Université Paris Cité

École doctorale Sciences Mathématiques de Paris Centre (ED 386)
Laboratoire : Institut de Recherche en Informatique Fondamentale (UMR 8243)

Optimization of string transducers

Par **Gaëtan DOUÉNEAU-TABOT**

Thèse de doctorat en informatique

Dirigée par **Olivier CARTON**
Et par **Emmanuel FILIOT**

Présentée et soutenue publiquement le 23/11/2023 devant un jury composé de :

Olivier CARTON, Professeur des Universités, Université Paris Cité, Directeur

Emmanuel FILIOT, Maître de Recherche FNRS, Université Libre de Bruxelles, Co-directeur

Mikołaj BOJAŃCZYK, Full Professor, University of Warsaw, Rapporteur

Anca MUSCHOLL, Professeure des Universités, Université de Bordeaux, Rapporteure

Nathalie BERTRAND, Directrice de Recherche, INRIA, Examinatrice

Pierre-Alain REYNIER, Professeur des Universités, Aix-Marseille Université, Examinateur

Paul GASTIN, Professeur des Universités émérite, ENS Paris-Saclay, Membre invité

Version of 8th December 2023

Dans l'optique des Castaliens, la vie du siècle était un élément arriéré et de valeur secondaire, une existence de désordre et d'instincts primitifs, fait de passions et de dispersion, sans beauté, sans rien qui méritât le désir. Mais le siècle et sa vie étaient en vérité infiniment plus grands et plus riches qu'un Castalien ne pouvait se les représenter, le monde était plein de devenir, d'histoire, d'essais et d'éternels recommencements ; il était chaotique, mais il était la patrie et le sol nourricier de tous les destins.

Hermann Hesse, *Le Jeu des perles de verre*
(trad. J. Martin)

Résumé en français

Titre : Optimisation de transducteurs sur les mots.

Résumé : Les transducteurs sont des machines à états finis qui calculent des fonctions (ou des relations) des mots vers les mots. Ils peuvent être considérés comme des programmes dont la mémoire est limitée et qui manipulent des chaînes de caractères. Ces machines ont été étudiées depuis longtemps en informatique fondamentale, au sein de la théorie des automates, et sont utilisées dans de nombreux domaines comme la compilation, le traitement des langages naturels, ou le traitement des flux de données.

Dans la littérature, de nombreux modèles de transducteurs ont été définis grâce à des fonctionnalités qui permettent d'augmenter l'expressivité des machines (comme le non-déterminisme, la bidirectionnalité ou l'imbrication). Dans ce contexte, une question naturelle est celle de l'appartenance à une classe : étant donnée une fonction calculée par un transducteur avec des fonctionnalités « complexes », peut-on la calculer avec un transducteur « plus simple » ? Certains de ces problèmes ont déjà été résolus, et ils sont en général considérés comme difficiles. D'un point de vue pratique, ils s'interprètent comme des questions d'optimisation de programmes : étant donné un programme qui utilise beaucoup de ressources, peut-on construire un programme équivalent qui est « plus efficace » ?

Cette thèse propose de résoudre plusieurs problèmes d'appartenance entre des classes de transductions existantes, à la fois sur les mots finis et infinis. Les modèles bien connus de transducteurs bidirectionnels et de transducteurs à jetons sont notamment étudiés. À chaque fois, la procédure d'appartenance est non triviale, et elle s'avère effective (dans le sens où elle construit un transducteur « plus simple » dès qu'il en existe un). C'est pourquoi les résultats de ce manuscrit peuvent être vus comme des techniques d'optimisation de programmes. En outre, nous résolvons ces problèmes par une méthode générique basée sur l'utilisation de propriétés sémantiques (c'est-à-dire qui parlent intrinsèquement des fonctions) et syntaxiques (qui parlent des transducteurs qui calculent ces fonctions).

Enfin, cette thèse fournit de nouveaux modèles et de nouvelles caractérisations pour décrire des classes de transductions connues. Ces résultats améliorent et complètent la compréhension de ces classes. L'auteur est convaincu que les différentes techniques développées dans ce manuscrit fournissent une boîte à outils pour étudier d'autres problèmes d'appartenance, qui sont encore ouverts.

Mots-clefs : automate fini, transducteur fini, transducteur bidirectionnel, transducteur à jetons, transducteur à registres, fonction rationnelle, fonction régulière, fonction polyrégulière, problème d'appartenance à une classe, optimisation de programmes.

Abstract

Title: Optimization of string transducers.

Abstract: Transducers are finite-state machines which compute functions (or relations) from words to words. They can be seen as simple programs with limited memory which manipulate strings. These machines have been studied for long in fundamental computer science as a part of automata theory, and are used in many areas such as compiling, natural language processing or stream processing.

Various transducer models have been defined in the literature, thanks to many features (such as non-determinism, two-wayness or nesting) which enable to increase the expressive power of the machines. In this setting, a natural question is to solve the related class membership problems: given a function computed by a transducer with "complex" features, can it be computed by a "simpler" transducer? Some of these problems have been solved in the literature, using somehow disparate proof techniques. They are generally considered as difficult. In practice, such problems can be interpreted as program optimization issues: given a program using a lot of resources, can we build a "more efficient" equivalent program?

This thesis solves various membership problems between existing classes of transductions, both over finite or infinite words. Among others, the celebrated models of two-way transducers and pebble transducers are investigated in detail. Each time, the membership procedure is non-trivial and turns out to be effective (in the sense that it builds a "simpler" transducer whenever it exists). Therefore our results can be considered as program optimization statements. Furthermore, we offer a systematic high-level strategy for solving these problems, which relies on semantic properties (i.e. dealing intrinsically with the functions) as well as syntactic properties (referring to the transducers which compute these functions).

Additionally, this thesis provides new computation models and characterizations in order to capture known classes of transductions. These results complete the previous understanding of these classes and provide new insights on their expressive power. The author believes that the various techniques of this manuscript form a rather extensive toolbox for investigating other open membership problems.

Keywords: finite-state automaton, finite-state transducer, two-way transducer, pebble transducer, streaming string transducer, rational function, regular function, polyregular function, class membership problem, program optimization.

Remerciements

Mes premiers remerciements vont à mes deux directeurs, Olivier et Emmanuel. En 2015, alors que j'étais en prépa[1], j'avais acheté le cours de langages formels d'Olivier [Car14] pour en savoir plus sur les automates finis. J'ignorais alors que j'en viendrais à faire une thèse sur le sujet. Je remercie également l'ensemble de mon jury de thèse (Anca, Mikołaj, Nathalie, Paul et Pierre-Alain) pour leurs retours constructifs sur ce manuscrit, ainsi que pour nos nombreux échanges pendant ma soutenance.

Merci à la Direction générale de l'armement (DGA) et au corps des Ingénieurs de l'armement, qui m'ont permis de réaliser cette thèse en parallèle de mon poste d'architecte à *DGA Ingénierie & Projets*. Je remercie en particulier Patrick, Erwan, Didier, Thomas, Pascal et Alban. Merci également à tous mes collègues du ministère, qui ont supporté mes digressions sur l'importance des transducteurs déterministes bidirectionnels ou sur les carrières des docteurs dans l'administration publique française.

Cette thèse n'aurait probablement jamais vu le jour sans différents professeurs marquants de ma scolarité: Frédéric E et Nicolas S au collège-lycée, et par la suite mes différents professeurs de langages formels: Denis P en prépa, Paul et Sylvain[2] à l'ENS Cachan, Olivier, Thomas, Jean-Éric et Jacques au MPRI. J'ai en outre une pensée pour les encadrants de mes stages de recherche: à Rennes (Sophie et François), à Aix-la-Chapelle (Erich) et à Bruxelles-Cachan-Paris-Saclay (Emmanuel et Paul). Merci également aux nombreux chercheurs avec qui j'ai pu échanger des idées pendant mes trois années de thèse (en particulier Aliaume, Cécilia, Thomas, Mikołaj, Nathan, Sandra, Sarah et Tito).

Merci à tous mes amis, notamment ceux qui ont été en doctorat en même temps que moi: Aliaume, Dorian, Damien, Émilie, Julia, Laurent et Rebecca. J'ai une pensée spéciale pour Aliaume, mon co-équipier sur les projets les plus insensés depuis 2015, toujours prêt à réfléchir sur la recherche, la pédagogie ou l'administration publique. Puisque tu sembles avoir choisi ta voie (même si d'autres horizons sont toujours possibles), j'espère à mon tour que tu t'y plairas. Merci également à Laurent, mon co-équipier sur d'autres projets insensés (de préférence de nuit et à pied). Émilie, maintenant que tu es *à nouveau* en prépa, j'espère que nous trouverons tous le temps pour reprendre l'écriture d'un certain livre. Je pense également à mes camarades de prépa (notamment Christophe, Clément, Côme, Maëlyne, Pierre, Solenn et Vincent), avec qui j'ai partagé le sens de l'amitié autant que la joie des mathématiques. Enfin, merci à Maria de m'avoir supporté pendant la rédaction de ces presque 300 pages.

Rien n'aurait été possible sans les conseils avisés de mes parents, et grâce à la patience de ma mère dans ses nombreuses relectures de mes productions, qu'il s'agisse de rapports ou d'articles de recherche. Je pense aussi et en particulier à mon grand-père Léon, qui m'a certainement donné le goût des sciences.

Enfin, je ne serais sans doute jamais venu à bout de la rédaction de mes travaux de recherche sans les différentes œuvres, du baroque jusqu'au XXème siècle, que j'ai pu écouter en boucle[3] devant mon ordinateur. Un grand merci à France Musique qui me fait découvrir chaque jour de nouveaux titres.

[1] C'était une *belle époque*, dont je garderai toujours un souvenir émerveillé.
[2] Deux cours dans les styles assez différents, mais tous les deux limpides.
[3] Les boucles sont un concept important dans l'étude des automates finis.

Contents

Introduction en français

Oh ! ma France ! ô ma délaissée !

Louis Aragon, « Les Ponts-de-Cé », *Les Yeux d'Elsa*

Disclaimer: this chapter contains a French translation of the forthcoming Introduction chapter. The reader is invited to preferentially read the English version of this chapter since the English terminology (and not the French one) will be used in the rest of manuscript.

Optimisation de programmes et problèmes d'appartenance

L'*optimisation de programmes* consiste à modifier la *syntaxe* (l'implémentation) d'un programme afin de le rendre plus efficace tout en préservant sa *sémantique* (son comportement). En pratique, « rendre plus efficace » signifie souvent que le programme modifié consomme moins de ressources (temps d'exécution, mémoire, etc.). Ainsi, un algorithme de tri dont le temps d'exécution est asymptotiquement $\mathcal{O}(n \log(n))$ sur les entrées de taille n peut être vu comme une optimisation d'un algorithme de tri en $\mathcal{O}(n^2)$.

Optimisation en pratique. Rendre les programmes les plus efficaces possible est essentiel en pratique. D'une part, optimiser la consommation asymptotique de ressources est nécessaire pour que les programmes puissent passer à l'échelle sur des entrées de grande taille. C'est le cas des *programmes pour flux de données*, qui traitent une séquence arbitrairement longue d'éléments en quasi *temps réel*[4]. D'autre part, il est possible de chercher des programmes efficaces sur les petites entrées[5], auquel cas optimiser la consommation exacte de ressources est plus pertinent que l'étudier asymptotiquement.

L'optimisation peut être effectuée à plusieurs niveaux d'abstraction, du point de vue *algorithmique* (conception d'algorithmes et de structures de données) jusqu'au niveau du *code machine* (par exemple, optimiser un code en assembleur pour le rendre plus efficace sur une architecture d'ordinateur donnée). Toutefois, ce processus tend à complexifier le code, ce qui le rend plus difficile à maintenir ou à déboguer. Il est donc pertinent d'effectuer l'optimisation à la fin de la phase de développement, comme souligné par Knuth dès les années 1970 : « l'optimisation prématurée est la racine de tous les maux » [Knu74]. En complément de ces difficultés, l'*optimisation manuelle d'un programme* peut être une tâche longue (car elle nécessite de réfléchir) et risquée (car elle peut introduire des bogues) pour le programmeur.

[4] De manière informelle, le programme ne dispose que de peu de temps et de mémoire pour traiter chaque élément.

[5] Un moyen simple d'optimiser le temps d'exécution d'un programme sur les petites entrées est de pré-calculer et de coder en dur le résultat du programme sur toutes les entrées dont la taille est inférieure à une certaine borne. Cependant, cette construction augmente fortement la taille du programme lui-même.

Optimisation automatisée de programmes. Le paragraphe précédent milite en faveur de techniques *automatisées* pour l'optimisation de programmes. Dans ce cadre, l'objectif est de concevoir un méta-programme qui prend un programme en entrée et renvoie automatiquement un programme optimisé ayant la même sémantique. Cette tâche peut être considérée comme une forme particulière de *synthèse automatique des programmes*[6], dans laquelle la spécification d'entrée serait déjà un programme.

De nombreuses optimisations automatisées (en particulier pour le code de bas niveau) sont déjà mises en œuvre dans les compilateurs ou dans les processeurs[7]. Cependant, ces optimisations ne produisent en général pas un code *optimal* (au sens où il n'existerait pas de « meilleur » code) : elles suivent plutôt des méthodes *heuristiques* pour améliorer l'utilisation des ressources dans certains cas connus. Ces heuristiques donnent déjà des résultats impressionnants en pratique (voir par exemple [Leu00]).

L'objectif pratique de ce manuscrit est de décrire des procédures d'optimisation qui garantissent que le programme construit est toujours optimal[8]. Ces procédures supposent qu'une métrique a été choisie au préalable pour comparer l'efficacité des programmes et ainsi définir ce que signifie « optimal ». Des résultats classiques d'indécidabilité rendent rapidement impossible la construction d'un programme optimal en général, c'est pourquoi nous restreindrons l'étude à des programmes « simples ».

De l'optimisation de programmes aux problèmes d'appartenance. D'un point de vue fondamental, l'objectif de ce manuscrit est d'étudier les *problèmes d'appartenance à une sous-classe*. Considérons une classe P de programmes (par exemple, les programmes dont le temps d'exécution est polynomial dans la taille n de l'entrée) et une sous-classe $\mathsf{P'} \subseteq \mathsf{P}$ de programmes cibles considérés comme « efficaces » (par exemple, les programmes dont le temps d'exécution est $\mathcal{O}(n)$). Dans ce cadre, le problème d'appartenance de P à $\mathsf{P'}$ est formellement défini comme suit :

- **Entrée :** un programme $\pi \in \mathsf{P}$ dont la sémantique est une fonction f ;
- **Question :** est-ce qu'il existe un programme $\pi' \in \mathsf{P}'$ dont la sémantique est f ?

En d'autres termes, ce problème demande si une fonction de la « grande » classe $\mathcal{C}$ de la Figure 1 appartient en fait à la « petite » classe $\mathcal{C}'$. Il ne traite que de la sémantique et pas de la syntaxe.

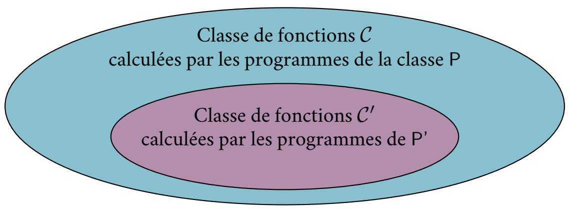


Figure 1: Représentation du problème d'apparence entre les classes P et $\mathsf{P'}$.

Résoudre[9] le problème d'appartenance signifie construire un algorithme qui répond automatiquement à la **question** lorsqu'on lui donne un programme $\pi \in \mathsf{P}$ en **entrée**. Un tel algorithme peut presque être vu comme une procédure d'optimisation, à ceci près qu'il se contente d'indiquer si un programme optimisé $\pi' \in \mathsf{P'}$ existe, mais il ne le construit pas explicitement. Néanmoins, pour tous les problèmes d'appartenance à une classe qui sont résolus dans ce manuscrit, la preuve est *effective* et elle construit π'.

[6] La *synthèse de programme* consiste à construire un programme qui satisfait une spécification formelle donnée.

[7] De telles optimisations sont susceptibles de modifier la façon dont le processeur manipule sa mémoire, et donc de placer des informations non spécifiées à des endroits inattendus. Sans trop de surprise, cette caractéristique peut créer des failles de sécurité, comme récemment la vulnérabilité CVE-2022-40982 « downfall » sur les processeurs Intel.

[8] Dans ce cas, l'optimisation est parfois appelée *superoptimization* [Mas87], mais nous ne suivrons pas cette terminologie.

[9] Formellement, nous devrions dire « montrer que le problème d'appartenance est *décidable* ».

Automates finis

Les programmes « simples » considérés dans ce manuscrit sont des *machines à états finis*. Formellement, une machine à états finis est un modèle de calcul qui possède un nombre fini d'*états* internes. A tout moment de son exécution, elle est dans un certain état et effectue une *transition* d'un état vers un autre lorsqu'elle lit un nouvel item d'entrée. Autrement dit, il s'agit de la description abstraite d'un programme dont la mémoire de travail a une taille *bornée*, c'est-à-dire qu'elle ne dépend pas de la taille de l'entrée. Des machines à états finis sont implémentées dans de nombreux dispositifs qui exécutent une séquence prédéterminée d'actions, comme les distributeurs automatiques ou les automates industriels.

Automates finis et langages réguliers. Les *automates finis déterministes* sont une classe particulière de machines à états finis dont l'entrée est un *mot* (une séquence de caractères à valeurs dans un ensemble fini) et dont la sortie est soit « oui », soit « non ». Le mot d'entrée est parcouru de gauche à droite par l'automate (cf. Figure 2) qui effectue une transition à chaque caractère lu. Ce modèle a de nombreuses applications en informatique (notamment pour l'algorithmique des flux de données, l'algorithmique du texte, la vérification formelle, la théorie du contrôle, les protocoles réseau, la conception de circuits imprimés, etc.) et dans des domaines connexes comme la linguistique ou la bio-informatique.

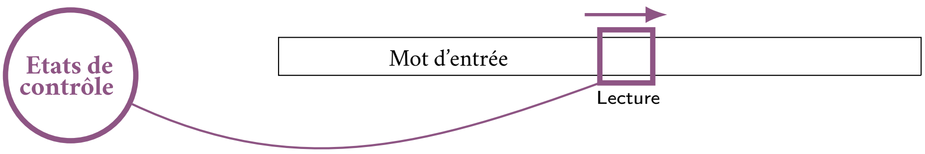


Figure 2: Fonctionnement d'un automate déterministe à un sens.

L'ensemble des mots d'entrée pour lesquels l'automate répond « oui » est appelé le *langage* calculé par l'automate. Les langages calculés par les automates finis sont dits *langages réguliers*, et ils sont considérés comme l'une des pierres angulaires de l'informatique fondamentale. Ils bénéficient de plusieurs descriptions équivalentes en termes d'expressions (les *expressions régulières* [Kle56]), de logique (la logique monadique du second ordre [Büc60, Elg61, Tra62]) et d'algèbre (les monoïdes et congruences [Ner58]).

Automate minimal. Que signifie l'« optimisation de programmes » dans le cadre des automates finis ? Un premier objectif peut être d'optimiser la mémoire utilisée par la machine. Dans ce cas, étant donné un automate, nous cherchons à construire automatiquement un autre automate avec un nombre *minimal* d'états qui calcule le même langage. Ce problème a été résolu depuis longtemps, par exemple avec les algorithmes de Moore [Moo56] ou de Hopcroft [Hop71].

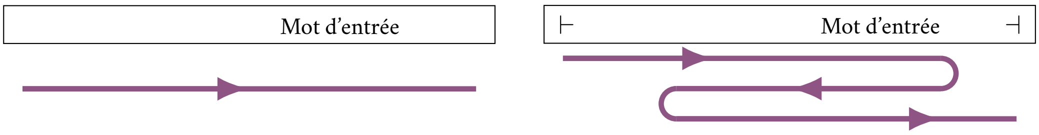
Étant donné un langage régulier, il existe en réalité un unique automate déterministe qui le calcule et dont le nombre d'états est minimal. Cet unique objet est appelé l'*automate minimal* du langage. En conséquence, les algorithmes de minimisation mentionnés ci-dessus ne se contentent pas de réduire le nombre d'états : ils construisent en fait un *objet canonique* (dans le sens où il ne dépend que du langage, mais pas de l'automate qui a été donné en entrée) associé à un langage régulier donné. De manière générale, la construction de modèles canoniques est très pertinente pour résoudre les problèmes d'appartenance, car ces objets sont les mieux à mêmes de rendre explicites des informations qui sont propres à la sémantique. En outre, cette construction fournit une procédure pour décider si deux machines ont la même sémantique (en les « canonisant » puis en comparant les résultats).

Problèmes d'appartenance à des sous-classes de langages réguliers. Une autre question importante dans la théorie des automates est de comprendre les sous-classes de langages réguliers définies en restreignant l'une de leurs caractérisations (automates, expressions, logique ou algèbre). Naturellement, « comprendre une classe » est un objectif informel, mais une manière classique d'y parvenir est de résoudre le problème d'appartenance à la classe en question. En effet, les techniques développées dans ce cadre permettent généralement d'obtenir des informations approfondies sur les sous-classes.

Cette approche a été initiée par Schützenberger [Sch65], qui a fourni une procédure d'appartenance à la classe des *langages sans étoile* (une sous-classe des langages réguliers décrite[10] par des *expressions sans étoile*, qui sont une restriction des expressions régulières). Il s'avère qu'un langage régulier est sans étoile si et seulement si son automate minimal vérifie une propriété syntaxique appelée *apériodicité*[11]. Puisque cette propriété est décidable, le problème d'appartenance à la sous-classe peut donc être résolu. Dans la littérature ultérieure, cette stratégie de preuve (examiner les propriétés syntaxiques de l'automate minimal) a permis de résoudre de nombreux autres problèmes d'appartenance [Str94]. Ce sujet de recherche est encore actif de nos jours et plusieurs problèmes restent ouverts (voir par exemple [Pin17]).

Au-delà des automates finis. De manière générale, ajouter des fonctionnalités simples au modèle d'automate déterministe ne permet pas d'augmenter son expressivité. Nous mentionnons en particulier les extensions suivantes du modèle de base (qui lui sont équivalentes) :

- les *automates non déterministes*, qui permettent de « deviner » une propriété de l'entrée pendant une exécution, et d'en vérifier la validité par la suite. La transformation effective d'un automate non déterministe en un automate déterministe est un exercice classique ;
- les automates *bidirectionnels* (déterministes ou non), qui peuvent se déplacer vers la droite et vers la gauche sur leur entrée, alors que le modèle mentionné jusqu'à présent (que nous appellerons désormais automate *unidirectionnel*) n'est capable que de se déplacer vers la droite (comparer la Figure 3a et la Figure 3b). L'équivalence entre les deux provient de [She59] ;



(a) Exécution d'un automate unidirectionnel. (b) Exécution d'un automate bidirectionnel.

Figure 3: Exécutions d'un automate unidirectionnel et d'un automate bidirectionnel.

- les *automates imbriqués* (bidirectionnels ou non) (déterministes ou non), qui peuvent appeler des automates auxiliaires pendant leur exécution. Dans ce manuscrit, nous mentionnerons plus en détail le modèle des *automates à jetons*[12] tel qu'introduit dans [EH99].

Autrement dit, toutes les variantes « raisonnables » des automates déterministes ne calculent pas mieux que les langages réguliers, ce qui justifie encore l'importance et la robustesse de cette classe. Une exception notable est l'utilisation d'une *pile* auxiliaire, qui augmente radicalement le pouvoir expressif des automates. Les automates unidirectionnels non déterministes avec pile sont appelés *automates à pile* et calculent la célèbre classe des *langages algébriques*. Il est bien connu (voir par exemple [HMU07]) que le problème d'appartenance d'un langage algébrique aux langages réguliers est indécidable[13].

[10] Cette sous-classe possède également des caractérisations en termes d'automates, de logique et d'algèbre.
[11] Les automates apériodiques sont aussi appelés automates *sans compteur* [MP71].
[12] La traduction littérale de « pebble automata » est en réalité « automate à galets » ou « à cailloux ».
[13] Ce problème est néanmoins décidable en partant de la sous-classe des *langages algébriques déterministes* [Ste67].

Transducteurs finis

Ce manuscrit se concentre sur les *transducteurs finis*, qui sont des automates finis enrichis avec des sorties. Formellement, un transducteur est une machine à états finis définie en partant d'un modèle d'automate et en ajoutant une sortie sur chacune de ses transitions. Sur une entrée donnée, la machine renvoie la concaténation des sorties produites le long des transitions de son exécution : elle calcule donc une *fonction* (lorsqu'elle est déterministe) ou une *relation* (lorsqu'elle est non déterministe) des mots vers les mots. Les transducteurs sont utilisés dans de nombreux domaines tels que la compilation [FCL10, Chapter 3], le traitement des langages naturels [MPR08] ou l'arithmétique des ordinateurs. De plus, ils fournissent un environnement plus complet que les automates finis pour modéliser des programmes simples.

Expressivité des transducteurs. Il est possible de définir une grande variété de modèles de transducteurs, qui sont unidirectionnels ou bidirectionnels, déterministes ou non, imbriqués ou non, etc. Le comportement d'un transducteur bidirectionnel déterministe est par exemple illustré dans la Figure 4.

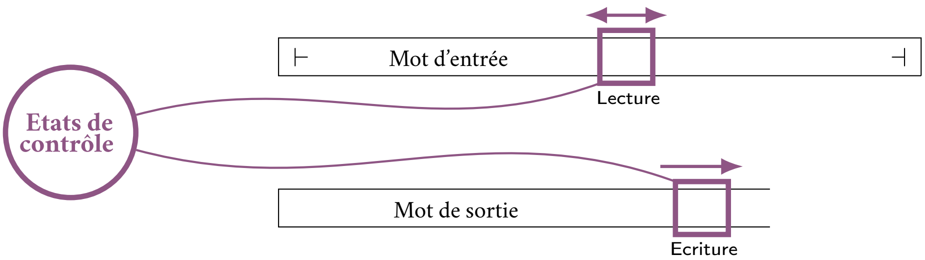


Figure 4: Fonctionnement d'un transducteur bidirectionnel déterministe.

Contrairement au cas des automates, ces différents modèles de transducteurs n'ont pas la même expressivité. En conséquence, la théorie des fonctions calculées par les transducteurs tend à être plus complexe que l'étude des langages calculés par les automates, comme observé par Scott : « les fonctions calculées par les machines sont plus importantes - ou au moins plus fondamentales - que les langages que ces dernières calculent » [Sco67, Section 5]. Les phénomènes suivants se produisent :

- les *transducteurs non déterministes* sont plus expressifs que les *transducteurs déterministes*. Une raison évidente à ce phénomène est que les transducteurs non déterministes calculent des relations, alors que les transducteurs déterministes ne peuvent calculer que des fonctions. Plus subtilement, même les transducteurs non déterministes *fonctionnels* (c'est-à-dire qui calculent uniquement des fonctions) tendent à être plus expressifs que les transducteurs déterministes ;
- les *transducteurs bidirectionnels* sont plus expressifs que les transducteurs unidirectionnels. Cela vient du fait que les transducteurs bidirectionnels sont capables de renverser des (morceaux de) leur entrée, en la lisant de la droite vers la gauche, alors que les machines unidirectionnelles (même non déterministes) sont forcées de la lire de la gauche vers la droite ;
- les *transducteurs imbriqués* sont plus expressifs que les transducteurs non imbriqués. Intuitivement, l'argument est que les transducteurs imbriqués peuvent imiter les boucles « pour » imbriquées et donc produire des sorties dont la taille est polynomiale en celle de l'entrée, alors que les transducteurs non imbriqués ne produisent que des sorties de taille linéaire.

Modèles de transducteurs étudiés. Les travaux récents se concentrent notamment sur :

- les *transducteurs déterministes unidirectionnels*, qui calculent la classe des *fonctions séquentielles* ;
- les *transducteurs non déterministes unidirectionnels fonctionnels* qui calculent les *fonctions rationnelles* ;
- les *transducteurs déterministes bidirectionnels* qui calculent les *fonctions régulières* ;

▶ les *transducteurs à jetons* (= bidirectionnels imbriqués) qui calculent les *fonctions polyrégulières*.

Ces classes de fonctions sont représentées en Figure 5, où toutes les inclusions sont strictes.

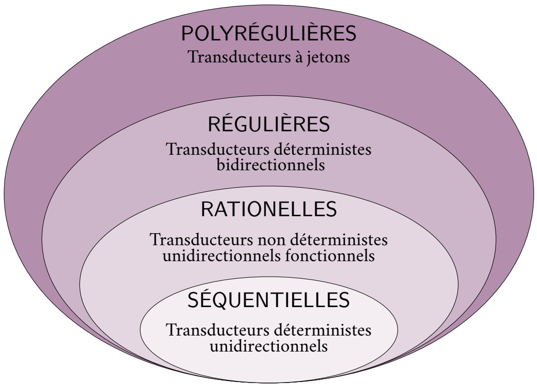


Figure 5: Classes de fonctions calculées par les transducteurs de mots finis.

Transducteurs bidirectionnels et fonctions régulières. La classe des *fonctions régulières* est souvent considérée comme l'équivalent le plus naturel[14] des langages réguliers. Elle a été étudiée ses nombreuses propriétés comme sa clôture par composition de fonctions [CJ77] ou la décidabilité du problème d'équivalence [Gur80]. Des caractérisations équivalentes de cette classe ont été données en termes d'expressions (en adaptant les expressions régulières [AFR14, DGK18, BDK18, BR18] ou comme composition de fonctions de base [BS20]) ou de logique [EH01, DFL18].

Un modèle substantiellement différent, appelé *transducteurs à registres sans copies*, capture également la classe des fonctions régulières [AC10]. De manière informelle, un transducteur à registres est un automate déterministe unidirectionnel enrichi avec des registres qui stockent des morceaux de la sortie. Les registres sont mis à jour à chaque transition. Ce modèle est à la fois plus simple (car unidirectionnel) et plus complexe (car il manipule des registres) qu'un transducteur bidirectionnel. Puisqu'il ne parcourt qu'une fois son entrée, il constitue un modèle pertinent de *programme pour flux de données*.

Transducteurs à jetons et fonctions polyrégulières. Le modèle appelé *transducteur à jetons* est obtenu en imbriquant des transducteurs déterministes bidirectionnels [MSV00, EM02, Boj18]. Un *transducteur à 1 jeton* est simplement un transducteur bidirectionnel. Un *transducteur à 2 jetons* est constitué d'un transducteur bidirectionnel qui, lorsqu'il se trouve dans une position de son d'entrée, peut appeler des transducteurs bidirectionnels auxiliaires. Ces derniers prennent en entrée le mot d'origine où la position de l'appel est marquée (nous disons qu'un *jeton* est *déposé* à cette position). Le transducteur principal renvoie finalement la concaténation de toutes les sorties de ses appels auxiliaires. Plus généralement, un *transducteur à k jetons* pour $k \geqslant 1$ est constitué de transducteurs bidirectionnels imbriqués jusqu'à une profondeur k. Une exécution partielle d'un transducteur à 3 jetons est illustrée en Figure 6.

Un *transducteur à k jetons* peut aussi être vu comme un programme qui exécute des boucles « pour » imbriquées, dans lequel le i-ème indice de boucle imbriquée est la position du i-ième jeton. De ce point de vue, il est facile d'observer qu'un transducteur à k jetons peut produire une sortie dont la taille est polynomiale en la longueur n de l'entrée, et plus précisément en $\mathcal{O}(n^k)$ puisqu'il a k boucles imbriquées.

[14]Cette notion est hautement informelle, et les autres classes sont aussi des équivalents très naturels des langages réguliers.

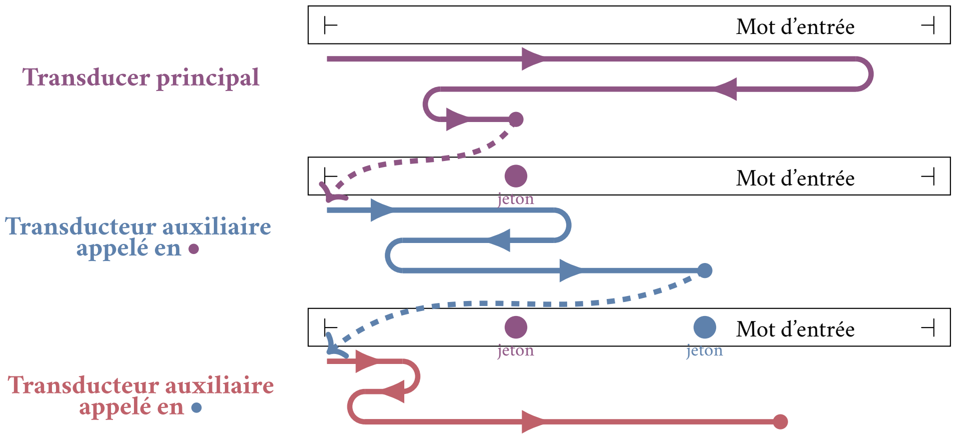


Figure 6: Exécution d'un transducteur à 3 jetons.

Comme mentionné ci-dessus, la classe des *fonctions polyrégulières* est définie comme la classe des fonctions calculées par des *transducteurs à jetons*. Plusieurs propriétés de cette classe telles que sa clôture par composition de fonctions [EM02] sont connues depuis longtemps. L'étude assez exhaustive de Bojańczyk [Boj18] a créé un regain intérêt récent pour les fonctions polyrégulières. Plusieurs caractérisations équivalentes ont été données, en termes d'expressions (comme composition de fonctions de base [Boj18]) ou de logique [BKL19]. D'autres formalismes équivalents ont été introduits, comme un langage de programmation impératif nommé *transducteurs à boucles « pour »*, un langage de programmation fonctionnel basé sur le λ-calcul, ou un système de types spécifique [Boj18, Boj23a].

Problèmes d'appartenance pour les transducteurs

La diversité des classes de fonctions calculées par des transducteurs fait apparaître de nombreux problèmes d'appartenance qui n'existaient pas pour les automates (puisque tous les modèles étaient équivalents). Certains ont été résolus dans la littérature, via des techniques de preuves assez disparates.

La question des modèles canoniques. Comme observé dans le cas des automates, une approche naturelle pour résoudre les problèmes d'appartenance à une sous-classe est de décrire une procédure pour transformer une machine en un *objet canonique*, c'est-à-dire qui ne dépend que de la sémantique de la machine et pas de sa syntaxe. Dans le cas des transducteurs, des modèles canoniques peuvent être construits pour les fonctions séquentielles et rationnelles [RS91, Cho03, FGL19].

Ces modèles canoniques ont été utilisés avec succès pour décider si une fonction rationnelle est *sans étoile* [FGL19] (la notion de *sans étoile* pour les fonctions rationnelles étant définie comme un analogue des langages sans étoile). En outre, ils permettent de décider si une fonction rationnelle est en réalité séquentielle. Historiquement, ce résultat a en fait été obtenu dans plusieurs articles sans utiliser de modèles canoniques [Cho77, WK95, BCPS03]: la preuve classique consiste à montrer que tout transducteur non-déterministe unidirectionnel (et pas seulement l'objet canonique) qui calcule une fonction séquentielle vérifie une *propriété syntaxique* (décidable), souvent appelée *propriété de jumelage*.

Optimisation des transducteurs bidirectionnels. La construction d'un modèle canonique n'est malheureusement pas connue en général[15] pour les transducteurs bidirectionnels et les fonctions régulières.

[15] Néanmoins, il est possible de construire un modèle canonique dans le cas des *fonctions régulières avec sémantique d'origine* (voir [Boj14]). Des résultats partiels sont également connus dans cas restreints de transducteurs bidirectionnels [LLN+11].

Par conséquent, il semble difficile de décider des propriétés qui concernent la sémantique de ces fonctions, car elles peuvent être représentées de plusieurs manières (apparemment) sans lien les unes avec les autres. En particulier, décider si une fonction régulière est *sans étoile* (là encore, cette notion étant formellement définie par analogie avec les langages sans étoiles) est un problème ouvert.

Il est néanmoins possible de décider si une fonction régulière est rationnelle [FGRS13, BGMP18]. Les preuves de ce résultat reposent sur une étude assez combinatoire du comportement des transducteurs bidirectionnels. Une fois de plus, ce résultat peut être vu comme une procédure d'optimisation de programme puisqu'il construit un transducteur unidirectionnel (= plus efficace) dès qu'il en existe un.

Optimisation des transducteurs à jetons. Etant donnée une fonction calculée par un transducteur à ℓ jetons, une question très naturelle est de savoir si elle peut être calculée par un transducteur à k jetons pour $k \leqslant \ell$ fixé. Ce problème s'interprète facilement en termes d'optimisation, puisqu'il s'agit de passer d'un programme comportant ℓ boucles imbriquées (c'est-à-dire dont le temps d'exécution est $\mathcal{O}(n^\ell)$ sur des entrées de taille n) à un programme ne comportant que k boucles imbriquées (donc en $\mathcal{O}(n^k)$). De manière équivalente, il s'agit de savoir si la profondeur d'appel de fonctions peut être minimisée.

Comme expliqué plus haut, un transducteur à k jetons produit un mot dont la taille est en $\mathcal{O}(n^k)$ lorsque n est la taille de l'entrée. Nous pourrions donc conjecturer qu'une fonction calculée par un transducteur à ℓ jetons peut être calculée par un transducteur à k jetons si et seulement si sa sortie est en $\mathcal{O}(n^k)$. Ce résultat est vrai pour $k = 1$ et permet de décider si une fonction est calculable par un transducteur à 1 seul jeton[16] [Boj22]. Cependant, la conjecture est fausse en général : pour tout $k \geqslant 3$, il existe une fonction dont la sortie est $\mathcal{O}(n^2)$ mais qui ne peut pas être calculée par un transducteur ayant moins de k jetons [Boj22, Boj23b]. Les problèmes d'appartenance associés sont ouverts.

Transducteurs de mots infinis et calculabilité. Des automates traitant les *mots infinis* (= séquences infinies de caractères) ont été étudiés dès les origines de la théorie des automates, à la suite des travaux de Büchi [Büc62]. Ces machines sont essentiellement construites comme les automates de mots finis, à ceci près que leur exécution est infinie puisqu'elles doivent lire l'intégralité de leur entrée. Elles définissent un analogue célèbre des langages réguliers pour les mots infinis, appelés *langages ω-réguliers* (voir par exemple [PP04] pour une introduction). Il n'aura pas échappé au lecteur que, dans la pratique, les entrées d'un programme sont rarement infinies. C'est effectivement le cas, néanmoins les mots infinis peuvent être considérés comme une manière de représenter des *flux de données* arbitrairement longs.

Plusieurs modèles de transducteurs à entrée et sortie infinies ont été étudiés dans la littérature. Les plus célèbres d'entre eux sont définis par analogie avec les transducteurs de mots finis:

- les *transducteurs déterministes unidirectionnels*, qui calculent les *fonctions séquentielles de mots infinis*;
- les *transducteurs non-déterministes unidirectionnels*, pour les *fonctions rationnelles de mots infinis*;
- les *transducteurs déterministes bidirectionnels* enrichis avec une fonctionnalité supplémentaire appelée *ω-anticipation*, qui calculent les *fonctions régulières de mots infinis* [AFT12]. De manière informelle, une *ω-anticipation* permet à la machine de vérifier une propriété « infinie » de son entrée, comme par exemple : « est-ce que le caractère 0 apparaît un nombre infini de fois ? ».

Ces classes de fonctions sont robustes et possèdent de nombreuses caractérisations équivalentes et propriétés algorithmiques. En outre, il est possible de décider si une fonction rationnelle de mots infinis est séquentielle [BC04] . Les trois classes mentionnées ci-dessus sont représentées en Figure 7.

Bien que robustes, les fonctions rationnelles et régulières de mots infinis souffrent d'une différence majeure avec le cas des mots finis. Le lecteur devrait être convaincu que toutes les transductions de mots finis mentionnées ci-dessus sont *calculables*, au sens où elles peuvent être écrites dans n'importe quel

[16] Comme les transducteurs à 1 jeton sont les transducteurs bidirectionnels, il décide si une fonction polyrégulière est régulière.

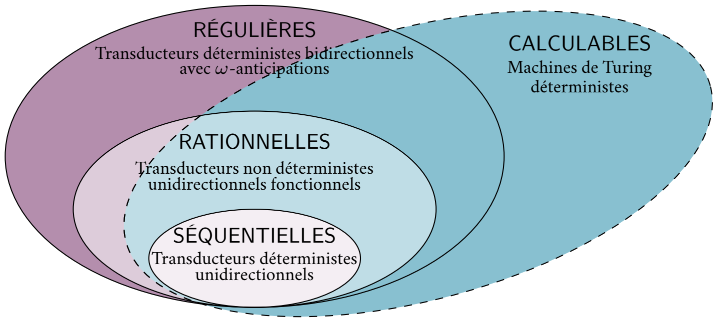


Figure 7: Classes de fonctions calculées sur les mots infinis.

langage de programmation ou, de manière équivalente, calculées par une *machine de Turing déterministe*. Ce n'est plus le cas ici : l'utilisation d'*ω-anticipations* ou de non-déterminisme dans les exécutions infinies permet de détecter, par exemple, si l'entrée contient un nombre infini de fois un caractère donné. Malheureusement, une telle propriété ne peut être vérifiée par un programme déterministe.

Un problème essentiel pour la pratique est donc de savoir si une fonction régulière de mots infinis est calculable ou non. Cette question a été récemment résolue et une procédure a été fournie pour construire un programme (machine de Turing déterministe) équivalent lorsqu'il en existe un [DFKL20]. En outre, les fonctions régulières de mots infinis qui sont calculables sont sémantiquement caractérisées[17] comme les fonctions régulières qui sont *continues* pour une certaine topologie.

Contributions de ce manuscrit

Ce manuscrit explore la plupart des résultats des sept articles publiés par l'auteur au cours de sa thèse [DFG20, Dou21, Dou22, CD22, Dou23, CDL23, CDFW23]. Plusieurs améliorations et clarifications sont proposées par rapport aux énoncés originaux. En outre, les résultats sont présentés dans un formalisme unifié. Plus concrètement, les contributions de ce manuscrit sont doubles :

- ▶ nous résolvons plusieurs *problèmes d'appartenance* entre des classes de transductions, à la fois sur les mots finis et infinis. Toutes les questions étudiées dans ce manuscrit portent sur des modèles de transducteurs qui existent déjà dans la littérature[18] et leurs solutions sont non triviales. A chaque fois, la procédure d'appartenance s'avère effective (dans le sens où elle construit un transducteur « plus simple » lorsqu'il en existe un) et elle peut donc être considérée comme une procédure d'*optimisation de programmes*. Ces résultats sont résumés dans la Table 9 ;
- ▶ nous fournissons de *nouveaux modèles de calcul* et de *nouvelles caractérisations* pour décrire plusieurs classes de transductions déjà connues. Ces résultats offrent une nouvelle compréhension de leurs pouvoirs expressifs et de leurs limites. En outre, le fait de disposer de plusieurs représentations d'un même objet s'avère très utile pour résoudre les problèmes d'appartenance associés.

Résolution des problèmes d'appartenance. Au-delà des résultats en eux-mêmes, l'auteur estime que les techniques de preuve développées dans ce manuscrit pour résoudre les problèmes d'appartenance sont également précieuses. En effet, nous suivons une stratégie systématique pour résoudre le problème

[17] Formellement, ces résultats ne sont vrais que si l'on considère les fonctions « à extension près ». Mais nous oublions délibérément cette précision dans une approche introductive informelle.
[18] Autrement dit, nous n'introduisons pas artificiellement de nouveaux modèles afin de résoudre des problèmes d'appartenance créés de toutes pièces par nos propres définitions.

d'appartenance d'une classe P de transducteurs vers une sous-classe P'. Celle-ci consiste à chercher des *caractérisations sémantiques* et *syntaxiques* de la sous-classe, comme décrit dans le Méta-théorème 8.

Méta-théorème 8 (Problème d'appartenance $\mathsf{P} \to \mathsf{P}'$)

Soit f une fonction calculée par un transducteur $\mathscr{T}$ de la classe P. Sont équivalents:

(1) f peut être calculée par un transducteur de la sous-classe P' ;
(2) f vérifie une certaine *propriété sémantique* $(\mathcal{F})$;
(3) $\mathscr{T}$ vérifie une certaine *propriété syntaxique* $(\mathcal{T})$.

En outre, $(\mathcal{T})$ est décidable et la construction Item (3) $\Rightarrow$ Item (1) est effective.

Meta-preuve du Méta-théorème 8. Item (1) $\Rightarrow$ Item (2) est en général assez facile. Pour Item (2) $\Rightarrow$ Item (3), nous utilisons des arguments combinatoires de « pompage ». Item (3) $\Rightarrow$ Item (1) constitue la procédure d'optimisation proprement dite : c'est la partie la plus difficile de la preuve. ◀

Formellement, la décidabilité du problème d'appartenance de P à P' découle du fait que la propriété $(\mathcal{T})$ est décidable. La propriété sémantique $(\mathcal{F})$ est non seulement un outil dans la preuve, mais elle est aussi utile pour montrer à la main qu'une fonction f donnée est calculable ou non par un transducteur de P'. Les différentes propriétés utilisées dans ce manuscrit sont résumées dans la Table 9.

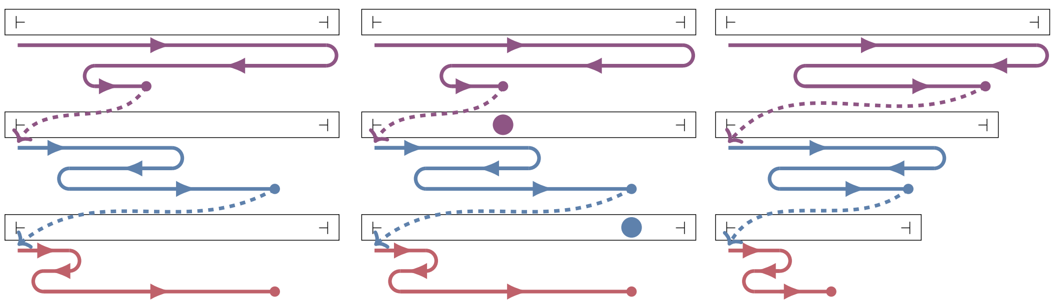
Remarquons que le Méta-théorème 8 ne traite pas d'un *objet canonique* associé à la fonction f : la propriété $(\mathcal{T})$ s'applique à tout transducteur de P. De cette façon, nous contournons les difficultés liées à la construction de modèles canoniques, au prix de preuves quelque peu combinatoires. Nous nous appuierons néanmoins sur un objet canonique pour montrer l'avant-dernière ligne de la Table 9.

Problème d'appartenance	**Propriété sémantique**	**Propriété syntaxique**	**Résultat**
Transducteur aveugle à ℓ jetons ↓ Transducteur aveugle à k jetons	Sortie de taille $\mathcal{O}(n^k)$	Transducteur pompable (Definition 3.17)	Theorem 3.12
Transducteur myope à ℓ jetons ↓ Transducteur myope à k jetons	Sortie de taille $\mathcal{O}(n^k)$	Transducteur pompable (Definition 3.25)	Theorem 3.13
Transducteur à ℓ billes ↓ Transducteur à k billes	Sortie de taille $\mathcal{O}(n^k)$	Transducteur avec haltères (Lemma 4.47)	Theorem 4.11
Transducteur récursif à billes ↓ Transducteur à k billes	Sortie de taille $\mathcal{O}(n^k)$	Transducteur avec cycles lourds (Lemma 4.47)	Theorem 4.12
Transducteur à ℓ jetons avec sortie dans $\mathbb{Z}$ ou $\mathbb{N}$ ↓ Transducteur à k jetons avec sortie dans $\mathbb{Z}$ ou $\mathbb{N}$	Sortie de taille $\mathcal{O}(n^k)$	Transducteur pompable (Definition 5.50)	Theorem 5.25

Transducteur à jetons avec sortie dans $\mathbb{Z}$ ou $\mathbb{N}$ ↓ Transducteur aveugle à jetons avec sortie dans $\mathbb{Z}$ ou $\mathbb{N}$	Fonction répétitive (Definition 6.13)	Transducteur permutable (Definition 6.28)	Theorem 6.17
Transducteur à jetons avec sortie dans $\mathbb{Z}$ ↓ Transducteur apériodique à jetons avec sortie dans $\mathbb{Z}$	Fonction lisse (Definition 7.15)	Transducteur *canonique* apériodique (Definition 7.50)	Theorem 7.19
Transducteur non déterministe unidirectionnel de mots infinis ↓ Transducteur déterministe bidirectionnel de mots infinis	Fonction continue (Proposition-Definition 8.41)	Transducteur avec propriété de jumelage (Lemma 10.8)	Theorem 10.1

Table 9: Principaux problèmes d'appartenance à une classe résolus dans ce manuscrit.

Optimisation de variantes des transducteurs à jetons. Concrètement, les premiers résultats de ce manuscrit concernent des variantes des *transducteurs à jetons*. Rappelons (cf. section précédente) que pour $1 \leqslant k \leqslant \ell$, les fonctions calculées par les transducteurs à k jetons ne coïncident malheureusement pas en général avec les fonctions calculées par les transducteurs à ℓ jetons dont la sortie est de taille $\mathcal{O}(n^k)$. En outre, les problèmes d'appartenance afférents sont ouverts.



(a) Transducteur aveugle à 3 jetons. (b) Transducteur myope à 3 jetons. (c) Transducteur à 3 billes.

Figure 10: Comportement de variantes des transducteurs à jetons.

Afin d'obtenir des résultats d'optimisation tout en contournant cette difficulté, nous nous concentrons sur trois sous-classes des transducteurs à k jetons (existantes dans la littérature), qui sont définies en affaiblissant la manière dont les machines sont imbriquées:

- les *transducteurs aveugles à k jetons* de [NNP21][19] qui sont des transducteurs à k jetons dans lesquels une machine auxiliaire ne voit aucun jeton[20]. En d'autres termes, ces machines sont des fonctions imbriquées qui ne fournissent pas la position actuelle comme argument lors d'un appel. Ce comportement est illustré dans la Figure 10a (à comparer avec la Figure 6) ;

[19] La terminologie originale de [NNP21] est « transducteurs à k jetons sans comparaisons ».
[20] C'est pourquoi le transducteur est dit « aveugle ».

- les *transducteurs myopes à k jetons* de [EHS07][21], qui sont des transducteurs à k jetons dans lesquels une machine auxiliaire ne peut voir que le jeton déposé par son parent, mais pas l'historique complet des jetons précédents[22] Ce comportement est illustré dans la Figure 10b ;
- les *transducteurs à k billes* de [EHV99], qui sont des transducteurs à k jetons dans lesquels l'entrée d'une machine auxiliaire n'est que le préfixe de l'entrée originale, tronqué à la position de l'appel. La taille de l'entrée diminue donc à chaque appel imbriqué (cf. Figure 10c). Les transducteurs à k billes peuvent être vus comme une restriction des transducteurs myopes à k jetons.

Pour tout $1 \leqslant k \leqslant \ell$, nous montrons qu'une fonction calculée par un transducteur aveugle à ℓ jetons (resp. par un transducteur myope à ℓ jetons, resp. par un transducteur à ℓ billes) peut être calculée par un transducteur aveugle à k jetons (resp. par un transducteur myope à k jetons, resp. par un transducteur à k billes) si et seulement si sa sortie est de taille $\mathcal{O}(n^k)$. Les problèmes d'appartenance afférents sont décidables et les constructions sont effectives, ce qui fournit des procédures d'optimisation. De manière assez surprenante, le lien entre la profondeur d'imbrication k et la taille de la sortie n'est plus vrai dès lors que l'on considère des modèles plus puissants que les transducteurs myopes.

Les transducteurs à k billes ont été étendus pour définir le modèle des *transducteurs récursifs à billes*, dans lesquels les appels entre machines peuvent être *récursifs* (la profondeur d'imbrication n'est donc plus bornée par k). Ces objets récursifs peuvent produire des sorties dont la taille est exponentielle en celle de l'entrée. Nous montrons qu'une fonction calculée par un transducteur récursif à billes peut être calculée par un transducteur à k billes pour un certain $k \geqslant 1$ si et seulement si sa sortie est de taille $\mathcal{O}(n^k)$. Le problème d'appartenance est décidable et la construction est effective, ce qui donne un autre résultat d'optimisation : cette procédure supprime la récursivité chaque fois que c'est possible. Les différentes classes de fonctions calculées par ces modèles sont comparées en Figure 11.

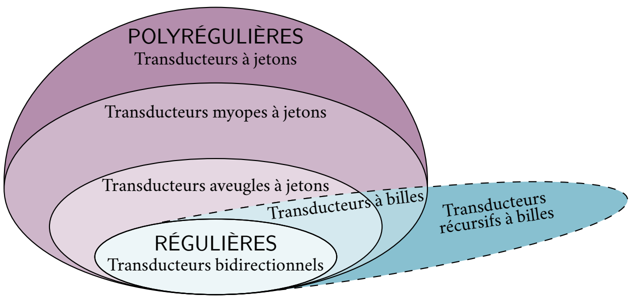


Figure 11: Classes de fonctions calculées par des variantes des transducteurs à jetons.

Dans le cas des transducteurs aveugles à jetons et des transducteurs myopes à jetons, les techniques de preuve pour résoudre les problèmes d'appartenances sont très proches. Elles s'appuient sur des structures algébriques appelées *forêts de factorisation* [Sim90], qui constituent un outil permettant de décomposer le comportement d'un transducteur bidirectionnel en un nombre fini de briques élémentaires. Pour les transducteurs (récursifs ou non) à billes, les techniques de preuve sont assez différentes et reposent sur la correspondance avec les *transducteurs à registres* (cf. paragraphe suivant).

Transducteurs à billes et transducteurs à registres. Nous avons vu dans la partie précédente que la classe des fonctions calculées par les transducteurs bidirectionnels (fonctions régulières) est également

[21] La terminologie originale de [EHS07] est « transducteurs à k jetons invisibles ». En anglais, nous utilisons le terme « last » au lieu de « myope », pour signifier que le transducteur peut uniquement voir le dernier jeton posé.

[22] C'est pour cette raison que le transducteur est « myope » : il ne voit que le jeton le plus récent ($\simeq$ il ne voit pas « loin »).

calculée par les *transducteurs à registres sans copies*, qui sont des machines unidirectionnelles à registres. Dans ce cadre, le terme « sans copie » signifie essentiellement que la valeur d'un registre (qui est un morceau de la sortie finale) ne peut pas être dupliquée au cours d'une exécution.

Lorsque la condition « sans copie » est supprimée, le modèle de transducteurs à registres peut calculer des sorties dont la taille est exponentielle en celle de l'entrée. Nous montrons que ce modèle est équivalent aux *transducteurs récursifs à billes* mentionnés ci-dessus. En outre, pour tout $k \geqslant 1$, nous définissons des conditions sur les copies (moins restrictives que « sans copie ») qui rendent ce modèle équivalent aux *transducteurs à k billes*. Ces résultats créent une nouvelle intuition des transducteurs à billes, en montrant que leur comportement est lié aux *programmes pour flux de données*.

Transducteurs à jetons avec sortie commutative. Nous nous concentrons ensuite sur les *transducteurs à jetons dont la sortie est dans $\mathbb{Z}$ ou dans $\mathbb{N}$* (formellement, le transducteur produit une suite de nombres lors de son exécution, et il renvoie finalement leur somme). Ces machines peuvent être considérées comme des boucles imbriquées qui calculent un nombre. Dans ce contexte, une intuition clef est que l'ordre dans lequel la sortie est produite n'a pas d'importance, en raison de la commutativité. Nous montrons tout d'abord que la classe des fonctions calculées par des transducteurs à jetons avec sortie dans $\mathbb{Z}$ (resp. $\mathbb{N}$) coïncide avec la classe des fonctions calculées par des transducteurs à billes ou par les transducteurs myopes à jetons avec sortie dans $\mathbb{Z}$ (resp. $\mathbb{N}$), ce qui n'était pas le cas en Figure 11. En outre, ces fonctions décrivent une sous-classe naturelle d'une classe célèbre appelée *séries $\mathbb{Z}$-rationnelles*[23] (resp. *séries $\mathbb{N}$-rationnelles*), et le problème d'appartenance associé est décidable.

Nous montrons également que les transducteurs à jetons avec sortie dans $\mathbb{Z}$ ou $\mathbb{N}$ peuvent être optimisés. Dans ce cadre, nous considérons que la « taille » d'un nombre est sa *valeur absolue* et nous montrons qu'une fonction peut être calculée par un transducteur à k jetons si et seulement si sa sortie est en $\mathcal{O}(n^k)$. Ce résultat a déjà été obtenu par Schützenberger [Sch62], mais notre preuve est différente et s'appuie sur les forêts de factorisation. Le cas de $\mathbb{Z}$ est un peu délicat car la présence d'entiers négatifs permet de « supprimer » des parties de la sortie (ce qui n'était pas possible avec les mots).

Transducteurs aveugles avec sortie commutative. Nous observons ensuite que les transducteurs aveugles à jetons (toujours avec sorties dans $\mathbb{Z}$ ou $\mathbb{N}$) sont strictement moins expressifs que les transducteurs à jetons. Nous étudions et résolvons alors le problème d'appartenance associé. En termes d'optimisations de programmes, ce résultat permet de simplifier un programme avec des boucles « pour » en faisant en sorte que les indices de ses boucles imbriquées fonctionnent de manière indépendante. Pour la première fois dans ce manuscrit, il n'est plus possible d'utiliser la taille de la sortie comme *propriété sémantique* pour séparer les classes, puisque qu'elles peuvent toutes deux avoir des sorties de taille polynomiale. Nous introduisons donc une nouvelle propriété appelée *répétitivité* et montrons qu'elle caractérise les fonctions calculables par des transducteurs aveugles parmi les fonctions calculées par des transducteurs à jetons avec sortie dans $\mathbb{Z}$ ou $\mathbb{N}$. Une fois encore, la preuve s'appuie sur les forêts de factorisation.

Modèles canoniques pour les transducteurs à jetons avec sortie dans $\mathbb{Z}$. Étant donné un transducteur à jetons avec sortie dans[24] $\mathbb{Z}$, nous décrivons une procédure effective qui permet de construire un *objet canonique* associé à la fonction qu'il calcule. Cet objet canonique peut être vu comme une forme particulière de transducteur à billes. Nous l'appelons le *transducteur résiduel* de la fonction, et son comportement s'inspire fortement de celui de l'*automate minimal* d'un langage régulier.

[23] De manière générale, les *séries $\mathbb{S}$-rationnelles* sont des fonctions des mots vers un *semi-anneau* $\mathbb{S}$ qui sont calculées par un modèle de transducteurs appelé *$\mathbb{S}$-automates pondérés* (voir [Sak09]). Ici, nous utilisons uniquement $\mathbb{S} := (\mathbb{Z}, +, \times)$ ou $(\mathbb{N}, +, \times)$.

[24] L'auteur n'a pas connaissance d'un moyen d'adapter la construction aux sorties dans $\mathbb{N}$ uniquement.

Pour les langages réguliers (resp. pour les fonctions rationnelles), la construction d'un modèle canonique a été utilisée avec succès pour décider si un langage (resp. une fonction) est *sans étoile*. Nous transférons la notion d'*absence d'étoile* aux fonctions calculées par les transducteurs à jetons avec sortie dans $\mathbb{Z}$ ou $\mathbb{N}$, et fournissons de nombreuses caractérisations des sous-classes de fonctions associées. Enfin, nous montrons comment décider si une fonction calculée par un transducteur à jetons avec une sortie en $\mathbb{Z}$ est sans étoile. La preuve repose sur une condition sémantique de *lissité*, qui se traduit sous forme d'une propriété syntaxique (décidable) du transducteur résiduel que nous appelons *apériodicité* (adaptée de la notion d'*apériodicité* mentionnée plus haut pour les automates).

Fonctions régulières déterministes de mots infinis. Les autres résultats de ce manuscrit concernent les transducteurs de mots infinis. Les questions étant plus complexes dans ce cadre, nous ne traitons pas de machines imbriquées et nous nous concentrons sur les *transducteurs bidirectionnels de mots infinis*.

Nous définissons la classe des *fonctions régulières déterministes de mots infinis* comme la classe des fonctions calculées par des transducteurs déterministes bidirectionnels de mots infinis. De manière surprenante, cette classe n'a pas été étudiée dans la littérature, contrairement aux *fonctions régulières de mots infinis* qui sont obtenues en ajoutant des *ω-anticipations* aux transducteurs bidirectionnels. Même si les fonctions régulières déterministes sont plus faibles que les fonctions régulières (cf. Figure 12), elles sont plus pertinentes en pratique puisque toute fonction régulière déterministe s'avère *calculable*.

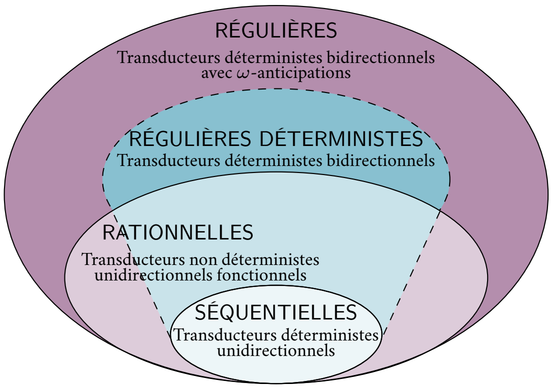


Figure 12: Classes de fonctions de mots infinis.

Nous étudions les principales propriétés des fonctions régulières déterministes et montrons qu'elles forment une classe robuste et naturelle de fonctions de mots infinis, close notamment par composition de fonctions. En outre, nous introduisons deux modèles alternatifs qui capturent cette classe :

- les *transducteurs à registres (sans copie) de mots infinis* (définis en adaptant les transducteurs à registres de mots finis). Ils peuvent être vus comme des *programmes pour flux de données*;
- les *transducteurs déterministes bidirectionnels* avec une version affaiblie des *ω-anticipations* appelées *anticipations finies*. Ce modèle est principalement utilisé comme un outil dans les preuves.

Une caractérisation en termes de compositions de fonctions de base est également présentée[25]. Des résultats similaires sont déjà connus pour les fonctions régulières de mots finis ou infinis, mais les preuves dans le cas des fonctions régulières déterministes doivent surmonter des difficultés supplémentaires.

[25] Une description en termes de logique est aussi obtenue dans [CDFW23] mais nous ne la présentons pas dans ce manuscrit.

Déterminisation des fonctions rationnelles de mots infinis. Depuis [DFKL20], il est conjecturé que la classe des fonctions régulières déterministes est exactement[26] la classe des fonctions régulières qui sont *calculables*/*continues*. Nous apportons une réponse partielle à cette conjecture en montrant que toute fonction rationnelle de mots infinis qui est calculable/continue est en fait régulière déterministe. L'auteur considère ce résultat difficile comme l'un des joyaux de ce manuscrit. Puisque la continuité est décidable, nous savons donc décider le problème d'appartenance des fonctions rationnelles aux fonctions déterministes régulières. En pratique, ce résultat permet donc de construire un programme déterministe à mémoire bornée qui calcule une fonction rationnelle, dès qu'il en existe un.

Plan chapitre par chapitre

Dans le Chapitre 1, nous rappelons les définitions et les propriétés principales de plusieurs modèles de transducteurs de mots finis. Le Chapitre 2 fournit une boîte à outils pour l'étude des transducteurs bidirectionnels, qui est utilisée dans les Chapitres 3, 5 et 6. Dans le Chapitre 3, nous décrivons les variantes des transducteurs à jetons appelées *transducteurs aveugles à jetons* et *transducteurs myopes à jetons*, et nous montrons comment les optimiser. Dans le Chapitre 4, nous présentons les *transducteurs à billes*, nous les mettons en relation avec les *transducteurs à registres* et nous montrons comment les optimiser.

Le Chapitre 5 s'intéresse aux *transducteurs à jetons dont la sortie est dans* $\mathbb{Z}$ *ou* $\mathbb{N}$ et montre comment ils peuvent être optimisés. De plus, il relie les fonctions calculées par ces machines aux célèbres classes de *séries* $\mathbb{Z}$*-* et $\mathbb{N}$*-rationnelles*. Dans le Chapitre 6, nous décidons si une fonction calculée par un transducteur à jetons avec sortie dans $\mathbb{Z}$ ou $\mathbb{N}$ peut être calculée par un transducteur aveugle. Le Chapitre 7 décrit comment transformer un transducteur à jetons avec sortie dans $\mathbb{Z}$ en un *objet canonique*. Ce résultat est utilisé pour décider si une fonction calculée par ce modèle est *sans étoile*.

Dans le Chapitre 8, nous rappelons les définitions et les propriétés principales de plusieurs modèles de transducteurs de mots infinis. Le Chapitre 9 étudie les propriétés des *fonctions régulières déterministes de mots infinis*, et fournit de nombreuses caractérisations de cette classe. Dans le Chapitre 10, nous montrons qu'une *fonction rationnelle de mots infinis* est *régulière déterministe* si et seulement si elle est continue, ce qui permet de résoudre un dernier problème d'appartenance.

[26] Ici encore, une formulation formelle de cette conjecture devrait parler d'extensions de fonctions.

Introduction

C'est que j'avais besoin de vous pour un mystère
Que je veux pénétrer.

Camille Saint-Saëns, L. Détroyat, A. Silvestre, *Henri VIII*

Program optimization and class membership problems

Program optimization consists in modifying the *syntax* (implementation) of a program in order to make it more efficient with respect to some metrics, while preserving its *semantics* (behavior). In practice, being "more efficient" often means that the modified program uses fewer ressources, e.g. in terms of execution time or memory. For instance, a sorting algorithm whose asymptotic time complexity is $\mathcal{O}(n \log(n))$ over inputs of size n can be seen as an optimization of a $\mathcal{O}(n^2)$ sorting algorithm[1].

Optimization in practice. Making programs as efficient as possible is essential in practice. On the one hand, optimizing the asymptotic time or memory consumption is necessary to make programs scale over large inputs. An extreme case of large inputs comes along with *streaming programs*, which have to treat an unbounded sequence of items by doing a single pass in a nearly *real-time* fashion[2]. On the other hand, one can look for programs which are efficient over small inputs[3], in which case optimizing the exact resources consumption is more relevant than studying their asymptotic behavior.

Optimization can be performed at several levels of abstraction, from the *algorithmic* point of view (designing algorithms and data structures whose resources consumption is optimal) to the *machine code* level (e.g. optimizing the assembly code to make it as efficient as possible with respect to a given computer architecture). However, this process tends to complexify the code, and thus makes it harder to maintain or debug. It is therefore relevant to perform optimization at the end of the development stage, as emphasized long ago by Knuth: "premature optimization is the root of all evil" [Knu74]. In addition to these difficulties, *handmade program optimization* may be a long (since it requires to think) and risky (since it may introduce bugs if the semantics is not exactly preserved) task for the programmer.

Automated program optimization. The previous paragraph advocates for doing *automated program optimization*. This task consists in designing a meta-program which takes a program as input and automatically produces an optimized program having the same semantics. It can therefore be seen as a particular form of *automated program synthesis*[4], where the input specification is already a program.

[1] This is for instance the case of *merge sort* with respect to *insertion sort*.

[2] Informally, the program is only allowed to use limited processing time and memory per item.

[3] A simple way to optimize the time complexity of a program over small inputs is to pre-compute and hardcode the output of any input whose size is smaller than a given bound. However, this construction heavily increases the size of the program itself.

[4] *Program synthesis* consists in constructing a program which satisfies a given formal specification.

Numerous automated optimizations (in particular those for low-level code) have been implemented for long in compilers and processors[5]. However, these optimizations generally do not produce an *optimal* code (in the sense that no "better" code would exist): they rather follow *heuristic* methods for improving resource usage in some known cases, which already provides impressive results in practice [Leu00].

The practical purpose of this manuscript is describe optimization procedures overs classes of *simple programs*, which ensure that the program produced is always optimal[6]. Obtaining such results presupposes that a metrics has been chosen to compare the efficiency of programs and therefore define what "optimal" means. Furthermore, considering only "simple" programs is unavoidable since classical undecidability statements make it hopeless to perfectly optimize programs in general.

From program optimization to class membership problems. From a fundamental perspective, the goal of this manuscript is to study *class membership problems*. Let us consider a class P of programs (e.g. programs whose execution time is polynomial in the input size n) and a subclass $\mathsf{P}' \subseteq \mathsf{P}$ of target programs which are considered as "efficient" (e.g. programs whose execution time is $\mathcal{O}(n)$). In this setting, the class membership problem from P to P' is formally defined as follows:

- **Input:** a program $\pi \in \mathsf{P}$ whose semantics is a function f;
- **Question:** is there a program $\pi' \in \mathsf{P}'$ whose semantics is f?

In other words, this problem asks whether a function from the "big" class $\mathcal{C}$ of Figure 1 belongs to the "small" class $\mathcal{C}'$. Observe that it only deals with semantics of programs and not with their syntax.

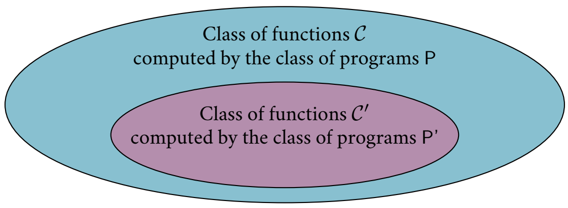


Figure 1: Global picture of a class membership problem form P to P'.

Solving[7] the class membership problem means building an algorithm which automatically answers the **question** when given a program $\pi \in \mathsf{P}$ as **input**. Such an algorithm can nearly be seen as an optimization procedure, with the difference that it only says if an optimized program of $\pi' \in \mathsf{P}'$ exists, but it is not required to explicitly build it. However, for all the class membership problems which are solved in this manuscript, the proof turns out to be *effective* and enables to build an optimized program.

Finite automata

When dealing with "simple" programs, we shall in fact consider *finite-state machines*. Formally, such a machine is a model of computation which has a finite number of internal *states*. At any given time of its computation, it is in exactly one state, and it performs a *transition* from one state to another in response to some input. In other words, it is an abstract description for a program whose working memory has *bounded* size, i.e. which does not depend on the size of the input. Finite state-machines are implemented

[5] Such optimizations are susceptible to modify the way the processor manipulates its memory, and therefore to put unexpected information at unexpected places. Unsurprisingly, this feature creates security flaws which can be exploited to get access to unauthorized information, such as the recent CVE-2022-40982 "downfall" vulnerability on Intel processors.

[6] In this case, optimization is sometimes called *superoptimization* [Mas87], but we shall not follow this terminology.

[7] Formally, one should say "showing that the class membership problem is *decidable*".

in many devices which perform a pre-determined sequence of actions depending on a sequence of events, such as vending machines or programmable logic controllers in industry.

Finite automata and regular languages. *Finite deterministic automata* are a particular class of finite-state machines whose input consists of a *word* (which is a sequence of characters from a finite set) and whose output is either "yes" or "no". The input word is processed by the automaton in a streaming fashion (as depicted in Figure 2) and it performs a transition each time it reads a new character. This model has been used for a large variety of applications in computer science (including streaming algorithms, text algorithms, formal verification via model checking, control theory, network protocols, design of hardware systems, etc.) and in related areas such as formal linguistics or computational biology.

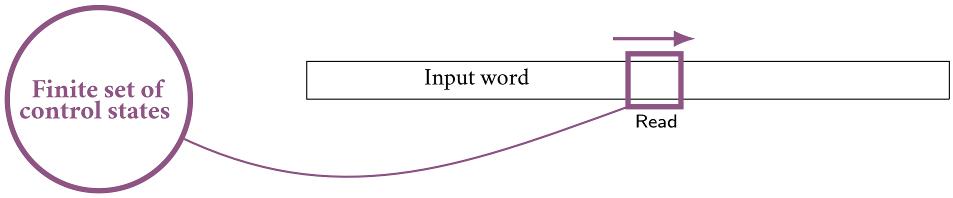


Figure 2: Behavior of a finite one-way deterministic automaton.

The set of input words for which the automaton answers "yes" is called the *language* computed by the automaton. The class of languages computed by finite automata is well-known under the name of *regular languages*, which are considered as one of the cornerstones of theoretical computer science. They enjoy several other descriptions in terms of expressions (*regular expressions* [Kle56]), logics (monadic second-order logic [Büc60, Elg61, Tra62]) or algebra (monoids and congruences [Ner58]).

Minimal automaton. When considering optimization problems for finite automata, a first natural challenge is to optimize the memory used by the machine. More explicitly, given an automaton, we intend to build another automaton with *minimal* number of states which computes the same language. This problem has been solved for long with e.g. Moore [Moo56] or Hopcroft [Hop71] algorithms.

It turns out that given a regular language, there exists a unique deterministic automaton with minimal number of states which computes it. This object is called the *minimal automaton* of the language. Therefore the aforementioned minimization algorithms not only reduce the number of states, but they also build a *canonical model* associated to a given regular language (in the sense that it only depends on the language, but not on the automaton that was given as input). Generally speaking, the construction of canonical models is especially relevant for solving class membership problems, since such objects often reveal informations about the semantics. Furthermore, it provides a procedure for deciding if two machines have the same semantics (by canonizing both of them and comparing the results).

Subclass membership problems for regular languages. Another prominent question in automata theory is to understand the subclasses of regular languages defined by restricting certain of their equivalent definitions (automata, expressions, logic or algebra). Naturally, "understanding a class" is a purely informal goal, but one standard way to do so is to solve the appropriate class membership problems. Indeed, the techniques developed in this setting generally provide deep insights on the subclasses.

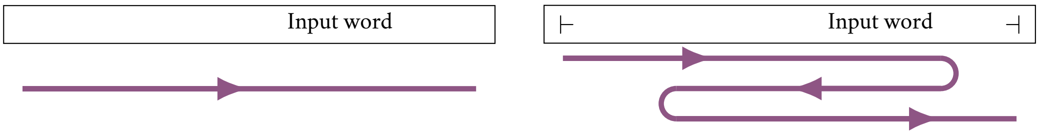
This approach was initiated by Schützenberger [Sch65], who provided a membership procedure for the class of *star-free languages* (a subclass of regular languages described by[8] *star-free expressions*, which

[8] This subclass also enjoys characterizations in terms of automata, logics and algebra.

correspond to a restriction of regular expressions). It turns out that a regular language is star-free if and only if its minimal automaton enjoys a *syntactic property* called *aperiodicity*[9]. An effective membership procedure follows since this property can be decided. In subsequent literature, the strategy of solving membership problems by looking at decidable syntactic properties of the minimal automaton has provided membership procedures for numerous other subclasses [Str94]. This line of research is still active nowadays since several subclass membership problems remain open (see e.g. [Pin17]).

Beyond finite automata. In a general fashion, adding simple features to finite deterministic automata does not increase their expressive power. Let us highlight the following equivalent extensions:

- *non-deterministic automata*, which are able to make "guesses" along a computation, and later check their validity. Transforming such an automaton into a deterministic one is classical exercice;
- *two-way* (deterministic or non-deterministic) *automata*, which can perform right and left moves on their input, while the model mentioned so far (that we shall from now on call *one-way automata*) is only able to move right (compare Figures 3a and 3b). Equivalence follows from [She59];



(a) Computation of a one-way automaton.

(b) Computation of a two-way automaton.

Figure 3: Behavior of one-way and two-way automata.

- *nested* (one-way or two-way) (deterministic or non-deterministic) *automata*, which are able to call auxiliary automata during their computation. In this manuscript, we shall mention in more detail the particular nested model of *pebble automata* which was introduced in [EH99].

Otherwise stated, all reasonable variants of finite automata can compute no more regular languages, which hints once more that this class is especially robust and fundamental. A notable exception is the use of a *stack* as an auxiliary feature, which radically increases the expressive power of finite automata. One-way non-deterministic automata with stack are called *pushdown automata* and compute the celebrated class of *context-free languages*. However, it is a classical exercice (see e.g. [HMU07]) to show that the class membership problem from context-free languages to regular languages is undecidable[10].

Finite transducers

This manuscript focuses of *finite transducers*, which are finite automata enhanced with outputs. More formally, such a finite-state machine is defined by starting from a finite automaton model and adding outputs on the transitions. The machine finally returns the concatenation of the outputs produced along its transitions, therefore it computes a *function* (when deterministic) or a *relation* (when non-deterministic) from words to words. Transducers are very useful in a lot of areas like compiling [FCL10, Chapter 3], natural language processing [MPR08] or computer arithmetic. Furthermore, they provide a more comprehensive environnement than finite automata for modeling streaming programs.

Expressive power of transducers. By starting from the according models of finite automata, it is possible to define a variety of transducer models which are either one-way or two-way, deterministic or not, nested or not, etc. The behavior of a two-way deterministic transducer is e.g. depicted in Figure 4.

[9] The latter is also called *counter-freeness* in the literature [MP71].

[10] The problem is however decidable when starting from the subclass of *deterministic context-free languages* [Ste67].

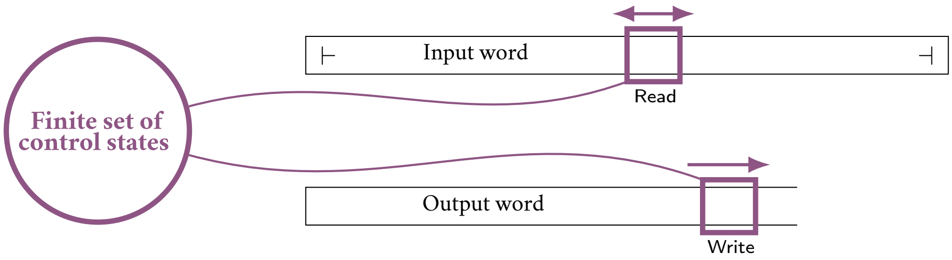


Figure 4: Behavior of a two-way deterministic transducer.

Contrary to the case of automata, these various transducer models do not have in general the same expressive power. As a consequence, the theory of functions computed by transducers tends to be more challenging than the study of languages computed by automata, following an early remark of Scott: "the functions computed by the various machines are more important - or at least more basic - than the sets accepted by these devices" [Sco67, Section 5]. Informally, the following striking phenomena occur:

- *non-deterministic transducers* are more expressive than *deterministic transducers*. A trivial reason for this phenomenon is that non-deterministic transducers compute *relations* whereas deterministic transducers can only compute functions. More interestingly, even *functional* (i.e. which compute functions) non-deterministic transducers tend to be more expressive than the deterministic ones;
- *two-way transducers* are more expressive than *one-way transducers*. This comes from the fact that two-way transducers are able to return reversed (portions of) their input, by reading it from right to left, while (even non-deterministic) one-way machines are forced to read it from left to right;
- *nested transducers* are more expressive than non-nested transducers. Intuitively, the argument is that nested transducers can mimic nested "for" loops and therefore produce outputs whose size is polynomial in the input size, while non-nested transducers only produce outputs of linear size.

Celebrated transducer models. Recent literature focuses on the following prominent models:

- *one-way deterministic transducers*, which compute the class of *sequential functions*;
- *functional one-way non-deterministic transducers*, which compute the class of *rational functions*;
- *two-way deterministic transducers*, which compute the class of *regular functions*;
- *pebble transducers* (= *nested two-way transducers*) which compute the class of *polyregular functions*.

These various classes of functions are depicted in Figure 5, where all inclusions are (implicitly) strict.

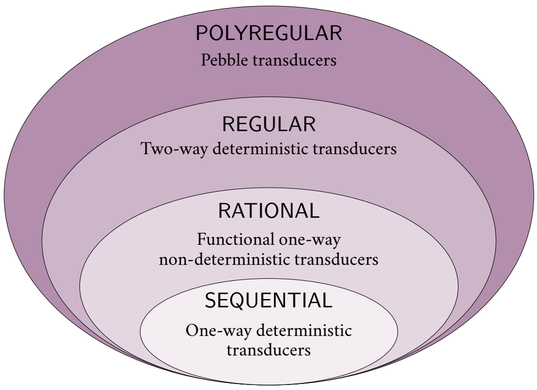


Figure 5: Classes of functions computed by transducers of finite words.

Two-way transducers and regular functions. The specific class of *regular functions* has been considered for long as the most natural counterpart[11] of regular languages. It has been studied for its properties such as closure under function composition [CJ77] or decidability of the equivalence problem [Gur80]. Equivalent descriptions have been given in terms of expressions (regular-like expressions [AFR14, DGK18, BDK18, BR18] or composition of basic functions [BS20]) or logics [EH01, DFL18].

A substantially different model called *copyless streaming string transducers* was also shown to compute exactly the class of regular functions [AC10]. Such a machine consists of a one-way deterministic automaton enriched with registers which store portions of the output and are updated at each transition. This model is at the same time simpler (it reads the input only once) and more complex (it uses registers) than a two-way transducer. It is especially relevant as a model of *streaming programs*.

Pebble transducers and polyregular functions. The transducer model called *pebble transducer* is built by nesting two-way deterministic transducers [MSV00, EM02, Boj18]. Informally, a 1-*pebble transducer* is simply a two-way transducer. A 2-*pebble transducer* consists of a two-way transducer which, when on any position of its input word, can call auxiliary two-way transducers. The latter take as input the original input word in which the position of the call is marked (we say that a *pebble* is *dropped* in this position). The main transducer finally returns the concatenation of all the outputs returned by its auxiliary calls. More generally, a k-*pebble transducer* for $k \geqslant 1$ consists of nested two-way transducers with nesting depth k. A partial computation of a 3-pebble transducer is depicted in Figure 6.

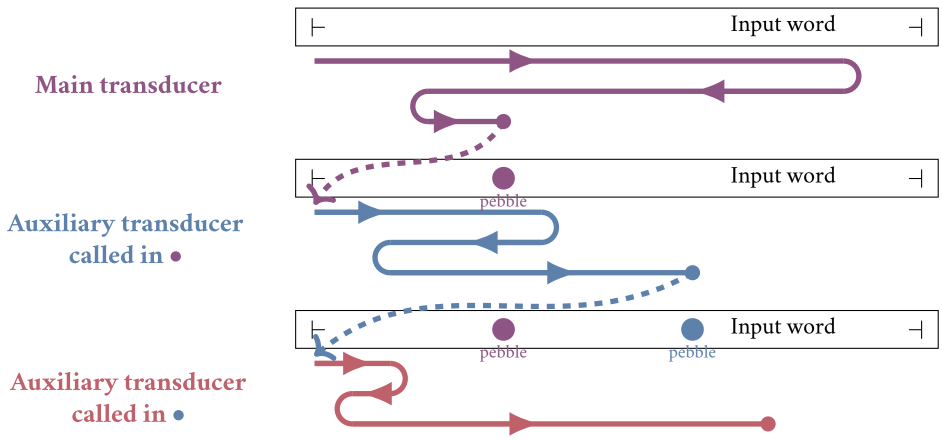


Figure 6: Computation of a 3-pebble transducer.

A k-pebble transducer can also be seen as a program which executes nested (two-way) "for loops". In this setting, the position of the i-th pebble for $1 \leqslant i \leqslant k$ corresponds to the index of a i-th nested loop. From this perspective, it is easy to see that a k-pebble transducer can produce an output whose size is polynomial in the input length n, more precisely in $\mathcal{O}(n^k)$ since it has k nested loops.

As mentioned above, the class of *polyregular functions* is defined as the class of functions computed by pebble transducers. Several properties such as closure under function composition [EM02] are known for long. A recent regain of interest for polyregular functions has followed from Bojańczyk's extensive study [Boj18]. Several equivalent descriptions of this class have been given, in terms of expressions (with composition of basic functions [Boj18]) and logics [BKL19]. Other equivalent formalisms have been introduced, among them an imperative programming language named *for transducers*, a functional programming language in the spirit of λ-calculus, and a specific type system [Boj18, Boj23a].

[11] This notion is highly informal, and the other classes are also very natural counterparts of regular languages.

Class membership problems for finite transducers

The various classes of functions computed by finite transducers specify various membership problems which do not exist in the case of finite automata (since all automata models are equivalent). Solutions to certain of these problems are available in the literature, with very disparate proof techniques.

The quest for canonical models. As mentioned for finite automata, a very natural approach for solving class membership problems is to describe a procedure for transforming any machine into a *canonical* one, i.e. which only depends on the semantics of the machine. In the case of transducers, it is known how to build canonical objects for sequential and rational functions [RS91, Cho03, FGL19].

These canonical models have successfully been used to decide whether a rational function is *star-free* rational [FGL19] (*star-freeness* being defined here as an analogue of the eponymous notion for regular languages). Furthermore, they provide a way to decide the class membership problem from rational functions to sequential functions. Historically, this last problem was in fact shown decidable in various papers without using canonical models [Cho77, WK95, BCPS03]: the classical proof consists in showing that any (and not only a canonical one) one-way non-deterministic transducer which computes a sequential function verifies a (decidable) *syntactic property* which is often called *twinning property*.

Optimization of two-way transducers. When it comes to two-way transducers and regular functions, the construction of a canonical model is unfortunately not known in general[12]. As a consequence, deciding intrinsic properties of such functions is believed to be difficult, since they can be described in several (seemingly) unrelated manners. In particular, deciding *star-freeness* of regular functions (which is once again defined by analogy with star-free regular languages) is an open problem.

The class membership problem from regular functions to rational functions was nevertheless shown decidable [FGRS13, BGMP18]. The proofs of this result rely on a rather combinatorial study of the behavior of two-way transducers. Once more, this result can be considered as a program optimization procedure since it builds a one-way (= more efficient) transducer whenever it exists.

Optimization of pebble transducers. A very natural question is to decide for $1 \leqslant k \leqslant \ell$ whether a function given by an ℓ-pebble transducer can be computed by a k-pebble transducer. This problem is especially relevant in terms of optimization since it asks whether a program with ℓ nested loops (i.e. whose execution time is $\mathcal{O}(n^\ell)$ on inputs of size n) can be transformed into a program with k nested loops only. Equivalently, it asks whether the nesting depth of nested functions can be minimized.

As mentioned in the previous section, a k-pebble transducer produces an output whose size is $\mathcal{O}(n^k)$ when n is the input size. It is therefore natural to conjecture that a function given by an ℓ-pebble transducer can be computed by a k-pebble transducer if and only if its output is $\mathcal{O}(n^k)$. This result holds for $k = 1$ and it was used to solve the membership problem from ℓ-pebble transducers for any $\ell \geqslant 1$ to 1-pebble transducers[13] [Boj22]. However, the general conjecture does not hold: for all $k \geqslant 3$ there exists a function whose output is $\mathcal{O}(n^2)$ but which cannot be computed by a transducer having less than k pebbles [Boj22, Boj23b]. The related class membership problems are open.

Transducers of infinite words and computability. Automata over *infinite words* (= infinite sequences of characters) have been studied since the early days of automata theory, following the seminal work of

[12] It is however known how to build a canonical model in the case of *regular functions* with *origin semantics* (see [Boj14]). Partial results are also known for *sweeping transducers* (which can only change direction at the borders of the input), see e.g. [LLN+11].

[13] Since 1-pebble transducers are simply two-way transducers, it decides whether a polyregular function is regular.

Büchi [Büc62]. They are roughly defined as automata over finite words, but perform infinite computations in order to read their whole input. Such machines compute a celebrated analogue of regular languages over infinite words, which is called *ω-regular languages* (see e.g. [PP04] for an introduction). The reader may argue that inputs are rarely infinite in real life. This is indeed the case, but infinite words can be understood as an abstraction of *arbitrarily long inputs* processed by *streaming programs*.

