this motivates the current growing interest in cyclic proofs.

Cyclic proofs for regular languages. Here, we aim at exploring the problem of language inclusion in the framework of cyclic proofs. Notice that the Kleene star is a least fixed point operator, and the ω power is a greatest fixed point, so we expect cyclic proofs to be well-suited to deal with regular expressions using these operators.

Das and Pous [DP18] explored this question in the context of finite words, with standard regular expressions whose only fixed point operator is the Kleene star. They exhibit a cyclic proof system for regular expression inclusion, that they prove sound and complete, even in its cut-free variant. To our knowledge, the cyclic proof approach to inclusion of regular expressions was not explored in the case of infinite and transfinite words, and this is the purpose of the present work.

Contributions. We design a non-well-founded proof system for the inclusion of transfi-

nite expressions. The notion of proof tree is replaced by a proof forest, whose branches can be of ordinal length. We show that our system is sound (in its most general version) and complete (even for cut-free cyclic proofs). We also show that the validity criterion for cyclic proofs is decidable. In the case of infinite words, our system is similar to systems for linear μ-calculus as introduced in [DHL06], except that we use hypersequents as in [DP18].

The main new difficulty when jumping from finite to infinite or transfinite words is the explosion in the number of non-deterministic choices one is faced with when trying to match a word to an expression. This explains the use of hypersequents, and leads to a slightly more intricate system than [DP18]. In the transfinite case, the branches of the proof tree become transfinite as well, thereby requiring additional care in the study of the system.

In order to prove the completeness of our system, we show that cyclic proofs can be effectively built from the expressions for which we want to prove inclusion. To show that the resulting proofs are correct, we use a model of automata (close to the one from [Cho78]) recognizing these transfinite languages. This allows us to show that cut-free, finitely representable proofs are enough to prove any true inclusion, and that these proofs can be computed.

The cyclic cut-free completeness of our system allows us to obtain a PSPACE algorithm for inclusion of transfinite expressions. This matches the known lower bound: inclusion of regular expressions is PSPACE-hard already for finite words. PSPACE membership is folklore for inclusion of ω-regular expressions as well, but to our knowledge, this upper bound is a new result for transfinite expressions. Let us note however that since automata models were already defined for transfinite expressions [Cho78; Bed96], it is plausible, that a PSPACE algorithm can also be obtained more directly through these models. This PSPACE-completeness result can be compared with the result from [DR10], stating PSPACE-completeness of LTL satisfiability on transfinite words.

Related works. In addition to related works that were already mentioned, let us comment on the link between our results and the recent paper [CR20], which studies cyclic proofs for first-order logic extended with least and greatest fixed points. The validity criterion in [CR20] is very similar to ours, and as they note, their general framework allows to embed reasonings on infinite words as a special case. One advantage of our system for ω-regular expressions is that although it is less general, it is much more convenient to manipulate ω-regular languages. Moreover, the use of hypersequents allows us to obtain cut-free regular completeness, which is not the case in [CR20]. On the other hand, our work on transfinite expressions is orthogonal to [CR20], as in such expressions, the ω operator is no longer a greatest fixed point.

Outline. We will start by describing the system for infinite words in Section 3.2, and first prove our results in this restricted case. We then show in Section 3.3 how the system can be modified to accommodate transfinite words, and how the results can be lifted to this setting.

3.2 The case of ω-regular expressions

In this part, we do not yet look at truly transfinite expressions such as $(a^+b^\omega)^\omega$, but only at ω-regular ones, which are the ones describing languages of words of length at most ω. More formally, these expressions can be described by the following grammar.

- Regular expressions: $e, f ::= a \mid e + f \mid e \cdot f \mid e^+$
- ω-regular expressions: $g, h ::= e \mid e^\omega \mid e \cdot g \mid g + h$, where e ranges over regular expressions.

To associate a language $\mathcal{L}(g)$ of finite or infinite words to an ω-regular expression g, it suffices to interpret each constructor on languages in the standard way:

$\mathcal{L}(0) = \emptyset$	$\mathcal{L}(e + f) = \mathcal{L}(e) \cup \mathcal{L}(f)$	$\mathcal{L}(e \cdot f) = \mathcal{L}(e) \cdot \mathcal{L}(f) = \{uv \mid u \in \mathcal{L}(e), v \in \mathcal{L}(f)\}$
$\mathcal{L}(a) = \{a\}$	$\mathcal{L}(e^+) = \bigcup_{n>0} \mathcal{L}(e)^n$	$\mathcal{L}(e^\omega) = \mathcal{L}(e)^\omega = \{u_1 u_2 \cdots \mid \forall i, u_i \in \mathcal{L}(e)\}$

We avoid the use of ε, and we use e^+ instead of e^*, to guarantee that an expression e^ω only accepts infinite words.

We design a proof system S_ω that will provide a certificate for any inclusion between the languages of two such expressions. Starting with the special case of ω-regular expressions allows us to introduce most proof techniques, while staying in a more familiar framework. We also claim that already in this case, such a proof system can bring new insights, as it can offer interesting trade-offs compared to automata models (see Conclusion).

3.2.1 The proof system S_ω

The proof system described in this section is strongly inspired from [DP17], the novelty being the introduction of ω.

Rules for building preproofs

We will first describe the sequents of the system S_ω, *i.e.* the shape of any label of a node in a proof tree. These are identical to the ones we use later, in the proof system for generalized expressions.

Definition (Sequent)**:**
We call *sequent* a pair (Γ, B), noted $\Gamma \to B$, where Γ is a list of expressions and B is a non-empty finite set of such lists. In the rest of the chapter, upper case Greek letters will be used for lists of expressions, and upper case Latin letters for sets of lists. Γ will be called the *left side* of the sequent and B its *right side*. Their contents will be denoted as follows, with brackets isolating each list in B:

$\Gamma = e_1, \ldots, e_n$ $\qquad$ $B = \langle f_1^1, \ldots, f_1^{k_1} \rangle; \ldots; \langle f_m^1, \ldots, f_m^{k_m} \rangle$

Languages are associated to such lists and sets of lists in the following way:

$\mathcal{L}(\Gamma) = \mathcal{L}(e_1 \cdot \ldots \cdot e_n)$ $\qquad$ $\mathcal{L}(B) = \mathcal{L}(f_1^1 \cdot \ldots \cdot f_1^{k_1} + \ldots + f_m^1 \cdot \ldots \cdot f_m^{k_m})$

The sequent $\Gamma \to B$ is called *sound* if the inclusion $\mathcal{L}(\Gamma) \subseteq \mathcal{L}(B)$ holds.

$$\frac{}{\to \langle\rangle}\ \text{id} \qquad \frac{e,\Gamma \to B \qquad f,\Gamma \to B}{e+f,\Gamma \to B}\ +\text{-l} \qquad \frac{\Gamma \to \langle e,\Lambda\rangle;\langle f,\Lambda\rangle;B}{\Gamma \to \langle e+f,\Lambda\rangle;B}\ +\text{-r}$$

$$\frac{\Gamma \to B}{\Gamma \to B;C}\ \text{wkn} \qquad \frac{\Gamma,e,f,\Lambda \to B}{\Gamma,e\cdot f,\Lambda \to B}\ \cdot\text{-l} \qquad \frac{\Gamma \to \langle\Lambda,e,f,\Theta\rangle;B}{\Gamma \to \langle\Lambda,e\cdot f,\Theta\rangle;B}\ \cdot\text{-r}$$

$$\frac{\Lambda \to \langle\Theta_1\rangle;\ldots;\langle\Theta_n\rangle}{\Gamma,\Lambda \to \langle\Gamma,\Theta_1\rangle;\ldots;\langle\Gamma,\Theta_n\rangle}\ \text{match} \qquad \frac{e,\Gamma \to B \qquad e,e^+,\Gamma \to B}{e^+,\Gamma \to B}\ *\text{-l} \qquad \frac{\Gamma \to \langle e,\Lambda\rangle;\langle e,e^+,\Lambda\rangle;B}{\Gamma \to \langle e^+,\Lambda\rangle;B}\ *\text{-r}$$

$$\left(\frac{\Lambda \to \langle e\rangle \qquad \Gamma,e,\Theta \to B}{\Gamma,\Lambda,\Theta \to B}\ \text{cut}\right) \qquad \frac{e,e^\omega \to B}{e^\omega \to B}\ \omega\text{-l} \qquad \frac{\Gamma \to \langle e,e^\omega\rangle;B}{\Gamma \to \langle e^\omega\rangle;B}\ \omega\text{-r}$$

Figure 3.1: The rules of the system S_ω for ω-regular expressions.
Γ,Λ,Θ are lists of expressions; B,C are sets of such lists; e,f are ω-regular expressions.
Rules wkn, match, cut will sometimes be abbreviated w,m,c.

Note that those sequents have more structure than those described in Section 1.6 (or in [DP18]). We need that additional structure to be able to deal with non-determinism in limit behaviours (*e.g.* $\mathcal{L}((a+b)^\omega) \subseteq \mathcal{L}((b^*a)^\omega + (b^*a)^*b^\omega)$). This kind of sequent is usually called hypersequent, but we will keep calling them sequents as those are the only ones we will use.

To describe our proof system, we now need to define the notion of proof tree. These are usually finite objects, but in our setting we allow infinite trees.

A *tree* is a non-empty, prefix-closed subset of $\{0,1\}^*$. We typically represent it with the root ε at the bottom, and the sons $v0$ and $v1$ of a node v (if they exist) are represented above v, respectively on the left and on the right.

A *branch* of a tree $T \subseteq \{0,1\}^*$ is a prefix-closed subset of T that do not contain two words of the same length, *i.e.* two nodes at the same depth of the tree. A branch of T is *maximal* if it is not strictly contained in another branch of T.

A *preproof* is given by a tree and a labelling π of its nodes by sequents in such a way that for any node v with children $v_1,\ldots,v_n$ (with $n \in \{0,1,2\}$), the expression $\dfrac{\pi(v_1) \quad \cdots \quad \pi(v_n)}{\pi(v)}$ is an instance of a rule from Figure 3.1.

A preproof is called *cyclic* or *regular* if it has finitely many distinct subtrees. Such a proof can be represented using a finite tree, where each leaf x not closed with an id rule is equipped with a pointer to a node y below x, indicating that the infinite trees rooted in x and y are identical. Examples of this representation can be found in Figure 3.2.

Threads and validity condition

Some preproofs satisfying the conditions described above actually prove wrong inclusions, meaning that we can build such a tree with an unsound sequent at its root. An example of such a preproof can be found in Figure 3.2. This illustrates the need for a validity condition that will rule out such unsound preproofs. We need a few more definitions before we can state this validity condition.

Definition (Occurrences)**:**
We will need to consider particular occurrences of expressions in preproofs. To this end, we define the notion of *occurrence* of an expression in a preproof, which is formally defined as a tuple (v, i, j) where $v \in \{0,1\}^*$ is a node of the tree, i is the index of a list in the sequent labelling that node, and j the index of an expression in that list. This means we actually need to represent the right side of any sequent by a sorted set of lists of expressions, using some arbitrary order over lists. In the rest of the chapter, in order to lighten notations, we will abstract away this formalism and just use the word "expression" to point to particular occurrences of expressions in a preproof.

If S is a sequent, we will note $pos(S)$ the set of expression positions in S, that is identifiers (i, j) that are compatible with the sequent S.

Definition (Principal expression)**:**
In a sequent of a preproof where a rule r is applied, an expression is called *principal* for r if it is the one corresponding to the lower case expression in the lower side of the rule r in Figure 3.1. Note that there is no principal expression when the rule is id, wkn, cut or match, since these rules do not contain lower case letters in the lower sequent.
Ancestors: Given an expression e in the lower part of a rule, its *immediate ancestors* are:

- if e is principal: the lower case expressions in the upper sequents of the rule
- if e is in a list Γ or a set of list B: its copies in the same position in each copy of Γ (resp. B) on the upper sequents.

Note that an expression can have between 0 (expression in C in the wkn rule) and 3 (e^+ in any $*$ rule) immediate ancestors.

Definition (Thread)**:**
A *thread* is a path in the graph of immediate ancestry (also called the logical flow graph [Bus91]). We say that a thread witnesses a *v-unfolding* if the current expression is principal for either a $*$-l rule or an ω-r rule. As in Figure 3.2, threads will be represented by coloured lines, with bullets to mark v-unfoldings.

Note that we purposely talk about the "graph" of immediate ancestry, and not the "tree". Since the right part of a sequent is a set, it does not keep track of multiplicity, and two threads can merge when going upwards. For instance, if we apply the rule $\dfrac{\Gamma \to \langle e, e^\omega \rangle}{\Gamma \to \langle e^\omega \rangle; \langle e, e^\omega \rangle}\ \omega\text{-r}$, the red and blue threads are merged. We need to allow that phenomenon in order to be able to build finitely representable proofs.

We can now define the *validity condition*, that makes a preproof into an actual proof.

Definition (Validity condition)**:**
A thread is *validating* if it witnesses infinitely many v-unfoldings. A preproof is *valid*, and is then called a *proof*, if all its infinite branches contain a validating thread.

We will call *$*$-l thread* (resp. *ω-r thread*) a validating thread on the left side (resp. right side) of sequents, as it witnesses infinitely many $*$-l (resp. ω-r) rules.

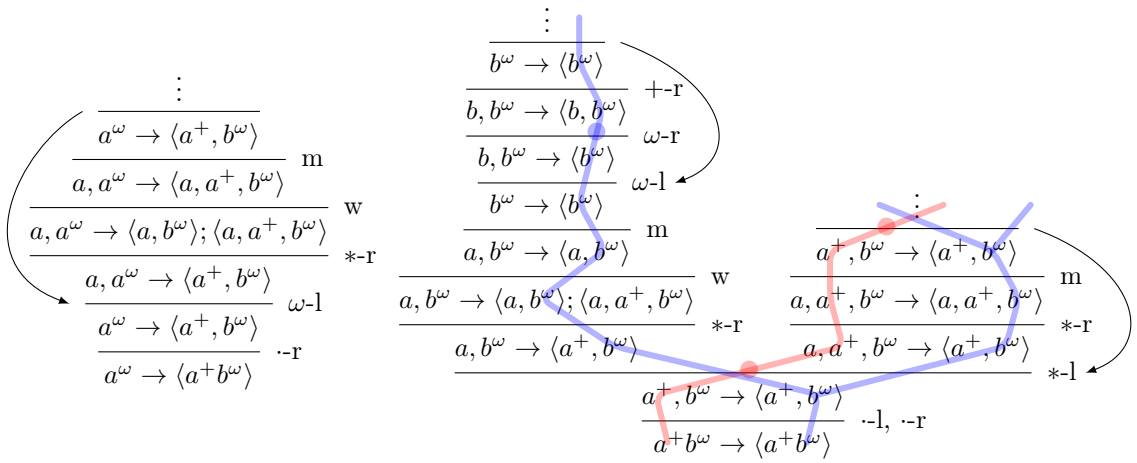


Figure 3.2: An invalid preproof (left) and a valid one (right)

Let us give an intuition for this validity condition. A proof has to guarantee that any word generated by the left side expression can be parsed in the right side one. Branches with a $*$-l thread do not correspond to a word on the left side, so there is nothing to verify and the branch can be accepted. On the other hand, when a legitimate infinite word from the left side has to be parsed on the right side, it must involve an expression e^ω where e is matched to infinitely many factors. This corresponds to an ω-r thread.

We give two examples of preproofs in Figure 3.2. The left one is an invalid preproof of a wrong inclusion. The validity condition is not satisfied, since there are no $*$-l or ω-r rules.

The right one is an actual proof. It is comb-shaped, with a "main" branch always going to the right. We can get a validating thread for any branch of that preproof, by taking the red thread on the rightmost branch, and a blue thread on all other branches.

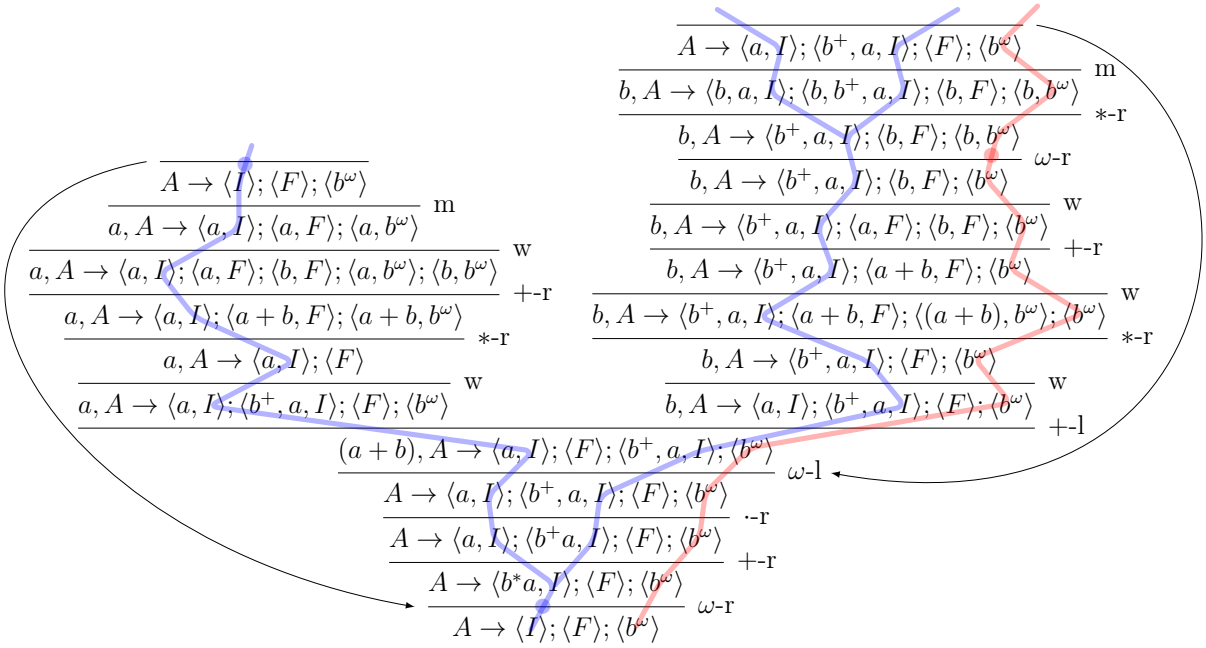
We provide an example of a non-trivial inclusion in Example 3.1 below.

Example 3.1 (A cyclic proof of a non-trivial inclusion)**:**
The proof in Figure 3.3 shows that $\mathcal{L}((a+b)^\omega) \subseteq \mathcal{L}((b^*a)^\omega + (a+b)^*b^\omega)$, *i.e.* any infinite word on alphabet $\{a, b\}$ has either infinitely or finitely many a's. Black arrows, called *backpointers*, identify sequents that are roots of isomorphic subtrees. To simplify the reading, we temporarily suspend our naming convention regarding capital letters, and we note $A = (a+b)^\omega$ (for "All"), $I = (b^*a)^\omega$ (for "Infinite"), and $F = (a+b)^+, b^\omega$ (for "Finite"). Note that our expressions cannot use the Kleene star : b^*a is just a shorthand for $b^+a + a$, and that is why $(a+b)^*b^\omega$ is represented by $\langle F \rangle; \langle b^\omega \rangle$.

This proof forest is valid: the only options for limit branches are either going only right after some point, or going left infinitely many times. In the first case, following the expression b^ω (in red) gives us an ω-r thread, and in the second one we get such a thread by following the sequent I (blue).

We recall the notions of soundness and completeness, which will be our main objective with the systems presented here.

Definition (Soundness and completeness)**:**



$A = (a + b)^\omega$, $I = (b^*a)^\omega$, and $F = (a + b)^+, b^\omega$

Figure 3.3: A (valid) proof tree

A proof system is sound if the conclusions of all of its valid proofs are sound. It is complete if, for any sound sequent $\Gamma \to B$, there is a proof with conclusion $\Gamma \to B$.

3.2.2 Soundness of the system S_ω

In this part, we want to prove that any proof (*i.e.* any valid preproof) derives a sound sequent. We will do that without any assumption of regularity, since we want every proof from the S_ω system to be correct, and not just the regular fragment. We will show soundness of the system with cuts, since this is more general and it allows to write proofs more conveniently. Notice that incorporating the cuts significantly increases the difficulty: unlike what happens in a finitary proof system, it is not enough here to prove that the cut rule is locally sound. Since the cuts can be used infinitely many times along a branch, it calls for a careful argument.

The following first result is an easy consequence of the local soundness of our rules, obtained by induction on the tree:

Lemma 3.1:
Any finite preproof derives a sound sequent.

To prove the general case, we take any valid proof tree P in S_ω, with a root sequent $\Gamma_0 \to B_0$. We take an arbitrary $w \in \mathcal{L}(\Gamma_0)$, and we show that $w \in \mathcal{L}(B_0)$.

We create a tree $P(w)$ that will be a subtree of the original one, with additional information labelling its nodes. The purpose of the tree $P(w)$ is to prove the membership of w in $\mathcal{L}(B_0)$.

Definition (Sequents of $P(w)$, label-soundness)**:**
The sequents of $P(w)$ are similar to the ones of a preproof, but we additionally label each expression e on the left side of sequents with a word u. Given a list of expressions Γ, we will denote Γ' a labelling of its expressions with words, represented as a list of pairs (expression, word). If $\Gamma' = (e_1, u_1), \dots, (e_k, u_k)$, we say that (the labelling of) Γ' is *correct* if for each $i \in [1, k]$, we have $u_i \in \mathcal{L}(e_i)$. We define $\mathrm{concat}(\Gamma')$ as the word $u_1 \dots u_k$. We additionally say that a sequent $\Gamma' \to B$ is *label-sound* if $\mathrm{concat}(\Gamma') \in \mathcal{L}(B)$.

We will build $P(w)$ by transforming the initial proof by induction from the root. First, we take a correct labelling Γ'_0 of Γ_0 such that $\mathrm{concat}(\Gamma'_0) = w$. Then we move upwards while replacing each rule by the corresponding one in the table below, while satisfying the condition specified in the table (if possible). This tree will be a subtree of the initial one, since we only keep one successor at rules $+$-l and $*$-l.

Rule	New rule	Condition
$\dfrac{\Gamma, e, f, \Lambda \to B}{\Gamma, e \cdot f, \Lambda \to B}$ $\cdot$-l	$\dfrac{\Gamma', (e, u), (f, v), \Lambda' \to B}{\Gamma', (e \cdot f, uv), \Lambda' \to B}$ $\cdot$-l	$(u, v) \in \mathcal{L}(e) \times \mathcal{L}(f)$
$\dfrac{\Gamma, e_1, \Lambda \to B \quad \Gamma, e_2, \Lambda \to B}{\Gamma, e_1 + e_2, \Lambda \to B}$ $+$-l	$\dfrac{\Gamma', (e_i, u), \Lambda' \to B}{\Gamma', (e_1 + e_2, u), \Lambda' \to B}$ $+$-l	$u \in \mathcal{L}(e_i)$
$\dfrac{e, \Gamma \to B \quad e, e^+, \Gamma \to B}{e^+, \Gamma \to B}$ $*$-l	$\dfrac{(e, u), \Gamma \to B}{(e^+, u), \Gamma \to B}$ $*$-l	$u \in \mathcal{L}(e)$
$\dfrac{e, \Gamma \to B \quad e, e^+, \Gamma \to B}{e^+, \Gamma \to B}$ $*$-l	$\dfrac{(e, u), (e^+, v), \Gamma' \to B}{(e^+, uv), \Gamma' \to B}$ $*$-l	$(u, v) \in \mathcal{L}(e) \times \mathcal{L}(e^+)$
$\dfrac{\Gamma, e, e^\omega \to B}{\Gamma, e^\omega \to B}$ ω-l	$\dfrac{\Gamma', (e, u), (e^\omega, v) \to B}{\Gamma', (e^\omega, uv) \to B}$ $*$-l	$(u, v) \in \mathcal{L}(e) \times \mathcal{L}(e^\omega)$
$\dfrac{\Lambda \to \langle e \rangle \quad \Gamma, e, \Theta \to B}{\Gamma, \Lambda, \Theta \to B}$ c	$\dfrac{\Lambda' \to \langle e \rangle \quad \Gamma', (e, u), \Theta' \to B}{\Gamma', \Lambda', \Theta' \to B}$ c	$u = \mathrm{concat}(\Lambda')$

Notice that if the labelling of the bottom sequent is correct, this guarantees us that we can choose a correct labelling for the upper sequent as well, while satisfying the condition in the table. It is only because of the cut rule that we cannot simply propagate correctness of labellings from the root. Moreover, all these rules are label-sound, in the sense that their lower sequents are label-sound whenever the upper ones are.

If at some point the condition cannot be met, we stop there and call the current node a dead leaf. This is only important for the sake of a complete definition, since we will prove that this actually never occurs if the initial preproof is valid (Lemma 3.5).

We deal with the remaining rules (right rules, wkn and match) by simply copying the pairs (expression, word) from bottom to top on the left side.

Lemma 3.2:
In $P(w)$, any infinite branch has an ω-r thread.

Proof. This follows from the fact that any expression e^+ on the left of a sequent in $P(w)$ is associated with a finite word, and therefore can only be principal for finitely many $*$-l rules, each one decreasing the size of that word (notice that we use an infinite descent argument here). The validity condition then ensures the result. □

Lemma 3.3:
In $P(w)$, no branch goes infinitely many times to the left at cut rules.

Proof. Let us consider an infinite branch β of $P(w)$. By Lemma 3.2, the branch β has an ω-r thread. Since the right side of a sequent is not preserved when going to the left at a cut rule, it can only happen finitely many times in β. □

If β_1, β_2 are maximal branches of $P(w)$, we note $\beta_1 < \beta_2$ if β_1 is to the left of β_2, *i.e.* at the first cut where they differ, β_1 goes left and β_2 goes right. We call $\leq$ the reflexive closure of $<$. We then get the following result.

Lemma 3.4:
The maximal branches of $P(w)$ are well-ordered by $\leq$.

Proof. Let us call C the set of maximal branches in $P(w)$, ordered from left to right. More formally, we can see an element of C as a (finite or infinite) word over $\{0, 1\}$, in which each letter corresponds to the choice made at a cut rule: 0 for left and 1 for right (cut rules are the only branching rules in $P(w)$). With this, the left-to-right order is simply the lexicographic order: $\beta_1 < \beta_2$ if their first different bit is 0 in β_1 and 1 in β_2.

We take a nonempty subset $X \subseteq C$. Consider the subtree T_X of $P(w)$ formed by the branches in X. Note that this tree could contain branches that are not in X. For instance, if $X = \{0, 10, 110, 1110, \ldots\}$, then T_X also contains the branch $11111\ldots$, noted 1^ω. We call m the leftmost branch in T_X: we can define it by induction, by going left whenever possible. We want to show that $m \in X$. If $m = 1^\omega$, then clearly $m \in X$. Otherwise, by Lemma 3.3, m can be noted $\tau 01^\omega$. There must be a branch $\beta \in X$ starting with $\tau 0$ as well. By minimality of m, we have $m \leq \beta$, and since $m = \tau 01^\omega$, we have $m \geq \beta$. Therefore $m = \beta$, and $m \in X$.

We finally get that m is the smallest element in X. Any subset of C has a smallest element, so $(C, \leq)$ is well-founded. □

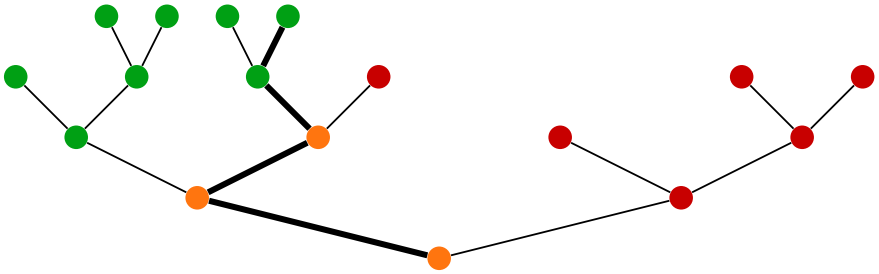
The main result for the soundness proof is Lemma 3.5, which also ensures that there is no dead leaf in $P(w)$.

Lemma 3.5:
In $P(w)$, all labellings are correct and all sequents are label-sound.

Proof. We first give the main idea, see below for the detailed proof. We proceed by well-founded induction on the maximal branches, using the left-to-right order from Lemma 3.4. The only interesting case is when dealing with cuts.

When encountering a cut rule of the form $\dfrac{\Lambda' \to \langle e \rangle \qquad \Gamma', (e, u), \Theta' \to B}{\Gamma', \Lambda', \Theta' \to B}$ cut , we need to show that the label (e, u) is correct, *i.e.* provided $u \in \mathcal{L}(\Lambda')$, we have $u \in \mathcal{L}(e)$. This will be obtained thanks to the induction hypothesis, guaranteeing that the sequent $\Lambda' \to \langle e \rangle$ on the left of the cut rule is label-sound.

Let us now write the complete proof. We proceed by well-founded induction on the maximal branches. The property we want to prove is the following one: "Given a maximal branch β of $P(w)$, any labelling on β is correct. Moreover, any sequent above the last left cut of β is label-sound."



We only show the branchings from cut rules is this drawing of $P(w)$. The thick branch is the induction step we just completed. Green nodes correspond to sequents that have been proven both correctly labelled and label-sound, orange ones those we only proved to be correctly labelled at that point, and red ones are the remaining ones.

Figure 3.4: Illustration of the proof of Lemma 3.5.

Assume that, for some maximal branch β, every other one on its left verifies the property.

We prove by induction that the labellings of β are correct. By definition of $P(w)$, this is preserved when we go upwards at any rule other than cut. This is also the case when we go on the left of a cut rule. We need to prove that it still works when we go on the right, in a rule such as: $\dfrac{\Lambda' \to \langle e \rangle \qquad \Gamma', (e,u), \Theta' \to B}{\Gamma', \Lambda', \Theta' \to B}$ cut with $u = \mathrm{concat}(\Lambda')$.