Various transducer models with infinite input and output have been studied in the literature. The most celebrated of these are defined by analogy with transducers of finite words:

- *one-way deterministic transducers*, which compute the class of *sequential functions of infinite words*;
- *one-way non-deterministic transducers*, which compute the class of *rational functions of infinite words*;
- *two-way deterministic transducers* with an extra feature called *ω-lookarounds*, which compute *regular functions of infinite words* [AFT12]. Informally, an ω-lookaround enables the machine to check an "infinite" property of its input, such as "does the character 0 occurs infinitely many times?".

These robust classes enjoy various characterizations and algorithmic properties. Furthermore, the class membership problem from rational functions to sequential functions is decidable [BC04]. The three aforementioned classes are depicted in Figure 7 (where all inclusions are implicitly strict).

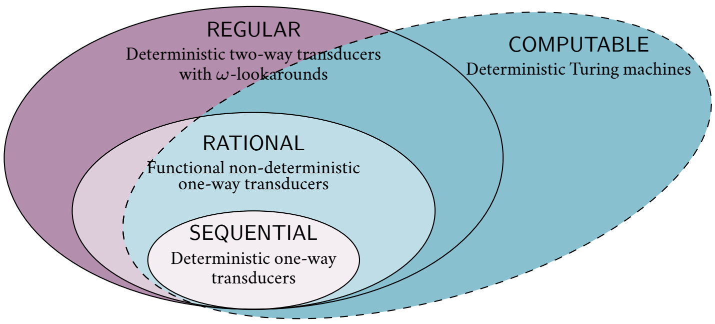


Figure 7: Classes of functions over infinite words.

However, rational and regular functions of infinite words both suffer from a severe downside when it comes to effective implementations, which is a major difference with the case of finite words. Indeed, the reader should be convinced that all the aforementioned transductions of finite words are *computable*, in the sense that they can be written in any programming language, or equivalently computed by a *deterministic Turing machine*. This is no longer the case here: the use of *ω-lookarounds* or non-determinism along infinite computations makes it possible to detect e.g. if the input contains infinitely many times a given character. Nevertheless, such a property cannot be verified by a deterministic device.

As a consequence, a prominent class membership problem for practical applications is to decide whether a regular function of infinite words is computable or not. This question was recently solved and a procedure was provided for building an equivalent program (deterministic Turing machine) whenever it exists [DFKL20]. Interestingly, the regular functions of infinite words which are computable are semantically characterized[14] as the regular functions which are *continuous* for some topology.

[14] Formally, all these results are only true up to considering *extensions* of the function domains, but we deliberately omit this precision in an informal introductive approach.

Contributions of this manuscript

This manuscript explores most of the results from the seven papers published by the author during his PhD thesis [DFG20, Dou21, Dou22, CD22, Dou23, CDL23, CDFW23]. Several improvements and clarifications are proposed with respect to the original statements. Furthermore, the results are presented through a unified formalism. More concretely, the contributions of this manuscript are twofold:

- we solve various *class membership problems* for transductions of finite and infinite words. All the questions deal with transducer models which already exist in the literature[15] and the solutions given are non-trivial. Each time, the membership procedure turns out to be *effective* (in the sense that it builds a "simpler" transducer whenever it exists) and it can therefore be considered as a *program optimization* procedure. These results are summarized in Table 9;
- we provide new *computation models* and *characterizations* for capturing pre-existing classes of transductions. These results complete the previous understanding of these classes by providing new insights on their expressive power. Furthermore, having various representations which highlight different properties of the same object is helpful for solving class membership problems.

Techniques for solving class membership problems. Apart from the final results in themselves, the author believes that the proof techniques developed in this manuscript for deciding membership problems are also valuable. Indeed, when solving the membership problem from a given class of transducers P to a subclass P', we shall follow a generic high-level proof strategy. This strategy consists in looking for *semantic* and *syntactic* characterizations of the subclass, as described in Meta-theorem 8.

Meta-theorem 8 (Class membership problem $\mathsf{P} \to \mathsf{P}'$)

Let f be a function computed by a transducer $\mathscr{T}$ of the class P. The following are equivalent:

(1) f can be computed by a transducer of the subclass P';
(2) f verifies some *semantic property* $(\mathcal{F})$;
(3) $\mathscr{T}$ verifies some *syntactic property* $(\mathcal{T})$.

Furthermore $(\mathcal{T})$ is decidable and the construction Item (3) $\Rightarrow$ Item (1) is effective.

Meta-proof of Meta-theorem 8. Item (1) $\Rightarrow$ Item (2) is in general rather easy once $(\mathcal{F})$ has been chosen. For Item (2) $\Rightarrow$ Item (3), we rely on combinatorial "pumping" arguments. Item (3) $\Rightarrow$ Item (1) is the actual optimization procedure and the most difficult part of the proof. ◀

Formally, the decidability of the membership problem from P to P' follows from the fact that the property $(\mathcal{T})$ is decidable. Furthermore, the semantic property $(\mathcal{F})$ is not only a tool in the proof, but also useful for showing by hand that a given function f is computable or not by a transducer from P'. The various semantic and syntactic properties used in this manuscript are summarized in Table 9.

Membership problem	Semantic property	Syntactic property	Statement
Blind ℓ-pebble transducer $\downarrow$ Blind k-pebble transducer	Output size in $\mathcal{O}(n^k)$	Pumpability of any transducer (Definition 3.17)	Theorem 3.12
Last ℓ-pebble transducer $\downarrow$ Last k-pebble transducer	Output size in $\mathcal{O}(n^k)$	Pumpability of any transducer (Definition 3.25)	Theorem 3.13

[15] In other words, we do not introduce new artificial models in order to solve artificial class membership problems.

ℓ-marble transducer $\downarrow$ k-marble transducer	Output size in $\mathcal{O}(n^k)$	Barbells in any transducer (Lemma 4.47)	Theorem 4.11
Recursive marble transducer $\downarrow$ k-marble transducer	Output size in $\mathcal{O}(n^k)$	Heavy cycles in any transducer (Lemma 4.47)	Theorem 4.12
ℓ-pebble transducer with output in $\mathbb{Z}$ or $\mathbb{N}$ $\downarrow$ k-pebble transducer with output in $\mathbb{Z}$ or $\mathbb{N}$	Output size in $\mathcal{O}(n^k)$	Pumpability of any transducer (Definition 5.50)	Theorem 5.25
Pebble transducer with output in $\mathbb{Z}$ or $\mathbb{N}$ $\downarrow$ Blind pebble transducer with output in $\mathbb{Z}$ or $\mathbb{N}$	Repetitiveness (Definition 6.13)	Permutability of any transducer (Definition 6.28)	Theorem 6.17
Pebble transducer with output in $\mathbb{Z}$ $\downarrow$ Aperiodic pebble transducer with output in $\mathbb{Z}$	Smoothness (Definition 7.15)	Aperiodicity of a *canonical* transducer (Definition 7.50)	Theorem 7.19
One-way non-deterministic transducer of infinite words $\downarrow$ Two-way deterministic transducer of infinite words	Continuity (Proposition-Definition 8.41)	Twinning property of any transducer (Lemma 10.8)	Theorem 10.1

Table 9: Main class membership problems solved in this manuscript.

Observe that Meta-theorem 8 does not deal with a *canonical object* associated to the function f: the property $(\mathcal{T})$ is applicable to any transducer of P. This way, we circumvent the difficulties which are inherent to the constructions of canonical models, at the cost of doing somehow combinatorial proofs. We shall nevertheless rely on a canonical object for showing the penultimate line of Table 9.

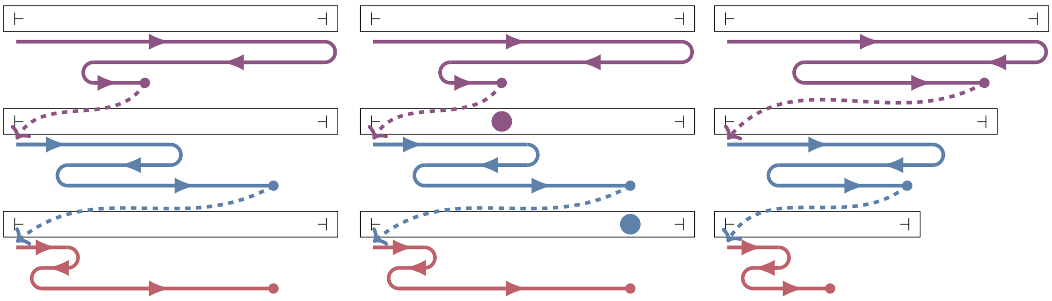
Optimization for variants of pebble transducers. The first results of this manuscript deal with variants of pebble transducers. Recall from the previous section that for $1 \leqslant k \leqslant \ell$, the functions computed by k-pebble transducers do not coincide in general with the functions computed by ℓ-pebble transducers whose output size is $\mathcal{O}(n^k)$. Furthermore, the related class membership problems are open.

In order to provide optimization results while overcoming this issue, we focus on three pre-existing subclasses of k-pebble transducers, which are defined by weakening the nesting behavior :

- *blind k-pebble transducers* from [NNP21][16], which are k-pebble transducers where an auxiliary transducer cannot see[17] the pebbles marking the nested calls done by its ancestors. In other words, it corresponds to nested functions which do not provide the current position as an argument when doing a nested call. This behavior is depicted in Figure 10a (to be compared with Figure 6);

[16] The original terminology of [NNP21] is *comparison-free k-pebble transducers*.
[17] That is why the machine is said to be "blind".

- ▶ *last k-pebble transducers* from [EHS07][18], which are k-pebble transducers where an auxiliary transducer can only see the pebble dropped by its parent, but no the full history of the former pebbles. This behavior is depicted in Figure 10b (the purple pebble disappears in the third input);
- ▶ *k-marble transducers* from [EHV99], which are k-pebble transducers where the input of an auxiliary transducer is only the prefix of the original input which ends in the calling position. Hence the size of the input decreases at each nested call. This behavior is depicted in Figure 10c. Observe that k-marble transducers can be seen as a restriction of last k-pebble transducers.



(a) Blind 3-pebble transducer. (b) Last 3-pebble transducer. (c) 3-marble transducer.

Figure 10: Behavior of variants of pebble transducers.

We show that for all $1 \leqslant k \leqslant \ell$, a function computed by a blind ℓ-pebble transducer (resp. by a last ℓ-pebble transducer, resp. by a ℓ-marble transducer) can be computed by a blind k-pebble transducer (resp. by a last k-pebble transducer, resp. by a k-marble transducer) if and only if its output has size $\mathcal{O}(n^k)$. The membership problems are decidable and the constructions are effective, which yields optimization procedures. Surprisingly enough, the characterization of the minimal nesting depth by the size of the output is *tight* for last pebble transducers, in the sense that it fails for more powerful models.

Marble transducers have been extended to the model of *recursive marble transducers*, in which the nested calls are allowed to be *recursive* (hence the nesting depth is no longer bounded). Such recursive machines can produce outputs whose size is exponential in the input. We show that a function computed by a recursive marble transducer can be computed by a k-marble transducer for some $k \geqslant 1$ if and only if its output has size $\mathcal{O}(n^k)$. The membership problem is decidable and the conversion is effective, which yields another optimization result: this procedure removes recursion whenever it is possible. The various classes of functions computed by the aforementioned models are compared in Figure 11.

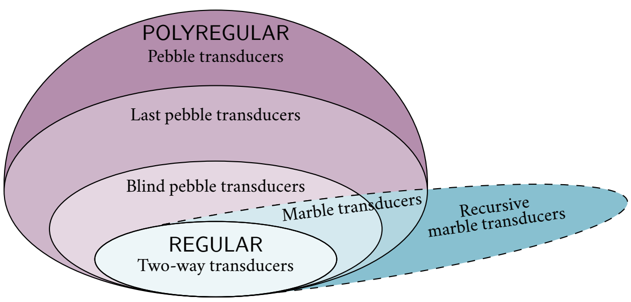


Figure 11: Classes of functions computed by variants of pebble transducers.

[18]The original terminology of [EHS07] is *invisible k-pebble transducers*.

The proof techniques for solving the class membership problems for blind pebble transducers and last pebble transducers are very close. They both rely on algebraic structures called *factorization forests* [Sim90], which are a versatile tool for decomposing the behavior of a two-way transducer into a finite number of elementary patterns. For (recursive) marble transducers, the proof techniques are rather different and rely on the correspondence with *streaming string transducers* (cf. next paragraph).

Marble transducers and streaming string transducers. Recall that the class of functions computed by two-way transducers (regular functions) is also captured by *copyless streaming string transducers*, which are one-way transducers with registers. In this setting, the term "copyless" roughly means that the value of a register (which is a portion of the final output) cannot be duplicated during a computation.

When dropping the copyless restriction of *streaming string transducers*, one is able to compute functions whose output size is exponential in the input size. We show that this model is equivalent to the aforementioned *recursive marble transducers*. Furthermore, for all $k \geqslant 1$ we devise copy restrictions (weaker than copylessness) which make it equivalent to *k-marble transducers*. These results shed a new light on the marble transducer model, by showing that its behavior is related to *streaming algorithms*.

Pebble transducers with commutative output. We then focus on *pebble transducers whose output lies in $\mathbb{Z}$ or $\mathbb{N}$* (formally, the transducer produces integers along its computation and finally returns the sum of these integers). Such machines can be understood as nested loops which compute an integer. In this setting, a key intuition is that the order in which the output is produced has no importance, due to commutativity. We first show that the class of functions computed by pebble transducers with output in $\mathbb{Z}$ (resp. $\mathbb{N}$) coincides with the class of functions computed by marble transducers or last pebble transducers with output in $\mathbb{Z}$ (resp. $\mathbb{N}$), which does not hold when the outputs are words (cf. Figure 11). Furthermore, these functions describe a natural subclass of a celebrated class called *$\mathbb{Z}$-rational series*[19] (resp. *$\mathbb{N}$-rational series*) and that the according class membership problem is decidable.

We additionally provide an optimization result for pebble transducers with output in $\mathbb{Z}$ or $\mathbb{N}$. In this setting, we consider that the "size" of an integer is its *absolute value* and show that a function can be computed by a k-pebble transducer whenever its output size is in $\mathcal{O}(n^k)$. This result roughly reformulates a statement of Schützenberger [Sch62], with a different proof which relies on factorization forests. The case of $\mathbb{Z}$ is a bit tricky since the presence of negative integers enables to "remove" portions of the output (which was not possible with words) and thus it can make the output size lower than expected.

Blind pebble transducers with commutative output. Interestingly enough, blind pebble transducers (still with output in $\mathbb{Z}$ or $\mathbb{N}$) turn out to be strictly weaker than pebble/last pebble/marble transducers. Thus we investigate and solve the according class membership problem. In terms of program optimization, this result provides a way to simplify a program with "for" loops by making its nested loop indices work in an independent fashion. For the first time in this manuscript, it is no longer possible to use the output size of the functions as a *semantic property* for discriminating between the classes, since both can have outputs of polynomial size. Therefore we introduce a new property named *repetitiveness* and show that it characterizes the functions computable by blind pebble transducers among those computed by pebble transducers with output in $\mathbb{Z}$ or $\mathbb{N}$. Once again, the proof relies on factorization forests.

Canonical models for pebble transducers with output in $\mathbb{Z}$. Given a pebble transducer with output in[20] $\mathbb{Z}$, we describe a procedure which builds a *canonical model* associated to the function that it computes.

[19] In a general fashion, *$\mathbb{S}$-rational series* are functions from words to a *semiring* $\mathbb{S}$ which are computed by a transducer model called *$\mathbb{S}$-weighted automata* (see e.g. [Sak09] for an introduction). Here we only consider the cases $\mathbb{S} := (\mathbb{Z}, +, \times)$ or $(\mathbb{N}, +, \times)$.

[20] Interestingy, the author is not aware of a way to adapt the construction to output in $\mathbb{N}$.

This canonical model can be seen as a specific form of marble transducer. We call it the *residual transducer* of the function, and its behavior is inspired by that of the *minimal automaton* of a regular language.

Recall that for regular languages (resp. rational functions), the construction of a canonical model has been successfully used to decide the membership problem for *star-free languages* (resp. *star-free rational functions*), which are subclasses of independent interest. We shift the notion of *star-freeness* to the functions computed by pebble transducers with output in $\mathbb{Z}$ or $\mathbb{N}$, and provide multiple equivalent characterizations of the related subclasses of functions. We finally show that one can decide if a function computed by a pebble transducer with output in $\mathbb{Z}$ is star-free. The proof relies on a semantic condition called *smoothness*, which translates to a (decidable) syntactic property of residual transducer that we call *aperiodicity* (which is adapted from the notion of *aperiodicity* for one-way automata).

Deterministic regular functions of infinite words. The remaining results of this manuscript concern transducers of infinite words. Since the literature is less advanced in this setting, we do not deal with nested machines and we focus on the model of *two-way transducers of infinite words*.

We introduce the class of *deterministic regular functions of infinite words* as the class of functions computed by two-way deterministic transducers of infinite words. Surprisingly enough, this class has never been investigated in the literature, contrary to the well-studied *regular functions of infinite words* which are defined by adding *ω-lookarounds* to two-way transducers. Even if deterministic regular functions are weaker than the regular ones (as depicted in Figure 12), they turn out to be more relevant when it comes to practical applications. Indeed, any deterministic regular function is effectively *computable*.

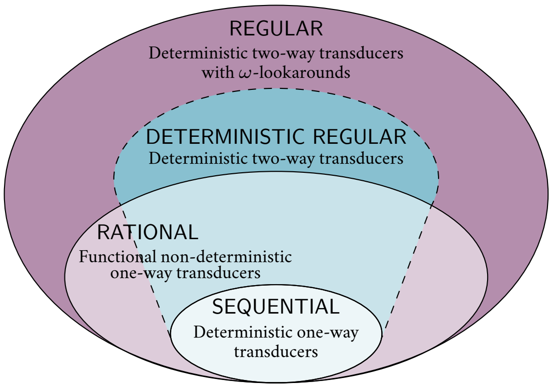


Figure 12: Classes of functions of infinite words.

We study the main properties of deterministic regular functions and show that they form a robust and natural class of functions of infinite words, which is closed under function composition. Furthermore, we introduce two meaningful computation models which also capture this class:

- *copyless streaming string transducers of infinite words* (which are defined by adapting copyless streaming string transducers of finite words). Such machines roughly describe *streaming programs*;
- two-way deterministic transducers enhanced with a weakened version of ω-lookarounds called *finite lookarounds*. This model is mainly used as a powerful tool in the proofs.

An equivalent description in terms of compositions of basic functions is additionally presented[21]. Similar results are known in the literature for regular functions of finite or infinite words, but the proofs in the case of deterministic regular functions have to overcome specific additional difficulties.

[21] A logical description is also available in [CDFW23] but the author chose not to present this result in this manuscript.

Determinization of rational functions of infinite words. It is conjectured since [DFKL20] that the class of deterministic regular functions is exactly[22] the class of regular functions which are *computable*/*continuous*. This result is believed to be rather difficult.

We provide a partial answer to this conjecture by showing that any computable/continuous rational function of infinite words is deterministic regular. The author considers this hard result as one of the jewels of this manuscript. Since continuity/computability is decidable, it solves the class membership problem from rational functions of infinite words to deterministic regular functions. In practice, it enables to build a deterministic program with bounded memory which computes a rational function.

Chapter by chapter outline

In Chapter 1, we recall the main definitions and properties of several transducer models of finite words (*one-way deterministic transducers*, *one-way non-deterministic transducers*, *two-way deterministic transducers*, *pebble transducers*). Chapter 2 provides a toolbox for the study of two-way transducers which is useful in Chapters 3, 5 and 6. In Chapter 3, we describe variants of pebble transducers called *blind pebble transducers* and *last pebble transducers* and show how to optimize them. In Chapter 4, we introduce *marble transducers*, relate them with *streaming string transducers* and show how to optimize them.

Chapter 5 focuses on *pebble transducers whose output is in $\mathbb{Z}$ or $\mathbb{N}$* (and no longer in finite words) and shows how they can be optimized. Furthermore, it connects the functions computed by these machines to the celebrated classes of *$\mathbb{Z}$-* and *$\mathbb{N}$-rational series*. In Chapter 6, we solve the class membership problem from pebble transducers with output in $\mathbb{Z}$ or $\mathbb{N}$ to blind pebble transducers with output in $\mathbb{Z}$ or $\mathbb{N}$. Chapter 7 describes a procedure for transforming a pebble transducer with outputs in $\mathbb{Z}$ into a *canonical object*. This result is leveraged to show that *star-freeness* is decidable in this setting.

In Chapter 8, we recall the main definitions and properties of several transducer models of infinite words. Chapter 9 studies the main properties of *deterministic regular functions of infinite words* and provide equivalent characterizations of this class, among others in terms of *streaming string transducers of infinite words*. In Chapter 10, we show that a *rational function of infinite words* is *deterministic regular* if and only if it is *continuous*, which solves yet another class membership problem.

[22]As for Footnote 14, a formal formulation would have to deal with extensions of functions.

How to read this document

Et puis mille neuf cent radiateurs, vingt-trois mille mètres carrés de linoléum, deux cent douze kilomètres de fils électriques, mille cinq cent robinets, cinquante-sept hydrants, cent soixante-quinze extincteurs ! Ca compte, hein ? C'est immense, immense. Par exemple, combien crois-tu que nous ayons de water-closets ?

Albert Cohen, *Belle du Seigneur*

Prerequisites

The reader is assumed to be familiar with the basics of automata theory, which includes the notions of *finite automaton*, *regular language* and *finite monoid*. The books [Sip12, Chapter 1] (in English) and [Car14, Chapitre 1] (in French) provide an introduction to this subject. Even though this manuscript is about *finite transducers*, the underlying automata models are ubiquitous (in particular in the proofs).

Previous knowledge about finite transducers is useful but not necessary. Indeed, all the transducers models used in this manuscript are defined in detail, in particular in Chapter 1 (over finite words) and Chapter 8 (over infinite words). Their known properties are furthermore recalled when needed.

No pre-requisites in *logics* are required. Indeed, even if the theory of transductions and formal languages is tightly connected to logics, this manuscript does not deal with this relationship.

Hyperlinks and numbering

This manuscript was written using Thomas Colcombet's package *knowledge*[1]. Roughly, this package enables to define a *term* and later re-use this term while creating an internal hyperlink which points to its definition. The reader is invited to jump within the document thanks to these hyperlinks. Some PDF readers even offer an overview of the definition when hovering above an hyperlink. Furthermore, most of them include a feature which enables to go back to the original page after a jump[2].

The use of hyperlinks is only relevant for an electronic version of this manuscript. In order to simplify the use of backward references in a printed version, the author has chosen to use a continuous numbering for definitions, theorems, propositions, lemmas, claims, examples, open problems, conjectures, figures, equations, algorithms and tables. These numbers are prefixed by the number of the chapter.

[1] `https://www.ctan.org/pkg/knowledge`.

[2] For instance with the keys ⌘+^ on Apple's *Preview*.

Dependencies between chapters

The main dependencies[3] between the chapters of this manuscript are depicted in Figure 13, where the three different colors correspond to the three main parts.

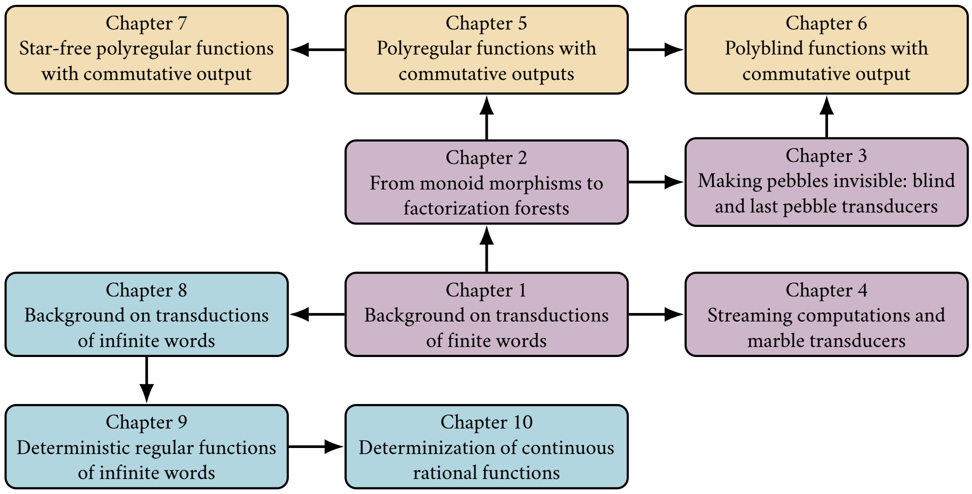


Figure 13: Dependency graph of this manuscript.

Notations and conventions

In this section we provide an overview of the main notations and writing conventions which are used throughout this manuscript. Most of them are re-defined when used for the first time.

Equality versus definition. The symbol $=$ is generally used to denote an equality between two objects which have already been defined. The symbol $:=$ is used to define new objects and to assign variables in algorithms. For instance, $x := v$ defines an object x which has value v.

Sets and functions. $\mathbb{N}$ (resp. $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$) denotes the set of *non-negative integers* (resp. *integers*, *rational numbers*, *real numbers*, *complex numbers*). If $m, n \in \mathbb{Z}$, we let $[i{:}j] := \{i, i{+}1, \dots, j{-}1, j\}$ if $i \leqslant j$ and $[i{:}j] := \varnothing$ otherwise. If S is a finite or countable set, we let $|S| \in \mathbb{N} \cup \{\infty\}$ be its *cardinality* (or size).

Set inclusion (resp. strict set inclusion) is denoted $\subseteq$ (resp. $\subset$). If S, T are two sets, then $S \rightarrow T$ denotes the type of *total functions* from S to T. Furthermore, $S \rightharpoonup T$ denotes the type of *partial functions* from S to T, i.e. of functions which are defined on a subset of S. If f has type $S \rightharpoonup T$, we denote its *domain* (i.e. the subset of S on which f is defined) by $\mathsf{Dom}(f)$.

Multisets. A *multiset* is a set with multiplicities, i.e. where elements can be duplicated. If $\mathfrak{M}$ is a finite or countable multiset, we let $|\mathfrak{M}| \in \mathbb{N} \cup \{\infty\}$ be its cardinality (including multiplicities). Furthermore we let $|\mathfrak{M}|_s \in \mathbb{N} \cup \{\infty\}$ be the number of occurrences of the element s in $\mathfrak{M}$. We write $s \in \mathfrak{M}$ to denote that $\mathfrak{M}$ contains at least one occurrence of s, i.e. $|\mathfrak{M}|_s \geqslant 1$.

[3] Other minor dependencies of references exist anyway from Chapter i to Chapter j for all $1 \leqslant i < j \leqslant 10$.

We use double braces $\{\!\{\cdots\}\!\}$ to denote multisets, for instance $\mathfrak{M} := \{\!\{s, s, t\}\!\}$ is such that $|\mathfrak{M}|_s = 2$ and $|\mathfrak{M}|_t = 1$. We denote by $\{\!\{s_1 \ddagger r_1, \dots, s_n \ddagger r_n\}\!\}$ a multiset containing n distinct elements $s_1, \dots, s_n$ of respective multiplicities $r_1, \dots, r_n$. For instance we have $\mathfrak{M} = \{\!\{s \ddagger 2, t \ddagger 1\}\!\}$.

Machines. Symbols $\mathscr{A}, \mathscr{B}, \mathscr{L}, \mathscr{P}, \mathscr{S}, \mathscr{T}$, etc. (in the `mathscr` font) are used in this manuscript to denote automata and transducers. If $\mathscr{T}$ is such a machine, the expressions $[\![\mathscr{T}]\!]$ or $[\![\![\mathscr{T}]\!]\!]$ usually denote (variants of) its *semantics*, i.e. the function or the language that it computes.

Finite and infinite words. Capital letters A, B, C denote *alphabets*, which are finite sets of symbols called *letters*. The symbols $a, b, c, 0, 1$ and $\#$ generally denote letters from an alphabet.

The set A^* denotes the set of *finite words* (i.e. finite sequences) over the alphabet A. The *empty word* is denoted ε. We let $A^+ := A^* \smallsetminus \{\varepsilon\}$. The set A^ω denotes the set of *infinite words* (i.e. infinite sequences) over A and $A^\infty := A^* \cup A^\infty$ is the set of both finite and infinite words.

If $u \in A^\infty$, we let $|u| \in \mathbb{N} \cup \{\infty\}$ be its *length*. For $1 \leqslant i \leqslant |u|$, $u[i] \in A$ denotes the *i-th letter*[4] of u. If $a \in A$, we let $|u|_a := |\{1 \leqslant i \leqslant |u| \mid u[i] = a\}| \in \mathbb{N} \cup \{\infty\}$ be the number of occurrence of letter a in u. If $I = \{i_1 < i_2 < \cdots\} \subseteq [1{:}|u|]$, we let $u[I] := u[i_1] \cdots u[i_\ell]$. In particular, if $1 \leqslant i \leqslant j \leqslant |u|$, $u[i{:}j]$ stands for $u[i]u[i{+}1] \cdots u[j]$ and if $j < i$ then $u[i{:}j] = \varepsilon$. We write $u[i{:}]$ for $u[i{:}|u|]$.

Word languages and word functions. If A is an alphabet, we say that $L \subseteq A^*$ (resp. $L \subseteq A^\omega$) is a *language* of finite words (resp. of infinite words). In this case, the function $\mathbf{1}_L \colon A^* \to \{0, 1\}$ (resp. $\mathbf{1}_L \colon A^\omega \to \{0, 1\}$) denotes the *characteristic function* of the language L. We let $\mathsf{RegLang}(A)$ (resp. $\omega\mathsf{RegLang}(A)$) be the set of *regular languages* of A^* (resp. of *ω-regular languages* of A^ω).

If A, B are alphabets and f has type $A^* \rightharpoonup B^*$ or $A^\omega \rightharpoonup B^\omega$, we use the roman letters u, v, w, etc. to denote the words of A^* or A^ω and the greek letters α, β, γ, etc. for the words of B^* or B^ω.

Word prefixes. Given $u, v \in A^\infty$, we write $u \sqsubseteq v$ when u is a *prefix* of v (i.e when $|u| \leqslant |v|$ and for all $1 \leqslant i \leqslant |u|$, $u[i] = v[i]$). We write $u \sqsubset v$ when u is a *strict prefix* of u. We say that two words u and v are *mutual prefixes* if either $u \sqsubseteq v$ or $v \sqsubseteq u$. Given $u, v \in A^\infty$, we let $u \wedge v$ be the (finite or infinite) *longest common prefix* of u and v. If u and v are mutual prefixes, we let $u \vee v$ be the *longest* of them.

Algebraic objects. If $\mathbb{S} := \mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}$ or $\mathbb{C}$, we denote by $\mathbb{S}[X]$ the set of *polynomials* in X which have coefficients in $\mathbb{S}$. Furthermore, $\mathbb{S}[X_1, \dots, X_n]$ denotes the set of *multivariate polynomials* in $X_1, \dots, X_n$. Given $\gamma \in \mathbb{C}$, the value $|\gamma|$ denotes[5] its *module* (in particular, it is the *absolute value* if $x \in \mathbb{R}$).

Given finite sets S, T, we denote by $\mathsf{M}_{S,T}(\mathbb{S})$ the set of matrices with coefficients in $\mathbb{S}$ and whose lines (resp. columns) are indexed by S (resp. T). If $m, n \geqslant 0$, we write $\mathsf{M}_{m,n}$ for $\mathsf{M}_{[1:m],[1:n]}$.

The symbols $\mathbb{M}$ and $\mathbb{T}$ (resp. μ and ν) generally denote *finite monoids* (resp. *monoid morphisms*).

[4] Beware that numbering of sequences therefore begins at 1 (and not at 0).

[5] The reader may observe that $|\cdot|$ denotes at the same time the cardinality of a set or a multiset, the length of a word and the modulus of a complex number. However, the context will always make it clear since these objects have different types.

Part I

Optimization of pebble transducers

Chapter 1

Background on transductions of finite words

Le galet n'est pas une chose facile à bien définir.
Si l'on se contente d'une simple description l'on peut dire d'abord que c'est une forme ou un état de la pierre entre le rocher et le caillou.
Mais ce propos déjà implique de la pierre une notion qui doit être justifiée. Qu'on ne me reproche pas en cette matière de remonter plus loin même que le déluge.

Francis Ponge, « Le galet », *Le parti pris des choses*

As mentioned in Introduction, the class of *regular languages* can be described to by various models of finite automata, which can be either be deterministic or non-deterministic, and either process their input by doing a single pass (one-way) or by also doing left moves (two-way). The notion of regularity for languages can be lifted to functions of finite words by defining various model of *finite transducers*. These machines are built by applying a generic receipt: we start from an finite automaton model which recognizes regular languages and we add outputs labels on its transitions.

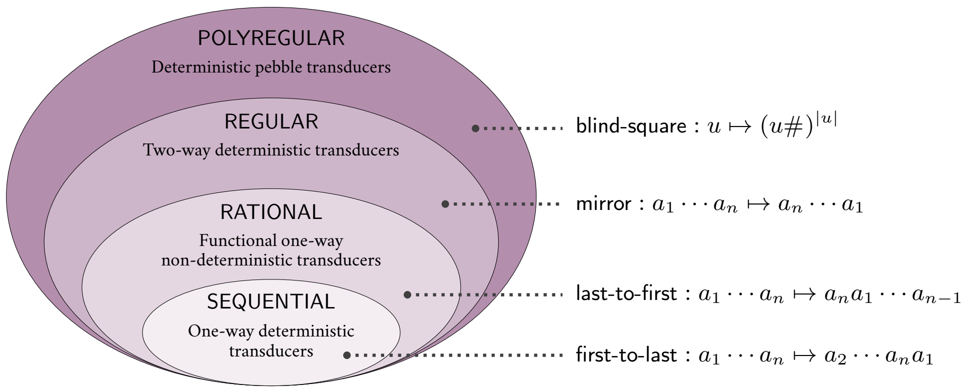


Figure 1.1: Classes of functions over finite words described in Chapter 1.

The goal of this chapter is to give a library of the various transducer models which will be studied in Part I. More precisely, we shall define in Sections 1.1 to 1.3 the following machine models:

- *one-way deterministic transducers*, which define the class of *sequential functions*;
- *one-way non-deterministic transducers*, which define the class of *rational functions*;
- *two-way deterministic transducers*, which define the class of *regular functions*;
- *pebble transducers*, which define the class of *polyregular functions*.

These classes are often considered as several functional counterparts of regular languages, due to their robustness and their algorithmic properties. Their expressive powers are compared in Figure 1.1.

We also recall that all the membership problems from one class to another are known to be decidable. These decidability results can be understood as program optimization techniques, since they enable to transform a complex device into an equivalent simpler one, whenever it is possible.

1.1 One-way transductions

The classes of functions described in Section 1.1 are computed by one-way finite automata enhanced with the ability to produce outputs along their runs. These machines can either be deterministic (which defines sequential functions) or non-deterministic (rational functions).

1.1.1 Sequential functions

We first describe the simplest machine model considered in this manuscript, named *one-way deterministic transducer*. It can be seen as a finite-state machine a one-way read-only input tape and a one-way write-only output tape, as depicted in Figure 1.18. It is thus a very basic kind of streaming algorithm.

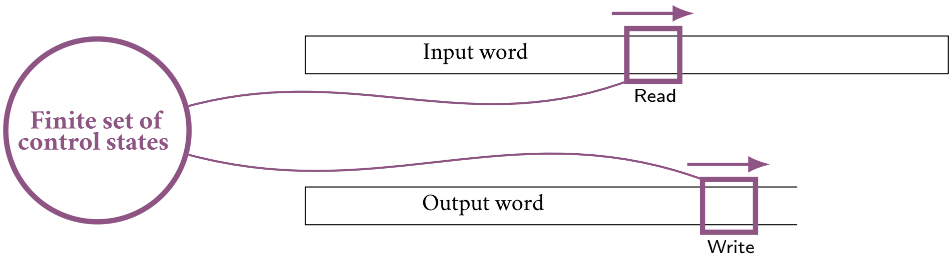


Figure 1.2: Behavior of a one-way deterministic transducer.

Such machines have been studied since the early days of automata theory [Gin62, GR66, Eil74].

Definition 1.3 (One-way deterministic transducer)

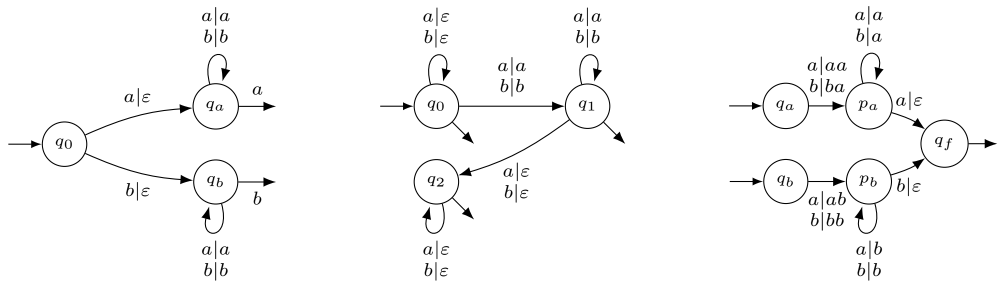
A *one-way deterministic transducer* (1DT) $\mathscr{T} = (A, B, Q, q_0, F, \delta, \lambda)$ consists of:

- an input alphabet A and an output alphabet B;
- a finite set of states Q with an initial state $q_0 \in Q$;
- a final output function $F: Q \rightharpoonup B^*$;
- a transition function $\delta: Q \times A \rightharpoonup Q$;
- an output function $\lambda: Q \times A \rightharpoonup B^*$.

The semantics of the 1DT is defined as follows. We write $q \xrightarrow{a|\alpha} q'$ whenever $\delta(q,a) = q'$ and $\lambda(q,a) = \alpha$. A *run* labelled by a word $a_1 \cdots a_n \in A^*$ is a sequence $p_0 \xrightarrow{a_1|\alpha_1} p_1 \cdots \xrightarrow{a_n|\alpha_n} p_n$. We say that the run is *initial* if $p_0 = q_0$, and *final* if $p_n \in \mathsf{Dom}(F)$. A run is *accepting* if it is both initial and final. Let us define the function $[\![\mathscr{T}]\!] \colon A^* \rightharpoonup B^*$ computed by $\mathscr{T}$. Let $u \in A^*$ be the input, then $[\![\mathscr{T}]\!](u)$ is defined if and only if there exists an accepting run of $\mathscr{T}$ labelled by u (observe that it has to be unique). If $p_0 \xrightarrow{a_1|\alpha_1} p_1 \cdots \xrightarrow{a_n|\alpha_n} p_n$ denotes this run, we let $[\![\mathscr{T}]\!](u) := \alpha_1 \cdots \alpha_n F(p_n)$.

Example 1.4 (First to last)

Let $A := \{a,b\}$. The function first-to-last: $A^+ \to A^+$ which maps au to ua (resp. bu to ub) for $u \in A^*$ is computed by the 1DT depicted in Figure 1.5a (initial states have incoming arrows, whereas final states have outing arrows labelled by their outputs).



(a) 1DT computing the function first-to-last.

(b) Real-time 1NT computing the relation factors.

(c) Real-time and unambiguous 1NT computing last-to-first.

Figure 1.5: Functions and relations computed by 1DT and 1NT.

Definition 1.6 (Sequential function)

The class of *sequential functions*[1] is the class of functions computed by 1DT.

It is easy to see that the domain of a sequential function is a regular language, since it is described by the one-way deterministic automaton $(A, Q, q_0, \mathsf{Dom}(F), \delta)$. This class of functions enjoy numerous properties (see e.g. [Sak09] for a survey) such as closure under composition, or a purely semantics characterization in terms of pre-images of regular languages and bounded variations [GR66, Sch77].

Given a sequential function computed by a 1DT one can effectively compute a *canonical* (in the sense that it is intrinsically associated to the function, and does not depend on the machine which was given to describe it) 1DT that computes it. The construction of this canonical 1DT is inspired by the minimization algorithm of deterministic automata, see e.g. [Cho03] or [CP17, Section 4].

1.1.2 Rational functions

It is well-known that non-deterministic automata are as expressive as the deterministic ones, since both models compute regular languages. It is thus very natural to add non-determinism, together with ε-transitions, to 1DT. The study of the relations described by such non-deterministic machines goes back to [EM65], following the models introduced in [RS59, Chapter III].

[1] In part of the literature (e.g. [Sch77, Cho77] or more recently [FGRS13]), our sequential functions are called *subsequential functions*. In their case, the term *sequential* is devoted to the functions where $F(q) = \varepsilon$ for all $q \in Q$.

Definition 1.7 (One-way non-deterministic transducer)

A *one-way non-deterministic transducer* (1NT) $\mathscr{N} = (A, B, Q, I, F, \Delta, \lambda)$ consists of:

- an input alphabet A and an output alphabet B;
- a finite set of states Q with initial states $I \subseteq Q$ and final states $F \subseteq Q$;
- a transition relation $\Delta \subseteq Q \times (A \cup \{\varepsilon\}) \times Q$;
- an output function $\lambda \colon \Delta \to B^*$.

The semantics of a 1NT is defined in a similar way to that of a 1DT. We write $q \xrightarrow{u|\alpha} q'$ whenever $(q, u, q') \in \Delta$ (beware that here $u \in A \cup \{\varepsilon\}$) and $\alpha = \lambda(q, u, q')$. A *run* labelled by an input word $u_1 \cdots u_n \in A^*$ is a sequence $p_0 \xrightarrow{u_1|\alpha_1} q_1 \cdots \xrightarrow{u_n|\alpha_n} p_n$. The word $\alpha_1 \cdots \alpha_n \in B^*$ is said to be the *output* along the run. We say that the run is *initial* if $p_0 \in I$, and *final* if $p_n \in F$. It is *accepting* if it is both initial and final. The relation $[\![\mathscr{N}]\!] \subseteq A^* \times B^*$ computed by $\mathscr{N}$, is defined as follows:

$$[\![\mathscr{N}]\!] := \{(u, \alpha) \mid \alpha \in B^* \text{ is output along some accepting run labelled by } u\},$$

i.e. an input word is mapped to all the possible outputs of accepting runs labelled by itself.

Observe that 1DT can been seen as a particular case of 1NT. Indeed, even if our definition of 1NT has no final function $F \colon Q \rightharpoonup B^*$, it can be encoded e.g. using *ε-transitions* (i.e. transitions of shape (q, ε, q') in Δ). We say that a 1NT is *real-time* if it has no ε-transition.

Example 1.8 (Factors)

The relation factors $\subseteq A^* \times A^*$ defined by $(u, \alpha) \in$ factors if and only if α is a factor of u is computed by the real-time 1NT depicted in Figure 1.5b.

In this manuscript, we focus on the functions which are described by the various machines, and not on the relations (since the latter cannot be computed by deterministic algorithms). Let us recall the classical notions of *functionality* and *unambiguity* in order to describe the functions computed by 1NT. Definition 1.9 is made generic so that it can be re-used for other non-deterministic models.

Definition 1.9 (Functionality, unambiguity)

A non-deterministic machine is said to be *functional* if it computes a relation $r \subseteq A^* \times B^*$ such that for all $u \in A^*$, there exists at most one $\alpha \in B^*$ such that $(u, \alpha) \in r$ (in other words, r can be seen as a partial function $A^* \rightharpoonup B^*$). The machine is said to be *unambiguous* if for all input $u \in A^*$, there exists a most one accepting run labelled by u.

Observe that any unambiguous machine is functional. The converse does not hold, but it is well-known that any functional 1NT can be transformed in a real-time and unambiguous 1NT computing the same function (the result follows from [Eil74, Theorem IX.2.2] which deals with uniformization, see also [CL11, Remark 12]). This result requires to ignore the input ε, whose image is necessarily ε in a real-time 1NT, but may not be empty in general (and in particular for a 1DT with a final function).

Definition 1.10 (Rational function)

The class of *rational functions* is the class of functions computed by functional 1NT.

It turns out that rational functions are strictly more expressive than sequential functions, roughly because non-determinism enables one to describe local transformations which depend on properties of the future of the input string (for instance that depend on its last letter, see Example 1.11).

Example 1.11 (Last to first)

Let $A := \{a, b\}$. The function last-to-first: $A^+ \to A^+$ that maps ua to ua (resp. ub to bu) for $u \in A^*$ is computed by the real-time and unambiguous 1NT depicted in Figure 1.5c. Contrary to the function first-to-last of Example 1.4, it is easy to show that last-to-first is not sequential.

1.1.2.1 Equivalent formalisms. Rational functions are closed under composition (it is also the case for the relations computed by 1NT). Furthermore, this class is captured by several formalisms, including a logical model called *order-preserving monadic second-order transductions* (order-preserving MSO transductions for short), see [Boj14, Theorem 4.1] or [Fil15, Theorem 4].

Rational functions can also be described as compositions of sequential functions and sequential functions which process the input from right to left, as shown in Proposition 1.12 originating from [EM65, Theorem 7.8]. We let mirror : $u = a_1 \cdots a_n \mapsto \widetilde{u} := a_n \cdots a_1$ be the mirror image function.

Proposition 1.12 (Rational = left sequential ∘ right sequential)

A function is rational if and only if it can be written as a composition $f \circ \mathsf{mirror} \circ g \circ \mathsf{mirror}$ where f and g are sequential functions.

In other words, rational function can be computed in a deterministic fashion, at the cost of processing the input both from left to right and from right to left. This very idea is the heart of the bimachine model introduced in [Sch61b] and later named in [Eil74]. Such a machine produces an output in each position of its input word, depending on a regular property of the input where the current position is distinguished. We recall that $\mathsf{RegLang}(A)$ denotes the set of regular languages of A^*.

Definition 1.13 (Bimachine)

A *bimachine* $\mathscr{B} = (A, B, \lambda)$ consists of:

- an input alphabet A and an output alphabet B;
- an output function $\lambda\colon \mathsf{RegLang}(A) \times A \times \mathsf{RegLang}(A) \rightharpoonup B^*$ such that:
  (1) $\mathsf{Dom}(\lambda)$ is finite;
  (2) for all $(L, a, R) \neq (L', a, R') \in \mathsf{Dom}(\lambda)$, $RaL \cap R'aL' = \varnothing$.

The function $[\![\mathscr{B}]\!]\colon A^* \rightharpoonup B^*$ computed by $\mathscr{B}$ is defined as follows. Let $u \in A^*$, then by the last item of Definition 1.13 for all $1 \leqslant i \leqslant |u|$, there exists at most one $(L_i, u[i], R_i) \in \mathsf{Dom}(\lambda)$ such that $u[1{:}i{-}1] \in L_i$ and $u[i{+}1{:}|u|] \in R_i$. If such an $(L_i, u[i], R_i)$ exists for all $1 \leqslant i \leqslant |u|$, we let $[\![\mathscr{B}]\!](u) := \lambda(L_1, u[1], R_1) \cdots \lambda(L_{|u|}, u[|u|], R_{|u|})$, and otherwise $[\![\mathscr{B}]\!](u)$ is undefined.

Note that Item (1) of Definition 1.13 ensures that the machine has a finite description (the regular languages in $\mathsf{Dom}(\lambda)$ can be given e.g. by one or several finite automata, or equivalently by a morphism into a finite monoid). The behavior of a bimachine is depicted in Figure 1.14.

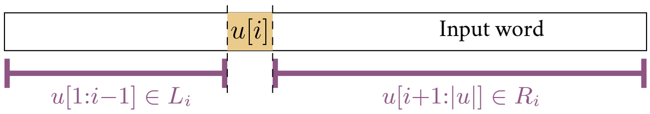


Figure 1.14: Behavior of a bimachine when producing $\lambda(L_i, u[i], R_i)$.

Example 1.15 (Last to first)

The function last-to-first of Example 1.11 is computed by bimachine inputting $a_1 \cdots a_n \in A^+$ and outputs $a_1 a_n$ in position 1, a_i in position i for $1 < i < n$ and ε in position n.

Now, we reformulate Proposition 1.12 by recalling that bimachines compute rational functions. We safely ignore in Proposition 1.16 the case of input ε (but it can formally be treated by adding a specific output value, at the cost of burdening Definition 1.13).

Proposition 1.16 (Rational = Bimachine)

A function is rational if and only if it can be computed by a bimachine.

One of the most striking results on bimachines is the existence and computability of a canonical bimachine associated to each rational function [RS91], generalizing the aforementioned result of 1DT. The construction of [RS91] has been refined in [FGL19] to decide whether a rational function can be described by some bimachine whose languages are star-free (see Open question 7.5).

1.1.2.2 Decision problems. It is well-known that the equivalence problem for rational functions is decidable, while the same problem for relations computed by 1NT becomes undecidable (see e.g. [Ber13, Chapter 4] for a survey). Furthermore, one can decide if a 1NT is functional.

Let us discuss the membership problem from rational functions to sequential functions. This problem was first shown decidable in [Cho77, Corollaire 3.5], and a polynomial-time complexity bound was later given in [WK95, Theorem 4.3] and [BCPS03][2]. The proof consists in showing that the runs of a 1NT computing a sequential function must follow specific patterns, which are often called *twinning properties* (see Lemma 10.8 for similar ideas over infinite words). Furthermore, one can build a 1DT which computes the function, whenever it exists. Hence this construction can be seen as a first program optimization result, since it enables to build a simple deterministic machine whenever it exists.

Theorem 1.17 (Rational $\to$ Sequential)

One can decide if a rational function (given by a 1NT) is sequential. If this property holds, one can build a 1DT which computes the function.

1.2 Regular functions

In this section, we describe finite-states machines which can travel back and forth on their input, without modifying it. For the machine to detect the borders of its input without leaving it, a symbol $\vdash$ (resp. $\dashv$) is added before the first (resp. after the last) position. This definition builds the model of *two-way deterministic automaton* (when seen as an acceptor for languages) whose study was initiated in the 1950s. Surprisingly enough, it turns out that despite the ability of backwards reference, these machines compute no more than regular languages (see [RS59, Theorem 15], which refers to the main result of [She59]).

1.2.1 Two-way transducers

A *two-way transducer* consists of a two-way automaton which produces outputs along its transitions. It can also be seen as a finite-state machine with two tapes, as depicted in Figure 1.18.

[2] Beware that these papers use the subsequential terminology.

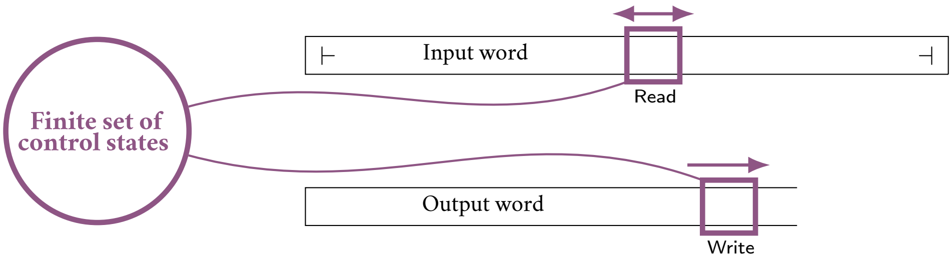


Figure 1.18: Behavior of a two-way deterministic transducer.

This model is mentioned in [She59, Note 4], but its study formally began with [AHU69].

Definition 1.19 (Two-way deterministic transducer)

A *two-way deterministic transducer* (2DT) $\mathscr{T} = (A, B, Q, q_0, F, \delta, \lambda)$ consists of:

- an input alphabet A and an output alphabet B;
- a finite set of states Q with an initial state $q_0 \in Q$ and a set $F \subseteq Q$ of final states;
- a transition function $\delta\colon Q \times (A \uplus \{\vdash, \dashv\}) \rightharpoonup Q \times \{\triangleleft, \triangleright\}$ such that for all $q \in F$ and $a \in A \uplus \{\vdash, \dashv\}$, $\delta(q, a)$ is undefined;
- an output function $\lambda\colon Q \times (A \uplus \{\vdash, \dashv\}) \rightharpoonup B^*$ with same domain as δ.

A *configuration* of $\mathscr{T}$ over a word $u \in (A \uplus \{\vdash, \dashv\})^*$ is a tuple (q, i) where $q \in Q$ is the current state and $1 \leqslant i \leqslant |u|$ is the current position of the reading head. The *transition relation* $\rightarrow$ of $\mathscr{T}$ over input u is defined as follows. If (q, i) is a configuration, the transition $(q, i) \rightarrow (q', i')$ holds whenever either $\delta(q, u[i]) = (q', \triangleleft)$ and $i' = i - 1$ (move left), or $\delta(q, u[i]) = (q', \triangleright)$ and $i' = i + 1$ (move right). A *run* of $\mathscr{T}$ labelled by u is a finite sequence of configurations $(q_1, i_1) \rightarrow \cdots \rightarrow (q_n, i_n)$.

The partial function $[\![\mathscr{T}]\!]\colon A^* \rightharpoonup B^*$ computed by $\mathscr{T}$ is defined as follows. Let $u \in A^*$ be the input, and consider the runs of $\mathscr{T}$ over the word $\vdash u \dashv$. We say that a run is *initial* if it starts in $(q_0, 1)$, and *final* if it ends in a configuration of shape $(q, |\vdash u \dashv|)$ with $q \in F$. A run is *accepting* if it both initial and final. The output $[\![\mathscr{T}]\!](u)$ is defined if and only if there exists a (necessarily unique) accepting run $(q_1, i_1) \rightarrow \cdots \rightarrow (q_n, i_n)$ labelled by $\vdash u \dashv$. In this case, $[\![\mathscr{T}]\!](u)$ is the concatenation $\lambda(q_1, u[i_1]) \cdots \lambda(q_n, u[i_n])$ of the outputs produced along this run.

An initial run of a 2DT is depicted informally in Figure 1.20.

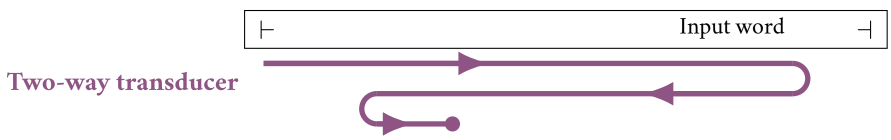


Figure 1.20: Initial run of a two-way transducer.

Example 1.21 (Copy)

The function $u \mapsto uu$ can be computed by a 2DT which performs a first left-to-right pass on its input (until $\dashv$), then a right-to-left pass (until $\vdash$), a finally a second left-to-right pass.

Example 1.22 (Reverse)

The function mirror: $u \mapsto \widetilde{u}$ can be computed by a 2DT which performs a left-to-right pass followed by a right-to-left pass while producing outputs.

Example 1.23 (Map copy reverse)

The function map-copy-reverse: $(A \uplus \{\#\})^* \to (A \uplus \{\#\})^*$ has an input of shape $u_1\# \dots \#u_n$ where each $u_i \in A^*$ and outputs $u_1\#\widetilde{u_1}\# \dots \#u_n\#\widetilde{u_n}$. It can be computed by a 2DT which visits each factor u_i from left to right (to output $u_i\#$), then from left to right (to output $\widetilde{u_i}\#$), and finally from left to right (only to reach the next factor).

It is not hard to show that none of the functions from Examples 1.21 to 1.23 is rational.

Definition 1.24 (Regular functions)

The class of *regular functions* is the class of functions computed by 2DT.

1.2.2 Normalization and origin semantics

The first goal of this section is to claim that any 2DT^ω can be *normalized*, i.e. put in a somehow simple shape while removing domain issues. Next, we define an extended semantics for 2DT^ω, which enables to describe which input positions were used to produce which output letters. This notion, called *origin semantics*, was first introduced in [VDKT93] and later investigated in detail for transductions in [Boj14]. We shall not study origin semantics for itself, but only as a tool to deal with nested 2DT. Indeed, such machines are 2DT^ω which "call" other 2DT^ω in any position of the input, while marking this position.

1.2.2.1 Normalized two-way transducers. Since two-way automata recognize regular languages, it is easy to see that the domain of a regular function is (effectively) a regular language. We can therefore complete any partial regular function into a total one, by modifying the given 2DT so that is does a first pass on the input to check if it belongs to the domain, and produces a specific output if it is not the case. This motivates the definition of *normalized* 2DT, which also deals with the end-markers.

Definition 1.25 (Normalized two-way transducer)

We say that a two-way transducer $\mathscr{T} = (A, B, Q, q_0, F, \delta, \lambda)$ is *normalized* if the following holds:

- it computes a total function, i.e. it has exactly one accepting run on each $\vdash u \dashv$;
- for all $q \in Q$ and $a \in A$, $\lambda(q, a) \in B \cup \{\varepsilon\}$ (at most one letter);
- for all $q \in Q$, $\lambda(q, \vdash) = \lambda(q, \dashv) = \varepsilon$ (no outputs on the end-markers).

From now on, we freely assume that our 2DT are always normalized. Once more, we (safely) ignore the case of input ε, for which the output is necessarily ε in a normalized machine.

Let $\rho := (q_1, i_1) \to \cdots \to (q_n, i_n)$ be the run of a normalized 2DT over $\vdash u \dashv$. We say that the sequence $\rho' := (q_j, i_j - 1)_{1 \leqslant j \leqslant n \text{ and } 1 < i_j < |\vdash u \dashv|}$ is an *n-run* of $\mathscr{T}$ labelled by $u \in A^*$. Roughly, ρ' is ρ in which we removed all configurations on letters $\vdash$ and $\dashv$, while offsetting in order to send the remaining positions into $[1{:}|u|]$. Thus it is nearly a run of $\mathscr{T}$ labelled by u, but it may not be formally the case (due to the borders). Observe that the production along ρ' is the same as that along ρ. If ρ is accepting (resp. initial, resp. final), we say that ρ' is *accepting* (resp. *initial*, resp. *final*).

1.2.2.2 Origin semantics. Given a normalized 2DT, now we present an extension of its semantics which captures the precise relation between the output and the input positions. In practice, the *origin semantics* consists in labelling each position of the output by the input position in which it was created.

If $u \in B^*$ and $i \in \mathbb{N}$, we let $u \ltimes i := (u[1], i) \cdots (u[|u|], i) \in (B \times \mathbb{N})^*$.

Definition 1.26 (Origin semantics)

Let $\mathscr{T} = (A, B, Q, q_0, F, \delta, \lambda)$ be a normalized 2DT. We let the function $f\colon A^* \to (B \times \mathbb{N})^*$ computed by $\mathscr{T}$ in *origin semantics* be defined as follows. Given $u \in A^*$, if $(q_1, i_1) \to \cdots \to (q_n, i_n)$ be the accepting n-run of $\mathscr{T}$ labelled by u, then:

$$f(u) := (\lambda(q_1, u[i_1]) \ltimes i_1)(\lambda(q_2, u[i_2]) \ltimes i_2) \cdots (\lambda(q_n, u[i_n]) \ltimes i_n).$$

Observe that the function $[\![\mathscr{T}]\!]\colon A^* \to B^*$ computed by $\mathscr{T}$ is the first component of f.

Example 1.27 (Copy, reverse)

The function $a_1 \cdots a_n \mapsto (a_1, 1) \cdots (a_n, n)(a_1, 1) \cdots (a_n, n)$ is computed in origin semantics by the 2DT from Example 1.21. The function $a_1 \cdots a_n \mapsto (a_n, n) \cdots (a_1, 1)$ is computed in origin semantics by the 2DT from Example 1.22.

1.2.3 Two-way transducers with lookarounds

In this section, we extend the model of 2DT with an extra feature called lookarounds. Intuitively, a 2DT with lookarounds is a 2DT enhanced with the ability to choose its transitions depending on a regular property of its input where the current position is distinguished. In other words, it is a mix between a 2DT and a bimachine. Lookarounds are a key concept in the study of 2DT, since they provide more flexibility when building machines. They were first studied in [HU67] for two-way automata.

Definition 1.28 (Two-way transducer with lookarounds)

A two-way deterministic transducer (2DT) with *lookarounds* consists of a modified two-way deterministic transducer $\mathscr{T} = (A, B, Q, q_0, F, \delta, \lambda)$ such that:

- the transition function δ has type $(Q \times \mathsf{RegLang}(A) \times A \times \mathsf{RegLang}(A)) \rightharpoonup Q$;
- the output function λ has type $(Q \times \mathsf{RegLang}(A) \times A \times \mathsf{RegLang}(A)) \rightharpoonup B^*$;
- $\mathsf{Dom}(\delta) = \mathsf{Dom}(\lambda)$ and this set is finite;
- for all $(q, L, a, R) \neq (q, L', a, R') \in \mathsf{Dom}(\delta)$, we have $LaR \cap L'aR' = \varnothing$.

The semantics of a 2DT with lookarounds is similar to that of 2DT, with the difference that the transition relation is no longer local. Let $u \in A^*$ and (q, i) be a configuration of $\mathscr{T}$ over u, then by the last item of Definition 1.28 there exists at most one tuple $(q, L, u[i], R) \in \mathsf{Dom}(\delta)$ such that $u[1{:}i{-}1] \in L$ and $u[i{+}1{:}n] \in R$. The transition $(q, i) \to (q', i')$ holds whenever either $\delta(q, L, u[i], R) = (q', \triangleleft)$ and $i' = i - 1$, or $\delta(q, L, u[i], R) = (q', \triangleright)$ and $i' = i + 1$. The notion of *run* is defined accordingly. The function computed by a 2DT with lookarounds is defined as for a 2DT.

Observe that no $\vdash$ nor $\dashv$ are needed in this case, since the machine can detect the borders with the lookarounds. The notions of *normalized* machine (and *n-run*) can be extended to 2DT with lookarounds. The function computed in *origin semantics* by a 2DT with lookarounds is defined as for a 2DT.

Example 1.29 (Rational functions using a lookaround)

It is easy to see that any bimachine can be simulated by a 2DT with lookarounds which has a single state. As a consequence, any rational function can be computed by this model.

The celebrated tree construction from [HU67, Lemma 3] has been leveraged to show that lookarounds do not give extra expressiveness to 2DT. An improved construction (used for 2DT composition[3]) is given in [DFJL17, Section 4]. Furthermore, the proofs preserve the origin semantics of the machines (this result will be used when dealing with pebble transducers in Section 1.3.2), as claimed in Theorem 1.30.

Theorem 1.30 (Lookarounds removal)

Given a normalized 2DT with lookarounds, one can build an normalized 2DT (without lookarounds) which computes the same function in origin semantics.

In particular, 2DT with lookarounds are as expressive as 2DT.

As an easy consequence of Example 1.29 and Theorem 1.30, one can observe that rational functions are a subclass of the regular ones. Note that this property was not obvious at first glance, due to the fact that 2DT are deterministic while 1NT are not. Furthermore, the inclusion is strict.

1.2.4 Basic properties of regular functions

The goal of this section is to recall well-known properties of regular functions.

1.2.4.1 Composition and decomposition. Closure under composition of regular functions can be shown by doing a product construction and crucially relying on Theorem 1.30. The curious reader is invited to consult Section 9.5 for a similar detailed proof in the (more complex) case of deterministic regular functions of infinite words. In the literature, closure under composition was first claimed in [CJ77], and a more efficient construction (in exponential time) is given in [DFJL17].

Theorem 1.31 (Composition of regular functions)

The class of regular functions is (effectively) closed under composition.

Now, let us give an analogue of Proposition 1.12 which decomposed rational functions using the sequential ones. To our knowledge, the next result is not stated explicitly in the literature, but a generalization to infinite alphabets has been proven in [BS20, Theorem 13].

Theorem 1.32 (Decomposition of regular functions)

A function is regular if and only if can be written as a composition of sequential functions (or rational functions) and map-copy-reverse functions. The conversions are effective.

1.2.4.2 Equivalent formalisms. Several equivalent descriptions of regular functions have been studied. In [EH01][4], a logical model called *monadic second-order transductions* (MSO transductions for short) was shown equivalent to regular functions. Informally, an MSO transduction is a collection of MSO

[3] Closure under composition (Theorem 1.31) and lookaround removal (Theorem 1.30) are in fact equivalent.

[4] This seminal paper created a renewed interest for the study of regular functions, which continued to this day.

formulas with 2 free first-order variables, whose semantics shows how to encode an output word within a bounded number of copies of the input. Various formalisms that use combinators, in the spirit of *regular expressions* for regular languages, have also been shown to capture regular functions [AFR14, DGK18, BDK18, BR18]. Another equivalent transducer model, called copyless streaming string transducers, will be discussed in Corollary 4.35 which originates from [AC10].

On the other hand, extending 2DT with non-determinism does not increase their expressive power when they define functions: functional *two-way non-deterministic transducers* describe no more than regular functions (this result was first shown in [Eng81, Theorem 4.9]). For this reason, we shall never consider two-way non-deterministic transducers in this manuscript.

1.2.4.3 Decision problems: class membership and equivalence. The class membership problem from regular functions to rational functions was first shown decidable in [FGRS13, Theorem 1]. An elementary complexity bound was later given in [BGMP18, Theorem 3.3], see also [MP19, Theorem 11] for a survey of the complexity upper and lower bounds. Once more, the result enables to effectively build a simpler machine which computes the function, whenever it is possible.

Theorem 1.33 (Regular $\rightarrow$ Rational)

One can decide if a regular function is rational. If this property holds, one can build a 1NT which computes the function.

As a generalization of the rational case, equivalence of regular functions is decidable. This result was first shown in [Gur80, Theorem 1] by reduction to equivalence of some bounded counter machines model. A modern and more conceptual proof, which relies on Hilbert's basis theorem, is available e.g. in [Boj19]. However, there is no known canonical object to describe regular functions[5].

Theorem 1.34 (Equivalence of regular functions)

Given two regular functions $f, g \colon A^* \rightharpoonup B^*$, one can decide if $f = g$.

1.3 Polyregular functions

It is easy to observe that if $f \colon A^* \to B^*$ is regular, then $|f(u)| = \mathcal{O}(|u|)$. Indeed, the lengths of the accepting run of a 2DT with states Q, labelled by $u \in A^*$, is at most $(|u|{+}2)|Q|$ (otherwise it would visit twice the same configuration). In Section 1.3, we describe a class of functions whose output can be polynomial in the input size, named *polyregular functions*.

1.3.1 Pebble transducers

A basic idea for enriching two-way automata is to allow them to drop a bounded number k of marks on their input string. This way, the number of configurations is a polynomial (of degree k) in the input size. However, a direct implementation of this idea yields the same expressive power as deterministic Turing machines with logarithmic space[6], see [Iba71, Corollary 3.5] and [Har72].

The model of *k-pebble automaton* is built by forcing the k marks (called *pebbles*) to follow a stack discipline: the i-th pebble can only (re)moved if it is the last one (i.e. if no $(i{+}1)$-th pebble is dropped).

[5] However, building a canonical object is easy when considering regular functions with origin semantics, see [Boj14].
[6] The latter are miles away from finite automata and regular languages.

From this perspective, k-pebble automata can be seen as machines which execute nested (two-way) "for loops": intuitively, the i-th mark corresponds to the index of the i-th nested loop. This formalism was first used to enrich tree-walking automata in [EH99][7]. It was shown that over finite words, pebble automata recognize no more than regular languages (the situation for trees languages is far more complex, see [BSSS06, EH06]). Pebble transducers were then introduced by adding an output mechanism to pebble automata, and studied over trees as a model for XML-based query languages [MSV00]. Their restriction to finite words was first considered in [EM02]. Recently, a regain of interest for pebble transducers over finite words has followed from Bojańczyk's extensive study [Boj18].

In this manuscript, we do not exactly follow the definitions of [EM02, Boj18] for pebble transducers, but we present instead a definition based on nested 2DT (both definitions are equivalent, which is discussed in detail in Section 1.3.2). Let a 1-*pebble transducer* be simply a 2DT. A 2-*pebble transducer* consists of a *head* 2DT which, when on any position i of its input word, can call auxiliary 2DT. The latter take as input the original input with a pebble dropped on position i (formally, this position is marked). The head 2DT then outputs the concatenation of all the outputs produced along its calls. We generalize this idea in Definition 1.36, by defining a k-*pebble transducer* for $k \geqslant 1$ as a tree of height k.

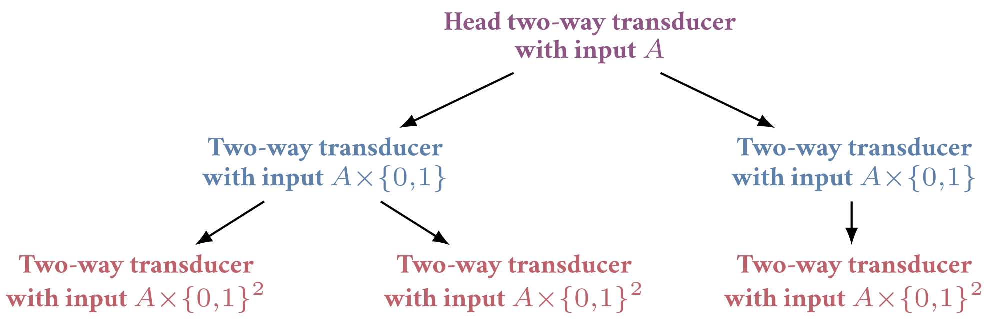


Figure 1.35: Syntax of a 3-pebble transducer with input alphabet A.

We write $a\langle t_1\rangle \cdots \langle t_n\rangle$ to denote a tree whose root node is labelled by a and whose rooted subtrees are $t_1, \dots, t_n$. Formally, a k-pebble transducer is a tree of height k whose nodes are labelled by normalized 2DT. The root has input alphabet A, its children $A \times \{0, 1\}$, etc.

Definition 1.36 (Pebble transducer)

Let $k \geqslant 1$ and $\mathscr{T}$ be a normalized 2DT with input alphabet A. We say that $\mathscr{P}$ is a k-*pebble transducer* with input alphabet A, output alphabet B and *head* $\mathscr{T}$ if:

- either $k = 1$, $\mathscr{P} = \mathscr{T}$ and it has output alphabet B;
- or $k \geqslant 2$, $\mathscr{P}$ is a tree $\mathscr{T}\langle\mathscr{P}_1\rangle \cdots \langle\mathscr{P}_p\rangle$ with $p \geqslant 1$ and:
  - the subtrees $\mathscr{P}_1, \dots, \mathscr{P}_p$ are $(k-1)$-pebble transducers with input alphabet $A \times \{0, 1\}$, output alphabet B, and respective heads $\mathscr{T}_1, \dots, \mathscr{T}_p$;
  - $\mathscr{T}$ has output alphabet $\{\mathscr{T}_1, \dots, \mathscr{T}_p\}$.

We say that a 2DT $\mathscr{T}$ is a *submachine* of the pebble transducer $\mathscr{P}$ if $\mathscr{T}$ labels a node in the tree which defines $\mathscr{P}$. The tree structure of a 3-pebble transducer is depicted in Figure 1.35. Note that each submachine has an input alphabet which depends on its height in the tree.

[7] Contrary to a persistent belief, the *pebble automata* introduced by Globerman and Harel in [GH96, Definition 4.1] do not match the notion of pebble that is considered in the rest of the literature, e.g. in [EH99]. Indeed, they add additional constraints in the spirit of the marbles of Chapter 4. The author thanks Nguyễn for this observation.

Given $u \in A^*$ and $1 \leqslant i \leqslant |u|$, we let $u{\bullet}i \in (A \times \{0,1\})^*$ be itself where position i is marked, i.e. $(u[1],0)\cdots(u[i{-}1],0)(u[i],1)(u[i{+}1],0)\cdots(u[|u|],0)$. If $\mathscr{T}$ is the head of the k-pebble transducer $\mathscr{P}$, we define the function computed by $\mathscr{T}$ within $\mathscr{P}$, denoted $[\![\mathscr{T}]\!]\colon A^* \to B^*$, by induction:

- if $k = 1$, then $[\![\mathscr{T}]\!]$ is $[\mathscr{T}]\colon A^* \to B^*$ which follows the usual 2DT semantics;
- otherwise $\mathscr{T}$ has output alphabet $T := \{\mathscr{T}_1, \dots, \mathscr{T}_p\}$ and the functions $[\![\mathscr{T}_1]\!], \dots, [\![\mathscr{T}_1]\!]$ have been defined by induction. Let $g\colon A^* \to (T \times \mathbb{N})^*$ be the function computed by $\mathscr{T}$ in origin semantics. Given $u \in A^*$, if $g(u) = (t_1, i_1)\cdots(t_n, i_n)$, then we let:

$$[\![\mathscr{T}]\!](u) := [\![t_1]\!](u{\bullet}i_1)\cdots[\![t_n]\!](u{\bullet}i_n).$$

The function $f\colon A^* \to B^*$ computed by $\mathscr{P}$ is defined as $[\![\mathscr{T}]\!]$ for its head $\mathscr{T}$. We generalize the notation $[\![\mathscr{T}]\!]$ to any submachine $\mathscr{T}$ of $\mathscr{P}$, by observing that it is the head of a subtree.

The nested behavior of a 3-pebble transducer is depicted in Figure 1.37.

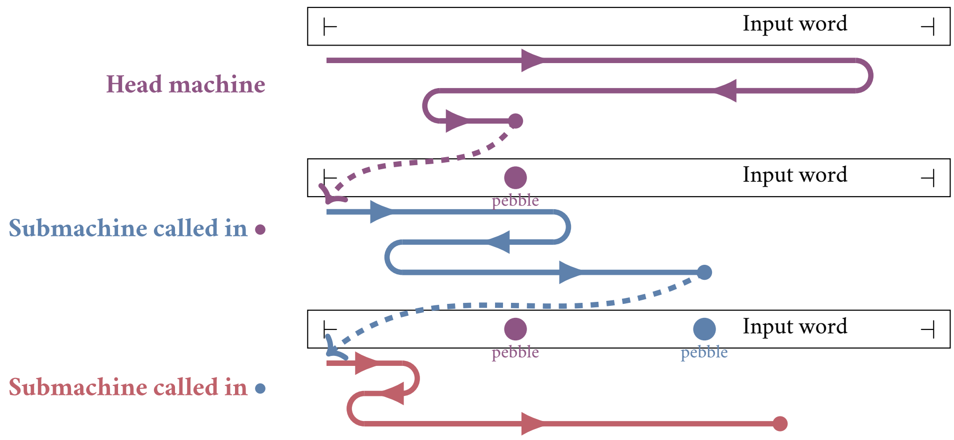


Figure 1.37: Behavior of a 3-pebble transducer that calls submachines.

Example 1.38 (Blind square and square)

- The function blind-square: $A \to A \uplus \{\#\}$ mapping u to $(u\#)^{|u|}$ can be computed by a 2-pebble transducer of shape $\mathscr{T}\langle\mathscr{T}'\rangle$. The head $\mathscr{T}$ calls $\mathscr{T}'$ on each position $1 \leqslant i \leqslant |u|$ of $u \in A^*$, and $[\![\mathscr{T}']\!](u{\bullet}i) = [\![\mathscr{T}']\!](u{\bullet}i) = u\#$ (the underlining is not used by $\mathscr{T}'$).
- The function square: $A \to A \uplus \{\#\}$ mapping u to $(u{\bullet}1)\#\cdots(u{\bullet}|u|)\#$ can be computed by a 2-pebble transducer similar to that of blind-square (but now using the underlining).

Example 1.39 (Prefixes)

The function prefixes: $A^* \to (A \uplus \{\#\})^*, u \mapsto u[1{:}1]\#u[1{:}2]\#\cdots\#u[1{:}|u|]\#$ can be computed by a 2-pebble transducer which makes a nested call in each position of the input.

In the following, we use the term *pebble transducer* to denote a k-pebble transducer for some $k \geqslant 1$.

Definition 1.40 (Polyregular functions)

The class of *polyregular functions* is the class of functions computed by pebble transducers.

Observe that for $1 \leqslant \ell \leqslant k$, an ℓ-pebble transducer can be simulated by a k-pebble transducer.

1.3.2 Robustness and variants of the model

We describe here several variants of the k-pebble transducer model and explain informally why they have the same expressiveness as k-pebble transducers for all $k \geqslant 1$.

1.3.2.1 Lookarounds. It is natural to ask what happens when using normalized 2DT with lookarounds as the submachines of a k-pebble transducer. This enhancement does not increase expressiveness, since lookarounds can be removed on 2DT while preserving the origin semantics (Theorem 1.30), thus preserving the semantics of pebble transducers. Lookarounds will be useful in our proofs.

1.3.2.2 Non-total transducers. Another possibility is to allow the submachines to compute non-total functions. This yields an alternative semantics of pebble transducers: the output is defined if and only if the head has an accepting run and the calls to the submachines along this run are defined.

One can show by induction on $k \geqslant 1$ that a k-pebble transducer with non-total submachines computes a function whose domain is a regular language (we use lookarounds in the induction step to detect if a given call either will fail or accept and in this case produce an output). Thus one can build a "classical" k-pebble transducer (i.e. which follows Definition 1.36) which computes an extension of the function and produces a distinguished symbol if the input is not in the original domain.

1.3.2.3 Side effects. As a generalization of non-totality, one can consider that when a submachine is called, it modifies the inner state of its parent (for instance, we can add a specific function which maps the final states of the child to states of the parent). Once more, one can show that this model does not provide additional expressiveness. The proof sketch would be similar to that of Section 1.3.2.2.

1.3.2.4 Output in the inner nodes. In our model, the submachines that label the inner nodes are only allowed to call their children. One could allow them to directly produce portions of the output, by adding B to their output alphabet. Such a feature can be simulated within the "classical" model, by adding specific descendants nodes which produce the constant function $u \mapsto b$ for $b \in B$.

1.3.2.5 Undistinguished pebbles. In Figure 1.35, the leaf (red) submachines use $A \times \{0,1\}^2$ as input alphabet. Informally, this means that the submachines are able to distinguish the position of the first call (first $\{0,1\}$) from the positions of the second one (second $\{0,1\}$). One can define an alternative model where any submachine (except the head which has input alphabet A) has input alphabet $A \times \{0,1\}$ and where all the calling positions are only marked with 1 (even if there were two calls in the same position). This means that submachine is only able to see, in a given position, if some call was done in this position. Any "classical" k-pebble transducer can be transformed in a k-pebble transducer of this shape, by encoding the lost information within the tree structure (using many more submachines).

1.3.2.6 Definitions from the literature. Finally, let us compare our pebble transducers to the historical definitions of [EM02, Boj18]. These papers define a k-pebble transducer as a single 2DT which drops at most k pebbles on its input, while following a stack policy. It is easy to see that this model is equivalent to ours, when allowing non-total transducers, side effects and outputs in the inner nodes. It follows from Sections 1.3.2.2 to 1.3.2.4 that both models are equivalent (ours is syntactically more restrictive).

1.3.3 Basic properties of polyregular functions

First of all, let us claim that polyregular functions preserve regular languages by inverse image. As a consequence, this result also holds for regular, rational and sequential functions. Proposition 1.41 is stated explicitly in [Boj18, Theorem 1.7], however the key argument (that pebble automata recognize no more than regular languages) was already known since [EH99].

Proposition 1.41 (Regular pre-images)

Let $f\colon A^* \to B^*$ be a polyregular function and $L \subseteq B^*$ be a regular language. Then the language $f^{-1}(L) \subseteq A^*$ is (effectively) regular.

Remark 1.42 (Direct images)

This result is completely false for direct images, even for regular functions. Indeed, it is easy to build a 2DT which maps a word of shape $(ab)^n$ to $a^n b^n$ and produces $\#$ otherwise.

In practice, Proposition 1.41 enables to check whether a given polyregular function f fits a specification of shape $u \in L_{\mathsf{in}} \Rightarrow f(u) \in L_{\mathsf{out}}$ for regular languages $L_{\mathsf{in}} \subseteq A^*$ and $L_{\mathsf{out}} \subseteq B^*$. Indeed, checking this property is equivalent to checking that $f^{-1}(B^* \smallsetminus L_{\mathsf{out}}) \cap L_{\mathsf{in}} = \varnothing$.

1.3.3.1 Composition and decomposition. Closure under composition of polyregular functions was first shown in [EM02, Theorem 2]. An optimal number of pebble layers for the composition (second part of Theorem 1.43) was later given in [Eng15, Theorem 11], see also [Boj18, Theorem 2.6].

Theorem 1.43 (Composition of polyregular functions)

The class of polyregular functions is (effectively) closed under composition.

If $f\colon A^* \to B^*$ is computed by a k-pebble transducer and $g\colon B^* \to C^*$ by a ℓ-pebble transducer, then one can build a $(k\ell)$-pebble transducer which computes $g \circ f$.

Remark 1.44 (Optimality of composition)

Theorem 1.43 provides in general an optimal number of nested layers. Indeed, if $P \in \mathbb{N}[X]$ (resp. $Q \in \mathbb{N}[X]$) is a polynomial with nonnegative integer coefficients, of degree k (resp. ℓ), then $1^n \mapsto 1^{P(n)}$ (resp. $1^n \mapsto 1^{Q(n)}$) can be computed by a k- (resp. ℓ-) pebble transducer but their composition $1^n \mapsto 1^{P(Q(n))}$ cannot be computed using only $k\ell{-}1$ nested layers.

As a “decomposition” result, we claim that the function square of Example 1.38 contains the seeds of the expressiveness of polyregular functions with respect to regular functions. Theorem 1.45 follows from [Boj18, Section 6]. It can be seen as an analogue of Theorem 1.32.

Theorem 1.45 (Decomposition of polyregular functions)

A function is polyregular if and only if can be written as a composition of regular functions and square functions. The conversions are effective.

1.3.3.2 Equivalent formalisms. It follows from [BKL19, Theorem 7] that polyregular functions are exactly the functions computed by the logical model of *monadic second-order interpretations* (MSO interpretations for short), which roughly extends the aforementioned MSO transductions. Informally, a *k-dimensional MSO interpretation* is a collection of MSO formulas with $2k$ free first-order variables, whose semantics describes how to encode of the output word as k-tuples of positions of the input word. As such, it can be seen as a k-pebble transducer which moves its pebbles without a stack discipline, but where the transitive closure of the moves is MSO definable (see the discussion that ends [BKL19, Section 2.2]). However, for a fixed $k \geqslant 2$, k-pebble transducers compute a strict subclass of the functions described by k-dimensional MSO interpretations (see [Boj22, Section 6]).

Several other equivalent formalisms have been introduced, among them an imperative programming language named *for transducers* [Boj18, Section 3], a functional programming language in the spirit of λ-calculus [Boj18, Section 4], and the functions definable in a specific type system [Boj23a].

1.3.3.3 Decision problems: class membership and equivalence. Recall that if $f\colon A^* \to B^*$ is regular (i.e. computed by a 1-pebble transducer), then $|f(u)| = \mathcal{O}(|u|)$. In fact, this linear bound completely characterizes regular functions among the polyregular ones, as stated in Theorem 1.46. This result is claimed in [Boj22, Example 11] (see also [Boj23b, Theorem 2.3]). As pointed out in [Boj22, footnote 6], this result is also a consequence of [EIM21, Corollary 45] which studies tree transducers.

Theorem 1.46 (Polyregular $\to$ Regular)

A polyregular function $f\colon A^* \to B^*$ is regular if and only if $|f(u)| = \mathcal{O}(|u|)$. This property is decidable. If it holds, one can build a 2DT which computes f.

A major open question for polyregular functions is decidability of their equivalence problem. To our knowledge, the first mention of this question over finite words lies in [Eng15, Section 6]. We shall present in Chapters 4 and 5 two subclasses of polyregular functions for which equivalence is decidable.

Open question 1.47 (Equivalence of polyregular functions)

Given two polyregular functions $f, g\colon A^* \to B^*$, is it decidable whether $f = g$?

1.3.4 Asymptotic growth and optimization

It is easy to observe that if f is computed by a k-pebble transducer, then $|f(u)| = \mathcal{O}(|u|^k)$. Following Theorem 1.46, one could conjecture that the least possible $k \geqslant 1$ such that a polyregular function f can be computed by an k-pebble transducer is the least possible $k \geqslant 1$ such that $|f(u)| = \mathcal{O}(|u|^k)$. This result was claimed as the main theorem of a LICS 2020 paper by Lhote[8]. However, it was disproven one year later by Bojańczyk in [Boj22, Theorem 6.3], after a discussion with Kiefer, Nguyên, Pradic and the author of this manuscript. Hence, Lhote's paper contains an unrecoverable error.

Theorem 1.48 (Quadratic growth can require 3 layers)

The function inner-squaring$\colon u_1\#\cdots\#u_n \mapsto (u_1\#)^n \cdots (u_n\#)^n$ can be computed by a 3-pebble transducer and is such that $|\text{inner-squaring}(u)| = \mathcal{O}(|u|^2)$.

However, inner-squaring cannot be computed by a 2-pebble transducer.

Theorem 1.48 was also re-proven in [KNP23, Section 2] using elementary techniques.

[8]The paper is entitled "Pebble minimization of polyregular functions". We chose not to add it in the bibliography, in order to prevent a quick reader from going to this paper without knowing that it contains an unrecoverable error.

1.3.4.1 Minimal nesting depth and asymptotic growth are not related. As a mitigation, it is natural to ask whether there exists $k \geqslant 3$ such that any polyregular function f with $|f(u)| = \mathcal{O}(|u|^2)$ can be computed by a k-pebble transducer. Bojańczyk gives in [Boj23b, Section 3] a negative answer to this question by studying the family of functions $\mathsf{alternating\text{-}square}_k$ for $k \geqslant 2$, that we describe in the next paragraphs. Other counterexamples were given in [KNP23, Section 4] using different proof techniques, adapted from the arguments of [EM02] which study the image languages of pebble transducers.

In the following, A denotes an alphabet. We write $\langle t_1 \rangle \cdots \langle t_n \rangle$ to denote a finite tree whose root is not labelled, and whose subtrees are $t_1, \dots, t_n$. We build by induction on $k \geqslant 1$ the set Trees^A_k:

- Trees^A_1 is A^*;
- for $k \geqslant 2$, Trees^A_k is the set of trees $\langle t_1 \rangle \cdots \langle t_n \rangle$ where $t_1, \dots, t_n \in \mathsf{Trees}^A_{k-1}$.

In other words, Trees^A_k is the set of *complete* trees of *height* k whose leaves are labelled by words of A^* and whose inner nodes have no labels. Using the notation $\langle \cdots \rangle$, let us observe that Trees^A_k (for k fixed) can be seen as a regular word language over the alphabet $A \uplus \{\langle, \rangle\}$.

Example 1.49 (Alternating square)

The function $\mathsf{alternating\text{-}square}_1 \colon \mathsf{Trees}^A_2 \to \mathsf{Trees}^{A \uplus \{\#\}}_3$ takes as input a tree $\langle u_1 \rangle \langle u_2 \rangle \cdots \langle u_n \rangle$ and outputs the tree $\langle \langle u_1 \# u_1 \rangle \langle u_1 \# u_2 \rangle \cdots \langle u_1 \# u_n \rangle \rangle \cdots \langle \langle u_n \# u_1 \rangle \langle u_n \# u_2 \rangle \dots \langle u_n \# u_n \rangle \rangle$. In other words, the output leaves are the pairs of the input leaves, ordered lexicographically. This function can be computed by a 3-pebble transducer which uses its two first layers to mark which pair are going to be produced, and the last layer to indeed output this pair.

More generally, we define the function $\mathsf{alternating\text{-}square}_k \colon \mathsf{Trees}^A_{k+1} \to \mathsf{Trees}^{A \uplus \{\#\}}_{2k+1}$ for $k \geqslant 1$. It will output a tree whose leaves labels are tuples $u \# v$ for $u, v \in A^*$ leaves of the input tree, but the ordering of these tuples is very specific. The function $\mathsf{alternating\text{-}square}_2$ is described by Algorithm 1.50.