In this instance, $u \in \mathcal{L}(e)$ if and only if the sequent $\Lambda' \to \langle e \rangle$ is label-sound (because u is the word in Λ' by definition of $P(w)$). To prove that it is the case, we use the induction hypothesis for the maximal branch going to that sequent, then always on the right at cut rules. This branch is on the left of the current one, so according to the second part of the induction hypothesis for this branch, the sequent $\Lambda' \to \langle e \rangle$ is label-sound. This completes the proof of the first part of the induction.

In order to prove the second part, we call v the node just above the last left cut of β. We are only interested in the proof rooted in v.

The branch β is the rightmost branch in that subtree. This means that if we take any sequent S of that tree that is not in β, the branch going only right from S guarantees the label-soundness of S by induction hypothesis. An illustration of this situation is provided in Figure 3.4. Consequently, the label-soundness of a sequent in β above v implies that of the sequent below.

If β is finite, we can get the label-soundness of any sequent above v by induction.

Suppose now that β is infinite, and therefore validated by an ω-r thread (Lemma 3.2). In other words, the corresponding expression f^ω on the right is unfolded infinitely many times. Figure 3.5 presents what happens between two consecutive ω-r rules applied to that thread. We get the finite proof on the right by removing left rules and changing the context. A segment of the word on the left has been matched to f by finite soundness (Lemma 3.1), because we already proved that $u_i \in \mathcal{L}(e_i)$ if the couple appears on that branch.

In the end, we found a match for every iteration of f, which proves that all sequents of β above the first ω-r rule in the proof rooted in v are label-sound. We get the label-

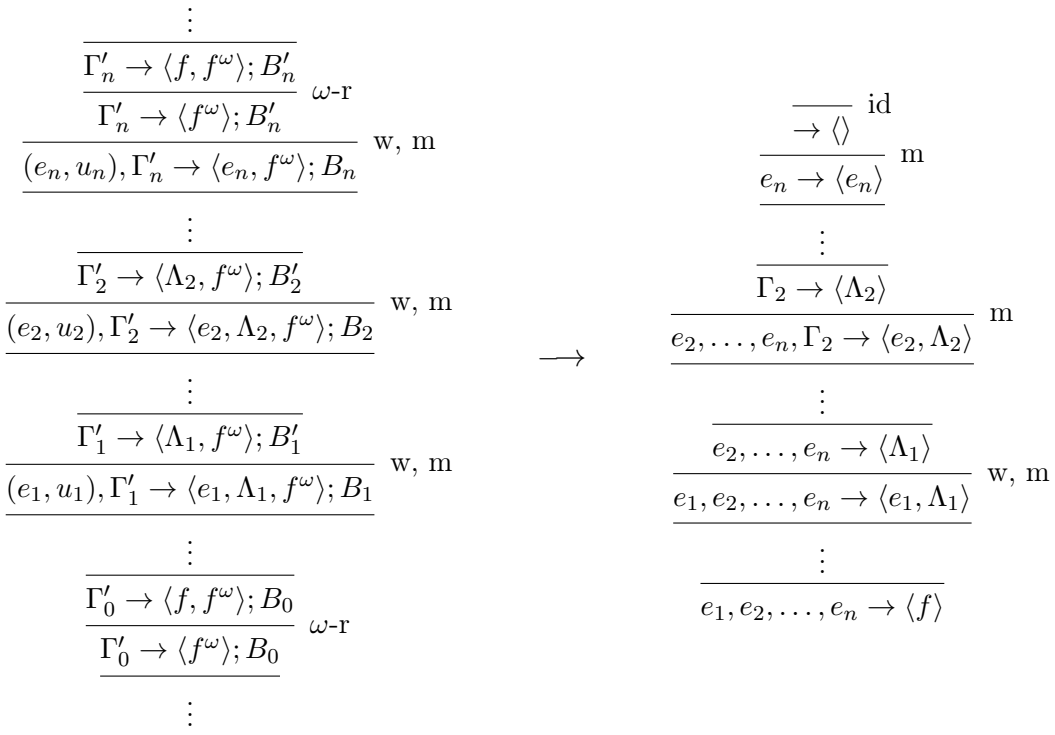


Figure 3.5: Using finite soundness between two consecutive ω-r rules in $P(w)$

soundness of the remaining interval (between v and the first ω-r rule) by straightforward induction using the local label-soundness of the rules in this branch.

This completes the last part of the induction hypothesis. By well-founded induction, this is true for every maximal branch. In particular, this is true for the very last one: the one that always goes on the right at a cut rule. The induction hypothesis for this branch implies that the whole proof tree $P(w)$ is label-sound, and that $w \in \mathcal{L}(B_0)$ (recall that the root of the initial proof is $\Gamma_0 \to B_0$, with $w \in \Gamma_0$). □

Theorem 3.1:
Any valid proof from S_ω is sound.

Proof. For any word w in $\mathcal{L}(\Gamma_0)$, there is a way to correctly label Γ_0 into a Γ'_0 with $\mathrm{concat}(\Gamma'_0) = w$. By Lemma 3.5, this correct labelling can be propagated through $P(w)$, and we obtain that the root sequent $\Gamma'_0 \to B_0$ is label-sound, so $w \in \mathcal{L}(B_0)$. Thus we have indeed $\mathcal{L}(\Gamma_0) \subseteq \mathcal{L}(B_0)$, showing that any valid proof is sound. □

Remark 3.1:
This result is similar to the soundness in [DHL06], but the shape of the sequents differs, and the transfinite case that follows uses an extension of our proof.

3.2.3 Cut-free regular completeness of the system S_ω

In order to prove the completeness of the system, we want to show that any sound sequent can be derived. Moreover, if we want this system to be interesting from a computational point of view, we need to obtain finitely representable proofs. Lemma 3.7 will help us do that by ensuring a finite number of different sequents in a proof. Then we will build a regular proof via a deterministic saturation process. However, we first need to define the following closure that will help control the number of different expressions that can appear in a proof.

Definition (Fischer-Ladner closure)**:**
The *Fischer-Ladner closure* $\mathcal{C}$ of an expression is defined by induction as follows:

- $\mathcal{C}(a) = \{a\}$
- $\mathcal{C}(e + f) = \mathcal{C}(e) \cup \mathcal{C}(f) \cup \{e + f\}$
- $\mathcal{C}(e \cdot f) = (\mathcal{C}(e) \cdot \{f\}) \cup \mathcal{C}(f)$
- $\mathcal{C}(e^+) = \mathcal{C}(e) \cup (\mathcal{C}(e) \cdot \{e^+\}) \cup \{e^+\}$
- $\mathcal{C}(e^\omega) = (\mathcal{C}(e) \cdot \{e^\omega\}) \cup \{e^\omega\}$

where $P \cdot Q$ stands for $\{u \cdot v \mid u \in P \text{ and } v \in Q\}$.

This is not exactly the original definition of the Fischer-Ladner closure. We adapted it to better fit the unfolding from the left of our system, yet the main idea remains the same. This closure respects the following property, proven by a straightforward induction.

Lemma 3.6:
The Fischer-Ladner closure satisfies the following properties.

- The Fischer-Ladner closure of an expression is finite.
- If $f \in \mathcal{C}(e)$, then $\mathcal{C}(f) \subseteq \mathcal{C}(e)$.

With that, we can now bound the number of different sequents in a proof:

Lemma 3.7:
In a preproof without cut, there can only be finitely many different sequents.

Proof. Let us call $\text{expr}(\Gamma)$ the expression formed by concatenating the expressions in Γ, and similarly $\text{expr}(B)$ is the expression $\bigcup_{\Gamma \in B} \text{expr}(\Gamma)$. We can verify that for every rule except cut, if $\Gamma_0 \to B_0$ is the conclusion sequent and if $\Gamma_1 \to B_1$ is a premise sequent, then $\text{expr}(\Gamma_1)$ is in the Fischer-Ladner closure of $\text{expr}(\Gamma_0)$, and similarly $\text{expr}(B_1)$ is in the closure of $\text{expr}(B_0)$. We can also note that given an expression, there are only finitely many ways to subdivide it into a list or into a set of lists, which gives us finitely many possible sequents using Lemma 3.6. Notice that we rely here on the fact that we allow unfolding of $\cdot^+$ and $\cdot^\omega$ only at the beginning of a list, thereby preventing multiple consecutive unfoldings of the same expression. □

Outermost operation	Left side	Right side
$+$	$\dfrac{e, \Gamma \to B \qquad f, \Gamma \to B}{e + f, \Gamma \to B}$ $+$-l	$\dfrac{a, \Gamma \to \langle e, \Lambda\rangle; \langle f, \Lambda\rangle; B}{a, \Gamma \to \langle e + f, \Lambda\rangle; B}$ $+$-r
$\cdot$	$\dfrac{e, f, \Gamma \to B}{e \cdot f, \Gamma \to B}$ $\cdot$-l	$\dfrac{a, \Gamma \to \langle e, f, \Lambda\rangle; B}{a, \Gamma \to \langle e \cdot f, \Lambda\rangle; B}$ $\cdot$-r
$.^+$	$\dfrac{e, \Gamma \to B \qquad e, e^+, \Gamma \to B}{e^+, \Gamma \to B}$ $*$-l	$\dfrac{a, \Gamma \to \langle e, \Lambda\rangle; \langle e, e^+, \Lambda\rangle; B}{a, \Gamma \to \langle e^+, \Lambda\rangle; B}$ $*$-r
$.^\omega$	$\dfrac{e, e^\omega \to B}{e^\omega \to B}$ ω-l	$\dfrac{a, \Gamma \to \langle f, f^\omega\rangle; B}{a, \Gamma \to \langle f^\omega\rangle; B}$ ω-r

Figure 3.6: Invertible rules of the system, without the match rule

We will now take a sound sequent, and build a preproof using only invertible instances of our rules, *i.e.* rules that are locally sound in both directions: the premises are true if and only if the conclusion is true. We proceed by induction on the outermost operation of the first expression of a list.

We first apply greedily the invertible rules from Figure 3.6. Notice that at each step, the first expression of the list becomes a (strict) subexpression of the previous one. Since the subexpression relation is well-founded, we must at some point obtain a finite tree with leaves of the form $a, \Gamma \to \langle a_1, \Gamma_1\rangle; \ldots; \langle a_n, \Gamma_n\rangle$ (with a and a_i letters). Moreover, each of those leaves are sound sequents, since all the rules we applied were invertible. For each leaf of this form, we can now remove each $\langle a_i, \Gamma_i\rangle$ with $a_i \neq a$ using the wkn rule, then match the remaining as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \to \langle \Gamma_{i_1}\rangle; \ldots; \langle \Gamma_{i_k}\rangle}}{a, \Gamma \to \langle a, \Gamma_{i_1}\rangle; \ldots; \langle a, \Gamma_{i_k}\rangle}\ \text{match}}{a, \Gamma \to \langle a_1, \Gamma_1\rangle; \ldots; \langle a_n, \Gamma_n\rangle}\ \text{wkn}}{\vdots}$$

Since the bottom sequent is sound, we know that the top one is too (we only removed useless options). We can therefore repeat the process to get an infinite tree with only identity rules at the leaves. Any sequent in this tree is sound by a straightforward induction. We now need to check that this is a valid tree.

First note that, as this process will always reach a match rule in a finite number of steps, any infinite branch passes through infinitely many match rules, therefore processing an ω-word (every match rule corresponds to a new letter in the word). In other words, to each infinite branch β of the preproof, we can associate an infinite word $word(\beta)$ corresponding to the sequence of match rules performed along β.

Lemma 3.8:
If β is an infinite branch starting in the root sequent $\Gamma_0 \to B_0$, then either β contains a $*$-l thread, or $word(\beta) \in \mathcal{L}(\Gamma_0)$.

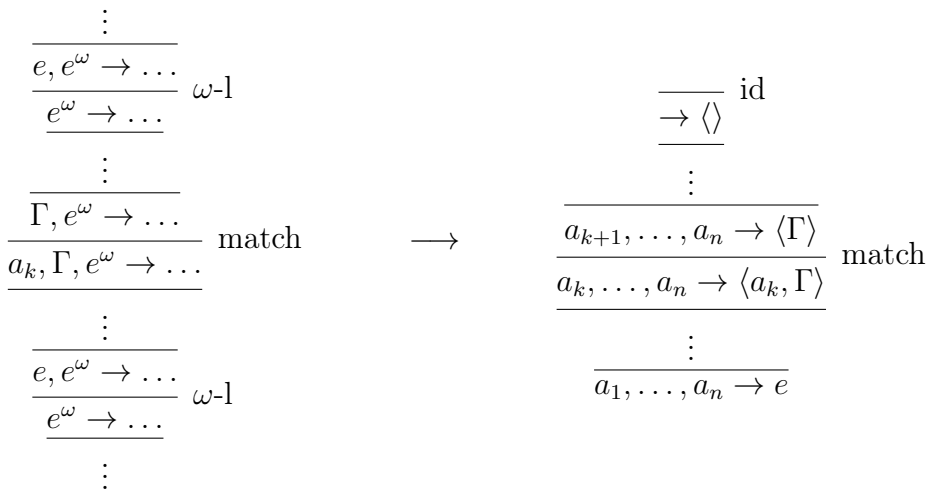


Figure 3.7: Using finite soundness between two consecutive ω-l rules

Proof. If the branch β does not have a $*$-l thread, then each expression $\cdot^+$ is unfolded finitely many times, meaning that there are necessarily infinitely many ω-l rules (to avoid depletion of the list). These rules apply to the same expression e^ω, because the syntax of the rule ω-l does not allow another expression $\cdot^\omega$ on the left of the sequent. This corresponds to unfolding infinitely many times this expression.

If we call w the word read between two consecutive instances of ω-l rules, then we want to prove that $w \in \mathcal{L}(e)$. We can do that with a simple transformation of that segment of a branch, which gets rid of e^ω and transfers the expression e from the left of the sequent to its right. The transformation changes each left rule into its right counterpart (with possibly a wkn rule), according to the following table. It keeps the match rules and ignores any other rules. This process is described in Figure 3.7.

Initial left rule	Condition	New right rule
$\dfrac{\Gamma, e, f, \Lambda, e^\omega \to B}{\Gamma, e \cdot f, \Lambda, e^\omega \to B}$ $\cdot$-l	None	$\dfrac{u \to \langle \Gamma, e, f, \Lambda \rangle}{u \to \langle \Gamma, e \cdot f, \Lambda \rangle}$ $\cdot$-r
$\dfrac{\Gamma, e_0, \Lambda \to B \quad \Gamma, e_1, \Lambda \to B}{\Gamma, e_0 + e_1, \Lambda \to B}$ $+$-l	Branch goes to side i	$\dfrac{u \to \langle e_i, \Gamma \rangle}{u \to \langle e_0 + e_1, \Gamma \rangle}$ $+$-r, w
$\dfrac{e, \Gamma \to B \quad e, e^+, \Gamma \to B}{e^+, \Gamma \to B}$ $*$-l	Branch goes left	$\dfrac{u \to \langle e, \Gamma \rangle}{u \to \langle e^+, \Gamma \rangle}$ $*$-r, w
$\dfrac{e, \Gamma \to B \quad e, e^+, \Gamma \to B}{e^+, \Gamma \to B}$ $*$-l	Branch goes right	$\dfrac{u \to \langle e, e^+, \Gamma \rangle}{u \to \langle e^+, \Gamma \rangle}$ $*$-r, w
$\dfrac{\Gamma, e^\omega \to B'}{a, \Gamma, e^\omega \to B}$ match	None	$\dfrac{u \to \langle \Gamma \rangle}{a, u \to \langle a, \Gamma \rangle}$ match

We obtain a proof of $w \in \mathcal{L}(e)$, for any word w read between two unfoldings of e^ω. We can prove the exact same way that the word read before the first ω-l is in the language of the list Γ_0 without the expression e^ω, which completes the proof. □

Theorem 3.2:
The regular and cut-free fragment of S_ω is complete for ω-regular expressions.

Proof. Given a sound sequent $\Gamma_0 \to B_0$, we consider the preproof defined above, and we prove its validity.

Let us consider an infinite branch β without $*$-l thread. Let $w = word(\beta)$, by Lemma 3.8 we have $w \in \mathcal{L}(\Gamma_0)$. Since $\Gamma_0 \to B_0$ is sound, we have $w \in \mathcal{L}(B_0)$. We will use this to build an ω-r thread validating the branch β. To do so, the intuition is that at each disjunctive choice in the right-hand side, we choose according to a parsing witnessing $w \in \mathcal{L}(B_0)$. However we cannot do that in a greedy manner, as illustrated in Example 3.2.

Let us describe how we build a validating thread for our branch. We start with a list from B_0 that contains our word w, and take the last expression of this list (the one containing $\cdot^\omega$) to begin our thread. We then build it going upwards and always staying on an expression containing $\cdot^\omega$.

The only choices we have to make when building this thread upwards are when we meet the rule $\dfrac{\Gamma \to \langle e, \Sigma\rangle; \langle e, e^+, \Sigma\rangle; B}{\Gamma \to \langle e^+, \Sigma\rangle; B}$ $*$-r or the rule $\dfrac{a, \Gamma \to \langle e, \Lambda\rangle; \langle f, \Lambda\rangle; B}{a, \Gamma \to \langle e + f, \Lambda\rangle; B}$ $+$-r . In the first case ($*$-r rule), there is a smallest integer n such that e^+ can be replaced with e^n in the lower sequent while preserving the fact that the current remainder of w is in the language of the list. We will then continue while treating e^+ as e^n, and at every $*$-r rule on that thread we either go to e if $n = 1$ or e, e^{n-1} otherwise. This replacement is purely "virtual": we simply keep it in mind as a guide to pick a thread.

In the second case ($+$-r rule), there is at least one side containing our word (without the prefix we already read), so we simply choose it. Virtual replacements of some e^+ by e^n are still taken into consideration here, as can be seen in Example 3.2.

We will necessarily unfold infinitely many times the $\cdot^\omega$ expression chosen at the beginning, since we match all letters of w while keeping the invariant that it belongs to the chosen list.

In the end, we get a valid proof for any sound sequent, which proves the completeness of our system, using only regular proofs thanks to Lemma 3.7. Note that we could settle here for a weaker match rule, that would only match the first letter. □

The following example illustrates why we need some kind of look-ahead ability to build a validating thread in the completeness proof, instead of a simpler greedy algorithm.

Example 3.2:
Let us consider the preproof from Figure 3.8, which only contains a single branch. We want to produce a validating thread for that branch. If we proceed greedily at $*$-r rules and $+$-r rules, with only the heuristic "choose the first disjunct containing the left-hand side word", then we will end up with the blue thread, that never gets rid of the expression e^+ and therefore never is principal for an ω-r rule.

This is why we need the previous construction to create a validating thread, like the red one on the example, which gets rid of the e^+ expressions by being able to look ahead. We decide at the first $*$-r rule that e^+ will be read as e^2 in that branch, which means that we no longer have a choice for the red thread at the $+$-r rule above. Indeed, virtually

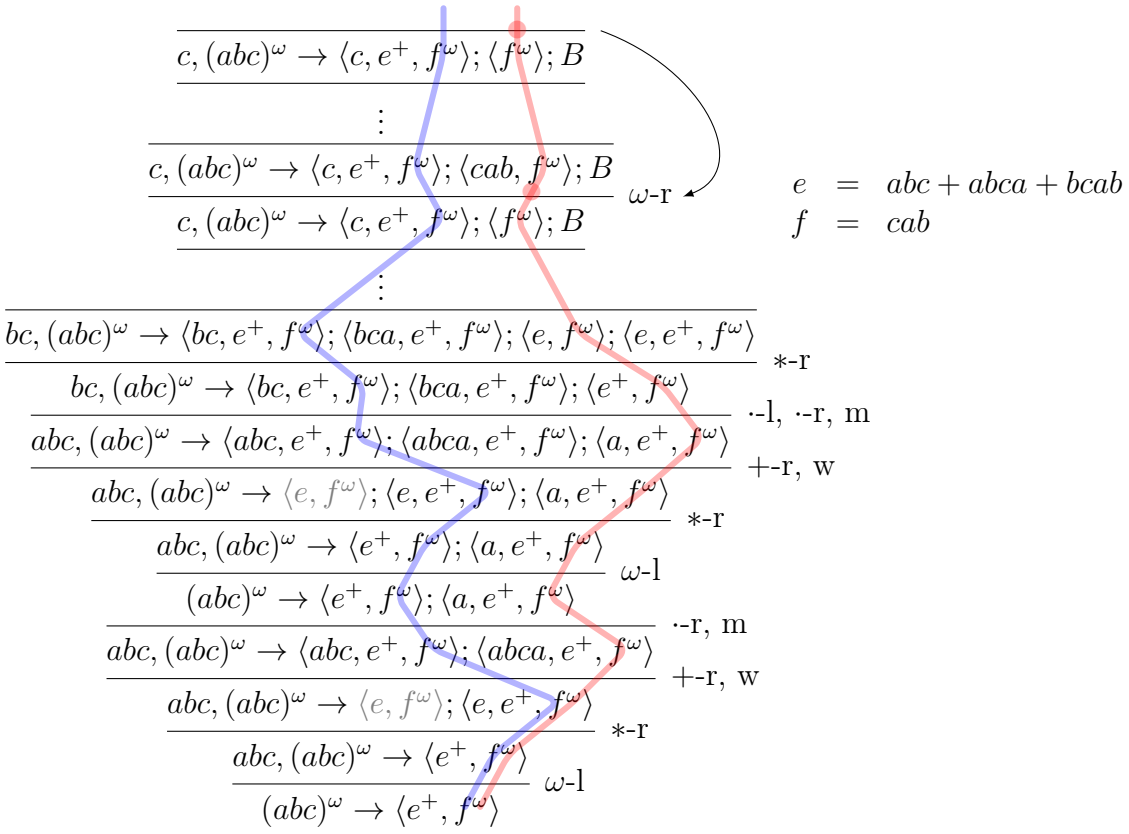


Figure 3.8: An example where a greedy process might fail to build a validating thread (Lists that do not contain the infinite word from the left are grayed)

replacing e^+ by e^2 at the beginning leads to forbidding the blue thread, since the list it chooses is replaced by $\langle abc, e^1, f^\omega \rangle$, which does not contain $(abc)^\omega$.

3.2.4 Deciding the validity criterion

Given a preproof in our system, we want to decide whether it satisfies the validity criterion. This section therefore is dedicated to proving the following theorem:

Theorem 3.3:
It is decidable in PSPACE whether a given cyclic preproof of S_ω is valid (more precisely PSPACE in the size of the largest sequent).

The arguments are similar to those in e.g. [LJB01; DHL06]. We summarize here the main ideas, we will build on them in the next section and for the transfinite case in Section 3.3.5.

We start by introducing an auxiliary notion:

Definition (Sequent transition)**:**
Given two sequents S_1, S_2, a *transition* from S_1 to S_2 is a function $\varphi : pos(S_1) \times pos(S_2) \to \{\nmid, |, \blacklozenge\}$. It encodes a way of linking S_1 to S_2 by threads: the value $\varphi(p_1, p_2)$ will be equal to:

- $\nmid$ if there is no thread from p_1 to p_2,
- $|$ if there is a thread with no v-unfolding from p_1 to p_2,
- $\blacklozenge$ if there is a thread with v-unfolding from p_1 to p_2.

We only represent here non-trivial transitions, *i.e.* we consider only threads of length at least 1. Notice that here S_1 and S_2 are sequents in the finite representation of the proof tree, so they might represent an infinite set of sequents in the unfolded proof tree. Therefore, there might be several ways of linking them by threads, yielding different transitions φ.

Composing transitions: We define an order on $\{\nmid, |, \blacklozenge\}$ by setting $\nmid < | < \blacklozenge$, and a product law $\cdot$ by setting $\nmid$ as absorbing and $|$ as neutral.

If we have transitions φ from S_1 to S_2, and φ' from S_2 to S_3, they can be composed to yield a transition $\varphi'' = \varphi \odot \varphi'$ from S_1 to S_3. This composed transition is defined by $\varphi''(p_1, p_3) = \max_{p_2 \in pos(S_2)} \varphi(p_1, p_2) \cdot \varphi(p_2, p_3)$. This gives to the set of transitions a structure of finite monoid.

Guessing a bad transition: A *self-transition* on a sequent S is a transition from S to S. A self-transition φ is called *idempotent* if $\varphi \odot \varphi = \varphi$. An idempotent transition on S is called *bad* if, for all $p \in pos(S)$, we have $\varphi(p, p) \neq \blacklozenge$.

The validity algorithm is based on the following observation:

Lemma 3.9:
A regular proof is invalid if and only if it contains a bad idempotent transition.

Proof. This is a standard application of Ramsey's Theorem, see e.g. [DHL06, Thm 4]. □

We can finally design a nondeterministic algorithm, which will guess such a bad idempotent transition. It amounts to guessing a branch and a segment along this branch witnessing the idempotent bad transition. The transition φ is computed on-the-fly on this segment. Since keeping a transition φ in memory only takes polynomial space, and NPSpace =PSpace, we end up with a PSpace algorithm.

Remark 3.2:
If the size of sequents is logarithmic in the size of the proof, this algorithm is actually in LogSpace. This is put to use in the next section.

3.2.5 PSpace inclusion algorithm via proof search

We will now combine the above algorithm with our completeness result, in order to obtain a PSpace algorithm for inclusion of ω-regular expression. This matches the known complexity of expression inclusion, which is PSpace-complete even in the case of finite words.

We are now given only the sequent we aim to prove, and we will non-deterministically explore its proof as built in Section 3.2.3. Notice that this proof can be exponential in the size of the root sequent, but this is not a problem, since the algorithm only guesses a branch and follows it on-the-fly. We only have to ensure that each sequent, and therefore each transition φ, is polynomial in the size of the root sequent. This might however not be the case, because a list $\langle e^+, \Lambda\rangle$ can be unfolded into $\langle e, \Lambda\rangle; \langle e, e^+, \Lambda\rangle$, thereby duplicating an arbitrary sequent Λ. Iterating this could lead to sets of exponential size.

This is solved by adding some syntactic sugar in our system: the sequent $\langle e^+, \Lambda\rangle$ will be unfolded into $\langle e, e^+?, \Lambda\rangle$. More precisely, we perform the following rule replacement:

$$\frac{\Gamma \to \langle e, \Lambda\rangle; \langle e, e^+, \Lambda\rangle; B}{\Gamma \to \langle e^+, \Lambda\rangle; B}\ *\text{-r} \quad \rightsquigarrow \quad \frac{\Gamma \to \langle e, e^+?, \Lambda\rangle; B}{\Gamma \to \langle e^+, \Lambda\rangle; B}\ *\text{-r}$$

The notation $e?$ means that e optional. This is expressed by adding the following pseudo-rule:

$$\frac{\Gamma \to \langle \Lambda\rangle; \langle e, \Lambda\rangle; B}{\Gamma \to \langle e?, \Lambda\rangle; B}\ ?.$$

This does not change the behaviour of the system, but guarantees that all sequents stay polynomial in the size of the root sequent. We will prove this fact using the following generalization of Fischer-Ladner closure:

Definition (?-closure)**:**
The ?-closure $\mathcal{C}_?$ of an expression is defined by induction as follows:

- $\mathcal{C}_?(a) = \{a\}$
- $\mathcal{C}_?(e + f) = \mathcal{C}_?(e) \cup \mathcal{C}_?(f) \cup \{e + f\}$
- $\mathcal{C}_?(e \cdot f) = (\mathcal{C}_?(e) \cdot \{f\}) \cup \mathcal{C}_?(f)$
- $\mathcal{C}_?(e^+) = (\mathcal{C}_?(e) \cdot \{e^+?\}) \cup \{e^+\} \cup \{e^+?\}$

- $\mathcal{C}_?(e^\omega) = (\mathcal{C}_?(e) \cdot \{e^\omega\}) \cup \{e^\omega\}$
- $\mathcal{C}_?(e?) = \mathcal{C}_?(e) \cup \{e?\}$

where $P \cdot Q$ stands for $\{u \cdot v \mid u \in P \text{ and } v \in Q\}$

With that definition $\mathcal{C}_?$ is indeed a closure: if $f \in \mathcal{C}_?(e)$, then $\mathcal{C}_?(f) \subseteq \mathcal{C}_?(e)$. Moreover, $\mathcal{C}_?(e)$ is polynomial in the size of e:

Lemma 3.10:
For any $f \in \mathcal{C}_?(e)$, $|f| \leq 2|e|^2$, where $|\cdot|$ is the number of nodes in the parse tree of the expression.

Proof. We proceed by induction on $|e|$.

- If $e = a$ is a letter, then the result holds for any f in $\mathcal{C}_?(e)$.
- If $e = e_1 + e_2$, then either $f = e_1 + e_2$, or $f \in \mathcal{C}_?(e_1) \cup \mathcal{C}_?(e_2)$. In the first case, the result holds trivially, and in the second case, it follows from the induction hypothesis for e_1 and e_2.
- If $e = e_1 \cdot e_2$, then either $f \in \mathcal{C}_?(e_1) \cdot \{e_2\}$, in which case $|f| \leq 2|e_1|^2 + |e_2| \leq 2|e|^2$, or $f \in \mathcal{C}_?(e_2)$, and the result holds too.
- If $e = e_1^+$, then if $f \in \mathcal{C}_?(e_1) \cdot e_1^+?$, we have $|f| \leq 2|e_1|^2 + |e_1| + 2 \leq 2(|e_1|+1)^2 = 2|e|^2$. The other cases are $f = e$ or $f = e?$, and the result holds.
- If $e = e_1^\omega$ and $f \in \mathcal{C}_?(e) \cdot \{e^\omega\}$, then $|f| \leq 2|e_1|^2 + |e_1| + 1 \leq 2|e|^2$ as above. The remaining case is $f = e$, and the result holds.
- If $e = e_1?$ then either $f \in \mathcal{C}_?(e_1)$ or $f = e$, and the result still holds.

This completes the proof of the Lemma. □

We can now bound the size of sequents using this result. For any list in the tree constructed by the algorithm, the concatenation of that list is in the closure of the concatenation of some list at the root. This is proven by transfinite induction, since this is preserved when going up any rule, and when jumping to a limit sequent. Moreover, with our use of the $\cdot$ rules, there is only a linear number of possible lists resulting in a given concatenation. This means that the total number of different lists in the tree is polynomial. Moreover, we know with Lemma 3.10 that each expression in $\mathcal{C}_?(e)$ is of polynomial size in e, which means that any sequent contains a polynomial number of polynomial size lists, and is therefore of polynomial size.