Algorithm 1.50: Computing the $\mathsf{alternating\text{-}square}_2$ function

Function $\mathsf{alternating\text{-}square}_2(u)$
  $u \in A^*$ represents a tree of height $k{+}1$
  $i_0 := j_0 :=$ root node of u
  for i_1 **ranging from left to right on the children of** i_0 **do**
    Output $\langle$
    for j_1 **ranging from left to right on the children of** j_0 **do**
      Output $\langle$
      for i_2 **ranging from left to right on the children of** i_1 **do**
        Output $\langle$
        for j_2 **ranging from left to right on the children of** j_1 **do**
          /* Now i_2 and j_2 are leaves which belong to A^* */
          Output $\langle i_2 \# j_2 \rangle$
        end
        Output $\rangle$
      end
      Output $\rangle$
    end
    Output $\rangle$
  end

For $k \geqslant 1$, the function $\mathsf{alternating\text{-}square}_k$ are is a mere generalization of Algorithm 1.50 which has nested "for" loops over indices $i_1, j_1, i_2, j_2, \dots, i_k, j_k$. Note that $|\mathsf{alternating\text{-}square}_k(u)| = \mathcal{O}(|u|^2)$

since the output is a tree of bounded height and each pair of input leaves occurs exactly once. Furthermore, it can be computed by a $(2k+1)$-pebble transducer which uses its $2k$ first layers to simulate the "for" loops, and the $(2+1)$-th layer to output the marked pair of leaves. However, $2k$ layers do not suffice, as claimed in Theorem 1.51 which originates from [Boj23b, Section 3].

Theorem 1.51 (Quadratic growth can require k layers)

Let $k \geqslant 1$. The function $\mathsf{alternating\text{-}square}_k$ can be computed by a $(2k+1)$-pebble transducer and is such that $|\mathsf{alternating\text{-}square}_k(u)| = \mathcal{O}(|u|^2)$.

However, $\mathsf{alternating\text{-}square}_k$ cannot be computed by a $2k$-pebble transducer.

In other words, the minimal number ℓ of nested layers required to compute a polyregular function does not only depend on its asymptotic growth, but also on the (word) combinatorics of its output. As a consequence, we believe that optimizing the number of layers is currently a rather hard problem.

Open question 1.52 (Optimization of pebble transducers)

Given a polyregular function $f : A^* \to B^*$ and $k \geqslant 2$, is it decidable whether f can be computed by some k-pebble transducer?

1.3.4.2 Positive results. In Chapters 3 and 4, we study three restricted variants of pebble transducers (namely, blind pebble transducers, last pebble transducers and marble transducers) for which the minimal number k of nested layers required to compute a function f is exactly the least k such that $|f(u)| = \mathcal{O}(|u|^k)$. In Chapter 5, we shall see that the result also holds for pebble transducers whose output values lie in $\mathbb{N}$ (i.e. unary output[9]) or in $\mathbb{Z}$. We leverage these results to provide algorithms for minimizing the number of nested layers used in such machines, i.e. an automated way to optimize string programs.

Interestingly, asymptotic growth is nevertheless connected to logics: a polyregular function f can be described by a k-dimensional monadic second-order interpretation if and only if $|f(u)| = \mathcal{O}(|u|^k)$ (see [Boj22, Theorem 6.1] and [Boj23b, Theorem 2.3]). The proof techniques for this result are related to those of Chapters 2, 3, 5 and 6 and rely on factorization forests[10].

[9] In the case of unary outputs, pebble and marble transducers turn out to be equivalent (cf. Chapter 5) and the result is therefore a consequence of the aforementioned result for marble transducers.

[10] A different proof has been obtained by Lhote in an unpublished work. The techniques used are close to those of Chapter 4.

Chapter 2

From monoid morphisms to factorization forests

La Nature est un temple où de vivants piliers
Laissent parfois sortir de confuses paroles;
L'homme y passe à travers des forêts de symboles
Qui l'observent avec des regards familiers.

Charles Baudelaire, « Correspondances », *Les Fleurs du mal*

This chapter can be understood as a toolbox for unravelling the behavior of two-way transducers, whose study is at the heart of this manuscript. The notions and results presented here will be re-used in particular in Chapters 3, 5, 6 and 9. A reader who is in hurry is invited to skip the current chapter, and to use the hyperlinks for going back when needed to its definitions and its results.

In Section 2.1, we first present folklore results concerning the transition monoids of two-way transducers. We apply these tools in Section 2.2 to re-prove two known results for two-way transducers. The latter should be considered as a warm-up towards the involved proof techniques for various decision problems which are studied in Chapters 3, 5 and 6.

In Section 2.3, we recall the main properties of Simon's celebrated factorization forests [Sim90]. Then, we describe how they can be used as a versatile tool for studying two-way transducers and pebble transducers, following the techniques introduced in [Dou21, Dou22, Dou23].

2.1 Monoids and crossing sequences of two-way transducers

The goal of this section is to recall standard tools for studying the runs of two-way transducers. Given a factor of the input, the basic idea is to describe all runs which move in a left-to-left, left-to-right, right-to-right and right-to-left fashion on this factor, as depicted in Figure 2.1. In the rest of Section 2.1, we fix a 2DT denoted $\mathscr{T} = (A, B, Q, q_0, F, \delta, \lambda)$. We let $\overleftarrow{Q} := \{\overleftarrow{q} \mid q \in Q\}$ and $\overrightarrow{Q} := \{\overrightarrow{q} \mid q \in Q\}$.

2.1.1 Transition morphisms of two-way transducers

Our first goal is to build the following functions, which describe the behavior of $\mathscr{T}$ along words:

- the *extended transition function* $\delta^* \colon (\overrightarrow{Q} \uplus \overleftarrow{Q}) \times (A \uplus \{\vdash, \dashv\})^* \rightharpoonup \overrightarrow{Q} \uplus \overleftarrow{Q}$;
- and the *extended output function* $\lambda^* \colon (\overrightarrow{Q} \uplus \overleftarrow{Q}) \times (A \uplus \{\vdash, \dashv\})^* \rightharpoonup B^*$ with same domain as δ.

Let $u \in (A \uplus \{\vdash, \dashv\})^*$ and $q \in Q$. Intuitively, $\delta(\overrightarrow{q}, u) = \overrightarrow{p}$ and $\lambda(\overrightarrow{q}, u) = \alpha$ means that the longest finite run labelled by u which starts in q in the leftmost position of u will eventually leave u "on the right" in state p, and the output produced along this run is α. This intuition is depicted in Figure 2.1. The ideas behind this definition originate from [She59, Proof of Theorem 2] for two-way automata.

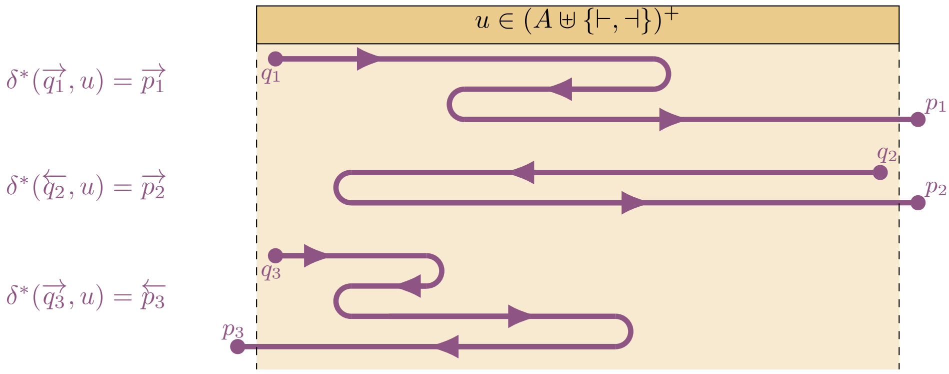


Figure 2.1: Extended transition function of a 2DT.

Formally, we let $\delta(\_, \varepsilon)$ be the identity function and $\lambda(\_, \varepsilon)$ be the constant function $v \mapsto \varepsilon$. For $q \in Q$ and $u \in (A \uplus \{\vdash, \dashv\})^+$, consider the longest run of $\mathscr{T}$ labelled by u which begins in $(q, 1)$, then:

- if this run is infinite (it loops inside u), then $\delta^*(\overrightarrow{q}, u) := \perp$ (undefined);
- otherwise the run is finite and denoted $\mathsf{maxi\text{-}run}(\overrightarrow{q}, u) := (q_1, i_1) \to \cdots \to (q_n, i_n)$. Then:
  - if $\delta(q_n, u[i_n]) = \perp$ (undefined, the machine gets blocked), then $\delta^*(\overrightarrow{q}, u) := \perp$;
  - if $\delta(q_n, u[i_n]) = (p, \lhd)$ (necessarily $i_n = 1$), then we let $\delta^*(\overrightarrow{q}, u) := \overleftarrow{p}$;
  - if $\delta(q_n, u[i_n]) = (p, \rhd)$ (necessarily $i_n = |u|$), then we let $\delta^*(\overrightarrow{q}, u) := \overrightarrow{p}$.

If $\delta^*(\overrightarrow{q}, u) \neq \perp$, we let $\lambda^*(\overrightarrow{q}, u)$ be the concatenation $\lambda(q_1, u[i_1]) \cdots \lambda(q_n, u[i_n])$. We build the longest finite run $\mathsf{maxi\text{-}run}(\overleftarrow{q}, u)$ in a similar way by starting in configuration $(q, |u|)$ instead of $(q, 1)$, and the functions $\delta^*(\overleftarrow{q}, u)$ and $\lambda^*(\overleftarrow{q}, u)$ accordingly.

Now we are ready to present the classical notion of *transition morphism* of a two-way transducer. The reader is invited to consult e.g. [Car14, Théorème 3.84] or [DGK18, Section 2.5] for more details. On purpose, we shall not define the transition morphism over $(A \uplus \{\vdash, \dashv\})^*$ but only over A^*. Indeed, the end-markers $\vdash$ and $\dashv$ only play a "fixed" role that will be taken into account in Definition 2.5.

Proposition-Definition 2.2 (Transition morphism, transition monoid)

Let μ be the function mapping $u \in A^*$ to the function $\delta^*(\_, u) \colon \overrightarrow{Q} \uplus \overleftarrow{Q} \rightharpoonup \overrightarrow{Q} \uplus \overleftarrow{Q}$. The set $\mathbb{T} := \mu(A^*)$ can be equipped by an operation which makes $\mu \colon A^* \to \mathbb{T}$ be a monoid morphism. We say that μ (resp. $\mathbb{T}$) is the *transition morphism* (resp. the *transition monoid*) of $\mathscr{T}$.

Note that μ is the currying of δ^* over its second argument (which is in A^*). We shall not explicitly describe the monoid product of $\mathbb{T}$, but it intuitively describes how the runs of $\mathscr{T}$ can be composed when concatenating words (checking that it indeed defines a monoid is easy but tedious).

Remark 2.3 (Surjectivity)

The morphism $\mu\colon A^* \to \mathbb{T}$ is a surjective by construction of $\mathbb{T}$.

2.1.2 Crossing sequences and productions

From now on, we assume that $\mathscr{T}$ is normalized, thus it has an accepting run over any input $\vdash u \dashv$ with $u \in A^*$. Given a set of positions $I \subseteq [1{:}|u|]$, we define the *crossing sequence* over I in u as the portions of the accepting run of $\mathscr{T}$ over $\vdash u \dashv$ whose positions are in I, as depicted in Figure 2.4.

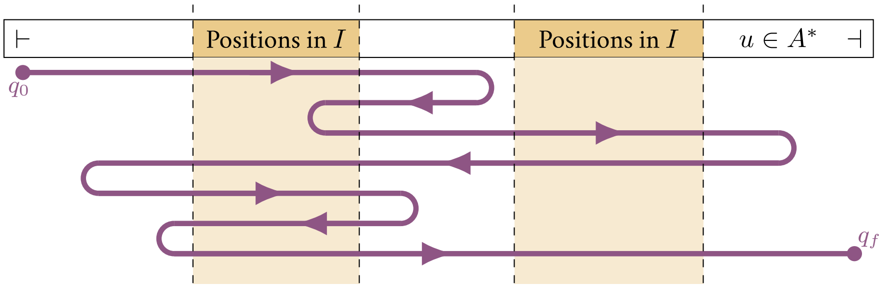


Figure 2.4: Crossing sequence over I of a normalized 2DT.

Definition 2.5 (Crossing sequence)

Let $u \in A^+$, $I \subseteq [1{:}|u|]$ and $(q_1, i_1) \to (q_2, i_2) \to \cdots \to (q_n, i_n)$ be the unique accepting run of $\mathscr{T}$ labelled by $\vdash u \dashv$. The *crossing sequence* of $\mathscr{T}$ over I in u is defined as:

$$\mathsf{cross}^u_{\mathscr{T}}(I) := (q_j, i_j)_{1 \leqslant j \leqslant n \text{ such that } i_j \in I}.$$

We are ready to define the *production* along I, which corresponds to the output produced along the corresponding crossing sequence. Given a word $\alpha \in B^*$, we let $\mathsf{parikh}(\alpha)$ be its *Parikh image*, that is the *multiset* of the letters of v. For instance $\mathsf{parikh}(abac) = \{\!\!\{a, a, b, c\}\!\!\}$. Considering the Parikh image is useful when dealing only with the size of the output, which was our goal in Section 1.3.4.

Definition 2.6 (Production of a two-way transducer)

Let $u \in A^+$, $I \subseteq [1{:}|u|]$ and $\mathsf{cross}^u_{\mathscr{T}}(I) = (q_1, i_1), (q_2, i_2), \dots, (q_n, i_n)$. The *production* of $\mathscr{T}$ over I in u is defined as $\mathsf{prod}^u_{\mathscr{T}}(I) := \mathsf{parikh}(\lambda(q_1, u[i_1]) \cdots \lambda(q_n, u[i_n]))$.

Given a multiset $\mathfrak{M}$, we let $|\mathfrak{M}|_s$ denote the number of occurrences of the element s in $\mathfrak{M}$. We claim that productions enjoy an additive property, which can easily be checked by looking at Figure 2.4. This property will be especially useful when slicing an input word along a factorization forest in Section 2.3.

Claim 2.7 (Productions are additive)

Let $u \in A^+$ and $I, J \subseteq [1{:}|u|]$ be disjoint. Then $\mathsf{prod}^u_{\mathscr{T}}(I \uplus J) = \mathsf{prod}^u_{\mathscr{T}}(I) \uplus \mathsf{prod}^u_{\mathscr{T}}(J)$.

In particular, if $f\colon A^* \to B^*$ is the function computed by $\mathscr{T}$ and $I_1, \dots, I_n$ is a partition of $[1{:}|u|]$, then $\mathsf{parikh}(f(u)) = \mathsf{prod}^u_{\mathscr{T}}(I_1) \uplus \cdots \uplus \mathsf{prod}^u_{\mathscr{T}}(I_n)$.

Finally, we focus on the case when I is an interval (i.e. $I = u[i{:}j]$ for some $1 \leqslant i \leqslant j \leqslant |u|$) through the notion of μ-context. We also establish a first relation between transition morphism and productions.

Definition 2.8 (Word-context, μ-context)

Given a triple $(v_0, u, v_1) \in A^* \times A^+ \times A^*$, we say it describes a *word-context* written $v_0 \lfloor u \rfloor v_1$.

Let $\mu\colon A^* \to \mathbb{M}$ be a monoid morphism. Given a triple $(m_0, u, m_1) \in \mathbb{M} \times A^+ \times \mathbb{M}$, we say it describes a *μ-context* which is written $m_0 \lfloor u \rfloor m_1$

Let us overload the definition of our production operator. If $v_0 \lfloor u \rfloor v_1$ is a word-context, we let $\mathsf{cross}_{\mathscr{T}}(v_0 \lfloor u \rfloor v_1) \coloneqq \mathsf{cross}_{\mathscr{T}}^{v_0 u v_1}([|v_0|{+}1{:}|v_0 u|])$ and $\mathsf{prod}_{\mathscr{T}}(v_0 \lfloor u \rfloor v_1) \coloneqq \mathsf{prod}_{\mathscr{T}}^{v_0 u v_1}([|v_0|{+}1{:}|v_0 u|])$. Now, we show that the production over the word-context $v_0 \lfloor u \rfloor v_1$ only depends on the abstraction $\mu(v_0) \lfloor u \rfloor \mu(v_1)$ when $\mu\colon A^* \to \mathbb{T}$ is the transition monoid of $\mathscr{T}$.

Proposition-Definition 2.9 (Production in a context)

Let $\mu\colon A^* \to \mathbb{T}$ be the transition morphism of $\mathscr{T}$ and $m_0 \lfloor u \rfloor m_1$ be a μ-context. For all word-context $v_0 \lfloor u \rfloor v_1$ such that $\mu(v_0) = m_0$ and $\mu(v_1) = m_1$, the value $\mathsf{prod}_{\mathscr{T}}(v_0 \lfloor u \rfloor v_1)$ is the same.

We define $\mathsf{prod}_{\mathscr{T}}(m_0 \lfloor u \rfloor m_1)$ as this value.

Proof idea. By Claim 2.7, we only have to show that for all $a \in A$ and word-contexts $v_0 \lfloor a \rfloor v_1$, $v_0' \lfloor a \rfloor v_1'$ with $\mu(v_0) = \mu(v_0')$ and $\mu(v_1) = \mu(v_1')$, we have $\mathsf{prod}_{\mathscr{T}}(v_0 \lfloor a \rfloor v_1) = \mathsf{prod}_{\mathscr{T}}(v_0 \lfloor a \rfloor v_1)$. We show by induction on $j \geqslant 1$ that if $(q_1, |v_0{+}1|), \dots, (q_j, |v_0{+}1|) \sqsubseteq \mathsf{cross}_{\mathscr{T}}(v_0 \lfloor a \rfloor v_1)$ (recall that $\sqsubseteq$ is used to denote a prefix), then $(q_1, |v_0'{+}1|), \dots, (q_j, |v_0'{+}1|) \sqsubseteq \mathsf{cross}_{\mathscr{T}}(v_0' \lfloor a \rfloor v_1')$. The base case follows since $q_1 = \delta^*(q_0, v_0) = \delta^*(q_0, v_0')$ by definition of the transition morphism and since $(q_1, |v_0{+}1|)$ is the first visit in $|v_0{+}1|$ in $v_0 a v_1$. The induction step is similar, depending on whether the transition $\delta(q_j, a)$ moves right or left. Finally, we apply λ. ◀

Observe that $\mathsf{prod}_{\mathscr{T}}(m_0 \lfloor u \rfloor m_1)$ is well-defined for all $m_0, m_1 \in \mathbb{T}$. Indeed, since $\mu\colon A^* \to \mathbb{T}$ is surjective (see Remark 2.3), one can find a word-context such that $\mu(v_0 \lfloor u \rfloor v_1) = m_0 \lfloor u \rfloor m_1$.

2.2 Applications: pumping lemmas for two-way transducers

As a ludic interlude between the rather arid Sections 2.1 and 2.3, we apply our tools for deciding if $f(A^*)$ is finite when $f\colon A^* \to B^*$ is a regular function. We leverage the technique to provide a well-known "pumping lemma" for 2DT. Our notions may seem to be over-engeenering for these simple applications, but they provide a warm-up towards the difficult proofs of the next chapters.

2.2.1 Deciding if a regular function has finite image

We first introduce the notion of *μ-K-iterator*, which roughly consists in a μ-context whose word can be duplicated without breaking its structure. This notion will be generalized in Section 5.3.3 when dealing with counting transducers, which are roughly pebble transducers with commutative output.

Definition 2.10 (Iterator)

Let $\mu\colon A^* \to \mathbb{M}$ be a monoid morphism and $K \geqslant 0$. We say that a μ-context $m_0e\lfloor u \rfloor em_1$ is a *μ-K-iterator* if $m_0, e, m_1 \in \mathbb{M}$, $u \in A^+$, $|u| \leqslant K$ and $e = \mu(u)$ is an idempotent[1].

As their name suggest, μ-K-iterators can be "iterated" in rather a smooth way.

Claim 2.11 (Pumping iterators)

Let $f\colon A^* \to B^*$ be computed by a normalized 2DT $\mathscr{T}$ with transition monoid $\mu\colon A^* \to \mathbb{T}$. Let $m_0e\lfloor u \rfloor em_1$ be a μ-K-iterator and let $v_0\lfloor u \rfloor v_1$ be such that $\mu(v_0) = m_0e$ and $\mu(v_1) = em_1$. There exists $N \geqslant 0$ such that for all $X \geqslant 0$, $|f(v_0u^Xv_1)| = |\mathsf{prod}_{\mathscr{T}}(m_0e\lfloor u \rfloor em_1)|X + N$.

Proof. Immediate by Claim 2.7 and Proposition-Definition 2.9. ◀

In order to use Claim 2.11, one needs to find μ-K-iterators within arbitrary (large enough) input words. This is the purpose of Claim 2.12, which is a first step towards Section 2.3. This result easily follows from Ramsey's theorem, see e.g. [Jec21] for a generalization and precise bounds.

Claim 2.12 (Towards factorization forests)

One can build a computable $R\colon \mathbb{N} \to \mathbb{N}$ such that the following holds if $\mu\colon A^* \to \mathbb{M}$ is a morphism into a finite monoid. For all $w \in A^*$ such that $|w| \geqslant R(|\mathbb{M}|)$, there exist $w_0, w_1 \in A^*$ and $t_0, u, t_1 \in A^+$ such that $w = w_0t_0ut_1w_1$ and $\mu(t_0) = \mu(u) = \mu(t_1)$ is idempotent.

Using Claims 2.11 and 2.12, one can obtain a decidable characterization (in terms of productions) of the 2DT which compute a function whose image is finite. The same methodology will be applied (in a more complex setting) for solving decision problems in Chapters 3, 5 and 6.

Proposition 2.13 (Regular functions with finite image)

Let $f\colon A^* \to B^*$ be computed by a normalized 2DT with transition monoid $\mu\colon A^* \to \mathbb{T}$. Then $f(A^*)$ is finite if and only if $|\mathsf{prod}_{\mathscr{T}}(m_0e\lfloor u \rfloor em_1)| = 0$ for all μ-$R(|\mathbb{T}|)$-iterator $m_0e\lfloor u \rfloor em_1$.

As a consequence, one can decide if a regular function has finite image.

Proof idea. The "if" direction follows from Claim 2.11 and the fact that μ is surjective. For the converse, we use Claims 2.11 and 2.12 to show that, within long enough words, factors can be removed without changing the output size. The property is decidable since there is only a finite number of μ-$R(|\mathbb{T}|)$-iterators, and their productions can effectively be determined. ◀

2.2.2 A pumping lemma for regular functions

In Claim 2.11, we have somehow described a "commutative" pumping lemma for 2DT, since we only considered the length of the output. Several results of this shape will be stated and used in Part II. In the current section, we explain what happens when really considering the word output.

The first step is to get with Lemma 2.15 a fine-grained understanding of the idempotents in the transition monoid. This classical result roughly corresponds to [Boj18, Sublemma 6.8.2]. A variant over infinite words and under weaker hypotheses will be discussed in Lemma 9.48.

[1] Recall that an element $e \in \mathbb{M}$ is said to be *idempotent* if $e \cdot e = e$.

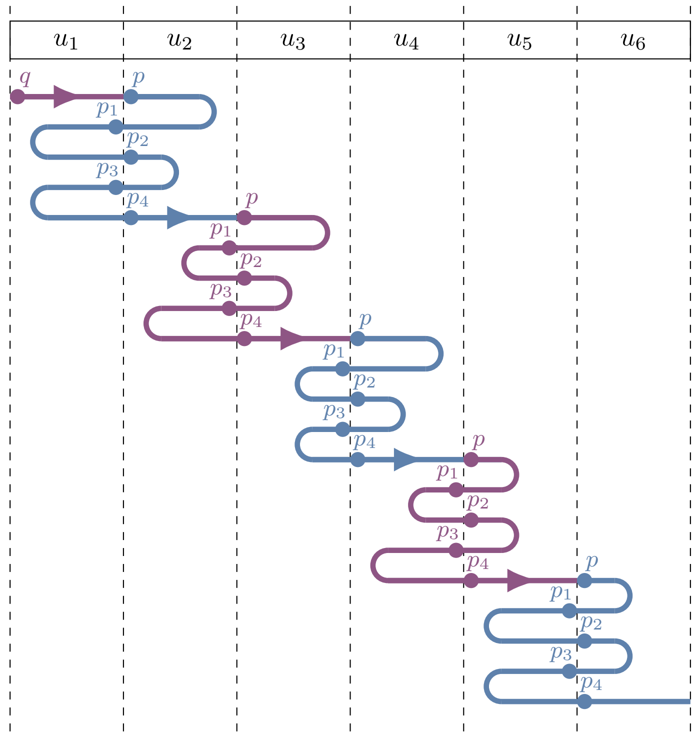


Figure 2.14: Shape of a run along a block of idempotents factors.

Lemma 2.15 (Runs in idempotent blocks)

Let $\mathscr{T} = (A, B, Q, q_0, F, \delta, \lambda)$ be a 2DT with transition monoid $\mu \colon A^* \to \mathbb{T}$. Let $e \in \mathbb{T}$ be an idempotent and $u = u_1 \cdots u_n$ be such that $u_i \in A^+$ and $e = \mu(u_i)$ for all $1 \leqslant i \leqslant n$. If $\delta^*(\overrightarrow{q}, u_1) = \overrightarrow{p}$, then maxi-run$(\overrightarrow{q}, u)$ has shape maxi-run$(\overrightarrow{q}, u_1) \to \rho_2 \to \cdots \to \rho_n$ where:

(1) for all $2 \leqslant i \leqslant n$, ρ_i starts in the first configuration of ρ which visits u_i;
(2) for all $2 \leqslant i \leqslant n$, ρ_i begins with a configuration of shape $(p, \_)$ (i.e. it starts in p);
(3) for all $2 \leqslant i \leqslant n$, ρ_i only visits the positions of u_i and u_{i-1} (it cannot go back to u_{i-2}).

Proof. We have $\delta^*(\overrightarrow{q}, u_1) = \overrightarrow{p}$, thus $\delta^*(\overrightarrow{q}, u_1 \cdots u_{i-1}) = \overrightarrow{p}$ for all $2 \leqslant i \leqslant n$. This means that the factor u_i is visited by maxi-run$(\overrightarrow{q}, u)$, and furthermore that this visit starts in state p, giving Items (1) and (2) by defining ρ_i accordingly. For Item (3), let $i \geqslant 3$ (for $i = 2$ the result is obvious), we show that ρ_i only visits u_i and u_{i-1}. First, observe that this run does not visit u_{i+1} by construction of ρ_{i+1}. Second, let us consider the state r seen in the last visit of the first position of u_{i-1} in ρ_{i-1}. Since $\mu(u_{i-1}u_i) = \mu(u_{i-1})$, we have $\delta^*(\overrightarrow{r}, u_{i-1}u_i) = \delta^*(\overrightarrow{r}, u_{i-1}) = \overrightarrow{p}$ (the last equality follows from Item (2), because it describes the beginning of ρ_i). This means that when starting from r in the first position of u_{i-1}, $\mathscr{T}$ will execute the end of ρ_{i-1}, then ρ_i, and it will eventually leave $u_{i-1}u_i$ “by the right”. Hence the run ρ_i stays in $u_{i-1}u_i$, until it goes to u_{i+1} in state p (and this is by construction the beginning of ρ_{i+1}). ◀

The shape of maxi-run$(\overrightarrow{q}, u)$ from Lemma 2.15 is depicted in Figure 2.14. Observe that it must cross the border between each u_i and u_{i+1} a fixed number of times, and that the states visited during this crossing must always be the same (because it meets the same idempotent everywhere).

As a consequence of Lemma 2.15, we obtain an abstract pumping lemma for regular functions. The next result is stated e.g. in [Roz86, Théorème 1][2]. A variant of this statement will be used in Proposition 3.14 for showing a separation result between classes of functions computed by nested 2DT.

Proposition 2.16 (Pumping lemma for regular functions)

Let $f\colon A^* \to B^*$ be a regular function. There exists $N \geqslant 0$ such that the following holds for all $w \in A^*$ with $|w| \geqslant N$. There exist $v_0, v_1 \in A^*$, $u \in A^+$, $n \geqslant 0$, $\alpha_0, \dots, \alpha_n \in B^*$, $\beta_1, \dots, \beta_n \in B^+$ such that $w = v_0 u v_1$ and $f(v_0 u^{X+1} v_1) = \alpha_0 \beta_1^X \alpha_1 \cdots \beta_n^X \alpha_n$ for all $X \geqslant 0$.

Proof. Let $\mu : A^* \to \mathbb{T}$ be the monoid morphism of a 2DT^ω $\mathscr{T} = (A, B, Q, q_0, F, \delta, \lambda)$ which computes f. By Claim 2.12 one can factorize any long enough word $w \in A^*$ under the shape $w = w_0 t_0 u t_1 w_1$ where $t_0, u, t_1 \in A^+$ are such that $\mu(t_0) = \mu(u) = \mu(t_1)$ is idempotent. Consider the word $t_0 u^{X+1} t_1$ for $X \geqslant 0$ and let $q \in Q$ be such that $\delta^*(\overrightarrow{q}, t)$ has shape $\overrightarrow{p}$. By applying Lemma 2.15, one can factor $\text{maxi-run}(\overrightarrow{q}, t_0 u^{X+1} t_1)$ as $\text{maxi-run}(\overrightarrow{q}, t_0) \to \rho_2 \to \cdots \to \rho_{X+2}$ where ρ_2 visits the first u and t_0, ρ_i for $3 \leqslant i \leqslant X{+}3$ visits the $(i{-}1)$-th and the $(i{-}2)$-th u, and ρ_{X+3} visits t_1 and the last u. For all $3 \leqslant i \leqslant X{+}3$, the runs ρ_i are the same (up to position shifting) since they move on the same input uu, starting in the middle in state p. Hence they have the same output $\beta \in B^*$. Furthermore this value does not depend on X. All in all, the output along $\text{maxi-run}(\overrightarrow{q}, t_0 u^{X+1} t_1)$ has shape $\alpha_0 \beta^X \alpha_1$ for some $\alpha_0, \alpha_1 \in B^*$. The result follows by showing that the accepting run of $\mathscr{T}$ on input $\vdash w_0 t_0 u t_1 w_1 \dashv$ can be decomposed as the concatenation of a bounded number of runs of the previous shape. ◀

2.3 Factorization forests

In Section 2.2, we saw that words whose image is an idempotent of the transition monoid are useful to duplicate or remove pieces of an accepting run. We used Ramsey's theorem to show that idempotents occur in long enough words. In this section, we first recall Simon's factorization forest theorem [Sim90, Theorem 6.1] which goes one step further than Ramsey. Roughly, given a word, this result builds a tree structure which factorizes it while exhibiting idempotents "everywhere". Various proofs of this theorem have been given since Simon's, and it yielded several applications such as characterizations of subclasses of regular languages [PW97, BP09, KA10] or string matching algorithms [Boj09, Section 2].

Secondly, we describe the machinery introduced in [Dou21, Dou22, Dou23] in order to iterate idempotent portions of words using factorization forests. These technical results will be used as basic tools in the proofs of Chapters 3, 5 and 6 to deal with the asymptotic growth of polyregular functions.

2.3.1 Simon's theorem

If $\mu : A^* \to \mathbb{M}$ is a morphism into a finite monoid and $u \in A^*$, a *μ-factorization forest* (also called *μ-forest* in the following) of u is an unranked tree structure described in Definition 2.17. Recall that $\langle t_1 \rangle \cdots \langle t_n \rangle$ denotes a finite tree whose root is not labelled, and whose subtrees are $t_1, \dots, t_n$.

Definition 2.17 (Factorization forest)

Let $\mu\colon A^* \to \mathbb{M}$ be a monoid morphism and $u \in A^*$. We say that $\mathcal{F}$ is a *μ-forest* of u if:

[2] More recently, this theorem was re-proven as the main result of [Smi14]. See also [Bas17, Section 3.3].

- either $u = a \in A$ and $\mathcal{F} = a$;
- or $\mathcal{F} = \langle \mathcal{F}_1 \rangle \cdots \langle \mathcal{F}_n \rangle$, $u = u_1 \cdots u_n$ and for all $1 \leqslant i \leqslant n$, $\mathcal{F}_i$ is a μ-forest of $u_i \in A^+$. Furthermore, if $n \geqslant 3$ then $\mu(u) = \mu(u_1) = \cdots = \mu(u_n)$ is an idempotent of $\mathbb{M}$.

Remark 2.18 (Pruning a factorization forest)

If $\langle \mathcal{F}_1 \rangle \cdots \langle \mathcal{F}_n \rangle$ is a μ-forest and $I \subseteq [1{:}n]$, then $\langle \mathcal{F}_i \rangle_{i \in I}$ is also a μ-forest.

We use the standard notions of *node*, *root*, *leaf*, *child*, *sibling*, *descendant*, *ancestor* (defined in a non-strict way: a node is itself one of its ancestors/descendants). We define the *height* of a tree by induction: a leaf has height 1 and the height of $\langle t_1 \rangle \cdots \langle t_n \rangle$ is $1+$ the maximum of the heights of $t_1, \dots, t_n$.

Given a morphism $\mu\colon A^* \to \mathbb{M}$, $d \geqslant 1$ and $u \in A^+$, we let $\mathsf{Forests}^d_\mu(u)$ be the set of all μ-forests of $u \in A^+$ of height at most d. By removing some variables in this definition, we let:

- $\mathsf{Forests}_\mu(u) := \bigcup_{d \geqslant 1} \mathsf{Forests}^d_\mu(u)$;
- $\mathsf{Forests}^d_\mu := \bigcup_{u \in A^+} \mathsf{Forests}^d_\mu(u)$;
- $\mathsf{Forests}_\mu := \bigcup_{d \geqslant 1} \bigcup_{u \in A^+} \mathsf{Forests}^d_\mu(u)$.

A tree $\mathcal{F} \in \mathsf{Forests}_\mu$ can be seen as a word over the alphabet $A \uplus \{\langle, \rangle\}$ (this observation directly follows from Definition 2.17). If $\mathbb{M}$ is finite, it is easy to see that $\mathsf{Forests}^d_\mu$ is a regular language (beware that it is not the case of $\mathsf{Forests}_\mu$). We let the function $\mathsf{word}_\mu\colon \mathsf{Forests}_\mu \to A^+$ be the morphism which removes the letters in $\{\langle, \rangle\}$, i.e. mapping $\mathcal{F} \in \mathsf{Forests}_\mu(u)$ to $u \in A^+$. We define $\mathsf{word}^d_\mu\colon \mathsf{Forests}^d_\mu \to A^+$ as the restriction of word_μ to $\mathsf{Forests}^d_\mu$ for $d \geqslant 1$.

We denote by $\mathsf{Nodes}_\mathcal{F}$ the set of nodes of $\mathcal{F}$. In order to simplify the statements, we identify a node $\mathfrak{t} \in \mathsf{Nodes}_\mathcal{F}$ with the *subtree* rooted in this node. Thus $\mathsf{Nodes}_\mathcal{F}$ can also be seen as the set of subtrees of $\mathcal{F}$, and $\mathcal{F} \in \mathsf{Nodes}_\mathcal{F}$. We say that a node is *idempotent* if is has at least 3 children (recall that it this case, applying $\mu \circ \mathsf{word}_\mu$ to any of its rooted subtrees yields the same idempotent value in $\mathbb{M}$).

Example 2.19 (Factorization forest)

Let $\mathbb{M} := (\{-1, 1, 0\}, \times)$ and $\mu\colon \mathbb{M}^* \to \mathbb{M}$ be the product. We depict in Figure 2.20 a μ-forest $\mathcal{F}$ of the word $(-1)(-1)0(-1)00000000 \in \mathbb{M}^*$. Double lines denote idempotent nodes.

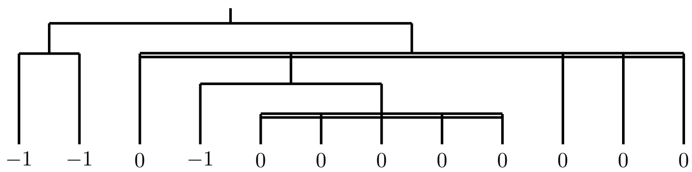


Figure 2.20: A factorization forest $\mathcal{F} \in \mathsf{Forests}_\mu((-1)(-1)0(-1)00000000)$.

Now, we claim that given a fixed morphism $\mu\colon A^* \to \mathbb{M}$ into a finite monoid, there exist $d \geqslant 0$ and a function $\mathsf{forest}_\mu\colon A^+ \to \mathsf{Forests}^d_\mu$ which is a pseudo-inverse of word_μ. In particular, Theorem 2.21 shows the existence a μ-forest of bounded height of all $u \in A^+$, which is the original result of [Sim90] (with $d = 9|\mathbb{M}|$, our $d = 3|\mathbb{M}|$ was obtained later in [Kuf08]). Once this existence is known, it is easy to show that a rational pseudo-inverse of word_μ can be built, see e.g. [Boj09, Lemma 3].

Theorem 2.21 (Rational Simon)

Given a morphism $\mu\colon A^* \to \mathbb{M}$ into a finite monoid, one can build a rational function denoted $\mathsf{forest}_\mu\colon A^+ \to \mathsf{Forests}_\mu^{3|\mathbb{M}|}$ such that $\mathsf{word}_\mu^{3|\mathbb{M}|} \circ \mathsf{forest}_\mu$ is the identity function over A^+.

Remark 2.22 (Sequential Simon)

We shall meet in Section 9.6.1 a variant of factorization forests for which forest_μ can be computed by a sequential function. However, this computation by a simpler model is done at the cost of weakening the structural conditions which define idempotent nodes.

2.3.2 Iterable nodes and skeletons

In this subsection, $\mu\colon A^* \to \mathbb{M}$ denotes a fixed morphism into a finite monoid. Our goal is to describe how a μ-factorization forests enables to partition a word $u \in A^+$, so that subwords can be iterated without changing the global behavior of μ. We follow the definitions introduced in [Dou21, Dou22]; similar ideas are presented in [Boj23b, Section 2] under the formalism of tree grammars.

First, we define *iterable nodes* as the middle children of idempotent nodes. Intuitively, such nodes can be removed or iterated while preserving the μ-forest structure.

Definition 2.23 (Iterable nodes)

Let $u \in A^+$ and $\mathcal{F} \in \mathsf{Forests}_\mu(u)$, we define the set $\mathsf{Iters}_\mathcal{F}$ of *iterable nodes* of $\mathcal{F}$ as follows:

- if $\mathcal{F} = a \in A$ then $\mathsf{Iters}_\mathcal{F} := \varnothing$;
- otherwise if $\mathcal{F} = \langle \mathcal{F}_1 \rangle \cdots \langle \mathcal{F}_n \rangle$, then:

$$\mathsf{Iters}_\mathcal{F} := \{\mathcal{F}_i \mid 2 \leqslant i \leqslant n-1\} \cup \bigcup_{1 \leqslant i \leqslant n} \mathsf{Iters}_{\mathcal{F}_i}.$$

The *skeleton* of a node is built by selecting inductively its "leftmost" and "rightmost" descendants.

Definition 2.24 (Skeleton, frontier)

Let $u \in A^+$, $\mathcal{F} \in \mathsf{Forests}_\mu(u)$ and $\mathfrak{t} \in \mathsf{Nodes}_\mathcal{F}$. We define the *skeleton* of $\mathfrak{t}$, denoted $\mathsf{Skel}_\mathcal{F}(\mathfrak{t})$, by:

- if $\mathfrak{t} = a \in A$ is a leaf, then $\mathsf{Skel}_\mathcal{F}(\mathfrak{t}) := \{\mathfrak{t}\}$;
- otherwise if $\mathfrak{t} = \langle \mathcal{F}_1 \rangle \cdots \langle \mathcal{F}_n \rangle$, then $\mathsf{Skel}_\mathcal{F}(\mathfrak{t}) := \{\mathfrak{t}\} \cup \mathsf{Skel}_\mathcal{F}(\mathcal{F}_1) \cup \mathsf{Skel}_\mathcal{F}(\mathcal{F}_n)$.

The *frontier* of $\mathfrak{t}$ is the set $\mathsf{Fr}_\mathcal{F}(\mathfrak{t}) \subseteq [1{:}|u|]$ containing the positions of u which belong to $\mathsf{Skel}_\mathcal{F}(\mathfrak{t})$ (when seen as leaves of the μ-forest $\mathcal{F}$ over u).

Example 2.25 (Iterable nodes, skeleton, frontier)

The μ-forest of Figure 2.20 is represented again in Figure 2.26. Its iterable nodes are depicted with blue circles. The skeleton of the root node is depicted with red circles.

It is easy to observe if d is fixed, then for all $\mathcal{F} \in \mathsf{Forests}_\mu^d(u)$ and $\mathfrak{t} \in \mathcal{F}$, the size of $\mathsf{Skel}_\mathcal{F}(\mathfrak{t})$ or $\mathsf{Fr}_\mathcal{F}(\mathfrak{t})$ are bounded independently from $\mathcal{F}$ and $\mathfrak{t}$. Indeed, in this case a skeleton can be seen as a binary tree (thus it has bounded branching) of height at most d (thus it has bounded height).

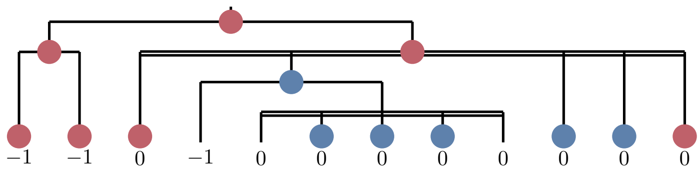


Figure 2.26: Iterable nodes and skeleton of the root for the μ-forest from Figure 2.20.

Now, we claim that the skeletons of the iterable nodes and of the root partition a given forest. As a consequence, we obtain a partition of the positions of the word conform to this forest. The proof of the Claim 2.27 is immediate by induction. Also note that a skeleton or a frontier cannot be empty.

Claim 2.27 (Partition of skeletons)

Let $u \in A^+$ and $\mathcal{F} \in \mathsf{Forests}_\mu(u)$. The set of skeletons $\{\mathsf{Skel}_\mathcal{F}(\mathfrak{t}) \mid \mathfrak{t} \in \mathsf{Iters}_\mathcal{F} \cup \{\mathcal{F}\}\}$ is a partition of $\mathsf{Nodes}_\mathcal{F}$. The set of frontiers $\{\mathsf{Fr}_\mathcal{F}(\mathfrak{t}) \mid \mathfrak{t} \in \mathsf{Iters}_\mathcal{F} \cup \{\mathcal{F}\}\}$ is a partition of $[1{:}|u|]$.

2.3.3 Node dependence

Since it partitions the nodes in a top-down fashion, Claim 2.27 enables to associate an iterable node (or the root) to any position of the word. This way, define the notion of *origin* of a leaf in a forest.

Definition 2.28 (Origin of a leaf)

Let $u \in A^+$ and $\mathcal{F} \in \mathsf{Forests}_\mu(u)$. Given a position $1 \leqslant i \leqslant |u|$, we define the *origin* of i in $\mathcal{F}$, denoted $\mathsf{origin}_\mathcal{F}(i)$ as the unique node $\mathfrak{t} \in \mathsf{Iters}_\mathcal{F} \cup \{\mathcal{F}\}$ such that $i \in \mathsf{Fr}_\mathcal{F}(\mathfrak{t})$.

Thanks to origins, we roughly forget the positions of the word and focus on the nodes which belong to $\mathsf{Iters}_\mathcal{F} \cup \{\mathcal{F}\}$. When considering a k-pebble transducer, we will study the relative position of k-tuples of such nodes. Intuitively, we say that two iterable nodes are *independent* if they are "far enough" in the tree, and thus can be iterated independently and without altering the μ-forest structure.

Definition 2.29 (Node observation)

Let $\mathcal{F} \in \mathsf{Forests}_\mu$ and $\mathfrak{t}, \mathfrak{t}' \in \mathsf{Nodes}_\mathcal{F}$. We say that $\mathfrak{t} \in \mathsf{Nodes}_\mathcal{F}$ *observes* $\mathfrak{t}' \in \mathsf{Nodes}_\mathcal{F}$ if either $\mathfrak{t}'$ is an ancestor of $\mathfrak{t}$, or $\mathfrak{t}'$ is the immediate right or left sibling of an ancestor of $\mathfrak{t}$.

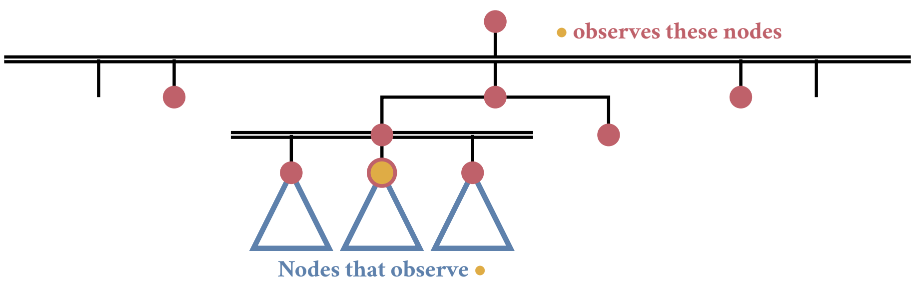


Figure 2.30: Nodes that observe • and that • observes.

The intuition behind the notion of observation (which is *not* symmetrical) is depicted in Figure 2.30. Since the nodes which a given $\mathfrak{t}$ observes are more or less its ancestors, the number of such nodes only depends on the height of the forest, as explained in Claim 2.31. Note the converse does not hold: the number of nodes which observe $\mathfrak{t}$ may not be bounded (since they are children).

Claim 2.31 (Bounded observation)

Let $d \geqslant 1$, $\mathcal{F} \in \mathsf{Forests}^d_\mu$ and $\mathfrak{t} \in \mathsf{Nodes}_\mathcal{F}$, then $\mathfrak{t}$ observes at most $3d$ nodes of $\mathcal{F}$.

Now, we introduce the symmetrized version of observation, named *dependence*.

Definition 2.32 (Node dependence)

Let $\mathcal{F} \in \mathsf{Forests}_\mu$ and $\mathfrak{t}, \mathfrak{t}' \in \mathsf{Nodes}_\mathcal{F}$. We say that $\mathfrak{t}$ and $\mathfrak{t}'$ are *dependent* if either $\mathfrak{t}$ observes $\mathfrak{t}'$, or $\mathfrak{t}'$ observes $\mathfrak{t}$. Otherwise they are said to be *independent*.

Finally, we justify that independent tuples of iterable nodes enable to factorize the word in a way which makes μ-K-iterators occur. We define the relation $\preccurlyeq$ over $\mathsf{Iters}_\mathcal{F} \cup \{\mathcal{F}\}$ by $\mathfrak{t}' \preccurlyeq \mathfrak{t}'$ if and only if $\mathsf{min}(\mathsf{Fr}_\mathcal{F}(\mathfrak{t})) \leqslant \mathsf{min}(\mathsf{Fr}_\mathcal{F}(\mathfrak{t}'))$, which defines a total ordering thanks to Claim 2.27.

Lemma 2.33 (Pairwise independent nodes)

Let $u \in A^+$, $\mathcal{F} \in \mathsf{Forests}_\mu(u)$ and $\mathfrak{t}_1 \preccurlyeq \cdots \preccurlyeq \mathfrak{t}_k \in \mathsf{Iters}_\mathcal{F}$ be pairwise independent. There exist words $v_0, \dots, v_k \in A^*, u'_1, \dots, u'_k, u_1, \dots, u_k, u''_1, \dots, u''_k \in A^+$, such that:

- $u = v_0(u'_1 u_1 u''_1)v_1 \cdots v_{k-1}(u'_k u_k u''_k)v_k$;
- for all $1 \leqslant j \leqslant k$, $e_j := \mu(u'_j) = \mu(u_j) = \mu(u''_j)$ is an idempotent of $\mathbb{M}$;
- for all $1 \leqslant j \leqslant k$, the positions of factor u_j in u are $[\mathsf{min}(\mathsf{Fr}_\mathcal{F}(\mathfrak{t}_j)){:}\mathsf{max}(\mathsf{Fr}_\mathcal{F}(\mathfrak{t}_j))]$. In particular, this means that $u_j = \mathsf{word}_\mu(\mathfrak{t}_j)$.

A formal proof of Lemma 2.33 is probably not the most convincing way to show its correctness. Instead of such a proof, we encourage the reader to have a look at Figure 2.34 which depicts a forest in the case $k = 2$. Since the iterable nodes $\mathfrak{t}_1$ and $\mathfrak{t}_2$ are independent, their right and left siblings exist and the factors of u below these nodes are disjoint. Furthermore, the image of these factors under μ must be independent. The result follows by considering these factors for the u'_j, u_j and u''_j.

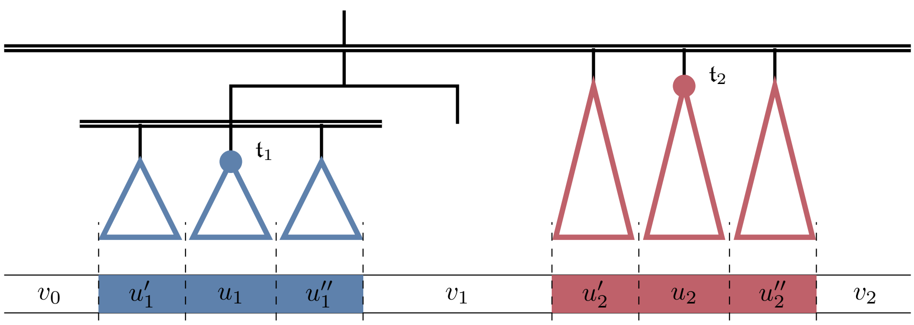


Figure 2.34: Construction of Lemma 2.33 for two independent nodes $\mathfrak{t}_1 \preccurlyeq \mathfrak{t}_2$.

This result will also be used to show Lemma 5.48 in Chapter 5, which provides a more precise statement concerning the productions of a model called k-counting transducers (the latter can roughly be understood as k-pebble transducers whose output is commutative).

Chapter 3

Making pebbles invisible: blind and last pebble transducers

FAUST, *impérieusement*
Je veux.

MÉPHISTOPHÈLÉS, *s'incline en signe de soumission et conclut philosophiquement*
Je crains que ce ne soit le dernier de vos vœux.

Lili Boulanger, E. Adenis, *Faust et Hélène*

We have observed in Section 1.3.4 that for $k \geqslant 1$, the functions computed by k-pebble transducers do not coincide with the polyregular functions f such that $|f(u)| = \mathcal{O}(|u|^k)$. Furthermore, it is open whether one can decide if a polyregular function can be computed by a k-pebble transducer for a given $k \geqslant 2$. In other words, it is unknown whether pebble transducers can automatically be optimized by removing nested layers. This goal this chapter is to study subclasses of polyregular functions for which the relation between minimal number of nested layers and asymptotic growth holds. These classes will furthermore describe robust and meaningful variants of polyregular functions.

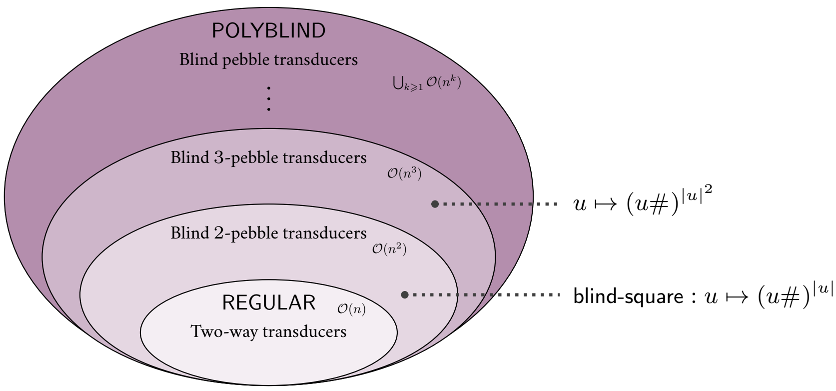


Figure 3.1: Classes of functions computed by blind pebble transducers.

For this purpose, we give in Section 3.1 two pre-existing restrictions of k-pebble transducers:

- *blind k-pebble transducers*, which are k-pebble transducers where a submachine cannot see the pebbles marking the nested calls done by its ancestors. They compute a robust class of functions called *polyblind functions*, whose properties are close to polyregular functions;
- *last k-pebble transducers*, which are k-pebble transducers where a submachine can only see the pebble dropped by its parent, but no the full stack of the former pebbles.

The classes of functions computed by blind pebble transducers are depicted in Figure 5.1.

In Sections 3.2 and 3.3, we show that a function f computed by a blind k-pebble transducer (resp. a last k-pebble transducer) can be computed by a blind ℓ-pebble transducer (resp. a last ℓ-pebble transducer) for a given $1 \leqslant \ell \leqslant k$ if and only if $|f(u)| = \mathcal{O}(|u|^\ell)$. This property is decidable and we provide an effective construction. Thus blind pebble transducers and last pebble transducers can be optimized. Our proofs make a heavy use of the factorization forests techniques which were presented in Chapter 2.

Finally, we claim in Section 3.4 that the result for last pebble transducers is tight, in the sense that the connection between number of layers and asymptotic growth does not hold for more powerful models.

The contributions presented in this chapter are based on the main theorems of [Dou23].

3.1 Blind and last pebble transducers

Let us describe two restrictions of pebble transducers. In Section 3.1.1, we present the blind pebble transducer model, which was first introduced in [NNP21, Definition 5.1] under the name "comparison-free pebble transducer". It was later studied in [Dou22, Dou23] with the "blind" terminology that we use in this manuscript. In Section 3.1.2, we move to last pebble transducers, which were first introduced in [EHS07, Section 2] over finite trees under the name "invisible pebble transducers" (the main difference is that their model allows unbounded nesting, see Section 4.5).

3.1.1 Blind pebble transducers

Blind pebble transducers can be described by the oxymoron "pebble transducers without pebbles". Intuitively, they are "blind" because a submachine cannot see the calling positions. In terms of nested "for" loops, they can be seen as programs where a loop index cannot be used inside nested loops. Recall that $a\langle t_1 \rangle \cdots \langle t_n \rangle$ denotes a tree whose root node is labelled by a and whose rooted subtrees are $t_1, \dots, t_n$. Formally, a blind k-pebble transducer is a tree of height k whose nodes are labelled by normalized 2DT.

Definition 3.2 (Blind pebble transducer)

Let $k \geqslant 1$ and $\mathscr{T}$ be a normalized 2DT with input alphabet A. We say that $\mathscr{B}$ is a *blind k-pebble transducer* with input alphabet A, output alphabet B and *head* $\mathscr{T}$ if:

- either $k = 1$, $\mathscr{B} = \mathscr{T}$ and it has output alphabet B;
- or $k \geqslant 2$, $\mathscr{B}$ is a tree $\mathscr{T}\langle \mathscr{B}_1 \rangle \cdots \langle \mathscr{B}_p \rangle$ with $p \geqslant 1$ and:
  - the subtrees $\mathscr{B}_1, \dots, \mathscr{B}_p$ are blind $(k-1)$-pebble transducers with input alphabet A, output alphabet B, and respective heads $\mathscr{T}_1, \dots, \mathscr{T}_p$;
  - $\mathscr{T}$ has output alphabet $\{\mathscr{T}_1, \dots, \mathscr{T}_p\}$.

If $\mathscr{T}$ is the head of the blind k-pebble transducer $\mathscr{B}$, we define the function computed by $\mathscr{T}$ within $\mathscr{B}$, denoted $[\![\mathscr{T}]\!] \colon A^* \to B^*$, by induction (in a similar way to pebble transducers):

- if $k = 1$, then $[\![\mathscr{T}]\!] := [\mathscr{T}] \colon A^* \to B^*$ follows the usual 2DT semantics;

- otherwise $\mathscr{T}$ has output alphabet $T := \{\mathscr{T}_1, \dots, \mathscr{T}_p\}$ and the functions $[\![\mathscr{T}_1]\!], \dots, [\![\mathscr{T}_1]\!]$ have been defined by induction. Let $g\colon A^* \to (T \times \mathbb{N})^*$ be the function computed by $\mathscr{T}$ in origin semantics[1]. Given $u \in A^*$, if $g(u) = (t_1, i_1) \cdots (t_n, i_n)$, then we let:

$$[\![\mathscr{T}]\!](u) := [\![t_1]\!](u) \cdots [\![t_n]\!](u).$$

The function $f\colon A^* \to B^*$ computed by $\mathscr{B}$ is defined as $[\![\mathscr{T}]\!]$ for its head $\mathscr{T}$. We say that a 2DT $\mathscr{T}$ is a *submachine* of the pebble transducer $\mathscr{B}$ if $\mathscr{T}$ labels a node in the tree structure of $\mathscr{B}$. We generalize the notation $[\![\mathscr{T}]\!]$ to any submachine $\mathscr{T}$ of $\mathscr{B}$, by observing that it is the head of a subtree.

The behavior of a blind pebble transducer is depicted in Figure 3.3 (to be compared with Figure 1.37).

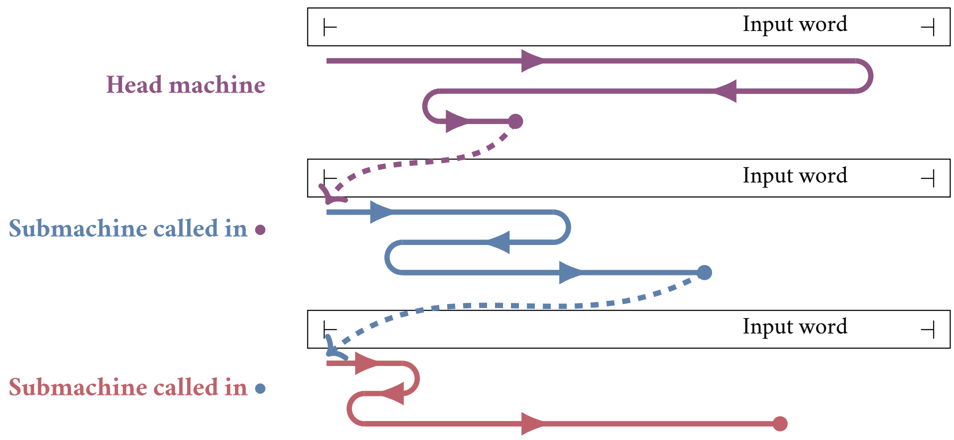


Figure 3.3: Behavior of a blind 3-pebble transducer.

Example 3.4 (Blind square)

The function blind-square: $A^* \to A^* \uplus \{\#\}, u \mapsto (u\#)^{|u|}$ is computed in Example 1.38 by a 2-pebble transducer which is in fact a blind 2-pebble transducer.

We use the term *blind pebble transducer* to denote a blind k-pebble transducer for some $k \geqslant 1$. Note that 1-pebble transducers, blind 1-pebble transducers and 2DT are the same.

Definition 3.5 (Polyblind functions)

The class of *polyblind functions* is the class of functions computed by blind pebble transducers.

3.1.1.1 Robustness and variants of the model. One can define variants of the blind k-pebble transducer model, in the spirit of the variants for k-pebble transducers described in Section 1.3.2 (that is, allowing submachines with lookarounds, or non-total submachines, or side effects, or output in the inner nodes). Such features do not modify the expressiveness of blind k-pebble transducers for $k \geqslant 1$.

The *comparison-free k-pebble transducers* introduced in [NNP21, Definition 5.1] coincide with our blind k-pebble transducers from Definition 3.2, when allowing non-total transducers, side effects and outputs in the inner nodes. Therefore both models have the same expressive power.

[1] Using origin semantics is not useful here, since no marks will be dropped. However, we have kept g in the definition in order to be consistent with the semantics of pebble transducers.

3.1.1.2 Basic properties. Now, we claim that polyblind functions are closed under composition, and we state an analogue of Theorem 1.45 which "decomposes" polyregular functions. Both results are a consequence of [NNP21, Theorem 6.1]. In view of these two properties, we claim the class of polyblind functions could also be considered as a robust and natural generalization of regular functions. However, there is no logical model known to capture this class (see e.g. [KNP23, Section 3] for a discussion), contrary to the aforementioned MSO interpretations which describe polyregular functions.

Theorem 3.6 (Composition of polyblind functions)

The class of polyblind functions is (effectively) closed under composition.

If $f\colon A^* \to B^*$ is computed by a blind k-pebble transducer and $g\colon B^* \to C^*$ by a blind ℓ-pebble transducer, then one can build a blind $(k\ell)$-pebble transducer that computes $g \circ f$.

Furthermore, it is easy to observe that Theorem 3.6 is optimal in the sense of Remark 1.44.

Theorem 3.7 (Decomposition of polyblind functions)

A function is polyblind if and only if can be written as a composition of regular functions and blind-square functions. The conversions are effective.

We shall see in Proposition 3.14 that blind pebble transducers are strictly less expressive than pebble transducers. Furthermore, the decision problem from polyregular to polyblind will be shown decidable in Chapter 6, when the outputs of the machines are in $\mathbb{N}$ (unary) or in $\mathbb{Z}$.

3.1.2 Last pebble transducers

Last pebble transducers can be seen as pebble transducers where only the "last" pebble dropped can be seen by a submachine (see Figure 3.9). If A is an alphabet, we let $\underline{A} := \{\underline{a} \mid a \in A\}$ be a disjoint underlined copy of A. In order to simplify the notations and since at most one letter will be distinguished, we identify the set $A \times \{0,1\}$ with $A \uplus \underline{A}$. In particular, $u{\bullet}i$ denotes the word $u[1{:}i{-}1]\underline{u[i]}u[i{+}1{:}|u|]$.

Definition 3.8 (Last pebble transducer)

Let $k \geqslant 1$ and $\mathscr{T}$ be a normalized 2DT with input alphabet $A \uplus \underline{A}$. We say that $\mathscr{L}$ is a *last k-pebble transducer* with input alphabet A, output alphabet B and *head* $\mathscr{T}$ if:

- either $k = 1$, $\mathscr{L} = \mathscr{T}$ and it has output alphabet B;
- or $k \geqslant 2$, $\mathscr{L}$ is a tree $\mathscr{T}\langle\mathscr{L}_1\rangle \cdots \langle\mathscr{L}_p\rangle$ with $p \geqslant 1$ and:
  - the subtrees $\mathscr{L}_1, \dots, \mathscr{L}_p$ are last $(k{-}1)$-pebble transducers with input alphabet A, output alphabet B, and respective heads $\mathscr{T}_1, \dots, \mathscr{T}_p$;
  - $\mathscr{T}$ has output alphabet $\{\mathscr{T}_1, \dots, \mathscr{T}_p\}$.

If $\mathscr{T}$ is the head of the last k-pebble transducer $\mathscr{B}$, we define the function computed by $\mathscr{T}$ within $\mathscr{B}$, denoted $[\![\mathscr{T}]\!]\colon (A \uplus \underline{A})^* \to B^*$, by induction (in a similar way to pebble transducers):

- if $k = 1$, then $[\![\mathscr{T}]\!] := [\mathscr{T}]\colon (A \uplus \underline{A})^* \to B^*$ follows the usual 2DT semantics;
- otherwise $\mathscr{T}$ has output alphabet $T := \{\mathscr{T}_1, \dots, \mathscr{T}_p\}$ and the functions $[\![\mathscr{T}_1]\!], \dots, [\![\mathscr{T}_1]\!]$ have been defined by induction. Let $g\colon A^* \to (T \times \mathbb{N})^*$ be the function computed by $\mathscr{T}$ in origin semantics. Given $u \in A^*$, if $g(u) = (t_1, i_1) \cdots (t_n, i_n)$, then we let:

$$[\![\mathscr{T}]\!](u) := [\![t_1]\!](\nu(u){\bullet}i_1) \cdots [\![t_n]\!](\nu(u){\bullet}i_n).$$

where $\nu\colon (A \uplus \underline{A})^* \to A^*$ is the morphism which erases the underlining.

The function $f\colon A^* \to B^*$ computed by $\mathscr{L}$ is defined as $[\![\mathscr{T}]\!]|_{A^*}$ for its head $\mathscr{T}$ (the restriction to A^* is due to the fact that underlinings are only used within nested calls). We say that a 2DT $\mathscr{T}$ is a *submachine* of the pebble transducer $\mathscr{L}$ if $\mathscr{T}$ labels a node in the tree structure of $\mathscr{L}$. We generalize the notation $[\![\mathscr{T}]\!]$ to any submachine $\mathscr{T}$ of $\mathscr{L}$, by observing that it is the head of a subtree.

The behavior of a last pebble transducer is depicted in Figure 3.9 (to be compared with Figure 3.3).

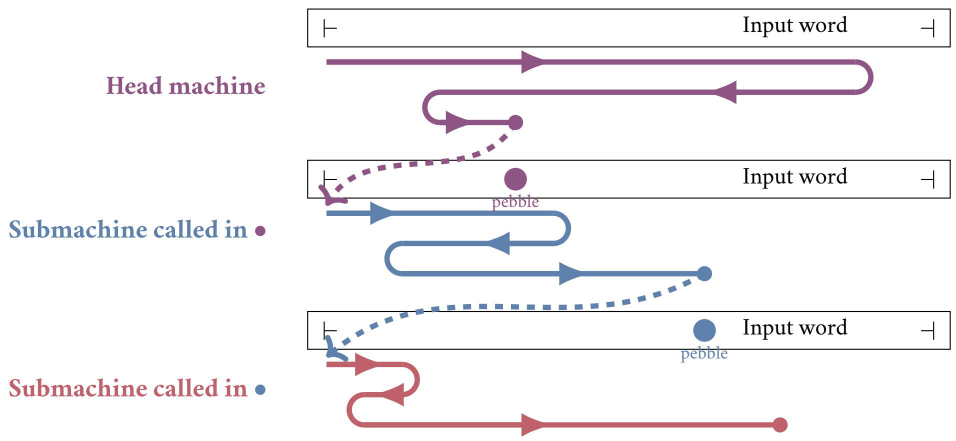


Figure 3.9: Behavior of a last 3-pebble transducer.

Example 3.10 (Square)

The function $\mathsf{square}\colon A \to A \uplus \{\#\}$ from Example 1.38 which maps u to $(u\bullet 1)\# \cdots (u\bullet|u|)\#$ can be computed by a last 2-pebble transducer.

It is easy to observe that last 2-pebble transducers and 2-pebble transducers are the same. We use the term *last pebble transducer* to denote a last k-pebble transducer for some $k \geqslant 1$. The respective expressive power of pebble transducers and last pebble transducers is discussed in Proposition 3.15.

3.1.2.1 Robustness and variants of the model.

One can define variants of the last k-pebble transducer model, in the spirit of the variants for k-pebble transducers described in Section 1.3.2 (that is, allowing submachines with lookarounds, or non-total submachines, or side effects, or output in the inner nodes). Such features do not modify the expressiveness of last k-pebble transducers for $k \geqslant 1$.

We shall see that the class of functions computed by last pebble transducers is not closed under composition. However, it is still closer under composition by a regular function. The next result is easy by leveraging standard proofs techniques, e.g. those of Theorems 1.31, 1.43 and 3.6.

Proposition 3.11 (Composition with regular functions)

For all $k \geqslant 1$, the class of functions computed by last k-pebble transducers is (effectively) closed under pre- and post-composition by regular functions.

3.1.3 Optimization theorems and consequences

The main goal of Chapter 3 is to show how blind k-pebble transducers and last k-pebble transducers can be optimized by minimizing the number $k \geqslant 1$ of layers needed to compute a function. These results are stated in Theorems 3.12 and 3.13 which both originate from [Dou23, Theorem 3.5]. The connection between number of layers and asymptotic growth is the key result to get decidability.

Theorem 3.12 (Optimization of blind pebble transducers)

Let $1 \leqslant \ell \leqslant k$ and $f\colon A^* \to B^*$ be computed by a blind k-pebble transducer. Then f can be computed by a blind ℓ-pebble transducer if and only if $|f(u)| = \mathcal{O}(|u|^\ell)$. This property is decidable. If it holds, one can build a blind ℓ-pebble transducer which computes f.

Proof. A detailed proof is presented in Section 3.2. It relies on the tools of Chapter 2. ◀

Theorem 3.13 (Optimization of last pebble transducers)

Let $1 \leqslant \ell \leqslant k$ and $f\colon A^* \to B^*$ be computed by a last k-pebble transducer. Then f can be computed by a blind ℓ-pebble transducer if and only if $|f(u)| = \mathcal{O}(|u|^\ell)$. This property is decidable. If it holds, one can build a last ℓ-pebble transducer which computes f.

Proof. A detailed proof is presented in Section 3.3. It relies on the tools of Chapter 2 and its sketch is very similar to that of Section 3.2, however it is far more involved. ◀

Let us show that Theorems 3.12 and 3.13 provide versatile tools for exploring the expressive power of blind pebble transducers and last pebble transducers. We first observe that polyblind functions are a strict subclass of polyregular functions. The next result originates from [NNP21, Theorem 8.3].

Proposition 3.14 (Separation between pebble and blind pebble transducers)

The functions inner-squaring, square and prefixes are not polyblind.

Proof. If inner-squaring was computable by a blind pebble transducer, it would be computable by a blind 2-pebble transducer by Theorem 3.12, thus by a 2-pebble transducer, a contradiction with Theorem 1.48. Now if square was polyblind, the classes of polyregular and polyblind functions would be equal by Theorems 1.45 and 3.6, so inner-squaring would be polyblind.

We propose a direct combinatorial proof for prefixes: $u \mapsto u[1{:}1]\#u[1{:}2]\#\cdots u[1{:}|u|]\#$. Assume that this function is polyblind, then by Theorem 3.12 it is computed by a blind 2-pebble transducer $\mathscr{T}\langle\mathscr{T}_1\rangle\cdots\langle\mathscr{T}_p\rangle$. By leveraging the proof techniques of Proposition 2.16 to a finite collection of transducers, one can find words $v_0, v_1 \in A^+, u \in A^+$ such that:

- $[\![\mathscr{T}]\!](v_0u^{X+1}v_1) = \alpha_0(\beta_1)^X\alpha_1\cdots(\beta_n)^X\alpha_n$ with $n \geqslant 0$, $\alpha_0,\dots,\alpha_n \in \{\mathscr{T}_1,\dots,\mathscr{T}_p\}^*$ and $\beta_1,\dots,\beta_n \in \{\mathscr{T}_1,\dots,\mathscr{T}_p\}^+$. Since $|\mathsf{prefixes}(v_0u^Xv_1)| = \theta(X^2)$, we have $n \geqslant 1$;
- for all $1 \leqslant j \leqslant p$, $[\![\mathscr{T}_j]\!](v_0u^{X+1}v_1) = \alpha_{0,j}(\beta_{1,j})^X\alpha_{1,j}\cdots(\beta_{\ell_j,j})^X\alpha_{\ell_j,j}$ with $\ell_j \geqslant 0$, $\alpha_{1,j},\dots,\alpha_{\ell_j,j} \in (A \uplus \#)^*$ and $\beta_{1,j},\dots,\beta_{\ell_j,j} \in (A \uplus \#)^+$.

By putting everything together and relabelling the words, we obtain:

$$[\![\mathscr{T}]\!](v_0u^{X+1}v_1) = \alpha_0(\delta_{0,1}\gamma_{1,1}^X\cdots\gamma_{m_1,1}^X\delta_{m_1,1})^X\alpha_1\cdots(\delta_{0,n}\gamma_{1,n}^X\cdots\gamma_{m_n,n}^X\delta_{m_n,n})^X\alpha_n$$

with for all $1 \leqslant j \leqslant n$, $m_j \geqslant 0$ $\delta_{0,j},\dots,\delta_{m_j,j} \in (A \uplus \#)^*$ and $\gamma_{0,j},\dots,\gamma_{m_j,j} \in (A \uplus \#)^+$. Since two maximal $\#$-free factors of $\mathsf{prefixes}(v_0(u)^Xv_1)$ cannot have the same size, we obtain

$\gamma_{i,j} \in A^+$ for all $1 \leqslant j \leqslant n$ and $1 \leqslant i \leqslant m_j$. For the same reason, for all $1 \leqslant j \leqslant n$ there is at most one $1 \leqslant i \leqslant m_j$ such that $\#$ occurs in $\delta_{i,j}$. Since $|\mathsf{prefixes}(v_0 u^{X+1} v_1)| = \theta(X^2)$, there exists $1 \leqslant j \leqslant n$ such that $m_j \geqslant 1$. There exists one (unique) $0 \leqslant i \leqslant m_j$ such that $\#$ occurs in $\delta_{i,j}$, because otherwise $\mathsf{prefixes}(v_0(u)^{X+1}v_1)$ would contain a $\#$-free factor of quadratic size in X. Now, any repetition $(\delta_{0,j}\gamma_{1,j}^X \cdots \gamma_{m_j,j}^X \delta_{m_j,j})^X$ contains several maximal $\#$-free factors of the same size, which yields a contradiction with the definition of prefixes. ◀

More generally, there is a strict hierarchy between blind pebbles, last pebbles and pebbles.

Proposition 3.15 (Separation between pebble, last pebble and blind pebble transducers)

Pebble transducers are strictly more expressive than last pebble transducers, which are strictly more expressive than blind pebble transducers. In more detail, the function inner-squaring can be computed by a 3-pebble transducer but not by a last pebble transducer; and the functions square and prefixes can be computed by a last 2-pebble transducer but not by a blind pebble transducer.

Proof. If inner-squaring was computable by a last pebble transducer, it would be computable by a last 2-pebble transducer by Theorem 3.13, a contradiction with Theorem 1.48. The result for square and prefixes follows from Examples 1.38 and 1.39 and Proposition 3.14. ◀

Finally, we claim that the class of functions computed by last pebble transducers is not closed under composition. Intuitively, this is due to the fact that composition would require to see two pebbles.

Proposition 3.16 (Lack of composition)

The class of functions computed by last pebble transducers is not closed under composition.

Proof. Since this class contains regular functions and square functions, its closure under composition would imply by Theorem 1.45 that it equals polyregular functions. ◀

3.2 Solving the optimization problem for blind transducers

This section is devoted to showing Theorem 3.12 (it will follow from Theorem 3.20), by following the proof of [Dou23, Section 5] which relies on factorization forests. The connection between asymptotic growth and number of nested layers for blind pebble transducers was also shown using different techniques in [NNP21, Theorem 7.1], but they neither study effectiveness nor decidability.

3.2.1 Pumpable transducers and asymptotic growth

Let us first give a necessary condition, named *pumpability*, for a blind k-pebble transducer to compute a function f such that $|f(u)| = \mathcal{O}(|u|^{k-1})$. If it does not hold, the function cannot be computed by a blind $(k-1)$-pebble transducer. Let the *transition morphism* of a blind pebble transducer $\mathscr{B}$ be the cartesian product of the transition morphisms of all the submachines of $\mathscr{B}$. Observe that it makes sense to consider the production of a submachine $\mathscr{T}$ in a μ-context when μ is the transition morphism of $\mathscr{B}$.

Definition 3.17 is probably harsh at a first reading, but the notion of *pumpability* is inspired from Lemma 2.15 for 2DT. Here, the idea is to build a pattern which describes how several μ-contexts for submachines can call each other, in a way which can be iterated. Observe that being pumpable can be decided by ranging over tuples of transition monoid elements and letters.

Definition 3.17 (Pumpable blind transducer)

Let $\mathscr{B}$ be a blind k-pebble transducer whose transition morphism is $\mu\colon A^* \to \mathbb{T}$. We say that the transducer $\mathscr{B}$ is *pumpable* if there exist:

- submachines $\mathscr{T}_1, \dots, \mathscr{T}_k$ of $\mathscr{B}$, such that $\mathscr{T}_1$ is the head of $\mathscr{B}$;
- $m_0, \dots, m_k, \ell_1, \dots, \ell_k, r_1, \dots, r_k \in \mathbb{T}$;
- $a_1, \dots, a_k \in A$ such that for all $1 \leqslant j \leqslant k$, $e_j := \ell_j \mu(a_j) r_j \in \mathbb{T}$ is idempotent;
- a permutation $\sigma\colon [1{:}k] \to [1{:}k]$;

such that if $\mathcal{M}_i^j := m_i e_{i+1} m_{i+1} \cdots e_j m_j$ for all $0 \leqslant i \leqslant j \leqslant k$, and if we define the following μ-context for all $1 \leqslant j \leqslant k$:

$$\mathcal{C}_j := \mathcal{M}_0^{\sigma(j)-1} e_{\sigma(j)} \ell_{\sigma(j)} \lfloor a_{\sigma(j)} \rfloor r_{\sigma(j)} e_{\sigma(j)} \mathcal{M}_{\sigma(j)}^k$$

then for all $1 \leqslant j \leqslant k-1$, $|\mathsf{prod}_{\mathscr{T}_j}(\mathcal{C}_j)|_{\mathscr{T}_{j+1}} \neq 0$ and $|\mathsf{prod}_{\mathscr{T}_k}(\mathcal{C}_k)| \neq 0$.

The behavior of a pumpable blind 2-pebble transducer is depicted in Figure 3.18 over a well-chosen input: it has a factor in which the head $\mathscr{T}_1$ calls a submachine $\mathscr{T}_2$, and a factor in which $\mathscr{T}_2$ produces a non-empty output. Furthermore both factors can be iterated while preserving the shape of the runs.

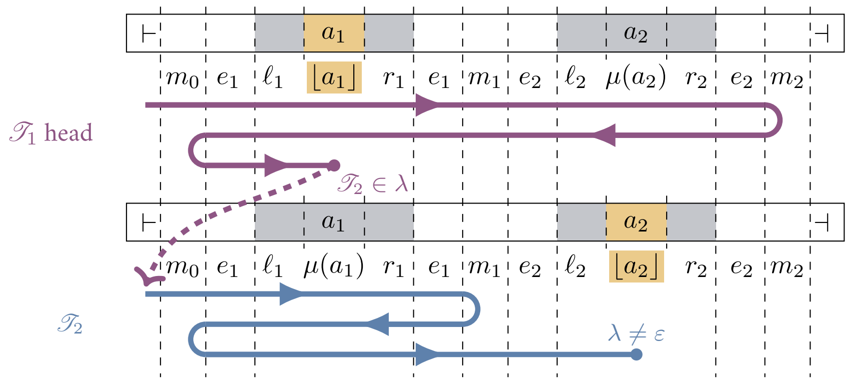


Figure 3.18: Pumpability in a blind 2-pebble transducer.

We first claim that pumpability is a sufficient condition for having asymptotic growth in $\theta(n^k)$.

Claim 3.19 (Pumpability $\Rightarrow$ Growth)

Let $f\colon A^* \to B^*$ be computed by a pumpable blind k-pebble transducer, then there exist $v_0, \dots, v_k \in A^*, u_1, \dots, u_k \in A^+$, such that $|f(v_0 u_1^X \cdots u_k^X v_k)| = \theta(X^k)$.

Proof. We use the notations of Definition 3.17. For $1 \leqslant j \leqslant k$, let $w_j, w'_j \in A^*$ be such that $\mu(w_j) = \ell_j$ and $\mu(w'_j) = r_j$, $v_0, \dots, v_k \in A^*$ be such that $\mu(v_j) = m_j$ for all $1 \leqslant j \leqslant k$ and $u_j := w_j a_j w'_j$, for $1 \leqslant j \leqslant k$. We show that $|[\![\mathscr{T}_1]\!](v_0 u_1^X \cdots u_k^X v_k)| = \theta(X^k)$.

From the properties of productions we get for all $X \geqslant 2$, $|[\![\mathscr{T}_k]\!](v_0 u_1^X \cdots u_k^X v_k)| \geqslant (X-2) \times |\mathsf{prod}_{\mathscr{T}_k}(\mathcal{C}_k)| \geqslant X-2$. Similarly, $|[\![\mathscr{T}_j]\!](v_0 u_1^X \cdots u_k^X v_k)|_{\mathscr{T}_{j+1}} \geqslant (X-2) \times |\mathsf{prod}_{\mathscr{T}_j}(\mathcal{C}_j)|_{\mathscr{T}_{j+1}} \geqslant X-2$ for $1 \leqslant j \leqslant k-1$ and $X \geqslant 2$. Therefore $|[\![\mathscr{T}_1]\!](v_0 u_1^X \cdots u_k^X v_k)| \geqslant (X-2)^k$. ◀

Now we are ready to state a refinement of Theorem 3.12.

Theorem 3.20 (Removing one blind pebble layer)

Let $k \geqslant 2$ and $f\colon A^* \to B^*$ be a function computed by a blind k-pebble transducer $\mathscr{B}$. The following conditions are equivalent:

(1) $|f(u)| = \mathcal{O}(|u|^{k-1})$;
(2) $\mathscr{B}$ is not pumpable;
(3) f can be computed by a blind $(k-1)$-pebble transducer.

Furthermore, this property is decidable and the construction is effective.

Proof. Item (3) $\Rightarrow$ Item (1) in Theorem 3.20 is obvious and Item (1) $\Rightarrow$ Item (2) is Claim 3.19. Furthermore, we have observed above that pumpability is decidable. Item (2) $\Rightarrow$ Item (3) is shown (in an effective fashion) in Section 3.2.2. It is the main body of this proof. ◀

3.2.2 Removing a nested layer in a non-pumpable transducer

In Section 3.2.2, we show Item (2) $\Rightarrow$ Item (3) in Theorem 3.20. Let us fix $k \geqslant 2$ and $\mathscr{B}$ a blind k-pebble transducer which is not pumpable and computes a function $f\colon A^* \to B^*$. Our goal is to build a blind $(k-1)$-pebble transducer computing f. Let $\mu\colon A^* \to \mathbb{T}$ be the transition morphism of $\mathscr{B}$, our new machine will compute the composition of:

- the rational function from Theorem 2.21, $\mathsf{forest}_\mu\colon A^* \to \mathsf{Forests}_\mu^{3|\mathbb{T}|}$;
- the function $f \circ \mathsf{word}_\mu^{3|\mathbb{T}|}\colon \mathsf{Forests}_\mu^{3|\mathbb{T}|} \to B^*$, computed by a blind $(k-1)$-pebble transducer $\overline{\mathscr{B}}$. We shall allow its submachines to have lookarounds, since as explained in Section 3.1.1.1 this feature does not modify the expressiveness of the models.

Once these steps are achieved, it follows from Theorem 3.6 that the composition $f = f \circ \mathsf{word}_\mu^{3|\mathbb{T}|} \circ \mathsf{forest}_\mu$ can effectively be computed by a blind $(k-1)$-pebble transducer.

3.2.2.1 Construction of $\overline{\mathscr{B}}$. The rest of this section is devoted to building $\overline{\mathscr{B}}$ and justifying the correctness of the construction. We first describe its submachines, which are of two kinds:

- for each submachine $\mathscr{T}$ of $\mathscr{B}$, $\overline{\mathscr{B}}$ has a submachine old-$\mathscr{T}$. The latter behaves as $\mathscr{T}$ does, with the difference that it takes a μ-forest as input and that it makes calls to the old-$\mathscr{T}'$ instead of the $\mathscr{T}'$. The behavior of old-$\mathscr{T}$ is detailed in Algorithm 3.21 when $\mathscr{T}$ is not a leaf of $\mathscr{B}$. The case of a leaf is obtained by modifying Line 7 to produce exactly the output (in[2] B) of $\mathscr{T}$;
- for each submachine $\mathscr{T}$ of $\mathscr{B}$ which is not a leaf, $\overline{\mathscr{B}}$ has a submachine new-$\mathscr{T}$. The goal of new-$\mathscr{T}$ is to simulate $\mathscr{T}$ as old-$\mathscr{T}$ does, while inlining the nested calls within its own run (i.e. removing a nesting layer) in two cases: if the call is done in a position which depends on the root of the forest, or if its children are leaves of $\mathscr{B}$. This behavior is detailed in Algorithm 3.21.

Finally, the transducer $\overline{\mathscr{B}}$ is obtained by defining new-$\mathscr{T}$ as its head, where $\mathscr{T}$ is the head of $\mathscr{B}$. Furthermore, we remove the submachines old-$\mathscr{T}$ or new-$\mathscr{T}$ which are never called. It remains to justify that the construction is correct and indeed defines a blind $(k-1)$-pebble transducer. The key property of $\overline{\mathscr{B}}$ is that it only make nested calls in positions whose origins are iterable nodes (i.e. not the root).

3.2.2.2 $\overline{\mathscr{B}}$ is correct. We first justify that $\overline{\mathscr{B}}$ computes the function $f \circ \mathsf{word}_\mu^{3|\mathbb{T}|}$. If old-$\mathscr{T}$ is used in $\overline{\mathscr{B}}$, it is easy to see that $[\![\text{old-}\mathscr{T}]\!] = [\![\mathscr{T}]\!] \circ \mathsf{word}_\mu^{3|\mathbb{T}|}$. In a similar fashion, if new-$\mathscr{T}$ is used in $\overline{\mathscr{B}}$, then $[\![\text{new-}\mathscr{T}]\!] = [\![\mathscr{T}]\!] \circ \mathsf{word}_\mu^{3|\mathbb{T}|}$. The result follows by considering the head.

[2]Recall that since $\mathscr{T}$ is normalized, the output along a transition is either a letter or ε.

Algorithm 3.21: Submachines of the blind $(k-1)$-pebble transducer $\overline{\mathscr{B}}$

1 **Submachine** old-$\mathscr{T}(\mathcal{F})$
2 /* Suppose that $\mathscr{T}$ has shape $(A, C, Q, q_0, F, \delta, \lambda)$. */
3 $u := \mathsf{word}^{3|\mathbb{T}|}_{\mu}(\mathcal{F})$ /* Input word of $\mathscr{T}$. */
4 $(q_1, i_1) \to \cdots \to (q_n, i_n) :=$ accepting n-run of $\mathscr{T}$ labelled by u
5 **for** $1 \leqslant j \leqslant n$ **do**
6 **if** $\mathscr{T}' := \lambda(q_j, u[i_j]) \neq \varepsilon$ **then**
7 **Call submachine** old-$\mathscr{T}'(\mathcal{F})$
8 **end**
9 **end**
10 **Submachine** new-$\mathscr{T}(\mathcal{F})$
11 /* Suppose that $\mathscr{T}$ has shape $(A, C, Q, q_0, F, \delta, \lambda)$ */
12 $u := \mathsf{word}^{3|\mathbb{T}|}_{\mu}(\mathcal{F})$ /* Input word of $\mathscr{T}$ */
13 $(q_1, i_1) \to \cdots \to (q_n, i_n) :=$ accepting n-run of $\mathscr{T}$ labelled by u
14 **for** $1 \leqslant j \leqslant n$ **do**
15 **if** $\mathscr{T}' := \lambda(q_j, u[i_j]) \neq \varepsilon$ **then**
16 **if** $i_j \in \mathsf{Fr}_{\mathcal{F}}(\mathcal{F})$ **then**
17 /* The set $\mathsf{Fr}_{\mathcal{F}}(\mathcal{F})$ has bounded size. */
18 **Inline the code of** old-$\mathscr{T}'(\mathcal{F})$
19 **else if** $\mathscr{T}'$ is a leaf of $\mathscr{B}$ **then**
20 /* The output of $\mathscr{T}'$ is bounded by Lemma 3.23. */
21 **Inline the code of** old-$\mathscr{T}'(\mathcal{F})$
22 **else**
23 **Call submachine** new-$\mathscr{T}'(\mathcal{F})$
24 **end**
25 **end**
26 **end**

3.2.2.3 $\overline{\mathscr{B}}$ has $k-1$ nested layers. The next step towards showing that $\overline{\mathscr{B}}$ is a blind $(k-1)$-pebble transducer is to show that it has exactly $k-1$ (and not k) nested layers. Formally, we say that a submachine of a blind pebble transducer has *height* $h \geqslant 1$ if it is the head of a subtree of height h. Our goal is to show that the head of $\overline{\mathscr{B}}$ has height $k-1$, which is equivalent to saying that $\overline{\mathscr{B}}$ has $k-1$ nested layers.