If the size of sequents is bounded by M, the size of any transition is in $O(M^2)$, so a bad idempotent transition, if it exists, can be computed on-the-fly using polynomial space. Since the rules used in the preproof described in Section 3.2.3 follow deterministically from the root sequent, we can indeed use nondeterminism to guess a bad branch.

Thus we obtain a nondeterministic PSPACE algorithm for inclusion of ω-regular expressions, *via* proof search in the system S_ω.

3.3 Transfinite words and proof forests

The goal is now to adapt the system in order to deal with transfinite expressions, recognizing language of transfinite words.

3.3.1 Ordinals and transfinite words

In this part we provide a brief summary of what is an ordinal, then define transfinite words using this notion, and provide an automata model to read these words.

Ordinals

We start by giving Von Neumann's definition of ordinals. Note that there are several other definitions, which are equivalent under reasonable axioms[1].

Definition (Ordinal)**:**
Formally, an *ordinal* α is a set that is well-ordered by the membership relation $\in$ (*i.e.* every non-empty subset of α has a smallest element for $\in$), and such that any element of α is also a subset of α. We will generally use lower-case Greek letters to name ordinals, and we will use $<$ instead of $\in$ when comparing ordinals, along with its reflexive closure $\leq$.

Remark 3.3:
This definition transfers to any element of an ordinal (since it is also a subset of that ordinal), meaning that the elements of an ordinal are ordinals.

We will not get too much into the detailed definitions and properties related to ordinals, but here are some basic ones that will be useful to us.

- We usually identify finite ordinals with integers: $0 = \emptyset$, $1 = \{\emptyset\}$, $2 = \{\emptyset, \{\emptyset\}\}$, *etc.*
- The successor of an ordinal α is $\alpha + 1 = \alpha \cup \{\alpha\}$. A non-zero ordinal that is not a successor is called a *limit ordinal*. The limit of an increasing sequence of ordinals $(\alpha_i)_{i\in\beta}$ indexed by a limit ordinal β is the smallest ordinal greater than all α_i.
- The first limit ordinal (corresponding to the set of integers) is called ω.
- Inductive reasoning applies to ordinals:

$$\big(\forall \alpha < \gamma, (\forall \beta < \alpha, P(\beta)) \Rightarrow P(\alpha)\big) \Rightarrow \forall \alpha < \gamma, P(\alpha)$$

- The *sum* of two ordinals is defined inductively by:

$$\begin{array}{rcl} \alpha + 0 & = & \alpha \\ \alpha + (\beta + 1) & = & (\alpha + \beta) + 1 \\ \alpha + \lim_{i\in\gamma} \beta_i & = & \lim_{i\in\gamma} (\alpha + \beta_i) \end{array}$$

  and it is associative, but not commutative: $\omega + 1 = \omega \cup \{\omega\}$ while $1 + \omega = \omega$.

[1] The axiom of regularity is enough to prove the equivalence with the definition with transitive set for instance.

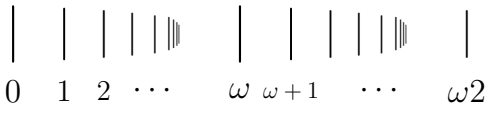


Figure 3.9: Visualization of the ordinal $\omega + \omega + 1 = \omega 2 + 1$. Each line represents one element, and they are ordered from left to right.

- The product and exponentiation can be defined similarly, but we do not detail those here.
- If β is a limit ordinal and $(x_i)_{i<\beta}$ is a sequence of length β, a subsequence $(x_{i_j})_{j<\omega}$ of length ω is said *cofinal* if, for all $i < \beta$, there exists $j \in \omega$ such that $i < i_j$.
- Given two ordinals $\alpha \leq \beta$, there is a unique γ such that $\alpha + \gamma = \beta$.

See Figure 3.9 for a visual representation of an ordinal.

Note that ω-words are words indexed by the ordinal ω. The next part generalizes this definition to any ordinal.

Transfinite words and expressions

Once ordinals are defined, we can introduce words indexed by these ordered sets, and some operations on these new words, which are slightly more intricate than the previous ones.

Definition (Transfinite word)**:**
A *transfinite word* over an alphabet Σ is given by a function $\alpha \to \Sigma$, where α is an ordinal called the length of the word.

Remark 3.4:
The words studied before are particular cases of this definition: a transfinite word of length α is a finite word if $\alpha < \omega$, and an ω-word if $\alpha = \omega$.

In this work, we will restrict the length α to be strictly smaller than ω^ω, *i.e.* α will be smaller than ω^k for some $k \in \mathbb{N}$. These ordinals describe the length of words obtained with expressions that are allowed to nest the ω-power finitely many times. These expressions will be formally defined later.

Definition (Concatenation of transfinite words)**:**
We define the *concatenation* of two words $u : \alpha \to \Sigma$ and $v : \beta \to \Sigma$, noted $u \cdot v : \alpha + \beta \to \Sigma$, as follows.

$$(u \cdot v)(\gamma) = \begin{cases} u(\gamma) & \text{if } \gamma < \alpha \\ v(\delta) & \text{if } \gamma \geq \alpha, \text{ where } \delta \text{ is the only ordinal such that } \alpha + \delta = \gamma \end{cases}$$

Prop 3.1:
The concatenation is associative.

Definition (Infinite concatenation)**:**
Now suppose we are given a sequence of words $U = (u_i : \alpha_i \to \Sigma)_{i\in\omega}$. Let us call β the limit of $\sum_{i=0}^{n} \alpha_i$ for $n \in \omega$. Then the concatenation of the words in U is a word $\prod U$ of length β defined as follows:

$$\prod U(\gamma) = (u_1 \cdot \ldots \cdot u_n)(\gamma) \text{ where } n \text{ is the smallest integer such that } \sum_{i=0}^{n} \alpha_i > \gamma$$

Notice that we could more generally define the concatenation of a sequence of words indexed by an ordinal greater than ω, but we will not need it here.

We can now define the expressions we will be using, called transfinite expressions. They are similar to ω-regular expressions, except that the ω operator can now be used freely.

Definition (Transfinite expressions)**:**
The *transfinite expressions* we consider are the ones generated by the following grammar [Cho78]:

$$e, f := a \mid e + f \mid e \cdot f \mid e^+ \mid e^\omega$$

Notice that this syntax generalizes ω-regular expressions, by allowing any nesting of $\cdot^\omega$ and $\cdot^+$. We use the non-empty version $\cdot^+$ of Kleene star to avoid having to deal with ε^ω as a special case. The language associated to an expression is defined by induction as follows.

- $\mathcal{L}(a) := \{a\}$.
- $\mathcal{L}(e + f) := \mathcal{L}(e) \cup \mathcal{L}(f)$.
- $\mathcal{L}(e \cdot f) := \mathcal{L}(e) \cdot \mathcal{L}(f) = \{u \cdot v \mid u \in \mathcal{L}(e) \text{ and } v \in \mathcal{L}(f)$.
- $\mathcal{L}(e^+) := \bigcup_{i=1}^{\infty} \mathcal{L}(e)^i$ where $\mathcal{L}(e)^i = \mathcal{L}(e) \cdot \ldots \cdot \mathcal{L}(e)$ (concatenated i times).
- $\mathcal{L}(e^\omega) := \{\prod U \mid U \in \mathcal{L}(e)^\omega\}$.

Example 3.3:
The expression $a^\omega b^\omega a$ corresponds to a singleton language, as it does not use the $+$ or $\cdot^+$ operators. This word can be created by taking the ordinal from Figure 3.9, and labelling each element of the first block (*i.e.* the first ω elements) with a, then the second block with b, and the single remaining element with a again.

An associated automata model

In order to manipulate these expressions, we shall use automata inspired from [Cho78]. Before that, we need to define the following objects.

Definition (A few tools)**:**

- Given a set Q of atomic states, $\mathcal{P}_n(Q)$ is defined by induction with $\mathcal{P}_0(Q) = Q$ and $\mathcal{P}_{k+1}(Q) = (\mathcal{P}(\mathcal{P}_k(Q)) \setminus \emptyset) \cup Q$. Note that if Q is finite, then so is $\mathcal{P}_n(Q)$.

- We say that a letter appears cofinally in a word of limit length if the letter can be found after any position in the word.

- Given a limit ordinal β and a word w of length at least β, we define $\mathcal{I}(w, \beta)$ as the set of letters seen cofinally in the prefix of w of length β (obtained by restricting the domain of w to β).

Definition:
For $n \in \mathbb{N}$, a (non-deterministic) *n-automaton* is given by $(Q, q_0, \Sigma, \Delta, F)$, where Q is a finite set of atomic states, Σ is the alphabet, $\Delta \subseteq \mathcal{P}_n(Q) \times \Sigma \times Q$ is the set of transitions, $F \subseteq \mathcal{P}_n(Q)$ is the set of final states.

Such an automaton will only be able to run on words of length smaller than ω^{n+1}.

A run of an n-automaton $\mathcal{A}$ on a word w of length α is given by a word ρ of length $\alpha + 1$ over the alphabet $\mathcal{P}_n(Q)$ such that:

- $\rho(0) = q_0$;

- $(\rho(\beta), w(\beta), \rho(\beta + 1)) \in \Delta$ for any $\beta < \alpha$;

- $\rho(\beta) = \mathcal{I}(\rho, \beta)$ for any limit ordinal $\beta \leq \alpha$.

The run r is accepting if $r(\alpha) \in F$, and a word is accepted by the automaton if there is an accepting run on this word, as is usual in non-deterministic models.

These automata look similar to those in [Cho78], but are slightly different in the fact that they give us more control over the limit transitions. Since a limit transition is given by all the states seen cofinally, it becomes easier to ensure that we remain in a given part of the automaton.

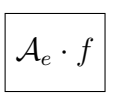
The equivalence between these automata and the expressions above was known in the formalism of [Cho78], we need a few adaptations here.

Lemma 3.11:
For any given expression, there is an associated automaton with the same accepted language. Moreover, this automaton is defined by induction on the expression, and therefore any sub-expression or unfolding of an expression is associated to a sub-automaton (with chosen initial and accepting states).

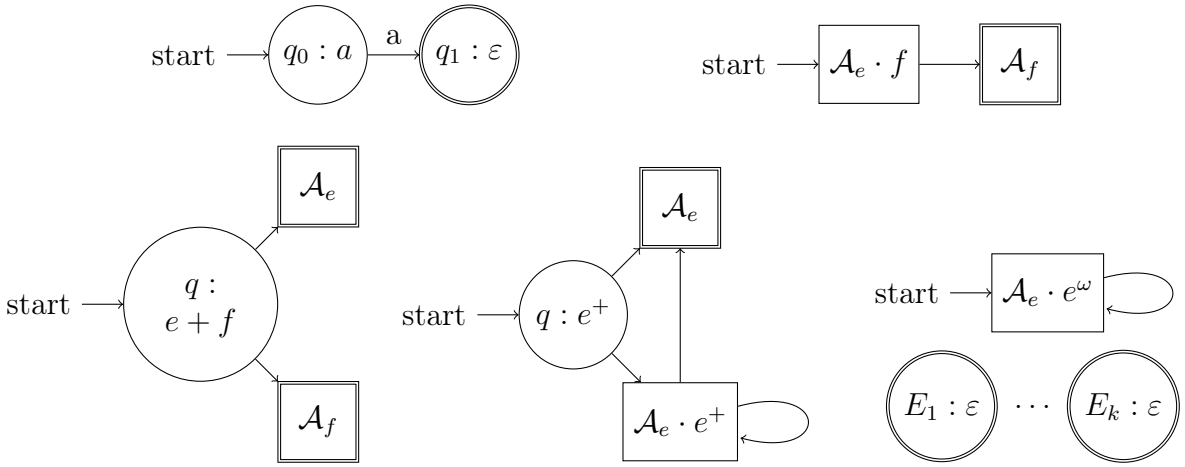
Proof. Let us first give a few notations for the construction of these automata.

- We label a state q with an expression e for the language it recognises, with this notation: 

.

- We use 

to denote a block containing the states and transitions from the automaton $\mathcal{A}_e$ recognizing $\mathcal{L}(e)$, with a renaming of the states if needed, and with $\cdot f$ added to each label (we extend this notation lo lists).



If $\mathcal{A}_e = (Q, q_0, \Sigma, \Delta, F)$, then $\{E_i\}_{1 \leq i \leq k}$ is the set of all subsets of $\mathcal{P}_n(Q)$ that contain an element of F.

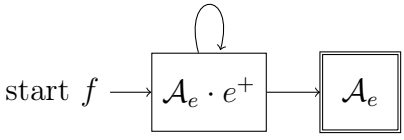
Figure 3.10: Inductive construction of the automaton associated to an expression

> Such a block can either have a double outline, in which case the accepting states of the global automaton are the ones of $\mathcal{A}_e$, or an outgoing edge to a block $\mathcal{A}_f$, in which case each final states takes the same transitions as the initial state of $\mathcal{A}_f$.
>
> We can also add a start transition to such a block to indicate that we take its initial state as the one of the global automaton.

As said in the lemma, we proceed by induction, each case being described in Figure 3.10. We can easily check that each step preserves the fact that the automaton recognizes the same language. The only difficulty might be in the understanding of the e^ω step. The only way to reach an accepting state for this automaton is to see infinitely many accepting states of $\mathcal{A}_e$ without going out of $\mathcal{A}_e$ (at least after some point). Since these states have no outgoing transitions except for those we added to go back to the beginning of $\mathcal{A}_e$ (as can be seen in the constructions of Figure 3.10), this is equivalent to reading e^ω.

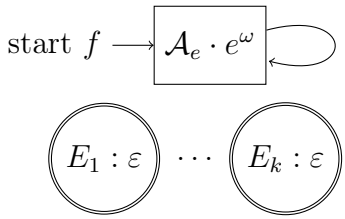
We however want to add a slight complication here. The only point of that part is to make sure that the automaton resulting from an unfolding of e^+ or e^ω is actually a sub-automaton of the one for the initial expression (e^+ or e^ω). For instance, the automaton for $b(ab)^+$ should not be the concatenation of those for b and $(ab)^+$, but instead a sub-automaton of $(ab)^+$.

Let us look at the case of the concatenation of two expressions f and e^+ (which can be within some context) where the automaton for f can be embedded in the one for e by simply changing the initial state (and possibly removing unattainable states). Then instead of creating the automaton for concatenation from Figure 3.10, we can take the following one:



where "start f" designates the initial state for f instead of e (but the loop still follow the same conventions and ignores that state). With that, we ensure that the automaton associated to an unfolding of e^+ is actually embedded into the automaton for e^+.

We proceed similarly when e^+ is replaced by e^ω, creating the following automaton.