We first show by induction that if $\mathscr{T}$ has height h in $\mathscr{B}$, then old-$\mathscr{T}$ (if used) has height h in $\overline{\mathscr{B}}$ (this proof is easy). Second, we show by induction on $2 \leqslant h \leqslant k$ that if $\mathscr{T}$ has height h in $\mathscr{B}$, then new-$\mathscr{T}$ (if used) has height $h-1$ in $\overline{\mathscr{B}}$. Indeed, the base case $h = 2$ is justified by Line 21 in Algorithm 3.21 (there are no nested calls since we inline the code of all the children). For $h > 2$ the machine new-$\mathscr{T}$ either inlines the code of old-$\mathscr{T}'$ (which has height $h-1$) or makes a nested call to new-$\mathscr{T}'$ (which has height $h-2$ by induction hypothesis), thus it has height $h-1$.

3.2.2.4 Each submachine of $\overline{\mathscr{B}}$ is a two-way transducer (with lookaheads). Each old-$\mathscr{T}$ can be implemented by a 2DT which moves on the leaves of $\mathcal{F}$ while following variable $1 \leqslant i_j \leqslant |u|$. It remains to justify that each new-$\mathscr{T}$ which occurs in $\overline{\mathscr{B}}$ can also be implemented in a similar fashion.

Since $\mathcal{F} \in \mathsf{Forests}^{3|\mathbb{T}|}_{\mu}$, the size of $\mathsf{Fr}_{\mathcal{F}}(\mathcal{F})$ is bounded, and one can easily build a lookaround which enables to detect whether $i_j \in \mathsf{Fr}_{\mathcal{F}}(\mathcal{F})$ holds, in the sense of Claim 3.22.

Claim 3.22 (Frontiers can be detected)

One can build regular languages $R, L \subseteq (A \uplus \{\langle, \rangle\})^*$ such that the following conditions are equivalent for all $\mathcal{F} \in \mathsf{Forests}_\mu^{3|\mathbb{T}|}$ (seen as a word) and $1 \leqslant i \leqslant |\mathcal{F}|$:

- $\mathcal{F}[1{:}i{-}1] \in L$ and $\mathcal{F}[i{+}1{:}|\mathcal{F}|] \in R$;
- $\mathcal{F}[i]$ encodes a leaf of $\mathcal{F}$ which belongs to $\mathsf{Fr}_\mathcal{F}(\mathcal{F})$.

It remains to explain how the "Inline the code" instructions of Lines 18 and 21 are implemented:

- if $i_j \in \mathsf{Fr}_\mathcal{F}(\mathcal{F})$, then new-$\mathscr{T}$ inlines the code of old-$\mathscr{T}'$ by executing on the leaves of $\mathcal{F}$ the same moves and outputs as $\mathscr{T}'$ does on input u. Once this simulation is ended, new-$\mathscr{T}$ has to go back to leaf i_j. This can be done by storing in the state that i_j was the ℓ-th leaf of $\mathsf{Fr}_\mathcal{F}(\mathcal{F})$ (ℓ being bounded), and using the lookaround of Claim 3.22 to recover this position;
- otherwise $\mathscr{T}'$ is a leaf of $\mathscr{B}$, that is a 2DT with output alphabet B. In this case, new-$\mathscr{T}$ inlines the code of old-$\mathscr{T}'$ by producing $[\![\mathscr{T}']\!](u)$ without moving. Indeed, we claim that $[\![\mathscr{T}']\!](u)$ is bounded independently from u and $\mathcal{F} \in \mathsf{Forests}_\mu^{3|\mathbb{T}|}$ (thus some lookaround can be used to determine the exact output among a bounded number of possibilities). More precisely, we claim that for all $i' \notin \mathsf{Fr}_\mathcal{F}(\mathcal{F})$, $|\mathsf{prod}^u_{\mathscr{T}'}(i')| = 0$. Indeed, if $|\mathsf{prod}^u_{\mathscr{T}'}(i')| \neq 0$ for such an $i' \notin \mathsf{Fr}_\mathcal{F}(\mathcal{F})$ when reaching Line 21 of Algorithm 3.21 in an execution of $\overline{\mathscr{B}}$, it is easy to observe that the conditions of Lemma 3.23 hold, which yields a contradiction. This lemma is the key argument of this proof: observe that it relies on the non-pumpability of $\mathscr{B}$.

Lemma 3.23 (Key lemma for removing one last pebble layer)

Let $u \in A^+$ and $\mathcal{F} \in \mathsf{Forests}_\mu(u)$. Assume that there exist a sequence $\mathscr{T}_1, \dots, \mathscr{T}_k$ of submachines of $\mathscr{B}$ and a sequence of positions $1 \leqslant i_1, \dots, i_k \leqslant |u|$ such that:

- $\mathscr{T}_1$ is the head of $\mathscr{B}$;
- for all $1 \leqslant j \leqslant k{-}1$, $|\mathsf{prod}^u_{\mathscr{T}_j}(i_j)|_{\mathscr{T}_{j+1}} \neq 0$;
- $|\mathsf{prod}^u_{\mathscr{T}_k}(i_k)| \neq 0$;
- for all $1 \leqslant j \leqslant k$, $i_j \notin \mathsf{Fr}_\mathcal{F}(\mathcal{F})$.

Then $\mathscr{B}$ is pumpable.

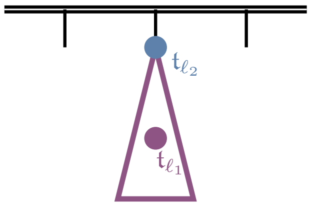
Proof. Assume that the conditions of Lemma 3.23 hold and let $\mathfrak{t}_j := \mathsf{origin}_\mathcal{F}(i_j)$ for all $1 \leqslant j \leqslant k$. Observe that for all $1 \leqslant j \leqslant k$ we have $\mathfrak{t}_j \in \mathsf{Iters}_\mathcal{F}$. If the $\mathfrak{t}_j$ are pairwise independent, then the pumpability of $\mathscr{B}$ follows from Lemma 2.33 (intuitively, the factors $\mathsf{word}_\mu(\mathfrak{t}_j)$ of u are pairwise "far enough" and furthermore their images under μ are idempotent).

Now, we suppose that the $\mathfrak{t}_j$ are not necessarily pairwise independent. Let us show how to make the number of dependent couples of $(\mathfrak{t}_{j_1}, \mathfrak{t}_{j_2})$ decrease strictly, while preserving the properties of Lemma 3.23. Repeating this process will enable us to make all the nodes pairwise independent.

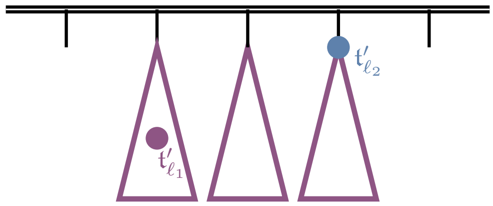
Assume that $\mathfrak{t}_{\ell_1}$ observes $\mathfrak{t}_{\ell_2}$ for some $1 \leqslant \ell_1 \neq \ell_2 \leqslant k$. To simplify the proof, we assume that $\mathfrak{t}_{\ell_2}$ is an ancestor of $\mathfrak{t}_{\ell_1}$ (the case of the immediate sibling of an ancestor is similar). Let $\mathcal{F}'$ be $\mathcal{F}$ in which the subtree $\mathfrak{t}_{\ell_2}$ has been copied 3 times (since $\mathfrak{t}_{\ell_2}$ is an iterable node, then $\mathcal{F}'$ still a μ-forest), see Figure 3.24. We define for $1 \leqslant j \leqslant k$ the nodes $\mathfrak{t}'_j \in \mathsf{Nodes}_{\mathcal{F}'}$ as follows:

- if $j = \ell_2$, then we let $\mathfrak{t}'_j$ be (the root of) the third copy of $\mathfrak{t}_j$;
- else if $\mathfrak{t}_j$ was a descendant of $\mathfrak{t}_{\ell_2}$ (including $\mathfrak{t}_{\ell_1}$ but not $\mathfrak{t}_{\ell_2}$), then we let $\mathfrak{t}'_j$ be the corresponding node in the first copy of $\mathfrak{t}_{\ell_2}$ (see Figure 3.24b);
- else if $\mathfrak{t}_j$ is in the rest of $\mathcal{F}$, we let $\mathfrak{t}'_j$ be the corresponding node in the rest of $\mathcal{F}'$.

Observe that now, $\mathfrak{t}'_{\ell_1}$ and $\mathfrak{t}'_{\ell_2}$ are not dependent. Furthermore if $\mathfrak{t}_{j_1}$ and $\mathfrak{t}_{j_2}$ were independent in $\mathcal{F}$ for $1 \leqslant j_1, j_2 \leqslant k$, then $\mathfrak{t}'_{j_1}$ and $\mathfrak{t}'_{j_2}$ are also independent. Let $u' := \mathsf{word}_\mu(\mathcal{F}')$, we define



(a) Original μ-forest $\mathcal{F}$.



(b) Modified μ-forest $\mathcal{F}'$.

Figure 3.24: Duplicating a subtree in $\mathcal{F}$ so that $\mathfrak{t}'_{\ell_1}$ and $\mathfrak{t}'_{\ell_2}$ become independent.

$1 \leqslant i'_1, \dots, i'_k \leqslant |u'|$ as the positions which correspond to the former $1 \leqslant i_1, \dots, i_k \leqslant |u|$ in the frontiers of $\mathfrak{t}'_1, \dots, \mathfrak{t}'_k$ in the new μ-forest $\mathcal{F}'$. Since we have only duplicated an iterable node, observe that $\mu(u[1{:}i_j{-}1])\lfloor u[i_j] \rfloor \mu(u[i_j{+}1{:}|u|]) = \mu(u[1{:}i'_j{-}1])\lfloor u'[i'_j] \rfloor \mu(u'[i'_j{+}1{:}|u'|])$ for all $1 \leqslant j \leqslant k$. Thus $\mathsf{prod}^{u}_{\mathscr{T}_j}(i_j) = \mathsf{prod}^{u'}_{\mathscr{T}_j}(i'_j)$ and the conditions of Lemma 3.23 still hold. ◀

Thus $\overline{\mathscr{B}}$ is a blind $(k{-}1)$-pebble transducer which computes $f \circ \mathsf{word}^{3|\mathbb{T}|}_{\mu}$. The result follows.

3.3 Solving the optimization problem for last transducers

This section is devoted to showing Theorem 3.13 (it will follow from Theorem 3.28, which is an adaptation of Theorem 3.20). The proof scheme is similar to that of Section 3.2 for blind pebble transducers, while being far more involved. We follow the presentation of [Dou23, Section 6].

3.3.1 Pumpable transducers and aymptotic growth

We first introduce a notion of *pumpability* for last pebble transducers, whose intuition is depicted in Figure 3.26. The formal definition is more cumbersome than for blind pebble transducers, since we need to keep track of the fact that the calling position is marked. Let the *transition morphism* of a last pebble transducer $\mathscr{L}$ be the cartesian product of the transition morphisms of all the submachines of $\mathscr{L}$, thus it is a surjective mapping of type $(A \uplus \underline{A})^* \to \mathbb{T}$ where $\mathbb{T}$ is the (finite) *transition monoid*.

Definition 3.25 (Pumpable last transducer)

Let $\mathscr{L}$ be a last k-pebble transducer whose transition morphism is $\mu\colon (A \uplus \underline{A})^* \to \mathbb{T}$. We say that the transducer $\mathscr{L}$ is *pumpable* if there exist:

- submachines $\mathscr{T}_1, \dots, \mathscr{T}_k$ of $\mathscr{L}$, such that $\mathscr{T}_1$ is the head of $\mathscr{L}$;
- $m_0, \dots, m_k, \ell_1, \dots, \ell_k, r_1, \dots, r_k \in \mu(A^*) \subseteq \mathbb{T}$;
- $a_1, \dots, a_k \in A$ such that for all $1 \leqslant j \leqslant k$, $e_j := \ell_j \mu(a_j) r_j$ is idempotent;
- a permutation $\sigma\colon [1{:}k] \to [1{:}k]$;

such that if we let $\mathcal{M}^j_i := m_i e_{i+1} m_{i+1} \cdots e_j m_j$ for all $0 \leqslant i \leqslant j \leqslant k$, and if we define the following μ-context:

$$\mathcal{C}_1 := \mathcal{M}^{\sigma(1)-1}_0 e_{\sigma(1)} \ell_{\sigma(1)} \lfloor a_{\sigma(1)} \rfloor r_{\sigma(1)} e_{\sigma(1)} \mathcal{M}^k_{\sigma(1)}$$

and for all $1 \leqslant j \leqslant k{-}1$ the μ-context:

$$\begin{aligned}
\mathcal{C}_{j+1} &:= \mathcal{M}_0^{\sigma(j)-1} e_{\sigma(j)} \ell_{\sigma(j)} \mu(\underline{a_{\sigma(j)}}) r_{\sigma(j)} e_{\sigma(j)} \mathcal{M}_{\sigma(j)}^{\sigma(j+1)-1} \\
&\quad e_{\sigma(j+1)} \ell_{\sigma(j+1)} \lfloor a_{\sigma(j+1)} \rfloor r_{\sigma(j+1)} e_{\sigma(j+1)} \mathcal{M}_{\sigma(j+1)}^{k} && \text{if } \sigma(j) < \sigma(j+1); \\
\mathcal{C}_{j+1} &:= \mathcal{M}_0^{\sigma(j)-1} e_{\sigma(j+1)} \ell_{\sigma(j+1)} \lfloor a_{\sigma(j+1)} \rfloor r_{\sigma(j+1)} e_{\sigma(j+1)} \\
&\quad \mathcal{M}_{\sigma(j+1)}^{\sigma(j)-1} e_{\sigma(j)} \ell_{\sigma(j)} \mu(\underline{a_{\sigma(j)}}) r_{\sigma(j)} e_{\sigma(j)} \mathcal{M}_{\sigma(j)}^{k} && \text{otherwise;}
\end{aligned}$$

then for all $1 \leqslant j \leqslant k{-}1$, $|\mathsf{prod}_{\mathscr{T}_j}(\mathcal{C}_j)|_{\mathscr{T}_{j+1}} \neq 0$, and $|\mathsf{prod}_{\mathscr{T}_k}(\mathcal{C}_k)| \neq 0$.

As for blind pebble transducers, observe that pumpability for last pebble transducers can be decided by ranging over tuples of transition monoid elements and letters.

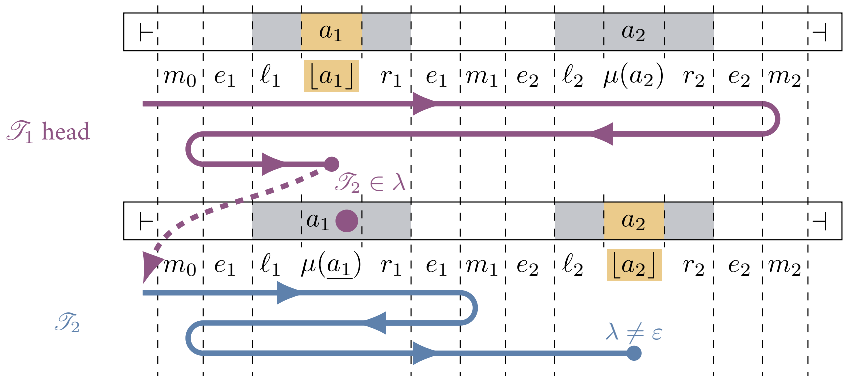


Figure 3.26: Pumpability in a last 2-pebble transducer.

Now, we provide an analogue of Claim 3.19, showing that our notion of pumpability is correct.

Claim 3.27 (Pumpability $\Rightarrow$ Growth)

Let $f\colon A^* \to B^*$ be computed by a pumpable last k-pebble transducer, then there exist $v_0, \dots, v_k \in A^*, u_1, \dots, u_k \in A^+$, such that $|f(v_0 u_1^X \cdots u_k^X v_k)| = \theta(X^k)$.

Proof. The proof is similar to that of Claim 3.19. We use the notations of Definition 3.25. For $1 \leqslant j \leqslant k$, let $w_j, w'_j \in A^*$ be such that $\mu(w_j) = \ell_j$ and $\mu(w'_j) = r_j$, $v_0, \dots, v_k \in A^*$ be such that $\mu(v_j) = m_j$ for all $1 \leqslant j \leqslant k$, $u_j := w_j a_j w'_j$ and $\underline{u_j} := w_j \underline{a_j} w'_j$ for $1 \leqslant j \leqslant k$.

To simply the notations, we assume that $\sigma\colon [1{:}k] \to [1{:}k]$ is the identity function. We first observe for all $X \geqslant 2$, that $|[\![\mathscr{T}_1]\!](v_0 u_1^X \cdots u_k^X v_k)|_{\mathscr{T}_2} \geqslant (X{-}2) \times \mathsf{prod}_{\mathscr{T}_1}(\mathcal{C}_1) \geqslant X{-}2$. In a similar fashion, for all $2 \leqslant j \leqslant k{-}1$, $X \geqslant 2$ and $1 \leqslant Y \leqslant X{-}1$, we have:

$$|[\![\mathscr{T}_j]\!](v_0 \cdots v_{j-1}(u_j^Y \underline{u_j} u_j^{X-Y-1}) v_j u_{j+1}^X \cdots u_k^X v_k)|_{\mathscr{T}_{j+1}} \geqslant X{-}2.$$

Observe that the use of $u_j^Y \underline{u_j} u_j^{X-Y-1}$ means that the equation holds independently from the factor u_j which is marked, i.e. the factor in which the parent call was done. In a similar way, we get:

$$|[\![\mathscr{T}_k]\!](v_0 \cdots v_{k-2}(u_{k-1}^Y \underline{u_{k-1}} u_{k-1}^{X-Y-1}) v_{k-1} u_k^X v_k)| \geqslant X{-}2.$$

Using the semantics of last k-pebble transducers, we conclude that $|[\![\mathscr{T}_1]\!](v_0 u_1^X \cdots u_k^X v_k)| \geqslant (X{-}2)^k$ for all $X \geqslant 2$, and the result follows. ◀

Now we state an analogue of Theorem 3.20, which precises Theorem 3.13.

Theorem 3.28 (Removing one last pebble layer)

Let $k \geqslant 2$ and $f : A^* \to B^*$ be a function computed by a last k-pebble transducer $\mathscr{L}$. The following conditions are equivalent:

(1) $|f(u)| = \mathcal{O}(|u|^{k-1})$;
(2) $\mathscr{B}$ is not pumpable;
(3) f can be computed by a blind $(k-1)$-pebble transducer.

Furthermore, this property is decidable and the construction is effective

Proof. Item (3) $\Rightarrow$ Item (1) is obvious and Item (1) $\Rightarrow$ Item (2) is Claim 3.27. Furthermore, we have observed above that pumpability is decidable. Item (2) $\Rightarrow$ Item (3) is shown (in an effective fashion) in Section 3.3.2. It is the main body of this proof. ◀

3.3.2 Removing a nested layer in a non-pumpable transducer

In Section 3.3.2, we show Item (2) $\Rightarrow$ Item (3) in Theorem 3.28. We follow the same proof sketch as in Section 3.2.2. Let us fix $k \geqslant 2$ and $\mathscr{L}$ a last k-pebble transducer which is not pumpable and computes a function $f : A^* \to B^*$. Our goal is to build a last $(k-1)$-pebble transducer for f. Let $\mu : (A \uplus \underline{A})^* \to \mathbb{T}$ be the transition morphism of $\mathscr{L}$ and $\varphi := \mu|_{A^*}$ be its restriction to A^*. Our new machine will be obtained as a composition (thanks to Proposition 3.11) of:

- the function $\mathsf{forest}_\varphi : A^* \to \mathsf{Forests}_\varphi^{3|\mathbb{T}|}$;
- the function $f \circ \mathsf{word}_\varphi^{3|\mathbb{T}|} : \mathsf{Forests}_\varphi^{3|\mathbb{T}|} \to B^*$, computed by a blind $(k-1)$-pebble transducer $\overline{\mathscr{L}}$. We allow the submachines to have lookarounds, as explained in Section 3.1.2.1.

We strongly advise the reader to read Section 3.2.2 as a warm-up before the current section. First, let us fix useful notations. Given $\mathcal{F} \in \mathsf{Forests}_\varphi^{3|\mathbb{T}|}$, $u = \mathsf{word}_\varphi^{3|\mathbb{T}|}(\mathcal{F})$ and $1 \leqslant i \leqslant |u|$, we denote by $\mathcal{F}{\blacksquare}i \in (A \uplus \underline{A} \uplus \{\langle, \rangle\})^*$ the forest $\mathcal{F}$ in which the i-th leaf is underlined (that is, the leaf $u[i]$ is changed to $\underline{u[i]}$). Beware that $\mathcal{F}{\blacksquare}i$ has no reason to be a μ-forest of $u{\bullet}i \in (A \uplus \underline{A})^*$. In order to obtain homogeneous notations for unmarked words, we let $u{\bullet}0$ (resp. $\mathcal{F}{\blacksquare}0$) be simply u (resp. $\mathcal{F}$).

3.3.2.1 Construction of $\overline{\mathscr{L}}$. Let us first describe the submachines of $\overline{\mathscr{L}}$:

- it has a submachine old-$\mathscr{T}$-from-(q_1,i_1)-to-(q_n,i_n) for each $\mathscr{T}$ a submachine of $\mathscr{L}$ and (q_1, i_1), (q_n, i_n) configurations of $\mathscr{T}$. This submachine is given an input $\mathcal{F}{\blacksquare}i$, where $\mathcal{F} \in \mathsf{Forests}_\varphi^{3|\mathbb{T}|}(u)$ for some $u \in A^*$. It mimics the (not necessarily accepting) n-run $(q_1, i_1) \to \cdots \to (q_n, i_n)$ of $\mathscr{T}$ labelled by $u{\bullet}i$. Its behavior is described in Algorithm 3.21 when $\mathscr{T}$ is not a leaf of $\mathscr{L}$. The case of a leaf is obtained by modifying Line 8 to output exactly the output (in B) of $\mathscr{T}$.
The reader might justifiably argue that we create an infinite number of submachines, since they are indexed by positions $1 \leqslant i_1, i_n \leqslant |u|$. In fact, such an indexing is only used to simplify the description of the functions, and we only build a finite number of submachines. Indeed, we shall always guarantee that the i_1-th and i_n-th leaves of the input $\mathcal{F}$ can be detected by the lookaround when the i-th leaf is marked. Hence the configurations (q_1, i_1) and (q_n, i_n) will be represented by using a bounded information, independently from the input $\mathcal{F}{\blacksquare}i$;
- it $\overline{\mathscr{L}}$ also has a submachine new-$\mathscr{T}$-from-(q_1,i_1)-to-(q_n,i_n) for $\mathscr{T}$ a submachine of $\mathscr{L}$ which is not a leaf. This submachine has the same behavior as old-$\mathscr{T}$-from-(q_1,i_1)-to-(q_n,i_n), while inlining portions of the nested calls of $\mathscr{T}$ within its own run. A major difference with the construction of

Section 3.2 is that here, we shall not inline entire calls but only well-chosen portions of them: this is the reason why our submachines are indexed by configurations.

Formally, new-$\mathscr{T}$-from-(q_1,i_1)-to-(q_n,i_n) is described in Algorithm 3.29. Whenever $\mathscr{T}$ is in position i_j of $u \bullet i$ and calls $\mathscr{T}'$, new-$\mathscr{T}$-from-(q_1,i_1)-to-(q_n,i_n) $(\mathcal{F} \blacksquare i)$ first slices the accepting run of $\mathscr{T}'$ on $\vdash u \bullet i_j \dashv$, with respect to $\mathsf{forest}_\varphi(u)$ and i_j, as explained in Definition 3.30. Then, it inlines the portions of ρ' which move on positions whose origins depend on $\mathsf{origin}_\mathcal{F}(i_j)$. On the other portions of runs, it makes a nested call, except if $\mathscr{T}'$ was a leaf of $\mathscr{L}$.

Algorithm 3.29: Submachines of the last $(k-1)$-pebble transducer $\overline{\mathscr{L}}$

Submachine old-$\mathscr{T}$-from-(q_1,i_1)-to-(q_n,i_n) $(\mathcal{F} \blacksquare i)$
  `/* Suppose that` $\mathscr{T}$ `has shape` $(A \uplus \underline{A}, C, Q, q_0, F, \delta, \lambda)$. `*/`
  $u := \mathsf{word}_\varphi^{3|\mathbb{T}|}(\mathcal{F})$ `/* Original unmarked input. */`
  $(q_1, i_1) \to \cdots \to (q_n, i_n) :=$ n-run of $\mathscr{T}$ from (q_1, i_1) to (q_n, i_n) over $u \bullet i$
  for $1 \leqslant j \leqslant n$ **do**
    if $\mathscr{T}' := \lambda(q_j, (u \bullet i)[i_j]) \neq \varepsilon$ **then**
      $(q'_1, i'_1) \to \cdots \to (q'_n, i'_{n'}) :=$ accepting n-run of $\mathscr{T}'$ labelled by $u \bullet i_j$
      Call submachine old-$\mathscr{T}'$-from-(q'_1,i'_1)-to-$(q'_{n'},i'_{n'})$ $(\mathcal{F} \blacksquare i_j)$
    end
  end

Submachine new-$\mathscr{T}$-from-(q_1,i_1)-to-(q_n,i_n) $(\mathcal{F} \blacksquare i)$
  `/* Suppose that` $\mathscr{T}$ `has shape` $(A, C, Q, q_0, F, \delta, \lambda)$. `*/`
  $u := \mathsf{word}_\varphi^{3|\mathbb{T}|}(\mathcal{F})$ `/* Original unmarked input. */`
  $(q_1, i_1) \to \cdots \to (q_n, i_n) :=$ n-run of $\mathscr{T}$ from (q_1, i_1) to (q_n, i_n) over $u \bullet i$
  for $1 \leqslant j \leqslant n$ **do**
    if $\mathscr{T}' := \lambda(q_j, (u \bullet i)[i_j]) \neq \varepsilon$ **then**
      $(q'_1, i'_1) \to \cdots \to (q'_{n'}, i'_{n'}) :=$ accepting n-run of $\mathscr{T}'$ labelled by $u \bullet i_j$
      $\ell_1, \dots, \ell_N :=$ slicing of $(q'_1, i'_1) \to \cdots \to (q'_{n'}, i'_{n'})$ with respect to $\mathcal{F}$ and i_j
      for $p = 1$ **to** $N-1$ **do**
        $m_1 := \ell_p$ and $m_2 := \ell_{p+1}-1$ `/* Bounds of a sub-n-run */`
        if $i'_{m_1}, \dots, i'_{m_2} \in \uparrow_{i_j}$ **then**
          `/*` $(q'_{m_1}, i'_{m_1}) \to \cdots \to (q'_{m_2}, i'_{m_2})$ `has bounded size. */`
          Inline the code of old-$\mathscr{T}$-from-(q'_{m_1},i'_{m_1})-to-(q'_{m_2},i'_{m_2}) $(\mathcal{F} \blacksquare i_j)$
        else if $i'_{m_1}, \dots, i'_{m_2} \in \downarrow_{i_j}$ **then**
          `/* Positions` $i'_{m_1}, \dots, i'_{m_2}$ `are ‘‘below’’` i_j `in` $\mathcal{F}$. `*/`
          Inline the code of old-$\mathscr{T}$-from-(q'_{m_1},i'_{m_1})-to-(q'_{m_2},i'_{m_2}) $(\mathcal{F} \blacksquare i_j)$
        else if $\mathscr{T}'$ is a leaf of $\mathscr{L}$ **then**
          `/* The output of` $\mathscr{T}'$ `along the n-run` $(q'_{m_1}, i'_{m_1}) \to \cdots \to (q'_{m_2}, i'_{m_2})$ `is empty by Lemma 3.34. */`
        else
          Call submachine new-$\mathscr{T}$-from-(q'_{m_1},i'_{m_1})-to-(q'_{m_2},i'_{m_2}) $(\mathcal{F} \blacksquare i_j)$
        end
      end
    end
  end

Definition 3.30 (Slicing)

Let $u \in A^+$, $\mathcal{F} \in \mathsf{Forests}_\varphi(u)$ and $1 \leqslant i \leqslant |u|$. We let:
- $\uparrow_i := \{1 \leqslant j \leqslant |u| \mid \mathsf{origin}_\mathcal{F}(i) \text{ observes } \mathsf{origin}_\mathcal{F}(j)\}$;
- $\downarrow_i := \{1 \leqslant j \leqslant |u| \mid \mathsf{origin}_\mathcal{F}(j) \text{ observes } \mathsf{origin}_\mathcal{F}(i)\}$.

Let $\rho = (q_1, i_1) \to \cdots \to (q_n, i_n)$ be a n-run of a 2DT $\mathscr{T}$ on $u\bullet i$. We build by induction the sequence $\ell_1, \ldots, \ell_N$, called the *slicing* of ρ with respect to $\mathcal{F}$ and i, by $\ell_1 := 1$ and:
- if $i_{\ell_j} \in \uparrow_i$ (resp. $i_{\ell_j} \in \downarrow_i \smallsetminus \uparrow_i$, resp. $i_{\ell_j} \in [1{:}|u|] \smallsetminus (\uparrow_i \cup \downarrow_i)$), then $\ell_{j+1} \geqslant \ell_j$ is defined as the smallest index such that $i_{\ell_{j+1}} \notin \uparrow_i$ (resp. $i_{\ell_{j+1}} \notin \downarrow_i \smallsetminus \uparrow_i$, resp. $i_{\ell_{j+1}} \in [1{:}|u|] \smallsetminus (\uparrow_i \cup \downarrow_i)$);
- if such an index does not exist, then $\ell_{j+1} := n{+}1$.

The slicing describes when ρ enters or leaves the sets $\uparrow_i$ and $\downarrow_i$, as depicted in Figure 3.31.

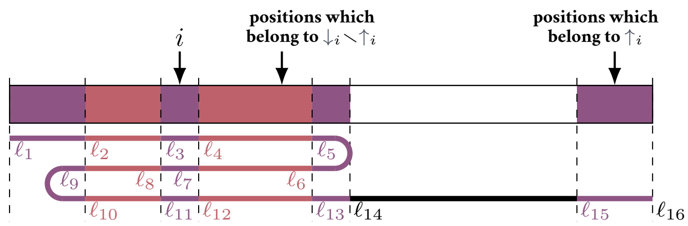


Figure 3.31: Slicing of the n-run ρ with respect to i and $\mathcal{F}$.

Finally, the transducer $\overline{\mathscr{L}}$ is obtained by defining new-$\mathscr{T}$-from-(q_1,i_1)-to-(q_n,i_n) as its head, where $\mathscr{T}$ is the head of $\mathscr{L}$ and (q_1, i_1), (q_n, i_n) are chosen so that n-run $(q_1, i_1) \to \cdots \to (q_n, i_n)$ is accepting. This head will be given an input of shape[3] $\mathcal{F}$■0. We also remove the submachines which are never called. It remains to justify that the construction is correct and defines a last $(k{-}1)$-pebble transducer.

The key property of $\overline{\mathscr{L}}$ is that it never makes a nested call in a position whose origin depends on the origin of the position of the previous call (which is the underlined position).

3.3.2.2 $\overline{\mathscr{L}}$ has $k{-}1$ nested layers. We say that a submachine of a last pebble transducer has *height* $h \geqslant 1$ if it is the head of a subtree of height h. Our goal is to show that the head of $\overline{\mathscr{L}}$ has height $k{-}1$, which is equivalent to saying that $\overline{\mathscr{L}}$ has $k{-}1$ nested layers.

We first show by induction that if $\mathscr{T}$ has height h in $\mathscr{B}$, then old-$\mathscr{T}$-from-(q_1,i_1)-to-(q_n,i_n) (if used) has height h in $\overline{\mathscr{L}}$ (this proof is easy). Second, we show by induction on $2 \leqslant h \leqslant k$ that if $\mathscr{T}$ has height h in $\mathscr{B}$, then new-$\mathscr{T}$-from-(q_1,i_1)-to-(q_n,i_n) (if used) has height $h{-}1$ in $\overline{\mathscr{B}}$. Indeed, the base case $h = 2$ is justified by Line 28 in Algorithm 3.29 (there can be no nested calls). For $h > 2$ the machine new-$\mathscr{T}$-from-(q_1,i_1)-to-(q_n,i_n) either inlines the code of old-$\mathscr{T}'$-from-(q'_{m_1},i'_{m_1})-to-(q'_{m_2},i'_{m_2}) (which has height $h{-}1$) or makes a nested call to new-$\mathscr{T}'$-from-(q'_{m_1},i'_{m_1})-to-(q'_{m_2},i'_{m_2}) (which has height $h{-}2$ by induction hypothesis), thus it has height $h{-}1$.

3.3.2.3 Each submachine of $\overline{\mathscr{L}}$ is a two-way transducer (with lookaheads). Apart from the representation of i_1 and i_n, it should be clear that each old-$\mathscr{T}$-from-(q_1,i_1)-to-(q_n,i_n) can be implemented by a 2DT. Indeed, it will move as before on the leaves of $\mathcal{F}$ while following variable $1 \leqslant i_j \leqslant |u|$. Now, we justify that each new-$\mathscr{T}$-from-(q_1,i_1)-to-(q_n,i_n) can also be implemented by a 2DT.

[3] Recall that $\mathcal{F}$■0 is just an homogenous notation for $\mathcal{F}$ with no marks.

First, note that since $\mathcal{F}$ has bounded height, the number N given by the slicing in Line 17 of Algorithm 3.29 is bounded by some $B \geqslant 0$. Furthermore, one can build a lookaround which detects the i'_{ℓ_p}-th leaf of $\mathcal{F}$ whenever the i_j-the leaf is underlined (the next result generalizes Claim 3.22).

Claim 3.32 (Slices can be detected)

For all submachine $\mathscr{T}$ which is not the head of $\mathscr{L}$, $a \in A \uplus \underline{A}$ and $1 \leqslant p \leqslant B$, one can build regular languages $L, R \subseteq (A \uplus \underline{A} \uplus \{\langle, \rangle\})^*$ such that:

- for all $u \in A^+$ and for all $1 \leqslant i \leqslant |u|$;
- for all $\mathcal{F} \in \mathsf{Forests}_\varphi^{3|\mathbb{T}|}(u)$ such that $(q_1, i_1) \to \cdots \to (q_n, i_n)$ is the accepting n-run of $\mathscr{T}$ labelled by $u \bullet i$, whose slicing with respect to $\mathcal{F}$ and i is $\ell_1, \ldots, \ell_N$ (with $N \leqslant B$);
- for all $1 \leqslant i' \leqslant |\mathcal{F}|$;

the following conditions are equivalent:

- $(\mathcal{F} \blacksquare i)[1{:}i'{-}1] \in L$, $(\mathcal{F} \blacksquare i)[i'] = a$ and $(\mathcal{F} \blacksquare i)[i'{+}1{:}|\mathcal{F}|] \in R$;
- $\mathcal{F}[i']$ is the i_{ℓ_p}-th leaf of $\mathcal{F}$ (i.e. it encodes position i_{ℓ_p} of u).

Proof. Recall that the slicing describe the indices when the positions of the n-run cross the borders between the sets $\uparrow_i$, $\downarrow_i \smallsetminus \uparrow_i$ and $[1{:}|u|] \smallsetminus (\uparrow_i \cup \downarrow_i)$. Since the behavior of a n-run can be described using regular languages (recall the transition monoid of a 2DT), we only need to show that the leaves whose positions encode the borders of $\uparrow_i$ and $\downarrow_i \smallsetminus \uparrow_i$ can be detected using regular languages.

For $\uparrow_i$ the result is clear since $|\uparrow_i|$ is a bounded. For $\downarrow_i \smallsetminus \uparrow_i$ we use Claim 3.33, which implies that this set is a bounded union of intervals (observe that it is also the case of $[1{:}|u|] \smallsetminus (\uparrow_i \cup \downarrow_i)$).

Claim 3.33 (Intervals of dependent positions)

Let $\mathfrak{t} := \mathsf{origin}_\mathcal{F}(i)$. Assume that $\mathfrak{t} \in \mathsf{Iters}_\mathcal{F}$ (since $\mathfrak{t}$ is an origin, it equivalent to $\mathfrak{t} \neq \mathcal{F}$) and let $\mathfrak{t}_1$ (resp. $\mathfrak{t}_2$) be its immediate left (resp. right) sibling, then:

$$\downarrow_i \smallsetminus \uparrow_i = [\mathsf{min}(\mathsf{Fr}_\mathcal{F}(\mathfrak{t}_1)){:}\ \mathsf{max}(\mathsf{Fr}_\mathcal{F}(\mathfrak{t}_2))] \smallsetminus \{\mathsf{Fr}_\mathcal{F}(\mathfrak{t}_1), \mathsf{Fr}_\mathcal{F}(\mathfrak{t}), \mathsf{Fr}_\mathcal{F}(\mathfrak{t}_2)\}.$$

Proof. Let us assume that $\mathfrak{t}_1$ and $\mathfrak{t}_2$ are iterable nodes of $\mathcal{F}$ (the other cases are similar). By considering the forest of Figure 2.30, one sees that $\downarrow_i$ is $[\mathsf{min}(\mathsf{Fr}_\mathcal{F}(\mathfrak{t}_1)){:}\ \mathsf{max}(\mathsf{Fr}_\mathcal{F}(\mathfrak{t}_2))]$. We conclude since $\mathfrak{t}$, $\mathfrak{t}_1$ and $\mathfrak{t}_2$ are the only nodes that observe $\mathfrak{t}$ and that $\mathfrak{t}$ observes. ◀

Therefore $\downarrow_i \smallsetminus \uparrow_i$ is an union of a bounded number of intervals (since the frontiers have bounded size). The borders of these intervals can easily be detected using regular languages. ◀

This analysis justifies why each i_{ℓ_p} can be encoded in a bounded way (thus it is also the case of $i_{\ell_p - 1}$ since it belongs to $\{i_{\ell_p}{-}1, i_{\ell_p}, i_{\ell_p}{+}1]\}$) while being detectable by a lookaround. It remains to explain how the "Inline the code" instructions of Lines 23 and 26 are implemented:

- if $i'_{m_1}, \ldots, i'_{m_2} \in \uparrow_{i_j}$, then $m_2{-}m_1$ must be bounded (because $|\uparrow_i|$ is bounded). Hence the submachine old-$\mathscr{T}'$-from-(q'_{m_1}, i'_{m_1})-to-(q'_{m_2}, i'_{m_2}) $(\mathcal{F} \blacksquare i_j)$ performs a bounded run. We inline its code by producing its bounded output without moving[4] from the current i_j-th leaf. However, when $\mathscr{T}'$ calls some $\mathscr{T}''$ on position i'_ℓ, we need to call some old-$\mathscr{T}''$-from-$(\_,\_)$-to-$(\_,\_)$ $(\mathcal{F} \blacksquare i'_\ell)$. But we cannot do this operation, since we are in leaf i_j and not in i'_ℓ. The solution is that the inlined

[4]The reader may suggest to make the submachine new-$\mathscr{T}'$-from-(q'_{m_1}, i'_{m_1})-to-(q'_{m_2}, i'_{m_2}) $(\mathcal{F} \blacksquare i)$ directly move on the leaves $i'_{m_1}, \ldots, i'_{m_2}$ of $\mathcal{F}$ to perform the inlining of Line 26. However, this idea is not correct. Indeed, if the 2DT does so, it will not be able to go back to position i_j afterwards: since $\mathsf{origin}_\mathcal{F}(i_{m_1})$ is roughly an ancestor of $\mathsf{origin}_\mathcal{F}(i_j)$, we intuitively lose information when going from leaf i_j to leaf to i'_{m_1}.

code calls a new submachine old-$\mathscr{T}''$-from-(_, _)-to-(_, _)-pebble-i'_ℓ which behaves as follows: given an input $\mathcal{F}$■i_j, it simulates an execution of old-$\mathscr{T}$-from-(_, _)-to-(_, _) ($\mathcal{F}$■i'_ℓ). In other words, it makes "as though" the i'_ℓ-th leaf was marked, instead of the i_j-th one. As above, since $i'_\ell \in \uparrow_{i_j}$, it can be encoded as a bounded information (thus we only create a finite number of submachines) and one can build a lookaround which enables to if the current position is this leaf.

- if $i'_{m_1}, \dots, i'_{m_2} \in \downarrow_{i_j} \setminus \uparrow_{i_j}$, then the nodes $\mathsf{origin}_\mathcal{F}(i'_{m_1}), \dots, \mathsf{origin}_\mathcal{F}(i'_{m_2})$ are roughly below $\mathsf{origin}_\mathcal{F}(i)$ in $\mathcal{F}$ (see Figure 2.30). Thus we inline old-$\mathscr{T}'$-from-(q'_{m_1},i'_{m_1})-to-(q'_{m_2},i'_{m_2}) ($\mathcal{F}$■i_j), by moving[5] on leaves $i'_{m_1}, \dots, i'_{m_2}$. We keep track of the (bounded) height of $\mathsf{origin}_\mathcal{F}(i_j)$ above the current $\mathsf{origin}_\mathcal{F}(i'_\ell)$. We use a lookaround to detect when (q'_{m_2}, i'_{m_2}) is reached, and we finally go back to leaf i_j.

3.3.2.4 $\overline{\mathscr{L}}$ is correct. It remains to justify that $\overline{\mathscr{L}}$ computes the function $f \circ \mathsf{word}_\mu^{3|\mathbb{T}|}$. First, it is easy to see that the old-$\mathscr{T}$-from-(_, _)-to-(_, _) ($\mathcal{F}$■i) are correct with respect to their (informal) specification. So are the aforementioned old-$\mathscr{T}''$-from-(_, _)-to-(_, _)-pebble-_.

For showing the correctness of new-$\mathscr{T}$-from-(q_1,i_1)-to-(q_2,i_2) ($\mathcal{F}$■i), we only need to show that when Line 28 of Algorithm 3.29 is reached, then the output of $\mathscr{T}'$ along the n-run $(q'_{m_1}, i'_{m_1}) \to \cdots \to (q'_{m_2}, i'_{m_2})$ labelled by u•i_j must be empty (as we make no calls in this case). This result is obtained by observing that otherwise the conditions of Lemma 3.34 hold, which yields a contradiction.

Lemma 3.34 (Key lemma for removing one last pebble layer)

Let $u \in A^+$ and $\mathcal{F} \in \mathsf{Forests}_\varphi(u)$. Assume that there exists a sequence $\mathscr{T}_1, \dots, \mathscr{T}_k$ of submachines of $\mathscr{L}$ and a sequence of positions $1 \leqslant i_1, \dots, i_k \leqslant |u|$ such that:

- $\mathscr{T}_1$ is the head of $\mathscr{L}$;
- $|\mathsf{prod}^u_{\mathscr{T}_1}(i_1)|_{\mathscr{T}_2} \neq 0$ and $|\mathsf{prod}^{u \bullet i_{k-1}}_{\mathscr{T}_k}(i_k)| \neq 0$;
- for all $2 \leqslant j \leqslant k{-}1$, $|\mathsf{prod}^{u \bullet i_{j-1}}_{\mathscr{T}_j}(i_j)|_{\mathscr{T}_{j+1}} \neq 0$;
- for all $1 \leqslant j \leqslant k{-}1$, $\mathsf{origin}_\mathcal{F}(i_j)$ and $\mathsf{origin}_\mathcal{F}(i_{j+1})$ are independent;

Then $\mathscr{L}$ is pumpable.

Proof. The proof is similar to that of Lemma 3.23. The goal is to show that for $1 \leqslant j \leqslant k$, the $\mathfrak{t}_j := \mathsf{origin}_\mathcal{F}(i_j)$ can be chosen pairwise independent (in the hypothesis, it is only assumed for the consecutive pairs $(\mathfrak{t}_j, \mathfrak{t}_{j+1})$), since Lemma 2.33 enables to conclude if it is the case.

For this, we show once more how to make the number of dependent nodes decrease strictly, while preserving the properties of Lemma 3.34. Assume that $\mathfrak{t}_{\ell_1}$ observes $\mathfrak{t}_{\ell_2}$ for some $1 \leqslant \ell_1 \neq \ell_2 \leqslant k$ (note that ℓ_1 and ℓ_2 cannot be consecutive). To simplify the proof, we assume that $\mathfrak{t}_{\ell_2}$ is an ancestor of $\mathfrak{t}_{\ell_1}$. We let $u' \in A^+$, $\mathcal{F}' \in \mathsf{Forests}_\varphi(u')$, $\mathfrak{t}'_1, \dots, \mathfrak{t}'_k \in \mathsf{Nodes}_{\mathcal{F}'}$ and $1 \leqslant i'_1, \dots, i'_k \leqslant |u'|$ be defined as the proof of Lemma 3.23 (recall Figure 3.24).

Now, we justify that $\mathsf{prod}^{u \bullet i_{j-1}}_{\mathscr{T}_j}(i_j) = \mathsf{prod}^{u' \bullet i'_{j-1}}_{\mathscr{T}_j}(i'_j)$ for all $2 \leqslant j \leqslant k$. This is the only difference with the proof of Lemma 3.23. For this, we let $v := u \bullet i_{j-1}$ and $v' := u' \bullet i'_{j-1}$ and we show that $\mu(v[1{:}i_j{-}1]) \lfloor v[i_j] \rfloor \mu(v[i_j{+}1{:}|v|]) = \mu(v'[1{:}i'_j{-}1]) \lfloor v'[i'_j] \rfloor \mu(v'[i'_j{+}1{:}|v'|])$ by distinguishing the following cases:

- if both i_{j-1} and i_j belong to the subtree rooted in $\mathfrak{t}_{\ell_2}$, then $j \neq \ell_2$ (since otherwise $\mathsf{origin}_\mathcal{F}(i_j)$ and $\mathsf{origin}_\mathcal{F}(i_{j-1})$ would be dependent) and similarly $j{-}1 \neq \ell_2$. The result holds because we iterate an iterable node and $\mathfrak{t}_{i'_{j-1}}$ and $\mathfrak{t}_{i'_j}$ are still in the same subtree;
- if both i_{j-1} and i_j do not belong to the subtree rooted in $\mathfrak{t}_{\ell_2}$, the argument is similar;

[5] The technique of the first item cannot be applied here, since the length of the inlined run $(m_2{-}m_1)$ may not be bounded.

- if i_{j-1} is in the subtree rooted in $\mathfrak{t}_{\ell_2}$ but not i_j (the converse is similar), we use once more the fact $\mathsf{origin}_{\mathcal{F}}(i_j)$ and $\mathsf{origin}_{\mathcal{F}}(i'_j)$ are independent. Indeed, it implies that i_j cannot be "below" an immediate sibling of $\mathfrak{t}_{\ell_2}$. Hence duplicating the iterable node corresponding to $\mathfrak{t}_{\ell_2}$ will not change the monoid value between positions i_{j-1} and i_j. ◀

Thus $\overline{\mathscr{L}}$ is a blind $(k-1)$-pebble transducer which computes $f \circ \mathsf{word}_{\varphi}^{3|\mathbb{T}|}$. The result follows.

3.4 Discussion: beyond one visible pebble

In this section, we claim that the correspondence between asymptotic growth and nested layers for last pebble transducers is tight, in the sense that it fails for more complex subclasses of pebble transducers[6].

It is not hard (however quite tedious) to modify the definition of a last k-pebble transducer (Definition 3.8) in order to define a model of *last-last k-pebble transducer*. The latter consists in a pebble transducer where the position of the two previous calls are marked on the input of a submachine. In other words, the last pebble is visible, but also the penultimate one (hence the "last-last"). Note that for $k = 1, 2$ and 3, a last-last k-pebble transducer is exactly the same as a k-pebble transducer.

As an immediate consequence of Theorem 1.48, we see that the function inner-squaring is such that $|\mathsf{inner\text{-}squaring}(u)| = \mathcal{O}(|u|^2)$ and can be computed by a last-last 3-pebble transducer, but not by a last-last 2-pebble transducer. Therefore the connection between minimal recursion height and growth of the output fails. However, this result is somehow artificial. Indeed, a last-last 2-pebble transducer is a somehow degenerate case, since it can only see one last pebble. More interestingly, we extend the failure result to each level of the hierarchy, by re-using the counterexample of Theorem 1.51.

Proposition 3.35 (Quadratic growth can require k layers)

Let $k \geqslant 1$. The function $\mathsf{alternating\text{-}square}_k$ can be computed by a last-last $(2k+1)$-pebble transducer and is such that $|\mathsf{alternating\text{-}square}_k(u)| = \mathcal{O}(|u|^2)$.

However, $\mathsf{alternating\text{-}square}_k$ cannot be computed by a last-last $2k$-pebble transducer

Proof. Thanks to Theorem 1.51, we only have to justify that $\mathsf{alternating\text{-}square}_k$ can be computed by a last-last $(2k+1)$-pebble transducer. For $k = 2$, we observing that only the two last loop indices are useful when executing Algorithm 1.50, since we range over children. This observation can be generalized to any $k \geqslant 1$ with loop indices $i_1, j_1, i_2, j_2, \dots, i_k, j_k$. ◀

One mystery may remain for the amazed reader: concretely, why is it impossible to generalize the proof of Section 3.3 to pebble transducers (or last-last pebble transducers)? To explain this, let us consider a 3-pebble transducer denoted $\mathscr{P} = \mathscr{T}_1\langle\mathscr{T}_2\langle\mathscr{T}_3\rangle\rangle$. One can get inspired by last 3-pebble transducers to define relevant notions of *transition morphism* and *pumpability* for 3-pebble transducers. Suppose that $\mathscr{T}_1\langle\mathscr{T}_2\langle\mathscr{T}_3\rangle\rangle$ is not pumpable. Let us try to show that it is equivalent to a 2-pebble transducer $\overline{\mathscr{P}}$ by following a proof similar to that of Section 3.3.2

Let us describe the behavior of $\overline{\mathscr{P}}$ when simulating $\mathscr{T}_1\langle\mathscr{T}_2\langle\mathscr{T}_3\rangle\rangle$. If $\mathscr{T}_1$ calls $\mathscr{T}_2$ in a position i_1, then it inlines in $\mathscr{T}_1$ the portions of run of $\mathscr{T}_2$ in positions i_2, whose origin depends on that of i_1. However, the portions of runs of $\mathscr{T}_2$ in the positions i_2, whose origin is independent from that of i_1, cannot be inlined, thus they correspond to a nested call. Now, if $\mathscr{T}_2$ calls $\mathscr{T}_3$ in such an independent position i_2, then $\overline{\mathscr{P}}$ should inline the whole run of $\mathscr{T}_3$ in $\mathscr{T}_2$. This run can be split in 3 cases:

[6] Hence last pebble transducer is somehow the "last" model for which the correspondence between growth and number of layers holds. This observation was meant to be a pun in the title of [Dou23] (*Pebble minimization: the last theorems*) together with the (more personal) fact that this paper is likely to be the last non-co-authored research paper of the author.

(1) the portions of the run of $\mathscr{T}_3$ in positions i_3, whose origin in independent from that of i_1 or from that of i_2. Along these portions, $\mathscr{T}_3$ must produce empty outputs, due to pumpability;
(2) the portions of the run of $\mathscr{T}_3$ in positions i_3 origin depends on that of i_2. These portions can be inlined by using the techniques presented in Section 3.3.2;
(3) the portions of the run of $\mathscr{T}_3$ in positions i_3 whose origin depends on that of i_1. These portions cannot be inlined by $\mathscr{T}_2$: indeed, if $\mathscr{T}_2$ moves to such positions, it will be unable to go back to i_2 afterwards (this information was lost). This is precisely why the proof would fail.

As a conclusion, let us concretely illustrate the obstruction mentioned in Item (3) by re-using the counterexample inner-squaring from Theorem 1.48. After Example 3.36, the reader should be convinced that we have provided a good understanding of the limits of optimization for pebble transducers.

Example 3.36 (Inner squaring)

Recall that inner-squaring : $u_1\# \cdots \# u_n \mapsto (u_1\#)^n \cdots (u_n\#)^n$ can be computed by a 3-pebble transducer $\mathscr{P} = \mathscr{T}_1\langle \mathscr{T}_2 \langle \mathscr{T}_3 \rangle\rangle$. Roughly, $\mathscr{T}_1$ drops a pebble on u_i to indicate that it is currently being written, $\mathscr{T}_2$ drops a pebble in u_j to indicate that it produces the j-th copy of u_i. Finally, $\mathscr{T}_3$ goes on the factor u_i and outputs it. Thus $\mathscr{T}_3$ is exactly producing an output in positions which "depend" on the position of the first pebble: this is precisely the case of Item (3).

Chapter 4

Streaming computations and marble transducers

> Elle a semblé sourire, et, plus audacieux,
> On se dit : « L'Immortelle est peut-être une femme ! »
> Et vers la main de marbre on tend sa main de flamme.
>
> Théophile Gautier, « Ne touchez pas aux marbres »,
> *Un douzain de sonnets*

In this chapter, we present yet another variant of pebble transducers, named *marble transducers* after [EHV99]. Informally, a *k-marble transducer* is a last k-pebble transducer in which a *submachine* is only allowed to move on the prefix which ends in the calling position. Hence the size of the input decreases at each nested call. We shall extend marble transducers to *recursive marble transducers*, where the nested calls are no longer required to describe a bounded tree (we allow recursion between the *submachines*). Such recursive machines can produce outputs whose size is exponential in the input. The relationship between marble transducers and the models of the previous chapters is presented in Figure 4.1.

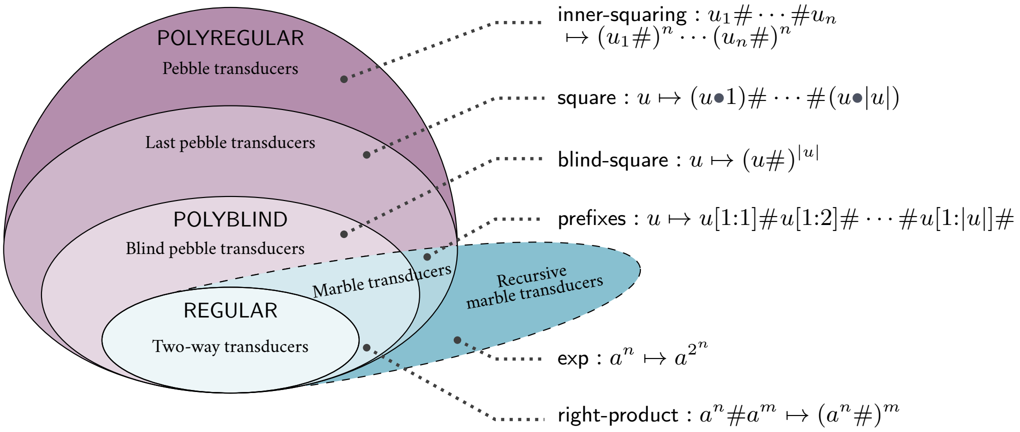


Figure 4.1: Classes of functions studied in Chapters 3 and 4.

The main reason for introducing marble transducers is presented in Section 4.2: we show that recursive marble transducers have the same expressive power as *streaming string transducers*. The latter is a celebrated model from [AC10] which consists in a one-way automaton using registers to produce its output string. Since this machine is one-way, it processes its input in a "streaming" fashion, which is meaningful for practical applications. In Section 4.3, we describe a syntactic restriction on streaming string transducers, called *k-layeredness*, which makes them equivalent to k-marble transducers. In particular, we recover a classical result showing that 1-layered (also known as *copyless*, since the value of a register cannot be duplicated) streaming string transducers describe the class of regular functions.

The third main result of this chapter, presented in Section 4.4, shows that streaming string transducers (and therefore marble transducers and recursive marble transducers) can be optimized, i.e. that the number of nested layers can be minimized. We also show that the connection between asymptotic growth and nested layers holds: a function f computed a recursive marble transducer can be computed by a k-marble transducer if and only if $|f(u)| = \mathcal{O}(|u|^k)$. Interestingly, the proof techniques are very specific to streaming string transducers and significantly different from those of Chapter 3: we no longer use factorization forests but *weighted automata*. We thus claim that streaming string transducers shed a new light on understanding the asymptotic growth of nested two-way transducers.

Finally, we discuss in Section 4.5 what happens when allowing recursion for the aforementioned models of pebble transducers, last pebble transducers and blind pebble transducers.

The contributions presented in this chapter are based on the main theorems of [DFG20].

4.1 Marble transducers and recursion

Over trees, *marble automata* were first introduced as a variant of pebble automata in [EHV99]. Their definition was inspired by the "checking tree pushdown transducers" from [ERS78]. We first describe in Section 4.1.1 a model of *k-marble transducer* by adapting the definitions of k-pebble transducers and their variants. In Section 4.1.2, we describe a more expressive model called *recursive marble transducer*.

4.1.1 Marble transducers

As mentioned above, for $k \geqslant 1$ a *k-marble transducer* can be seen as a last k-pebble transducer in which the a submachines cannot use the whole input, but only its prefix which ends in the position of the call. The behavior of a 3-marble transducer is depicted in Figure 4.3 (compare with Figure 3.9).

Definition 4.2 (Marble transducer)

Let $k \geqslant 1$ and $\mathscr{T}$ be a normalized 2DT with input alphabet A. We say that $\mathscr{M}$ is a *k-marble transducer* with input alphabet A, output alphabet B and *head* $\mathscr{T}$ if:

- either $k = 1$, $\mathscr{M} = \mathscr{T}$ and it has output alphabet B;
- or $k \geqslant 2$, $\mathscr{M}$ is a tree $\mathscr{T}\langle \mathscr{M}_1 \rangle \cdots \langle \mathscr{M}_p \rangle$ with $p \geqslant 1$ and:
  - the subtrees $\mathscr{M}_1, \dots, \mathscr{M}_p$ are $(k-1)$-marble transducers with input alphabet A, output alphabet B, and respective heads $\mathscr{T}_1, \dots, \mathscr{T}_p$;
  - $\mathscr{T}$ has output alphabet $\{\mathscr{T}_1, \dots, \mathscr{T}_p\}$.

If $\mathscr{T}$ is the head of the k-marble transducer $\mathscr{M}$, we define the function computed by $\mathscr{T}$ within $\mathscr{M}$, denoted $[\![\mathscr{T}]\!] \colon A^* \to B^*$, by induction (in a similar way to pebble transducers):

- if $k = 1$, then $[\![\mathscr{T}]\!] := [\mathscr{T}] \colon A^* \to B^*$ follows the usual 2DT semantics;

- otherwise $\mathscr{T}$ has output alphabet $T := \{\mathscr{T}_1, \dots, \mathscr{T}_p\}$ and the functions $[\![\mathscr{T}_1]\!], \dots, [\![\mathscr{T}_1]\!]$ have been defined by induction. Let $g\colon A^* \to (T \times \mathbb{N})^*$ be the function computed by $\mathscr{T}$ in origin semantics. Given $u \in A^*$, if $g(u) = (t_1, i_1) \cdots (t_n, i_n)$, then we let:

$$[\![\mathscr{T}]\!](u) := [\![t_1]\!](u[1{:}i_1{-}1]) \cdots [\![t_n]\!](u[1{:}i_n{-}1]).$$

The function $f\colon A^* \to B^*$ computed by $\mathscr{M}$ is defined as $[\![\mathscr{T}]\!]$ for its head $\mathscr{T}$. We say that a 2DT $\mathscr{T}$ is a *submachine* of the marble transducer $\mathscr{M}$ if $\mathscr{T}$ labels a node in the tree structure of $\mathscr{M}$. We generalize the notation $[\![\mathscr{T}]\!]$ to any submachine $\mathscr{T}$ of $\mathscr{M}$, by observing that it is the head of a subtree.

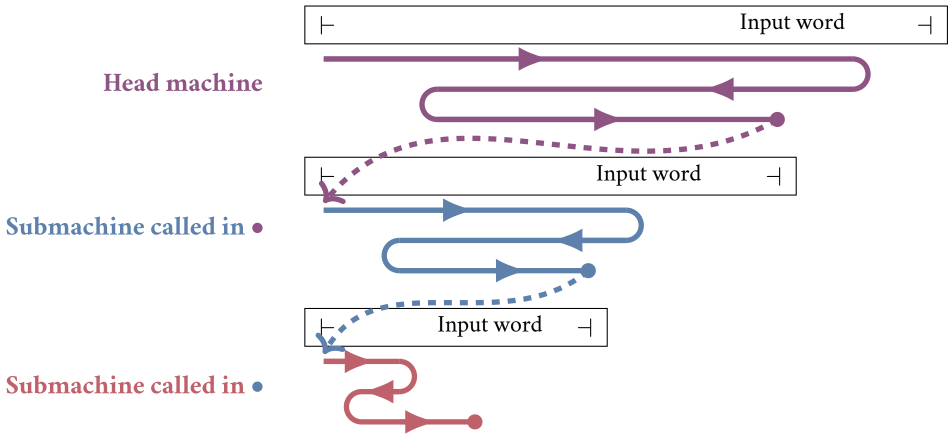


Figure 4.3: Behavior of a 3-marble transducer.

Example 4.4 (Right product)

Let $A := \{a, \#\}$. The function right-product: $a^m\#a^n \mapsto (a^m\#)^n$ where $\#$ is a fresh symbol can be computed by a 2-marble transducer $\mathscr{T}_1\langle\mathscr{T}_2\rangle$ where $\mathscr{T}_1$ calls the submachine $\mathscr{T}_2$ in each position of a^n, and $\mathscr{T}_2$ outputs $a^m\#$ each time. Observe that right-product is polyblind.

Example 4.5 (Prefixes)

The function prefixes: $u \mapsto u[1{:}1]\#u[1{:}2]\# \cdots \#u[1{:}|u|]\#$ can be computed by a 2-marble transducer. Recall that prefixes is not polyblind by Proposition 3.14.

We use the term *marble transducer* to denote a k-marble transducer for some $k \geqslant 1$.

4.1.1.1 Robustness and variants of the model.

One can define variants of the k-marble transducer model, in the spirit of the variants for k-pebble transducers described in Section 1.3.2 (that is, allowing submachines with lookarounds, or non-total submachines, or side effects, or output in the inner nodes). Such features do not modify the expressiveness of k-marble transducers for $k \geqslant 1$.

The next result deals with composition properties. It is obtained by easily leveraging standard proofs techniques, e.g. those of Theorem 1.31 or Theorems 1.43 and 3.6.

Proposition 4.6 (Composition with regular and rational functions)

For all $k \geqslant 1$, the class of functions computed by k-marble transducers is effectively closed under post-composition by regular functions and under pre-composition by rational functions.

We shall see in Proposition 4.21 that this class of functions is not closed under composition, nor even under pre-composition by regular functions. Intuitively, this is due to the fact that marble transducers are not a symmetrical model with respect to mirror images, since they make nested calls "on the left".

4.1.2 Recursive marble transducers

In this section, we present an extension of marble transducers, which are built by turning their tree structure into a graph structure. In other words, we allow the submachines not only to call their children in a tree, but to call *any* other submachine, including themselves. In yet other words, we enable the submachines to perform recursive calls, hence we name this model *recursive marble transducer*.

Definition 4.7 (Recursive marble transducer)

A *recursive marble transducer* $\mathscr{M}$ with input alphabet A and output alphabet B consists of:

- a finite collection $\mathscr{T}_1, \dots, \mathscr{T}_p$ of normalized 2DT with input alphabet A and output alphabet $\{\mathscr{T}_1, \dots, \mathscr{T}_p\} \uplus B$, called the *submachines* of $\mathscr{M}$;
- a distinguished $\mathscr{T} \in \mathscr{T}_1, \dots, \mathscr{T}_p$ called the *head* of $\mathscr{M}$.

The functions computed by the submachines $\mathscr{T}_1, \dots, \mathscr{T}_p$ within $\mathscr{M}$, are defined in a mutually recursive fashion and denoted by $[\![\mathscr{T}_1]\!], \dots, [\![\mathscr{T}_p]\!] : A^* \to B^*$. This recursion is well-founded since the size of the input strictly decreases when making recursive calls.

Formally, let $T := \{\mathscr{T}_1, \dots, \mathscr{T}_p\}$. Given $1 \leqslant j \leqslant p$, let $g \colon A^* \to ((T \uplus B) \times \mathbb{N})^*$ be the function computed by $\mathscr{T}_j$ in origin semantics. Given $u \in A^*$, if $g(u) = (t_1, i_1) \cdots (t_n, i_n)$, then we let:

$$[\![\mathscr{T}_j]\!](u) := [\![t_1]\!](u[1{:}i_1{-}1]) \cdots [\![t_n]\!](u[1{:}i_n{-}1]).$$

where $[\![b]\!]$ for $b \in B$ denotes the constant function $v \mapsto b$.

The function $f \colon A^* \to B^*$ computed by $\mathscr{M}$ is defined as $[\![\mathscr{T}]\!]$ for its head $\mathscr{T}$. Contrary to all of the models described in Chapters 1 and 3, recursive marble transducers can produce outputs whose size is exponential in the input, as explained in Examples 4.8 and 4.9.

Example 4.8 (Exponential output)

The function $\mathsf{exp} \colon a^n \mapsto a^{2^n}$ can be computed by a recursive marble transducer $\mathscr{M} = \{\mathscr{T}\}$, where $\mathscr{T}$ outputs a on input ε, and otherwise calls itself twice from the last position of its input.

Example 4.9 (Right exponential)

Let $A := \{a, \#\}$. The function $\mathsf{right\text{-}exp} \colon a^m \# a^n \mapsto (a^m \#)^{2^n}$ where $\#$ is a fresh symbol can be computed by a recursive marble transducer inspired by that of Example 4.8.

4.1.2.1 Robustness and variants of the model.

Note that marble transducers are a particular case of recursive marble transducers. Furthermore, one can show that if f is computed by a recursive marble transducer, then $|f(u)| = 2^{\mathcal{O}(|u|)}$. As a immediate consequence, the class of functions computed by such machines is not closed under composition (consider e.g. the function $\mathsf{exp} \circ \mathsf{exp}$).

It is easy to see that allowing the submachines of recursive marble transducers to use lookarounds does not modify the expressiveness of the model. However, non-total submachines or side effects (see Section 1.3.2.3) may raise more issues, since one cannot proceed by induction on the tree of calls (as claimed in Section 1.3.2.2) to show that the domain of the function is a regular language. In fact, this is still the case. Indeed, the model of *marble automata* introduced in [EHV99] coincides with the underlying automata of our recursive marble transducers when allowing side effects, and it follows from in [EHV99, Proposition 4.2] that this model captures exactly regular languages[1].

Another easy consequence of [EHV99, Proposition 4.2] is that recursive marble transducers preserve regular languages by inverse images. Observe that this result could already be deduced in the case of marble transducers, as a particular case of pebble transducers and using Proposition 1.41.

Proposition 4.10 (Regular pre-images)

Let $f\colon A^* \rightarrow B^*$ be computed by a recursive marble transducer and $L \subseteq B^*$ be a regular language. Then $f^{-1}(L) \subseteq A^*$ is (effectively) a regular language.

4.1.3 Optimization theorems

We are ready to state the main optimization results of this chapter, which connect the asymptotic growth of marble transducers to the minimal number of layers needed to represent a function. The following Theorems 4.11 and 4.12 both originate from [DFG20, Theorem 17].

Theorem 4.11 (Optimization of marble transducers)

Let $1 \leqslant \ell \leqslant k$ and $f\colon A^* \rightarrow B^*$ be computed by a k-marble transducer. Then f can be computed by an ℓ-marble transducer if and only if $|f(u)| = \mathcal{O}(|u|^{\ell})$. This property is decidable. If it holds, one can build an ℓ-marble transducer which computes f.

It is very likely that a proof of Theorem 4.11 can be done by following the sketch of Section 3.3 and using factorization forests. However, in the current chapter we shall rely on different techniques and use the forthcoming correspondence between marble transducers and *layered streaming string transducers*. These techniques are very specific to marble transducers, furthermore they can naturally be generalized to optimize recursive marble transducers, yielding Theorem 4.12.

Theorem 4.12 (Optimization of recursive marble transducers)

Let $f\colon A^* \rightarrow B^*$ be computed by a recursive marble transducer. Then f can be computed by a marble transducer if and only if $|f(u)| = \mathcal{O}(|u|^k)$ for some $k \geqslant 0$. This property is decidable. If it holds, one can build a marble transducer which computes f.

Proof of Theorems 4.11 and 4.12. Given a recursive marble transducer computing a function f, we transform it into a *deterministic streaming string transducer* (DSST) using Theorem 4.17. Then, we apply Theorem 4.41 to determine the least $k \geqslant 1$ such that $|f(u)| = \mathcal{O}(|u|^k)$ if it exists, and in this case we build a *(k,K)-bounded* DSST which computes f. Finally, we convert this machine in a k-marble transducer thanks to Item (1) $\Rightarrow$ Item (2) of Theorem 4.34 (shown in Section 4.3.4). ◀

[1] Their result is in fact more general, since it deals with regular tree languages.

4.2 Streaming string transducers

The *streaming string transducer* model was introduced by Alur and Cerný [AC10]. Such machines can produce outputs whose size is exponential in the input size. The goal of this section is to describe this model and show that its expressive power is the same as that of recursive marble transducers.

4.2.1 Streaming string transducers of finite words

Intuitively, a *streaming string transducer* consists of a one-way deterministic automaton enriched with a finite set $\mathfrak{R}$ of registers that store strings over some output alphabet B. This machine has nothing to do with the model of *register automaton* (see e.g. [NSV04]): the latter works over infinite alphabets and can compare the registers to select the transitions. It will never be the case here: the registers cannot be read, their only purpose is to contain portions of the final output. These registers are modified using *substitutions*, i.e. functions of type $\mathfrak{R} \to (B \uplus \mathfrak{R})^*$. We denote by $\mathcal{S}_{\mathfrak{R}}^B$ the set of these substitutions. A substitution s can be extended as a morphism from $(B \uplus \mathfrak{R})^*$ to $(B \uplus \mathfrak{R})^*$ by mapping each $\mathfrak{r} \in \mathfrak{R}$ to $s(\mathfrak{r})$ and each $b \in B$ to itself. Substitutions can be composed, as explained in Example 4.13.

Example 4.13 (Substitutions)

Let $\mathfrak{R} := \{\mathfrak{r}, \mathfrak{s}\}$ and $B := \{b\}$. Consider the substitutions $s_1 := \mathfrak{r} \mapsto b, \mathfrak{s} \mapsto b\mathfrak{r}\mathfrak{s}b$ and $s_2 := \mathfrak{r} \mapsto \mathfrak{r}b, \mathfrak{s} \mapsto \mathfrak{r}\mathfrak{s}$, then $s_1 \circ s_2(\mathfrak{r}) = s_1(\mathfrak{r}b) = bb$ and $s_1 \circ s_2(\mathfrak{s}) = s_1(\mathfrak{r}\mathfrak{s}) = bb\mathfrak{r}\mathfrak{s}b$.

As their name suggest, streaming string transducers process their input in a “streaming” fashion, i.e. in a single pass from left to right, contrary to two-way transducers.

Definition 4.14 (Streaming string transducer)

A *deterministic streaming string transducer* (DSST) $\mathscr{S} = (A, B, Q, q_0, \delta, \mathfrak{R}, \lambda, \tau)$ consists of:

- an input alphabet A and an output alphabet B;
- a finite set of states Q with an initial state $q_0 \in Q$;
- a transition function $\delta\colon Q \times A \rightharpoonup Q$;
- a finite set $\mathfrak{R}$ of registers;
- an initial function $\iota\colon \mathfrak{R} \to B^*$;
- a register update function $\lambda\colon Q \times A \to \mathcal{S}_{\mathfrak{R}}^B$;
- an output function $\tau\colon Q \to (\mathfrak{R} \cup B)^*$.

In Section 2.1.1, we have defined the extended transition function and extended output function for 2DT. We define extended functions for DSST in a similar (even easier) fashion:

- the *extended transition function* $\delta^*\colon Q \times A^* \to Q$ defined inductively by $\delta^*(q, \varepsilon) = q$ for all $q \in Q$, and $\delta^*(q, ua) = \delta(\delta^*(q, u), a)$ for all $q \in Q, a \in A$ and $u \in A^*$;
- the *extended output function* $\lambda^*\colon Q \times A^* \to \mathcal{S}_{\mathfrak{R}}^B$ defined inductively by $\lambda^*(q, \varepsilon)(\mathfrak{r}) = \mathfrak{r}$ for all $q \in Q$ and $\mathfrak{r} \in \mathfrak{R}$, and $\lambda^*(q, ua) = \lambda^*(q, u) \circ \lambda(\delta^*(q, u), a)$ for all $q \in Q, a \in A$ and $u \in A^*$. Intuitively, this construction describes “the substitution applied when starting from state q and reading u ”. When reading new letters, we add substitutions “on the right”, which means that if $\lambda(q, a)(\mathfrak{r}) = cc$, $\lambda(q, a)(\mathfrak{s}) = d$ and $\lambda(\delta(q, a), b)(\mathfrak{r}) = \mathfrak{r}\mathfrak{s}$, then $\lambda^*(q, ab)(\mathfrak{r}) = ccd$.

For all $\mathfrak{r} \in \mathfrak{R}$ and $u \in A^*$, we define the substitution $[\![\cdot]\!]_u\colon \mathfrak{R} \to B^*$ which provides “the values of the registers after reading u” by $[\![\mathfrak{r}]\!]_u := (\iota \circ \lambda^*(q_0, u))(\mathfrak{r})$. As a substitution, we can extend $[\![\cdot]\!]_u$ to a function $(\mathfrak{R} \uplus B)^* \to B^*$. Now, we define the function $[\![\mathscr{S}]\!]\colon A^* \to B^*$ computed by the DSST. Given

$u \in A^*$, we let $[\![\mathscr{S}]\!](u) := [\![\tau(\delta^*(q_0, u))]\!]_u$. In other words, the output function is used to combine the final values $[\![\mathfrak{r}]\!]_u$ of the registers, obtained after reading the whole word.

Example 4.15 (Reverse)

The reverse function $u \mapsto \widetilde{u}$ from Example 1.22 can be computed by a DSST with a single state and a single register $\mathfrak{r}$. When reading a letter a, the DSST updates $\mathfrak{r} \mapsto a\mathfrak{r}$. Finally it outputs $\mathfrak{r}$.

Example 4.16 (Exponential output)

The function $\mathsf{exp} : a^n \mapsto a^{2^n}$ from Example 4.8 can be computed by a DSST with a single state and a single register $\mathfrak{r}$ initialized to a and updated $\mathfrak{r} \mapsto \mathfrak{r}\mathfrak{r}$ at each transition.

One can provide an alternative definition of DSST where the underlying one-way deterministic automaton $(A, Q, q_0, \delta, \mathsf{Dom}(\tau))$ is not complete (i.e. the functions δ, λ and τ may not be total). Obviously, the function computed by such a machine would have a regular language as domain. Therefore, non-totality does not give additional expressiveness to the model.

4.2.2 Equivalence with recursive marble transducers and consequences

The main goal of Section 4.2 is to link DSST to recursive marble transducers in Theorem 4.17, which from originates [DFG20, Theorem 10]. We also discuss low hanging consequences.

Theorem 4.17 (Recursive marble transducers = DSST)

Recursive marble transducers and DSST compute the same class of functions. Furthermore, both conversions are effective.

Proof. Section 4.2.3 describes the conversion from DSST to recursive marble transducers and Section 4.2.4 describes the reverse transformation. ◀

Theorem 4.17 enables to transfer known results on DSST to recursive marble transducers.

Corollary 4.18 (Equivalence of recursive marble transducers)

Given two recursive marble transducers, one can decide if they compute the same function.

Proof. Equivalence of DSST is known to be decidable by [FR17] (whose proof is nearly entirely based on [CK86]). See [Boj19, Section3] for a more self-contained and generic result. ◀

Theorem 4.17 enables to investigate the expressive power of our machines. The next result originates from [Eng81, Theorem 3.16]. It is also explicit in [DFG20, Section 6] and [NNP21, Theorem 8.1].

Proposition 4.19 (Separation for marble transducers)

The functions blind-square and square cannot be computed by a recursive marble transducer.

Proof. It is easy to observe that the class of functions computed by recursive marble transducers is closed under post-composition by a morphism. We only need to show that blind-square : $u \mapsto$

$(u\#)^{|u|}$ cannot be computed by such a machine. Assume by contradiction that this function is computed by a DSST $\mathscr{S} = (A, A \uplus \{\#\}, Q, q_0, \delta, \mathfrak{R}, \iota, \lambda, \tau)$ and let $a \in A$.

Given $m \geqslant 0$, we let $q_m := \delta^*(q_0, a^m)$. By the pigeonhole principle, there exist $q \in Q$ and an infinite set $I \subseteq \mathbb{N}$ such that $q_m = q$ for all $m \in I$. Now for $m, n \geqslant 0$, let $\mathfrak{R}_{m,n} \subseteq \mathfrak{R}$ be the set of registers which occur in $\lambda_{m,n} := \lambda^*(q_m, a^n)(\tau(q_{m+n})) \in (\mathfrak{R} \uplus A \uplus \{\#\})^*$, that is the registers that will be used in the output after reading a^n, assuming that a^m was already read. For all $m \in I$ and $n \geqslant 0$, we observe that $\lambda^*_{m,n} = \lambda^*(q, a^n)(\tau(\delta(q, a^n)))$ only depends on $n \geqslant 0$, hence so does $\mathfrak{R}_n := \mathfrak{R}_{m,n}$. Furthermore, given a fixed $n \geqslant 0$, we have $\mathfrak{R}_n \neq \varnothing$ since otherwise $|[\![\mathscr{S}]\!](a^{m+n})|$ would be bounded when m variates in I. By the pigeonhole principle there exist $\varnothing \neq \mathfrak{T} \subseteq \mathfrak{R}$ and an infinite set $J \subseteq \mathbb{N}$ such that $\mathfrak{R}_n = \mathfrak{T}$ for all $n \in J$.

Let $n_0 < n_1 \in J$. We claim that for all $\mathfrak{r} \in \mathfrak{T}$, and $m \in I$, $|[\![\mathfrak{r}]\!]_{a^m}| < 2(m+n_0)+2$. Indeed, otherwise $\#a^{m+n_0}\#$ would be a factor of $[\![\mathfrak{r}]\!]_{a^m}$ since the value of $\mathfrak{r}$ is used within the output on a^{m+n_0}, which contradicts the fact that this value is also used within the output on a^{m+n_1}. We conclude that $|[\![\mathscr{S}]\!](a^{m+n_0})| = \mathcal{O}(m)$ when m variates in I, which yields a contradiction. ◀

As a consequence of Propositions 3.15 and 4.19 and Examples 4.4, 4.5 and 4.8, all inclusions between the classes of functions computed by blind pebble transducers, marble transducers and last pebble transducers are strict (already for machines with 2 layers), as depicted in Figure 4.1.

We shall see in Section 5.1 that when the output lies in commutative monoid (in particular, over a unary output alphabet $B = \{b\}$) marble transducers and pebble transducers have the same expressive power. In other words, only the horizontal blue ellipse of Figure 4.1 exists in this case. Over non-unary alphabets, the related class membership problems have not been studied. Open question 4.20 seems to be a first reasonable step, which might be solved by generalizing the proof techniques of Proposition 4.19. This question is meaningful since it asks whether a function is "streamable".

Open question 4.20 (Last pebble transducers → Marble transducers)

Given a function $f\colon A^* \to B^*$ computed by a last pebble transducer, is it decidable whether f can be computed by a marble transducer?

Finally, we provide a non-closure property which originates from [DFG20, Claim 27]. It roughly means that the model of marble transducers is not symmetrical with respect to reversing the output.

Proposition 4.21 (Non-composition of marble transducers)

The function left-product : $a^m\#a^n \mapsto (a^n\#)^m$ cannot be computed by a recursive marble transducer. As a consequence, the classes of functions computed by marble transducers or recursive marble transducers are not closed under pre-composition by regular functions.

Proof idea. For showing that left-product is not computable, we follow exactly the same sketch as for the proof of Proposition 4.19. The consequence comes by observing that left-product is the pre-composition of right-product from Example 4.4 by the mirror function. ◀

4.2.3 From streaming string transducers to recursive marble transducers

The goal of this section is to show one half of Theorem 4.17, by describing how to transform a DSST into an equivalent recursive marble transducer. Consider an DSST $\mathscr{S} = (A, B, Q, q_0, \delta, \mathfrak{R}, \iota, \lambda, \tau)$ computing a function $f\colon A^* \to B^*$. We describe a recursive marble transducer $\mathscr{M}$ which computes f.

4.2.3.1 Submachines of $\mathscr{M}$. For all $\mathfrak{r} \in \mathfrak{R}$, $\mathscr{M}$ has a submachine value-of-$\mathfrak{r}$ which outputs $[\![\mathfrak{r}]\!]_u$ on input $u \in A^*$. If $u = u'a$, it first determines the substitution $\mathfrak{r} \mapsto \alpha$ which is applied when reading letter a (by moving forward from the last position of u while simulating the transitions of $\mathscr{S}$) and then makes recursive calls to compute the values of $[\![\mathfrak{s}]\!]_{u'}$ for $\mathfrak{s}$ occurring in α.

Algorithm 4.22: Submachine which computes the value of $\mathfrak{r} \in \mathfrak{R}$

Submachine value-of-$\mathfrak{r}(u)$
  if $u = \varepsilon$ **then**
    Output $\iota(\mathfrak{r})$ `/* Base case */`
  else if $u = u'a$ **then**
    $q := \delta^*(q_0, u')$ `/* Computed by doing a left to right pass. */`
    $\alpha := \lambda(q, a)(\mathfrak{r})$ `/* Current substitution` $\mathfrak{r} \mapsto \alpha$. `*/`
    for j **in** $\{1, \dots, |\alpha|\}$ **do**
      if $\alpha[j] \in B$ **then**
        Output $\alpha[j]$
      else
        Call submachine value-of-$\alpha[j](u')$
      end
    end
  end

4.2.3.2 Head of $\mathscr{M}$. Finally, the head of $\mathscr{M}$ is an extra specific submachine value-of-τ which uses recursive calls to produce $[\![\tau(\delta^*(q_0, u))]\!]_u$ on input $u \in A^*$. It should be clear that $\mathscr{M}$ computes f.

4.2.4 From recursive marble transducers to streaming string transducers

The goal of this section is to show the other half of Theorem 4.17, by describing how to transform a recursive marble transducer into an equivalent DSST. Let $\mathscr{M}$ be a recursive marble transducer computing a function $f\colon A^* \to B^*$, we describe a DSST $\mathscr{S}$ which computes f.

To simplify the notations, we assume that $\mathscr{M}$ consists of a single submachine $\mathscr{T}$. The main idea is to adapt the classical transformation from two-way automata to one-way automata from [She59], by making $\mathscr{S}$ keeping track of the right-to-right and initial runs of $\mathscr{T}$. Observe that due to the presence of recursive calls, the same portions of runs can be executed multiple times during the computation[2].

4.2.4.1 Information stored by $\mathscr{S}$. Assume that the head $\mathscr{T}$ has shape $(A, B, Q, q_0, F, \delta, \lambda)$ and let $\to$ be its transition relation. We use the notations introduced in Section 2.1.1 for the extended transition function and extended output function of a 2DT. After reading $u \in A^*$, $\mathscr{S}$ will store:

(1) informations about the right-to-right runs of $\mathscr{T}$ labelled by $\vdash u$:
- (a) for all $p \in Q$ such that $\delta^*(\overleftarrow{p}, \vdash u)$ has shape $\overrightarrow{q}$, the state q, stored in the states of $\mathscr{S}$;
- (b) for all $p \in Q$ such that $\delta^*(\overleftarrow{p}, \vdash u)$ has shape $\overrightarrow{q}$, the value $\lambda^{**}(\overleftarrow{p}, \vdash u)$, which is the output produced by $\mathscr{M}$ along maxi-run$(\overleftarrow{p}, \vdash u)$. Roughly, $\lambda^{**}(\overleftarrow{p}, \vdash u)$ concatenates the outputs of the recursive calls[3] along the run maxi-run$(\overleftarrow{p}, \vdash u)$ of $\mathscr{T}$. Thus $\lambda^{**}(\overleftarrow{p}, \vdash u) \in B^*$ whereas $\lambda^*(\overleftarrow{p}, \vdash u) \in (B \uplus \{\mathscr{T}\})^*$ since the latter does not execute the recursive calls (it only writes their names). This value is stored in a register right$_p$ of $\mathscr{S}$;

[2] This behavior is translated into register copies by our construction.
[3] This definition is relevant since the recursive calls are done on a prefix of u, which is itself a prefix of our input. However, for a pebble transducer, the output of a nested call would depend on the whole input, which is not yet known.

(2) informations about the beginning of the initial run labelled by $\vdash u$:
 (a) if $\delta^*(\overrightarrow{q_0}, \vdash u)$ has shape $\overrightarrow{q}$, the state q, stored within the states of $\mathscr{S}$;
 (b) if $\delta^*(\overrightarrow{q_0}, \vdash u)$ has shape $\overrightarrow{q}$, the output $\lambda^{**}(\overrightarrow{q_0}, \vdash u)$ produced along this run by $\mathscr{M}$. This value $\lambda^{**}(\overrightarrow{q_0}, \vdash u)$ is stored in a register out of $\mathscr{S}$.

4.2.4.2 Updating the right-to-right and initial runs. Assume that $\mathscr{S}$ has computed the elements of Items (1) and (2) for some $u \in A^*$. Let $a \in A$ and $p_0 \in Q$, we first explain in Claim 9.17 how maxi-run$(\overleftarrow{p_0}, \vdash ua)$ can be described by recombining the informations about $\vdash u$.

Claim 4.23 (Updating right-to-right runs)

$\delta^*(\overleftarrow{p_0}, \vdash ua) = \overrightarrow{q}$ if and only there exist $0 \leqslant n < |Q|$ and $q_1, p_1, \dots, q_n, p_n \in Q$ such that:

- $\delta(p_n, a) = (\triangleright, q)$ and for all $0 \leqslant i < n$, $\delta(p_i, a) = (\triangleleft, q_{i+1})$;
- for all $1 \leqslant i < n$, $\delta^*(\overleftarrow{q_i}, u) = \overrightarrow{p_i}$.

In this case, we have:

$$\lambda^{**}(\overleftarrow{p_0}, \vdash ua) = [\![\lambda(p_0, a)]\!](u)\ \lambda^{**}(\overleftarrow{q_1}, \vdash u)\ [\![\lambda(p_1, a)]\!](u)\ \cdots\ \lambda^{**}(\overleftarrow{q_n}, \vdash u)\ [\![\lambda(p_n, a)]\!](u)$$

where for all $0 \leqslant i \leqslant n$, $[\![\lambda(p_i, a)]\!] := [\![t_1]\!] \cdots [\![t_m]\!]$ if $[\![\lambda(p_i, a)]\!] = t_1 \cdots t_m$, thus it denotes the concatenation of the submachines called when doing this transition.

Proof idea. If $\delta^*(\overleftarrow{p_0}, \vdash ua) = \overrightarrow{q}$, then maxi-run$(\overleftarrow{p_0}, \vdash ua)$ has the same structure as the run of Figure 4.24. We get $n < |Q|$ since one cannot have $p_i = p_j$ for $0 \leqslant i < j \leqslant n$. ◀

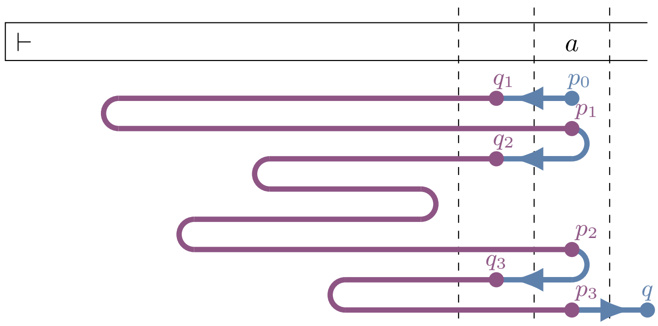


Figure 4.24: Structure of a right-to-right run starting in configuration $(p_0, |\vdash ua|)$.

Now, $\mathscr{S}$ can compute[4] the states $q_1, p_1, \dots, q_n, p_n, q \in Q$ whenever they exist. Let us explain how to update the value contained in right$_q$ by doing a substitution. The values $\lambda^{**}(\overleftarrow{p_i}, \vdash u)$ are stored in registers right$_{p_i}$, thus they can directly be used in the update. Furthermore, for all $0 \leqslant i \leqslant n$, if $[\![\lambda(p_i, a)]\!] = t_1 \cdots t_m$, then for all $1 \leqslant j \leqslant m$, the value t_j is either:

- $b \in B$, and then $b = [\![b]\!](u)$ can directly be written in the substitution;
- or $\mathscr{T}$, but then $[\![\mathscr{T}]\!](u)$ is obtained by considering the accepting run of $\mathscr{T}$ labelled by $\vdash u \dashv$. Since $\mathscr{T}$ is normalized, this run produces no output when reading $\dashv$, thus its output can be decomposed as the concatenation of $\lambda^{**}(\overrightarrow{q_0}, \vdash u)$ and $\lambda^{**}(\overleftarrow{p}, \vdash u)$ for some $p \in Q$. Therefore it can be computed by using the values stored in the registers out and right$_p$ for $p \in Q$.

The updates of Item (2) are done by a similar construction for maxi-run$(\overrightarrow{q_0}, \vdash u)$.

[4]Since this computation is bounded, it is in fact hardcoded in the transitions of $\mathscr{S}$.

4.2.4.3 Output function of $\mathscr{S}$. Once the whole input is read, $\mathscr{S}$ can recombine all pieces of information in order to obtain the output of $\mathscr{M}$. This construction is similar to that of the update.

4.3 Layered streaming string transducers

In this section, we show that for all $k \geqslant 1$, a syntactic restriction on DSST called *k-layeredness* enables to capture exactly the expressive power of k-marble transducers. For $k = 1$, we recover the original result of [AC10] which shows that *copyless* streaming string transducers exactly compute regular functions. The correspondences between the various models are depicted in Figure 4.25.

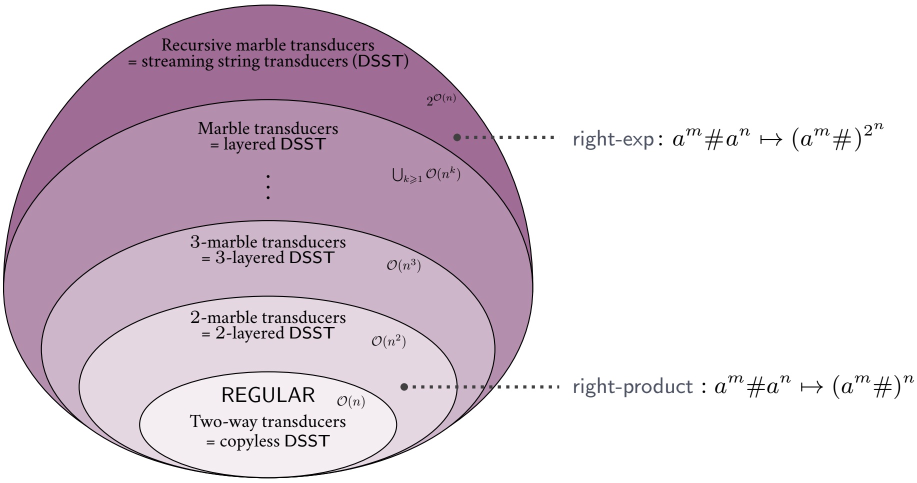


Figure 4.25: Classes of functions computed by marble transducers and recursive marble transducers.

4.3.1 Copy restrictions for substitutions

Intuitively, the way for a DSST to produce outputs of exponential growth is to have substitutions of shape $\mathfrak{r} \mapsto \mathfrak{r}\mathfrak{r}$ (duplication of a register value), and then to repeat this operation along a computation. The notion of *k-layeredness* is a syntactic guarantee for avoiding such behaviors.

4.3.1.1 Copyless and K-bounded DSST. We first recall the notions of *copyless* and *K-bounded* DSST, which originate from [AC10, AFT12]. The main idea is to forbid register copies.

Definition 4.26 (Copyless DSST)

A substitution $s\colon \mathfrak{R} \to (\mathfrak{R} \uplus B)^*$ is *copyless* if every $\mathfrak{s} \in \mathfrak{R}$ occurs at most once in $\{s(\mathfrak{r}) \mid \mathfrak{r} \in \mathfrak{R}\}$. A DSST is said to be *copyless* if all its substitutions are copyless.

Note that if $f\colon A^* \to B^*$ is a function computed by a copyless DSST, then $|f(u)| = \mathcal{O}(|u|)$. Indeed, the values contained in the registers are at most linear in the size of the prefix read so far.

Example 4.27 (Copyless substitutions)

The substitution $s_1 := \mathfrak{r} \mapsto b, y \mapsto b\mathfrak{r}\mathfrak{s}b$ is copyless, while $s_2 := \mathfrak{r} \mapsto \mathfrak{r}b, \mathfrak{s} \mapsto \mathfrak{r}\mathfrak{s}$ is not.

Now, we recall the notion of *K-bounded* DSST as presented in [DJR18, Section 2.1][5]. Intuitively, it means that even if a register value can be duplicated, it can never be used more than K times within another register during the computation. Thus the size of the output is still linear.

Definition 4.28 (Bounded DSST)

Let $K \geqslant 1$. A substitution $s\colon \mathfrak{R} \to (\mathfrak{R} \uplus B)^*$ is *K-bounded* if for all $\mathfrak{r}, \mathfrak{s} \in \mathfrak{R}$, $\mathfrak{s}$ occurs at most K times in $s(\mathfrak{r})$. A DSST $\mathscr{S} := (A, B, Q, q_0, \mathfrak{R}, \iota, \lambda, \tau)$ is said to be *K-bounded* if for all $q \in Q, u \in A^*$, the substitution $\lambda^*(q, u)$ is K-bounded.

The definition of copyless DSST was "local", in the sense that we only gave conditions on the substitutions $\lambda(q, a)$ for $a \in A$. It is equivalent to a "global" definition saying that each $\lambda^*(q, u)$ for $u \in A^*$ is copyless, because the composition of copyless substitutions is also copyless. Since a copyless substitution is 1-bounded, it follows that copyless DSST are 1-bounded. However, the converse does not hold because 1-boundedness allows substitutions of shape $\mathfrak{r} \mapsto \mathfrak{r}, \mathfrak{s} \mapsto \mathfrak{r}$. Furthermore, a "local" definition of K-bounded DSST (i.e. putting conditions only on the $\lambda(q, a)$) would be weaker than our "global" one. Indeed, it is easy to see that the composition of K-bounded substitutions may not be K-bounded.

Example 4.29 (Composition of 1-bounded substitutions)

The substitutions $s_1 := \mathfrak{r} \mapsto \mathfrak{r}, \mathfrak{s} \mapsto \mathfrak{r}$ and $s_2 := \mathfrak{r} \mapsto \varepsilon, \mathfrak{s} \mapsto \mathfrak{r}\mathfrak{s}$ are 1-bounded. However, $s_1 \circ s_2(\mathfrak{s}) = \mathfrak{r}\mathfrak{r}$, hence this substitution is not 1-bounded.

4.3.1.2 k-layered and (k, K)-bounded DSST.

Now, we intend to define a restriction of DSST which forces the output to have polynomial size (but not necessarily linear). We thus introduce the notion of *k-layeredness*, which originates from [DFG20, Definition 13]. The idea is to force the set $\mathfrak{R}$ of registers to be partitioned in k layers $\mathfrak{R}_1, \dots, \mathfrak{R}_k$, so that each layer $\mathfrak{R}_i$ is "copyless in itself", but can use many copies of the registers belonging to layers $\mathfrak{R}_j$ for $j < i$.

Definition 4.30 (k-layered DSST)

Let $k \geqslant 1$. A DSST $(A, B, Q, q_0, \delta, \mathfrak{R}, \iota, \lambda, \tau)$ is said to be *k-layered* if there exists a partition $\mathfrak{R}_1, \dots, \mathfrak{R}_k$ of $\mathfrak{R}$, such that $\forall q \in Q, \forall a \in A$, the following holds:

- $\forall 1 \leqslant i \leqslant k$, only registers from $\mathfrak{R}_1, \dots, \mathfrak{R}_i$ appear in $\lambda(q, a)(\mathfrak{r})$ for $\mathfrak{r} \in \mathfrak{R}_i$;
- $\forall 1 \leqslant i \leqslant k$, each register $\mathfrak{s} \in \mathfrak{R}_i$ appears at most once in $\{\lambda(q, a)(\mathfrak{r}) \mid \mathfrak{r} \in \mathfrak{R}_i\}$.

Observe that 1-layered DSST are exactly copyless DSST. The update mechanism of a 3-layered DSST is depicted in Figure 4.32 below. The reader is invited to check that if $f\colon A^* \to B^*$ is a function computed by a k-layered DSST, then $|f(u)| = \mathcal{O}(|u|^k)$.

Example 4.31 (Right product)

The function right-product$\colon a^m \# a^n \mapsto (a^m \#)^n$ from Example 4.4 is computed by a 2-layered

[5] Several definitions of K-bounded DSST coexist in the literature. A different one can be found e.g. in [AFT12, Definition 5].

DSST with $\mathfrak{R}_0 = \{\mathfrak{r}\}$ and $\mathfrak{R}_1 = \{\mathfrak{s}\}$ as follows. First, when reading $a^m\#$, the machine stores $a^m\#$ in $\mathfrak{r}$, while keeping ε in $\mathfrak{s}$. Then, each time it sees a a, it applies $\mathfrak{r} \mapsto \mathfrak{r}, \mathfrak{s} \mapsto \mathfrak{r}\mathfrak{s}$.

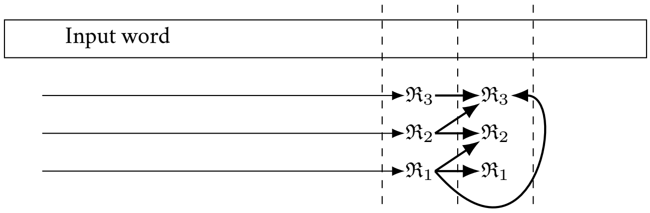


Figure 4.32: Update of the registers in a 3-layered DSST.

For $k \geqslant 1$, we also define a *(k, K)-bounded* DSST in a similar fashion as a k-layered DSST, except that each layer is no longer "copyless in itself" but "K-bounded in itself".

Definition 4.33 (Bounded DSST with layers)

Let $k, K \geqslant 1$. We say that a DSST $\mathscr{S} := (A, B, Q, q_0, \delta, \mathfrak{R}, \iota, \lambda, \tau)$ is *(k, K)-bounded* if there exists a partition $\mathfrak{R}_1, \dots, \mathfrak{R}_k$ of $\mathfrak{R}$ such that for all $q \in Q, u \in A^*$:

- for all $0 \leqslant i \leqslant k$, only registers from $\mathfrak{R}_1, \dots, \mathfrak{R}_i$ appear in $\{\lambda(q, u)(\mathfrak{r}) \mid \mathfrak{r} \in \mathfrak{R}_i\}$;
- for all $1 \leqslant i \leqslant k$ and $\mathfrak{r}, \mathfrak{s} \in \mathfrak{R}_i$, $\mathfrak{s}$ occurs at most K times in $\lambda(q, u)(\mathfrak{r})$.

In particular, a $(1, K)$-bounded DSST is a exactly a K-bounded DSST. As mentioned above for $k = 1$, k-layered DSST are syntactically more restrictive than $(k, 1)$-bounded DSST. The main interest of bounded DSST is their use as an intermediate model in the proofs.

4.3.2 Equivalence with marble transducers

Now we are ready to claim that layeredness is a restriction of DSST which exactly captures the power of marble transducers. The next result originates from [DFG20, Theorem 15].

Theorem 4.34 (Marble transducers = layered/bounded DSST)

Given $f : A^* \rightharpoonup B^*$ and $k \geqslant 0$, the following conditions are equivalent:

(1) f is computed by a k-marble transducer;
(2) f is computed by a (k, K)-bounded DSST for some $K \geqslant 0$;
(3) f is computed by a k-layered DSST.

The conversions are effective.

Proof. Item (3) ⇒ Item (1) is shown in Section 4.3.3. Item (2) ⇒ Item (3) is shown in Section 4.3.4. For Item (1) ⇒ Item (2), we first transform the k-marble transducer in a (not necessarily k-layered) DSST using Theorem 4.17. Since the function f computed by a k-marble transducer is such that $|f(u)| = \mathcal{O}(|u|^k)$, we use Theorem 4.41 to build a (k, K)-bounded DSST for f. ◀

The case $k = 1$ in Theorem 4.34 provides the celebrated result of [AC10] (shown using different techniques) which relates regular functions and copyless DSST. From a practical point of view, this statement provides a streaming machine model for implementing regular functions. Indeed, if a DSST is copyless, one can implement its transitions in constant time (assuming that its registers are represented by doubly linked lists), since we never duplicate the content of a register.

Corollary 4.35 (Two-way transducers = copyless/bounded DSST)

Given $f : A^* \rightharpoonup B^*$, the following conditions are equivalent:

(1) f is computed by a 2DT (i.e. f is regular);
(2) f is computed by a K-bounded DSST for some $K \geqslant 0$;
(3) f is computed by a copyless DSST.

The conversions are effective.

4.3.3 From layered streaming string transducers to marble transducers

The goal of this section is to show Item (3) $\Rightarrow$ Item (1) in Theorem 4.34. Given k-layered DSST, we describe how to build an equivalent k-marble transducer. The main idea is to mimic the proof of Section 4.2.3, while making nested calls only when they are absolutely necessary, that is when going from one layer to another. Let $\mathscr{S} = (A, B, Q, q_0, \delta, \mathfrak{R}, \iota, \lambda, \tau)$ be a k-layered DSST.

4.3.3.1 Case $k = 1$. We first suppose that $k = 1$, that is $\mathscr{S}$ is copyless. In this case, our procedure is similar to that of [DJR18, DFJL17]. We simulate the recursion of Algorithm 4.22, but without using submachines. Lemma 4.36 crucially relies on a clever use of copylessness.

Lemma 4.36 (Simulation of Algorithm 4.22 by a two-way transducer)

One can build a 2DT with lookarounds $\mathscr{T}$ with designated states $p_{\mathfrak{r}}$ and $r_{\mathfrak{r}}$ for $\mathfrak{r} \in \mathfrak{R}$, such that the following holds. For all input $u \in A^*$, $1 \leqslant i \leqslant |u|$ and $\mathfrak{r} \in \mathfrak{R}$, the longest run of $\mathscr{T}$ labelled by $\vdash u \dashv$ which starts in configuration $(p_{\mathfrak{r}}, |\vdash u[1{:}i]|)$ and moves on $\vdash u[1{:}i]$ has the following property: it outputs $[\![\mathfrak{r}]\!]_{u[1:i]}$ and it ends in configuration $(r_{\mathfrak{r}}, |\vdash u[1{:}i]|{+}1)$.

Proof. The idea is to simulate the whole computation of Algorithm 4.22 without making recursive calls. When in position $|\vdash u[1{:}i]|$ of input $\vdash u \dashv$, $\mathscr{T}$ first uses its lookaround to determine the state $q := \delta^*(q_0, u[1{:}i{-}1])$ of $\mathscr{S}$, and the substitution $\alpha := \lambda(q, u[i])(\mathfrak{r})$. Then it simulates the "for" loop of Line 7 and outputs $\alpha[j]$ if it belongs to B. Now if $\mathfrak{s} := \alpha[j]$ is a register, then $\mathscr{T}$ moves left to position $i{-}1$ and goes to state $p_{\mathfrak{s}}$. By induction, we assume that $\mathscr{T}$ can repeat this process to output $[\![\mathfrak{s}]\!]_{u[1:|u|-1]}$, go back to position $|\vdash u[1{:}i]|$ in state $r_{\mathfrak{s}}$. Since $\mathscr{S}$ was copyless, $\mathfrak{s} \in \mathfrak{R}$ occurs a most once in the whole set $\{\lambda(q, u[i])(\mathfrak{t}) \mid \mathfrak{t} \in \mathfrak{R}\}$ (which can be determined using a lookaround). Thus $\mathscr{T}$ can recover the fact that it was computing index $\alpha[j]$ in $\mathfrak{r} \mapsto \alpha$ and pursue the loop. When the loop is ended, it moves right in position $|\vdash u[1{:}i]|{+}1$ and goes to state $r_{\mathfrak{r}}$. ◀

The final 2DT which simulates $\mathscr{S}$ is built by using $\mathscr{T}$ from Lemma 4.36 to compute the values of the registers which occur in the final output function.

Remark 4.37 (Two-way copyless SST)

Going further, one can adapt the proof of Section 4.3.3.1 to show that a *two-way copyless streaming string transducer* (defined as a DSST, but allowing two-way moves) can be transformed in an equivalent 2DT. As a consequence, this model as expressive as copyless (one-way) DSST.

4.3.3.2 Case $k > 1$. Let $\mathfrak{R}_1, \dots, \mathfrak{R}_k$ be the partition of the registers of $\mathscr{S}$. Roughly, we want the k-marble transducer to have a submachine value-of-$\mathfrak{r}$ for all $\mathfrak{r} \in \mathfrak{R}$, $1 \leqslant i \leqslant k$ and $\mathfrak{r} \in \mathfrak{R}_i$, but this submachine will only call value-of-$\mathfrak{s}$ for $\mathfrak{s} \in \mathfrak{R}_j$ with $j < i$ (hence there is no recursion).

Formally, the submachine value-of-$\mathfrak{r}$ behaves as described in Algorithm 4.22. When executing the "for" loop of Line 7, it outputs $\alpha[j]$ if it belongs to B. Now if $\mathfrak{s} := \alpha[j]$ is a register, there are two cases:

- either $\mathfrak{s} \in \mathfrak{R}_j$ for some $j < i$. In this case, value-of-$\mathfrak{r}$ makes a nested call to value-of-$\mathfrak{s}$.
- or $\mathfrak{s} \in \mathfrak{R}_i$. In this case, value-of-$\mathfrak{r}$ simulates the computation of value-of-$\mathfrak{s}$ by moving backward without making a nested call, as explained in Lemma 4.36.

The head of the final k-marble transducer is built by using the various value-of-$\mathfrak{r}$ to compute the values of the registers which occur in the final output function.

4.3.4 From bounded to layered streaming string transducers

The goal of this section is to show Item (2) $\Rightarrow$ Item (3) in Theorem 4.34. Given (k, K)-bounded DSST for some $K \geqslant 0$, we describe how to build an equivalent k-layered DSST.

It is known (see e.g. [AFT12]) that a K-bounded DSST can be transformed in an equivalent copyless DSST. We first claim a stronger[6] result in Lemma 4.40: the copyless DSST preserves the input positions in which the output letters were created. Formally, *origin semantics* of a DSST is obtained by labelling the letters stored within the registers by input positions, as we did for 2DT in Definition 1.26.

Definition 4.38 (Origin semantics for DSST)

Let $\mathscr{S} = (A, B, Q, q_0, \delta, \mathfrak{R}, \iota, \lambda, \tau)$ be a DSST. For all $\mathfrak{r} \in \mathfrak{R}$ and $u \in A^*$, we let the value of $\mathfrak{r}$ in *origin semantics* after reading u be the word of $(B \times \mathbb{N})^*$ defined by:

- if $u = \varepsilon$ and $\alpha := \iota(\mathfrak{r})$, then the value is $(\alpha[1], 0) \cdots (\alpha[|\alpha|], 0)$;
- if $u = u'a$, for all $\mathfrak{r} \in \mathfrak{R}$ let $\alpha_{\mathfrak{r}}$ be the value of $\mathfrak{r}$ in origin semantics after reading u'. Let $s \colon \mathfrak{R} \uplus B \rightarrow (B \uplus \mathbb{N})^*$ be the function which maps $\mathfrak{r} \in \mathfrak{R}$ to $\alpha_{\mathfrak{r}}$ and $b \in B$ to $(b, |u|)$. The value of $\mathfrak{r}$ in *origin semantics* after reading u is $s \circ \lambda(\delta^*(q_0, u'), a)$.

The function $f \colon A^* \rightarrow (B \times \mathbb{N})^*$ computed by $\mathscr{S}$ in *origin semantics* is defined accordingly, where the letters occurring in $\tau(\delta^*(q, u))$ are labelled by $|u|+1$.

As for 2DT, the first component of f is simply the function computed by $\mathscr{S}$.

Example 4.39 (Reverse)

The 2DT from Example 4.15 computes $a_1 \cdots a_n \mapsto (a_n, n) \cdots (a_1, 1)$ in origin semantics.

Now, we claim that origin semantics can be preserved when transforming a K-bounded DSST into a copyless one. We postpone the proof of Lemma 4.40 to Part III of the current manuscript.

Lemma 4.40 (From bounded to copyless)

Given a K-bounded DSST, one can build a copyless DSST which computes the same function in origin semantics. In particular, it computes the same function when forgetting origins.

Proof sketch. The proof goes over the proof of Section 9.3 which shows a similar result over infinite words, but without dealing explicitly with origin semantics. We nevertheless argue in Section 9.3.4.2 that origin semantics is in fact preserved by this transformation. ◀

[6] Even if the construction presented in [AFT12] is likely to preserve origin semantics, this result is not explicit. Therefore, we are not aware of a way to directly use the result of [AFT12] as a blackbox for showing Item (2) $\Rightarrow$ Item (3) in Theorem 4.34.

Now, let us show by induction on $k \geqslant 1$ that given a (k, K)-bounded DSST, one can build an equivalent k-layered DSST, by using Lemma 4.40 for the induction step.

Consider a (k, K)-bounded DSST $\mathscr{S} = (A, B, Q, q_0, \mathfrak{R}, \iota, \lambda, \tau)$ whose partition of registers is $\mathfrak{R}_1, \ldots, \mathfrak{R}_k$. We let $\mathfrak{R}' := \bigcup_{1 \leqslant i < k} \mathfrak{R}_i$. Without loss of generalities, we assume that τ has type $Q \to (\mathfrak{R}_k \uplus B)^*$ (it does not use $\mathfrak{R}'$). One can build several DSST from $\mathscr{S}$:

- a K-bounded DSST $\mathscr{S}_k$ with registers $\mathfrak{R}_k$ and output alphabet $\mathfrak{R}' \uplus B$. It consists in the machine induced by $\mathscr{S}$ when updating only registers in $\mathfrak{R}_k$ and considering $\mathfrak{R}'$ as letters;
- for all $\mathfrak{r} \in \mathfrak{R}'$, a $(k-1, K)$-bounded DSST $\mathscr{S}_\mathfrak{r}$ which is the machine induced by $\mathscr{S}$ on $\mathfrak{R}'$, whose output function is always $\mathfrak{r}$. Thus it computes the value of $\mathfrak{r}$ in $\mathscr{S}$.

By induction hypothesis, for all $\mathfrak{r} \in \mathfrak{R}'$, one can build from $\mathscr{S}_\mathfrak{r}$ a $(k-1)$-layered DSST denoted $\mathscr{U}_\mathfrak{r}$, which computes the value of $\mathfrak{r}$ in $\mathscr{S}$. Furthermore, by Lemma 4.40 one can build a copyless DSST $\mathscr{U}_k$ which computes the same function as $\mathscr{S}_k$ in origin semantics.

Now, consider the DSST obtained by merging $\mathscr{U}_k$ and $\mathscr{U}_\mathfrak{r}$ for $\mathfrak{r} \in \mathfrak{R}'$ (formally, we do the product of their states and transition functions), using the output function of $\mathscr{U}_k$, and replacing each mention of $\mathfrak{r} \in \mathfrak{R}'$ in the updates of $\mathscr{U}_k$ by the according current output of $\mathscr{U}_\mathfrak{r}$. This machine $\mathscr{U}$ is k-layered. Furthermore, $\mathscr{U}$ computes the same function as $\mathscr{S}$: the key argument is that $\mathscr{U}_k$ uses each $\mathfrak{r} \in \mathfrak{R}_k$ exacly in the same positions as $\mathscr{S}_k$ did, because origin semantics is preserved.

4.4 Solving the optimization problem for streaming transducers

The goal of this section is to show Theorem 4.41, which is the optimization statement of Chapter 4. This result originates from [DFG20, Section 5]. It was already known for $k = 1$ in [FR17, Theorem 5.2].

Theorem 4.41 (Optimization of streaming string transducers)

Let $f\colon A^* \to B^*$ be computed by an DSST. For all $k \geqslant 1$, f can computed by a k-layered DSST if and only if $|f(u)| = \mathcal{O}(|u|^k)$. One can decide if such a $k \geqslant 1$ exists, compute the minimal one, and in this case build a (k, K)-bounded (or k-layered) DSST which computes f for some $K \geqslant 1$.

Proof sketch. We first transform in Section 4.4.1 a DSST computing f into a *simple* DSST, i.e. which has no states and uses no letters apart in its initial function. If this machine has a (decidable) property called *barbell*[7], we show that there exist $v_0, u, v_1 \in A^*$ such that $|f(v_0 u^X v_1)| = 2^{\Omega(X)}$, thus f cannot be computed by a k-layered DSST. Otherwise, we find in Section 4.4.2 the minimal $k \geqslant 0$ such that $|f(u)| = \mathcal{O}(|u|^k)$ and build a (k, K)-bounded DSST which computes f. ◀

The rest of this section is devoted to a detailed proof of Theorem 4.41. As a side result, we obtain in Claim 4.46 that if $f\colon A^* \to B^*$ is computed by a DSST, then the function $|f|\colon A^* \to \mathbb{N}, u \mapsto |f(u)|$ (which forgets everything but the length) is a *rational series* over the semiring $(\mathbb{N}, +, \times)$. This is one of the motivations for the detailed study of polyregular functions with output in $\mathbb{Z}$ or $\mathbb{N}$ in Part II.

4.4.1 From streaming string transducers to $\mathbb{N}$-weighted automata

The goal of this section is to build a *$\mathbb{N}$-weighted automaton* computing the size of the registers in a given DSST. This statement is formalized in Claim 4.46 at the end of the section.

[7] The term *dummbell* is also used in the literature.

4.4.1.1 Simple DSST. The first step is to make our machine as restricted as possible. We say that a DSST $\mathscr{S} = (A, B, Q, q_0, \delta, \mathfrak{R}, \iota, \lambda, \tau)$ is *simple* if it has a single state (i.e. $Q = \{q_0\}$), and its substitutions and output do not use letters (i.e. $\lambda\colon Q \times A \to \mathcal{S}_{\mathfrak{R}}^{\varnothing}$ and $F : Q \to \mathfrak{R}^*$). To simplify the notations, we write $(A, B, \mathfrak{R}, \iota, \lambda, F)$ for a simple DSST, where $\lambda\colon A \to \mathcal{S}_{\mathfrak{R}}^{\varnothing}$ and $\tau \in \mathfrak{R}^*$. Indeed, the states and transition function are useless. It is easy to show[8] that a DSST can always be simplified[9].

Claim 4.42 (Simplification of DSST)

Given a DSST, one can build an equivalent simple DSST.

Proof idea. Let $\mathscr{T} = (A, B, Q, q_0, \delta, \mathfrak{R}, \iota, \lambda, \tau)$ be a DSST. We can assume that it uses no letters in the substitutions by storing them in constant registers (using the initial function) indexed by B. To remove the states, we let $\mathfrak{R}' := Q \times \mathfrak{R}$ be our new register set. In our new machine, the register $(q, \mathfrak{r})$ will contain the value of $\mathfrak{r}$ if q is the current state of $\mathscr{T}$, and ε otherwise. ◀

4.4.1.2 Weighted automata. Given a simple DSST, we build a $(\mathbb{N}, +, \times)$-*weighted automaton* which computes the size of the words stored in the registers along a run of the DSST. This correspondance will enable to transfer the results on asymptotic growth of $(\mathbb{N}, +, \times)$-weighted automata to DSST.

Formally, a *semiring* $(\mathbb{S}, +, \times)$ consists of a commutative monoid $(\mathbb{S}, +)$ and a monoid $(\mathbb{S}, \times)$ such that $\times$ distributes over $+$ and the neutral for $+$ is absorbing for $\times$. In this manuscript, semirings will either be $(\mathbb{N}, +, \times)$, $(\mathbb{Z}, +, \times)$ or $(\mathbb{Q}, +, \times)$, thus no deeper understanding of the theory of semirings is required for the reader. Given finite sets S, T, we denote by $\mathsf{M}_{S,T}(\mathbb{S})$ the set of matrices with coefficients in $\mathbb{S}$ and whose lines (resp. columns) are indexed by S (resp. T). Observe that $\mathsf{M}_{S,S}(\mathbb{S})$ equipped with matrix product is a monoid. If $m, n \geqslant 0$, we write $\mathsf{M}_{m,n}$ for $\mathsf{M}_{[1:m],[1:n]}$.

Given a semiring $(\mathbb{S}, +, \times)$, a $(\mathbb{S}, +, \times)$-*weighted automaton* (also known as $(\mathbb{S}, +, \times)$-*automaton* or $(\mathbb{S}, +, \times)$-*linear representation*) is a machine model which computes a function with output in $\mathbb{S}$. Starting from the seminal results of Schützenberger in [Sch61a], weighted automata over various semirings have been deeply investigated in the literature. They are are often considered as a quantitative counterpart of finite automata. The reader is invited to consult e.g. the monograph [BR11] for a survey.

Definition 4.43 ($\mathbb{S}$-weighted automaton)

An $\mathbb{S}$-*weighted automaton* $\mathscr{W} := (A, Q, I, F, \mu)$ consists of:

- an input alphabet A;
- a finite set Q of states;
- an initial (resp. final) vector $I \in \mathsf{M}_{1,Q}(\mathbb{S})$ (resp. $F \in \mathsf{M}_{Q,1}(\mathbb{S})$);
- a monoid morphism $\mu\colon A^* \to \mathsf{M}_{Q,Q}(\mathbb{S})$.

The function $g\colon A^* \to \mathbb{S}$ computed by $\mathscr{W}$ is defined by $g(u) := I\mu(u)F$ for $u \in A^*$. The class of functions computed by $\mathbb{S}$-weighted automata, is called $\mathbb{S}$-*rational series*.

Example 4.44 (Exponential)

The function $u \mapsto 2^{|u|}$ is computed by a $(\mathbb{N}, +, \times)$-weighted automaton with initial vector $\begin{pmatrix}1\end{pmatrix}$, final vector $\begin{pmatrix}1\end{pmatrix}$ and morphism $u \mapsto \begin{pmatrix}2\end{pmatrix}^{|u|}$.

[8] The reader is invited to observe that the construction does not preserve k-layeredness for some $k \geqslant 1$. In particular, being copyless generally requires to uses states and letters in the substitutions.

[9] See [BDSW17] for a similar construction in the context of *polynomial automata*.

Example 4.45 (Length)

The function $u \mapsto |u|$ is computed by a $\mathbb{N}$-weighted automaton with initial vector $\begin{pmatrix} 1 & 0 \end{pmatrix}$, final vector $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and morphism $u \mapsto \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}^{|u|}$.

4.4.1.3 Flow automaton. Consider a simple DSST $\mathscr{S} := (A, B, \mathfrak{R}, \iota, \lambda, \tau)$ computing a function $f: A^* \to B^*$. We let its *flow automaton* $\mathscr{W}$ be the $\mathbb{N}$-weighted automaton $(A, \mathfrak{R}, I, F, \mu)$ defined by:

- for all $\mathfrak{r} \in \mathfrak{R}$, $I[\mathfrak{r}] := |\iota[\mathfrak{r}]|$ (number of letters initialized in $\mathfrak{r}$);
- for all $\mathfrak{r} \in \mathfrak{R}$, $F[\mathfrak{r}] := |\tau|_{\mathfrak{r}}$ (number of occurrences of $\mathfrak{r}$ in τ);
- for all $a \in A$, $\mathfrak{r}, \mathfrak{s} \in \mathfrak{R}$, $\mu(a)[\mathfrak{r}, \mathfrak{s}] := |\lambda(a)(\mathfrak{s})|_{\mathfrak{r}}$ (number of occurrences of $\mathfrak{r}$).

It is easy to see that $\mathscr{W}$ computes the length of the registers values in $\mathscr{S}$.

Claim 4.46 (Flow automaton is correct)

For all $u \in A^*$ and $\mathfrak{r} \in \mathfrak{R}$, we have $(I\mu(u))[\mathfrak{r}] = |[\![\mathfrak{r}]\!]_u|$. In particular, $\mathscr{W}$ is a weighted automaton which computes $|f|: A^* \to \mathbb{N}, u \mapsto |f(u)|$.

Without loss of generalities, we can assume that the flow automaton is *trim* (i.e. for all $q \in Q$, there exist words $u, v \in A^*$ such that $(I\mu(u))[q] \neq 0$ and $(\mu(v)F)[q] \neq 0$). Indeed, if $\mathfrak{r} \in \mathfrak{R}$ is such that $(I\mu(u))[\mathfrak{r}] = 0$ for all $u \in A^*$, then $\mathfrak{r}$ always had value ε and thus could be removed everywhere in $\mathscr{S}$. Similarly, if $(\mu(v)F)[\mathfrak{r}] = 0$ for all $v \in A^*$, then $\mathfrak{r}$ was never used in the output.

4.4.2 Asymptotic growth of $\mathbb{N}$-weighted automata

Thanks to the flow automaton, we can focus on understanding the asymptotic growth of the functions computed by $\mathbb{N}$-automata. The constructions below are very similar to those used in [WS91] for computing the degree of ambiguity of non-deterministic finite automata.

The first step towards deciding polynomial growth is to understand which syntactic properties make a function unbounded. The next lemma originates from [MS77] and [WS91, Theorem 6.1].

Lemma 4.47 (Patterns for unboundedness)

Let $\mathscr{W} := (A, Q, I, \mu, F)$ be a trim $\mathbb{N}$-automaton computing a function $g: A^* \to \mathbb{N}$. Then $g(u) = \mathcal{O}(1)$ if and only if $\mathscr{W}$ does not have the following patterns:

- a *heavy cycle* on a state q: a word $u \in A^*$ such that $\mu(u)[q, q] \geqslant 2$;
- a *barbell* from p to $q \neq p$: a word $u \in A^*$ such that $\mu(u)[p, p] \geqslant 1$, $\mu(u)[p, q] \geqslant 1$ and $\mu(u)[q, q] \geqslant 1$.

The shapes of heavy cycles and barbells are depicted in Figure 4.48 (when seeing automata as graphs). It is well-known that the presence of a heavy cycle or a barbell between states can be decided. For the barbell, given $p \neq q \in Q$, the set of $u \in A^*$ such that $\mu(u)[p, p] \geqslant 1$ is a regular language. The same holds for $\mu(u)[p, q] \geqslant 1$ and $\mu(u)[p, q] \geqslant 1$. We finally check the emptiness of their intersection. For the heavy cycle, given $q \in Q$, consider the (computable) set C of states p such that $\mu(v)[p, q] \geqslant 1$ and $\mu(v')[q, p] \geqslant 1$ for some $v, v' \in A^*$. If there exist $a \in A$ and $q_0, q_1 \in C$ such that $\mu(a)[q_0, q_1] \geqslant 2$, there is a heavy cycle on q. Otherwise, $\mu(a)[q_0, q_1] \in \{0, 1\}$ for all $q_0, q_1 \in C$ and there is a heavy cycle on q if and only if the restriction of $\mathscr{W}$ to C (seen as a non-weighted automaton) is ambiguous.

(a) Heavy cycle on q

(b) Barbell from p to $q \neq p$

Figure 4.48: Patterns that create unboundedness in a trim $\mathbb{N}$-automaton.

It is easy to observe that heavy cycles directly lead to exponential behaviors.

Claim 4.49 (Heavy cycle $\Rightarrow$ lower exponential bound)

Let $\mathscr{W} := (A, Q, I, \mu, F)$ be a trim $\mathbb{N}$-automaton computing a function $g\colon A^* \to \mathbb{N}$. If $\mathscr{W}$ has a heavy cycle, there exist $v_0, u, v_1 \in A^*$ such that $g(v_0 u^X v_1) = 2^{\Omega(X)}$.

Proof. If the heavy cycle is $u \in A^+$ such that $\mu(u)[q, q] \geqslant 2$, observe that $\mu(u^X)[q, q] \geqslant 2^X$. ◀

Now, we show that the absence of heavy cycles implies that the function computed has polynomial growth. Furthermore, the states of the $\mathbb{N}$-automaton can be partitioned in a somehow layered way.

Lemma 4.50 (No heavy cycles $\Rightarrow$ polynomial growth)

Let $\mathscr{W} := (A, Q, I, \mu, F)$ be a trim $\mathbb{N}$-automaton computing a function $g\colon A^* \to \mathbb{N}$. If $\mathscr{W}$ has no heavy cycles, there exists a (unique) $k \geqslant 0$ such that the following holds:

- ▶ $g(u) = \mathcal{O}(|u|^k)$;
- ▶ $g(v_0 u_1^X v_1 \cdots v_{k-1} u_k^X v_k) = \theta(X^k)$ for some $v_0, u_1, v_1 \dots, u_k, v_k \in A^*$.

This value $k \geqslant 0$ is computable and one can build a partition $Q = \biguplus_{0 \leqslant i \leqslant k} Q_i$ such that:

(1) for all $p, q \in Q$, if there exist $q \in Q$ and $u \in A^*$ with $\mu(u)[p, q] \geqslant 1$, then $p \in S_i, q \in S_j$ with some $i \leqslant j$;
(2) for all $1 \leqslant i \leqslant k$, $p, q \in S_i$ and $u \in A^*$, $\mu(u)[p, q] = \mathcal{O}(1)$;
(3) for all $1 \leqslant i \leqslant k$ and $q \in S_i$, $(I\mu(u))[q] = \mathcal{O}(|u|^i)$.

Proof. The main idea is to regroup the states of $\mathscr{W}$ between which there are no barbells, since they should describe "bounded" portions of the automaton.

We first build an directed graph G which describes the barbells that we can meet. Its vertices are the states Q, and there is an edge from q_0 to q_1 if and only if there exist $p_0 \neq p_1 \in Q$ such that:

- ▶ $\exists v_0, v_1 \in A^*$ such that $\mu(v_0)[q_0, p_0] \geqslant 1$ and $\mu(v_1)[p_1, q_1] \geqslant 1$;
- ▶ there is a barbell from p_0 to p_1.

We claim that the absence of heavy cycles in $\mathscr{W}$ forces the graph G to have no cycles.

Claim 4.51 (Barbell graph is acyclic)

The graph G is a *directed acyclic graph* (i.e. it has no cycles).

Proof. Assume that there exists a cycle $(q_0, q_1)(q_1, q_2) \dots (q_{n-1}, q_0)$ in G. By transitivity of the construction, there exists $w \in A^*$ such that $\mu(w)[q_0, q_{n-1}] \geqslant 1$. Furthermore there exist $p \neq p' \in Q$ and $v, v' \in A^*$ such that $\mu(v)[q_{n-1}, p] \geqslant 1$ and $\mu(v')[p', q_0] \geqslant 1$, and there is a barbell from p to p'. Thus there is $u \in A^*$ such that $\mu(u)[p, p] \geqslant 1$, $\mu(u)[p, p'] \geqslant 1$ and $\mu(u)[p', p'] \geqslant 1$, hence $\mu(uu)[p, p'] \geqslant 2$. Putting everything together, we obtain a heavy cycle since $\mu(wvuuv')[q_0, q_0] \geqslant 2$, which yields a contradiction. ◀

Since G is acyclic, one can define its *minimal* states which have no incoming edges. Furthermore, given a state $q \in Q$, we define its *height* as the maximal length of a path going from a minimal state to q (minimal states having height 0). We denote by k the maximal height over all states, we first show that it provides a lower bound on the degree of a polynomial bounding g.

Claim 4.52 (Lower polynomial bound)

There exist $v_0, u_1, v_1 \dots, u_k, v_k \in A^*$ such that $g(v_0 u_1^X v_1 \cdots v_{k-1} u_k^X v_k) = \Omega(X^k)$.

Proof. By definition of k, one can find states $q_1 \neq q_1', \dots, q_k \neq q_k' \in Q$ such that:

- for all $1 \leqslant i \leqslant k$, there exists $u_i \in A^*$ such that $\mu(u_i)[q_i, q_i] \geqslant 1$ and $\mu(u_i)[q_i, q_i'] \geqslant 1$ and $\mu(u_i)[q_i', q_i'] \geqslant 1$;
- for all $1 \leqslant i < k$, there exists $v_i \in A^*$ such that $\mu(v_i)[q_i', q_{i+1}] \geqslant 1$.

Now, observe that $\mu(u_1^X v_1 \cdots v_{k-1} u_k^X)[q_1, q_k'] \geqslant X^k$. Since the $\mathbb{N}$-automaton is trim, one can find $v_0, v_k \in A^*$ such that $g(v_0 u_1^X v_1 \cdots v_{k-1} u_k^X v_k) \geqslant X^k$. ◀

Finally, we consider the partition $Q_0, \dots, Q_k$ of Q, where Q_i is the set of states of height i. It remains to show that $|g(u)| = \mathcal{O}(|u|^k)$ and check the last properties of Lemma 4.50.

Claim 4.53 (Layered registers)

The following statements hold:

(1) for all $0 \leqslant i \leqslant k$ and $p \in Q_i$, if there exist $q \in Q$ and $u \in A^*$ such that $\mu(u)[p, q] \geqslant 1$, then $p \in Q_j$ for some $i \leqslant j$.
(2) for all $0 \leqslant i \leqslant k$ and $p, q \in Q_i$, $\mu(u)[p, q] = \mathcal{O}(1)$;
(3) for all $0 \leqslant i \leqslant k$, for all $\forall p, q \in \bigcup_{j \leqslant i} Q_j$, $\mu(u)(p, q) = \mathcal{O}(|u|^i)$.

Proof.

(1) Assume that $\mu(u)[p, q] \geqslant 1$, then every path in G from a minimal state s to p of length $0 \leqslant i \leqslant k$ can be completed in a path from s to q (of length at least i).
(2) Due to Item (1), the non-empty terms in the sum that defines $\mu(u)[p, q]$ are indexed by states of Q_i. Thus it is sufficient to show that the sub-automaton "induced" on Q_i is bounded, i.e. that is has no barbells by Lemma 4.47. This is indeed not possible, since otherwise the states of Q_i would not have the same height in G.
(3) The result is shown by induction on $0 \leqslant i \leqslant k$. We deal with $i = k$. Let $S = \bigcup_{j<k} Q_j$, $T \coloneqq Q_j$ and $\mu_S \colon A^* \to \mathbb{N}^{S \times S}, u \mapsto (\mu(u)[p, q])_{p \in S, q \in S}$ be the co-restriction of μ to $S \times S$. We define similarly $\mu_T \colon A^* \to \mathbb{N}^{T \times T}$ and $\mu_{S,T} \colon A^* \to \mathbb{N}^{S \times T}$.
It follows from Item (1) that for all $u \in A^*$, $\mu(u)$ has an upper triangular form:

$$\mu(u) = \begin{pmatrix} \mu_S(u) & \mu_{S,T}(u) \\ 0 & \mu_T(u) \end{pmatrix}$$

and by matrix product we get:

$$\mu_{S,T}(u) = \sum_{1 \leqslant i \leqslant |u|} \mu_S(u[1{:}(i-1)]) \mu_{S,T}(u[i]) \mu_T(u[(i+1){:}|u|]).$$

Using induction hypothesis, we show that each value of this matrix is in $\mathcal{O}(|u|^k)$. ◀

Finally, the partition $Q_0, \dots, Q_k$ can be computed since the presence of a barbell between two states can be decided (as mentioned after Lemma 4.47). ◀

As a consequence, the converse of Claim 4.49 holds and one can decide if a function g computed by a $\mathbb{N}$-weighted automaton is such that $g(u) = \mathcal{O}(|u|^k)$ for some $k \geqslant 0$.

4.4.3 Asymptotic growth in a DSST

Through the flow automaton, this above procedure can be transferred from $\mathbb{N}$-weighted automata to simple DSST. It remains to show that when this property holds, one can build an equivalent (k, K)-bounded DSST. However, if $\mathscr{S} := (A, B, \mathfrak{R}, \iota, \lambda, \tau)$ is a DSST computing a function $f\colon A^* \to B^*$ such that $|f(u)| = \mathcal{O}(|u|^k)$, then Lemma 4.50 partitions its registers in $k{+}1$ layers. In fact, the registers from the first layer only contain bounded values, hence they can be removed up to adding states.

Claim 4.54 (Building a bounded DSST)

Given a simple DSST whose flow automaton verifies the condition of Lemma 4.50 with $k \geqslant 1$, one can build an equivalent (k, K)-bounded DSST for some $K \geqslant 0$.

Proof. Let $\mathscr{S} := (A, B, \mathfrak{R}, \iota, \lambda, \tau)$ be the simple DSST and $\mathfrak{R}_0, \dots, \mathfrak{R}_k$ be the partition of $\mathfrak{R}$ given by Lemma 4.50. By Item (2), there exists $L \geqslant 0$ such that $|[\![\mathfrak{r}]\!]_u| \leqslant L$ for all $x \in \mathfrak{R}_0$ and $u \in A^*$. Thus one can remove the registers $\mathfrak{R}_0$ and hardcode the content of each $\mathfrak{r} \in \mathfrak{R}_0$ in the states. The transition function is defined accordingly. The new update function is defined by replacing the mention of $\mathfrak{r} \in \mathfrak{R}_0$ by its bounded content (which can be determined using the current state). The resulting DSST is (k, K)-bounded by Items (1) and (2) of Lemma 4.50. ◀

Claim 4.54 concludes the construction of a (k, K)-bounded DSST.

4.5 Discussion: recursion for other models

In Definition 4.7, we changed our point of view on nested 2DT, by enabling recursion. It is thus very natural to ask what happens when enabling recursion for pebble transducers, blind pebble transducers and last pebble transducers. In this section, we discuss the relevance of such extensions.

A first observation is that, contrary to recursive marble transducers, recursion for these models has no reason to be well-founded since the size of the input does not decrease strictly when making a call. For blind pebble transducers, the situation is even worse: the input is exactly the same when making a recursive call. Thus, if the same submachine is used twice in the recursion stack of a *recursive blind pebble transducer*, the calls will loop forever. All in all, it does not make sense to define recursive blind pebble transducers, except if they have no recursion, but in this case they are simply blind pebble transducers.

When trying to define *recursive pebble transducers* through Definition 1.36, we first observe that since the input alphabet is enriched at each nested call, one would need arbitrarily large input alphabets to enable unbounded recursion stacks. This is not satisfying since the definition of the machine may become infinite. A solution to overcome this issue is to use only a finite number of "colors" for the pebbles, and allow a submachine to see the colors of the pebbles dropped in some position, but not their number (the case of a single color is in fact mentioned in Section 1.3.2.5 for pebble transducers). However, this model would not preserve regular languages by inverse images, contrary to pebble transducers (Proposition 1.41) and recursive marble transducers (Proposition 4.10), as explained in Example 4.55.

Example 4.55 (Recursive pebble transducer)

Let $A := \{a, b\}$. The indicator function of the (non-regular) language $\{a^n b^n \mid n \geqslant 0\}$ can be computed by a recursive pebble transducer which makes recursive calls in positions $1, 2n, 2, 2n-1, \dots,$ until it reaches the middle of the word.

As a consequence, recursive pebble transducers are far more expressive than all the models described in Part I. We believe that its study is out of reach (and even out of scope) for automata theory.

Recursive last pebble transducers can be defined in a easier way, since only one pebble is visible at any time of the computation. This model was in fact introduced over trees in [EHS07, Section 2]. It provides a natural generalization of recursive marble transducers. It follows from [EHS07, Theorem 5][10] that it preserves regular languages by inverse image (which generalizes Proposition 4.10). We believe that recursive last pebble transducers are a valuable object of study in our context. In particular, Conjecture 4.56 claims that they can be optimized, which generalizes[11] Theorem 4.12.

Conjecture 4.56 (Optimization of recursive last pebble transducers)

A function $f: A^* \to B^*$ computed by a recursive last pebble transducer can be computed by a last pebble transducer if and only if $|f(u)| = \mathcal{O}(|u|^k)$ for some $k \geqslant 0$. This result can be shown by adapting the techniques of Chapter 3 and the construction is effective.

If Conjecture 4.56 holds, the function inner-squaring cannot be computed by a recursive last pebble transducer because of Theorem 1.48 (the proof argument is same as in Proposition 3.14). As a consequence, it seems that recursive last pebble transducers do not capture the whole class of polyregular functions. A generic model which subsumes both recursive last pebble transducers and pebble transducers is presented in [EHS07, Section 2]. It consists in allowing a bounded number k of pebbles to be always visible within the unbounded recursion stack of a recursive last pebble transducer. This extended model preserves regular languages by inverse images (see again [EHS07, Theorem 5]).

[10] This result is in fact shown for tree languages. Interestingly, the author is not aware of a way to adapt the proof to words languages without using trees as an intermediate model.

[11] However, the proof techniques of Chapter 4 do not seem to be relevant in this context, since they are very specific to streaming string transducers. We believe that the techniques of Chapter 3 are more versatile.

Part II

Class membership problems for commutative outputs

Chapter 5

Polyregular functions with commutative outputs

> L'ordre est le plaisir de la raison: mais le désordre est le délice de l'imagination.
>
> Paul Claudel, *Le soulier de satin*

In Part I we have studied word-to-word transductions, i.e. functions of type $A^* \to B^*$ computed by transducers which concatenate output words from B^* when making their transitions. The mainspring of Part II is to study what happens when replacing the free monoid B^* by some (infinite) commutative monoid $\mathbb{S}$, which defines functions of type $A^* \to \mathbb{S}$. In this setting, the key observation is that the order in which the output is produced no longer has importance, due to commutativity.

We shall mainly focus on the cases $\mathbb{S} := \mathbb{N}$ (which corresponds to word-to-word functions with unary output alphabet) and $\mathbb{S} := \mathbb{Z}$. The main goal of Chapter 5 is to provide optimization results for pebble transducers with output in $\mathbb{S}$, similar to the results of Chapters 3 and 4. Furthermore, we shall see that they describe robust and natural classes of functions from finite words to integers.

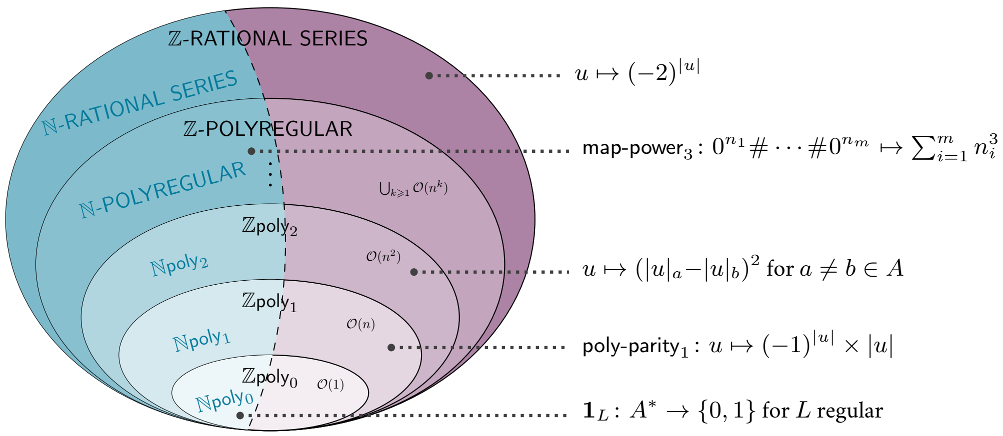


Figure 5.1: Classes of functions studied in Chapter 5 with $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$.

Given a commutative monoid $\mathbb{S}$, we define in Section 5.1 the class of *$\mathbb{S}$-polyregular functions* as the class of functions computed by pebble transducers with output in $\mathbb{S}$. We show that it coincides with the class of functions computed by marble transducers or last pebble transducers with output in $\mathbb{S}$ (which is a major difference with the case of word-to-word functions). Furthermore, we describe another equivalent model called *counting transducers*, which simplifies pebble transducers with output in $\mathbb{S}$ thanks to commutativity. This model is roughly the same as Schützenberger's *finite counting automata* [Sch62].

For $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$, we claim in Section 5.2 that $\mathbb{S}$-polyregular functions describe a robust subclass of $(\mathbb{S}, +, \times)$-rational series[1]. Furthermore, we provide an optimization result for $\mathbb{S}$-polyregular functions, by showing that $f\colon A^* \to \mathbb{S}$ computed by a k-pebble transducer can be computed by a ℓ-pebble transducer for a given $1 \leqslant \ell \leqslant k$ if and only if[2] $|f(u)| = \mathcal{O}(|u|^\ell)$. This property is depicted in Figure 5.1 and all conversions between the models are effective. As a consequence, the lack of correspondence between asymptotic growth and number of nested layers for word-to-word polyregular functions (recall Section 1.3.4) is exclusively related to the (non-commutative) word combinatorics of the output.

The proof of the optimization result goes over Sections 5.3 to 5.5. The case $\mathbb{S} := \mathbb{N}$ already follows from the equivalence with marble transducers and the results of Chapter 4. However, the case $\mathbb{S} := \mathbb{Z}$ is more complex since the presence of negative integers enables to "remove" portions of the output, and thus it can make the asymptotic growth lower than expected. Therefore the author is not aware of a way to adapt the proof of Chapter 4 in this setting, and we instead leverage the tools from Chapters 2 and 3. These techniques also provide a stepping stone towards the more involved proof of Chapter 6.

Finally, we discuss in Section 5.6 partial results for the class membership problem from $\mathbb{Z}$-polyregular functions to $\mathbb{N}$-polyregular functions, which is open to the knowledge of the author.

The contributions presented in this chapter are based on the proof techniques of [Dou21, Dou22] and on part of the results of [CDL23].

5.1 Polyregular functions with commutative output

The goal of this section is to introduce the class of *polyregular functions* which have output in an (infinite) commutative monoid $\mathbb{S}$. In this setting, we shall obtain a simpler description than with pebble transducers: the main idea is that since the output is commutative, the ordering in which the reading heads are moved has no importance, and one-way moves are in fact sufficient (see Algorithm 5.13).

5.1.1 Pebble transducers with commutative output

Let $(\mathbb{S}, +)$ be a (possibly infinite) commutative monoid. We define *$\mathbb{S}$-polyregular functions* as a class of functions of type $A^* \to \mathbb{S}$ where A is a finite alphabet. In the case $\mathbb{S} = \mathbb{N}$, the goal is to capture the functions $f\colon A^* \to \mathbb{N}$ such that the function $g\colon A^* \to \{1\}^*, u \mapsto 1^{f(u)}$ is polyregular. In the case $\mathbb{S} = \mathbb{Z}$, we want to capture the functions $f\colon A^* \to \mathbb{Z}$ which are obtained by summing the output letters of a polyregular function $g\colon A^* \to \{\pm 1\}^*$. More generally, we let $\mathbb{S}^*$ be the set of finite words over $\mathbb{S}$.

Definition 5.2 ($\mathbb{S}$-polyregular functions)

The class of *$\mathbb{S}$-polyregular functions* is the class of functions of shape $\mathsf{sum} \circ g\colon A^* \to \mathbb{S}$ where $g : A^* \to \mathbb{S}^*$ is polyregular[3] and $\mathsf{sum}\colon \mathbb{S}^* \to \mathbb{S}$ is the sum operation in $\mathbb{S}$.

[1] In the literature, rational series over other semirings such as $(\mathbb{Z} \cup \{\infty\}, \min, +)$ have also been studied. We shall not deal with such series in this manuscript, and it is well-known that such classes are incomparable with $(\mathbb{Z}, +, \times)$-rational series.

[2] Here $|\cdot|$ denotes at the same time the absolute value of an integer and the size of a word.

We denote by $\mathbb{S}$poly the class of $\mathbb{S}$-polyregular functions. More precisely, for all $k \geqslant 1$, we denote by $\mathbb{S}\mathsf{poly}_k$ the class of functions of shape $\mathsf{sum} \circ g \colon A^* \to \mathbb{S}$ where the function $g \colon A^* \to \mathbb{S}^*$ is computed by a k-pebble transducer. We let $\mathbb{S}\mathsf{poly}_0$ be the class of functions $f \colon A^* \to \mathbb{S}$ whose image $f(A^*)$ is finite and such that $f^{-1}(\{\delta\})$ is a regular language for all $\delta \in \mathbb{S}$. We also let $\mathbb{S}\mathsf{poly}_{-1}$ be the singleton set which contains the constant function $u \mapsto 0$ where 0 denotes the neutral element of the commutative monoid $(\mathbb{S}, +)$. Observe that $\mathbb{S}\mathsf{poly}_k \subseteq \mathbb{S}\mathsf{poly}_{k+1}$ for all $k \geqslant -1$.

Example 5.3 (Counting letters)

Let $k \geqslant 0$ and $a_1, \dots, a_k \in A$. Let $\mathsf{nb}_{a_1,\dots,a_k} \colon A^* \to \mathbb{N}$ be the function which maps u to $|u|_{a_1} \times \cdots \times |u|_{a_k}$. It belongs to $\mathbb{N}\mathsf{poly}_k$ since a k-pebble transducer can use its k layers to find all the tuples of positions $(i_1, \dots, i_k)$ of its input which are labelled by $(a_1, \dots, a_k)$.

Example 5.4 (Map power)

Let $k \geqslant 0$ and $A := \{0, \#\}$. Let $\mathsf{map\text{-}power}_k \colon A^* \to \mathbb{N}$ be the function which maps an input of shape $0^{n_1}\# \cdots \#0^{n_m} \in A^*$ to $\sum_{i=1}^{m} n_i^k$. This function belongs to $\mathbb{N}\mathsf{poly}_k$.

Example 5.5 (Polynomial parity)

Let $\mathsf{poly\text{-}parity}_k \colon A^* \to \mathbb{Z}$ be the function mapping u to $(-1)^{|u|} \times |u|^k$. This function belongs to $\mathbb{Z}\mathsf{poly}_k$ thanks to a k-pebble transducer which produces an output either in $\{+1\}^*$ or $\{-1\}^*$.

Now we suggest in Claim 5.6 that $\mathbb{S}$-polyregular functions are essentially trivial when $\mathbb{S}$ is finite. In the rest of Part II, we shall mainly focus on the cases $\mathbb{S} = \mathbb{N}$ and $\mathbb{S} = \mathbb{Z}$.

Claim 5.6 (Finite monoids)

If $\mathbb{S}$ is a finite commutative monoid, then $\mathbb{S}\mathsf{poly} = \mathbb{S}\mathsf{poly}_0$.

Proof idea. It is sufficient to show that for all $k \geqslant 0$, $f \in \mathbb{S}\mathsf{poly}_k$ and $\delta \in \mathbb{S}$, $f^{-1}(\{\delta\})$ is a regular language. This result is shown by induction on $k \geqslant 0$ when starting from a k-pebble transducer. The induction step is similar to the argument of Section 1.3.2.2 for showing that pebble transducers with non-total submachines compute functions whose domain is regular. ◀

Observe that one can shift closure properties from polyregular to $\mathbb{S}$-polyregular functions.

Proposition 5.7 (Pre-composition by regular functions)

For all $k \geqslant 0$, the class $\mathbb{S}\mathsf{poly}_k$ is (effectively) closed under pre-composition by regular functions.

Proof. The result for $k = 0$ follows from the fact that regular functions preserve regular languages by inverse images. For $k \geqslant 1$, we rely on Theorem 1.43 which implies that the class of functions computed by k-pebble transducers is closed under pre-composition by regular functions. ◀

[3] In Chapter 1, we have defined polyregular functions of type $A^* \to B^*$ when B is finite. Here $\mathbb{S}$ is not finite in general, thus we formally consider polyregular functions of type $A^* \to F^*$ where F is a finite subset of $\mathbb{S}$.

Jump to contents

5.1.2 Counting transducers

Now we describe a simple computation model named *counting transducers*, which captures $\mathbb{S}$-polyregular functions. It will be used to show the main results of Chapters 5 and 6.

If A is an alphabet, we denote by $\mathsf{RegProp}_k(A)$ the set of regular languages over $A \times \{0,1\}^k$. The idea is to encode distinguished positions within the boolean components. If $L \in \mathsf{RegProp}_k(A)$, we define the *counting function* $\#L\colon A^* \to \mathbb{N}$ as follows for $u \in A^*$ (recall that for $1 \leqslant i \leqslant |u|$, the word $u{\bullet}i \in (A \times \{0,1\})^*$ is defined as $(u[1],0)\cdots(u[i-1],0)(u[i],1)(u[i+1],0)\cdots(u[|u|],0)$):

$$\#L(u) := |\{(i_1,\dots,i_k) \in [1{:}|u|]^k \mid u{\bullet}i_1{\bullet}i_2\cdots{\bullet}i_k \in L\}|.$$

Informally, such a function describes the number of assignments[4] of k pebbles dropped on input u, while verifying the regular property L. Since other inputs are not used, one can assume without losing generalities that any word $v \in L$ has shape $u{\bullet}i_1{\bullet}i_2\cdots{\bullet}i_k$ for $1 \leqslant i_1,\dots,i_k \leqslant |u|$.

Example 5.8 (Indicator functions)

If $L \in \mathsf{RegProp}_0(A)$, then $\#L$ is the indicator function $\mathbf{1}_L\colon A^* \to \{0,1\}$ of the language L (which is a regular property).

Example 5.9 (Map power)

One can build $L \in \mathsf{RegProp}_k(A)$ such that $\#L = \mathsf{map\text{-}power}_k$, by making L check if the k marked positions belong to the same 0^{n_i} factor.

We present the computation model of *counting transducers*, which relies on the $\#L$ functions. It is inspired by the *finite counting automata* which were introduced by Schützenberger in [Sch62]. An equivalent definition is presented in [CDL23] by using a logical formalism[5].

Definition 5.10 (Counting transducer)

Let $k \geqslant 0$. A *k-counting transducer* $\mathscr{T} = (A, \mathbb{S}, (\delta_i, L_i)_{1\leqslant i\leqslant n})$ consists of:

- an input alphabet A and an output commutative monoid $\mathbb{S}$;
- a finite sequence $(\delta_i, L_i)_{1\leqslant i\leqslant n}$ of *production pairs* with $\delta_i \in \mathbb{S}$ and $L_i \in \mathsf{RegProp}_k(A^*)$.

The semantics of the k-counting transducer $\mathscr{T}$ is defined as follows. First, the commutative monoid $(\mathbb{S}, +)$ can be equipped with a product operation $\mathbb{S} \times \mathbb{N} \to \mathbb{S} : (\delta, n) \mapsto \delta \cdot n := \delta + \cdots + \delta$ (n times). Given a function $g\colon A^* \to \mathbb{N}$ and $\delta \in \mathbb{S}$, one can define the function $\delta \cdot g\colon A^* \to \mathbb{S}$ which maps u to $\delta \cdot g(u)$. The function computed by $\mathscr{T}$ is defined as $\sum_{i=1}^n \delta_i \cdot \#L_i$.

Example 5.11 ($\mathbb{S}\mathsf{poly}_0$)

The class $\mathbb{S}\mathsf{poly}_0$ coincides with the class of functions computed by 0-counting transducers. It contains the functions $\sum_{i=1}^n \delta_i \cdot \mathbf{1}_{L_i}$ where $\delta_i \in \mathbb{S}$ and $L_i \subseteq A^*$ is regular for all $1 \leqslant i \leqslant n$.

[4] Given a language $L \in \mathsf{RegProp}_k(A)$, one can build a *monadic second-order formula* (MSO formula for short) $\varphi(x_1,\dots,x_k)$ where $x_1,\dots,x_k$ are free first-order variables, such that $\#L(u)$ is the number of assignments $x_1,\dots,x_k$ which make φ true in the model $u \in A^*$ (see e.g. [Tho97]). This formalism is equivalent to ours, but we chose not to use it.

[5] In [CDL23], the counting transducers are built by using MSO formulas with free variables instead of languages of RegProp. They can therefore be seen as a particular case of the MSO interpretations from [BKL19]. Once more, we chose not to use this equivalent formalism, since we never deal with logic in this manuscript.

Example 5.12 (Polynomial parity)

Let $\mathbf{1}_{\text{odd}}$ (resp. $\mathbf{1}_{\text{even}}$) be the indicator function of words of odd (resp. even) length. For all $k \geqslant 0$ and $u \in A^*$, we have $\text{poly-parity}_k(u) = \mathbf{1}_{\text{even}}(u) \times |u|^k - \mathbf{1}_{\text{odd}}(u) \times |u|^k$. This function can therefore be computed by a k-counting transducer with two production pairs.

Observe that a k-counting transducer can be seen as an algorithm with k nested (one-way) for loops, as described in Algorithm 5.13 for $k = 3$ (and $n = 2$). When reaching the most inner loop, it selects an output depending on a regular property of the input in which the position of the loop indices are marked. In other words, a k-counting transducer can be seen as some kind of bimachine with k pebbles.

Algorithm 5.13: Implementation of a 3-counting transducer with nested loops.

for i_1 **in** $[1{:}|u|]$ **do**
    for i_2 **in** $[1{:}|u|]$ **do**
        for i_3 **in** $[1{:}|u|]$ **do**
            if $u \bullet i_1 \bullet i_2 \bullet i_3 \in L$ **then**
                Output δ
            end
            if $u \bullet i_1 \bullet i_2 \bullet i_3 \in L'$ **then**
                Output δ'
            end
        end
    end
end

Note that if $k \geqslant 1$ and $L \in \text{RegProp}_k(A)$ then $\#L(\varepsilon) = 0$. Therefore, if f is computed by a k-counting transducer, then $f(\varepsilon)$ is the neutral element of $\mathbb{S}$. We shall freely assume that $f(\varepsilon)$ can be chosen as any element of $\mathbb{S}$, up to adding a specific output. Under this assumption, it is easy to see that for all $0 \leqslant \ell \leqslant k$, a k-counting transducer can always simulate an ℓ-counting transducer.

5.1.3 Equivalence between pebbles, marbles and counting

In this section our goal is to (easily) show that the class of functions computed by counting transducers with output $\mathbb{S}$ is exactly the class of $\mathbb{S}$-polyregular functions. In the context of commutative outputs, we also deduce that pebble and marble transducers have the same expressive power.

Let us first note that the languages $L \in \text{RegProp}_k(A)$ can be normalized in a left-to-right fashion.

Claim 5.14 (Normalization of counting functions)

Let $k \geqslant 0$ and $L \in \text{RegProp}_k(A)$. One can build $L_1, \dots, L_n \in \bigcup_{\ell \leqslant k} \text{RegProp}_\ell(A)$ such that:

- $\#L = \#L_1 + \cdots \#L_n$;
- for all $1 \leqslant j \leqslant n$, if $L_j \in \text{RegProp}_\ell(A)$ and $u \bullet i_1 \bullet i_2 \cdots \bullet i_\ell \in L_j$ then $i_1 < \cdots < i_\ell$.

Proof idea. Split L in disjoint languages depending on the relative positions of the $i_1, \dots, i_k$. ◀

Now we are ready to show Theorem 5.15. This easy result originates from [Dou21, Corollary 4.5][6] for the case of $\mathbb{S} = \mathbb{N}$ and from [CDL23, Proposition II.11] for $\mathbb{S} = \mathbb{Z}$.

[6] This corollary from [Dou21] in fact relies on [Dou21, Lemma 4.4], which shows a stronger result for nested DSST.

Theorem 5.15 (Pebble = Marble = Counting)

Let $\mathbb{S}$ be a commutative monoid. Given $f : A^* \to \mathbb{S}$ and $k \geqslant 1$, the following are equivalent:

(1) $f = \mathsf{sum} \circ g$ for $g : A^* \to \mathbb{S}^*$ computed by a k-pebble transducer (i.e. $f \in \mathbb{S}\mathsf{poly}_k$);
(2) $f = \mathsf{sum} \circ g$ for $g : A^* \to \mathbb{S}^*$ computed by a k-marble transducer;
(3) f is computed by a k-counting transducer.

The conversions are effective.

Proof. Item (2) $\Rightarrow$ Item (1) is trivial. For Item (1) $\Rightarrow$ Item (3), we consider a k-pebble transducer $\mathscr{P}$ which computes a function $g : A^* \to \mathbb{S}^*$. To simplify the notations, we assume that $\mathscr{P}$ has shape $\mathscr{T}_1\langle\mathscr{T}_1 \cdots \langle\mathscr{T}_k\rangle\rangle$ (i.e. it consists in a single branch). In this case:

- for all $1 \leqslant j \leqslant k{-}1$, the 2DT $\mathscr{T}_j$ has shape $(A \times \{0,1\}^{j-1}, \mathscr{T}_{j+1}, Q_j, q_{j,0}, F_j, \delta_j, \lambda_j)$;
- the 2DT $\mathscr{T}_k$ has shape $(A \times \{0,1\}^{k-1}, \mathbb{S}, Q_k, q_{k,0}, F_k, \delta_k, \lambda_k)$.

Since $\mathscr{T}_1, \dots, \mathscr{T}_k$ are normalized, they produce at most one letter at each transition. By using the transition monoids, one can build for all $q_1, \dots, q_k \in Q_1 \times \cdots \times Q_k$, a regular language $L_{q_1,\dots,q_k,\delta} \in \mathsf{RegProp}_k(A)$ such that $u{\bullet}i_1{\bullet}i_2 \cdots {\bullet}i_k \in L_{q_1,\dots,q_k,\delta}$ if and only if:

- for all $1 \leqslant j \leqslant k$, the configuration (q_j, i_j) occurs in the accepting n-run of $\mathscr{T}_j$ labelled by $u{\bullet}i_1{\bullet}i_2 \cdots {\bullet}i_{j-1}$ (in other words, $(q_j,i_j) \in \mathsf{cross}_{\mathscr{T}_j}^{u\bullet i_1 \bullet i_2 \cdots \bullet i_{j-1}}(\{i_j\})$);
- $1 \leqslant j \leqslant k{-}1$, $\lambda_j(q_j, u[i_j]) = \mathscr{T}_{j+1}$ and $\lambda_k(q_j, u[i_j]) = \delta \in \mathbb{S}$.

Finally, we build a k-counting transducer whose pairs are the $(L_{q_1,\dots,q_k,\delta}, \delta)$. Since $\mathbb{S}$ is commutative, it is easy to see that this k-counting transducer computes $\mathsf{sum} \circ g$.

For Item (3) $\Rightarrow$ Item (2), we first show the following result by leveraging Claim 5.14.

Claim 5.16 (Marbles for basic counting functions)

Let $k \geqslant 1$ and $L \in \mathsf{RegProp}_k(A)$. One can build a k-marble transducer which computes the function of type $A^* \to \{1\}^*$ mapping $u \in A^*$ to $1^{\#L(u)}$.

Proof idea. Thanks to Claim 5.14 (and up to simulating the execution of several marble transducers), one can assume that whenever $u{\bullet}i_1{\bullet}i_2 \cdots {\bullet}i_k \in L$ then $i_1 < i_2 < \cdots < i_k$. Let $\mu : A \times \{0,1\}^k \to \mathbb{M}$ be a morphism into a finite monoid which recognizes L. The head $\mathscr{T}$ of our k-marble transducer behaves as follows on input $u \in A^*$: for all $1 \leqslant i \leqslant |u|$, it makes a nested call to in position i to a submachine $\mathscr{T}_m$ where:

$$m := \mu((u[i], 0, \dots, 0, 1)(u[i{+}1], 0, \dots, 0, 0) \cdots (u[|u|], 0, \dots, 0, 0)).$$

Intuitively, this operation corresponds to giving m as an argument of the submachine, so that it keeps track of the portion of the input that it can no longer see. The nested calls are built in a inductive similar fashion. Finally, a leaf submachine uses lookarounds and the aforementioned monoid information to determine if its current position together with the positions of the nested calls describe positions $i_1 < \cdots < i_k$ such that $u{\bullet}i_1{\bullet}i_2 \cdots {\bullet}i_k \in L$. ◀

Now let us consider a k-counting transducer whose pairs are (δ_i, L_i) for $1 \leqslant i \leqslant n$. For all $1 \leqslant i \leqslant n$, we apply Claim 5.16 in order to build a k-marble transducer $\mathscr{M}_i$ which computes $u \mapsto 1^{\#L_i(u)}$. By replacing each output word 1^n by $\delta_i \cdot n \in \mathbb{S}$, one can build a k-marble transducer $\overline{\mathscr{M}_i}$ which compute some $\overline{g_i} : A^* \to \mathbb{S}^*$ such that $\delta_i \cdot \#L_i = \mathsf{sum} \circ \overline{g_i}$. Finally, we build a k-marble transducer which sequentially simulates the transducers $\mathscr{M}_i$ for $1 \leqslant i \leqslant n$. ◀

As an immediate consequence of Theorem 5.15, note that for $k \geqslant 1$ we have $f \in \mathbb{S}\mathsf{poly}_k$ if and only if $f = \mathsf{sum} \circ g$ for some $g : A^* \to \mathbb{S}^*$ computed by a last k-pebble transducer. We shall see in Chapter 6 that this result does not hold for blind pebble transducers.

If F is a class of functions of type $A^* \to \mathbb{N}$, we define $\mathsf{Span}_{\mathbb{S}}(F) := \{\sum_i \delta_i \cdot f_i \mid \delta_i \in \mathbb{S}, f_i \in F\}$. A synthetic reformulation of Theorem 5.15 is given by Corollary 5.17.

Corollary 5.17 (Linear combinations)

For all $k \geqslant 0$, $\mathbb{S}\mathsf{poly}_k = \mathsf{Span}_{\mathbb{S}}(\{\#L \mid L \in \mathsf{RegProp}_k\})$.

Proof. Use Item (1) $\Leftrightarrow$ Item (3) in Theorem 5.15. ◀

5.2 Rational series and membership problems

We have observed in Chapter 4 that the class of functions of type $A^* \to \{1\}^*$ computed by k-marble transducers is was (up to identifying 1^n with $n \in \mathbb{N}$) the class of $(\mathbb{N}, +, \times)$-rational series $f\colon A^* \to \mathbb{N}$ such that $f(u) = \mathcal{O}(|u|^k)$. Thanks to Theorem 5.15, this class of functions is exactly $\mathbb{N}\mathsf{poly}_k$. As a consequence, one can minimize the number k of layers which is needed to compute such a function.

The goal of Section 5.2 is to discuss in more detail the connection between $\mathbb{N}$-polyregular functions and $(\mathbb{N}, +, \times)$-rational series and to generalize it to $\mathbb{Z}$-polyregular functions and $(\mathbb{Z}, +, \times)$-rational series. Furthermore, we also provide an optimization result for $\mathbb{Z}$-polyregular functions and show that the asymptotic growth of a function is connected to the minimal number of layers needed to compute it. We are not aware of a way to adapt the techniques of Chapter 4 (which use weighted automata) to the semiring $(\mathbb{Z}, +, \times)$, therefore we shall rely on factorization forests instead.

5.2.1 Combinators for rational series

Let $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$. It is well-known since Schützenberger that the class of $(\mathbb{S}, +, \times)$-rational series introduced in Definition 4.43 (often $\mathbb{S}$-rational series in the following) can be described using the indicator functions $\mathbf{1}_L$ of regular languages $L \subseteq A^*$ and basic combinators, in the spirit of *regular expressions*. Given $f, g : A^* \to \mathbb{S}$ and $\delta \in \mathbb{S}$, we define the following combinators:

- the *external product* $\delta \cdot f\colon u \mapsto \delta \times f(u)$;
- the *sum* $f + g : u \mapsto f(u) + g(u)$;
- the *Hadamard product* $f \times g\colon u \mapsto f(u) \times g(u)$;
- the *Cauchy product* $f \otimes g\colon u \mapsto \sum_{u=vw} f(v) \times g(w)$;
- if and only if $f(\varepsilon) = 0$, the *Kleene star* $f^* := \sum_{n \geqslant 0} f^n$ where $f^0 : \varepsilon \mapsto 1, u \neq \varepsilon \mapsto 0$ is neutral for Cauchy product and $f^{n+1} := f \otimes f^n$.

The characterization of $\mathbb{S}$-rational series is recalled in Theorem 5.18 (see [BR11, Theorem 7.1 p 17]).

Theorem 5.18 (Regular expressions for rational series)

Let $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$. The class of $\mathbb{S}$-rational series is the smallest class of functions of type $A^* \to \mathbb{S}$ which contains the indicator functions of regular languages and is closed under external products, sums, Cauchy products and Kleene stars. Furthermore, it is closed under Hadamard products.

It is easy to see from Corollary 5.17 that the class of $\mathbb{S}$-polyregular functions is closed under taking Cauchy products and Hadamard products, as claimed in Lemma 5.19.

Lemma 5.19 (Closure properties of $\mathbb{S}$-polyregular functions)

Let $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$. The class of $\mathbb{S}$-polyregular functions is closed under Cauchy products and Hadamard products. More precisely, if $f \in \mathbb{S}\mathsf{poly}_k$ and $g \in \mathbb{S}\mathsf{poly}_\ell$, then $f \otimes g \in \mathbb{S}\mathsf{poly}_{k+\ell+1}$ and $f \times g \in \mathbb{S}\mathsf{poly}_{k+\ell}$. The constructions are effective.

Proof. For all $f, g, h : A^* \to \mathbb{S}$ and $\gamma, \delta \in \mathbb{Z}$, we have $(\gamma \cdot f + \delta \cdot g) \otimes h = \gamma \cdot (f \otimes g) + \delta \cdot (g \otimes h)$ and $(\gamma \cdot f + \delta \cdot g) \times h = \gamma \cdot (f \times g) + \delta \cdot (g \times h)$. Therefore, it is sufficient to show the results when $f = \#L$ and $g = \#R$ with $L \in \mathsf{RegProp}_k(A)$ and $R \in \mathsf{RegProp}_\ell(A)$.

We only deal with the (most difficult) case of the Cauchy product. For all $u \in A^*$ we have:

$$
\begin{aligned}
(\#L \otimes \#R)(u) &= \sum_{0 \leqslant i \leqslant |u|} \sum_{1 \leqslant i_1, \dots, i_k \leqslant i} \sum_{i < j_1, \dots, j_\ell \leqslant |u|} \mathbf{1}_{(u[1{:}i]) \bullet i_1 \bullet \cdots \in L} \times \mathbf{1}_{(u[i+1{:}|u|]) \bullet j_1 \bullet \cdots \in R} \\
&= \#L(\varepsilon) \times \#R(u) \\
&+ \underbrace{\sum_{1 \leqslant i \leqslant |u|} \sum_{1 \leqslant i_1, \dots, i_k \leqslant i} \sum_{i < j_1, \dots, j_\ell \leqslant |u|} \mathbf{1}_{(u[1{:}i] \bullet i_1) \bullet \cdots \in L} \times \mathbf{1}_{(u[i+1{:}|u|]) \bullet (j_1 - i) \bullet \cdots \in R}}_{= \#S(u)}
\end{aligned}
$$

where $S \in \mathsf{RegProp}_{k+\ell+1}(A)$ is such that for all $u \in A^*$ and $1 \leqslant i_1, \dots, i_k, j_1, \dots, j_\ell, i \leqslant |u|$, $u \bullet i_1 \bullet \cdots \bullet i_k \bullet j_1 \bullet \cdots \bullet j_\ell \bullet i \in S$ holds if and only if the conditions $1 \leqslant i_1, \dots, i_k \leqslant i$, $i < j_1, \dots, j_\ell \leqslant |u|$, $(u[1{:}i]) \bullet i_1 \bullet \cdots \in L$ and $(u[i+1{:}|u|]) \bullet (j_1 - i) \bullet \cdots \in R$ hold. ◀

However, we note in Example 5.20 that $\mathbb{S}$-polyregular functions are not closed under Kleene stars.

Example 5.20 (Kleene star)

The function $\mathsf{power\text{-}2} \colon u \mapsto (-2)^{|u|}$ is not $\mathbb{Z}$-polyregular since $|\mathsf{power\text{-}2}(u)| \neq \mathcal{O}(|u|^k)$ for some $k \geqslant 0$. However, $\mathsf{power\text{-}2} = ((-3) \cdot \mathbf{1}_{A^+})^*$ and $(-3) \cdot \mathbf{1}_{A^+}$ is $\mathbb{Z}$-polyregular.

As a consequence of Lemma 5.19, if $L \subseteq A^*$ is regular and $f \in \mathbb{S}\mathsf{poly}_k$, then $\mathbf{1}_L \otimes f \in \mathbb{S}\mathsf{poly}_{k+1}$. Lemma 5.21 states that such functions actually generate the whole space $\mathbb{S}\mathsf{poly}_{k+1}$.

Lemma 5.21 (Inductive construction of $\mathbb{S}$-polyregular functions)

Let $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$. For all $k \geqslant 0$, the following equality holds and the conversions are effective:

$$\mathbb{S}\mathsf{poly}_{k+1} = \mathsf{Span}_{\mathbb{S}}(\{\mathbf{1}_L \otimes f \mid L \text{ regular language}, f \in \mathbb{S}\mathsf{poly}_k\}).$$

Proof. As in the proof of Lemma 5.19, it is sufficient to show that $\#S$ for $S \in \mathsf{RegProp}_{k+1}(A)$ can be written as a linear combination of $\mathbf{1}_L \otimes f$ where R is regular and $f \in \mathbb{S}\mathsf{poly}_k$. By Claim 5.14 and since $\mathbf{1}_{\{\varepsilon\}} \otimes f = f$ for all $f \colon A^* \to \mathbb{S}$, one can assume that if $u \bullet i_1 \cdots \bullet i_k \in S$ then $i_1 < i_2 < \cdots < i_k$. Let $\mu : A \times \{0,1\}^k \to \mathbb{M}$ be a morphism into a finite monoid which recognizes L. It is easy to check that one can build $L_m \in \mathsf{RegProp}_0(A)$ and $R_m \in \mathsf{RegProp}_k(A)$ for $m \in \mathbb{M}$ such that $\#L(u) = \sum_{m \in \mathbb{M}} \sum_{1 \leqslant i_1 \leqslant |u|} \mathbf{1}_{L_m}(u[1{:}i_1]) \times \#R_m(u[i_1{+}1{:}|u|])$ for all $u \in A^+$. If we enforce $\varepsilon \notin L_m$ for all $m \in M$, this equation can be transformed into a Cauchy product. ◀

5.2.2 $\mathbb{S}$-polyregular functions as $\mathbb{S}$-rational series

Now, we are ready for $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$ to characterize the class of $\mathbb{S}$-polyregular functions as a subclass of $(\mathbb{S}, +, \times)$-rational series. Theorem 5.22 is presented in [Dou22, Theorem 3.3] for $\mathbb{N}$-polyregular functions and in [CDL23, Theorem II.18] for $\mathbb{Z}$-polyregular functions. Both proofs are not self-contained and rely on classical results on $\mathbb{S}$-rational series that we shall not detail in this manuscript. From now on, $|\cdot|$ is used to denote both the size of a word and the absolute value of an integer.

Theorem 5.22 ($\mathbb{S}$-polyregular functions as $\mathbb{S}$-rational series)

Let $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$. Given $f : A^* \to \mathbb{S}$, the following conditions are equivalent:

(1) f is $\mathbb{S}$-polyregular;
(2) f is a $\mathbb{S}$-rational series and $|f(u)| = \mathcal{O}(|u|^k)$ for some $k \geqslant 0$;
(3) f belongs the smallest class of functions of type $A^* \to \mathbb{S}$ containing the indicator functions of regular languages, and closed under external products, sums and Cauchy products.

The conversions are effective and one can decide if a $\mathbb{S}$-rational series is $\mathbb{S}$-polyregular.

Proof sketch. Equivalence between Item (1) and Item (3) follows from Lemmas 5.19 and 5.21. Equivalence between Item (2) and Item (3) follows from [BR11, Exercise 1.2 p 169] in the case of $\mathbb{S} = \mathbb{N}$. For $\mathbb{S} = \mathbb{Z}$, this result follows from [BR11, Corollary 2.6 p 159]. However, these results are not explicitly claimed to be effective. To ensure effectivness, one can start from a $\mathbb{S}$-rational series $f\colon A^* \to \mathbb{S}$, enumerate all the $\mathbb{S}$-polyregular functions $g\colon A^* \to \mathbb{S}$, rewrite them as $\mathbb{S}$-rational series (using Item (3) $\Rightarrow$ Item (2)) and check whether $f = g$ since this property can be decided for $\mathbb{S}$-rational series [BR11, Corollary 3.6 p 38]. Finally, given a $\mathbb{S}$-rational series $f\colon A^* \to \mathbb{S}$, one can decide if $|f(u)| = \mathcal{O}(|u|^k)$ for some $k \geqslant 0$ thanks to [BR11, Corollary 2.4 p 159]. ◀

As mentioned above, the results of Section 4.4 and Theorem 5.15 already gave the equivalence between Item (1) and Item (2) in the case of $\mathbb{N}$-polyregular functions.

Example 5.23 (Polynomial parity)

The function $\mathsf{poly\text{-}parity}_1\colon u \mapsto (-1)^{|u|}|u|$ belongs to $\mathbb{Z}\mathsf{poly}_1$. It can be written $\mathbf{1}_{\mathsf{odd}} \otimes \mathbf{1}_{\mathsf{odd}} + \mathbf{1}_{\mathsf{even}} \otimes \mathbf{1}_{\mathsf{even}} - \mathbf{1}_{\mathsf{even}} \otimes \mathbf{1}_{\mathsf{odd}} - \mathbf{1}_{\mathsf{odd}} \otimes \mathbf{1}_{\mathsf{even}} - \mathbf{1}_{\mathsf{odd}} + \mathbf{1}_{\mathsf{even}}$ and is computed by the $\mathbb{Z}$-automaton:

$$\left(A, [1{:}2], \begin{pmatrix} -1 & 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \mu\colon u \mapsto \begin{pmatrix} -1 & 1 \\ 0 & -1 \end{pmatrix}^{|u|} \right).$$

We finally transfer the decidability of equivalence from $\mathbb{S}$-rational series to $\mathbb{S}$-polyregular functions.

Corollary 5.24 (Equivalence problem)

Let $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$. One can decide if two $\mathbb{S}$-polyregular functions are equal.

Proof. Equivalence is decidable for $\mathbb{Z}$-rational series by [BR11, Corollary 3.6 p 38]. ◀

5.2.3 Optimization theorem for $\mathbb{S}$-polyregular functions

As observed above, a consequence of Theorem 5.15 and of the results of Section 4.4 is that if $f \in \mathbb{N}\mathsf{poly}$ and $k \geqslant 0$, then $f \in \mathbb{N}\mathsf{poly}_k$ if and only if $f(u) = \mathcal{O}(|u|^k)$, and that one can optimize $\mathbb{N}$-polyregular functions. Now our goal is now to show a similar result for $\mathbb{Z}$-polyregular functions.

The author is not aware of a direct way to adapt the proof techniques of Section 4.4 to $\mathbb{Z}$-polyregular functions. Indeed, this previous proof consists in finding patterns in a $(\mathbb{N}, +, \times)$-weighted automaton. Once such a pattern is found, it provides a lower bound on the asymptotic growth of the output. However, in a $(\mathbb{Z}, +, \times)$-weighted automaton, the existence of such a pattern does not provide a global lower bound for the function, because the output produced along this pattern could be compensated by the output produced along another pattern, due to the presence of both negative and positive outputs. In other words, understanding the asymptotic growth of $\mathbb{Z}$-polyregular functions requires a global understanding of the output produced, which is not achieved by using patterns in weighted automata. Therefore we shall instead generalize the notion of production (introduced for 2DT in Definition 2.6) to counting transducers and mix it with factorization forests as we did in the proofs of Section 2.2 and Chapter 3.