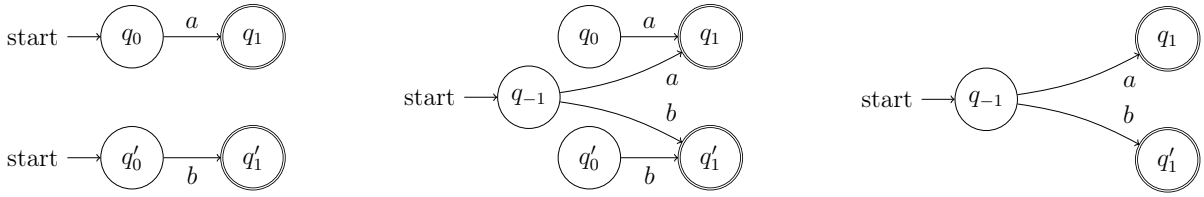


This completes the proof, as it gives the last case for the unfolding of an expression. □

Note that we can generalize this construction to lists of expressions by using the concatenation step several times. This will give us an automaton associated to the list.

Another remark we can make at that point is that, in the inductive construction, some states may become unattainable. For instance when building the automaton for $a+b$, we first build the ones for a and b, then we add an initial state that bypasses the ones of the previous automata, which therefore become useless, as illustrated in Example 3.4 below.

Example 3.4:
Here are the automata for a and b (on the left), and the one for $a+b$ (middle). In that case, q_0 and q_0' become unreachable, and we will not draw such states from now on, as shown on the right.



Notice that it is also shown in [Cho78] that expressions can be computed from automata, so the two models are expressively equivalent. This could be adapted to our variant as well, but as we will not use this direction here, we omit it.

3.3.2 Adapting the proof system

The new proof system: To build a proof system dealing with transfinite expressions, we will basically keep the same rules as in S_ω, except that ω operators are not required to appear at the end of lists anymore. This gives rise to the following relaxed rules for ω:

$$\frac{e, e^\omega, \Gamma \to B}{e^\omega, \Gamma \to B}\ \omega\text{-l} \qquad \frac{\Gamma \to \langle f, f^\omega, \Lambda\rangle; B}{\Gamma \to \langle f^\omega, \Lambda\rangle; B}\ \omega\text{-r}$$

Another difference will be that a preproof will no longer be a tree, but a *forest*, *i.e.* a set of trees with distinct roots. This will allow us to consider branches of ordinal length: after taking ω steps in a tree, a branch can "jump" to the root of another tree via a *limit condition*, analogous to the validity condition of the previous section.

Definition (Branches, threads and limit sequents)**:**
We define inductively these notions as follows. These definitions are mutually recursive, but well-founded: the notions are defined together for a fixed ordinal length, before going to the next one or the limit.

- A *transfinite (resp. limit) branch* is a transfinite sequence of sequent positions in the forest (resp. of limit length), starting at the main root sequent of the proof. The successor of a sequent must be just above it in the forest, and any non successor sequent must be the limit sequent of the limit branch before, as defined below.

- A *transfinite (resp. limit) thread* is a transfinite sequence of expression occurrences following a transfinite (resp. limit) branch, while respecting immediate ancestry for successor sequents, and going to the corresponding expression of the limit sequent when jumping to the limit sequent, as defined below.

  A limit thread with a cofinal sequence of expressions that are principal for a rule r is called a r thread.

- The *limit sequent* of a limit branch, when it exists, is a root sequent from some tree in the proof forest, possibly the tree containing this limit branch. We define it by considering the ω-l and ω-r limit threads following the branch cofinally. On the left side, there must be an ω-l thread, that is principal infinitely often on the same sequent of the form e^ω, Γ, such that no rule is applied on Γ after some point. The corresponding limit sequent will have Γ as left-hand side.

  We proceed similarly to get the lists on the right side of the limit sequent. Given an ω-r limit thread principal infinitely often on some list $\langle e^\omega, \Gamma\rangle$, with Γ untouched after some point, we will have a list $\langle\Gamma\rangle$ on the right-hand side of the limit sequent. Any list on the right that cannot meet these conditions is discarded in the limit sequent. In both cases (left and right of the sequent), we call e^ω the *frontier expression* of that list.

  The threads are prolonged to that limit sequent the natural way, by taking the limit of an inactive thread on the right of a frontier expression as the corresponding expression in the limit sequent.

These definitions are illustrated in Section 3.3.6, with an example of a transfinite proof, and a visualization of a limit branch of length ω^2 is given in Figure 3.11. Such a branch goes through ω trees, not necessarily distinct.

Definition (Validity condition, cyclic proof forest)**:**
A proof forest is *valid* if any limit branch either contains a cofinal $*$-l thread, or has its limit sequent appearing as the root of a tree in the proof forest.

A proof forest is called *cyclic* (or *regular*) if it is the unfolding of a finite graph (not necessarily connected), or equivalently if it contains finitely many non-isomorphic subtrees.

Let us call $\mathcal{S}_t$ this proof system for inclusion of transfinite expressions.

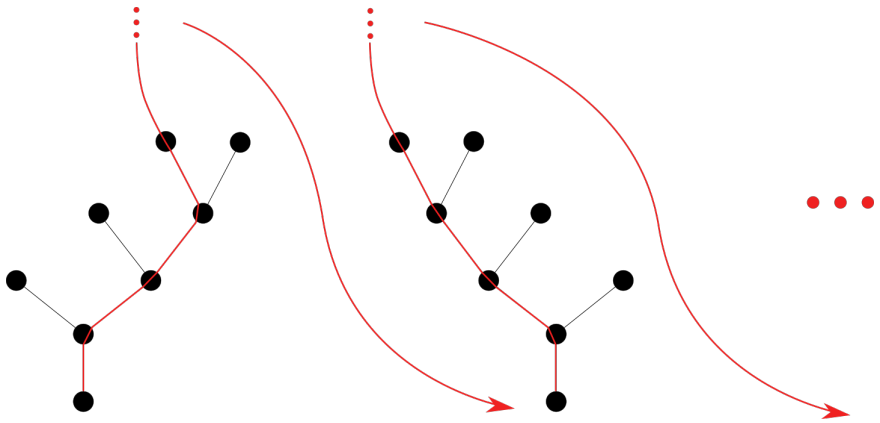


Figure 3.11: Example of a limit branch of length ω^2

3.3.3 Soundness

The soundness of $\mathcal{S}_t$ is shown in a similar way to the one of S_ω. We first note that as the rules are locally sound, Lemma 3.1 still holds for $\mathcal{S}_t$, meaning that any finite proof is sound.

Given a valid proof P in $\mathcal{S}_t$, with root sequent $\Gamma_0 \to B_0$, and a word $w \in \mathcal{L}(\Gamma_0)$, we can still build a labelled proof $P(w)$ to witness $w \in \mathcal{L}(B_0)$. This is done in the same way as before, but we also add the limit sequents for every new limit branch that appears, while preserving the labelling in the natural way.

Note that this process can lead to a bigger forest, since a single root sequent in the initial proof P can lead to several ones with different labellings in this proof. We can still build the forest using transfinite recursion, knowing that there are less than ω^ω trees.

We will reuse the concept of correct labelling and label-soundness of a sequent $\Gamma' \to B$, meaning that the word concat(Γ') is in $\mathcal{L}(\Gamma')$ (and is correctly split if Γ' is a list) and $\mathcal{L}(B)$ respectively.

Lemma 3.12:
In $P(w)$, any limit branch can be extended (*i.e.* has a limit sequent).

Proof. This is simply a consequence of the fact that we build $P(w)$ according to a parsing of w in Γ_0. Therefore, we cannot unfold infinitely many times a same expression e^+. Since the validity condition asks for either a $*$-l thread or a limit sequent, we must be in the second case on all limit branches of $P(w)$. □

Lemma 3.13:
In $P(w)$, there is no transfinite branch that goes infinitely many times on the left at a cut rule.

Proof. Suppose that there is a transfinite branch that does go infinitely many times on the left. Let us take the smallest prefix of that branch that still respects that condition (possible by well-foundedness). This is a limit branch (otherwise it can be made even smaller), which goes to the left premise of a cut rule cofinally, which cuts any ω-r thread. Since Lemma 3.12 ensures a limit sequent for that branch, there has to be an ω-r thread (the right side of a sequent is nonempty), hence the contradiction. □

As before, the maximal transfinite branches in $P(w)$ can be ordered from left-to-right, by comparing them at the first cut rule where they take a different direction (which exists by well-order property). We denote that order by $<$.

Lemma 3.14:
The order $<$ over the maximal transfinite branches of $P(w)$ is well-founded.

Proof. The proof is similar to the one of Lemma 3.4, the only notable change being that the word over $\{0, 1\}$ associated to a branch is now transfinite. □

Lemma 3.15:
In $P(w)$, all lists on the left side are correctly labelled, and all sequents are label-sound.

Proof. As in Lemma 3.5, we prove this by a transfinite induction on the left-to-right order $<$ on branches, which is well-founded by Lemma 3.14.

Let us call C the set of maximal transfinite branches from $P(w)$. Assume that, for some branch $\beta \in C$, every $\beta' < \beta$ verifies the property.

The first thing we want to prove is the correct labelling in β. This part is done as for S_ω, by induction on the branch. We need to add the limit case, since the branch is transfinite. Since limit sequents are untouched in the limiting process, the limit case of the induction is straightforward *i.e.* limit sequents are correctly labelled.

We now want to prove the second part of the induction. Let us call v the vertex just above the last left cut of β. We want to prove the label-soundness of the proof rooted in v. We call β_v the part of β above v. We know that in β_v, a sequent is label-sound if its successor in the branch is. We want to prove by transfinite induction on the length of β_v that the sequent at v is label-sound if the last sequent of the branch is (recall that in $\mathcal{S}_t$, all branches have a last sequent).

What we need for that is to prove that if the limit sequent of a limit branch is label-sound, then so is (at least) one sequent in that branch.

Let us consider such a limit branch, with limit sequent $\Gamma' \to B$. The word $\text{concat}(\Gamma')$ is in the language of some list $\Lambda \in B$. We can now use the exact same process as for S_ω (see Figure 3.5) to prove that there is a label-sound sequent $\Pi', \Gamma' \to \langle f^\omega, \Delta \rangle; D$ in the branch before. The only difference is that transfinite branches can be hidden between two ω-r rules, but they are dealt with using the induction hypothesis for shorter branches.

This completes the inductive proof for the label-soundness of β_v. Using the global induction on the well-order $<$ on branches, we get the final result. □

Theorem 3.4:
Any valid proof in $\mathcal{S}_t$ is sound.

Proof. By Lemma 3.15, the root sequent of $P(w)$ is label-sound, and this is true for any $w \in \mathcal{L}(\Gamma_0)$. This means that for any $w \in \mathcal{L}(\Gamma_0)$, we have $w \in \mathcal{L}(B_0)$, thus any valid proof has a sound root sequent. □

3.3.4 Cut-free regular completeness of $\mathcal{S}_t$

We start with the following observation, a straightforward generalization of Lemma 3.7:

Lemma 3.16:
In a proof forest without cut and without useless trees (that can be removed while preserving the validity), there can only be finitely many different sequents.

Theorem 3.5 (Completeness)**:**
Given two expressions e and f such that $\mathcal{L}(e) \subseteq \mathcal{L}(f)$, there exists a cut-free cyclic proof forest for $e \to \langle f \rangle$. Moreover, the construction is effective.

Proof sketch

We first provide a sketch of the proof, and the complete proof can be found below.

As before, we build the proof trees using a straightforward deterministic bottom-up process, which can be done algorithmically. This time however, in order to show that the obtained proof satisfies the validity condition, we use a model of transfinite automata that helps us to exhibit a validating thread or a limit sequent for each limit branch.

The idea of the proof is to follow the runs of automata $\mathcal{A}_e$ and $\mathcal{A}_f$ canonically associated to e and f, and to build a proof whose nodes are labelled by states of these automata. A state will be associated to each list of expressions, so for each sequent, we will have one state on the left side and possibly several on the right side. Notice that this intuitively corresponds to building a run in a product automaton $\mathcal{A}_e \times \mathcal{P}(\mathcal{A}_f)$, where $\mathcal{P}(\mathcal{A}_f)$ is a powerset automaton obtained from $\mathcal{A}_f$. Since the structure of automata closely follow the structure of expressions, we can always keep the wanted invariants. Limit nodes are built by looking at all the infinite threads in limit branches, and are labelled by the set of states seen cofinally in the corresponding runs. We thereby ensure that the resulting proof is valid.

Complete proof of Theorem 3.5

We will use the following notion in this proof:

Definition (Covering a state)**:**
Given two states p and q in an automaton, we say that p *covers* q if for any transition from $q \xrightarrow{a} r$, there is a transition $p \xrightarrow{a} r$.

We can note that the covering relation is transitive, and also that a language inclusion follows from it: if p covers q, then the language recognized from q is included in the one recognized from p.

We want to show that for any expressions e and f with $\mathcal{L}(e) \subseteq \mathcal{L}(f)$, there exists a proof forest for $e \to \langle f \rangle$.

To reach this result, we will first transform our lists of expressions into automata, then build proof forests from these automata. We will use the limit transitions of n-automata to ensure the validity condition.

Suppose we are given automata $\mathcal{A}_e$ and $\mathcal{A}_f$ associated to e and f respectively. We are going to create a proof tree in which each sequent comes with a labelling of its lists by

states of these automata.

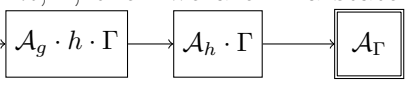
More precisely, when proceeding further in the proof, we associate to each list a state in the automaton $\mathcal{A}_e$ if we are on the left, and $\mathcal{A}_f$ on the right. This state must cover the initial state of the automaton canonically associated to the list.

We will now build the proof inductively, starting with the sequent $e \to \langle f \rangle$ in which the lists (of one element) e and f are associated to the initial state of their respective automaton. We first get rid of the two following cases on the sequent considered at any point.

- If the sequent is $\to \langle\rangle; B$ (for some set of lists B), then we can use the w rule followed by the id rule to close this branch, and the states associated to each list remain the same (it has to be an accepting state by induction, since it must recognize the empty word).

- If the sequent contains only lists starting with a single letter or empty (both on the left and right sides), we can remove any list that does not start with the same letter as the one on the left with the wkn rule, then use the match rule to get rid of this letter. Since each state associated to a list covers the initial state of an automaton for this list, we can take the transition labelled by the remaining letter from this initial state. For instance, if we look at the first automaton from Example 3.5, we jump from q_0 (with list a, a^ω, a) to q_1 (with a^ω, a) using the knowledge that we are in the language of q_{init}.

With these two cases out of the way, and as long as we preserve the soundness of each sequent by induction, we know that there are lists that have a first expression that is not a single letter. We proceed inductively in the first of these lists.

- If the list starts with a concatenation: $g \cdot h, \Gamma$, then we are in a state that covers the initial state of the automaton 

which also is the one associated to g, h, Γ so we can stay in the same state and transform the list into g, h, Γ using the $\cdot$-l or $\cdot$-r rule.

- If the list starts with a union: $g + h, \Gamma$, then we look at the corresponding automaton from Figure 3.10, and we build the two corresponding lists using $+$-l or $+$-r (in the first case, the lists are in different sequents while they stay in the same one in the second case). Although we leave the state unchanged, we can notice that it covers the initial states of both automata associated to g, Γ and h, Γ, provided that it did cover the initial state of the automaton associated to $g + h, \Gamma$.

- The case g^+, Γ is quite similar. We do the same thing as the previous case by reading g^+ as $g + g \cdot g^+$, using $*$-l or $*$-r rule. If the current state covers the initial state of the automaton associated to g^+, Γ, then it also covers the ones of the automata for g, g^+, Γ and g, Γ.

- This leaves us with the last case: g^ω, Γ, where the current state covers the initial state of a sub-automaton for g^ω, Γ. Here we simply unfold the expression using ω-l or ω-r, and remain in the same state which also works for g, g^ω, Γ.

This process allows us to create a single tree. We then repeat the following as long as it spans more trees.

For each limit branch, we want to make sure that the corresponding limit sequent does exist, so we create it if it was not already the root of a tree in the forest. The lists of such a limit sequent are determined by the definition of a transfinite branch, but we also need to label those with states. This is done by taking the set of states seen cofinally along the ω-l or ω-r thread generating each list.

Once we have created all possible new root sequents, we can generate the corresponding trees the same way as before, then repeat the process with any new limit branch. We only create finitely many trees because of Lemma 3.16 together with the fact that there are finitely many states.

This construction is illustrated in Example 3.5.

We now have a proof forest using the rules of our system (without cut), and we know that the state associated to a list covers one recognizing the language of this list. Notice that this proof forest can have several trees with same root. However, since the way we build trees depends only on expressions and not on their state labelling, these trees having the same root sequent will be isomorphic up to their labelling by states. We can therefore consider that they correspond to a unique tree in the actual proof.

We now want to verify the validity condition in this proof forest, in order to finish the completeness proof.

Let us consider a limit branch. We prove by induction on its length that it satisfies the validity condition, and that its limit sequent is sound if there is one.

The branch gives a transfinite run in $\mathcal{A}_e$, and there has to be an outermost loop (considering the inductive construction of the automaton) that is used cofinally.

After some point, the run never goes out of this loop (because coming back would mean using a more external loop). This loops corresponds to an unfolding in the left list, of either a $\cdot^+$ expression or a $\cdot^\omega$ one.

In the first case, we get a validating $*$-l thread by simply following this expression in the proof forest.

In the second case, we are unfolding an expression e^ω, so we see some sequent $e^\omega, \Gamma \to B$ at some point in the branch. The word $u \in \mathcal{L}(e^\omega)$ unfolded by the branch from this sequent on (we concatenate the letters from the m rules to form u) is of limit length (non successor ordinal).

Since each node of the proof describes a sound inclusion by induction, we know that $\mathcal{L}(e^\omega, \Gamma) \subseteq \mathcal{L}(B)$. If we take a word $v \in \mathcal{L}(\Gamma)$, we have $u \cdot v \in \mathcal{L}(B)$. We therefore know that $u \cdot v \in \Lambda$ for some list $\Lambda \in B$. But we also know that u is of limit length, and the only way to match such a word in Λ is using a $\cdot^\omega$ expression, so there has to be one last such expression cofinally unfolded during the matching of u (last because no other expression operator can create a word of limit length). We follow this expression to get an ω-r thread for our branch, which completes the proof of the validity condition. Moreover, the remaining list it kept in the limit sequent and contains v, which proves the soundness of that limit sequent (since this is true for any $v \in \mathcal{L}(\Gamma)$). This completes the proof of Theorem 3.5, except for the effectiveness part, which is taken care of by the following lemma.

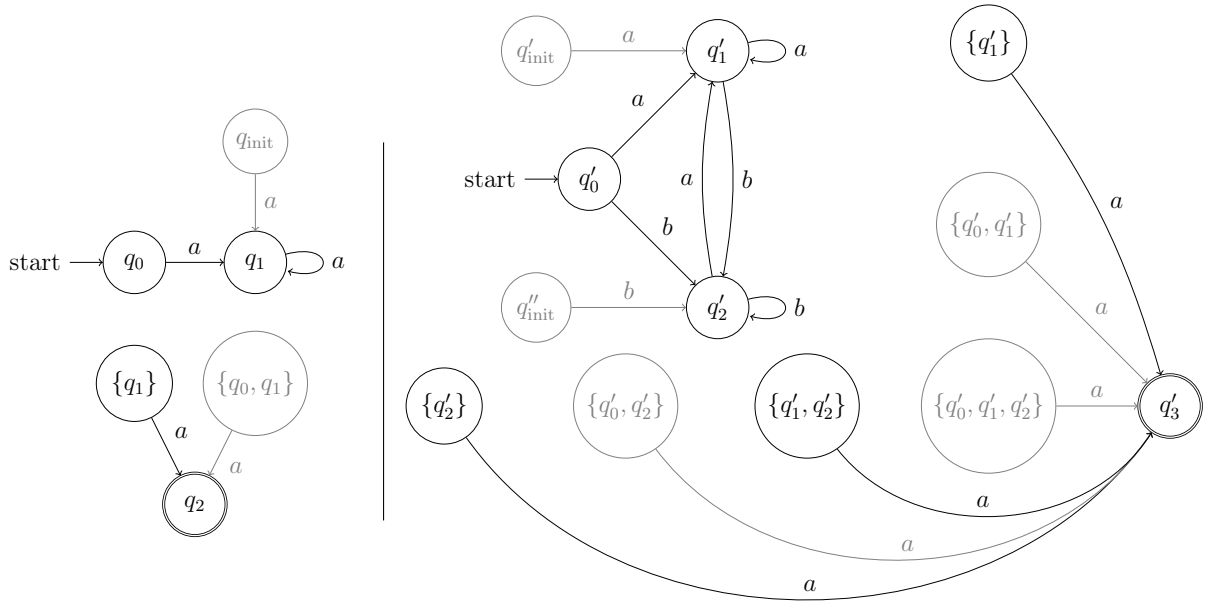


Figure 3.12: Automata for $a^\omega \cdot a \to \langle (a+b)^\omega \cdot a \rangle$

Lemma 3.17:
The proof forest can be effectively built from the expressions e and f.

Proof. First note that the construction of each tree (given its root sequent) can be done in a finite number of steps by following the proof of Theorem 3.5, since we can only create finitely many sequents (and we simply have to loop back to a previous copy when we see a sequent we have seen before). The difficulty resides in the computation of the limit sequents.

This is very similar to the proof of Theorem 3.3. We do the same construction to get limit sequents, then we add those sequents as said in the proof of Theorem 3.5. We then build the corresponding trees and go on with the new transitions (without forgetting the transitions created by the new tree). The iteration on this process will stop at some point, when there is no new root sequent added. □

Example 3.5:
To prove $a^\omega \cdot a \to \langle (a+b)^\omega \cdot a \rangle$, we create the two automata from Figure 3.12. Note that some states from the construction are inaccessible, represented in gray.

The proof forest we will obtain from this is given in Figure 3.13. We can see that the states are only updated when going through a match rule.

3.3.5 Decidability and Complexity

We generalize here the decidability and complexity results obtained in Sections 3.2.4 and 3.2.5:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{q_1 : a^\omega, a \to \langle q'_1 : (a+b)^\omega, a\rangle}\,\omega\text{-r}}{q_0 : a, a^\omega, a \to \langle q'_0 : a, (a+b)^\omega, a\rangle}\,\text{match}}{q_0 : a, a^\omega, a \to \langle q'_0 : a, (a+b)^\omega, a\rangle; \langle q'_0 : b, (a+b)^\omega, a\rangle}\,\text{wkn}}{q_0 : a, a^\omega, a \to \langle q'_0 : a+b, (a+b)^\omega, a\rangle}\,+\text{-r}}{q_0 : a, a^\omega, a \to \langle q'_0 : (a+b)^\omega, a\rangle}\,\omega\text{-r}}{q_0 : a^\omega, a \to \langle q'_0 : (a+b)^\omega, a\rangle}\,\omega\text{-l}}{q_0 : a^\omega, a \to \langle q'_0 : (a+b)^\omega \cdot a\rangle}\,\cdot\text{-r}}{q_0 : a^\omega \cdot a \to \langle q'_0 : (a+b)^\omega \cdot a\rangle}\,\cdot\text{-l}$$

$$\dfrac{\dfrac{}{q_2 : \to \langle q'_3 : \rangle}\,\text{id}}{\{q_1\} : a \to \langle \{q'_1\} : a\rangle}\,\text{match}$$

Figure 3.13: Proof created for $a^\omega \cdot a \to \langle (a+b)^\omega \cdot a\rangle$

Theorem 3.6:

- Given a cyclic preproof in $\mathcal{S}_t$, there is a PSPACE algorithm deciding whether it is valid (more precisely PSPACE in the size of the largest sequent).
- Given a sequent $\Gamma \to B$, there is a PSPACE algorithm deciding whether there is a valid proof of $\mathcal{S}_t$ with root $\Gamma \to B$.

As before, the second item is deduced from the first, together with Theorem 3.5.

Given a regular preproof that can be explored (or built) on-the-fly, we will again use the formalism of *transitions*: if S_1 and S_2 are sequents in the finite representation of a proof, we will use a function $\varphi : pos(S_1) \times pos(S_2) \to \{\dagger, |, \blacklozenge\}$ to sum up the information about threads from S_1 to S_2 in a particular path of the unfolded proof. We also mark the unfoldings of ω on the left, since we need those to compute limit sequents.

Limit processes

We have to account for the fact that a transition may now represent a path containing (nested) passages to the limit. We verify that such passages to the limit can be effectively computed, and incorporated in our saturation procedure. Remark that the information stored in a transition is enough to identify a frontier expression in an idempotent sequent. By another application of Ramsey's theorem, this will allow us to compute limit sequents, and build transitions corresponding to branches of any length (by keeping only threads to the right of frontier expressions). Now, according to the transfinite validity criterion, an idempotent transition is bad if it does not have a $*$-l thread or a limit sequent. As before, our goal is to guess a bad idempotent transition corresponding to a transfinite branch, if any exists. Notice that guessing such a transition involves guessing a starting point, and that starting points at different levels of ω nesting may differ. This means that our nondeterministic algorithm has to store a current prefix of guessed transition for each level, in order to build the final bad idempotent transition. An example of a run of this algorithm can be found in Section 3.3.6.

Compact notation

When building the proof on-the-fly according to the construction of Section 3.3.4, we also need to ensure that transitions stay of polynomial size. To this end, as in Section 3.2.5, we will use the compact notation $e?$ to avoid an exponential blow-up of sequent

size. Note that this simplified representation allows passage to the limit sequent, in the sense that the computation of the limit sequent of a branch using compact notation will yield a compact notation of the correct sequent. As before, this compact notation allows us to obtain a bound on the size of sequents which is polynomial with respect to the size of the root sequent, see 3.2.5 for details.

Thus, we obtain the following corollary, which is a new result to the best of our knowledge.

Corollary 3.1:
Deciding the inclusion of transfinite expressions is in PSPACE.

3.3.6 Example of a transfinite proof

We give in Figure 3.14 an example of a non-trivial transfinite proof of the following sequent

$$(ba^\omega + b^+)^\omega \to \langle((a+b)^\omega)^\omega + ((a+b)^\omega)^+ b^\omega + b^\omega\rangle.$$

We use the notations $e = ba^\omega + b^+$ and $f = (a+b)^\omega$.

In order to avoid adding another tree, we directly start with the sequent

$$e^\omega \to \langle f, f^\omega\rangle; \langle f, b^\omega\rangle; \langle f, f^+, b^\omega\rangle; \langle b^\omega\rangle.$$

Only the main threads are represented: some detours are allowed to them, but not represented to keep the proof readable. For instance, the green threads can follow the red ones for a finite number of times before returning to their branch.

Let us now consider what the validity checking algorithm would look like when applied to this proof. This nondeterministic algorithm explores branches to check their validity. Let us look at its run on the leftmost maximal branch of the proof.

We begin at the root sequent, which we chose as a starting point for the transition T_2 of length ω^2:

$$e^\omega \to \langle f^\omega\rangle; \langle f^+, b^\omega\rangle; \langle b^\omega\rangle$$

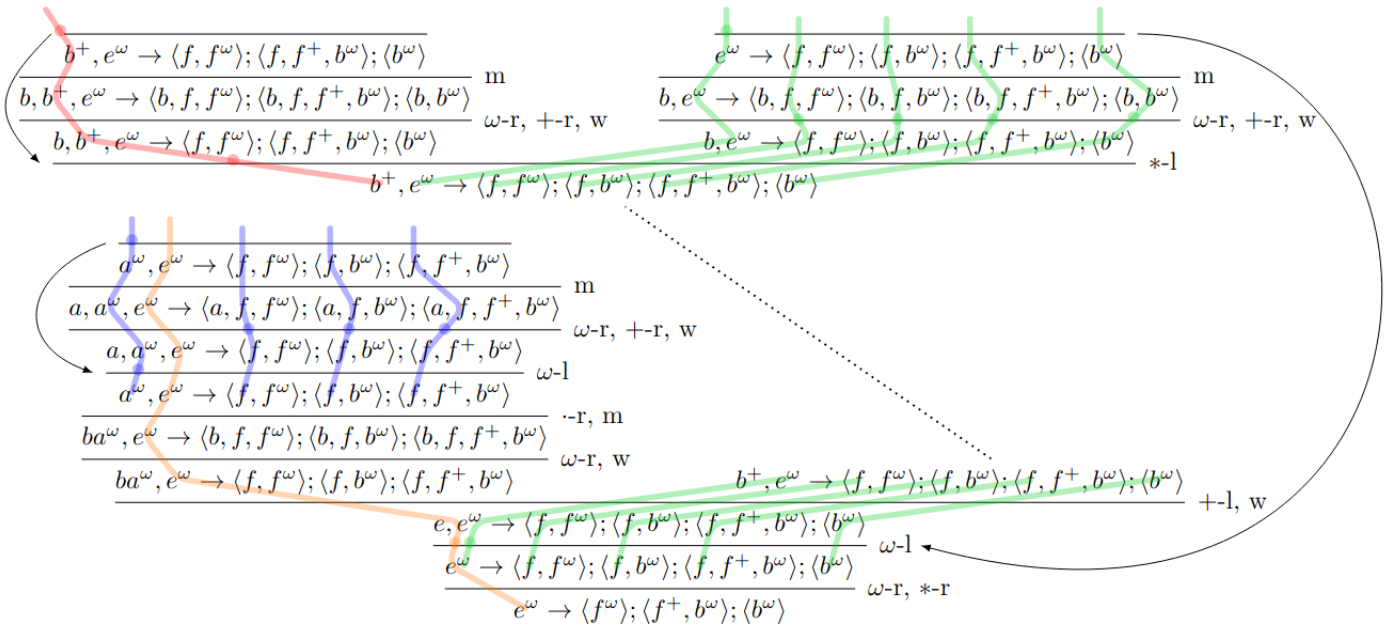
We then go up in the leftmost branch, while updating T_2 at each step. When we reach the following sequent, we choose it as a starting point for the transition T_1 of length ω:

$$a^\omega, e^\omega \to \langle f, f^\omega\rangle; \langle f, f^+, b^\omega\rangle; \langle f, b^\omega\rangle$$

We then keep going for 4 steps, at which point we are back to that same sequent, and T_1, T_2 hold the following transitions:

$$T_1 = \frac{a^\omega, e^\omega \to \langle f, f^\omega\rangle; \langle f, f^+, b^\omega\rangle; \langle f, b^\omega\rangle}{a^\omega, e^\omega \to \langle f, f^\omega\rangle; \langle f, f^+, b^\omega\rangle; \langle f, b^\omega\rangle} \qquad T_2 = \frac{a^\omega, e^\omega \to \langle f, f^\omega\rangle; \langle f, f^+, b^\omega\rangle; \langle f, b^\omega\rangle}{e^\omega \to \langle f^\omega\rangle; \langle f^+, b^\omega\rangle; \langle b^\omega\rangle}$$

Note that T_1 is idempotent, and we can now take its limit, corresponding to the limit of a branch of length ω. The associated transition is the following one.