Theorem 5.25 originates from [CDL23, Theorem III.3]. The proof of this result goes over Sections 5.3 to 5.5, and it also develops several tools that will be re-used in Chapter 6. Equivalence between asymptotic growth and minimal number of layers was already known since [Sch62] (see [BR11, Corollary 2.6 p 159] for a modern and more readable presentation). However, this historic proof is largely different from ours since it relies on the theory of *$\mathbb{Z}$-modules*. Furthermore, it is not clear[7], whether it provides decidability or effectiveness of the construction, thus to the knowledge of the author this result is new.

Theorem 5.25 (Optimization of pebble transducers with commutative output)

Let $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$. Let $f \in \mathbb{S}\mathsf{poly}$ and $k \geqslant 0$, then $f \in \mathbb{S}\mathsf{poly}_k$ if and only if $|f(u)| = \mathcal{O}(|u|^k)$. This property is decidable. If it holds, one can build a k-counting transducer which computes f.

Proof sketch. The main idea is to show that for all $k \geqslant 1$, a function $f \in \mathbb{S}\mathsf{poly}_k$ can be written as a sum $f_1 + f_2$ where $f_1 \in \mathbb{S}\mathsf{poly}_{k-1}$ and $f_2 \in \mathbb{S}\mathsf{poly}_k$ is such that $f_2 = 0$ whenever the additional condition $|f(u)| = \mathcal{O}(|u|^{k-1})$ holds. Intuitively, f_2 contains the "terms of highest degree" of f. For decidability, we provide a syntactic condition on k-counting transducers called pumpability, which is inspired by the similar notion presented in Chapter 3 for optimizing blind pebble transducers and last pebble transducers. Formally, Theorem 5.25 follows from Theorem 5.54. ◀

As a side result, we shall also show that if $f \in \mathbb{S}\mathsf{poly}_k \smallsetminus \mathbb{S}\mathsf{poly}_{k-1}$ for $k \geqslant 1$, there exist words $v_0, u_1, v_1, \dots, u_k v_k$ such that $|f(v_0 u_1^X v_1 \cdots u_k^X v_K) = \theta(X^k)$. In other words, we obtain a witness for its asymptotic growth. In the next Sections 5.3 and 5.5, we present a toolbox which will be used both for the proof of Theorem 5.25 and for showing the main result of Chapter 6.

5.3 Productions of counting transducers

This section is the counterpart of Section 2.1 when dealing with counting transducers instead of two-way transducers. It will serve as an advanced toolbox in the proofs of Chapters 5 and 6. We first adapt the notion of production to counting transducers, following the definitions of [Dou22, Section 5.2]. Intuitively, it enables to describe in a fine-grained way which elements of the output where produced from which positions. Then we provide several analogues of the results of Sections 2.1.2 and 2.2.1.

5.3.1 Productions over multisets of positions

Our first concern is to define the notion of *production* for counting transducers. Since such machines use several positions of their input at the same time, we shall use *multisets* (recall that such objects are sets

[7] A natural idea would be that the asymptotic growth of the output can "easily" be decided by looking at any minimal (see Section 7.6) $(\mathbb{Z}, +, \times)$-weighted automaton which computes the function. We are not aware of such a statement in the literature.

with multiplicities, i.e. where elements can be duplicated) of sets of positions to describe them.

Recall that double braces $\{\!\{\cdots\}\!\}$ denote multisets. We denote by $\{\!\{s_1\ddagger r_1, \ldots, s_n\ddagger r_n\}\!\}$ a multiset containing n distinct elements $s_1, \ldots, s_n$ of respective multiplicities $r_1, \ldots, r_n$ (therefore it contains $r_1 + \cdots + r_n$ elements). Given $k \geqslant 1$, we let $\mathsf{S}_k := \{(r_1, \ldots, r_n) \in \mathbb{N}^n \mid r_1 + \cdots + r_n = k\}$.

Given u a word, $I_1, \ldots, I_n$ disjoint subsets of $[1{:}|u|]$ and $(r_1, \ldots, r_n) \in \mathsf{S}_k$, we define the *production* of a k-counting transducer over the multiset $\{\!\{I_1\ddagger r_1, \ldots, I_n\ddagger r_n\}\!\}$ as the sum of all its outputs taken from tuples of positions of u which have exactly r_j components in I_j for all $1 \leqslant j \leqslant n$.

Definition 5.26 (Production of a k-counting transducer)

Let $\mathscr{T} = (A, \mathbb{S}, (\delta_i, L_i)_{1\leqslant i\leqslant m})$ be a k-counting transducer, $u \in A^*$ and $I_1, \ldots, I_n$ be disjoint subsets of $[1{:}|u|]$ and $(r_1, \ldots, r_n) \in \mathsf{S}_k$. We consider the multiset $M := \{\!\{I_1\ddagger r_1, \ldots, I_n\ddagger r_n\}\!\}$ and we define the set of tuples which have exactly r_j components in I_j:

$$\mathsf{Tuples}_k(M) := \{(i_1, \ldots, i_k) \mid |\{1 \leqslant \ell \leqslant k \mid i_\ell \in I_j\}| = r_j \text{ for all } 1 \leqslant j \leqslant n\}.$$

We then define the *production* of $\mathscr{T}$ over the multiset M in u as follows:

$$\mathsf{prod}^u_{\mathscr{T}}(M) := \sum_{i=1}^{n} \delta_i \cdot |\{(i_1, \ldots, i_k) \in \mathsf{Tuples}_k(M) \mid u\bullet i_1\cdots\bullet i_k \in L_i\}|.$$

Now, we show how productions from Definition 5.26 can be summed when splitting sets of positions in disjoint subsets. Claim 5.27 can be seen as an analogue of Claim 2.7 for counting transducers.

Claim 5.27 (Productions are additive)

Let $\mathscr{T}$ be a k-counting transducer which computes a function $f : A^* \to \mathbb{S}$. Let $u \in A^*$, $I_1, \ldots, I_n$ be disjoint subsets of $[1{:}|u|]$ and $(r_1, \ldots, r_n) \in \mathsf{S}_k$. If $I_1 = J_1 \uplus \cdots \uplus J_p$, then:

$$\mathsf{prod}^u_{\mathscr{T}}(\{\!\{I_1\ddagger r_1, \ldots, I_n\ddagger r_n\}\!\}) = \sum_{(j_1,\ldots,j_p)\in\mathsf{S}_{r_1}} \mathsf{prod}^u_{\mathscr{T}}(\{\!\{J_1\ddagger j_1, \ldots, J_p\ddagger j_p, I_2\ddagger r_2, \ldots, I_n\ddagger r_n\}\!\}).$$

In particular, if $I_1, \ldots, I_n$ is a partition of $[1{:}|u|]$ we get:

$$f(u) = \sum_{(r_1,\ldots,r_n)\in\mathsf{S}_k} \mathsf{prod}^u_{\mathscr{T}}(\{\!\{I_1\ddagger r_1, \ldots, I_n\ddagger r_n\}\!\}).$$

Proof. Let $M := \{\!\{I_1\ddagger r_1, \ldots, I_n\ddagger r_n\}\!\}$. For the first equation, it suffices to observe that:

$$\mathsf{Tuples}_k(M) = \biguplus_{(j_1,\ldots,j_p)\in\mathsf{S}_{r_1}} \mathsf{Tuples}_k(\{\!\{J_1\ddagger j_1, \ldots, J_p\ddagger j_p, I_2\ddagger r_2, \ldots, I_n\ddagger r_n\}\!\}).$$

To show the second equation of Claim 5.27, we first observe that $f(u) = \mathsf{prod}^u_{\mathscr{T}}(\{\!\{[1{:}|u|]\ddagger k\}\!\})$ and then we apply the first equation to obtain the desired sum. ◀

5.3.2 Productions over contexts

Now we focus on productions when the sets of positions $I_1, \ldots, I_n$ are intervals. For this purpose, we introduce the notion of *μ-k-context* as a generalization of μ-contexts from Definition 2.8. Pleasantly enough, this notion enables to abstract multisets, which are not especially easy to handle.

Definition 5.28 (Word-k-context, μ-k-context)

Let $k \geqslant 0$. Given tuples $(r_1, \dots, r_n) \in \mathsf{S}_k$ and $(v_0, u_1, v_1, \dots, u_n, v_n) \in (A^* \times A^+)^n \times A^*$, we say that they describe a *word-k-context* which is denoted $v_0 \lfloor u_1 \rfloor_{r_1} v_1 \cdots v_{n-1} \lfloor u_n \rfloor_{r_n} v_n$.

Let $\mu \colon A^* \to \mathbb{M}$ be a monoid morphism. Given tuples $(r_1, \dots, r_n) \in \mathsf{S}_k$ and $(m_0, u_1, m_1, \dots, u_n, m_n) \in (\mathbb{M} \times A^+)^n \times \mathbb{M}$, we say that they describe a *μ-k-context* which is denoted by the sequence $m_0 \lfloor u_1 \rfloor_{r_1} m_1 \cdots m_{n-1} \lfloor u_n \rfloor_{r_n} m_n$.

Remark 5.29 (Case $r_1 = \cdots = r_n = 1$)

In the particular case when $r_1 = \cdots = r_n = 1$, we simply write $v_0 \lfloor u_1 \rfloor v_1 \cdots v_{n-1} \lfloor u_n \rfloor v_n$ or $m_0 \lfloor u_1 \rfloor m_1 \cdots m_{n-1} \lfloor u_n \rfloor m_n$. Note that we must have $n = k$ in this case.

Observe that the concatenation of a μ-k-context and μ-k'-context gives a μ-$(k+k')$-context. Now us extend the definition of productions. Consider a word-k-context $v_0 \lfloor u_1 \rfloor_{r_1} \cdots \lfloor u_n \rfloor_{r_n} v_n$, and define $u := v_0 u_1 \cdots u_n v_n$ and I_j for $1 \leqslant j \leqslant n$ be the interval of positions of u_j inside u. Given a k-counting transducer $\mathscr{T}$, we define $\mathsf{prod}_{\mathscr{T}}(v_0 \lfloor u_1 \rfloor_{r_1} \cdots \lfloor u_n \rfloor_{r_n} v_n) := \mathsf{prod}^u_{\mathscr{T}}(\{\!\{ I_1 \ddagger r_1, \dots, I_n \ddagger r_n \}\!\})$.

In Proposition-Definition 2.9, we have shown that for two-way transducers, the production over a word-context $v_0 \lfloor u \rfloor v_1$ only depends on the μ-context $\mu(v_0) \lfloor u \rfloor \mu(v_1)$ when μ is the transition monoid of the machine. We provide a similar result for counting transducers in Proposition-Definition 5.31. It is first necessary to introduce in Definition 5.30 the notion of *transition morphism* for counting transducers. Even if such machines do not perform "transitions", we chose to keep this terminology. Recall that given $u \in A^*$, $u \ltimes 0$ denotes the word $(u[1], 0) \cdots (u[|u|], 0) \in (A \times \{0, 1\})^*$.

Definition 5.30 (Transition monoid, transition morphism)

Let $\mathscr{T} = (A, \mathbb{S}, (\delta_i, L_i)_{1 \leqslant i \leqslant n})$ be a k-counting transducer. Let $\varphi : (A \times \{0,1\}^k)^* \to \mathbb{M}$ be the product of the *syntactic morphisms*[8] of the regular languages $L_i \in \mathsf{RegProp}_k(A)$ for $1 \leqslant i \leqslant n$.

The *transition morphism* (resp. *transition monoid*) of $\mathscr{T}$ is defined as the morphism $\mu \colon A^* \to \mathbb{T}$ (resp. $\mathbb{T} \subseteq \mathbb{M}$) which maps u to $\varphi(u \ltimes 0 \cdots \ltimes 0)$, where $\mathbb{T}$ is chosen so that μ is surjective.

Observe that the transition morphism does not take marked letters into account. The main reason for this feature is that, in a word-k-context $v_0 \lfloor u_1 \rfloor_{r_1} \cdots \lfloor u_n \rfloor_{r_n} v_n$, marked letters are meant to be those of $u_1, \dots, u_n$. Now we are ready to state the analogue of Proposition-Definition 2.9.

Proposition-Definition 5.31 (Production in a k-context)

Let $k \geqslant 0$ and $\mathscr{T}$ be a k-counting transducer whose transition morphism is $\mu \colon A^* \to \mathbb{T}$. Let $m_0 \lfloor u_1 \rfloor_{r_1} \cdots \lfloor u_n \rfloor_{r_n} m_n$ be a μ-k-context, then for all word-k-context $v_0 \lfloor u_1 \rfloor_{r_1} \cdots \lfloor u_n \rfloor_{r_n} v_n$ such that $\mu(v_j) = m_j$ for all $1 \leqslant j \leqslant n$, the value $\mathsf{prod}_{\mathscr{T}}(v_0 \lfloor u_1 \rfloor_{r_1} \cdots \lfloor u_n \rfloor_{r_n} v_n)$ is the same.

We define $\mathsf{prod}_{\mathscr{T}}(m_0 \lfloor u_1 \rfloor_{r_1} \cdots \lfloor u_n \rfloor_{r_n} m_n)$ as this value.

Proof. Let $v_0 \lfloor u_1 \rfloor_{r_1} \cdots \lfloor u_n \rfloor_{r_n} v_n$ and $v'_0 \lfloor u_1 \rfloor_{r_1} \cdots \lfloor u_n \rfloor_{r_n} v'_n$ be word-k-contexts such that $\mu(v_j) = \mu(v'_j)$ for all $1 \leqslant j \leqslant n$. Let $u := v_0 u_1 \cdots u_n v_n$ (resp. $u' := v'_0 u_1 \cdots u_n v'_n$) and

[8] Using precisely syntactic morphisms is not useful: we only need to ensure that φ is a surjective monoid morphism which recognizes the languages L_i for $1 \leqslant i \leqslant n$. However, it enables to define "the" transition morphism in a unique fashion.

$I_1, \dots, I_n$ (resp. $I'_1, \dots, I'_n$) be the positions of $u_1, \dots, u_n$ in u (resp. in u'). Consider the unique monotone bijection $\sigma\colon \biguplus_{1 \leqslant j \leqslant n} I_j \to \biguplus_{1 \leqslant j \leqslant n} I'_j$. Then $(i_1, \dots, i_k) \mapsto (\sigma(i_1), \dots, \sigma(i_k))$ defines a bijection between $\mathsf{Tuples}_k(\{\!\{I_1 \ddagger r_1, \dots, I_n \ddagger r_n\}\!\})$ and $\mathsf{Tuples}_k(\{\!\{I'_1 \ddagger r_1, \dots, I'_n \ddagger r_n\}\!\})$.

Now let $(A, \mathbb{S}, (\delta_\ell, L_\ell)_{1 \leqslant \ell \leqslant m}) := \mathscr{T}$. For all $(i_1, \dots, i_k) \in \mathsf{Tuples}_k(\{\!\{I_1 \ddagger r_1, \dots, I_n \ddagger r_n\}\!\})$ and $1 \leqslant \ell \leqslant m$, we have $u \bullet i_1 \cdots \bullet i_k \bullet \in L_\ell$ if and only if $u' \bullet \sigma(i_1) \cdots \bullet \sigma(i_k) \bullet \in L_\ell$ by definition of the transition morphism of $\mathscr{T}$. The result follows. ◀

5.3.3 Iterators and pumping lemmas

When studying two-way transducers, we have introduced in Definition 2.10 the notion of μ-K-iterator as a μ-context whose word could be duplicated without breaking its structure. Such μ-K-iterators were then useful to device "pumping lemmas" for two-way transducers and study the asymptotic growth of their output. The goal of Section 5.3.3 is to generalize these properties to μ-k-contexts.

Definition 5.32 (Iterator)

Let $\mu\colon A^* \to \mathbb{M}$ be a monoid morphism and $k, K \geqslant 0$. Given $m_0, \dots, m_k, e_1, \dots, e_k \in \mathbb{M}$ and $u_1, \dots, u_k \in A^+$, we say that the μ-k-context $m_0 e_1 \boxed{\lfloor u_1 \rfloor} e_1 m_1 \cdots m_{k-1} e_k \boxed{\lfloor u_k \rfloor} e_k m_k$ is a *μ-(k, K)-iterator* if for all $1 \leqslant j \leqslant k$, $|u_j| \leqslant K$ and $e_j = \mu(u_j)$ is an idempotent.

As a first step towards a pumping lemma, we first describe in Claim 5.33 the shape of the production when a single word with idempotent image is iterated. Given $k \geqslant 0$ and $(r_1, \dots, r_n) \in \mathsf{S}_k$ we define its *abstract* $\mathsf{abs}(r_1, \dots, r_n)$ as the tuple obtained from $(r_1, \dots, r_n)$ by replacing the maximal blocks of shape $0, \dots, 0$ by a single 0. For instance $\mathsf{abs}(0,1,0,0,0,1,2) = (0,1,0,1,2) \in \mathsf{S}_4$. We define $\mathsf{A}_k := \{\mathsf{abs}(t) \mid t \in \mathsf{S}_k\}$ as the set of abstracts obtained from S_k. Observe that A_k is a finite subset of S_k. In the following, recall that $\mathbb{Z}[X]$ denotes the set of polynomials in X with coefficients in $\mathbb{Z}$.

Claim 5.34 (Iterating one idempotent)

For all $r \geqslant 0$ and for all $(s_1, \dots, s_n) \in \mathsf{A}_r$, there exists a polynomial $P_{(s_1, \dots, s_n)}(X) \in \mathbb{Z}[X]$ of degree at most r such that the following holds. Let $k \geqslant 0$ and $\mathscr{T}$ be a k-counting transducer whose transition morphism is $\mu\colon A^* \to \mathbb{T}$. Given $u \in A^+$ such that $e := \mu(u)$ is an idempotent of $\mathbb{T}$ and $\mathcal{L}, \mathcal{R}$ such that $\mathcal{L} \boxed{\lfloor u \rfloor}_r \mathcal{R}$ is a μ-k-context, we have for all $X \geqslant 2r{+}1$:

$$\mathsf{prod}_{\mathscr{T}}(\mathcal{L} \boxed{\lfloor u^X \rfloor}_r \mathcal{R}) = \sum_{(s_1, \dots, s_n) \in \mathsf{A}_r} \mathsf{prod}_{\mathscr{T}}(\mathcal{L} \boxed{\lfloor u \rfloor}_{s_1} \cdots \boxed{\lfloor u \rfloor}_{s_n} \mathcal{R}) \cdot P_{(s_1, \dots, s_n)}(X). \tag{5.34}$$

As a consequence, $\mathsf{prod}_{\mathscr{T}}(\mathcal{L} \boxed{\lfloor u^X \rfloor}_r \mathcal{R})$ is a polynomial in X of degree at most r. Furthermore, its coefficient in X is 0 if $r = 0$ and $\mathsf{prod}_{\mathscr{T}}(\mathcal{L}\, e \boxed{\lfloor u \rfloor} e\, \mathcal{R})$ if $r = 1$.

Proof. It follows from Claim 5.27 and Proposition-Definition 5.31 that for all $X \geqslant 0$:

$$\mathsf{prod}_{\mathscr{T}}(\mathcal{L} \boxed{\lfloor u^X \rfloor}_r \mathcal{R}) = \sum_{(r_1, \dots, r_X) \in \mathsf{S}_r} \mathsf{prod}_{\mathscr{T}}(\mathcal{L} \boxed{\lfloor u \rfloor}_{r_1} \cdots \boxed{\lfloor u \rfloor}_{r_X} \mathcal{R}).$$

Thanks to Claim 5.35, one can recombine various terms of this sum which are equal.

Claim 5.35 (Abstracts are a sufficient abstraction)

Let $(r_1, \dots, r_X) \in \mathsf{S}_r$ and $(s_1, \dots, s_n) := \mathsf{abs}(r_1, \dots, r_X)$, then:

$$\mathsf{prod}_{\mathscr{T}}(\mathcal{L} \lfloor u \rfloor_{r_1} \cdots \lfloor u \rfloor_{r_X} \mathcal{R}) = \mathsf{prod}_{\mathscr{T}}(\mathcal{L} \lfloor u \rfloor_{s_1} \cdots \lfloor u \rfloor_{s_n} \mathcal{R})$$

Proof idea. Transforming several consecutive $\lfloor u \rfloor_0$ in a single one corresponds to transforming the concatenation of several idempotents $e = \mu(u)$ in a single one. ◀

Therefore Equation (5.34) holds if we define for all $(s_1, \dots, s_n) \in \mathsf{A}_r$:

$$P_{(s_1,\dots,s_n)}(X) := |\{(r_1, \dots, r_X) \in \mathsf{S}_r \mid \mathsf{abs}(r_1, \dots, r_X) = (s_1, \dots, s_n)\}| .$$

Now we justify in Claim 5.36 that $P_{(s_1,\dots,s_n)}(X)$ is a polynomial.

Claim 5.36

For $X \geqslant 2r{+}1$, $P_{(s_1,\dots,s_n)}$ is a polynomial of $\mathbb{Z}[X]$ of degree at most[9] r.

Proof. If $s := (s_1, \dots, s_n) \in \mathsf{A}_r$, one has $n \leqslant 2r{+}1$ since there are no two consecutive 0. Let $0 \leqslant z \leqslant r{+}1$ be the number of zeros in s, then $P_s(X)$ is the number of tuples $(q_1, \dots, q_z) \in \mathsf{S}_{X-n}$ (it describes how many times we duplicate each 0). We show by induction on $z \geqslant 0$ that this value is a polynomial of degree at most $z{-}1$ whenever $X - n \geqslant 0$. ◀

As a consequence of Equation (5.34) and Claim 5.36, $\mathsf{prod}_{\mathscr{T}}(\mathcal{L} \lfloor u^X \rfloor_r \mathcal{R})$ is a polynomial in X for $X \geqslant 2r{+}1$. For $r = 0$, this polynomial is constant since the production is obviously constant. For $r = 1$, we $\mathsf{A}_1 = \{(0,1),(1,0),(0,1,0)\}$ and $P_{(0,1)}(X) = P_{(1,0)}(X) = 1$ and $P_{(0,1,0)}(X) = X - 2$ for $X \geqslant 3$. Hence the coefficient in X is $\mathsf{prod}_{\mathscr{T}}(\mathcal{L} \lfloor u \rfloor_0 \lfloor u \rfloor_1 \lfloor u \rfloor_0 \mathcal{R})$ which can be rewritten $\mathsf{prod}_{\mathscr{T}}(\mathcal{L}\, e \lfloor u \rfloor e\, \mathcal{R})$ by Proposition-Definition 5.31. ◀

We are ready to describe what happens when iterating k factors in μ-(k, K)-iterator. Lemma 5.37 gives an analogue of Claim 2.11. A similar result will be given in Claim 6.30. In the following, we denote by $\mathbb{Z}[X_1, \dots, X_k]$ the set of multivariate polynomials with coefficients in $\mathbb{Z}$.

Lemma 5.37 (Pumping iterators)

Let $k \geqslant 0$ and $f\colon A^* \to \mathbb{Z}$ be computed by a k-counting transducer $\mathscr{T}$ with transition morphism $\mu\colon A^* \to \mathbb{T}$. Let $K \geqslant 0$, $m_0 e_1 \lfloor u_1 \rfloor e_1 m_1 \cdots m_{k-1} e_k \lfloor u_k \rfloor e_k m_k$ be a μ-(k, K)-iterator and $v_0, \dots, v_k \in A^*$ be such that $\mu(v_j) = m_j$ for all $0 \leqslant j \leqslant k$.

Then the function $X_1, \dots, X_k \mapsto f(v_0 u_1^{X_1} v_1 \cdots v_{k-1} u_k^{X_k} v_k)$ is a polynomial of $\mathbb{Z}[X_1, \dots, X_k]$ of degree at most k, whenever $X_1, \dots, X_k \geqslant 2k{+}1$. Furthermore, the coefficient[10] of this polynomial in $X_1 \cdots X_k$ is $\mathsf{prod}(m_0 e_1 \lfloor u_1 \rfloor e_1 m_1 \cdots m_{k-1} e_k \lfloor u_k \rfloor e_k m_k)$.

Proof. Given $1 \leqslant \ell \leqslant k$, let $\beta(X_1, \dots, X_\ell) := v_0 u_1^{X_1} \cdots u_\ell^{X_\ell} v_\ell$. We show by induction on ℓ that if $(r, s) \in \mathsf{S}_k$ and $\mathcal{R}$ is μ-s-context, then $X_1, \dots, X_\ell \mapsto \mathsf{prod}_{\mathscr{T}}(\lfloor \beta(X_1, \dots, X_\ell) \rfloor_r \mathcal{R})$ is a polynomial of degree at most r for $X_1, \dots, X_\ell \geqslant 2k + 1$. Furthermore for $\ell > 0$, the coefficient of this polynomial in $X_1 \cdots X_\ell$ is 0 if $r < \ell$ and $\mathsf{prod}_{\mathscr{T}}(m_0 e_1 \lfloor u_1 \rfloor e_1 \cdots e_\ell \lfloor u_\ell \rfloor m_\ell\, \mathcal{R})$ if $r = \ell$.

[9] $P_{(s_1,\dots,s_n)}$ may not have degree r in general. For instance, $P_{(1,1,\dots,1,1)}(X) = 0$ for X large enough.

[10] The function $Q\colon X \mapsto f(v_0 u_1^X v_1 \cdots v_{k-1} u_k^X v_k)$ is also a polynomial for $X \geqslant 2k{+}1$. However, having several variables is necessary in our setting, since the coefficient of $Q(X)$ in X^k may not be $\mathsf{prod}(m_0 e_1 \lfloor u_1 \rfloor e_1 m_1 \cdots m_{k-1} e_k \lfloor u_k \rfloor e_k m_k)$. Indeed, if $f\colon a^n b^m \mapsto n^2 - n \times m$, then $f(a^X b^X) = 0$ for all $X \geqslant 0$.

The result is obvious for $\ell = 0$ (with $\beta() = v_0$) since the production is constant in this case. Let $\ell \geqslant 1$ and assume that the result holds for $\ell - 1$. We get by Claim 5.27:

$$\begin{aligned}\mathsf{prod}_{\mathscr{T}}(\lfloor \beta(X_1,\dots,X_\ell) \rfloor_r \mathcal{R}) &= \mathsf{prod}_{\mathscr{T}}(\lfloor \beta(X_1,\dots,X_{\ell-1}) u_\ell^{X_\ell} v_\ell \rfloor_r \mathcal{R}) \\ &= \sum_{(s_1,s_2,s_3)\in \mathsf{S}_r} \mathsf{prod}_{\mathscr{T}}(\lfloor \beta(X_1,\dots,X_{\ell-1}) \rfloor_{s_1} \lfloor u_\ell^{X_\ell} \rfloor_{s_2} \lfloor v_\ell \rfloor_{s_3} \mathcal{R}).\end{aligned}$$

Now we use Claim 5.33 to split the factor $u_\ell^{X_\ell}$ in several pieces. As a consequence, we get for all $X_\ell \geqslant 2r+1$ that the quantity $\mathsf{prod}_{\mathscr{T}}(\lfloor \beta(X_1,\dots,X_\ell) \rfloor_r \mathcal{R})$ equals:

$$\sum_{\substack{(r_1,r_2,r_3)\in \mathsf{S}_r \\ s=(s_1,\dots,s_n)\in \mathsf{A}_{r_2}}} P_s(X_\ell) \times \mathsf{prod}_{\mathscr{T}}(\lfloor \beta(X_1,\dots,X_{\ell-1}) \rfloor_{r_1} \lfloor u_\ell \rfloor_{s_1} \cdots \lfloor u_\ell \rfloor_{s_n} \lfloor \alpha_\ell \rfloor_{r_3} \mathcal{R}). \tag{5.38}$$

By induction hypothesis, each production which occurs in the sum of Equation (5.38) is a polynomial in $X_1, \dots, X_{\ell-1}$ when $X_1, \dots, X_{\ell-1} \geqslant 2k+1$. Therefore the function $X_1, \dots, X_\ell \mapsto \mathsf{prod}_{\mathscr{T}}(\lfloor \beta(X_1,\dots,X_\ell) \rfloor_r \mathcal{R})$ is a polynomial of degree at most r when $X_1, \dots, X_\ell \geqslant 2k+1$. It remains to study the coefficient in $X_1 \cdots X_\ell$ of this polynomial:

- ▶ if $r < \ell$, then by induction hypothesis the only terms in Equation (5.38) whose coefficients in $X_1 \cdots X_{\ell-1}$ is possibly non-zero are for $r_1 = r$ and $r_2 = r_3 = 0$, but then we have no X_ℓ since $r_2 = 0$. Hence the coefficient in $X_1 \cdots X_\ell$ is 0;
- ▶ if $r = \ell$, then by induction hypothesis three kinds of terms may have a non-zero coefficient:
  - ▶ $r_1 = \ell, r_2 = 0, r_3 = 0$ or $r_1 = \ell-1, r_2 = 0, r_3 = 1$, but then we also have no X_ℓ;
  - ▶ $r_1 = \ell-1$, $r_2 = 1$ and $r_3 = 0$. Then by using induction hypothesis and thanks to the last part of Claim 5.33, the coefficient in $X_1 \dots X_\ell$ has the desired shape. ◀

5.4 Factorization forests for counting transducers

In Chapter 3, the generic technique for optimizing blind pebble transducers and last pebble transducers was first to compute a μ-factorization forest of the input (where μ was the transition morphism), and then to use the structure of the forest in order to recompose the output with one less nested layer. We want to apply the same strategy for the case of counting transducers. To achieve this goal, Section 5.4 describes how factorization forests can be used to deal with the productions of counting transducers.

In Section 5.4.1 we lift the notion of production from words to μ-forests. Furthermore, we show how the function computed by a k-counting transducer can be decomposed as the sum of:

- ▶ a function $\mathsf{sum\text{-}dep}$, studied in Section 5.4.2, which is computable by $(k-1)$-counting transducer;
- ▶ a function $\mathsf{sum\text{-}ind}$, studied in Section 5.4.3. This function is related to the productions performed from the portions of the input which describe μ-(k, K)-iterators. As such, it can be seen as the term of “highest degree” of the original function, for which pumping lemmas can be applied.

5.4.1 Productions on multisets of nodes

Now, let us explain how to lift the notion of productions from multisets of positions to multisets of iterable nodes in a forest, thanks to the origins (recall Definition 2.28). Recall from Section 2.3 that $\mathsf{Forests}_\mu^{(d)}$ denotes the set of all μ-factorization forests (of height at most d).

Definition 5.39 (Production on a multiset of nodes)

Let $\mathscr{T}$ be a k-counting transducer and $\mu\colon A^* \to \mathbb{M}$ be a monoid morphism. Given $u \in A^+$, $\mathcal{F} \in \mathsf{Forests}_\mu(u)$, $\mathfrak{t}_1, \dots, \mathfrak{t}_n \in \mathsf{Iters}_\mathcal{F} \cup \{\mathcal{F}\}$ distinct[11] nodes and $(r_1, \dots, r_n) \in \mathsf{S}_k$, we define the *production* of $\mathscr{T}$ over $\{\!\{\mathfrak{t}_1 \ddagger r_1, \dots, \mathfrak{t}_n \ddagger r_n\}\!\}$ as follows:

$$\mathsf{prod}^{\mathcal{F}}_{\mathscr{T}}(\{\!\{\mathfrak{t}_1 \ddagger r_1, \dots, \mathfrak{t}_n \ddagger r_n\}\!\}) := \mathsf{prod}^{u}_{\mathscr{T}}(\{\!\{\mathsf{Fr}_\mathcal{F}(\mathfrak{t}_1) \ddagger r_1, \dots, \mathsf{Fr}_\mathcal{F}(\mathfrak{t}_n) \ddagger r_n\}\!\}).$$

We observe in Claim 5.40 that the output of $\mathscr{T}$ is obtained by ranging over all multisets of nodes. Recall from Lemma 2.33 that $\preccurlyeq$ denotes the total ordering of the nodes depending on the first position of their frontier. Recall that in $\{\!\{\mathfrak{t}_1, \dots, \mathfrak{t}_k\}\!\}$, the nodes $\mathfrak{t}_1, \dots, \mathfrak{t}_k$ are not assumed to be distinct.

Claim 5.40 (Decomposition of the output)

Let $f : A^* \to \mathbb{S}$ be the function computed by a k-counting transducer $\mathscr{T}$ and $\mu\colon A^* \to \mathbb{M}$ be a monoid morphism. For all $u \in A^+$ and $\mathcal{F} \in \mathsf{Forests}_\mu(u)$, we have:

$$f(u) = \sum_{\substack{(\mathfrak{t}_1, \dots, \mathfrak{t}_k) \in \mathsf{Iters}_\mathcal{F} \cup \{\mathcal{F}\} \\ \text{such that } \mathfrak{t}_1 \preccurlyeq \cdots \preccurlyeq \mathfrak{t}_k}} \mathsf{prod}^{\mathcal{F}}_{\mathscr{T}}(\{\!\{\mathfrak{t}_1, \dots, \mathfrak{t}_k\}\!\}).$$

Proof idea. We use Claim 2.31 to get disjoints sets, then we apply Claim 5.27. ◀

Now that we have defined productions over multisets of nodes, the main technique of Chapters 5 and 6 is to use the forest structure for deciding the properties of the function computed by $\mathscr{T}$. In Definition 2.32, we introduced the notion of dependent nodes, which roughly describes if two iterable nodes can be duplicated independently while preserving the forest structure. We shift this notion to multisets.

Definition 5.41 (Multiset dependance)

Let $\mathcal{F} \in \mathsf{Forests}_\mu$ and $\mathfrak{t}_1, \dots, \mathfrak{t}_k \in \mathsf{Nodes}_\mathcal{F}$. We say that the multiset $\{\!\{\mathfrak{t}_1, \dots, \mathfrak{t}_k\}\!\}$ is *independent* if the nodes $\mathfrak{t}_1, \dots, \mathfrak{t}_k$ are pairwise independent, and *dependent* otherwise.

In particular, if $\{\!\{\mathfrak{t}_1, \dots, \mathfrak{t}_k\}\!\}$ is independent, we have $\mathfrak{t}_i \neq \mathfrak{t}_j$ whenever $i \neq j$. As a consequence, any independent multiset $\{\!\{\mathfrak{t}_1, \dots, \mathfrak{t}_k\}\!\}$ can simply be written as a set $\{\mathfrak{t}_1, \dots, \mathfrak{t}_k\}$.

Given $\mathcal{F} \in \mathsf{Forests}_\mu$, we define the following sets of multisets:

- $\mathsf{Dep}^k_\mathcal{F} = \{\{\!\{\mathfrak{t}_1, \dots, \mathfrak{t}_k\}\!\} \text{ dependent} \mid \mathfrak{t}_1, \dots, \mathfrak{t}_k \in \mathsf{Iters}_\mathcal{F} \cup \{\mathcal{F}\}\}$;
- $\mathsf{Indep}^k_\mathcal{F} = \{\{\!\{\mathfrak{t}_1, \dots, \mathfrak{t}_k\}\!\} \text{ independent} \mid \mathfrak{t}_1, \dots, \mathfrak{t}_k \in \mathsf{Iters}_\mathcal{F} \cup \{\mathcal{F}\}\}$.

Now if $\mathscr{T}$ is a k-counting transducer with output $\mathbb{S}$ and transition monoid is $\mu : A^* \to \mathbb{T}$, we define the following functions of type $(A \cup \{\langle, \rangle\})^* \to \mathbb{S}$:

$$\mathsf{sum\text{-}dep}_\mathscr{T} : \mathcal{F} \mapsto \begin{cases} \displaystyle\sum_{M \in \mathsf{Dep}^k_\mathcal{F}} \mathsf{prod}^{\mathcal{F}}_{\mathscr{T}}(M) & \text{if } \mathcal{F} \in \mathsf{Forests}^{3|\mathbb{T}|}_\mu; \\ 0 & \text{otherwise.} \end{cases}$$

[11]Remark that if $\mathfrak{t}_1, \dots, \mathfrak{t}_n$ are distinct nodes from $\mathsf{Iters}_\mathcal{F} \cup \{\mathcal{F}\}$, then $\mathsf{Fr}_\mathcal{F}(\mathfrak{t}_1), \dots, \mathsf{Fr}_\mathcal{F}(\mathfrak{t}_n)$ are disjoint by Claim 2.31. Therefore Definition 5.39 makes sense with respect to Definition 5.26 which requires disjoint sets of positions.

and

$$\mathsf{sum\text{-}ind}_{\mathscr{T}} : \mathcal{F} \mapsto \begin{cases} \displaystyle\sum_{M \in \mathsf{Indep}^k_{\mathcal{F}}} \mathsf{prod}^{\mathcal{F}}_{\mathscr{T}}(M) & \text{if } \mathcal{F} \in \mathsf{Forests}^{3|\mathbb{T}|}_{\mu}; \\ 0 \quad \text{otherwise.} \end{cases}$$

which correspond respectively to the productions over dependent and independent multisets.

The bound $3|\mathbb{T}|$ may seem arbitrary, but recall from Theorem 2.21 that one can build a rational function $\mathsf{forest}_{\mu} \colon A^+ \to \mathsf{Forests}^{3|\mathbb{T}|}_{\mu}$ which computes a μ-forests of height at most $3|\mathbb{T}|$. As a consequence, one can recover the function computed by a counting transducer $\mathscr{T}$ thanks to $\mathsf{sum\text{-}dep}_{\mathscr{T}}$ and $\mathsf{sum\text{-}ind}_{\mathscr{T}}$, as explained in Proposition 5.42. This result is at the heart of the proofs of Chapters 5 and 6.

Proposition 5.42 (Decomposition of the output)

Let $f : A^* \to \mathbb{S}$ be the function computed by a k-counting transducer $\mathscr{T}$ whose transition monoid is $\mu : A^* \to \mathbb{M}$. Then $f = (\mathsf{sum\text{-}dep}_{\mathscr{T}} + \mathsf{sum\text{-}ind}_{\mathscr{T}}) \circ \mathsf{forest}_{\mu}$.

Proof idea. We use Claim 5.40 (multisets are either dependent or independent). ◀

We study the functions $\mathsf{sum\text{-}dep}_{\mathscr{T}}$ and $\mathsf{sum\text{-}ind}_{\mathscr{T}}$ separately in Sections 5.4.2 and 5.4.3. The main intuition is that if f were a polynomial, then $\mathsf{sum\text{-}ind}_{\mathscr{T}}$ would capture its terms of highest degrees.

5.4.2 Productions on dependent multisets

The goal of this section is to show in Lemma 5.43 that the function $\mathsf{sum\text{-}dep}_{\mathscr{T}}$ belongs to the class $\mathbb{S}\mathsf{poly}_{k-1}$. This is the main reason why solving membership problems in the current Chapter 5 and in Chapter 6 will be done by induction on $k \geqslant 1$: this way, the difficulties of the induction step will be condensed in the function $\mathsf{sum\text{-}ind}_{\mathscr{T}}$ (whose properties are studied in Section 5.4.3).

Lemma 5.43 (Productions on dependent multisets)

Let $\mathbb{S}$ be commutative[12] and $\mathscr{T}$ be a k-counting transducer with output in $\mathbb{S}$. One can build a $(k-1)$-counting transducer with output in $\mathbb{S}$ which computes the function $\mathsf{sum\text{-}dep}_{\mathscr{T}}$.

Proof. The main intuition for showing this result is that if a multiset of k nodes is dependent, then it has one less degree of freedom and therefore it can be described by using only $k-1$ nodes.

We assume that $\mathscr{T}$ has a single production pair, i.e. that it has shape $(A, \mathbb{S}, (\delta, L))$. For all $u \in A^+$ and $\mathcal{F} \in \mathsf{Forests}^{3|\mathbb{T}|}_{\mu}(u)$, observe that $\mathsf{sum\text{-}dep}_{\mathscr{T}}(\mathcal{F}) = \delta \cdot |S^{\mathcal{F}}|$ where:

$$\begin{aligned} S^{\mathcal{F}} := \Big\{ &(i_1, \dots, i_k) \in [1{:}|u|]^k \;\Big|\; u \bullet i_1 \bullet i_2 \cdots \bullet i_k \in L \\ &\text{and } \mathsf{origin}_{\mathcal{F}}(i_j), \mathsf{origin}_{\mathcal{F}}(i_{j'}) \text{ are dependent for some } j \neq j' \Big\}. \end{aligned}$$

Given $(i_1, \dots, i_k) \in S^{\mathcal{F}}$, we let $1 \leqslant j \leqslant k$ be the smallest index such that $\mathsf{origin}_{\mathcal{F}}(i_{j'})$ observes $\mathsf{origin}_{\mathcal{F}}(i_j)$ for some $j' \neq j$. We let $\sigma^{\mathcal{F}}((i_1, \dots, i_k)) := (i_1, \dots, i_{j-1}, i_{j+1}, \dots, i_k)$ be the tuple where i_j is removed. The function $\sigma^{\mathcal{F}}$ has type $S^{\mathcal{F}} \to [1{:}|u|]^{k-1}$.

Since we consider forests of bounded height, we first note in Claim 5.44 that the size of the pre-image of some tuple under $\sigma^{\mathcal{F}}$ has to be bounded. In other words, $\sigma^{\mathcal{F}}$ enables to reduce the dimension of the tuples, up to regrouping them into clusters of bounded size.

[12] We do not need to assume that $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$ in this proof.

Claim 5.44 (Bounded pre-images)

There exists $N \geqslant 0$ such that for all $u \in A^+, \mathcal{F} \in \mathsf{Forests}_\mu^{3|\mathbb{T}|}(u)$ and $1 \leqslant i_1, \dots, i_{k-1} \leqslant |u|$, the size of the set $(\sigma^\mathcal{F})^{-1}((i_1, \dots, i_{k-1})) \subseteq S^\mathcal{F}$ is less than N.

Proof. If $(i'_1, \dots, i'_k) \in (\sigma^\mathcal{F})^{-1}((i_1, \dots, i_{k-1}))$, it means that there exist $1 \leqslant j \leqslant k$ and $1 \leqslant p \leqslant k-1$ such that $\mathsf{origin}_\mathcal{F}(i_p)$ observes $\mathsf{origin}_\mathcal{F}(i'_j)$. There is only a bounded number of such nodes since $\mathcal{F}$ has bounded height and thanks to Claim 2.31. ◀

We justify in Claim 5.45 that regular languages can detect the size of the pre-images under $\sigma^\mathcal{F}$.

Claim 5.45 (Regular pre-images)

For all $0 \leqslant n \leqslant N$, one can build $L_n \in \mathsf{RegProp}_{k-1}(A \cup \{\langle, \rangle\})$ such that the following holds for all $u \in A^+$ and $\mathcal{F} \in \mathsf{Forests}_\mu^{3|\mathbb{T}|}(u)$: we have $\mathcal{F}\bullet\ell_1\bullet\ell_2 \cdots \bullet\ell_{k-1} \in L_n$ if and only if the positions $1 \leqslant \ell_1, \dots, \ell_{k-1} \leqslant |\mathcal{F}|$ correspond to leaves of $\mathcal{F}$ which encode positions $1 \leqslant i_1, \dots, i_{k-1} \leqslant |u|$ of the word u such that $|(\sigma^\mathcal{F})^{-1}((i_1, \dots, i_{k-1}))| = n$.

Proof idea. As shown in Claim 3.22, one can first build a regular language which detects whether the marked positions $1 \leqslant \ell_1, \dots, \ell_{k-1} \leqslant |\mathcal{F}|$ are leaves of $\mathcal{F}$. Assume that they account for positions $1 \leqslant i_1, \dots, i_{k-1} \leqslant |u|$, the additional idea is to count the number of positions $1 \leqslant i \leqslant |u|$ and $1 \leqslant p \leqslant k-1$ such that $\sigma^\mathcal{F}(i_1, \dots, i_p, i, i_p \dots, i_{k-1}) = (i_1, \dots, i_{k-1})$. Since there is only a bounded number of such candidate positions to check (recall the proof of Claim 5.44 and Claim 2.31), one can build a regular language which counts them and determines whether $|(\sigma^\mathcal{F})^{-1}((i_1, \dots, i_{k-1}))| = n$ holds. ◀

We conclude by observing that $|S^\mathcal{F}| = \sum_{n=0}^{N} n \times \#L_n(\mathcal{F})$ for all $\mathcal{F} \in \mathsf{Forests}_\mu^{3|\mathbb{T}|}$. ◀

5.4.3 Productions on independent multisets

Recall from Proposition 5.42 that the study of the function computed by a k-counting transducer $\mathscr{T}$ resumes to the study of the functions $\mathsf{sum\text{-}dep}_\mathscr{T}$ and $\mathsf{sum\text{-}ind}_\mathscr{T}$. We have shown in Section 5.4.3 that $\mathsf{sum\text{-}dep}_\mathscr{T}$ belongs to $\mathbb{S}\mathsf{poly}_{k-1}$. Now we focus on $\mathsf{sum\text{-}ind}_\mathscr{T}$, i.e. the productions on independent sets[13].

We first describe the concept of *linearization* of a (multi)set from $\mathsf{Indep}^k_\mathcal{F}$. It aims at abstracting the frontiers of its nodes as a μ-k-context. Roughly, it consists in replacing all letters which are not in some frontier by their image under the monoid morphism. This notion originates from [Dou22, Section E].

Definition 5.46 (Linearization)

Let $\mu : A^* \to \mathbb{M}$ be a morphism into a finite monoid, $u \in A^+$ and $\mathcal{F} \in \mathsf{Forests}_\mu(u)$. Given a set $M \in \mathsf{Indep}^k_\mathcal{F}$, we define its *linearization* by induction on the forest structure:

- if $|M| = 0$, then $\mathsf{lin}_\mathcal{F}(M) := \mu(u)$;
- if $M = \{\mathcal{F}\}$ then $\mathsf{lin}_\mathcal{F}(M) := \lfloor u[\mathsf{Fr}_\mathcal{F}(\mathcal{F})] \rfloor$;
- otherwise $\mathcal{F} = \langle \mathcal{F}_1 \rangle \cdots \langle \mathcal{F}_n \rangle$ and $\mathcal{F} \notin M$. For $1 \leqslant j \leqslant n$ we let[14] $M_j := M \cap \mathsf{Nodes}_{\mathcal{F}_j}$ and we define the concatenation $\mathsf{lin}_\mathcal{F}(M) := \mathsf{lin}_{\mathcal{F}_1}(M_1) \cdots \mathsf{lin}_{\mathcal{F}_n}(M_n)$.

[13] Recall that independent multisets are sets since all their elements are disjoint.

[14] Observe that $M_j \in \mathsf{Indep}^{|M_k|}_{\mathcal{F}_j}$ for all $1 \leqslant j \leqslant n$.

Example 5.47 (Linearization)

The linearization of the singleton set containing the topmost blue node in Figure 2.26 (recall that here $\mathbb{M} = (\{-1, 1, 0\}, \times)$) is $(-1)\times(-1)\times 0 \lfloor(-1)00\rfloor 0 = 0 \lfloor(-1)00\rfloor 0$.

Now we show in Lemma 5.48 that the linearization of an independent set is a μ-(k, K)-iterator whose production is the same as the production on the original set of nodes. The intuition is that since we remove letters which are "in the middle of idempotents", the behavior of $\mathscr{T}$ will not be modified.

Lemma 5.48 (Productions only depend on linearizations)

Let $\mathscr{T}$ be a k-counting transducer with transition morphism $\mu : A^* \to \mathbb{T}$. Let $K \geqslant 0$, $u \in A^+$, $\mathcal{F} \in \mathsf{Forests}^K_\mu(u)$ and $M \in \mathsf{Indep}^k_{\mathcal{F}}$. Then $\mathsf{prod}^{\mathcal{F}}_{\mathscr{T}}(M) = \mathsf{prod}_{\mathscr{T}}(\mathsf{lin}_{\mathcal{F}}(M))$.

Furthermore, if $M = \{\mathfrak{t}_1, \dots, \mathfrak{t}_k\}$ with $\mathfrak{t}_1 \preccurlyeq \cdots \preccurlyeq \mathfrak{t}_k$, then $\mathsf{lin}_{\mathcal{F}}(M)$ is a μ-$(k, 2^K)$-iterator of shape $m_0 e_1 \lfloor u_1 \rfloor e_1 m_1 \cdots m_{k-1} e_k \lfloor u_k \rfloor e_k m_k$ such that $\mu(u) = m_0 e_1 m_1 \cdots m_{k-1} e_k m_k$ and for all $1 \leqslant j \leqslant k$, $\mu(u_j) = e_j$ and $\mathsf{lin}_{\mathcal{F}}(\{\mathfrak{t}_j\}) = m_0 e_1 m_1 \cdots m_j e_j \lfloor u_j \rfloor e_j m_{j+1} \cdots e_k m_k$.

Proof sketch. Let us first focus on the case $M = \mathcal{F}$. We have $\mathsf{lin}_{\mathcal{F}}(M) = \lfloor u[\mathsf{Fr}_{\mathcal{F}}(\mathcal{F})] \rfloor$ and one has to justify that removing the letters which are not in $\mathsf{Fr}_{\mathcal{F}}(\mathcal{F})$ will not affect the production performed by a counting transducer. For this, we show in Claim 5.49 that the letters of $\mathsf{Fr}_{\mathcal{F}}(\mathcal{F})$ have the same environment in u and in $u[\mathsf{Fr}_{\mathcal{F}}(\mathcal{F})]$, which is formalized using μ-1-contexts.

Claim 5.49 (Context along a frontier)

Let $u \in A^+$, $\mathcal{F} \in \mathsf{Forests}_\mu(u)$, $\mathfrak{t} \in \mathsf{Nodes}_{\mathcal{F}}$, $I := \mathsf{Fr}_{\mathcal{F}}(\mathcal{F})$, $u' := u[I]$, and $\sigma : I \to [1{:}|u'|]$ be the unique monotone bijection. Then for all $i \in I$ the following μ-1-contexts are equal:

$$\mu(u[1{:}i{-}1]) \lfloor u[i] \rfloor \mu(u[i{+}1{:}|u|]) = \mu(u'[1{:}\sigma(i){-}1]) \lfloor u'[\sigma(i)] \rfloor \mu(u'[\sigma(i){+}1{:}|u'|]).$$

As an immediate consequence of this statement, we have $\mu(u) = \mu(u')$.

Proof of Claim 5.49. We show the result by induction on $\mathcal{F}$. It is obvious if $\mathcal{F} = a \in A$. Now, assume that $\mathcal{F} = \langle \mathcal{F}_1 \rangle \cdots \langle \mathcal{F}_n \rangle$ where $u = u_1 \cdots u_n$ is the according factorization. Then $u' = u'_1 u'_n$ where $u'_j := u_j[\mathsf{Fr}_{\mathcal{F}_j}(\mathcal{F}_j)]$ for $j \in \{1, n\}$. We only treat the case $n \geqslant 3$, i.e. when $\mu(u_1) = \cdots = \mu(u_n)$ is an idempotent. For $i \in I_1 := \mathsf{Fr}_{\mathcal{F}}(\mathcal{F}_1) = I \cap [1{:}|u_1|]$, we have $1 \leqslant \sigma(i) \leqslant |u'_1|$ and $\sigma|_{I_1} : I_1 \to [1, |u'_1|]$ is the monotone bijection. Then:

$$\begin{aligned}
&\mu(u[1{:}i{-}1]) \lfloor u[i] \rfloor \mu(u[i{+}1{:}|w|]) \\
&= \mu(u[1{:}i{-}1]) \lfloor u_1[i] \rfloor \mu(u_1[i{+}1{:}|u_1|]) \mu(u_2) \cdots \mu(u_n) \\
&= \mu(u_1[1{:}i{-}1]) \lfloor u_1[i] \rfloor \mu(u_1[i{+}1{:}|u_1|]) \mu(u_n) && \text{since } \mu(u_n) \text{ is idempotent;} \\
&= \mu(u'_1[1{:}\sigma(i){-}1]) \lfloor u'_1[\sigma(i)] \rfloor \mu(u'_1[\sigma(i){+}1{:}|u'_1|]) \mu(u'_n) && \text{by induction hypothesis;} \\
&= \mu(u'[1{:}\sigma(i){-}1]) \lfloor u'[\sigma(i)] \rfloor \mu(u'[\sigma(i){+}1{:}|u'|]).
\end{aligned}$$

The case of $i \in \mathsf{Fr}_{\mathcal{F}}(\mathcal{F}_n)$ is symmetrical. The case $n \leqslant 2$ is similar and easier. ◀

It is easy to deduce from Claim 5.49 that the result holds for $k = 1$ when $M = \{\mathcal{F}\}$. The bound 2^K follows from the fact that skeletons are binary trees of height at most K. The generalization of the result to arbitrary $k \geqslant 1$ and $M \in \mathsf{Indep}^k_{\mathcal{F}}$ follows from the same arguments as those of Lemma 2.33. The reader is invited to look back at Figure 2.34 which provides the desired μ-$(k, 2^K)$-iterator structure and justifies that Claim 5.49 can be applied[15] to each node. ◀

As a consequence of Lemma 5.48, the function $\mathsf{sum\text{-}ind}_{\mathscr{T}}$ deals with sets of positions which describe μ-$(k, 2^{3|\mathbb{T}|})$-iterators. Therefore, we shall be able to apply "pumping" technologies (e.g. Lemma 5.37) in order to show that this function enjoys good properties when solving a membership problem.

5.5 Solving the optimization problem for counting transducers

This section is devoted to showing Theorem 5.25 (it will follow from Theorem 5.54). The main idea is to follow a proof strategy which is similar to that of Sections 3.2 and 3.3.

We first give a necessary condition, named *pumpability*, for a k-counting transducer to compute a function f such that $|f(u)| = \mathcal{O}(|u|^{k-1})$. If it does not hold, the function cannot be computed by a $(k-1)$-counting transducer. Definition 5.50 can be seen as an analogue of Definitions 3.17 and 3.25.

Definition 5.50 (Pumpable counting transducer)

Let $k \geqslant 1$ and $\mathscr{T}$ be a k-counting transducer with transition morphism $\mu\colon A^* \to \mathbb{T}$. We say that $\mathscr{L}$ is *pumpable* if there exists a μ-$(k, 2^{3|\mathbb{T}|})$-iterator $m_0 e_1 \lfloor u_1 \rfloor e_1 \cdots e_k \lfloor u_k \rfloor e_k m_k$ such that $\mathsf{prod}_{\mathscr{T}}(m_0 e_1 \lfloor u_1 \rfloor e_1 \cdots e_k \lfloor u_k \rfloor e_k m_k) \neq 0$.

Observe that pumpability is decidable, since it merely consists in checking that the production of $\mathscr{T}$ is nul on a finite number of μ-k-contexts. Recall that the analogue pumpability notions of Sections 3.2 and 3.3 were decidable as well, for the same reasons.

Now we show in Lemma 5.51 that pumpability is a sufficient condition for having asymptotic growth in $\theta(|u|^k)$. This result crucially relies on Lemma 5.37 and is an analogue of Claims 3.19 and 3.27. However, its proof is slightly more subtle since one needs to ensure that no compensations can occur in $\mathbb{Z}$.

Lemma 5.51 (Pumpability $\Rightarrow$ Growth)

Let $k \geqslant 1$ and $f\colon A^* \to \mathbb{Z}$ be computed by a k-counting transducer which is pumpable. There exists $v_0, \ldots, v_k \in A^*$, $u_1, \ldots, u_k \in A^+$, such that $|f(v_0 u_1^X \cdots u_k^X v_k)| = \theta(X^k)$.

Proof. Let $\mathscr{T}$ be a k-counting transducer which is pumpable. There exists a μ-(k, K)-iterator $m_0 e_1 \lfloor w_1 \rfloor e_1 \cdots e_k \lfloor w_k \rfloor e_k m_k$ such that $\mathsf{prod}_{\mathscr{T}}(m_0 e_1 \lfloor w_1 \rfloor e_1 \cdots e_k \lfloor w_k \rfloor e_k m_k) \neq 0$. Since μ is surjective, one can find $v_0, \ldots, v_k \in A^*$ such that such that $\mu(v_j) = m_j$ for all $0 \leqslant j \leqslant k$. It follows from Lemma 5.37 that $X_1, \ldots, X_k \mapsto f(v_0 w_1^{X_1} v_1 \cdots v_{k-1} w_k^{X_k} v_k)$ is a polynomial of $\mathbb{Z}[X_1, \ldots, X_k]$ of degree at most k for $X_1, \ldots, X_k$ large enough. Furthermore, the coefficient of this polynomial in $X_1 \cdots X_k$ is $\alpha := \mathsf{prod}(m_0 e_1 \lfloor w_1 \rfloor e_1 m_1 \cdots m_{k-1} e_k \lfloor w_k \rfloor e_k m_k) \neq 0$.

We then rely on the following classical result for multivariate polynomials.

Claim 5.52 (Multivariate polynomials)

Let $P \in \mathbb{Q}[X_1, \ldots, X_k]$ be a polynomial of degree exactly ℓ. There exists $N_1, \ldots, N_k \geqslant 1$ such that $|P(N_1 X, \ldots, N_k X)| = \theta(X^\ell)$ when $X \to +\infty$.

[15] Having an independent multiset is crucial here. For instance, in the extreme opposite case where some nodes of M would be the same, then Claim 5.49 could not be applied to show that the production is preserved.

Proof. For all $N_1, \dots, N_k \geqslant 0$, $P_{N_1,\dots,N_k} : X \mapsto P(N_1 X, \dots, N_k X)$ is a polynomial in X of degree at most ℓ. Let $C(N_1, \dots, N_k)$ be the coefficient in X^ℓ of $P_{N_1,\dots,N_k}$, then C is a non-null multivariate polynomial (since P has a term of degree ℓ which is not null). Therefore (this is a classical algebraic argument) there exist $N_1, \dots, N_k \geqslant 1$ such that $C(N_1, \dots, N_k) \neq 0$. For this tuple, $P_{N_1,\dots,N_k}$ has degree exactly ℓ. Thus $|P_{N_1,\dots,N_k}(X)| = \theta(X^\ell)$. ◀

Let $N_1, \dots, N_k$ be given by Claim 5.52 for the polynomial P which has degree k. This result gives $|f(v_0 w_1^{N_1 X} v_1 \cdots v_{k-1} w_k^{N_k X} v_k)| = \theta(X^k)$. Thus we let $u_j := w_j^{N_j}$ for all $1 \leqslant j \leqslant k$. ◀

We also give an analogue of Lemmas 3.23 and 3.34. Lemma 5.53 shows that if $\mathscr{T}$ is not pumpable, then all the terms which define the function $\text{sum-ind}_{\mathscr{T}}$ have to be null.

Lemma 5.53 (Key lemma for removing one layer)

Let $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$ and $k \geqslant 1$. Given a function $f : A^* \to \mathbb{S}$ computed by a k-counting transducer $\mathscr{T}$ which is not pumpable and whose transition morphism is $\mu : A^* \to \mathbb{T}$, we have $\text{sum-ind}_{\mathscr{T}} = 0$ and therefore $f = \text{sum-dep}_{\mathscr{T}} \circ \text{forest}_\mu$ where $\text{sum-dep}_{\mathscr{T}} \in \mathbb{S}\text{poly}_{k-1}$.

Proof. Thanks to Lemma 5.43 and Proposition 5.42, we only need to show that $\text{sum-ind}_{\mathscr{T}} = 0$. We show a stronger result: all the terms which define this function are null when $\mathscr{T}$ is not pumpable. Indeed, assume by contradiction that $\text{prod}_{\mathscr{T}}^{\mathcal{F}}(M) \neq 0$ for some $\mathcal{F} \in \text{Forests}_\mu^{3|\mathbb{T}|}$ and (multi)set of nodes $M \in \text{Indep}_{\mathcal{F}}^k$. It follows from Lemma 5.48 that $\text{prod}_{\mathscr{T}}^{\mathcal{F}}(M) = \text{prod}_{\mathscr{T}}^{\mathcal{F}}(\text{lin}_{\mathcal{F}}(M))$ and that $\text{lin}_{\mathcal{F}}(M)$ is a μ-$(k, 2^{3|\mathbb{T}|})$-iterator. Therefore $\mathscr{T}$ should be pumpable. ◀

Now we are ready for the proof of Theorem 5.54, which is a refinement of Theorem 5.25. As mentioned above, the proof strategy is similar to that of Sections 3.2.2 and 3.3.2: we use the precomputation of a factorization forest in order to produce the same output while using one less layer.

Theorem 5.54 (Removing one counting layer)

Let $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$. Let $k \geqslant 1$ and $f\colon A^* \to \mathbb{S}$ be a function computed by a k-counting transducer $\mathscr{T}$ whose transition morphism is $\mu\colon A^* \to \mathbb{T}$. The following conditions are equivalent:

(1) $|f(u)| = \mathcal{O}(|u|^{k-1})$;
(2) for all $v_0, \dots, v_k \in A^*$, $u_1, \dots, u_k \in A^*$, $|f(v_0 u_1^X v_1 \cdots v_{k-1} u_k{}^X v_k)| = \mathcal{O}(X^{k-1})$;
(3) for all $v_0, \dots, v_k \in A^*$, $u_1, \dots, u_k \in A^*$, $|f(v_0 u_1^X v_1 \cdots v_{k-1} u_k{}^X v_k)| \neq \theta(X^k)$;
(4) $\mathscr{T}$ is not pumpable;
(5) $f \in \mathbb{S}\text{poly}_{k-1}$ (it can be computed by a $(k-1)$-counting transducer).

Furthermore, this property is decidable and the construction is effective.

Proof. Item (5) ⇒ Item (1) ⇒ Item (2) ⇒ Item (3) are obvious and Item (3) ⇒ Item (4) is Lemma 5.51. Let us show Item (4) ⇒ Item (5). It follows from Proposition 5.42 that $f = \text{sum-dep}_{\mathscr{T}} \circ \text{forest}_\mu$ where $\text{sum-dep}_{\mathscr{T}} \in \mathbb{S}\text{poly}_{k-1}$. Therefore, $f \in \mathbb{S}\text{poly}_{k-1}$ thanks to Proposition 5.7 which enables to pre-compose by the regular function forest_μ from Theorem 2.21. Decidability follows from the fact that pumpability is decidable, as observed right after Definition 5.50. ◀

As a consequence of Item (3) in Theorem 5.54, if $f \in \mathbb{S}\text{poly}_k \smallsetminus \mathbb{S}\text{poly}_{k-1}$, there exist $v_0, \dots, v_k \in A^*$, $u_1, \dots, u_k \in A^+$ such that $|f(v_0 u_1^X v_1 \cdots v_{k-1} u_k{}^X v_k)| = \theta(X^k)$. In other words, one can find some patterns which witness the asymptotic growth of the function when iterated.

5.6 Discussion: from $\mathbb{Z}$-polyregular to $\mathbb{N}$-polyregular

We have shown that the class membership problems about $\mathbb{S}$-rational series and $\mathbb{S}$-polyregular functions for $\mathbb{S} = \mathbb{Z}$ or $\mathbb{N}$ could be solved. In this section, we briefly discuss Open question 5.55.

Open question 5.55 (From $\mathbb{Z}$-polyregular to $\mathbb{N}$-polyregular)

Given a $\mathbb{Z}$-polyregular function, can we decide if it is $\mathbb{N}$-polyregular?

The author is not aware of an answer to this question in the literature, even when considering rational series instead of polyregular functions. However, it is well-known that non-negativity does not characterize[16] $\mathbb{N}$-polyregular functions within the $\mathbb{Z}$-polyregular ones, as recalled in Example 5.56.

Example 5.56 (Non-negativity does not characterize $\mathbb{N}$-polyregular)

If $f: A^* \to \mathbb{N}$ is $\mathbb{N}$-polyregular, then $f^{-1}(\{\delta\})$ is a regular language for all $\delta \in \mathbb{N}$ (this result follows from Proposition 1.41). The $\mathbb{Z}$-polyregular function $g: u \mapsto (|u|_a - |u|_b)^2$ for $a \neq b \in A$ is not $\mathbb{N}$-polyregular since $g^{-1}(\{0\}) = \{u \in A^* \mid |u|_a = |u|_b\}$. However $g(A^*) \subseteq \mathbb{N}$.

Now we discuss the particular case of unary inputs. In this setting, $\mathbb{N}$-polyregularity coincides with non-negativity. Theorem 5.57 can be found e.g. in [BR11, Proposition 2.1 p 137].

Theorem 5.57 (Non-negativity = $\mathbb{N}$-polyregularity)

Let $A = \{a\}$ be an alphabet which contains a single letter. A $\mathbb{Z}$-polyregular function $f: A^* \to \mathbb{N}$ is $\mathbb{N}$-polyregular if and only if $f(A^*) \subseteq \mathbb{N}$.

However, Theorem 5.57 cannot be extended to $\mathbb{N}$-rational series with unary input alphabet. Indeed, the function $a^n \mapsto 3^n + (-2)^n$ is $\mathbb{Z}$-rational and nonnegative but not $\mathbb{N}$-rational. This argument hints that solving Open question 5.55 is probably simpler than obtaining a result for rational series.

[16] This result implies that $\mathbb{Z}$ is not a *Fatou extension* of $\mathbb{N}$, see e.g. [BR11, Chapter 7].

Chapter 6

Polyblind functions with commutative output

LE SOUCI
Aveugle, l'homme l'est tout au long de sa vie.
Toi, deviens-le, Faust, à la fin.

FAUST, *aveugle*
La nuit semble s'accroître et se fait plus profonde ;
Mais au dedans, mon cœur rayonne de clarté
Et ce que j'ai conçu doit être exécuté.

Johann Wolfgang von Goethe, *Faust II (Hélène)*
(trad. J. Malaplate)

We have shown in Chapter 5 that the classes of functions computed by pebble transducers and marble transducers (and thus by last pebble transducers) with output in a commutative monoid $\mathbb{S}$ are the same. Such functions were said to be $\mathbb{S}$-polyregular. Furthermore, for $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$, we have shown that the number of layers required to compute a function can be optimized. The goal of Chapter 6 is to study the class of functions computed by blind pebble transducers with output in $\mathbb{S}$, named *$\mathbb{S}$-polyblind functions*. In particular, we show that one can decide if an $\mathbb{S}$-polyregular function is $\mathbb{S}$-polyblind for $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$.

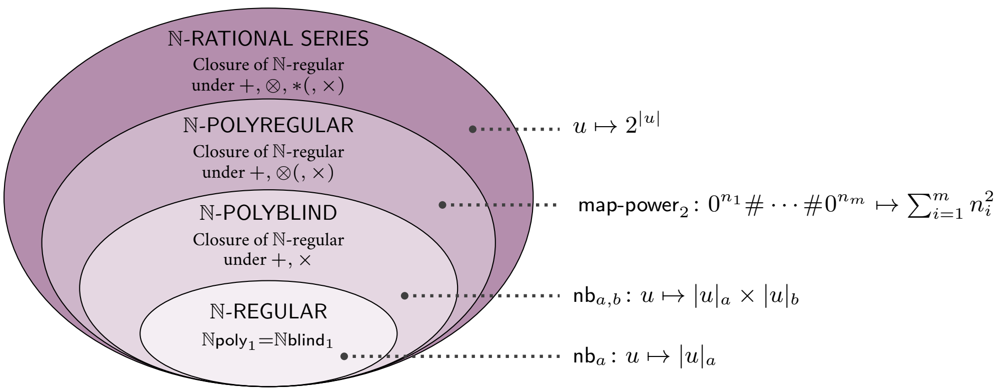


Figure 6.1: Classes of functions studied in Chapter 6 for $\mathbb{S} = \mathbb{N}$.

In Section 6.1 we introduce the class of $\mathbb{S}$-polyblind functions and claim that it is captured by *blind counting transducers*, which are a simple variant of the counting transducers from Chapter 5. The main difference is that counting transducers can check regular properties of tuples of positions, while blind counting transducers can only check properties of positions. Furthermore, we explain for $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$ how $\mathbb{S}$-polyblind functions can be described as a subclass of $\mathbb{S}$-rational series, in a similar fashion to $\mathbb{S}$-polyregular functions. This characterization is presented in Figure 6.1 for $\mathbb{S} := \mathbb{N}$.

The goal of Section 6.2 is to state the main result of Chapter 6, that is the decidability of the class membership problem from $\mathbb{S}$-polyregular to $\mathbb{S}$-polyblind for $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$. Intuitively, it provides a way to simplify a program with "for" loops by making its nested loop indices independent. For the first time in this manuscript (and contrary to the proofs of Chapters 3 to 5), it is no longer possible to use the asymptotic growth of the functions to discriminate between the classes. Indeed, both $\mathbb{S}$-polyregular to $\mathbb{S}$-polyblind may have polynomial growth. Therefore, we introduce a new semantic condition named *repetitiveness* and show that it characterizes $\mathbb{S}$-polyblind functions among the $\mathbb{S}$-polyregular ones. This result has several low hanging consequences. In particular, it enables to easily build separating examples between the two classes (see Figure 6.1). Furthermore, it yields an optimization result for $\mathbb{S}$-polyblind functions, which is similar to the result of Chapter 5 for $\mathbb{S}$-polyregular functions.

The proof of the membership result from $\mathbb{S}$-polyregular to $\mathbb{S}$-polyblind is rather involved and goes over Sections 6.3 to 6.5. It is built upon the tools introduced in Chapters 2 and 5 and crucially relies on the use of factorization forests to decompose the output of counting transducers. The main idea is to perform an induction, while insulating during the induction steps the terms of "highest degree" of the function. Interestingly, the semantic characterization of $\mathbb{S}$-polyblind functions thanks to repetitiveness is not only a consequence of this proof, but also a key technical tool for the induction step.

The contributions presented in this chapter are based on the results of [Dou21, Dou22], which focus on $\mathbb{N}$-polyregular functions. We observe that the proof also works for $\mathbb{Z}$-polyregular functions.

6.1 Polyblind functions with commutative output

We first introduce the class of *polyblind functions* which have output in a commutative monoid $\mathbb{S}$. Sections 6.1.1 and 6.1.2 can be seen as analogues of Section 5.1 for commutative outputs. In Section 6.1.3, we then connect this class of functions to $\mathbb{S}$-rational series for $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$.

6.1.1 Blind pebble transducers with commutative output

Let $(\mathbb{S}, +)$ be a (possibly infinite) commutative monoid. We define the class of *$\mathbb{S}$-polyblind functions* as functions of type $A^* \to \mathbb{S}$ where A is a finite alphabet. Definition 6.2 is the analogue of Definition 5.2.

Definition 6.2 ($\mathbb{S}$-polyblind functions)

The class of *$\mathbb{S}$-polyblind functions* is the class of functions of shape $\mathsf{sum} \circ g\colon A^* \to \mathbb{S}$ where $g\colon A^* \to \mathbb{S}^*$ is polyblind[1] (recall that $\mathsf{sum}\colon \mathbb{S}^* \to \mathbb{S}$ is the sum operation in $\mathbb{S}$).

Observe that $\mathbb{N}$-polyblind functions exactly capture the functions $f\colon A^* \to \mathbb{N}$ such that the function $g\colon A^* \to \{1\}^*, u \mapsto 1^{f(u)}$ is polyblind. We denote by $\mathbb{S}\mathsf{blind}$ the class of $\mathbb{S}$-polyblind functions. More precisely, for all $k \geqslant 1$, we denote by $\mathbb{S}\mathsf{blind}_k$ the class of functions of shape $\mathsf{sum} \circ g\colon A^* \to \mathbb{S}$ where the function $g\colon A^* \to \mathbb{S}^*$ is computed by a blind k-pebble transducer. We let $\mathbb{S}\mathsf{blind}_0 := \mathbb{S}\mathsf{poly}_0$. Note

[1] As for Definition 5.2, we consider in fact polyblind functions of type $A^* \to F^*$ for F a finite subset of $\mathbb{S}$.

that $\mathbb{S}\mathsf{blind}_1 = \mathbb{S}\mathsf{poly}_1$ and that $\mathbb{S}\mathsf{blind}_k \subseteq \mathbb{S}\mathsf{blind}_{k+1}$ and $\mathbb{S}\mathsf{blind}_k \subseteq \mathbb{S}\mathsf{poly}_k$ for all $k \geqslant 0$. We shall see in Section 6.2 that all these inclusions are strict for $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$.

Example 6.3 (Counting letters)

The function $\mathsf{nb}_{a_1,\dots,a_k} : u \mapsto |u|_{a_1} \times \cdots \times |u|_{a_k}$ belongs to $\mathbb{N}\mathsf{blind}_k$.

Example 6.4 (Polynomial parity)

The function $\mathsf{poly\text{-}parity}_k : u \mapsto (-1)^{|u|} \times |u|^k$ belongs to $\mathbb{Z}\mathsf{blind}_k$ thanks to a k-pebble transducer which produces an output either in $\{+1\}^*$ or $\{-1\}^*$.

One can shift closure properties from polyblind to $\mathbb{S}$-polyblind functions, as we did in Proposition 5.7 when showing closure properties of $\mathbb{S}$-polyregular functions.

Proposition 6.5 (Pre-composition by regular functions)

For all $k \geqslant 0$, the class $\mathbb{S}\mathsf{blind}_k$ is (effectively) closed under pre-composition by regular functions.

Proof. For $k \geqslant 1$, we rely on Theorem 3.6 which implies that the class of functions computed by blind k-pebble transducers is closed under pre-composition by regular functions. ◀

6.1.2 Blind counting transducers

Now we describe a simple variant of counting transducers which captures $\mathbb{S}$-polyblind functions. The main idea is that a *blind k-counting transducer* is a k-counting transducer which can only check regular properties of each position in a k tuple of positions, but not on the tuple itself[2]. In other words, it checks k-tuples of properties of $\mathsf{RegProp}_1$ instead of properties of $\mathsf{RegProp}_k$.

Definition 6.6 (Counting transducer)

Let $k \geqslant 0$. A *blind k-counting transducer* $\mathscr{T} = (A, \mathbb{S}, (\delta_i, (L_{i,j})_{1 \leqslant j \leqslant k})_{1 \leqslant i \leqslant n})$ consists of:

- an input alphabet A and an output commutative monoid $\mathbb{S}$;
- a sequence $(\delta_i, (L_{i,j})_{1 \leqslant j \leqslant k})_{1 \leqslant i \leqslant n}$ of *pairs* with $\delta_i \in \mathbb{S}$ and $L_{i,j} \in \mathsf{RegProp}_1(A)$.

The blind k-counting transducer $\mathscr{T}$ computes the function $\sum_{i=1}^{n} \delta_i \cdot (\#L_{i,1} \times \cdots \times \#L_{i,k})$. For all $1 \leqslant i \leqslant n$, one can build $R_i \in \mathsf{RegProp}_k(A)$ such that $\#R_i = \#L_{i,1} \times \cdots \times \#L_{i,k}$. Therefore a blind k-counting transducer can be seen as a particular case of k-counting transducer.

Example 6.7 (Polynomial parity)

The function $\mathsf{poly\text{-}parity}_k : u \mapsto \mathbf{1}_{\mathsf{even}}(u) \times |u|^k - \mathbf{1}_{\mathsf{odd}}(u) \times |u|^k$ can be computed by a blind k-counting transducer with two pairs.

A blind k-counting transducer can be seen as an algorithm with k nested (one-way) for loops, as described in Algorithm 6.8 for $k = 3$. The reader is invited to compare carefully Algorithms 5.13 and 6.8. As mentioned above, the key difference between them is the following: Algorithm 5.13 checks a regular property of the tuple of positions (i_1, i_2, i_3), while Algorithm 6.8 checks properties of i_1, i_2 and i_3 separately, and then recombines this information to select its output.

[2] One could imagine intermediate versions of this model, e.g. by allowing to check properties on pairs of positions but not on triples, etc. This gradation is somehow similar to the notion of *variable independence* in database theory, see e.g. [FT08].

Algorithm 6.8: Implementation of a blind 3-counting transducer with nested loops.

```
for i_1 in [1:|u|] do
    for i_2 in [1:|u|] do
        for i_3 in [1:|u|] do
            if u•i_1 ∈ L_1 and u•i_2 ∈ L_2 and u•i_3 ∈ L_3 then
                Output δ
            end
            if u•i_1 ∈ L'_1 and u•i_2 ∈ L'_2 and u•i_3 ∈ L'_3 then
                Output δ'
            end
        end
    end
end
```

We claim in Theorem 6.9 that blind counting transducers (unsurprisingly) compute the class of $\mathbb{S}$-polyblind functions. This easy result originates from [Dou21, Proposition 3.4] for $\mathbb{S} = \mathbb{N}$ (it is also a consequence of [NNP21, Corollary 5.7]). It is an analogue of Theorem 5.15 in our setting.

Theorem 6.9 (Blind pebble = Blind counting)

Let $\mathbb{S}$ be a commutative monoid and $k \geqslant 0$. A function $f\colon A^* \to \mathbb{S}$ belongs to $\mathbb{S}\mathsf{blind}_k$ if and only if it can be computed by a blind k-counting transducer. The conversions are effective.

Proof idea. The transformation from a blind k-counting transducer to $\mathbb{S}\mathsf{blind}_k$ is trivial, we focus on the converse one. We show it by induction for $k \geqslant 1$. The base case being trivial, let us consider a blind $(k+1)$-pebble transducer $\mathscr{T}\langle\mathscr{B}_1\rangle\cdots\langle\mathscr{B}_p\rangle$ whose subtrees $\mathscr{B}_1, \dots, \mathscr{B}_p$ have heads $\mathscr{T}_1, \dots, \mathscr{T}_p$. Let $T := \{\mathscr{T}_1, \dots, \mathscr{T}_p\}$ and $g\colon A^* \to T^*$ be the function computed by $\mathscr{T}$. For all $1 \leqslant i \leqslant n$, the function $u \mapsto |g(u)|_{\mathscr{T}_i}$ belongs to $\mathbb{S}\mathsf{blind}_1 = \mathbb{S}\mathsf{poly}_1$ thanks to Theorem 5.15. We conclude by induction since $[\![\mathscr{T}]\!](u) = \sum_{i=1}^{n} |g(u)|_{\mathscr{T}_i} \times [\![\mathscr{T}_i]\!](u)$ for all $u \in A^*$. ◀

6.1.3 $\mathbb{S}$-polyblind functions as $\mathbb{S}$-rational series

Now we characterize the class of $\mathbb{S}$-polyblind functions as a natural subclass of $(\mathbb{S}, +, \times)$-rational series for $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$. The results of this section are merely reformulations of Theorem 6.9.

We first give an analogue of Lemmas 5.19 and 5.21 when dealing with Hadamard product.

Lemma 6.10 (Closure properties of $\mathbb{S}$-polyblind functions)

Let $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$. The class of $\mathbb{S}$-polyblind functions is closed under Hadamard products. More precisely, if $f \in \mathbb{S}\mathsf{blind}_k$ and $g \in \mathbb{S}\mathsf{blind}_\ell$, then $f \times g \in \mathbb{S}\mathsf{blind}_{k+\ell}$. The construction is effective.

Furthermore, for all $k \geqslant 0$, the following equality holds and the conversions are effective:

$$\mathbb{S}\mathsf{blind}_{k+1} = \mathsf{Span}_{\mathbb{S}}(\{f \times g \mid f \in \mathbb{S}\mathsf{blind}_1, g \in \mathbb{S}\mathsf{blind}_k\}).$$

Proof. Use Theorem 6.9 and the definition of blind counting transducers. ◀

Example 6.11 (Counting letters)

For all $a_1, \dots, a_k \in A$, the function $\mathsf{nb}_{a_1,\dots,a_k}$ equals $\mathsf{nb}_{a_1} \times \cdots \times \mathsf{nb}_{a_k}$.

We refer to $\mathbb{S}\mathsf{poly}_1 = \mathbb{S}\mathsf{blind}_1$ as the class of $\mathbb{S}$-*regular functions* since it describes the function $\mathsf{sum} \circ f$ where f is regular[3]. It follows from Theorem 5.22 that $\mathbb{S}$-polyregular functions is the smallest class containing the $\mathbb{S}$-regular functions and closed under external products, sums and Cauchy products. Theorem 6.12 provides an analogue of this statement, it originates from [Dou22, Theorem 3.4].

Theorem 6.12 ($\mathbb{S}$-polyblind functions as $\mathbb{S}$-rational series)

Let $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$. A function $f : A^* \to \mathbb{S}$ is $\mathbb{S}$-polyblind if and only if it belongs to smallest class of functions of type $A^* \to \mathbb{S}$ containing the $\mathbb{S}$-regular functions[4] and closed under external products[5], sums and Hadamard products. The conversions are effective.

Proof. Apply Theorem 6.9 and Lemma 6.10. ◀

One may ask whether the notion of $\mathbb{S}$-polyblind functions can be generalized to define a larger class of *blind* $\mathbb{S}$-*rational series*, for instance by means of an equivalent of Kleene star using Hadamard product instead of the Cauchy one. We believe that such an extension is related to the extension of blind pebble transducers to recursive blind pebble transducers (which was discussed in Section 4.5) and therefore seems to be irrelevant for $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$. We note that [Cho17, Section 2.1] introduces a notion called *Hadamard star* on rational series, but it has no interest[6] for $(\mathbb{Z}, +, \times)$-rational series

6.2 Membership problem for $\mathbb{S}$-polyblind functions

The goal of this section is to state the main result of Chapter 6, which claims that for $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$, one can decide if a $\mathbb{S}$-polyregular function is $\mathbb{S}$-polyblind. We also provide a semantic condition called *repetitiveness* which characterizes these $\mathbb{S}$-polyblind functions. In addition to its own interest, this characterization will be used as a key ingredient within the proof of the decidability result.

6.2.1 Repetitive functions

We first introduce the notion of *repetitive* function, which originates from [Dou22, Definition 4.1]. Intuitively, if a function is repetitive and the same factor is repeated in two blocks of its input, then the value of its output will depend of the total number of iterations, but not on the size of the blocks. This means that the function cannot distinguish between two repetitions of the same factor. Formally, the notion of *repetitiveness* is presented in Definition 6.13. Since we shall intend to use pumping arguments, we provide a statement which deals with "long enough" repetitions of words.

[3] Due to commutativity, one can replace regular by rational in this statement. Hence this class could also be called $\mathbb{S}$-*rational functions*, but we avoid this terminology since it can create confusion with the (distinct) class of $\mathbb{S}$-rational series

[4] Contrary to what happens in Item (3) of Theorem 5.22, here one cannot replace $\mathbb{S}$-regular functions by indicator functions of regular languages. Indeed, otherwise the closure would describe no more than $\mathbb{S}\mathsf{poly}_0 = \mathbb{S}\mathsf{blind}_0$ because of Lemma 6.10.

[5] Since we start from $\mathbb{S}$-regular functions, this operation is in fact not needed.

[6] It is only defined for $(\mathbb{Q}, +, \times)$-rational series $f\colon A^* \to \mathbb{S}$ such that $\sum_{n \geqslant 0} f(u)^n$ converges for all $u \in A^*$.

Definition 6.13 (Repetitive function)

Let $k \geqslant 1$. We say that a function $f : A^* \to \mathbb{S}$ is *k-repetitive* if there exists $\Omega \geqslant 1$, such that the following holds. For all $s, v_0, u_1, v_1, \ldots, u_k, v_k, t \in A^*$ and $N \geqslant 1$ multiple of Ω, define:

$$W : \mathbb{N}^k \to A^*, X_1, \ldots, X_k \mapsto v_0 u_1^{NX_1} v_1 \cdots v_{k-1} u_k^{NX_k} v_k$$

and let $w := W(1, \ldots, 1)$. Then there exists a function $F : \mathbb{N}^k \to \mathbb{S}$ such that for all tuple $\overline{X} := X_1, \ldots, X_k \geqslant 3$ and $\overline{Y} := Y_1, \ldots, Y_k \geqslant 3$, we have:

$$f(sw^{2N-1} W(\overline{X}) w^{N-1} W(\overline{Y}) w^N t) = F(X_1 + Y_1, \ldots, X_k + Y_k).$$

Observe that if f is k-repetitive, then f is also ℓ-repetitive for all $1 \leqslant \ell \leqslant k$. Now let us give a few examples in order so see when this criterion holds, or not.

Example 6.14 (Counting letters)

The function $\mathsf{nb}_a : u \mapsto |u|_a$ is k-repetitive for all $k \geqslant 1$. Indeed, with the notations of Definition 6.13 it is easy to show that if $C := |sw^{2N-1} w^{N-1} w^N t|_a$ then:

$$\mathsf{nb}_a(sw^{2N-1} W(\overline{X}) w^{N-1} W(\overline{Y}) w^N t) = (X_1 + Y_1) N |u_1|_a + \cdots + (X_k + Y_k) N |u_k|_a + C.$$

More generally, the function $\mathsf{nb}_{a_1, \ldots, a_\ell} : u \mapsto |u|_{a_1} \times \cdots \times |u|_{a_\ell}$ is k-repetitive for all $k \geqslant 1$.

Example 6.15 (Unary input alphabet)

A function $f : \{1\}^* \to \mathbb{S}$ (with unary input alphabet) is k-repetitive for all $k \geqslant 1$. Indeed, with the notations of Definition 6.13, $\overline{X}, \overline{Y} \mapsto sw^{2N-1} W(\overline{X}) w^{N-1} W(\overline{Y}) w^N t \in \{1\}^*$ is a function of $X_1 + Y_1, \ldots, X_k + Y_k$, hence so is its image $f(sw^{2N-1} W(\overline{X}) w^{N-1} W(\overline{Y}) w^N t)$.

Example 6.16 (Map power)

For all $k \geqslant 2$, the function $\mathsf{map\text{-}power}_k : 0^{n_1} 1 \cdots 1 0^{n_m} \mapsto \sum_{i=1}^m n_i^k$ is not 1-repetitive. Let us choose any $\Omega \geqslant 1$ and fix $s = t := \varepsilon$, $u_1 := 0$ and $v_0 = v_1 := 1$, then:

$$\begin{aligned}\mathsf{map\text{-}power}_k(W(X_1, Y_1)) &= \mathsf{map\text{-}power}_k((10^\Omega 1)^{2\Omega - 1} 10^{\Omega X_1} 1 (10^\Omega 1)^{\Omega - 1} 10^{\Omega Y_1} 1 (10^\Omega 1)^\Omega) \\ &= \Omega^k (4\Omega - 2) + \Omega^k X_1^k + \Omega^k Y_1^k\end{aligned}$$

which is not a function of $X_1 + Y_1$ for $k \geqslant 2$.

6.2.2 Decidability result of $\mathbb{S}$-polyblind inside $\mathbb{S}$-polyregular

Now we are ready to decide and characterize $\mathbb{S}$-polyblind functions among the $\mathbb{S}$-polyregular ones. Theorem 6.17 originates from [Dou22, Theorem 4.6] in the case $\mathbb{S} := \mathbb{N}$. The proof of this result goes over Sections 6.3 to 6.5 and it relies once more on the factorization forests techniques which were introduced in Chapter 5, while being more involved than the previous proof.

Theorem 6.17 ($\mathbb{S}$-polyregular $\to$ $\mathbb{S}$-polyblind)

Let $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$. A function $f \in \mathbb{S}\mathsf{poly}_k$ is $\mathbb{S}$-polyblind if and only if it is k-repetitive. This property is decidable. If it holds, one can build a blind k-counting transducer which computes f.

Proof sketch. The main idea is to show by induction on $k \geqslant 1$ that if $f \in \mathbb{S}\mathsf{poly}_k$ is a k-repetitive function, then f can be written as a sum $f_1 + f_2$ where $f_1 \in \mathbb{S}\mathsf{poly}_{k-1}$ is $(k-1)$-repetitive (therefore $f_1 \in \mathbb{S}\mathsf{blind}_{k-1}$ by induction hypothesis) and $f_2 \in \mathbb{S}\mathsf{blind}_k$. Beware that f_1 and f_2 will not exactly be the same functions as those of the proof sketch of Theorem 5.25 in Chapter 5. For decidability, we provide a syntactic condition on k-counting transducers called permutability, which is inspired by pumpability. Formally, Theorem 6.17 follows from Theorem 6.51. ◀

Let us discuss low hanging consequences of Theorem 6.17. By leveraging Example 6.16, we first provide in Example 6.18 separating examples between $\mathbb{S}$-polyregular and $\mathbb{S}$-polyblind functions.

Example 6.18 (Strict hierarchies)

Let $k \geqslant 2$. The function $\mathsf{map\text{-}power}_k : 0^{n_1}1 \cdots 10^{n_m} \mapsto \sum_{i=1}^m n_i^k$ is $\mathbb{N}$-polyregular but neither $\mathbb{N}$-polyblind nor $\mathbb{Z}$-polyblind, since it is not 1-repetitive. In a similar fashion, the modified function $0^{n_1}1 \cdots 10^{n_m} \mapsto \sum_{i=1}^m (-1)^{n_i} n_i^k$ is $\mathbb{Z}$-polyregular but not $\mathbb{Z}$-polyblind.

Now we show in Corollary 6.19 that the classes of $\mathbb{S}$-polyregular and $\mathbb{S}$-polyblind functions coincide when the inputs are unary. This result originates from [Dou22, Corollary 4.9][7].

Corollary 6.19 (Unary input alphabet)

If the input alphabet is unary, the classes of $\mathbb{N}$-polyregular (resp. $\mathbb{Z}$-polyregular) and $\mathbb{N}$-polyblind (resp. $\mathbb{Z}$-polyblind) functions coincide. The transformations are effective.

Proof. A function with unary input alphabet is k-repetitive for all $k \geqslant 1$ by Example 6.15. ◀

Another consequence of Theorem 6.17 is presented in Corollary 6.20 and depicted in Figure 6.21.

Corollary 6.20 ($\mathbb{N}\mathsf{poly} \cap \mathbb{Z}\mathsf{blind} = \mathbb{N}\mathsf{blind}$)

$\mathbb{N}$-polyblind functions are exactly the $\mathbb{N}$-polyregular functions which are $\mathbb{Z}$-polyblind.

Proof. Any function of $\mathbb{N}\mathsf{blind}$ belongs both to $\mathbb{N}\mathsf{poly}$ and $\mathbb{Z}\mathsf{blind}$. Conversely, if a function f belongs both to $\mathbb{Z}\mathsf{blind}_k$ and $\mathbb{N}\mathsf{poly}_\ell$ for some $k, \ell \geqslant 0$, then $|f(u)| = \mathcal{O}(|u|^{\mathsf{min}(\ell,k)})$ and therefore by Theorem 5.25 one can assume that $\ell \leqslant k$. By Theorem 6.17, f is k-repetitive and thus ℓ-repetitive. Finally we get $f \in \mathbb{N}\mathsf{blind}_\ell$ by applying the other direction of Theorem 6.17. ◀

We also observe that Theorems 5.25 and 6.17 provide an optimization result for $\mathbb{S}$-polyblind functions (the result for $\mathbb{S} := \mathbb{N}$ is also a consequence of Theorem 3.12, but it is not the case of $\mathbb{S} := \mathbb{Z}$).

Corollary 6.22 (Optimization of blind pebble transducers with commutative output)

Let $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$. Let $f \in \mathbb{S}\mathsf{blind}$ and $k \geqslant 0$, then $f \in \mathbb{S}\mathsf{blind}_k$ if and only if $|f(u)| = \mathcal{O}(|u|^k)$. This property is decidable. If it holds, one can build a blind k-counting transducer computing f.

[7] This result was first claimed by Nguyên and Pradic in an unpublished note on polyregular functions with unary input.

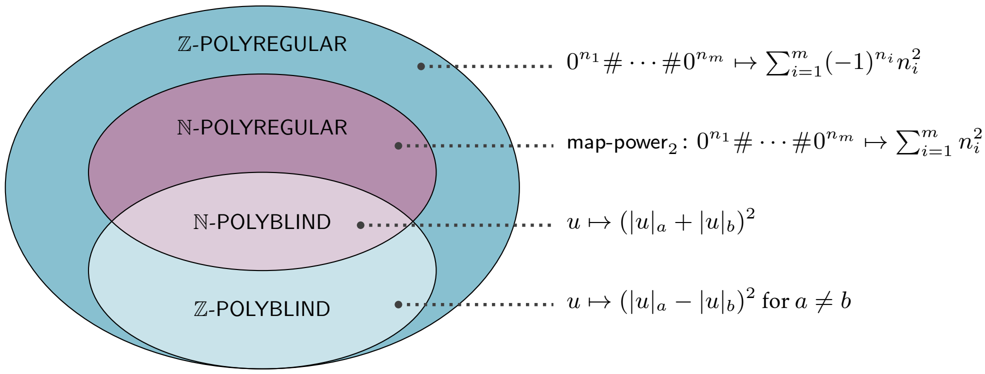


Figure 6.21: Relationship between $\mathbb{S}$-polyregular and $\mathbb{S}$-polyblind functions for $\mathbb{S} = \mathbb{N}$ and $\mathbb{Z}$.

Proof. Let $f \in \mathbb{S}$blind be such that $|f(u)| = \mathcal{O}(|u|^k)$. We get $f \in \mathbb{S}\text{poly}_k$ by Theorem 5.25 and furthermore f is k-repetitive by Theorem 6.17. Thus one can build a blind k-counting transducer which computes f by Theorem 6.17. The converse is obvious. ◀

The reader may ask whether the notion repetitiveness can be simplified to obtain a simpler semantic characterization. The author believes that considering 1-repetitiveness instead of k-repetitiveness is sufficient, however he is not aware of a proof of this more precise result.

6.3 Repetitive functions and permutable counting transducers

Let $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$. In order to show the optimization result for $\mathbb{S}$-polyregular functions in Chapter 5 (and the comparable results in Chapter 3) we have relied on an equivalence between the semantic condition $|f(u)| = \mathcal{O}(|u|^k)$ and the decidable property of counting transducers called pumpability. In the current setting of $\mathbb{S}$-polyblind functions, our goal is to use repetitiveness as a semantic condition and to replace pumpability by the concept of *permutability* which will be introduced in Section 6.3.2.

Formally, let $f : A^* \to \mathbb{S}$ be computed by a k-counting transducer $\mathscr{T}$. In a perfect world, the author would aim at showing that the following conditions are equivalent:

(1) f is $\mathbb{S}$-polyblind; (2) f is k-repetitive; (3) $\mathscr{T}$ is permutable.

We show Item (1) $\Rightarrow$ Item (2) in Section 6.3.1 and Item (2) $\Rightarrow$ Item (3) in Section 6.3.2. However, we do not know whether Item (3) $\Rightarrow$ Item (1) holds. We shall see in Sections 6.4 and 6.5 that permutability can nevertheless be used as a tool to build an inductive and effective proof of Item (2) $\Rightarrow$ Item (1).

6.3.1 Polyblind functions are repetitive

Let us observe that repetitiveness is preserved under the basic operations which build the class $\mathbb{Z}$blind.

Claim 6.23 (Preservation of repetitiveness under $\cdot$, $+$ and $\times$)

Let $k \geqslant 0$, $\delta \in \mathbb{Z}$ and $f, g : A^* \to \mathbb{Z}$ be k-repetitive. Then $\delta \cdot f$, $f + g$, and $f \times g$ are k-repetitive.

Proof idea. Let Ω_f (resp. Ω_g) be the constant Ω given by Definition 6.13 for f (resp. for g), then the constant $\Omega_f \Omega_g$ is suitable for $f + g$ and $f \times g$. Indeed, fix $s, v_0, u_1, v_1, \ldots, u_k, v_k, t \in A^*$ and let F (resp. G) the function given by Definition 6.13 for f (resp. for g), then $F + G$ (resp. $F \times G$) shows the result for $f + g$ (resp. $f \times g$). Furthermore Ω_f is suitable for $\delta \cdot f$. ◀

By combining Claim 6.23 with the properties of $\mathbb{S}$-regular functions, we obtain Lemma 6.24.

Lemma 6.24 (Polyblind ⇒ repetitive)

Let[8] $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$. A $\mathbb{S}$-polyblind function is k-repetitive[9] for all $k \geqslant 0$.

Proof. Thanks to Claim 6.23 and Theorem 6.12, it is enough to show the result when f is $\mathbb{S}$-regular, i.e. computed by a 1-counting transducer $\mathscr{T}$. Let $\mu\colon A^* \to \mathbb{T}$ be its transition morphism and Ω be the idempotence index of $\mathbb{T}$, i.e. the smallest $\Omega > 0$ such that m^Ω is idempotent for all $m \in \mathbb{T}$.

Let $k \geqslant 1$, $s, v_0, u_1, v_1, \dots, u_k, v_k, t \in A^*$ and $N \geqslant 1$ be a multiple of Ω. Define the function $W : \mathbb{N}^k \to A^*$ as we did in Definition 6.13 and let $w := W(1, \dots, 1)$. By definition of Ω and N, $e_i := \mu({u_i}^\Omega) = \mu({u_i}^N)$ is an idempotent for all $1 \leqslant i \leqslant k$. Hence $p := \mu(w) = \mu(v_0)e_1\mu(v_1)\cdots e_k\mu(v_k) = \mu(W(X_1, \dots, X_k))$ is independent of $X_1, X_2, \dots, X_k \geqslant 1$ and $e := p^N$ is idempotent. In order to simplify the notations, from now on we assume that $s = t = \varepsilon$. Let $\overline{X} := X_1 \dots, X_k \geqslant 3$ and $\overline{Y} := Y_1 \dots, Y_k \geqslant 3$, then we can decompose the productions as follows thanks to Claim 5.27 and Proposition-Definition 5.31:

$$\begin{aligned}
&f(w^{2N-1}W(\overline{X})w^{N-1}W(\overline{Y})w^N) = \mathsf{prod}_{\mathscr{T}}(\lfloor w^{2N-1} \rfloor pp^{N-1}pp^N) \\
&+ \mathsf{prod}_{\mathscr{T}}(p^{2N-1}\lfloor v_0u_1^{NX_1}\cdots v_k \rfloor p^{N-1}pp^N) + \mathsf{prod}_{\mathscr{T}}(p^{2N-1}p\lfloor w^{N-1} \rfloor pp^N) \\
&+ \mathsf{prod}_{\mathscr{T}}(p^{2N-1}pp^{N-1}\lfloor v_0u_1^{NY_1}\cdots v_k \rfloor p^N) + \mathsf{prod}_{\mathscr{T}}(p^{2N-1}pp^{N-1}p\lfloor w^N \rfloor) \\
&= \mathsf{prod}_{\mathscr{T}}(\lfloor w^{2N-1} \rfloor ep) + \mathsf{prod}_{\mathscr{T}}(e\lfloor w^{N-1} \rfloor ep) + \mathsf{prod}_{\mathscr{T}}(e\lfloor w^N \rfloor) \\
&+ \mathsf{prod}_{\mathscr{T}}(ep^{N-1}\lfloor v_0u_1^{NX_1}\cdots v_k \rfloor e) + \mathsf{prod}_{\mathscr{T}}(ep^{N-1}\lfloor v_0u_1^{NY_1}\cdots v_k \rfloor e).
\end{aligned} \tag{6.25}$$

The three first terms do not depend on $\overline{X}$ or $\overline{Y}$, thus only need to focus on the two last ones. We show in Claim 6.26 how to decompose their productions.

Claim 6.26 (Polynomial of degree $\leqslant$ 1)

For all $m, n \in \mathbb{T}$, there exists a polynomial $L \in \mathbb{Z}[X_1, \dots, X_k]$ of degree at most 1 such that $\mathsf{prod}_{\mathscr{T}}(m\lfloor v_0u_1^{NX_1}\cdots v_k \rfloor n) = L(\overline{X})$ for all $\overline{X} := X_1, \dots, X_k \geqslant 3$.

Proof sketch. We decompose $\mathsf{prod}_{\mathscr{T}}(m\lfloor v_0u_1^{NX_1}\cdots v_k \rfloor n)$ as the sum of $2k + 1$ terms using Claim 5.33. Then we apply Claim 5.33 to deal with the terms containing $\lfloor u_i^{NX_i} \rfloor$ (we crucially rely on the fact that $e_i = \mu(u_i^N)$ is an idempotent). ◀

Thus $\mathsf{prod}_{\mathscr{T}}(ep^{N-1}\lfloor v_0u_1^{NX_1}\cdots v_k \rfloor e) = L(\overline{X})$ and $\mathsf{prod}_{\mathscr{T}}(ep^{N-1}\lfloor v_0u_1^{NY_1}\cdots v_k \rfloor e) = L(\overline{Y})$ for all $X_1, \dots, X_k, Y_1, \dots, Y_k \geqslant 3$, for some polynomial L of degree at most 1. Thanks to Equation (6.25), there exists $C \in \mathbb{Z}$ such that $f(w^{2N-1}W(\overline{X})w^{N-1}W(\overline{Y})w^N) = L(\overline{X}) + L(\overline{Y}) + C$ for all $X_1, \dots, X_k, Y_1, \dots, Y_k \geqslant 3$. Since L is a polynomial of degree 1, we finally obtain the function F of Definition 6.13 by grouping the terms in X_i and Y_i for $1 \leqslant i \leqslant k$. ◀

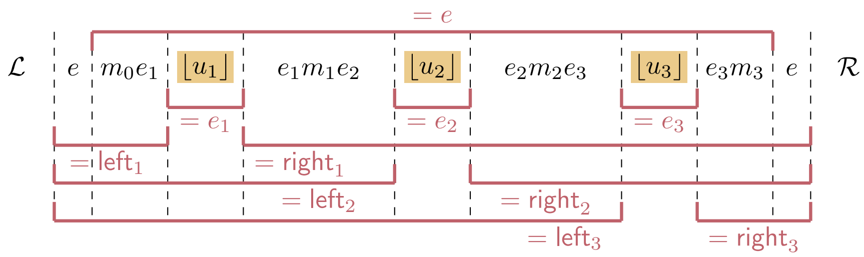
[8] The techniques can be adapted to show that this result holds for any commutative monoid $\mathbb{S}$.

[9] We in fact show a stronger result: the function F of Definition 6.13 turns out to be a polynomial in $X_1 + Y_1, \dots, X_k + Y_k$.

6.3.2 Repetitive functions are computed by permutable transducers

Now we describe a necessary condition, named *permutability*, for a k-counting transducer $\mathscr{T}$ to compute a function $f \in \mathbb{S}\mathsf{blind}$. It can be seen as an analogue of pumpability in a different setting. We shall not show that this condition is sufficient[10], however it will be enough for doing an inductive proof.

6.3.2.1 Permutability. Intuitively, permutability of $\mathscr{T}$ means that $\mathsf{prod}_{\mathscr{T}}(m_0 \lfloor u_1 \rfloor m_1 \cdots \lfloor u_k \rfloor m_k)$ only depends on the μ-1-contexts $m_0\mu(u_1)\cdots m_i \lfloor u_i \rfloor m_{i+1}\mu(u_{i+1})\cdots m_k$ for $1 \leqslant i \leqslant k$, where $\mu \colon A^* \to \mathbb{T}$ is the transition morphism of $\mathscr{T}$. In particular, this production does not depend on the relative position of the u_i nor on the m_i which separate them. This behavior is close to that of a blind k-counting transducer, as explained above when comparing Algorithms 5.13 and 6.8.



(a) A μ-$(3, 2^{3|\mathbb{T}|})$-iterator and the definition of left_j and right_j for $1 \leqslant j \leqslant 3$.

(b) Separating the $\lfloor u_j \rfloor$ with the left_j and right_j, and permuting them with σ.

Figure 6.27: Productions which must be equal in Definition 6.28, with $x = 3$ and $\sigma = (3, 1, 2)$.

Definition 6.28 (Permutable counting transducer)

Let $\mathscr{T}$ be a k-counting transducer whose transition morphism is $\mu \colon A^* \to \mathbb{T}$. We say that $\mathscr{T}$ is *permutable* if for all $(\ell, x, r) \in \mathsf{S}_k$, for all μ-$(\ell, 2^{3|\mathbb{T}|})$-iterator $\mathcal{L}$, for all μ-$(r, 2^{3|\mathbb{T}|})$-iterator $\mathcal{R}$, for all μ-$(x, 2^{3|\mathbb{T}|})$-iterator $m_0e_1 \lfloor u_1 \rfloor e_1 \cdots e_x \lfloor u_x \rfloor e_x m_x$ such that $e := m_0e_1m_1 \cdots e_xm_x$ is idempotent, for all permutation σ of $[1{:}x]$, the following holds.

Define $\mathsf{left}_j := em_0e_1 \cdots m_{j-1}e_j$ and $\mathsf{right}_j := e_jm_j \cdots e_xm_xe$ for $1 \leqslant j \leqslant x$, then:

$$\begin{aligned} &\mathsf{prod}_{\mathscr{T}} \left(\mathcal{L}\, e\, m_0m_0e_1 \lfloor u_1 \rfloor e_1 \cdots e_x \lfloor u_x \rfloor e_xm_x\, e\, \mathcal{R}\right) \\ =&\mathsf{prod}_{\mathscr{T}} \left(\mathcal{L}\ \mathsf{left}_{\sigma(1)}\ \lfloor u_{\sigma(1)} \rfloor\ \mathsf{right}_{\sigma(1)} \cdots \mathsf{left}_{\sigma(x)}\ \lfloor u_{\sigma(x)} \rfloor\ \mathsf{right}_{\sigma(x)}\ \mathcal{R}\right). \end{aligned}$$

A visual representation of permutability is depicted in Figure 6.27. Observe that this property is decidable for the same reasons than those for which pumpability was decidable. Indeed, it suffices to range over all $(\ell, x, r) \in \mathsf{S}_k$ and all μ-$(\ell, 2^{3|\mathbb{T}|})$-iterators, μ-$(r, 2^{3|\mathbb{T}|})$-iterators, μ-$(x, 2^{3|\mathbb{T}|})$-iterators (there are finitely many of them) and permutations σ, and to compute their productions.

6.3.2.2 Repetitiveness implies permutability. The remainder of Section 6.3.2 is devoted to showing that a k-counting transducer which computes a k-repetitive function $f \colon A^* \to \mathbb{Z}$ is permutable. The proof of this result is based on the iteration techniques developed in Section 5.3.3.

[10] The author does not know whether this property hold or not.

In Lemma 6.29, we roughly show that repetitiveness of the function implies that the counting transducer is permutable when restricting the condition to permutations $\sigma\colon [1{:}x] \to [1{:}x]$ such that $\sigma(j) = x$ for some $1 \leqslant j \leqslant x$, $\sigma(i) = i$ for $1 \leqslant i < j$ and $\sigma(i) = i{-}1$ for $j < i \leqslant x$.

Lemma 6.29 (Repetitive ⇒ Permutable with a simple permutation)

Let $k, K \geqslant 0$ and $f\colon A^* \to \mathbb{Z}$ be a k-repetitive function computed by a k-counting transducer $\mathscr{T}$ with transition morphism $\mu\colon A^* \to \mathbb{T}$. For all $(\ell, x, r) \in \mathsf{S}_k$, for all μ-(ℓ, K)-iterator $\mathcal{L}$, for all μ-(r, K)-iterator $\mathcal{R}$, for all μ-(x, K)-iterator $m_0 e_1 \lfloor u_1 \rfloor e_1 \cdots e_x \lfloor u_x \rfloor e_x m_x$ such that $e := m_0 e_1 m_1 \cdots e_x m_x$ is idempotent and for all $1 \leqslant j \leqslant x$, we have the following:

$$\mathsf{prod}_{\mathscr{T}}(\mathcal{L}\, e m_0 e_1 \lfloor u_1 \rfloor e_1 \cdots e_x \lfloor u_x \rfloor e_x m_x e\, \mathcal{R})$$
$$= \mathsf{prod}_{\mathscr{T}}\!\left(\mathcal{L}\, e m_0 \left(\prod_{i=1}^{j-1} e_i \lfloor u_i \rfloor e_i m_i\right) e_j m_j \left(\prod_{i=j+1}^{x} e_i \lfloor u_i \rfloor e_i m_i\right) e\, (\mathsf{left}_j \lfloor u_j \rfloor \mathsf{right}_j)\, \mathcal{R}\right).$$

Proof. The idea is to build a word in which the two productions compared in Lemma 6.29 occur. Then we shall iterate well-chosen factors in this word and use the repetitiveness of f to show that these productions must be equal. Let $N \geqslant 3$ be given by Definition 6.13. We let $\mathcal{L} = p_0 f_1 \lfloor v_1 \rfloor f_1 \cdots f_\ell \lfloor v_\ell \rfloor f_\ell p_\ell$ and $\mathcal{R} = p'_0 f'_1 \lfloor v'_1 \rfloor f'_1 \cdots f'_r \lfloor v'_r \rfloor f'_r p'_r$. For all $m \in \mathbb{T}$, let us fix a word $\nu(m) \in \mu^{-1}(\{m\})$. Then we define the following functions:

- $U\colon \mathbb{N}^\ell \to A^*, \overline{L} := L_1, \ldots, L_\ell \mapsto U(\overline{L}) := \nu(p_0) v_1^{L_1} \cdots v_\ell^{L_\ell} \nu(p_\ell)$;
- $W\colon \mathbb{N}^x \to A^*, \overline{X} := X_1, \ldots, X_x \mapsto W(\overline{X}) := \nu(m_0) {u_1}^{NX_1} \cdots {u_x}^{NX_x} \nu(m_x)$;
- $V\colon \mathbb{N}^r \to A^*, \overline{R} := R_1, \ldots, R_r \mapsto V(\overline{R}) := \nu(p'_0) {v'_1}^{R_1} \cdots {v'_r}^{R_r} \nu(p'_r)$;

Let $w := W(1, \ldots, 1)$. Observe that for all $\overline{X} \geqslant 1$, $\mu(W(\overline{X})) = \mu(w) = e$. We define the function $P\colon \mathbb{N}^\ell \times \mathbb{N}^x \times \mathbb{N} \times \mathbb{N}^r \to \mathbb{Z}$ which maps $(\overline{L}, \overline{X}, X'_j, \overline{R})$ to

$$f\left(U(\overline{L}) w^{2N-1} W(\overline{X}) w^{N-1} W(3, \ldots, 3, X'_j, 3, \ldots, 3) w^N V(\overline{R})\right).$$

where X'_j is in position j of $W(3, \ldots, 3, X'_j, 3, \ldots, 3)$.

Let $T := L_1 \cdots L_\ell X_1 \cdots X_{j-1} X_{j+1} \cdots X_x R_1 \cdots R_r$. By adapting the iteration techniques of Section 5.3.3, it is easy to show that P is a polynomial whose coefficients in TX_j and TX'_j describe the productions we are looking for. This intuition is formalized[11] in Claim 6.30.

Claim 6.30 (Pumping an iterator of $k{+}1$ elements)

For $\overline{L}, \overline{X}, X'_j, \overline{R} \geqslant 2k+1$, $P(\overline{L}, \overline{X}, X'_j, \overline{R})$ is a polynomial of $\mathbb{Z}[\overline{L}, \overline{X}, X'_j, \overline{R}]$and:

- the coefficient in TX_j of P is $\alpha := \mathsf{prod}_{\mathscr{T}}(\mathcal{L}\, e m_0 e_1 \lfloor u_1 \rfloor e_1 \cdots e_k \lfloor u_k \rfloor e_k m_k e\, \mathcal{R})$;
- the coefficient in TX'_j of P is $\alpha' := \mathsf{prod}_{\mathscr{T}}(\mathcal{L}\, e m_0 \left(\prod_{i=1}^{j-1} e_i \lfloor u_i \rfloor e_i m_i\right) e_j m_j \left(\prod_{i=j+1}^{x} e_i \lfloor u_i \rfloor e_i m_i\right) e \left(\mathsf{left}_j \lfloor u_j \rfloor \mathsf{right}_j\right) \mathcal{R})$.

Proof idea. Use Claim 5.33 and adapt the proof of Lemma 5.37. ◀

On the other hand, we obtain Claim 6.31 by leveraging the fact that f is k-repetitive. This result shows that the $P(\overline{L}, \overline{X}, X'_j, \overline{R})$ only depends on $X_j + X'_j$.

[11] We do not need to assume that f is k-repetitive to show this result. Indeed, it is uniquely related to iterators.

Claim 6.31 (Using repetitiveness)

There exists a function $F\colon \mathbb{N}^k \to \mathbb{N}$ such that for $\overline{L}, \overline{X}, X'_j, \overline{R} \geqslant 2k+1$:

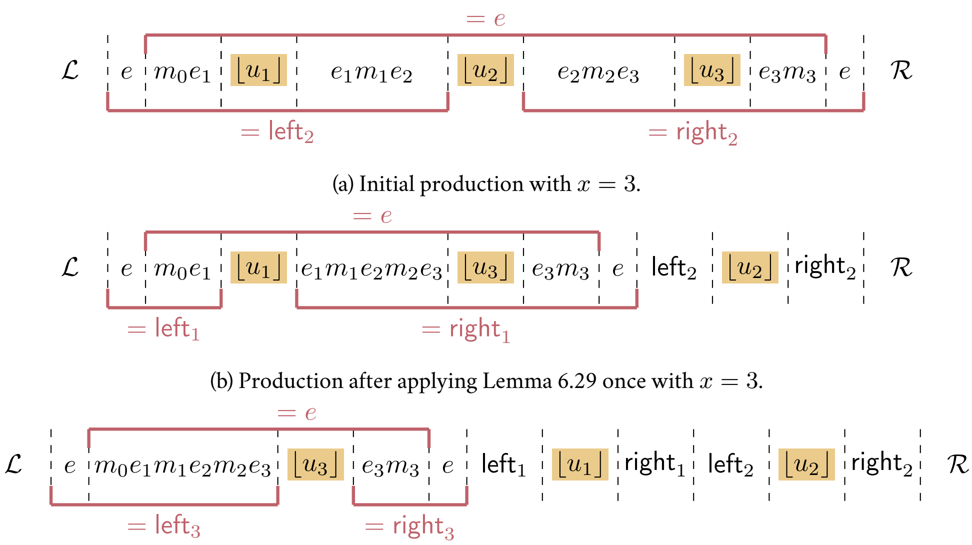
$$P(\overline{L}, \overline{X}, X'_j, \overline{R}) = F(\overline{L}, X_1, \ldots, X_{j-1}, X_{j+1}, \ldots, X_x, \overline{R}, X_j + X'_j).$$

Proof. The function f is x-repetitive[12] since $x \leqslant k$. Let $\overline{L}$ and $\overline{R}$ be fixed. By using the construction of P and Definition 6.13 (with $s := U(\overline{L})$ and $t := V(\overline{R})$), one shows that $\overline{X}, X'_j \mapsto P(\overline{L}, \overline{X}, X'_j, \overline{R})$ is a function of $X_1{+}3, \ldots, X_{j-1}{+}3, X_j{+}X'_j, X_{j+1}{+}3, \ldots, X_x{+}3$. ◀

From Claim 6.31 we deduce that for $\overline{L}, \overline{X}, X'_j, \overline{R} \geqslant 2k+1$:

$$\begin{aligned} &P(\overline{L}, X_1, \ldots, X_{j-1}, X_j, X_{j+1}, \ldots, X_x, X'_j, \overline{R}) \\ &= P(\overline{L}, X_1, \ldots, X_{j-1}, 2k{+}1, X_{j+1}, \ldots, X_x, X'_j{+}X_j{-}(2k{+}1), \overline{R}). \end{aligned} \tag{6.32}$$

By developing the polynomial of the last line of Equation (6.32), it is easy to see that the coefficients in TX_j and in TX'_j of P are equal. Hence Claim 6.30 implies that $\alpha = \alpha'$. ◀



(a) Initial production with $x = 3$.

(b) Production after applying Lemma 6.29 once with $x = 3$.

(c) Production after applying Lemma 6.29 once again with $x = 2$.

Figure 6.33: Proof idea for Lemma 6.34 with $x = 3$ and $\sigma = (3, 1, 2)$.

Now we are ready to show in Lemma 6.34 that repetitiveness implies permutability.

Lemma 6.34 (Repetitive ⇒ Permutable)

A k-counting transducer which computes a k-repetitive function $f : A^* \to \mathbb{Z}$ is permutable.

Proof idea. The proof proceeds by induction on $x \geqslant 1$, while relying on Lemma 6.29 to deal with the induction step. As an example, the induction steps of the proof are depicted in Figure 6.33 for $x = 3$ and $\sigma = (3, 1, 2)$: since u_2 has to be the last element after substitution, we first apply Lemma 6.29 with $x = 2$ and $j = 2$ to send it "on the right", then we do the same with u_1. ◀

[12] Note that 1-repetitiveness is not *a priori* sufficient. Indeed, the words which surround $u_i^{X_j}$ and $u_i'^{X'_j}$ are not the same.

6.4 Architectures and independent multisets

Let $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$ and $\mathscr{T}$ be k-counting transducer $\mathscr{T}$ which computes a function $f\colon A^* \to \mathbb{S}$. We have shown in Section 5.4 that f can effectively be written $(\mathsf{sum\text{-}dep}_{\mathscr{T}} + \mathsf{sum\text{-}ind}_{\mathscr{T}}) \circ \mathsf{forest}_{\mu}$, where $\mathsf{sum\text{-}dep}_{\mathscr{T}} \in \mathbb{S}\mathsf{poly}_{k-1}$. In a perfect world, the author would aim at showing that if $\mathscr{T}$ is permutable, then $\mathsf{sum\text{-}ind}_{\mathscr{T}} \in \mathbb{S}\mathsf{blind}_k$. Thereafter, we would be able to show by induction on $k \geqslant 1$ that if f is k-repetitive then $f \in \mathbb{S}\mathsf{blind}_k$. However, we believe that $\mathsf{sum\text{-}ind}_{\mathscr{T}} \in \mathbb{S}\mathsf{blind}_k$ does not hold[13].

In order to cope with this difficulty, we show Proposition 6.35 which provides a way to transform $\mathsf{sum\text{-}ind}_{\mathscr{T}}$ into a function of $\mathbb{S}\mathsf{blind}_k$, up to allowing an additional error term in $\mathbb{S}\mathsf{poly}_{k-1}$.

Proposition 6.35 (Decomposing $\mathsf{sum\text{-}ind}_{\mathscr{T}}$)

Let $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$, $k \geqslant 1$ and $\mathscr{T}$ be a permutable k-counting transducer with output in $\mathbb{S}$. One can build two functions $\mathsf{sum\text{-}ind}'_{\mathscr{T}} \in \mathbb{S}\mathsf{blind}_k$ and $\mathsf{sum\text{-}ind}''_{\mathscr{T}} \in \mathbb{S}\mathsf{poly}_{k-1}$ such that:

$$\mathsf{sum\text{-}ind}_{\mathscr{T}} = \mathsf{sum\text{-}ind}'_{\mathscr{T}} + \mathsf{sum\text{-}ind}''_{\mathscr{T}}.$$

Proof sketch. In Section 5.4.3, we have shown that the production of a k-counting transducer $\mathscr{T}$ on an independent (multi)set of nodes only depends on its linearization. The latter was a simple abstraction of the set of nodes and its environment. We improve this result when $\mathscr{T}$ is assumed to be permutable, by showing that the production only depends on a less precise abstraction named its *architecture*. Intuitively, this notion takes into account the fact that some nodes can be permuted. We then rely on architectures to build the functions $\mathsf{sum\text{-}ind}'_{\mathscr{T}}$ and $\mathsf{sum\text{-}ind}''_{\mathscr{T}}$. ◀

The rest of Section 6.4 is devoted to the detailed proof of Proposition 6.35. Formally, we define *architectures* in Section 6.4.1 and then justify in Section 6.4.2 that they have a suitable behavior with respect to productions. We justify in Section 6.4.3 that Proposition 6.35 holds for the $\mathbb{N}$-polyregular functions which counts the number of sets which have a given architecture in a forest. We conclude the proof of Proposition 6.35 in Section 6.4.4.

6.4.1 From linearizations to architectures

We first define the notion of *architecture* of an independent (multi)set of nodes. This abstraction is roughly a relaxation of the linearization of this set, when forgetting the relative positions of the nodes. This notion is presented in Definition 6.36 which originates from [Dou22, Definition 6.16].

Formally, the architecture of an independent set M in a forest is defined inductively in the same fashion as its linearization. The only difference is when we meet an idempotent node which has no elements of M in its rightmost nor in its leftmost subtree. In this case, we record the multiset containing the linearizations of each node taken independently, as a blind counting transducer would do. Recall that the *depth* of a node in a tree is defined inductively by starting from *root* which has depth 1. Given $\mathcal{F} \in \mathsf{Forests}^d_{\mu}$ and $\mathfrak{t} \in \mathsf{Nodes}_{\mathcal{F}}$, we let $\mathsf{depth}_{\mathcal{F}}(\mathfrak{t}) \in [1{:}d]$ be the depth of the node $\mathfrak{t}$ in $\mathcal{F}$.

Definition 6.36 (Architecture)

Let $k \geqslant 0$, $u \in A^+$, $\mathcal{F} \in \mathsf{Forests}_{\mu}(u)$ and $M \in \mathsf{Indep}^k_{\mathcal{F}}$. We define the *architecture* of M in $\mathcal{F}$ as a tree structure which is built inductively as follows:

- if $\mathcal{F} = a$, then[14] $k = 0$. We define $\mathsf{archi}_{\mathcal{F}}(M) := \mu(a)$;

[13] The intuition is that checking if two nodes are independent requires to compare their relative positions.

- otherwise $\mathcal{F} = \langle \mathcal{F}_1 \rangle \cdots \langle \mathcal{F}_n \rangle$ with $n \geqslant 1$:
  - if $k = 0$, we set $\mathsf{archi}_{\mathcal{F}}(M) = \langle \mu(u) \rangle$;
  - else if $M_1 := M \cap \mathsf{Nodes}_{\mathcal{F}_1} \neq \varnothing$, we let[15]:

$$\mathsf{archi}_{\mathcal{F}}(M) := \langle \mathsf{archi}_{\mathcal{F}_1}(M_1) \rangle \ \mathsf{archi}_{\langle \mathcal{F}_2 \rangle \cdots \langle \mathcal{F}_n \rangle}(M \setminus M_1).$$

  - else if $M_n := M \cap \mathsf{Nodes}_{\mathcal{F}_n} \neq \varnothing$, we define symmetrically:

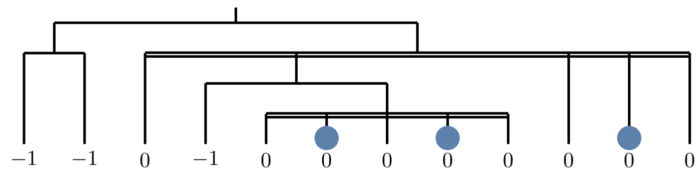
$$\mathsf{archi}_{\mathcal{F}}(M) := \mathsf{archi}_{\langle \mathcal{F}_1 \rangle \cdots \langle \mathcal{F}_{n-1} \rangle}(M \setminus M_n) \ \langle \mathsf{archi}_{\mathcal{F}_n}(M_n) \rangle.$$

  - else $M_1 = M_n = \varnothing$ but $k > 0$, thus $n \geqslant 3$ and $\mu(u)$ is idempotent. We define:

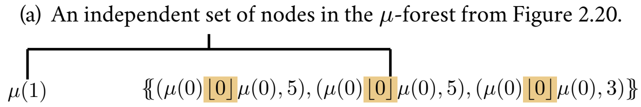
$$\mathsf{archi}_{\mathcal{F}}(M) := \langle \{\!\{ (\mathsf{lin}_{\mathcal{F}}(\{\mathfrak{t}\}), \mathsf{depth}_{\mathcal{F}}(\mathfrak{t})) \mid \mathfrak{t} \in M \}\!\} \rangle$$

Example 6.37 (Architecture)

The μ-forest of Figure 2.20 is represented again in Figure 6.38a. The blue circles of this figure describe an independent set of 3 nodes. Its architecture is given in Figure 6.38b. Let us explain how it is built. At the root, there is no node of M in the left subtree, hence we replace this left subtree by a leaf labelled with $\mu((-1)(-1)) = \mu(1)$. The right subtree is an idempotent node whose leftmost and rightmost subtrees have no node in M. We thus replace this idempotent node by a leaf containing the multiset of the linearizations and depths of the $\mathfrak{t} \in M$.



(a) An independent set of nodes in the μ-forest from Figure 2.20.



(b) The corresponding architecture.

Figure 6.38: A set of independent nodes and its architecture.

Now we observe that the number of architectures over forests of bounded[16] height is finite.

Claim 6.39 (Finite number of architectures)

Let $\mu \colon A^* \to \mathbb{M}$ be a morphism into a finite monoid and $k \geqslant 0$. The following set is finite:

$$\mathsf{Archis}_{\mu}^{k} := \{ \mathsf{archi}_{\mathcal{F}}(M) \mid M \in \mathsf{Indep}_{\mathcal{F}}^{k}, \mathcal{F} \in \mathsf{Forests}_{\mu}^{3|\mathbb{M}|} \}.$$

14 There are no iterable nodes thus $\mathsf{Indep}_{\mathcal{F}}^{k} = \varnothing$ for $k \geqslant 1$.

15 In this case, we have $\mathcal{F}_1 \notin M$ and $\mathcal{F}_2 \notin M$ by definition of independent nodes. Therefore $M_1 \in \mathsf{Indep}_{\mathcal{F}_1}^{|M_1|}$ and $M \setminus M_1 \in \mathsf{Indep}_{\langle \mathcal{F}_2 \rangle \cdots \langle \mathcal{F}_n \rangle}^{k-|M_1|}$. This justifies that the inductive definition of $\mathsf{archi}_{\mathcal{F}}(M)$ is correct.

16 As in previous statements, we shall use the bound $3|\mathbb{M}|$ since Theorem 2.21 builds forests of this height. However, this exact bound is not useful here and Claim 6.39 also holds for any other bound.

Proof. The set Archis^k_μ consists of tree structures of height at most $3|\mathbb{M}|$, whose branching is bounded by $k+3$ and whose leaves have labels in a finite set (either elements of $\mathbb{M}$, or multisets of at most k elements of shape $(m_0 \lfloor u \rfloor m_1, d)$ with $m_0, m_1 \in \mathbb{M}$, $|u| \leqslant 2^{3|\mathbb{M}|}$ and $1 \leqslant d \leqslant 3|\mathbb{M}|$). ◀

If $\mathfrak{A} \in \mathsf{Archis}^k_\mu$, we say that this architecture has *rank* k. Observe that the rank is well-defined (i.e. $\mathsf{Archis}^k_\mu \cap \mathsf{Archis}^\ell_\mu = \varnothing$ for $k \neq \ell$) since it is the sum of the sizes of the multisets which occur in $\mathfrak{A}$.

6.4.2 Productions on architectures

Now we show that the production of a permutable k-counting transducer over a (multi)set of independent nodes M only depends on $\mathsf{archi}_\mathcal{F}(M)$. This result enables us to define the notion of *production* over an architecture, as we did for μ-k-contexts in Proposition-Definition 5.31. The proof of Proposition-Definition 6.40 is performed by induction on the structure of the architecture, and we crucially rely on the permutability of the transducer to deal with the case when this architecture consists of a multiset.

Proposition-Definition 6.40 (Productions on architectures)

Let $\mathscr{T}$ be a permutable k-counting transducer whose transition morphism is $\mu\colon A^* \to \mathbb{T}$. Let $\mathfrak{A} \in \mathsf{Archis}^k_\mu$ be an architecture, then for all $\mathcal{F} \in \mathsf{Forests}^{3|\mathbb{T}|}_\mu$ and $M \in \mathsf{Indep}^k_\mathcal{F}$ such that $\mathfrak{A} = \mathsf{archi}_\mathcal{F}(M)$, the value $\mathsf{prod}^\mathcal{F}_\mathscr{T}(M)$ is the same.

We define the *production* of $\mathscr{T}$ over the architecture $\mathfrak{A}$, denoted $\mathsf{prod}_\mathscr{T}(\mathfrak{A})$, as this value.

Proof sketch. We show that the following statement holds[17] for all $0 \leqslant x \leqslant k$ and $\mathfrak{A} \in \mathsf{Archis}^x_\mu$:

- for all $\mathcal{F}, \mathcal{F}' \in \mathsf{Forests}^{3|\mathbb{T}|}_\mu$;
- for all $M \in \mathsf{Indep}^x_\mathcal{F}$ and $M' \in \mathsf{Indep}^x_{\mathcal{F}'}$ such that $\mathsf{archi}_\mathcal{F}(M) = \mathsf{archi}_{\mathcal{F}'}(M')$;
- for all $(r, \ell) \in \mathsf{S}_{k-x}$, for all μ-$(\ell, 2^{3|\mathbb{T}|})$-iterator $\mathcal{L}$ and for all μ-$(r, 2^{3|\mathbb{T}|})$-iterator $\mathcal{R}$;

we have $\mathsf{prod}_\mathscr{T}(\mathcal{L}\ \mathsf{lin}_\mathcal{F}(M)\ \mathcal{R}) = \mathsf{prod}_\mathscr{T}(\mathcal{L}\ \mathsf{lin}_{\mathcal{F}'}(M')\ \mathcal{R})$. Proposition-Definition 6.40 follows from this statement for $x = k$, thanks to Lemma 5.48 which shows that the production on a linearization is the same as the production on the original independent set. ◀

Let us consider the statement which is claimed in the above proof sketch. The rest of Section 6.4.2 is devoted to showing this result by induction on the tree structure of $\mathfrak{A}$. We distinguish several cases (the same disjunction will be used in Section 6.4.3 for showing a different result).

6.4.2.1 Cases for $x = 0$. In this case, we have either $\mathfrak{A} = a$ or $\mathfrak{A} = \langle m \rangle$ with $m \in \mathbb{T}$. Both cases are similar and we focus on the second one, which implies that $\mu(\mathsf{word}_\mu(\mathcal{F})) = \mu(\mathsf{word}_\mu(\mathcal{F}')) = m$. Therefore we obtain $\mathcal{L}\ \mathsf{lin}_\mathcal{F}(M)\ \mathcal{R} = \mathcal{L}\ \mathsf{lin}_\mathcal{F}(M')\ \mathcal{R}$ and the result follows.

6.4.2.2 Case $\mathfrak{A} = \langle \mathfrak{A}_1 \rangle \langle \mathfrak{A}_2 \rangle \cdots \langle \mathfrak{A}_p \rangle, x \geqslant 1, \mathfrak{A}_1$ has rank $x_1 \geqslant 1$ and is not a multiset. Let us define $\mathfrak{B} := \langle \mathfrak{A}_2 \rangle \cdots \langle \mathfrak{A}_p \rangle$ which has rank $y := x - x_1$. It follows from the construction of architectures that $\mathcal{F} = \langle \mathcal{F}_1 \rangle \langle \mathcal{F}_2 \rangle \cdots \langle \mathcal{F}_n \rangle$ with $n \geqslant 1$. Let $\mathcal{G} := \langle \mathcal{F}_2 \rangle \cdots \langle \mathcal{F}_n \rangle$ and $M_1 := M \cap \mathsf{Nodes}_{\mathcal{F}_1}$. We necessarily have $\mathsf{archi}_{\mathcal{F}_1}(M_1) = \mathfrak{A}_1$ and $\mathsf{archi}_\mathcal{G}(M \smallsetminus M_1) = \mathfrak{B}$ (indeed we have $M \smallsetminus M_1 \in \mathsf{Indep}^y_\mathcal{G}$). It follows from the definition of linearizations that $\mathsf{lin}_\mathcal{F}(M) = \mathsf{lin}_{\mathcal{F}_1}(M_1)\mathsf{lin}_\mathcal{G}(M \smallsetminus M_1)$. Furthermore, $\mathsf{lin}_{\mathcal{F}_1}(M_1)$ (resp. $\mathsf{lin}_\mathcal{G}(M \smallsetminus M_1)$) is a μ-$(x_1, 2^{3|\mathbb{T}|})$-iterator (resp. a μ-$(y, 2^{3|\mathbb{T}|})$-iterator) thanks to Lemma 5.48. Similar results hold for $\mathcal{F}'$ which can be decomposed as $\langle \mathcal{F}'_1 \rangle \mathcal{G}'$ and we have:

$$\mathsf{prod}_\mathscr{T}(\mathcal{L}\ \mathsf{lin}_\mathcal{F}(M)\ \mathcal{R}) = \mathsf{prod}_\mathscr{T}(\mathcal{L}\ \mathsf{lin}_{\mathcal{F}_1}(M_1)\mathsf{lin}_\mathcal{G}(M \smallsetminus M_1)\ \mathcal{R})$$

[17] Once more, the bound $3|\mathbb{T}|$ is not useful here, but considering such μ-forests will turn out to be sufficient.

$$
\begin{aligned}
&= \mathsf{prod}_{\mathscr{T}}(\mathcal{L}\,\mathsf{lin}_{\mathcal{F}'_1}(M'_1)\mathsf{lin}_{\mathcal{G}'}(M' \smallsetminus M'_1)\,\mathcal{R}) && \text{by induction hypothesis on } \mathfrak{A}_1 \text{ and then on } \mathfrak{B};\\
&= \mathsf{prod}_{\mathscr{T}}(\mathcal{L}\,\mathsf{lin}_{\mathcal{F}'}(M')\,\mathcal{R}).
\end{aligned}
$$

6.4.2.3 Case $\mathfrak{A} = \langle \mathfrak{A}_1 \rangle \langle \mathfrak{A}_2 \rangle \cdots \langle \mathfrak{A}_p \rangle$, $x \geqslant 1$, $\mathfrak{A}_p$ has rank $x_p \geqslant 1$ and is not a multiset. This case is similar to the previous one (we consider the rightmost child instead of the leftmost one).

6.4.2.4 Remaining case: $\mathfrak{A} = \langle \mathfrak{M} \rangle$ where $\mathfrak{M}$ is a multiset. If none of the previous cases occur, we necessarily have $\mathfrak{A} = \langle \mathfrak{M} \rangle$ where $\mathfrak{M}$ is a multiset of elements $(n_0 \lfloor u \rfloor n_1, d)$ such that $|\mathfrak{M}| = x$.

Let e be the idempotent such that $e = n_0 \mu(u) n_1$ for all $(n_0 \lfloor u \rfloor n_1, d)$ of $\mathfrak{M}$ (it is necessarily the same idempotent by construction of architectures and thanks to Lemma 5.48). Furthermore, we must have $\mathcal{F} = \langle \mathcal{F}_1 \rangle \cdots \langle \mathcal{F}_n \rangle$ and $\mathcal{F}' = \langle \mathcal{F}'_1 \rangle \cdots \langle \mathcal{F}'_{n'} \rangle$ with $n, n' \geqslant 3$ and $e = \mu(\mathsf{word}_\mu(\mathcal{F}_1)) = \cdots = \mu(\mathsf{word}_\mu(\mathcal{F}_n)) = \mu(\mathsf{word}_\mu(\mathcal{F}'_1)) = \cdots = \mathsf{word}_\mu(\mathcal{F}'_n)$. Furthermore, we have $M \cap \mathsf{Nodes}_{\mathcal{F}_1} = M \cap \mathsf{Nodes}_{\mathcal{F}_n} = M' \cap \mathsf{Nodes}_{\mathcal{F}'_1} = M' \cap \mathsf{Nodes}_{\mathcal{F}'_n} = \varnothing$.

It follows from Lemma 5.48 that $\mathsf{lin}_{\mathcal{F}}(M)$ is a μ-$(x, 2^{3|\mathbb{T}|})$-iterator which has shape $e\, m_0 e_1 \lfloor u_1 \rfloor e_1 \cdots e_x \lfloor u_x \rfloor e_x m_x\, e$ and such that $\mu(m_0 e_1 \lfloor u_1 \rfloor e_1 \cdots e_x \lfloor u_x \rfloor e_x m_x) = e$. In a similar fashion, $\mathsf{lin}_{\mathcal{F}'}(M') = e\, m'_0 e'_1 \lfloor u'_1 \rfloor e'_1 \cdots e'_x \lfloor u'_x \rfloor e'_x m'_x\, e$ with $\mu(m'_0 e'_1 \lfloor u'_1 \rfloor e'_1 \cdots e'_x \lfloor u'_x \rfloor e'_x m'_x) = e$.

Now, our goal is to use the permutability of $\mathscr{T}$ in order to show that the productions are the same. For $1 \leqslant j \leqslant x$ we define $\mathsf{left}_j := e m_0 e_1 \cdots m_{j-1} e_j$ and $\mathsf{right}_j := e_j m_j \cdots e_k m_k e$, and similarly $\mathsf{left}'_j := e m'_0 e'_1 \cdots m'_{j-1} e'_j$ and $\mathsf{right}'_j := e'_j m'_j \cdots e'_k m'_k e$. It follows from the last part of Lemma 5.48 and the construction of architectures that:

$$
\mathfrak{M} = \{\!\{ (\mathsf{left}_j \lfloor u_j \rfloor \mathsf{right}_j, d_j) \mid 1 \leqslant j \leqslant x \}\!\} = \{\!\{ (\mathsf{left}'_j \lfloor u'_j \rfloor \mathsf{right}'_j, d'_j) \mid 1 \leqslant j \leqslant x \}\!\}.
$$

for some $1 \leqslant d_j, d'_j \leqslant 3|\mathbb{T}|$. Therefore there exists a permutation σ of $[1{:}x]$ such that for all $1 \leqslant j \leqslant x$, $u'_j = u_{\sigma(j)}$, $\mathsf{left}'_j = \mathsf{left}_{\sigma(j)}$ and $\mathsf{right}'_j = \mathsf{right}_{\sigma(j)}$.

By putting everything together, we are ready to show that the productions are the same:

$$
\begin{aligned}
&\mathsf{prod}_{\mathscr{T}}(\mathcal{L}\,\mathsf{lin}_{\mathcal{F}}(M)\,\mathcal{R})\\
&= \mathsf{prod}_{\mathscr{T}}(\mathcal{L}\, e\, m_0 e_1 \lfloor u_1 \rfloor e_1 \cdots e_x \lfloor u_x \rfloor e_x m_x\, e\, \mathcal{R})\\
&= \mathsf{prod}_{\mathscr{T}}\left(\mathcal{L}\ \mathsf{left}_{\sigma(1)} \lfloor u_{\sigma(1)} \rfloor \mathsf{right}_{\sigma(1)} \cdots \mathsf{left}_{\sigma(x)} \lfloor u_{\sigma(x)} \rfloor \mathsf{right}_{\sigma(x)}\ \mathcal{R}\right) && \text{since } \mathscr{T} \text{ is permutable;}\\
&= \mathsf{prod}_{\mathscr{T}}\left(\mathcal{L}\ \mathsf{left}'_1 \lfloor u'_1 \rfloor \mathsf{right}'_1 \cdots \mathsf{left}'_x \lfloor u'_x \rfloor \mathsf{right}'_x\ \mathcal{R}\right)\\
&= \mathsf{prod}_{\mathscr{T}}(\mathcal{L}\, e\, m'_0 e'_1 \lfloor u'_1 \rfloor e'_1 \cdots e'_x \lfloor u'_x \rfloor e'_x m'_x\, e\, \mathcal{R}) && \text{since } \mathscr{T} \text{ is permutable;}\\
&= \mathsf{prod}_{\mathscr{T}}(\mathcal{L}\,\mathsf{lin}_{\mathcal{F}'}(M')\,\mathcal{R}).
\end{aligned}
$$

This result concludes the inductive proof of Proposition-Definition 6.40.

6.4.3 Counting the number of architectures

We have shown in Proposition-Definition 6.40 that the production of a permutable k-counting transducer over an independent multiset of nodes only depends on its architecture. Since the number of architectures is finite by Claim 6.39, this result enables to rewrite the function $\mathsf{sum\text{-}ind}_{\mathscr{T}}$ as follows:

$$
\mathsf{sum\text{-}ind}_{\mathscr{T}}(\mathcal{F}) = \sum_{M \in \mathsf{Indep}^k_{\mathcal{F}}} \mathsf{prod}^{\mathcal{F}}_{\mathscr{T}}(M)
$$

$$= \sum_{\mathfrak{A} \in \mathsf{Archis}_\mu^k} \sum_{\substack{M \in \mathsf{Indep}_{\mathcal{F}}^k \\ \mathsf{archi}_{\mathcal{F}}(M) = \mathfrak{A}}} \mathsf{prod}_{\mathscr{T}}^{\mathcal{F}}(M) \tag{6.41}$$

$$= \sum_{\mathfrak{A} \in \mathsf{Archis}_\mu^k} \mathsf{prod}_{\mathscr{T}}(\mathfrak{A}) \times \mathsf{count}_{\mathfrak{A}}(\mathcal{F})$$

where $\mathsf{count}_{\mathfrak{A}}(\mathcal{F}) := |\{M \in \mathsf{Indep}_{\mathcal{F}}^k : \mathsf{archi}_{\mathcal{F}}(M) = \mathfrak{A}\}|$. It describes the number of independent (multi)sets of nodes which have architecture $\mathfrak{A}$. Now we show how to compute this function as a sum of a $\mathbb{N}$-polyblind function and a $\mathbb{N}$-polyregular function with lower growth.

Observe that the functions $\mathsf{count}_{\mathfrak{A}}$ no longer depend on the productions of $\mathscr{T}$. Furthermore, thanks to Equation (6.41), it is sufficient to show Proposition 6.35 for the functions $\mathsf{count}_{\mathfrak{A}}$ with $\mathfrak{A} \in \mathsf{Archis}_\mu^k$. This is the purpose of Lemma 6.42. Recall that $\mathbb{N}\mathsf{poly}_{-1}$ only contains the null function.

Lemma 6.42 (Counting architectures)

Let $\mu\colon A^* \to \mathbb{M}$ be a monoid morphism and $k \geqslant 0$. Given $\mathfrak{A} \in \mathsf{Archis}_\mu^k$, one can build:

- a function $\mathsf{count}'_{\mathfrak{A}} : (A \cup \{\langle, \rangle\})^* \to \mathbb{N} \in \mathbb{N}\mathsf{blind}_k$;
- a function $\mathsf{count}''_{\mathfrak{A}} : (A \cup \{\langle, \rangle\})^* \to \mathbb{N} \in \mathbb{N}\mathsf{poly}_{k-1}$;

such that $\mathsf{count}_{\mathfrak{A}}(\mathcal{F}) = \mathsf{count}'_{\mathfrak{A}}(\mathcal{F}) + \mathsf{count}''_{\mathfrak{A}}(\mathcal{F})$ for all $\mathcal{F} \in \mathsf{Forests}_\mu^{3|\mathbb{T}|}$.

Proof sketch. The two functions are built simultaneously by a rather involved induction on the structure of $\mathfrak{A}$. The inductive case disjunction is similar to that of Section 6.4.2. ◀

The rest of Section 6.4.3 is devoted to the detailed inductive proof of Lemma 6.42. Since $\mathsf{Forests}_\mu^{3|\mathbb{T}|}$ is a regular language of $(A \cup \{\langle, \rangle\})^*$, one can assume that the input always belongs to this set.

6.4.3.1 Cases for $k = 0$.

In this case we have either $\mathfrak{A} = a$ or $\mathfrak{A} = \langle m \rangle$ for $m \in \mathbb{M}$, as observed in Section 6.4.2.2. Let us assume that $\mathfrak{A} = \langle m \rangle$, in this case $\mathsf{count}_{\mathfrak{A}}(\mathcal{F}) = 1$ if $\mathcal{F} = \langle \mathcal{F}_1 \rangle \cdots \langle \mathcal{F}_n \rangle$ with $n \geqslant 1$ and $\mu(\mathsf{word}_\mu(\mathcal{F})) = m$ and 0 otherwise. Hence $\mathsf{count}_{\mathfrak{A}}$ is the indicator function of a regular language and therefore it belongs to $\mathbb{N}\mathsf{poly}_0$. We let $\mathsf{count}'_{\mathfrak{A}} := \mathsf{count}_{\mathfrak{A}}$ and $\mathsf{count}''_{\mathfrak{A}} := u \mapsto 0$.

6.4.3.2 Case $\mathfrak{A} = \langle \mathfrak{A}_1 \rangle \langle \mathfrak{A}_2 \rangle \cdots \langle \mathfrak{A}_p \rangle$, $k \geqslant 1$, $\mathfrak{A}_1$ has rank $k_1 \geqslant 1$ and is not a multiset.

Let us define the architecture $\mathfrak{B} := \langle \mathfrak{A}_2 \rangle \cdots \langle \mathfrak{A}_p \rangle$ which has rank $\ell := k - k_1$.

Claim 6.43 (Counting product)

$$\mathsf{count}_{\mathfrak{A}}(\mathcal{F}) = \begin{cases} 0 \text{ if } \mathcal{F} \text{ is not of the form } \langle \mathcal{F}_1 \rangle \langle \mathcal{F}_2 \rangle \cdots \langle \mathcal{F}_n \rangle \text{ with } n \geqslant 1 \\ \mathsf{count}_{\mathfrak{A}_1}(\mathcal{F}_1) \times \mathsf{count}_{\mathfrak{B}}(\langle \mathcal{F}_2 \rangle \cdots \langle \mathcal{F}_n \rangle) \text{ otherwise.} \end{cases}$$

Proof. If $\mathsf{archi}_{\mathcal{F}}(M) = \langle \mathfrak{A}_1 \rangle \mathfrak{B}$, then $\mathcal{F} = \langle \mathcal{F}_1 \rangle \langle \mathcal{F}_2 \rangle \cdots \langle \mathcal{F}_n \rangle$ with $n \geqslant 1$ and furthermore $M \cap \mathsf{Nodes}_{\mathcal{F}_1} \neq \varnothing$. If $\mathcal{F}$ has this shape, let $\mathcal{G} := \langle \mathcal{F}_2 \rangle \cdots \langle \mathcal{F}_n \rangle$. With these notations, we have:

$$\begin{aligned} &|\{M \in \mathsf{Indep}_{\mathcal{F}}^k \mid \mathsf{archi}_{\mathcal{F}}(M) = \mathfrak{A}\}| \\ &= |\{(M_1, M_2) \mid M_1 \in \mathsf{Indep}_{\mathcal{F}_1}^{k_1}, \mathsf{archi}_{\mathcal{F}_1}(M_1) = \mathfrak{A}_1 \text{ and } M_2 \in \mathsf{Indep}_{\mathcal{G}}^{\ell}, \mathsf{archi}_{\mathcal{G}}(M_2) = \mathfrak{B}\}|. \end{aligned}$$

Indeed $M \mapsto (M \cap \mathsf{Nodes}_{\mathcal{F}_1}), (M \cap \mathsf{Nodes}_{\mathcal{G}})$ is a bijection between these two sets. ◀

By applying Claim 6.43 and using the functions which were built by induction hypothesis, we get for all $\mathcal{F} = \langle \mathcal{F}_1 \rangle \langle \mathcal{F}_2 \rangle \cdots \langle \mathcal{F}_n \rangle$ with $n \geqslant 1$ and $\mathcal{G} := \langle \mathcal{F}_2 \rangle \cdots \langle \mathcal{F}_n \rangle$:

$$\begin{aligned}
\mathsf{count}_{\mathfrak{A}}(\mathcal{F}) &= \left(\mathsf{count}'_{\mathfrak{A}_1}(\mathcal{F}_1) + \mathsf{count}''_{\mathfrak{A}_1}(\mathcal{F}_1)\right)\left(\mathsf{count}'_{\mathfrak{B}}(\mathcal{G}) + \mathsf{count}''_{\mathfrak{B}}(\mathcal{G})\right)\\
&= \underbrace{\mathsf{count}'_{\mathfrak{A}_1}(\mathcal{F}_1)\mathsf{count}'_{\mathfrak{B}}(\mathcal{G})}_{=:\mathsf{count}'_{\mathfrak{A}}(\mathcal{F})}\\
&\quad + \underbrace{\mathsf{count}'_{\mathfrak{A}_1}(\mathcal{F}_1)\mathsf{count}''_{\mathfrak{B}}(\mathcal{G}) + \mathsf{count}''_{\mathfrak{A}_1}(\mathcal{F}_1)\mathsf{count}'_{\mathfrak{B}}(\mathcal{G}) + \mathsf{count}''_{\mathfrak{A}_1}(\mathcal{F}_1)\mathsf{count}''_{\mathfrak{B}}(\mathcal{G})}_{=:\mathsf{count}''_{\mathfrak{A}}(\mathcal{F})}.
\end{aligned}$$

Now let us justify that the definitions of $\mathsf{count}'_{\mathfrak{A}}$ and $\mathsf{count}''_{\mathfrak{A}}$ verify our properties. We first observe that the functions $f_1 : \langle \mathcal{F}_1 \rangle \mathcal{G} \mapsto \mathsf{count}'_{\mathfrak{A}_1}(\mathcal{F}_1)$ (resp. $f_2 : \langle \mathcal{F}_1 \rangle \mathcal{G} \mapsto \mathsf{count}'_{\mathfrak{B}}(\mathcal{G})$) belongs to $\mathbb{N}\mathsf{blind}_{k_1}$ (resp. $\mathbb{N}\mathsf{blind}_{\ell}$). Indeed, since the input is assumed to have bounded height, a blind k_1-counting transducer (resp. by a blind ℓ-counting transducer) can detect the $\rangle$ which matches the first $\langle$, and simulate the computation of $\mathsf{count}'_{\mathfrak{A}_1}$ (resp. $\mathsf{count}'_{\mathfrak{B}}$) on $\mathcal{F}_1$ (resp. $\mathcal{G}$). Hence $\mathsf{count}'_{\mathfrak{A}} = f_1 \times f_2$ belongs to $\mathbb{N}\mathsf{blind}_k$ by Lemma 6.10. We show in a similar way, thanks to Lemma 5.19, that $\mathsf{count}''_{\mathfrak{A}}$ belongs to $\mathbb{S}\mathsf{poly}_{k-1}$.

6.4.3.3 Case $\mathfrak{A} = \langle \mathfrak{A}_1 \rangle \langle \mathfrak{A}_2 \rangle \cdots \langle \mathfrak{A}_p \rangle$, $k \geqslant 1$, $\mathfrak{A}_p$ has rank $k_p \geqslant 1$ and is not a multiset.

This case is similar to the previous one (we consider the rightmost child instead of the leftmost one).

6.4.3.4 Remaining case: $\mathfrak{A} = \langle \mathfrak{M} \rangle$ where $\mathfrak{M}$ is a multiset.

Recall that $k = |\mathfrak{M}|$ and let e be the idempotent such that $e = m_0 \mu(u) m_1$ for all $(m_0 \boxed{u} m_1, d) \in \mathfrak{M}$. Given $\mathcal{F} \in \mathsf{Forests}_{\mu}$ and a (multi)set $M \in \mathsf{Indep}^k_{\mathcal{F}}$, we define the multiset[18] $\mathsf{Contexts}_{\mathcal{F}}(M) := \{\!\{ (\mathsf{lin}_{\mathcal{F}}(\{\mathfrak{t}\}), \mathsf{depth}_{\mathcal{F}}(\mathfrak{t})) \mid \mathfrak{t} \in M \}\!\}$. Intuitively, it describes the information that a blind k-counting transducer can observe about M. We have:

$$\mathsf{count}_{\langle \mathfrak{M} \rangle}(\mathcal{F}) = \begin{cases} 0 \text{ if } \mathcal{F} \text{ does not have shape } \langle \mathcal{F}_1 \rangle \cdots \langle \mathcal{F}_n \rangle \\ \qquad \text{with } n \geqslant 3 \text{ and } \mu(\mathcal{F}_1) = \cdots = \mu(\mathcal{F}_n) = e; \\ \left|\{M \in \mathsf{Indep}^k_{\mathcal{F}} \mid M \subseteq \mathsf{Nodes}_{\langle \mathcal{F}_2 \rangle \cdots \langle \mathcal{F}_{n-1} \rangle} \text{ and } \mathsf{Contexts}_{\mathcal{F}}(M) = \mathfrak{M}\}\right| \\ \qquad \text{otherwise.} \end{cases}$$

From now, we assume the input $\mathcal{F}$ has shape $\langle \mathcal{F}_1 \rangle \cdots \langle \mathcal{F}_n \rangle$ where $\mu(\mathcal{F}_1) = \cdots = \mu(\mathcal{F}_n) = e$ and $n \geqslant 3$ (this is a regular property which can be checked by a blind k-counting transducer).

The construction of $\mathsf{count}'_{\langle \mathfrak{M} \rangle}$ and $\mathsf{count}''_{\langle \mathfrak{M} \rangle}$ is performed simultaneously by another induction on $|\mathfrak{M}| = k$ thanks to Lemma 6.44. This result explains how to remove the occurrences of a given element $(m_0 \boxed{u} m_1, d)$ of $\mathfrak{M}$ whose depth d is minimal[19] among the depths of the other elements of $\mathfrak{M}$. Given a multiset $\mathfrak{M}$, we write $\tau \in \mathfrak{M}$ to denote that $\mathfrak{M}$ contains at least one occurrence of τ, i.e. $|\mathfrak{M}|_{\tau} \geqslant 1$.

Lemma 6.44 (Removing one μ-1-context)

Let $\tau = (m_0 \boxed{u} m_1, d)$. Assume that $\mathfrak{M} := \mathfrak{M}_1 \uplus \{\!\{ \tau \ddagger r \}\!\}$ for some $r > 0$, that $\tau \not\in \mathfrak{M}_1$ and for all $(m'_0 \boxed{u'} m'_1, d') \in \mathfrak{M}_1$, one has $d \leqslant d'$. Then one can build $g' \in \mathbb{N}\mathsf{blind}_r$ and $g'' \in \mathbb{N}\mathsf{poly}_{k-1}$ such that $\mathsf{count}_{\langle \mathfrak{M} \rangle}(\mathcal{F}) = g'(\mathcal{F}) \times \mathsf{count}_{\langle \mathfrak{M}_1 \rangle}(\mathcal{F}) + g''(\mathcal{F})$.

Proof. In order to simplify the notations, we shall assume that neither $\mathcal{F}_1$ nor $\mathcal{F}_n$ have iterable nodes, therefore $\mathsf{count}_{\langle \mathfrak{M} \rangle}(\mathcal{F}) = |\{M \in \mathsf{Indep}^k_{\mathcal{F}} \mid \mathsf{Contexts}_{\mathcal{F}}(M) = \mathfrak{M}\}|$. Let $k_1 := k - r \geqslant 0$ (this way $|\mathfrak{M}_1| = k_1$) and $\mathsf{Candidates}(\mathcal{F}) := \{\mathfrak{t} \in \mathsf{Iters}_{\mathcal{F}} \mid \mathsf{Contexts}_{\mathcal{F}}(\{\mathfrak{t}\}) = \{\!\{\tau\}\!\}\}$.

[18] Observe that this multiset may have multiplicities even if M is a set.

[19] The reason why we consider an element with minimal depth is for using the fact that a node only observes a bounded number of nodes thanks to (as stated in Claim 2.31). A similar argument was already used in Chapter 3.

We first observe that $\mathsf{count}_{\langle\mathfrak{M}\rangle}$ can be decomposed when fixing its $\mathfrak{M}_1$ part:

$$\mathsf{count}_{\langle\mathfrak{M}\rangle}(\mathcal{F}) = \sum_{\substack{M_1 \in \mathsf{Indep}^{k_1}_{\mathcal{F}} \\ \mathsf{Contexts}_{\mathcal{F}}(M_1) = \mathfrak{M}_1}} |\{M_2 \subseteq \mathsf{Candidates}(\mathcal{F}) \mid M_1 \cup M_2 \in \mathsf{Indep}^{k}_{\mathcal{F}}\}| \tag{6.45}$$

Indeed $M \mapsto (M \cap \{\mathfrak{t} \mid \mathsf{Contexts}_{\mathcal{F}}(\{\mathfrak{t}\}) \neq \{\!\{\tau\}\!\}\}, M \cap \{\mathfrak{t} \mid \mathsf{Contexts}_{\mathcal{F}}(\{\mathfrak{t}\}) = \{\!\{\tau\}\!\}\})$ is a bijection between the set of sets $\{M \in \mathsf{Indep}^{k}_{\mathcal{F}} \mid \mathsf{Contexts}_{\mathcal{F}}(M) = \mathfrak{M}\}$ and the set of couples of sets $\{(M_1, M_2) \mid \mathsf{Contexts}_{\mathcal{F}}(M_1) = \mathfrak{M}_1, M_2 \subseteq \mathsf{Candidates}(\mathcal{F})$ and $M_1 \cup M_2 \in \mathsf{Indep}^{k}_{\mathcal{F}}\}$.

The construction of g' and g'' will depend on whether $|\mathsf{Candidates}(\mathcal{F})| < 3k_1 + 2r$ or not. This condition is a regular property of $\mathcal{F}$, thus it can be checked by a blind counting transducer.

First case: if $|\mathsf{Candidates}(\mathcal{F})| < 3k_1 + 2r$. We define $g'(\mathcal{F}) := 0$ and $g''(\mathcal{F}) := \mathsf{count}_{\langle\mathfrak{M}\rangle}(\mathcal{F})$. Indeed, it is easy to see that the function g'' is computed by a k-counting transducer which ranges over k-tuples of iterable nodes of $\mathcal{F}$. Furthermore, $g''(\mathcal{F}) = \mathcal{O}(|\mathcal{F}|^{k_1})$ thanks to Equation (6.45) since for a given M_1, by hypothesis there is only a bounded number of sets $M_2 \subseteq \mathsf{Candidates}(\mathcal{F})$ such that $M_1 \cup M_2 \in \mathsf{Indep}^{k}_{\mathcal{F}}$. Therefore $g'' \in \mathbb{N}\mathsf{poly}_{k_1}$ by applying Theorem 5.25 and thus $g'' \in \mathbb{N}\mathsf{poly}_{k-1}$ since $k_1 < k$.

Second case: if $|\mathsf{Candidates}(\mathcal{F})| \geqslant 3k_1 + 2r$. This case is more complex. Given $M_1 \in \mathsf{Indep}^{k_1}_{\mathcal{F}}$ such that $\mathsf{Contexts}_{\mathcal{F}}(M_1) = \mathfrak{M}_1$, we define:

$$\mathsf{Candidates}_{M_1}(\mathcal{F}) := \{\mathfrak{t} \in \mathsf{Candidates}(\mathcal{F}) \mid \{\mathfrak{t}\} \cup M_1 \in \mathsf{Indep}^{k_1+1}_{\mathcal{F}}\}$$

Now we show that this set only removes a bounded number of nodes from $\mathsf{Candidates}(\mathcal{F})$.

Claim 6.46 ($\mathsf{Candidates}_{M_1}(\mathcal{F})$ is nearly $\mathsf{Candidates}(\mathcal{F})$)

If $M_1 \in \mathsf{Indep}^{k_1}_{\mathcal{F}}$ is such that $\mathsf{Contexts}_{\mathcal{F}}(M_1) = \mathfrak{M}_1$, then:

$$|\mathsf{Candidates}(\mathcal{F}) \smallsetminus \mathsf{Candidates}_{M_1}(\mathcal{F})| \leqslant 3k_1.$$

Proof. The nodes of $\mathsf{Candidates}(\mathcal{F})$ have depth d, which is $\leqslant$ than the depths of the nodes of M_1. Therefore $\mathsf{Candidates}(\mathcal{F}) \smallsetminus \mathsf{Candidates}_{M_1}(\mathcal{F})$ is the set of nodes of $\mathsf{Candidates}(\mathcal{F})$ that some node from M_1 observes. There are at most $3|M_1| = 3k_1$ such nodes. Indeed, even the nodes of M_1 may observe up to $3|\mathbb{T}||M_1|$ nodes (recall Claim 2.31), there are at most $3|M_1|$ such nodes of depth exactly d. ◀

Recall that $\preccurlyeq$ is a total ordering defined by $\mathfrak{t}' \preccurlyeq \mathfrak{t}'$ if and only if $\min(\mathsf{Fr}_{\mathcal{F}}(\mathfrak{t})) \leqslant \min(\mathsf{Fr}_{\mathcal{F}}(\mathfrak{t}'))$. Let $\prec$ be the appropriate strict ordering. Let $\mathsf{Firsts}_{M_1}(\mathcal{F})$ denote the set containing the first $3k_1 - |\mathsf{Candidates}(\mathcal{F}) \smallsetminus \mathsf{Candidates}_{M_1}(\mathcal{F})| \geqslant 0$ elements of $\mathsf{Candidates}_{M_1}(\mathcal{F})$ (with respect to $\preccurlyeq$) and similarly let $\mathsf{Lasts}_{M_1}(\mathcal{F}) := \mathsf{Candidates}_{M_1}(\mathcal{F}) \smallsetminus \mathsf{Firsts}_{M_1}(\mathcal{F})$. Observe that[20] $|\mathsf{Lasts}_{M_1}(\mathcal{F})| = |\mathsf{Candidates}(\mathcal{F})| - 3k_1 \geqslant 2r$ and furthermore that this value *does not* depend on M_1, which will be a key argument in the following[21].

We say that two nodes $\mathfrak{t} \prec \mathfrak{t}' \in \mathsf{Lasts}_{M_1}(\mathcal{F})$ are *neighbors* if there is no $\mathfrak{t}'' \in \mathsf{Lasts}_{M_1}(\mathcal{F})$ such that $\mathfrak{t} \prec \mathfrak{t}'' \prec \mathfrak{t}'$. For all $M_1 \in \mathsf{Indep}^{k_1}_{\mathcal{F}}$, one can decompose the set which defines the

[20] By using the inclusions between the various sets, it is easy to show that:

$$\begin{aligned} |\mathsf{Lasts}_{M_1}(\mathcal{F})| &= |\mathsf{Candidates}_{M_1}(\mathcal{F})| - |\mathsf{Firsts}_{M_1}(\mathcal{F})| \\ &= |\mathsf{Candidates}_{M_1}(\mathcal{F})| - 3k_1 + |\mathsf{Candidates}(\mathcal{F})| - |\mathsf{Candidates}_{M_1}(\mathcal{F})| \\ &= |\mathsf{Candidates}(\mathcal{F})| - 3k_1 \geqslant 2r. \end{aligned}$$

[21] Intuitively, we are looking for independence between M_1 and M_2 in order to obtain a Hadamard product.

function $\mathsf{count}_{\langle\mathfrak{M}\rangle}(\mathcal{F})$ in Equation (6.45) as follows:

$$\begin{aligned}
&\{M_2 \subseteq \mathsf{Candidates}(\mathcal{F}) \mid M_1 \cup M_2 \in \mathsf{Indep}^k_{\mathcal{F}}\} \\
&= \{M_2 \subseteq \mathsf{Candidates}_{M_1}(\mathcal{F}) \mid M_1 \cup M_2 \in \mathsf{Indep}^k_{\mathcal{F}}\} \\
&= \{M_2 \subseteq \mathsf{Lasts}_{M_1}(\mathcal{F}) \mid M_1 \cup M_2 \in \mathsf{Indep}^k_{\mathcal{F}} \text{ and no } t, t' \in M_2 \text{ are neighbors}\} \\
&\uplus \{M_2 \subseteq \mathsf{Lasts}_{M_1}(\mathcal{F}) \mid M_1 \cup M_2 \in \mathsf{Indep}^k_{\mathcal{F}} \text{ and there exist neighbors } t, t' \in M_2\} \\
&\uplus \{M_2 \subseteq \mathsf{Candidates}_{M_1}(\mathcal{F}) \mid M_1 \cup M_2 \in \mathsf{Indep}^k_{\mathcal{F}} \text{ and } M_2 \cap \mathsf{Firsts}_{M_1}(\mathcal{F}) \neq \varnothing\}.
\end{aligned} \tag{6.47}$$

Let us define the function $P_r : \mathbb{N} \to \mathbb{N}$ which maps $X \geqslant 0$ to the cardinal of the set W of words $w \in \{0,1\}^*$ such that $|w| = X$, $|w|_1 = r$ and there are no two consecutive 1 in w.

Claim 6.48 (P_r computes an approximation)

If $M_1 \in \mathsf{Indep}^{k_1}_{\mathcal{F}}$ is such that $\mathsf{Contexts}_{\mathcal{F}}(M_1) = \mathfrak{M}_1$, then $P_r(|\mathsf{Candidates}(\mathcal{F})| - 3k_1) = |\{M_2 \subseteq \mathsf{Lasts}_{M_1}(\mathcal{F}) \mid M_1 \cup M_2 \in \mathsf{Indep}^k_{\mathcal{F}}$ and no $t, t' \in M_2$ are neighbors$\}|$.

Proof. First note that $\{M_2 \subseteq \mathsf{Lasts}_{M_1}(\mathcal{F}) \mid M_1 \cup M_2 \in \mathsf{Indep}^k_{\mathcal{F}}$ and no $t, t' \in M_2$ are neighbors$\} = \{M_2 \subseteq \mathsf{Lasts}_{M_1}(\mathcal{F}) \mid$ no $t, t' \in M_2$ are neighbors$\}$. Indeed, two nodes of the second set cannot be dependent by definition of neighbors, hence the condition $M_1 \cup M_2 \in \mathsf{Indep}^k_{\mathcal{F}}$ always holds. Recall that $|\mathsf{Lasts}_{M_1}(\mathcal{F})| = |\mathsf{Candidates}(\mathcal{F})| - 3k_1$. Finally, we observe that the definition of P_r corresponds to neighbors. ◀

Now, the key observation is that $P_r(|\mathsf{Candidates}(\mathcal{F})| - 3k_1)$ *does not* depend on M_1. Therefore it can be computed "independently" from the set M_1 which was chosen. Furthermore, this function is a polynomial in $\mathsf{Candidates}(\mathcal{F})$, therefore it is $\mathbb{N}$-polyblind.

Claim 6.49 (Polyblind part)

Let $g'(\mathcal{F}) := P_r(|\mathsf{Candidates}(\mathcal{F})| - 3k_1)$, then $g' \in \mathbb{N}\mathsf{blind}_r$.

Proof. The function which maps $w \in W$ to itself where each 10 factor (excepted the last one) is replaced by 1, is a bijection between W and $\{w \in \{0,1\}^{X-r+1} \mid |w|_r = 1\}$. Hence $P_r(X) = \binom{X-r+1}{r} = \frac{(X-r+1)!}{r!(X-2r+1)!}$, and therefore we obtain:

$$g'(\mathcal{F}) = \frac{1}{r!} \prod_{i=0}^{r-1} (|\mathsf{Candidates}(\mathcal{F})| - 3k_1 - r - i + 1).$$

It follows from Lemma 6.10 that $r! \times g' \in \mathbb{N}\mathsf{blind}_r$. Finally, dividing by $r!$ can be seen as the post-composition by a sequential function (with both unary input and output alphabets), which is still in $\mathbb{N}\mathsf{blind}_r$ thanks to Theorem 3.6. ◀

If we denote by $c_{M_1}(\mathcal{F})$ the cardinal of the two last terms of Equation (6.47), we get:

$$\mathsf{count}_{\langle\mathfrak{M}\rangle}(\mathcal{F}) = g'(\mathcal{F}) \times \mathsf{count}_{\langle\mathfrak{M}_1\rangle}(\mathcal{F}) + \underbrace{\sum_{\substack{M_1 \in \mathsf{Indep}^{\mathcal{F}}_{k_1} \\ \mathsf{Contexts}_{\mathcal{F}}(M_1) = \mathfrak{M}_1}} c_{M_1}(\mathcal{F})}_{=: g''(\mathcal{F})}.$$

It is easy to show that $g'' \in \mathbb{N}\mathsf{poly}_k$ (it can be computed by ranging over k-tuples of iterable nodes. Furthermore, $g''(\mathcal{F}) = \mathcal{O}(|\mathcal{F}|^{k-1})$: the intuition is that it describes sets M_2 which have one less degree of freedom. Therefore $g'' \in \mathbb{N}\mathsf{poly}_{k-1}$ by Theorem 5.25. ◀

We conclude the case of Section 6.4.3.4 by applying inductively Lemma 6.44.

6.4.4 Decomposing the independent sum

We are ready to conclude the proof of Proposition 6.35. Given a permutable k-counting transducer $\mathscr{T}$ with transition morphism $\mu\colon A^* \to \mathbb{T}$, we define the desired functions as follows:

$$\begin{aligned}\mathsf{sum\text{-}ind}'_{\mathscr{T}} &:= \sum_{\mathfrak{A} \in \mathsf{Archis}^k_\mu} \mathsf{prod}_{\mathscr{T}}(\mathfrak{A}) \times \mathsf{count}'_{\mathfrak{A}} \\ \mathsf{sum\text{-}ind}''_{\mathscr{T}} &:= \sum_{\mathfrak{A} \in \mathsf{Archis}^k_\mu} \mathsf{prod}_{\mathscr{T}}(\mathfrak{A}) \times \mathsf{count}''_{\mathfrak{A}}.\end{aligned}$$

It follows from Equation (6.41) and Lemma 6.42 that $\mathsf{sum\text{-}ind}_{\mathscr{T}} = \mathsf{sum\text{-}ind}'_{\mathscr{T}} + \mathsf{sum\text{-}ind}''_{\mathscr{T}}$. Furthermore, these statements also justify that $\mathsf{sum\text{-}ind}'_{\mathscr{T}} \in \mathbb{S}\mathsf{blind}_k$ and $\mathsf{sum\text{-}ind}''_{\mathscr{T}} \in \mathbb{S}\mathsf{poly}_{k-1}$.

6.5 Solving the $\mathbb{S}$-polyblind membership problem

This section is devoted to concluding the proof of Theorem 6.17 (it will directly follow from the more precise Theorem 6.51), by leveraging the tools introduced in Sections 6.3 and 6.4.

We first recall that a function computed by a permutable k-counting transducer can be decomposed as the sum of a function of $\mathbb{S}\mathsf{blind}_k$ and a function of $\mathbb{S}\mathsf{poly}_{k-1}$, which intuitively captures the terms whose "degree" is not maximal. Lemma 6.50 can be seen as an analogue of Lemma 5.53.

Lemma 6.50 (Permutable $\Rightarrow$ Blind + term of lower degree)

Let $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$ and $k \geqslant 1$. Given a function $f : A^* \to \mathbb{S}$ computed by a permutable k-counting transducer $\mathscr{T}$ whose transition morphism is $\mu : A^* \to \mathbb{T}$, one can decompose it as follows:

$$f = (\mathsf{sum\text{-}dep}_{\mathscr{T}} + \mathsf{sum\text{-}ind}'_{\mathscr{T}} + \mathsf{sum\text{-}ind}''_{\mathscr{T}}) \circ \mathsf{forest}_\mu$$

where furthermore $\mathsf{sum\text{-}ind}'_{\mathscr{T}} \in \mathbb{S}\mathsf{blind}_k$ and $\mathsf{sum\text{-}dep}_{\mathscr{T}} + \mathsf{sum\text{-}ind}''_{\mathscr{T}} \in \mathbb{S}\mathsf{poly}_{k-1}$.

Proof. Combine Propositions 5.42 and 6.35 and Lemma 5.43. ◀

We are ready to show Theorem 6.51, which originates from [Dou22, Theorem 5.1]. This result is obtained by induction on $k \geqslant 1$ by using the fact that since $\mathsf{sum\text{-}dep}_{\mathscr{T}} + \mathsf{sum\text{-}ind}''_{\mathscr{T}} \in \mathbb{S}\mathsf{poly}_{k-1}$, one can decide by induction hypothesis whether this function of "lower degree" is $\mathbb{S}$-polyblind. As mentioned in the introduction of Chapter 6, equivalence between repetitive and $\mathbb{S}$-polyblind functions is not only a nice consequence of this proof, but also a key ingredient to show the induction step. Indeed, we crucially rely on the fact that repetitiveness is preserved under subtractions.

Theorem 6.51 (Induction step for $\mathbb{S}$-polyregular $\to$ $\mathbb{S}$-polyblind)

Let $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$ and $k \geqslant 1$. Let $f\colon A^* \to \mathbb{S}$ be computed by a k-counting transducer with output monoid $\mathbb{S}$. The following conditions are equivalent:

(1) f is $\mathbb{S}$-polyblind;
(2) f is k-repetitive;
(3) $\mathscr{T}$ is permutable and $(\mathsf{sum\text{-}dep}_{\mathscr{T}} + \mathsf{sum\text{-}ind}''_{\mathscr{T}}) \circ \mathsf{forest}_\mu \in \mathbb{S}\mathsf{blind}_{k-1}$;

(4) f is computed by a blind k-counting transducer (i.e. $f \in \mathbb{S}\mathsf{blind}_k$).

Furthermore this property is decidable and the construction is effective.

Proof. The proof of this result is performed by induction on $k \geqslant 1$. Item (4) $\Rightarrow$ Item (1) is obvious. Item (1) $\Rightarrow$ Item (2) is exactly Lemma 6.24. Item (3) $\Rightarrow$ Item (4) follows from Lemma 6.50 and Proposition 6.5 for precomposing the sums with the regular function forest_μ.

The subtle point is Item (2) $\Rightarrow$ Item (3). To show it we first apply Lemma 6.29 to show that $\mathscr{T}$ is permutable. Now let $g := (\mathsf{sum\text{-}dep}_{\mathscr{T}} + \mathsf{sum\text{-}ind}''_{\mathscr{T}}) \circ \mathsf{forest}_\mu$, then $g \in \mathbb{S}\mathsf{poly}_{k-1}$ by Lemma 6.50 and Proposition 5.7. Furthermore Lemma 6.50 also shows that $\mathsf{sum\text{-}ind}'_{\mathscr{T}} \in \mathbb{S}\mathsf{blind}_k$, therefore $\mathsf{sum\text{-}ind}'_{\mathscr{T}} \circ \mathsf{forest}_\mu \in \mathbb{S}\mathsf{blind}_k$ by Proposition 6.5. Hence this function is k-repetitive by Lemma 6.24. Since $g = f - \mathsf{sum\text{-}ind}''_{\mathscr{T}} \circ \mathsf{forest}_\mu$ and the function f is k-repetitive, it follows from Claim 6.23 that g is k-repetitive and therefore $(k-1)$-repetitive. Since $g \in \mathbb{S}\mathsf{poly}_{k-1}$, one can apply Item (2) $\Rightarrow$ Item (4) by induction hypothesis[22] and therefore we get $g \in \mathbb{S}\mathsf{blind}_{k-1}$.

Decidability is obtained thanks to Item (3): one can decide if $\mathscr{T}$ is permutable and by induction hypothesis one can decide if $(\mathsf{sum\text{-}dep}_{\mathscr{T}} + \mathsf{sum\text{-}ind}''_{\mathscr{T}}) \circ \mathsf{forest}_\mu \in \mathbb{S}\mathsf{blind}_{k-1}$. ◀

In Chapter 7, we shall use similar proof techniques (based on jumping inductively between semantic and syntactic conditions) to decide whether a $\mathbb{Z}$-polyregular function is star-free $\mathbb{Z}$-polyregular. However, the overall structure of this proof will be quite different: we shall build a canonical model which describes $\mathbb{Z}$-polyregular functions, which is not the case in Chapter 6.

[22] The reader is invited to gaze in admiration at this argument. Indeed, as mentioned above, using the robustness of a *semantic* condition here becomes a key and nearly magical technique to prove the desired result by induction.