Following the blue thread leads back to that same tree.

Following the green thread leads to the new tree: $\dfrac{\dfrac{}{\to \langle\rangle}\ \text{id}}{\to \langle f^\omega\rangle; \langle b^\omega\rangle; \langle f^+, b^\omega\rangle; \langle\rangle}\ \text{w}$

And following the orange thread: $\dfrac{}{\to \langle\rangle}\ \text{id}$

Figure 3.14: Example of a proof in $\mathcal{S}_t$. The dotted line is used to continue building the tree further up, so that it can fit in the page width.

$$\frac{e^\omega \to \langle f^\omega\rangle; \langle f^+, b^\omega\rangle; \langle b^\omega\rangle}{a^\omega, e^\omega \to \langle f, f^\omega\rangle; \langle f, f^+, b^\omega\rangle; \langle f, b^\omega\rangle}$$

And we can update T_2 by composing its current value with that transition, to get this one:

$$T_2 = \frac{e^\omega \to \langle f^\omega\rangle; \langle f^+, b^\omega\rangle; \langle b^\omega\rangle}{e^\omega \to \langle f^\omega\rangle; \langle f^+, b^\omega\rangle; \langle b^\omega\rangle}$$

It is also idempotent, with limit transition:

$$\frac{\to \langle\rangle}{e^\omega \to \langle f^\omega\rangle; \langle f^+, b^\omega\rangle; \langle b^\omega\rangle}$$

The only list on the right corresponds to the part of $\langle f^\omega\rangle$ that is right of f^ω, *i.e.* the empty list. The algorithm reached the end of the branch, since this sequent is proved by the id rule, and has not found any invalid branch. Doing so for every branch ensures the validity of the tree.

3.4 Conclusion of Chapter 3

In our completeness proof, the sets of lists on the right sides of sequents perform some kind of powerset construction. In doing so, we avoid an intricate determinisation procedure such as the Safra construction [Saf88]. We believe it can be considered that this complexity of determinisation is "hidden" in the validity condition, following various infinite threads simultaneously. This has the advantage of modularity: we separate the pure powerset construction, located in the sequents of the proof, from the complexity of dealing with the acceptance condition, located in the validity condition of the proof. Whereas when determinising Büchi automata, these two causes for state-blowup are merged in the states of the resulting deterministic Rabin automaton. A more detailed investigation of this phenomenon and its advantages can be the subject of a future work.

Contrarily to what happens on ω words, the transfinite system $\mathcal{S}_t$ cannot be seen as an instance of a proof system for linear μ-calculus, as $\cdot^\omega$ is no longer a fixed point operator in the transfinite setting. This manifests concretely by the loss of symmetry between $\cdot^+$ and $\cdot^\omega$ in the validity condition when going from S_ω to $\mathcal{S}_t$.

Although this was not the focus of this work, the computational content of systems similar to the one presented here has been studied in the past: [KPP21] gives an equivalence between a restriction of the systems from [DP17; DP18] and Gödel's system T. Our forest-based system (instead of trees) might allow for an interesting generalization of those results, using some interpretation of those proofs that remains to be defined.

This system admits cut elimination, meaning that for any proof with cut, there is a cut-free proof of the same result. This is sometimes called cut-admissibility to distinguish from explicit cut elimination, *i.e.* an algorithm transforming a proof into a cut free one. We did not provide any such algorithm (better than the PSpace naive one rebuilding the proof), and this might be the focus of a future work.

Bibliography

[Abd+10] Parosh Aziz Abdulla et al. "Simulation Subsumption in Ramsey-Based Büchi Automata Universality and Inclusion Testing". In: *CAV*. Vol. 6174. LNCS. Springer Verlag, 2010, pp. 132–147. DOI: 10.1007/978-3-642-14295-6_14.

[AK21] Bader Abu Radi and Orna Kupferman. "Minimization and Canonization of GFG Transition-Based Automata". In: *CoRR* abs/2106.06745 (2021).

[AKL21] Bader Abu Radi, Orna Kupferman, and Ofer Leshkowitz. "A Hierarchy of Nondeterminism". In: *46th International Symposium on Mathematical Foundations of Computer Science, MFCS 2021*. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[AL17] Bahareh Afshari and Graham E. Leigh. "Cut-free completeness for modal mu-calculus". In: *LICS*. IEEE, 2017, pp. 1–12. DOI: 10.1109/LICS.2017.8005088.

[Bed96] Nicolas Bedon. "Finite Automata and Ordinals". In: *Theor. Comput. Sci.* 156.1&2 (1996), pp. 119–144. URL: https://doi.org/10.1016/0304-3975(95)00006-2.

[Ber+19] Nathalie Bertrand et al. "Controlling a population". In: *Log. Methods Comput. Sci.* 15.3 (2019).

[BK18] Marc Bagnol and Denis Kuperberg. "Büchi Good-for-Games Automata Are Efficiently Recognizable". In: *38th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2018*. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[BL21] Udi Boker and Karoliina Lehtinen. "History Determinism vs. Good for Gameness in Quantitative Automata". In: *41st IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2021*. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[BL22] Udi Boker and Karoliina Lehtinen. "Token Games and History-Deterministic Quantitative Automata". In: *Foundations of Software Science and Computation Structures - 25th International Conference, FOSSACS 2022*. Lecture Notes in Computer Science. Springer, 2022.

[Bok+13] Udi Boker et al. "Nondeterminism in the Presence of a Diverse or Unknown Future". In: *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013*. Lecture Notes in Computer Science. Springer, 2013.

[Bok+20] Udi Boker et al. "On the Succinctness of Alternating Parity Good-For-Games Automata". In: *40th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2020*. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[BP13] Filippo Bonchi and Damien Pous. "Checking NFA equivalence with bisimulations up to congruence". In: *POPL*. ACM, 2013, pp. 457–468. ISBN: 978-1-4503-1832-7. DOI: 10.1145/2429069.2429124.

[Bro05] James Brotherston. "Cyclic Proofs for First-Order Logic with Inductive Definitions". In: *TABLEAUX*. Vol. 3702. Lecture Notes in Artificial Intelligence. Springer Verlag, 2005, pp. 78–92. DOI: 10.1007/11554554_8.

[BS07] James Brotherston and Alex Simpson. "Complete Sequent Calculi for Induction and Infinite Descent". In: *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*. 2007, pp. 51–62. DOI: 10.1109/LICS.2007.16.

[Bus91] Samuel Buss. "The Undecidability of K-Provability". In: *Annals of Pure and Applied Logic* 53.1 (1991), pp. 75–102. DOI: 10.1016/0168-0072(91)90059-U.

[Cal+17] Cristian S. Calude et al. "Deciding parity games in quasipolynomial time". In: *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017*. ACM, 2017, pp. 252–263.

[CF19] Thomas Colcombet and Nathanaël Fijalkow. "Universal Graphs and Good for Games Automata: New Tools for Infinite Duration Games". In: *Foundations of Software Science and Computation Structures - 22nd International Conference, FOSSACS 2019*. Lecture Notes in Computer Science. Springer, 2019.

[Cho78] Yaacov Choueka. "Finite Automata, Definable Sets, and Regular Expressions over ω^n-Tapes". In: *J. Comput. Syst. Sci.* 17.1 (1978), pp. 81–97. DOI: 10.1016/0022-0000(78)90036-3. URL: https://doi.org/10.1016/0022-0000(78)90036-3.

[CHP07] Krishnendu Chatterjee, Thomas A. Henzinger, and Nir Piterman. "Generalized Parity Games". In: *Foundations of Software Science and Computational Structures, 10th International Conference, FOSSACS 2007*. Ed. by Helmut Seidl. Lecture Notes in Computer Science. Springer, 2007.

[CLS15] James Cranch, Michael R. Laurence, and Georg Struth. "Completeness results for omega-regular algebras". In: *J. Log. Algebr. Meth. Program.* 84.3 (2015), pp. 402–425. DOI: 10.1016/j.jlamp.2014.10.002. URL: https://doi.org/10.1016/j.jlamp.2014.10.002.

[Col09] Thomas Colcombet. "The theory of stabilisation monoids and regular cost functions". In: *Automata, languages and programming. Part II*. Vol. 5556. Lecture Notes in Comput. Sci. Berlin: Springer, 2009, pp. 139–150.

[Col12] Thomas Colcombet. "Forms of Determinism for Automata (Invited Talk)". In: *29th International Symposium on Theoretical Aspects of Computer Science, STACS 2012*. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012.

[CR20] Liron Cohen and Reuben N. S. Rowe. "Integrating Induction and Coinduction via Closure Operators and Proof Cycles". In: *Automated Reasoning*. Ed. by Nicolas Peltier and Viorica Sofronie-Stokkermans. Cham: Springer International Publishing, 2020, pp. 375–394. ISBN: 978-3-030-51074-9.

[Das18] Anupam Das. "On the logical complexity of cyclic arithmetic". In: *CoRR* abs/1807.10248 (2018).

[DHL06] Christian Dax, Martin Hofmann, and Martin Lange. "A Proof System for the Linear Time μ-Calculus". In: *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science*. Ed. by S. Arun-Kumar and Naveen Garg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 273–284.

[Dou+16] Amina Doumane et al. "Towards Completeness via Proof Search in the Linear Time μ-calculus: The case of Büchi inclusions". In: *LICS*. ACM, 2016, pp. 377–386. DOI: 10.1145/2933575.2933598.

[DP17] Anupam Das and Damien Pous. "A Cut-Free Cyclic Proof System for Kleene Algebra". In: *Automated Reasoning with Analytic Tableaux and Related Methods - 26th International Conference, TABLEAUX 2017, Brasília, Brazil, September 25-28, 2017, Proceedings*. Ed. by Renate A. Schmidt and Cláudia Nalon. Vol. 10501. Lecture Notes in Computer Science. Springer, 2017, pp. 261–277.

[DP18] Anupam Das and Damien Pous. "Non-Wellfounded Proof Theory For (Kleene+ Action)(Algebras + Lattices)". In: *27th EACSL Annual Conference on Computer Science Logic, CSL 2018, September 4-7, 2018, Birmingham, UK*. 2018, 19:1–19:18.

[DR10] Stephane Demri and Alexander Rabinovich. "The complexity of linear-time temporal logic over the class of ordinals". In: *Logical Methods in Computer Science* 6 (Sept. 2010). DOI: 10.2168/LMCS-6(4:9)2010.

[EJ91] E. Allen Emerson and Charanjit S. Jutla. "Tree Automata, Mu-Calculus and Determinacy (Extended Abstract)". In: *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*. IEEE Computer Society, 1991, pp. 368–377.

[GS16] David Gale and F. M. Stewart. "13. Infinite Games with Perfect Information". In: *Contributions to the Theory of Games (AM-28), Volume II*. Princeton University Press, 2016, pp. 245–266. DOI: doi:10.1515/9781400881970-014.

[HP06] Thomas A. Henzinger and Nir Piterman. "Solving Games Without Determinization". In: *Computer Science Logic, 20th International Workshop, CSL 2006*. 2006.

[Hro+00] Juraj Hromkovic et al. "Measures of Nondeterminism in Finite Automata". In: *Electronic Colloquium on Computational Complexity (ECCC)* 7 (Jan. 2000).

[KM19] Denis Kuperberg and Anirban Majumdar. "Computing the Width of Nondeterministic Automata". In: *Log. Methods Comput. Sci. (LMCS)* 15.4 (2019).

[Koz94] D. Kozen. "A Completeness Theorem for KLeene Algebras and the Algebra of Regular Events". In: *Information and Computation* 110.2 (1994), pp. 366–390. DOI: 10.1006/inco.1994.1037.

[KPP21] Denis Kuperberg, Laureline Pinault, and Damien Pous. "Cyclic proofs, system t, and the power of contraction". In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–28.

[KS15] Denis Kuperberg and Michał Skrzypczak. "On Determinisation of Good-for-Games Automata". In: *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015*. Lecture Notes in Computer Science. Springer, 2015.

[KS16] Ioannis Kokkinis and Thomas Studer. "Cyclic proofs for linear temporal logic". In: *Concepts of Proof in Mathematics, Philosophy, and Computer Science* 6 (2016), p. 171.

[LJB01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. "The Size-Change Principle for Program Termination". In: *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '01. New York, NY, USA: Association for Computing Machinery, 2001, pp. 81–92. ISBN: 1581133367. DOI: 10.1145/360204.360210. URL: https://doi.org/10.1145/360204.360210.

[LZ20] Karoliina Lehtinen and Martin Zimmermann. "Good-for-games ω-Pushdown Automata". In: *LICS 2020: 35th Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM, 2020, pp. 689–702.

[MH84] Satoru Miyano and Takeshi Hayashi. "Alternating finite automata on ω-words". In: *Theoret. Comput. Sci.* 32.3 (1984), pp. 321–330.

[Niw91] Damian Niwinski. "On the Cardinality of Sets of Infinite Trees Recognizable by Finite Automata". In: *Mathematical Foundations of Computer Science 1991, 16th International Symposium, MFCS'91*. Lecture Notes in Computer Science. Springer, 1991.

[NST19] Rémi Nollet, Alexis Saurin, and Christine Tasson. "PSPACE-Completeness of a Thread Criterion for Circular Proofs in Linear Logic with Least and Greatest Fixed Points". In: *Automated Reasoning with Analytic Tableaux and Related Methods - 28th International Conference, TABLEAUX 2019, London*. Lecture Notes in Computer Science. Springer, 2019.

[PSA17] Alexandros Palioudakis, Kai Salomaa, and Selim G. Akl. "Worst Case Branching and Other Measures of Nondeterminism". In: *Int. J. Found. Comput. Sci.* 28.3 (2017), pp. 195–210.

[Saf88] S. Safra. "On the complexity of omega-automata". In: *[Proceedings 1988] 29th Annual Symposium on Foundations of Computer Science*. 1988, pp. 319–327. DOI: 10.1109/SFCS.1988.21948.

[Sch20] Sven Schewe. "Minimising Good-For-Games Automata Is NP-Complete". In: *40th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2020*. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[Sim17] Alex Simpson. "Cyclic Arithmetic Is Equivalent to Peano Arithmetic". In: *FoSSaCS*. Vol. 10203. Lecture Notes in Computer Science. Springer Verlag, 2017, pp. 283–300. DOI: 10.1007/978-3-662-54458-7_17.

[Tho08] Wolfgang Thomas. "Church's Problem and a Tour through Automata Theory". In: *Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*. Lecture Notes in Computer Science. Springer, 2008.

[WS91] Andreas Weber and Helmut Seidl. "On the Degree of Ambiguity of Finite Automata". In: *Theor. Comput. Sci.* 88.2 (1991), pp. 325–349.

Index