Chapter 7

Star-free polyregular functions with commutative output

> De la musique avant toute chose,
> Et pour cela préfère l'Impair
> Plus vague et plus soluble dans l'air,
> Sans rien en lui qui pèse ou qui pose.
>
> Paul Verlaine, « Art poétique », *Jadis et Naguère*

The class of regular languages contains a celebrated subclass of independent interest named *star-free languages*, which has been studied since the early days of automata theory. This robust subclass admits several equivalent descriptions in terms of automata, logics, regular expressions and algebra. When coming to membership problems, one can decide if a regular language is star-free by effectively constructing its *minimal automaton* (or equivalently, its syntactic monoid) which is a canonical object describing the language, and checking if this machine has a (decidable) *aperiodicity* property.

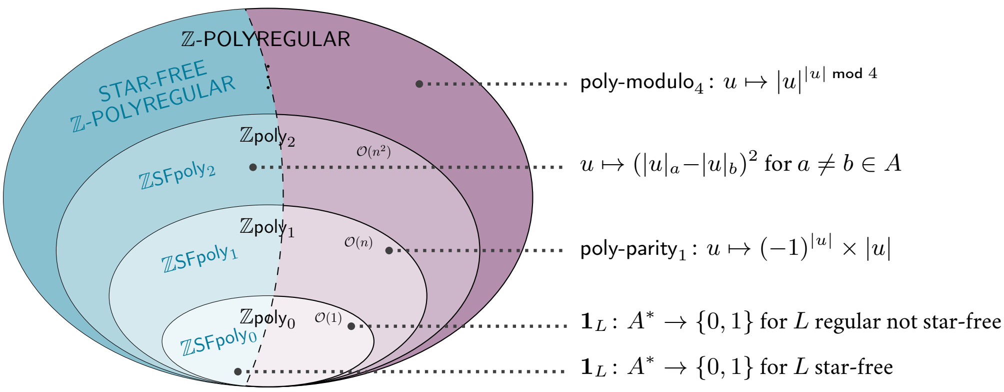


Figure 7.1: Classes of $\mathbb{Z}$-polyregular functions studied in Chapter 7.

The notion of star-freeness has been shifted from languages to functions, yielding e.g. classes of *star-free regular functions* or *star-free polyregular functions*. Intuitively, they are obtained by forbidding transducers to check "counting modulo" properties of their input. Multiple characterizations of these classes in terms of transducers and logics have been obtained over the past 10 years. However, the related class membership problems are still open in general and considered as difficult. The goal of Chapter 7 is to define a robust class of *star-free* $\mathbb{S}$*-polyregular functions* and show that one can decide if a $\mathbb{Z}$-polyregular function is star-free $\mathbb{Z}$-polyregular. The classes are depicted in Figure 7.1.

In Section 7.1 we introduce the class of star-free $\mathbb{S}$-polyregular functions and provide a description of this class in terms of *aperiodic* counting transducers and as a natural subclass of $\mathbb{S}$-rational series. These results are mere adaptations of those of Chapter 5 for $\mathbb{S}$-polyregular functions in general.

The goal of Section 7.2 is to state the main result of Chapter 7, that is the decidability of the membership problem from $\mathbb{Z}$-polyregular to star-free $\mathbb{Z}$-polyregular. As in Chapter 5 with repetitiveness, we introduce a semantic condition named *smoothness* and show that it characterizes star-free $\mathbb{Z}$-polyregular functions among the $\mathbb{Z}$-polyregular ones. This result has several low hanging consequences. In particular, it enables to easily build separating examples between the two classes (see Figure 7.1). Furthermore, it yields an optimization result for star-free $\mathbb{Z}$-polyregular functions.

The proof of the membership result from $\mathbb{Z}$-polyregular to star-free $\mathbb{Z}$-polyregular goes over Sections 7.3 to 7.5. It proceeds by induction as the proof of Chapter 6 does and uses smoothness as a key tool for the induction step. However, a major difference with Chapter 6 is that we show in Section 7.3 that given a $\mathbb{Z}$-polyregular function, it is possible to build canonical objects named the *residual transducers* of this function. These machines are inspired by the *residual automaton* of a regular language. We then show that star-freeness faithfully translates to an *aperiodicity* syntactic property for residual transducers.

In Section 7.6, we develop other characterizations of (star-free) $\mathbb{Z}$-polyregular functions by means of *eigenvalues* of matrices in $\mathbb{Z}$-weighted automata. Finally, we discuss in Section 7.7 why it seems hard to generalize the constructions of this chapter to the membership problems for star-free $\mathbb{N}$-polyregular functions or even (word-to-word) star-free regular functions. However, we conjecture that using semantic characterizations may still be relevant in this setting, at the cost of a combinatorial effort.

The contributions presented in this chapter are based on the results of [CDL23].

7.1 Star-free polyregular functions with commutative output

The class of *star-free languages* is a robust subclass of regular languages obtained by forbidding Kleene star in regular expressions (but allowing complementations instead). The study of this class goes back to Schützenberger's celebrated theorem [Sch65] which characterizes star-free languages as those whose syntactic monoid is *aperiodic*[1], which implies that star-freeness is decidable. The class of star-free languages enjoys other equivalent descriptions, e.g. in terms of first-order logics (see [MP71]). Furthermore, a great number of subclasses of star-free languages have been studied (see e.g. [Pin84]).

The notion of aperiodicity can be shifted from monoids to machines, as explained in Definition 7.2.

Definition 7.2 (Aperiodicity)

A machine is said to be *aperiodic* whenever its transition monoid is aperiodic.

Following Definition 7.2, Schützenberger's theorem can be reformulated by saying that a language is star-free if and only if it can be computed by some finite automaton which is *aperiodic*, or equivalently if the *minimal automaton* of this language is so. From this point of view, it is thus very natural to define functional counterparts of star-free languages by starting from aperiodic transducers. In particular, the

[1] Recall that a monoid $\mathbb{M}$ is *aperiodic* if there exists $\Omega \geqslant 0$ such that $m^{\Omega} = m^{\Omega+1}$ for all $m \in \mathbb{M}$.

class of *star-free regular functions*[2] is defined as the class of word-to-word functions which are computed by *aperiodic* 2DT. This class has been explored in detail and equivalent descriptions in terms of logics (thanks to *first-order transductions*, a restriction of the MSO transductions from Section 1.2.4.2) [FKT14, CD15, DJR18], or basic combinators [BDK18, DGK21] have been obtained.

Checking that a 2DT is aperiodic can be tough in practice, but the main intuition is that such a machine is not able to build the output depending on "counting modulo" properties of its input.

Example 7.3 (Map copy reverse)

The function map-copy-reverse is star-free regular.

Example 7.4 (Polynomial modulo)

If $m, n \geqslant 1$, we let $a \text{ mod } b$ be the remainder of the integer division of m by n. The function $u \mapsto 1^{|u|^{|u| \text{ mod } 2}}$ is regular but not star-free regular (see Example 7.20).

One of the current challenges in the theory of regular functions would be to derive analogues of Schützenberger's theorem for transductions. However, it is not known[3] whether canonical models can be built for regular functions, therefore Open question 7.5 is believed to be hard.

Open question 7.5 (Regular $\to$ Star-free regular)

Given a regular function, can we decide if it is star-free regular?

As mentioned after Proposition 1.16, Open question 7.5 is nevertheless known to be decidable in the restricted setting of rational functions thanks to [FGL19, Corollary 5.6]. The proof relies on the construction of a canonical bimachine which computes a given rational function.

7.1.1 Aperiodic pebble transducers

We say that a pebble transducer or a marble transducer[4] is *aperiodic* whenever all its submachines are aperiodic. The class of functions computed by aperiodic pebble transducers is said to be the class of *star-free polyregular functions*. Basic properties of this class, including closure under composition, are studied in [Boj18]. Furthermore, an equivalent description in terms of logics (thanks to *first-order interpretations*, a restriction of the MSO interpretations of Section 1.3.3.2) is shown in [BKL19, Theorem 7].

Example 7.6 (Squaring functions)

The functions blind-square, square and inner-squaring are star-free polyregular.

As in the rest of Part II, our goal in Chapter 7 is to focus on functions which have output in a commutative monoid $(\mathbb{S}, +)$. The class star-free $\mathbb{S}$-polyregular functions is defined by adapting Definition 5.2 which builds $\mathbb{S}$-polyregular functions from the polyregular ones.

[2] Even if the terminology star-free regular seems to be redundant, it is necessary to avoid ambiguity with other "star-free" classes of functions such as star-free polyregular functions.

[3] It is however known how to build canonical models for regular functions with origin semantics [Boj14].

[4] The definition would be less obvious for recursive marble transducers. Indeed, the author conjectures that the structure of recursive calls should also be taken into account to define a notion of aperiodicity for this model.

Definition 7.7 (Star-free polyregular functions)

The class of *star-free* $\mathbb{S}$*-polyregular functions* is the class of functions of shape $\mathsf{sum} \circ g \colon A^* \to \mathbb{S}$ where $g : A^* \to \mathbb{S}^*$ is star-free polyregular and $\mathsf{sum} \colon \mathbb{S}^* \to \mathbb{S}$ is the sum operation in $\mathbb{S}$.

We denote by $\mathbb{S}\mathsf{SFpoly}$ the class of star-free $\mathbb{S}$-polyregular functions. More precisely, for all $k \geqslant 1$, we denote by $\mathbb{S}\mathsf{SFpoly}_k$ the class of functions of shape $\mathsf{sum} \circ g \colon A^* \to \mathbb{S}$ where the function $g \colon A^* \to \mathbb{S}^*$ is computed by an aperiodic k-pebble transducer. We let $\mathbb{S}\mathsf{SFpoly}_0$ be the class of functions $f \colon A^* \to \mathbb{S}$ whose image $f(A^*)$ is finite and such that $f^{-1}(\{\delta\})$ is a star-free language for all $\delta \in \mathbb{S}$. We also let $\mathbb{S}\mathsf{SFpoly}_{-1}$ be the singleton set which contains the constant function $u \mapsto 0$.

Example 7.8 (Counting letters)

The function $\mathsf{nb}_{a_1,\dots,a_k} : u \mapsto |u|_{a_1} \times \cdots |u|_{a_k}$ belongs to $\mathbb{N}\mathsf{SFpoly}_k$.

Observe that when $\mathbb{S}$ is finite, we may not have[5] $\mathbb{S}\mathsf{SFpoly}_0 = \mathbb{S}\mathsf{SFpoly}$ (contrary to Claim 5.6 which states that an analogue result holds for $\mathbb{S}\mathsf{poly}$). However this result holds if $\mathbb{S}$ is both finite and aperiodic. In Chapter 7 we shall mainly focus on the case $\mathbb{S} := \mathbb{Z}$ when solving membership problems.

7.1.2 Aperiodic counting transducers

Following Definition 7.2, we say that a k-counting transducer is *aperiodic* if its transition monoid is so. If A is an alphabet, we denote by $\mathsf{SFProp}_k(A)$ the set of star-free languages over $A \times \{0,1\}^k$. Observe that if $L \in \mathsf{SFProp}_k(A)$ then in particular[6] $L \in \mathsf{RegProp}_k(A)$. It is easy to see that an aperiodic k-counting transducer[7] has shape $(A, \mathbb{S}, (\delta_i, L_i)_{1 \leqslant i \leqslant n})$ where $L_i \in \mathsf{SFProp}_k(A)$ for $1 \leqslant i \leqslant n$.

Example 7.9 (Map power)

For all $k \geqslant 0$, the function $\mathsf{map\text{-}power}_k \colon 0^{n_1}\# \cdots \# 0^{n_m} \mapsto \sum_{i=1}^{m} n_i^k$ can be computed by an aperiodic k-counting transducer.

Unsurprisingly, we show that the class of functions computed by aperiodic counting transducers with output $\mathbb{S}$ is exactly the class of star-free $\mathbb{S}$-polyregular functions. Theorem 7.10 is an analogue of Theorem 5.15 and it maintains the connection with marble transducers.

Theorem 7.10 (Aperiodic pebble = Aperiodic marble = Aperiodic counting)

Let $\mathbb{S}$ be a commutative monoid. Given $f : A^* \to \mathbb{S}$ and $k \geqslant 1$, the following are equivalent:

(1) $f = \mathsf{sum} \circ g$ for $g : A^* \to \mathbb{S}^*$ computed by an aperiodic k-pebble transducer;
(2) $f = \mathsf{sum} \circ g$ for $g : A^* \to \mathbb{S}^*$ computed by an aperiodic k-marble transducer;
(3) f is computed by an aperiodic k-counting transducer.

The conversions are effective.

[5] Indeed if $\mathbb{S} = (\mathbb{Z}/2\mathbb{Z}, +)$, the pre-image of $\{0\}$ under $\mathsf{sum} \circ (u \mapsto 1^{|u|})$ is the set of words of even length, which is not a star-free language. This case is however artificial since we have created periodicity thanks to the output monoid.

[6] Given a language $L \in \mathsf{SFProp}_k(A)$, one can build a *first-order formula* (FO formula for short) $\varphi(x_1, \dots, x_k)$ where $x_1, \dots, x_k$ are free first-order variables, such that $\#L(u)$ is the number of assignments $x_1, \dots, x_k$ which make φ true in the model $u \in A^*$ (see e.g. [Tho97]). As for $\mathsf{RegProp}_k(A)$, we chose to use the formalism of languages instead.

[7] In [CDL23], the aperiodic counting transducers are built by using FO formulas with free variables instead of languages of SFProp. They can therefore be seen as a particular case of the FO *interpretations* from [BKL19]. Once more, we chose not to use this equivalent formalism, since we never deal with logic in this manuscript.

Proof idea. We follow *mutatis mutandis* the proof of Theorem 5.15, while checking that aperiodicity is preserved at each step. The only tricky point is that when showing Item (3) $\Rightarrow$ Item (2), the original proof of Claim 5.16 builds a marble transducer whose submachines use lookarounds, which can be removed thanks to Theorem 1.30. In the current setting, we obtain a simple marble transducer whose submachines only use lookarounds to check the belonging of the marked input to star-free languages. One has to ensure that such *star-free lookarounds* can be removed while preserving both aperiodicity and origin semantics, which is done e.g. in [CD15, Theorem 20]. ◀

7.1.3 Star-free $\mathbb{S}$-polyregular functions as $\mathbb{S}$-rational series

Now we intend to characterize the class of star-free $\mathbb{S}$-polyregular functions as a natural subclass of $(\mathbb{S}, +, \times)$-rational series for $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$. This section is a mere adaptation of Section 5.2.

We first give an analogue of Lemmas 5.19 and 5.21. The single difference with these previous results is that we replace indicator functions of regular languages by those of star-free languages.

Lemma 7.11 (Closure properties of star free $\mathbb{S}$-polyregular functions)

Let $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$. The class of star-free $\mathbb{S}$-polyregular functions is closed under Hadamard products and Cauchy products. More precisely, if $f \in \mathbb{S}\mathsf{SFpoly}_k$ and $g \in \mathbb{S}\mathsf{SFpoly}_\ell$, then $f \otimes g \in \mathbb{S}\mathsf{SFpoly}_{k+\ell+1}$ and $f \times g \in \mathbb{S}\mathsf{SFpoly}_{k+\ell}$.

Furthermore, for all $k \geqslant 0$, the following equality holds and the conversions are effective:

$$\mathbb{S}\mathsf{SFpoly}_{k+1} = \mathsf{Span}_{\mathbb{S}}(\{\mathbf{1}_L \otimes f \mid L \text{ star-free language}, f \in \mathbb{S}\mathsf{SFpoly}_k\}).$$

Proof idea. We follow *mutatis mutandis* the proofs of Lemmas 5.19 and 5.21. ◀

Example 7.12 (Counting letters)

The function $\mathsf{nb}_a : u \mapsto |u|_a$ belongs to $\mathbb{Z}\mathsf{SFpoly}_1$ and it can be written as $\mathbf{1}_{A^*a} \otimes \mathbf{1}_{A^*}$. In a similar way, the function $\mathsf{nb}_{a,b} \in \mathbb{Z}\mathsf{SFpoly}_2$ can be written as $\mathbf{1}_{A^*a} \otimes \mathbf{1}_{A^*} \otimes \mathbf{1}_{bA^*} + \mathbf{1}_{A^*b} \otimes \mathbf{1}_{A^*} \otimes \mathbf{1}_{aA^*}$.

Finally, let us state Theorem 7.13 which is an analogue of Theorem 5.22. This result justifies the "star-free" terminology for star-free $\mathbb{Z}$-polyregular functions.

Theorem 7.13 (Star-free $\mathbb{S}$-polyregular functions as $\mathbb{S}$-rational series)

Let $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$. A function $f : A^* \to \mathbb{S}$ is star-free $\mathbb{S}$-polyregular if and only if it belongs to smallest class of functions of type $A^* \to \mathbb{S}$ containing the indicator functions of star-free languages and closed under external products, sums and Cauchy products.

Proof idea. We leverage Lemma 7.11. ◀

7.2 Membership problem for star-free $\mathbb{Z}$-polyregular functions

The goal of this section is to state the main result of Chapter 7, which claims that one can decide if a $\mathbb{Z}$-polyregular function is star-free. We also provide a semantic condition called *smoothness* which characterizes the star-free $\mathbb{Z}$-polyregular functions. As in the proof of Chapter 6, this characterization will turn out to be a key ingredient in the proof of the decidability result.

7.2.1 Smooth functions

We first introduce the notion of *smoothness* for $\mathbb{S}$-polyregular functions. It will serve as a semantic characterization of star-freeness, in the same way as repetitiveness for $\mathbb{S}$-polyblind functions in Chapter 6.

Recall from Schützenberger's theorem that a regular language $L \subseteq A^*$ is star-free if and only if its syntactic monoid is aperiodic. In other words, there must exist $\Omega \geqslant 1$ such that for all $v, u, w \in A^*$ either $vu^X w \in L$ for all $X \geqslant \Omega$ or $vu^X w \notin L$ for all $X \geqslant \Omega$. This result is reformulated in Claim 7.14.

Claim 7.14 (Indicator functions of star-free languages)

A regular language $L \subseteq A^*$ is star-free if and only if there exists $\Omega \geqslant 1$ such that for all $v, u, w \in A^*$ the function $X \mapsto \mathbf{1}_L(vu^X w)$ is constant for $X \geqslant \Omega$.

Our main goal is to extend Claim 7.14 to star-free $\mathbb{Z}$-polyregular functions. Recall from Example 7.8 that the function $\mathsf{nb}_{a,b} \colon u \mapsto |u|_{a_1} \times |u|_b$ is star-free $\mathbb{Z}$-polyregular. The function $X \mapsto \mathsf{nb}_{a,b}(vu^X w) = X^2|u|_a|u|_b + X(|vw|_a|u|_b + |vw|_b|u|_a) + |vw|_a|vw|_b$ is not ultimately constant, but it is a polynomial in X, which roughly means that no periodic behavior occurs when iterating u. This is the main intuition behind Definition 7.15 which originates from [CDL23, Definition II.29][8].

Definition 7.15 (Smooth function)

Let $k \geqslant 1$. A function $f \colon A^* \to \mathbb{Z}$ is said to be *k-smooth* if there exists $\Omega \geqslant 0$ such that for all $v_0, u_1, v_1, \dots, u_k, v_k \in A^*$, the function $X_1, \dots, X_k \mapsto f(v_0 u_1^{X_1} \alpha_1 \cdots u_k^{X_k} v_k)$ is a polynomial for $X_1, \dots, X_k \geqslant \Omega$.

Example 7.16 (Counting letters)

The function $\mathsf{nb}_{a_1,\dots,a_k} \colon u \mapsto |u|_{a_1} \times \cdots \times |u|_{a_k}$ is ℓ-smooth for all $\ell \geqslant 1$.

Example 7.17 (Polynomial modulo)

Let $k \geqslant 1$ and $A = \{a\}$. The function $\mathsf{poly\text{-}modulo}_k \colon u \mapsto |u|^{|u| \bmod k}$ is not 1-smooth. Indeed, $\mathsf{poly\text{-}modulo}_k(a^X) = X^{X \bmod k}$ is not a polynomial, even for X large enough.

Example 7.18 (Polynomial parity)

For all $k \geqslant 0$, the function $\mathsf{poly\text{-}parity}_k \colon u \mapsto (-1)^{|u|} \times |u|^k$ is not 1-smooth. Indeed we have $\mathsf{poly\text{-}parity}_k(a^X) = (-1)^X X^k$ which is not a polynomial, even for X large enough.

7.2.2 Decidability result of star-free inside $\mathbb{Z}$-polyregular

Now we are ready to decide and characterize the star-free functions among the $\mathbb{Z}$-polyregular ones. Theorem 7.19 originates from [CDL23, Theorem V.8] and its proof goes over Sections 7.3 to 7.5.

As mentioned in the beginning of Chapter 7, the proof of Theorem 7.19 does not rely on factorization forests but on building canonical objects for $\mathbb{Z}$-polyregular functions. Indeed, the function forest_μ

[8]The terminology used in [CDL23] is *ultimately 1-polynomial* instead of smooth. We chose to modify it here here in order to prevent confusion with the polynomials theirselves and since the term smooth better conveys the absence of periodic behaviors.

from Theorem 2.21 is regular, but it has no reason to be star-free regular in general[9], thus doing a pre-composition by this function is not relevant in our setting (contrary to Chapter 6).

Theorem 7.19 ($\mathbb{Z}$-polyregular $\to$ Star-free $\mathbb{Z}$-polyregular)

A function $f \in \mathbb{Z}\mathsf{poly}_k$ is star-free $\mathbb{Z}$-polyregular if and only if it is $(k+1)$-smooth. This property is decidable. If it holds, one can build an aperiodic k-counting transducer which computes f.

Proof sketch. We first show that given a function $f \in \mathbb{Z}\mathsf{poly}_k$, one can build nearly[10] canonical machines which compute f, called its *k-residual transducers*. The construction of such machines is inspired by the residual automaton of a regular language, which is well-known to reveal informations on the semantic properties of the language. A k-residual transducer can also be seen as a variant of a marble transducer which calls functions of $\mathbb{Z}\mathsf{poly}_{k-1}$ on suffixes of its input.

The main proof is done by induction on $k \geqslant 1$, by showing that if f is $(k+1)$-smooth then its k-residual transducer has an *aperiodicity* property and that furthermore it calls functions of $\mathbb{Z}\mathsf{poly}_{k-1}$ which are k-smooth (therefore they belong to $\mathbb{Z}\mathsf{SFpoly}_{k-1}$ by induction hypothesis). We then recombine these elements to show that f belongs to $\mathbb{Z}\mathsf{SFpoly}_k$. For decidability, we rely on the fact that aperiodicity is decidable. Formally, Theorem 7.19 follows from Theorem 7.54. ◀

We shall see in Section 7.6 that 1-smoothness turns out to be sufficient to characterize star-freeness. However, the author is not aware of a way to adapt these result for $\mathbb{N}$-polyregular functions. The obstacles towards a generalization are discussed in detail in Section 7.7.

Let us provide low hanging consequences of Theorem 7.19. By leveraging Example 7.18, we first provide in Example 7.20 separating examples between $\mathbb{Z}$-polyregular and star-free functions.

Example 7.20 (Polynomial modulo and parity)

The functions $\mathsf{poly\text{-}modulo}_k \colon u \mapsto |u|^{|u| \bmod k}$ and $\mathsf{poly\text{-}parity}_k \colon u \mapsto (-1)^{|u|} \times |u|^k$ are not 1-smooth, as shown in Examples 7.17 and 7.18. Hence they are respectively $\mathbb{N}$-polyregular and $\mathbb{Z}$-polyregular, but neither star-free $\mathbb{N}$-polyregular nor star-free $\mathbb{Z}$-polyregular.

We also observe that Theorems 5.25 and 7.19 provide an optimization result for star-free functions. Corollary 7.21 is an analogue of Corollary 6.22 which was shown for $\mathbb{S}$-polyblind functions.

Corollary 7.21 (Optimization of aperiodic pebble transducers with commutative output)

Let $f \in \mathbb{Z}\mathsf{SFpoly}$ and $k \geqslant 0$, then $f \in \mathbb{Z}\mathsf{SFpoly}_k$ if and only if $|f(u)| = \mathcal{O}(|u|^k)$. This property is decidable. If it holds, one can build an aperiodic k-counting transducer computing f.

Proof. Let $f \in \mathbb{Z}\mathsf{SFpoly}$ be such that $|f(u)| = \mathcal{O}(|u|^k)$. We get $f \in \mathbb{S}\mathsf{poly}_k$ by Theorem 5.25 and furthermore f is $(k+1)$-smooth by Theorem 7.19. Thus one can build an aperiodic k-counting transducer which computes f by Theorem 7.19. The converse is obvious. ◀

Remark 7.22 (Relation with the results of [DG19])

[DG19] introduces a notion of *aperiodicity* for $\mathbb{Z}$-weighted automata which defines a notion of

[9] Consider e.g. the forests obtained with the morphism $\mu \colon \{a\}^* \to \mathbb{Z}/2\mathbb{Z}$ such that $\mu(a) = 1$.

[10] There may be several k-residual transducers, but building any of them will be sufficient for our proof.

star-free $\mathbb{Z}$-rational series. However, this class is radically different from star-free $\mathbb{Z}$-polyregular functions: the functions poly-parity or $\mathbf{1}_{\text{even}}$ are considered as star-free in this paper.

7.3 Residual transducers for $\mathbb{Z}$-polyregular functions

The goal of this section is to show that any $\mathbb{Z}$-polyregular function can be computed by canonical objects called its *residual transducers*. The construction of these machines is built upon the construction of the residual automaton of a regular language. It is therefore a good candidate for expliciting information about the intrinsic properties of the function, and in particular its star-freeness.

In Section 7.3.1 we lift the well-known notion of *residual* from languages to functions. Then we introduce in Section 7.3.2 the model of *suffix deterministic transducer*, which can be seen as a very particular case of counting transducers. We finally show in Section 7.3.3 how, given a $\mathbb{Z}$-polyregular function, it is possible to build somehow canonical *suffix deterministic transducers* called *residual transducers*.

7.3.1 Residuals of a function

Our first goal is to lift the classical notion of *residual* from languages of A^* to functions of type $A^* \to \mathbb{S}$. It is well-known that a language is regular if and only if it has a finite number of residuals. Furthermore, the residuals of a regular language describe its intrinsic behavior since they are connected to its syntactic monoid through the *residual automaton* (see e.g. [Car14, Section 1.7]).

Formally, given $L \subseteq A^*$ and $u \in A^*$, the *residual language* $u^{-1}L$ is defined as $\{w \in A^* \mid uw \in L\}$. Our easy generalization to functions originates from [CDL23, Definition IV.1]. Similar definitions are presented in [Boj14, Section 2.1] when dealing with regular functions in origin semantics.

Definition 7.23 (Residual)

Let $\mathbb{S}$ be a commutative monoid, $f\colon A^* \to \mathbb{S}$ and $u \in A^*$. The *residual function* $u \triangleright f\colon A^* \to \mathbb{S}$ is defined as $w \mapsto f(uw)$. We let $\mathsf{Res}(f) := \{u \triangleright f \mid u \in A^*\}$ be the set of *residuals* of f.

It is easy to see that $u \triangleright \mathbf{1}_L = \mathbf{1}_{u^{-1}L}$ hence both notions coincide when dealing with languages. In particular, the set $\mathsf{Res}(\mathbf{1}_L)$ is finite if and only if L is regular. However, it is easy to observe that neither $\mathbb{S}$-polyregular functions nor $\mathbb{S}$-rational series have a finite number of residuals in general.

Example 7.24 (Residuals)

The residuals of the function $u \mapsto |u|^2 \in \mathbb{N}\mathsf{poly}_2$ are the functions $u \mapsto |u|^2 + 2n|u| + n^2$ for $n \geqslant 0$. The residuals of the function $u \mapsto (-2)^{|u|}$ are the functions $u \mapsto (-2)^{n+|u|}$ for $n \geqslant 0$.

Now we show in Claim 7.25 that $u \mapsto u \triangleright f$ defines a monoid action of A^* over $A^* \to \mathbb{S}$, which (effectively) preserves the classes of functions $\mathbb{S}\mathsf{poly}_k$ for $k \geqslant -1$.

Claim 7.25 (Residuals preserve $\mathbb{S}\mathsf{poly}_k$)

Let $k \geqslant -1$, $f \in \mathbb{S}\mathsf{poly}_k$ and $u \in A^*$, then $u \triangleright f \in \mathbb{S}\mathsf{poly}_k$. The construction is effective.

Proof. The function $g\colon w \mapsto uw$ is regular, thus $u \triangleright f = f \circ g \in \mathbb{S}\mathsf{poly}_k$ by Proposition 5.7. ◀

Jump to contents

From now on we focus on the case $\mathbb{S} := \mathbb{Z}$. We intend to show that if $f \in \mathbb{Z}\mathsf{poly}_k$ for some $k \geqslant 0$ the set $\mathsf{Res}(f)$ is finite up to identifying the functions whose difference is in $\mathbb{Z}\mathsf{poly}_{k-1}$. In order to formalize this identification, we first define the equivalence relations $\sim_k$ for $k \geqslant -1$.

Definition 7.26 ($\mathbb{Z}\mathsf{poly}_k$ equivalence)

Given $k \geqslant -1$ and $f, g \colon A^* \to \mathbb{Z}$, we let $f \sim_k g$ if and only if $f - g \in \mathbb{Z}\mathsf{poly}_k$

We observe that $\sim_k$ is compatible with $\triangleright$ and with the regular combinators which build $\mathbb{Z}\mathsf{poly}$.

Claim 7.27 (Properties of $\sim_k$)

For all $k \geqslant -1$, $\sim_k$ is an equivalence relation. Furthermore, the following holds for all $u \in A^*$, $L \subseteq A^*$, $\delta \in \mathbb{Z}$ and $f, f', g, g' \colon A^* \to \mathbb{Z}$:

(1) if $f \sim_k g$, then $u \triangleright f \sim_k u \triangleright g$ and $\delta \cdot f \sim_k \delta \cdot g$;
(2) if $f \sim_k g$ and $f' \sim_k g'$ then $f + f' \sim_k g + g'$;
(3) if $f \in \mathbb{Z}\mathsf{poly}_k$, then $u \triangleright (\mathbf{1}_L \otimes f) \sim_k (u \triangleright \mathbf{1}_L) \otimes f$.

Proof. The fact that $\sim_k$ is an equivalence relation is obvious from the properties of $\mathbb{Z}\mathsf{poly}_k$. For Item (1) assume that $f \sim_k g$, then $f - g \in \mathbb{Z}\mathsf{poly}_k$ and so $u \triangleright f - u \triangleright g = u \triangleright (f - g) \in \mathbb{Z}\mathsf{poly}_k$ by Claim 7.25. Therefore $u \triangleright f \sim_k u \triangleright g$ and $\delta \cdot f \sim_k \delta \cdot g$ is obvious. Item (2) is trivial. For Item (3), we proceed by induction on $|u|$. Indeed $a \triangleright (\mathbf{1}_L \otimes f) = (a \triangleright \mathbf{1}_L) \otimes f + \mathbf{1}_L(\varepsilon) \times (a \triangleright f)$ for all $a \in A$, therefore we obtain $a \triangleright (\mathbf{1}_L \otimes f) \sim_k (a \triangleright \mathbf{1}_L) \otimes f$ when $f \in \mathbb{Z}\mathsf{poly}_k$. ◀

By leveraging Claim 7.27, we obtain Lemma 7.28 which provides a finite abstaction of residuals.

Lemma 7.28 (Finite residuals up to $\sim_{k-1}$)

Let $k \geqslant 0$ and $f \in \mathbb{Z}\mathsf{poly}_k$, then the quotient set $\mathsf{Res}(f)/\sim_{k-1}$ is finite.

Proof. We first note that $u \triangleright (\delta \cdot f + \eta \cdot g) = \delta \cdot (u \triangleright f) + \eta \cdot (u \triangleright g)$ for all $f, g : A^* \to \mathbb{Z}$, $\delta, \eta \in \mathbb{Z}$ and $u \in A^*$. Hence it suffices to show that Lemma 7.28 holds on a set S of functions such that $\mathsf{Span}_{\mathbb{Z}}(S) = \mathbb{Z}\mathsf{poly}_k$. For $k = 0$, we chose $S := \{\mathbf{1}_L \mid L \text{ regular}\}$ and the result is clear since regular languages have finitely many residuals. For $k \geqslant 1$, we use Lemma 5.21 and choose $S := \{\mathbf{1}_L \otimes g \mid g \in \mathbb{Z}\mathsf{poly}_{k-1}, L \text{ regular}\}$. If $\mathbf{1}_L \otimes g \in S$, then by Claim 7.27 we get $u \triangleright (\mathbf{1}_L \otimes g) \sim_{k-1} (u \triangleright \mathbf{1}_L) \otimes g = \mathbf{1}_{u^{-1}L} \otimes g$. Since the regular language L has finitely many residuals, there are finitely many $\sim_{k-1}$-equivalence classes for the residual functions of $\mathbf{1}_L \otimes g$. ◀

We shall see in Corollary 7.44 that the implication of Lemma 7.28 turns out to be an equivalence. Observe for the moment that Lemma 7.28 does not hold for $\mathbb{Z}$-rational series. Indeed, Example 7.24 exhibits the $\mathbb{Z}$-rational series $f \colon u \mapsto (-2)^{|u|}$ such that $\mathsf{Res}(f)/\sim_k$ is infinite for all $k \geqslant -1$.

Finally, we note that $\sim_k$ is decidable for $\mathbb{Z}$-polyregular functions.

Lemma 7.29 (Decidability of $\sim_k$)

Given $k \geqslant -1$ and $f, g \in \mathbb{Z}\mathsf{poly}$, one can decide whether $f \sim_k g$ holds.

Proof. Let us first recall that $f \sim_{-1} g$ if and only if $f = g$, which is decidable thanks to Corollary 5.24. For $k \geqslant 0$, the result follows from Theorem 5.25. ◀

7.3.2 Suffix deterministic transducers

Given a function $f \in \mathbb{Z}\mathsf{poly}_k$, our goal is to build in Section 7.3.3 a transducer for f whose states are based on the finite set $\mathsf{Res}(f)/{\sim_{k-1}}$ (in the spirit of the residual automaton for regular languages). Using inductively this construction will build a somehow canonical object describing f.

In Section 7.3.2, we first introduce the model of *$\mathfrak{F}$-suffix deterministic transducer*, which originates from [CDL23, Definition IV.11] (under the less explicit name of *$\mathfrak{F}$-transducer*). It consists in a one-way deterministic automaton which can call functions from a class $\mathfrak{F}$ on suffixes of its input. This machine can roughly be seen as a head of a marble transducer[11], with the key differences that two-way moves are forbidden and that nested calls are performed on suffixes of the input.

Definition 7.30 (Suffix deterministic transducer)

Let $\mathfrak{F}$ be a set of functions[12] which have type $A^* \to \mathbb{Z}$. A *$\mathfrak{F}$-suffix deterministic transducer* ($\mathfrak{F}$-SDT) $\mathscr{T} = (A, Q, q_0, \delta, \mathfrak{F}, \lambda, F)$ consists of:

- an input alphabet A;
- a finite set of states Q with an initial state $q_0 \in Q$;
- a transition function $\delta \colon Q \times A \to Q$;
- an output function $\lambda \colon Q \times A \to \mathfrak{F}$;
- a final output function $F \colon Q \to \mathbb{Z}$.

Let us describe the semantics of a $\mathfrak{F}$-SDT. Given $q \in Q$, we define by induction on $u \in A^*$ the value $[\![\mathscr{T}]\!]_q(u) \in \mathbb{Z}$. For $u = \varepsilon$, we let $[\![\mathscr{T}]\!]_q(\varepsilon) := F(q)$. Otherwise for $u \in A^*$ and $a \in A$ we let $[\![\mathscr{T}]\!]_q(au) := [\![\mathscr{T}]\!]_{\delta(q,a)}(u) + \lambda(q,a)(u)$. Finally, the function computed by the $\mathfrak{F}$-SDT $\mathscr{T}$ is defined as $[\![\mathscr{T}]\!] := [\![\mathscr{T}]\!]_{q_0} \colon A^* \to \mathbb{Z}$. Observe that all the functions $[\![\mathscr{T}]\!]_q$ are total.

The *extended transition function* δ^* of $\mathscr{T}$ is defined as usual by $\delta^*(q, ua) := \delta(\delta^*(q,u), a)$ and $\delta^*(q, \varepsilon) = q$. Using this notation, observe that $[\![\mathscr{T}]\!]_q(u) = \sum_{vaw=u} \lambda(\delta^*(q,v),a)(w) + F(\delta^*(q,u))$. In other words, if $L_q := \{u \in A^* \mid \delta^*(q_0,u) = q\}$ for all $q \in Q$, we have:

$$[\![\mathscr{T}]\!](u) = \sum_{\substack{q \in Q\\ a \in A}} \mathbf{1}_{L_q a} \otimes \lambda(q,a) + \sum_{q \in Q} F(q) \cdot \mathbf{1}_{L_q} \otimes \mathbf{1}_{\{\varepsilon\}}. \tag{7.31}$$

Thus a $\mathfrak{F}$-SDT is more or less performing Cauchy products of shape $\mathbf{1}_L \otimes f$ for $f \in \mathfrak{F}$.

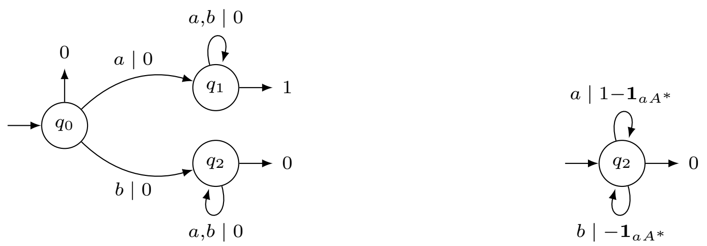
Example 7.32 (Suffix deterministic transducers)

We have depicted in Figure 7.33a a $\mathbb{Z}\mathsf{poly}_{-1}$-$\mathsf{SDT}$ which computes the indicator function $\mathbf{1}_{aA^*}$ for $A = \{a,b\}$. Since $\mathbb{Z}\mathsf{poly}_{-1} = \{0\}$, the output is only determined by the final state. Observe that this machine can be identified with the residual automaton of aA^*.

In Figure 7.33b we have depicted a $\mathbb{Z}\mathsf{poly}_0$-$\mathsf{SDT}$ which also computes $\mathbf{1}_{aA^*}$. It has a single state and "hides" its computation into the calls to functions of $\mathbb{Z}\mathsf{poly}_0$. One can check for instance that $1 = \mathbf{1}_{aA^*}(aab) = (1 - \mathbf{1}_{aA^*}(ab)) + (1 - \mathbf{1}_{aA^*}(b)) - \mathbf{1}_{aA^*}(\varepsilon) + 0$.

[11] However, this model is somehow orthogonal to that of blind pebble transducers, since making calls on suffixes implicitly means that the calling position is visible. Hence the author believes that the proofs of Chapter 7 do not provide relevant techniques for solving the membership problems towards $\mathbb{S}$-polyblind functions which were discussed Chapter 6.

[12] The set $\mathfrak{F}$ is not assumed to be finite. However, since Q and A are so, $\delta(Q \times A)$ is always a finite subset of $\mathfrak{F}$.



(a) A $\mathbb{Z}\mathsf{poly}_{-1}$-SDT computing $\mathbf{1}_{aA^*}$. (b) A $\mathbb{Z}\mathsf{poly}_0$-SDT computing $\mathbf{1}_{aA^*}$.

Figure 7.33: Two suffix deterministic transducers computing $\mathbf{1}_{aA^*}$.

7.3.3 Residual transducers

Now, we are ready to show that a function $f \in \mathbb{Z}\mathsf{poly}_k$ can be computed by specific $\mathbb{Z}\mathsf{poly}_{k-1}$-SDT named its *$k$-residual transducers*. Their transition function is uniquely defined by $\mathsf{Res}(f)/{\sim_{k-1}}$.

Definition 7.34 (Residual transducer)

Let $k \geqslant 0$, let $\mathscr{T} = (A, Q, q_0, \delta, \mathbb{Z}\mathsf{poly}_{k-1}, \lambda, F)$ be a $\mathbb{Z}\mathsf{poly}_{k-1}$-SDT and $f\colon A^* \to \mathbb{Z}$. We say that $\mathscr{T}$ is a *k-residual transducer* of f if the following conditions hold:

- $\mathscr{T}$ computes f;
- $Q = \mathsf{Res}(f)/{\sim_{k-1}}$;
- for all $u \in A^*$, $u \triangleright f \in \delta^*(q_0, u)$;
- $\lambda(Q, A) \subseteq \mathsf{Span}_{\mathbb{Z}}(\mathsf{Res}(f)) \cap \mathbb{Z}\mathsf{poly}_{k-1}$.

Let $L \subseteq A^*$ be a regular language. A 0-residual transducer of the indicator function $\mathbf{1}_L$ is exactly the minimal automaton of the language L. In particular, it must be unique. However, for $k \geqslant 1$ the k-residual transducer of $f \in \mathbb{Z}\mathsf{poly}_k$ may not be unique: two k-residual transducers share the same underlying automaton (A, Q, δ), but the labels λ are not required to be the same.

Example 7.35 (Residual transducers)

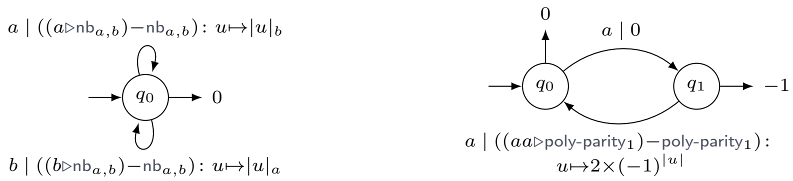
The $\mathbb{Z}\mathsf{poly}_{-1}$-SDT from Figure 7.33a is a 0-residual transducer of $\mathbf{1}_{aA^*}$. The $\mathbb{Z}\mathsf{poly}_0$-SDT from Figure 7.33b is a 1-residual transducer of $\mathbf{1}_{aA^*}$. Indeed, $b \triangleright \mathbf{1}_{aA^*} \sim_0 a \triangleright \mathbf{1}_{aA^*} \sim_0 \mathbf{1}_{aA^*}$, therefore $|\mathsf{Res}(\mathbf{1}_{aA^*})/{\sim_0}| = 1$. Thus a 1-residual transducer of $\mathbf{1}_{aA^*}$ has exactly one state q_0. Furthermore the labels of the transitions belong to $\mathsf{Span}_{\mathbb{Z}}(\mathsf{Res}(\mathbf{1}_{aA^*}))$ since $1 - \mathbf{1}_{aA^*} = (a \triangleright \mathbf{1}_{aA^*}) - \mathbf{1}_{aA^*}$.

Example 7.36 (Counting letters)

Let $A := \{a, b\}$. The function $\mathsf{nb}_{a,b}\colon u \mapsto |u|_a \times |u|_b \in \mathbb{Z}\mathsf{poly}_2$ has a single residual up to $\sim_1$-equivalence. A 2-residual transducer of $\mathsf{nb}_{a,b}$ is depicted in Figure 7.38a.

Example 7.37 (Polynomial parity)

Let $A := \{a\}$. The function $\mathsf{poly\text{-}parity}_1\colon u \mapsto (-1)^{|u|} \times |u| \in \mathbb{Z}\mathsf{poly}_1$ has two residuals up to $\sim_0$-equivalence. A 1-residual transducer of g is depicted in Figure 7.38b.



(a) A 2-residual transducer of $\mathsf{nb}_{a,b}$.　(b) A 1-residual transducer of $\mathsf{poly\text{-}parity}_1$.

Figure 7.38: Two residual transducers.

The main result of Section 7.3.2 is that one can build a k-residual transducer of any function from $\mathbb{Z}\mathsf{poly}_k$. This is the purpose of Theorem 7.39 which originates from [CDL23, Lemma IV.17]. Its proof relies on a simple algorithm which mimics the well-known construction of the residual automaton, while also dealing with the output labels. Furthermore, it crucially relies on the decidability of $\mathbb{Z}\mathsf{poly}_{k-1}$ inside $\mathbb{Z}\mathsf{poly}_k$ (Theorem 5.25) in order to compute the $\sim_{k-1}$-equivalence classes of $\mathsf{Res}(f)$.

Theorem 7.39 (Building a k-residual transducer)

Let $k \geqslant 0$. Given $f \in \mathbb{Z}\mathsf{poly}_k$, one can build a k-residual transducer of f.

Proof. Since $f \in \mathbb{Z}\mathsf{poly}$, recall that $\mathsf{Res}(f)/{\sim_{k-1}}$ is finite thanks to Lemma 7.28. In order to build a k-residual transducer, we apply Algorithm 7.40 which computes the set of residuals of f and the relations between them. In order to simplify the notations, the states of the $\mathbb{Z}\mathsf{poly}_{k-1}$-SDT are not labelled by the equivalence classes of $\mathsf{Res}(f)/{\sim_{k-1}}$, but directly by elements of the class. During the computation, the set Q contains the states for which all outing transitions have been created, while O contains states for which these transitions have not been created yet.

Algorithm 7.40: Computing a k-residual transducer of $f \in \mathbb{Z}\mathsf{poly}_k$

$O := \{\varepsilon \triangleright f\}$
$Q := \varnothing$
while $O \neq \varnothing$ **do**
　Choose some $u \triangleright f \in O$
　for $a \in A$ **do**
　　if $ua \triangleright f \not\sim_{k-1} v \triangleright f$ for all $v \triangleright f \in O \uplus Q$ **then**
　　　$O := O \uplus \{ua \triangleright f\}$
　　　$\delta(u \triangleright f, a) := ua \triangleright f$
　　　$\lambda(u \triangleright f, a) := (w \mapsto 0)$
　　else
　　　let $f \triangleright v \in O \uplus Q$ be such that $ua \triangleright f \sim_{k-1} v \triangleright f$
　　　$\delta(u \triangleright f, a) := v \triangleright f$
　　　$\lambda(u \triangleright f, a) := ua \triangleright f - v \triangleright f$
　　end
　end
　$O := O \setminus \{u \triangleright f\}$
　$Q := Q \uplus \{u \triangleright f\}$
　$F(u \triangleright f) := f(u)$
end

A partial execution of Algorithm 7.40 is depicted in Figure 7.41. In this figure, we assume that

f, $a \triangleright f$, $b \triangleright f$ and $aa \triangleright f$ belong to different $\sim_{k-1}$-equivalence classes, while $aa \triangleright f \sim_{k-1} b \triangleright f$. Nodes are labelled by their creation time. At this stage of the execution, $Q = \{\varepsilon \triangleright f\}$, $O = \{a \triangleright f, b \triangleright f\}$. The blue dashed node is not created because $aa \triangleright f \sim_{k-1} b \triangleright f$ and instead we add the red transition to $b \triangleright f$, which corresponds to the "else" branch of line 10 of Algorithm 7.40.

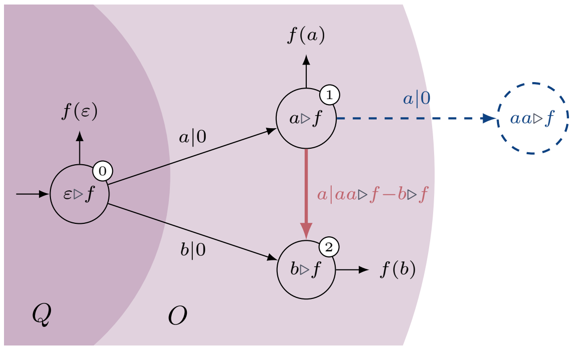


Figure 7.41: Example of a partial execution of Algorithm 7.40.

Now, let us justify the correctness and termination of Algorithm 7.40. First, we observe that the labels on the transitions have shape $u \triangleright f - v \triangleright f$ when $ua \triangleright f \sim_{k-1} v \triangleright f$, hence they describe functions of $\mathsf{Span}_{\mathbb{Z}}(\mathsf{Res}(f)) \cap \mathbb{Z}\mathsf{poly}_{k-1}$ by definition of $\sim_{k-1}$. Observe that the construction of these labels is effective since $f \in \mathbb{Z}\mathsf{poly}_k$ and that equivalence is decidable thanks to Theorem 5.25.

For the termination of Algorithm 7.40, we note that it maintains two sets O and Q such that $O \uplus Q \subseteq \mathsf{Res}(f)$ and for all $f, g \in O \uplus Q$ we have $f \not\sim_{k-1} g$ if $f \neq g$. Hence the algorithm terminates since $\mathsf{Res}(f)/\sim_{k-1}$ is finite and Q increases at every loop. At the end of its execution, we have for all $q \in Q$ and $a \in A$, that $\delta(q, a) \sim_{k-1} a \triangleright q$ and $\lambda(q, a) = a \triangleright q - \delta(q, a)$.

Finally, we show that Algorithm 7.40 builds a k-residual transducer of f. For this purpose, we show by induction on $n \geqslant 0$ that for all $a_1, \cdots, a_n \in A$, if $\delta^*(q_0, a_1 \cdots a_i) = q_i$ and $g_i := \lambda(q_{i-1}, a_i)$ for all $1 \leqslant i \leqslant n$, we have $q_n \sim_{k-1} a_1 \cdots a_n \triangleright f$ and for all $u \in A^*$:

$$f(a_1 \cdots a_n u) = \sum_{i=2}^{n} g_i(a_i \cdots a_n u) + q_n(u).$$

For $n = 0$ the result is obvious because $q_0 = f$. Now, assume that the result holds for some $n \geqslant 0$ and let $a_{n+1} \in A$. Let $q_{n+1} := \delta(q_n, a_{n+1})$ and $g_{n+1} := \lambda(q_n, a_{n+1})$. By induction hypothesis we have $q_n \sim_{k-1} a_1 \cdots a_n \triangleright f$ therefore $a_{n+1} \triangleright q_n \sim_{k-1} a_1 \cdots a_n a_{n+1} \triangleright f$ by Claim 7.27. Because $q_{n+1} = \delta(q_n, a_{n+1}) \sim_{k-1} a_{n+1} \triangleright q_n$, then $q_{n+1} \sim_{k-1} a_1 \cdots a_n a_{n+1} \triangleright f$. Now, let us fix $u \in A^*$. We have $f(a_1 \cdots a_n a_{n+1} u) = \sum_{i=2}^{n} g_i(a_i \cdots a_n a_{n+1} u) + q_n(a_{n+1} u)$ by induction hypothesis. But since $g_{n+1} = \lambda(q_n, a_{n+1}) = a_{n+1} \triangleright q_n - \delta(q_n, a_{n+1}) = a_{n+1} \triangleright q_n - q_{n+1}$ we get $q_n(a_{n+1} u) = g_{n+1}(u) + q_{n+1}(u)$. We conclude the proof that Algorithm 7.40 builds a k-residual transducer of f by considering $u = \varepsilon$ and the definition of the final output function F. ◀

Remark 7.42 (Canonical machine)

In Algorithm 7.40, one needs to "choose" a way to build the states associated to the residuals $u \triangleright f$ when ranging over the elements of O and the letters of A. Different choices may lead to different k-residual transducers, with the same transitions but different function labels. However, if we fix a (computable) ordering over A^* and use it to range over O and A, then Algorithm 7.40 builds a canonical (i.e. which only depends on the semantics of the input function) machine.

Now let us discuss two low-hanging consequences of Theorem 7.39. First, we observe in Corollary 7.43 that $\mathbb{Z}\mathsf{poly}_{k-1}$-SDT exactly compute the class $\mathbb{Z}\mathsf{poly}_k$.

Corollary 7.43 ($\mathbb{Z}\mathsf{poly}_k$ = $\mathbb{Z}\mathsf{poly}_{k-1}$-suffix deterministic transducers)

Let $k \geqslant 0$. A function $f\colon A^* \to \mathbb{Z}$ belongs to $\mathbb{Z}\mathsf{poly}_k$ if and only if it can be computed by a $\mathbb{Z}\mathsf{poly}_{k-1}$-SDT. The conversions are effective.

Proof. Theorem 7.39 shows that any function from $\mathbb{Z}\mathsf{poly}_k$ is effectively computed by its k-residual transducer, which is in particular a $\mathbb{Z}\mathsf{poly}_{k-1}$-SDT. Conversely, if $f\colon A^* \to \mathbb{Z}$ is computed by a $\mathbb{Z}\mathsf{poly}_{k-1}$-SDT, it follows from Equation (7.31) that f can be written as a linear combination of elements of shape $\mathbf{1}_L \otimes g$ where $g \in \mathbb{Z}\mathsf{poly}_{k-1}$. Therefore $f \in \mathbb{Z}\mathsf{poly}_k$ by Lemma 5.21. ◀

As another side result, we obtain a semantic description of $\mathbb{Z}\mathsf{poly}_k$ in terms of $\sim_{k-1}$-equivalence. Corollary 7.44 provides the converse of Lemma 7.28. It justifies that the class of $\mathbb{Z}$-polyregular functions describes a quantitative counterpart of regular languages, built by induction in a layered fashion.

Corollary 7.44 ($\mathbb{Z}\mathsf{poly}_k$ = Finite residuals up to $\sim_{k-1}$)

For all $k \geqslant 0$, $\mathbb{Z}\mathsf{poly}_k = \{f\colon A^* \to \mathbb{Z} \mid \mathsf{Res}(f)/{\sim_{k-1}} \text{ is finite}\}$.

Proof. Every map in $\mathbb{Z}\mathsf{poly}_k$ has finitely many residuals up to $\sim_{k-1}$ thanks to Lemma 7.28. Now let $f\colon A^* \to \mathbb{Z}$ be such that $\mathsf{Res}(f)/{\sim_{k-1}}$ is finite. Observe that Algorithm 7.40 from the proof of Theorem 7.39 can be applied *mutatis mutandis* to build a k-residual transducer of f (the only difference is that it is not effective when f is not explicitly given as function of $\mathbb{Z}\mathsf{poly}_k$). This machine is in particular a $\mathbb{Z}\mathsf{poly}_{k-1}$-SDT. Thanks to Corollary 7.43, it follows that $f \in \mathbb{Z}\mathsf{poly}_k$. ◀

It follows from [BR11, Corollary 5.4 p 14] that a function $f\colon A^* \to \mathbb{Z}$ is a $\mathbb{Z}$-rational series if and only if $\mathsf{Span}_{\mathbb{Z}}(\mathsf{Res}(f))$ has finite dimension (i.e. it is finitely generated as a $\mathbb{Z}$-module). When comparing Corollary 7.44 to this result, one obtains new insights on $\mathbb{Z}\mathsf{poly}$: contrary to $\mathbb{Z}$-rational series, this class has little to do with linear algebra, but it is intrinsically equipped with a layered structure.

7.4 Smooth functions and aperiodic residual transducers

This section can be understood as an analogue of Section 6.3 in the setting for star-free functions. When deciding if a $\mathbb{S}$-polyregular function was $\mathbb{S}$-polyblind, we have used the notion of repetitiveness together with the decidable property of permutability for counting transducers. For deciding star-freeness, our goal is to use smoothness as a semantic condition and to replace permutability for counting transducers by *aperiodicity* for residual transducers (to be introduced formally in Section 7.4.2).

Formally, let $f \in \mathbb{Z}\mathsf{poly}_k$ and $\mathscr{T}$ be a k-residual transducer of f. In a high-level perspective, the reader may aim at showing that the following conditions are equivalent:

(1) f is star-free; (2) f is k-smooth; (3) $\mathscr{T}$ is aperiodic.

We show Item (1) ⇒ Item (2) in Section 7.4.1 and Item (2) ⇒ Item (3) in Section 7.4.2 (with the difference that $(k+1)$-smooth is needed instead of k-smooth). However, Item (3) ⇒ Item (1) has no reason to hold since aperiodicity of a k-residual transducer will only deal with its transition function and not on its label functions in $\mathbb{Z}\mathsf{poly}_{k-1}$, while these functions may create non-star-free behaviors. Therefore, we will show in Section 7.5 how to perform an inductive and effective proof of Item (2) ⇒ Item (1).

7.4.1 Star-free functions are smooth

The goal of Section 7.4.1 is to show that star-free $\mathbb{Z}$-polyregular functions are k-smooth for all $k \geqslant 0$. We first recall that the result is well-known for indicator functions of star-free languages.

Claim 7.45 (Smooth indicator functions)

Let $L \subseteq A^*$ be a star-free language, then $\mathbf{1}_L$ is k-smooth for all $k \geqslant 1$.

Proof idea. Use the aperiodicity of some monoid recognizing L, as for Claim 7.14. ◀

Now we show that smoothness is preserved under the basic operations which build the class $\mathbb{Z}$poly.

Claim 7.46 (Preservation of smoothness under $\cdot$, $+$ and $\otimes$)

Let $k \geqslant 1$, $\delta \in \mathbb{Z}$ and $f, g \colon A^* \to \mathbb{Z}$ be k-smooth. Then $\delta \cdot f$, $f{+}g$, and $f{\otimes}g$ are k-smooth.

Proof. The result is obvious for $\delta \cdot f$ and $f{+}g$. We only deal with the case of the Cauchy product. For this proof, we first re-state in Claim 7.47 a classical result for Cauchy products of polynomials.

Claim 7.47 (Cauchy product of polynomials)

Let $P(X, X_1, \dots, X_k)$ and $Q(Y, Y_1, \dots, Y_\ell)$ be two polynomials, then:

$$P{\otimes}Q \colon Z, X_1, \dots, X_k, Y_1 \dots Y_\ell \mapsto \sum_{X=0}^{Z} P(X, X_1, \dots, X_k) Q(Z{-}X, Y_1, \dots, Y_\ell)$$

is a polynomial in $Z, X_1, \dots, X_k, Y_1, \dots, Y_\ell$.

Proof sketch. It suffices to check that the result holds for products of monomials, i.e. for:

$$(X^p X_1^{p_1} \cdots X_k^{p_k}){\otimes}(Y^q Y_1^{q_1} \cdots Y_\ell^{q_\ell}) = (X^p{\otimes}Y^q) \times X_1^{p_1} \cdots X_k^{p_k} Y_1^{q_1} \cdots Y_\ell^{q_\ell}.$$

Hence the only thing to check is that $Z \mapsto X^p{\otimes}X^q(Z) = \sum_{X=0}^{Z} X^p(Z{-}X)^q$ is a polynomial in Y, which is a classical result for polynomials. ◀

Now let us prove that $f{\otimes}g$ is k-smooth. Let $v_0, u_1, v_1, \dots, u_k, v_k \in A^*$, then:

$$\begin{aligned}
&(f{\otimes}g)(v_0 u_1^{X_1} v_1 \cdots u_k^{X_k} v_k) = f(v_0 u_1^{X_1} v_1 \cdots u_k^{X_k} v_k) g(\varepsilon) \\
&+ \sum_{j=0}^{k} \sum_{i=0}^{|v_j|-1} f(v_0 u_1^{X_1} v_1 \cdots u_j^{X_j} (v_j[1{:}i])) \times g((v_j[i{+}1{:}|v_j|]) u_{j+1}^{X_{j+1}} \cdots v_k) \\
&+ \sum_{j=1}^{k} \sum_{i=0}^{|u_j|-1} \sum_{Y=0}^{X_j-1} f(v_0 u_1^{X_1} v_1 \cdots u_j^{Y} (u_j[1{:}i])) \times g((u_j[i{+}1{:}|u_j|]) u_j^{X_j-Y-1} \cdots v_k)
\end{aligned} \tag{7.48}$$

Since f and g are assumed to be k-smooth (and therefore ℓ-smooth for all $1 \leqslant \ell \leqslant k$), there exists $\Omega \geqslant 1$ such that for all $X_1, \dots, X_k \geqslant \Omega$, the two first lines of Equation (7.48) describe polynomials in $X_1, \dots, X_k$. Let us focus on the last line. For this case, we observe that for all $1 \leqslant j \leqslant k$ and

$0 \leqslant i \leqslant |u_j| - 1$ the function which maps $X_1, \dots, X_k$ to:

$$\sum_{Y=0}^{X_j-1} f(v_0 u_1^{X_1} v_1 \cdots u_j^{Y}(u_j[1{:}i])) \times g((u_j[i{+}1{:}|u_j|])u_j^{X_j-Y-1} \cdots v_k)$$

is nearly[13] the Cauchy product of two polynomials by assumption of smoothness of f and g. We conclude by using Claim 7.47 to show that this Cauchy product is a polynomial. ◀

Finally, Lemma 7.49 can be seen as an analogue of Lemma 6.24 for repetitiveness.

Lemma 7.49 (Star-free ⇒ Smooth)

Let $\mathbb{S} := \mathbb{Z}$ or $\mathbb{N}$. A star-free $\mathbb{S}$-polyregular function is k-smooth for all $k \geqslant 1$.

Proof. We apply Theorem 7.13 together with Claims 7.45 and 7.46. ◀

7.4.2 Smooth functions are computed by aperiodic residual transducers

Now we describe a necessary condition, named *aperiodicity*, for a k-residual transducer to compute a function $f \in \mathbb{Z}\mathsf{SFpoly}_k$. It can be seen as an analogue of permutability.

Recall that a deterministic automaton is aperiodic if its transition monoid is so. It is easy to see that this property can be re-written as the absence of elementary loops labelled by a power u^n of some word for $n > 1$. We lift this notion to $\mathfrak{F}$-SDT in Definition 7.50 (observe that it does not deal with the set $\mathfrak{F}$).

Definition 7.50 (Aperiodic suffix deterministic transducer)

A $\mathfrak{F}$-SDT $(A, Q, q_0, \delta, \mathfrak{F}, \lambda, F)$ is said to be *aperiodic* if its underlying automaton is so, i.e. if for all $q \in Q$, $u \in A^*$ and $n \geqslant 1$ such that $\delta(q, u^n) = q$, we have $\delta(q, u) = q$.

Example 7.51 (Aperiodic $\mathfrak{F}$-SDT)

The transducers of Figures 7.33a, 7.33b and 7.38a are aperiodic, while the transducer of Figure 7.38b is not. Indeed, in this machine we have $\delta(q_0, aa) = q_0$ but $\delta(q_0, a) \neq q_0$.

Now let us show that a k-residual transducer of a $(k{+}1)$-smooth function is aperiodic. The proof of this result heavily relies on the definition of a k-residual transducer which uses $\sim_{k-1}$.

Lemma 7.52 (Smooth ⇒ Aperiodic)

Let $k \geqslant 0$. Let $f \in \mathbb{Z}\mathsf{poly}_k$ which is $(k{+}1)$-smooth and let $\mathscr{T}$ be a k-residual transducer of f. Then $\mathscr{T}$ is aperiodic.

Proof. Let $\mathscr{T} = (A, Q, q_0, \delta, \mathbb{Z}\mathsf{poly}_{k-1}, \lambda, F)$ be a k-residual transducer of f. Let $v, u \in A^*$ and suppose that $\delta(q_0, v) = \delta(q_0, vu^n)$ for some $n \geqslant 1$. We want to show that $\delta(q_0, vu) = \delta(q_0, v)$. Since $\delta(q_0, v) = \delta(q_0, vu^{nX})$ and $\delta(q_0, vu) = \delta(q_0, vu^{nX+1})$ for all $X \geqslant 1$, it is sufficient

[13] Recall that smoothness only identifies the terms of this sum with polynomials for $Y \geqslant \Omega$ and $X_j - Y - 1 \geqslant \Omega$. Formally, one would have to treat separately the terms in $Y \leqslant \Omega$ or $X_j - Y - 1 \leqslant \Omega$ in order to get a Cauchy product of polynomials.

to show that we have $\delta(q_0, vu^{nX+1}) = \delta(q_0, vu^{nX})$ for some $X \geqslant 1$. Let $\Omega \geqslant 1$ be the integer given by Definition 7.15 as a witness of the $(k+1)$-smoothness of f, we aim at showing that that $(vu^{n\Omega+1} \triangleright f) \sim_{k-1} (vu^{n\Omega} \triangleright f)$, which yields $\delta(q_0, vu^{n\Omega+1}) = \delta(q_0, vu^{n\Omega})$ by definition of a k-residual transducer. Therefore by Theorem 5.54, it is sufficient to show that for all $v_0, u_1, v_1, \cdots, u_k, v_k \in A^*$ we have the following:

$$|(vu^{n\Omega} \triangleright f - vu^{n\Omega+1} \triangleright f)(v_0 u_1^Y \cdots u_k^Y v_k)| = \mathcal{O}(Y^{k-1})$$

Because f is $(k+1)$-smooth, for all $X, Y \geqslant \Omega$, $f(vu^X v_0 u_1^Y \cdots u_k^Y v_k)$ is a polynomial $P(X, Y)$. We show that $|P(n\Omega, Y) - P(n\Omega+1, Y)| = \mathcal{O}(Y^{k-1})$. Since $f \in \mathbb{Z}\mathsf{poly}_k$, we obtain from Claim 5.52 that P has degree at most k. Therefore it can be written as $P_0(Y) + XP_1(Y) + \cdots + X^k P_k(Y)$ where $P_i(Y)$ is a polynomial in Y of degree at most $k - i$ for all $0 \leqslant i \leqslant k$. Thus:

$$\begin{aligned} |P(n\Omega, Y) - P(n\Omega + 1, Y)| &= \left| \sum_{i=1}^{k} P_i(Y)((n\Omega)^i - (n\Omega+1)^i) \right| \\ &\leqslant \sum_{i=1}^{k} |P_i(Y)|(n\Omega + 1)^i \end{aligned}$$

since the term P_0 vanishes when doing the subtraction. The bound in $\mathcal{O}(Y^{k-1})$ directly follows since the polynomials $P_i(Y)$ for $1 \leqslant i \leqslant k$ have degree at most $k-1$. ◀

7.5 Solving the star-free membership problem

This section is devoted to concluding the proof of Theorem 7.19 (it will directly follow from the more precise Theorem 7.54), by leveraging the tools introduced in Sections 7.3 and 7.4.

We first observe that a function computed by an aperiodic $\mathbb{Z}\mathsf{SFpoly}_{k-1}$-SDT belongs to $\mathbb{Z}\mathsf{SFpoly}_k$. Lemma 7.53 can be seen as an analogue of Lemmas 5.53 and 6.50 in the previous chapters. Its proof basically relies on the fact that an aperiodic finite automaton computes a star-free language.

Lemma 7.53 (Aperiodic ⇒ Star-free)

Let $k \geqslant 0$, an aperiodic $\mathbb{Z}\mathsf{SFpoly}_{k-1}$-SDT (effectively) computes a function of $\mathbb{Z}\mathsf{SFpoly}_k$.

Proof. Let $\mathscr{T} = (A, Q, q_0, \delta, \mathbb{Z}\mathsf{poly}_{k-1}, \lambda, F)$ be an aperiodic $\mathbb{Z}\mathsf{SFpoly}_{k-1}$-SDT which computes a function $f : A^* \to \mathbb{Z}$. Since the deterministic automaton (A, Q, q_0, δ) is aperiodic, it is well-known that for all $q \in Q$ the language $L_q := \{u \mid \delta(q_0, u) = q\}$ is star-free. So is $L_q a$ for all $a \in A$ and $q \in Q$. It follows from Equation (7.31) that f can be written as a linear combination of $\mathbf{1}_L \otimes g$ where L is star-free and $g \in \mathbb{Z}\mathsf{SFpoly}_{k-1}$. Therefore $f \in \mathbb{Z}\mathsf{SFpoly}_k$ by Lemma 7.11. ◀

We are ready to show Theorem 7.54, which originates from [CDL23, Theorem V.13]. This result is obtained by induction on $k \geqslant 1$ by using the fact that since the label functions of a k-residual transducer belong to $\mathbb{Z}\mathsf{poly}_{k-1}$, then one can decide by induction hypothesis whether these function of “lower degree” are star-free. As in the case of Chapter 6, equivalence between the semantic condition (smoothness) and star-free functions is not only a nice consequence of this proof, but also a key ingredient within the induction step. Indeed, we crucially rely on the fact that smoothness is preserved under linear combinations and residuals. All in all, the proof sketch is comparable to that of Theorem 6.51.

Theorem 7.54 (Induction step for $\mathbb{Z}$-polyregular $\to$ star-free $\mathbb{Z}$-polyregular)

Let $k \geqslant 0$ and $f \in \mathbb{Z}\mathsf{poly}_k$, the following conditions are equivalent:

(1) f is star-free $\mathbb{Z}$-polyregular;
(2) f is $(k+1)$-smooth;
(3) any k-residual transducer of f is aperiodic and has labels in $\mathbb{Z}\mathsf{SFpoly}_{k-1}$;
(4) there exists an aperiodic $\mathbb{Z}\mathsf{SFpoly}_{k-1}$-SDT which computes f;
(5) f is computed by an aperiodic k-counting transducer (i.e. $f \in \mathbb{Z}\mathsf{SFpoly}_k$).

Furthermore, this property is decidable and the constructions are effective.

Proof. The proof of this result is performed by induction on $k \geqslant 1$. Item (5) $\Rightarrow$ Item (1) is obvious. Item (1) $\Rightarrow$ Item (2) is exactly Lemma 7.49. Item (3) $\Rightarrow$ Item (4) follows from Theorem 7.39 which implies that a k-residual transducer exists. Item (4) $\Rightarrow$ Item (5) is exactly Lemma 7.53.

The subtle point is Item (2) $\Rightarrow$ Item (3). To show it we first apply Lemma 7.52 to show that any k-residual transducer of f is aperiodic. Furthermore, its label functions are $(k+1)$-smooth since this property is preserved under taking linear combinations (Claim 7.46) and residuals (which is obvious). In particular they are k-smooth. Since these functions belong to $\mathbb{Z}\mathsf{poly}_{k-1}$ by definition of a k-residual transducer, then one can apply Item (2) $\Rightarrow$ Item (5) by induction hypothesis[14] and therefore these label functions (effectively) belong to $\mathbb{Z}\mathsf{SFpoly}_{k-1}$.

Decidability is obtained thanks to Item (3): we first compute some k-residual transducer by applying Theorem 7.39 and we decide if it is aperiodic. Furthermore by induction hypothesis one can decide if its function labels (which are effectively built) belong to $\mathbb{Z}\mathsf{SFpoly}_{k-1}$. ◀

7.6 Aperiodicity through the lens of eigenvalues

In this section, we intend to give another characterization of $\mathbb{Z}$-polyregular functions and star-free $\mathbb{Z}$-polyregular functions among $\mathbb{Z}$-rational series. These results will provide a new perspective on star-freeness thanks to *eigenvalues*[15]. However, to the knowledge of the author, the techniques of Section 7.6 do not yield an effective decision procedure, contrary to the proof of Section 7.5.

Let $(\mathbb{S}, +, \times)$ be a semiring. Recall from Definition 4.43 that $(\mathbb{S}, +, \times)$-rational series are computed by the model of $(\mathbb{S}, +, \times)$-weighted automata. We say that a $(\mathbb{S}, +, \times)$-weighted automaton computing a function f is *minimal*, when it has a minimal number of states among all the $(\mathbb{S}, +, \times)$-weighted automata which compute f. The study of minimal weighted automata originates from [Sch61a] and plays an important role[16] in the theory of rational series (see e.g. [BR11, Chapter 2] for a survey).

Given a matrix $M \in \mathsf{M}_{n,n}(\mathbb{C})$, we let $\mathsf{Spec}(M) \subseteq \mathbb{C}$ be its *spectrum*, which is the set of all its eigenvalues. If $S \subseteq \mathsf{M}_{n,n}(\mathbb{C})$, we let its *spectrum* $\mathsf{Spec}(S) := \bigcup_{M \in S} \mathsf{Spec}(M)$ be the union of the spectra of its matrices. From now on, the notation $|\cdot|$ is also used for the *modulus* of a complex number (which is an extension of the absolute value of real numbers). We let $\mathbb{D} := \{\gamma \in \mathbb{C} \mid |\gamma| \leqslant 1\}$ be the *unit disc* and $\mathbb{U} := \{\gamma \in \mathbb{C} \mid \exists n \geqslant 1, \gamma^n = 1\}$ be the set of the *roots of unity*.

[14] As in the proof of Theorem 6.51, the reader is invited to gaze in admiration at this argument. Indeed, as mentioned above, using the robustness of a *semantic* condition here becomes a key and nearly magical technique to prove the desired result by induction.

[15] Recall that $\gamma \in \mathbb{C}$ is an *eigenvalue* of a matrix $M \in \mathsf{M}_{n,n}(\mathbb{C})$ if there exists $0 \neq V \in \mathsf{M}_{n,1}(\mathbb{C})$ such that $MV = \gamma V$.

[16] It is in particular used to show that equivalence of $(\mathbb{Q}, +, \times)$-rational series is decidable. However, contrary to the case of automata for languages, there exist in general several distinct minimal $(\mathbb{S}, +, \times)$-weighted automata for a rational series. In other words, a minimal weighted automaton is not in general a canonical object.

7.6.1 Spectra for $\mathbb{Z}$-polyregular functions

The goal of Section 7.6.1 is to show Theorem 7.56 which connects the notion of $\mathbb{Z}$-polyregular function to the eigenvalues of a minimal weighted automaton computing this function.

As a first step, let us observe in Claim 7.55 how the eigenvalues of a minimal weighted automaton are revealed by iterating words. This result uses classical arguments from the theory of rational series.

Claim 7.55 (Capturing eigenvalues)

Let $f\colon A^* \to \mathbb{Z}$ be a $\mathbb{Z}$-rational series and $(A, [1{:}n], I, F, \mu)$ be a minimal $\mathbb{Q}$-weighted automaton[17] computing f. Let $u \in A^*$ and $\gamma \in \mathsf{Spec}(\mu(u))$. There exist $\alpha_{i,j} \in \mathbb{C}$ for $1 \leqslant i, j \leqslant n$ and $v_1, w_1, \dots, v_n, w_n \in A^*$ such that $\gamma^X = \sum_{i,j=1}^{n} \alpha_{i,j} f(v_i u^X w_j)$ for all $X \geqslant 0$.

Proof. Let $u \in A^*$, $\gamma \in \mathsf{Spec}(\mu(u))$ and $0 \neq V \in \mathsf{M}_{n,1}(\mathbb{C})$ be such that $\mu(u)V = \gamma V$. We let $||V|| := {}^tVV$, observe that this value is a positive real number. It follows from [BR11, Proposition 2.1 p 32], since $\mathbb{Q}$ is a field and $(A, [1{:}n], I, F, \mu)$ is a minimal $\mathbb{Q}$-weighted automaton, that $\mathsf{Span}_{\mathbb{Q}}(\{\mu(u)F \mid u \in A^*\}) = \mathbb{Q}^n$. Hence there exists numbers $\delta_j \in \mathbb{C}$ and words $w_j \in A^*$ such that $V = \sum_{j=1}^{n} \delta_j \mu(w_j)F$. Symmetrically by [BR11, Proposition 2.1 p 32], there exists numbers $\eta_i \in \mathbb{C}$ and words $v_i \in A^*$ such that ${}^tV = \sum_{i=1}^{n} \eta_i I\mu(v_i)$. Therefore:

$$\gamma^X ||V|| = {}^tV\mu(u)^X V = \sum_{i,j=1}^{n} \eta_i \delta_j I \mu(v_i u^X w_j) F = \sum_{i,j=1}^{n} \eta_i \delta_j f(v_i u^X w_j).$$

The result follows by defining $\alpha_{i,j} := \eta_i \delta_j / ||V||$ for all $1 \leqslant i, j \leqslant n$. ◀

Theorem 7.56 originates from [CDL23, Theorem II.31]. This result provides yet another characterization of $\mathbb{Z}$-polyregular functions among $\mathbb{Z}$-rational series. The main intuition is that having eigenvalues whose modulus is strictly greater than 1 leads to exponential behaviors, while $\mathbb{Z}$-polyregular functions must have polynomial asymptotic growth (recall Theorem 5.22).

Theorem 7.56 ($\mathbb{Z}$-polyregular = eigenvalues are 0 or roots of unity)

Let $f\colon A^* \to \mathbb{Z}$ be a $\mathbb{Z}$-rational series, the following are equivalent:

(1) f is $\mathbb{Z}$-polyregular;
(2) there exists $\Omega \geqslant 1$ such that for all $v, u, w \in A^*$, the function $X \mapsto f(vu^{\Omega X}w)$ is a polynomial for X large enough;
(3) for all minimal $\mathbb{Q}$-weighted automaton (A, Q, I, F, μ) of f, $\mathsf{Spec}(\mu(A^*)) \subseteq \mathbb{U} \cup \{0\}$;
(4) for all minimal $\mathbb{Z}$-weighted automaton (A, Q, I, F, μ) of f, $\mathsf{Spec}(\mu(A^*)) \subseteq \mathbb{U} \cup \{0\}$;
(5) there exists a $\mathbb{Z}$-weighted automaton (A, Q, I, F, μ) of f such that $\mathsf{Spec}(\mu(A^*)) \subseteq \mathbb{D}$.

Proof. For Item (1) $\Rightarrow$ Item (2), consider a k-counting transducer whose transition monoid is $\mu\colon A^* \to \mathbb{T}$ and which computes the function f. The result is similar to Proposition 2.16 and it follows from Lemma 5.37 by choosing Ω such that $\mu(u^\Omega)$ is an idempotent of $\mu(A^*)$.

For Item (2) $\Rightarrow$ Item (3), let $(A, [1{:}n], I, F, \mu)$ be a minimal $\mathbb{Q}$-weighted automaton which computes f. Let $u \in A^*$ and $\gamma \in \mathsf{Spec}(\mu(u))$. Thanks to Claim 7.55, there exist $\alpha_{i,j}, v_i, w_j$ such that $\gamma^X = \sum_{1 \leqslant i,j \leqslant n} \alpha_{i,j} f(v_i u^X w_j)$ for X large enough. By assumption, $X \mapsto f(v_i u^{\Omega X} w_j)$ is a

[17] Recall from Definition 4.43 that in this case we have $\mu\colon A^* \to \mathsf{M}_{n,n}(\mathbb{Q})$.

polynomial for X large enough, hence so is $X \mapsto \sum_{1 \leqslant i,j \leqslant n} \alpha_{i,j} f(v_i u^X w_j) = \gamma^{\Omega X} = (\gamma^\Omega)^X$. This polynomial has to be constant and therefore $\gamma^\Omega \in \{0,1\}$, which implies that $\gamma \in \{0\} \cup \mathbb{U}$.

Item (3) $\Rightarrow$ Item (4) follows since a minimal $\mathbb{Z}$-weighted automaton of a $\mathbb{Z}$-rational series is also a minimal $\mathbb{Q}$-weighted automaton by [BR11, Theorem 1.1 p 121]. Item (4) $\Rightarrow$ Item (5) is trivial.

For Item (5) $\Rightarrow$ Item (1), we use [Bel05, Theorem 2.6] which shows that if $\mathsf{Spec}(\mu(A^*)) \subseteq \mathbb{D}$ then the coefficients of $\mu(u)$ are in $\mathcal{O}(|u|^k)$ for some $k \geqslant 0$. Therefore $|f(u)| = \mathcal{O}(|u|^k)$ (since it is a combination of the coefficients) and thus f is $\mathbb{Z}$-polyregular by Theorem 5.22. ◀

Beware that Items (3) and (5) do not deal with $\mathsf{Spec}(\mu(A))$ but with $\mathsf{Spec}(\mu(A^*))$. Using this more general statement is necessary since the eigenvalues of the product of two matrices may have nothing to do with the eigenvalues of the two original matrices. In the same vein, the set $(\mathsf{Spec}(\mu(A^*)), \times)$ has no reason to be a semigroup, even if $(\mathbb{U} \cup \{0\}, \times)$ is a monoid.

Example 7.57 (Polynomial parity)

Recall from Example 5.23 that the function $\mathsf{poly\text{-}parity}_1 \colon u \mapsto (-1)^{|u|}|u|$ is computed by the following $\mathbb{Z}$-weighted automaton (which turns out to be minimal):

$$\left(A, [1{:}2], \begin{pmatrix} -1 & 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \mu \colon u \mapsto \begin{pmatrix} -1 & 1 \\ 0 & -1 \end{pmatrix}^{|u|}\right).$$

The eigenvalues of any matrix in $\mu(A^*)$ belong to $\{\pm 1\}$.

Since $\mathbb{Q}$ is a field, it is well-known (see e.g. [BR11, Chapter 2]) that one can effectively compute a minimal $\mathbb{Q}$-weighted automaton which computes a $\mathbb{Z}$-rational series. However, given such a machine, the author is not aware of a direct[18] way to decide whether $\mathsf{Spec}(\mu(A^*)) \subseteq \mathbb{D}$ holds.

7.6.2 Spectra for star-free $\mathbb{Z}$-polyregular functions

Now we provide an analogue of Theorem 7.56 when dealing with star-free $\mathbb{Z}$-polyregular functions. Following the notion of aperiodicity for monoids, the main intuition is that eigenvalues of $\mathbb{U} \smallsetminus \{1\}$ lead to periodic behaviors of the function since they corresponds to non-trivial subgroups of $\mathbb{U}$.

Theorem 7.58 (Star-free $\mathbb{Z}$-polyregular = eigenvalues are 0 or 1)

Let $f \colon A^* \to \mathbb{Z}$ be a $\mathbb{Z}$-rational series, the following are equivalent:

(1) f is star-free $\mathbb{Z}$-polyregular;
(2) for all $v, u, w \in A^*$, the function $X \mapsto f(vu^X w)$ is a polynomial for X large enough;
(3) for all minimal $\mathbb{Q}$-weighted automaton (A, Q, I, F, μ) of f, $\mathsf{Spec}(\mu(A^*)) \subseteq \{0,1\}$;
(4) for all minimal $\mathbb{Z}$-weighted automaton (A, Q, I, F, μ) of f, $\mathsf{Spec}(\mu(A^*)) \subseteq \{0,1\}$;
(5) there exists a $\mathbb{Z}$-weighted automaton (A, Q, I, F, μ) of f such that $\mathsf{Spec}(\mu(A^*)) \subseteq \{0,1\}$.

Proof. For Item (1) $\Rightarrow$ Item (2) we rely on Theorem 7.19 since Item (2) describes 1-smoothness.

For Item (2) $\Rightarrow$ Item (3), let $(A, [1{:}n], I, F, \mu)$ be a minimal $\mathbb{Q}$-weighted automaton which computes f. Let $u \in A^*$ and $\gamma \in \mathsf{Spec}(\mu(u))$. Thanks to Claim 7.55, there exist $\alpha_{i,j}, v_i, w_j$ such that $\gamma^X = \sum_{1 \leqslant i,j \leqslant n} \alpha_{i,j} f(v_i u^X w_j)$ for X large enough. By assumption, $X \mapsto f(v_i u^X u_j)$ is a polynomial for X large enough, hence so is $X \mapsto \sum_{1 \leqslant i,j \leqslant n} \alpha_{i,j} f(v_i u^X w_j) = \gamma^X$. This polynomial has to be constant and therefore we obtain $\gamma \in \{0,1\}$.

[18] Observe that an "indirect" proof is always possible by combining Theorems 5.22 and 7.56.

Item (3) $\Rightarrow$ Item (4) follows since a minimal $\mathbb{Z}$-weighted automaton of a $\mathbb{Z}$-rational series is also a minimal $\mathbb{Q}$-weighted automaton by [BR11, Theorem 1.1 p 121]. Item (4) $\Rightarrow$ Item (5) is trivial.

For Item (5) $\Rightarrow$ Item (1), it is sufficient to show that f is k-smooth for all $k \geqslant 0$ thanks to[19] Theorem 7.19. Because the eigenvalues of the matrix $\mu(u) \in \mathsf{M}_{n,n}(\mathbb{Z})$ for $u \in A^*$ are all in $\{0, 1\}$, its *characteristic polynomial* splits over the field $\mathbb{Q}$, hence there exists $P \in \mathsf{M}_{n,n}(\mathbb{Q})$ such that $T := P\mu(u)P^{-1}$ is upper triangular with diagonal values in $\{0, 1\}$. In particular, $\mu(u^X) = \mu(u)^X = P^{-1}T^X P$ for all $X \geqslant 0$. It can be shown by induction that the coefficients of $X \mapsto T^X$ are polynomials for X large enough, hence so do the coefficients of $X \mapsto \mu(u^X)$. By doing sums and products of polynomials, we see that for all $k \geqslant 0$ and $v_0, u_1, v_1, \dots, u_k, v_k \in A^*$, the function $X_1, \dots, X_k \mapsto f(v_0 u_1^{X_1} \alpha_1 \cdots u_k^{X_k} v_k)$ is a polynomial for $X_1, \dots, X_k$ large enough. ◀

Observe that the equivalence between Items (1) and (2) in Theorem 7.58 provides a refinement of Theorem 7.19. Indeed, it means that 1-smoothness is sufficient to characterize the functions of $\mathbb{Z}\mathsf{poly}_k$ which are star-free (instead of $(k+1)$-smoothness). As for Theorem 7.56, the author is not aware of a way to use Theorem 7.58 for deciding the star-freeness of a $\mathbb{Z}$-polyregular function, even if a minimal $\mathbb{Q}$-weighted automaton for this function can effectively be computed.

Example 7.59 (Polynomial parity)

It follows from Example 7.57 that $\mathsf{poly\text{-}parity}_1 : u \mapsto (-1)^{|u|}|u|$ does not belong to $\mathbb{Z}\mathsf{SFpoly}$.

As a concluding remark, let us define the class $\mathbb{Z}\mathsf{SFrat}$ of $\mathbb{Z}$-rational series which are computed by some $\mathbb{Z}$-weighted automaton (A, Q, I, F, μ) such that $\mathsf{Spec}(\mu(A^*)) \subseteq \{\gamma \in \mathbb{R} \mid \gamma \geqslant 0\}$. Theorems 7.56 and 7.58 suggest that $\mathbb{Z}\mathsf{SFrat}$ is a natural candidate for extending *star-freeness* to the whole class of $\mathbb{Z}$-rational series. To the knowledge of the author, this class has never been studied in the literature.

Open question 7.60 (Star-free $\mathbb{Z}$-rational series)

Is the class $\mathbb{Z}\mathsf{SFrat}$ well-behaved? Does it coincide with the closure under some operations of $\mathbb{Z}\mathsf{SFpoly}$ together with the series of shape $u \mapsto {\delta_1}^{|u|_{a_1}} \cdots {\delta_k}^{|u|_{a_k}}$ for $\delta_1, \dots, \delta_k \in \mathbb{N}$?

7.7 Discussion: deciding star-freeness for other monoids

The goal if this section is to discuss the decidability of star-freeness for other classes of functions. The main conviction of the author is that, even if building canonical models can be done e.g. for $\mathbb{Z}$-polyregular functions (this chapter) or for rational functions[20] [FGL19], this strategy is approaching its limits. To the contrary, it would be relevant to generalize the techniques of Chapter 6, at the cost of dealing with combinatorial properties and building variants of factorization forests in a star-free fashion.

7.7.1 Star-free $\mathbb{N}$-polyregular functions

We first discuss the case of star-free $\mathbb{N}$-polyregular functions. Observe that Lemma 7.49 shows that such functions are k-smooth for all $k \geqslant 1$. Conjecture 7.61 is an analogue of Theorem 7.19, observe that it would imply $\mathbb{N}\mathsf{poly} \cap \mathbb{Z}\mathsf{SFpoly} = \mathbb{N}\mathsf{SFpoly}$ and furthermore that an optimization theorem holds for the classes $\mathbb{N}\mathsf{poly}_k$ (in the same way as Corollary 7.21 holds for $\mathbb{Z}\mathsf{poly}_k$).

[19] Because this proof relies on the difficult direction of Theorem 7.19, we do not know how to adapt Theorem 7.58 to $\mathbb{N}\mathsf{SFpoly}$.
[20] Recall that word-to-word rational functions have nothing to do with rational series.

Conjecture 7.61 (Star-free $\mathbb{N}$-polyregular functions)

A function $f \in \mathbb{N}\mathsf{poly}_k$ is star-free $\mathbb{N}$-polyregular if and only if it is $(k+1)$-smooth. This property is decidable. If it holds, one can build an aperiodic k-counting transducer which computes f.

However, the author is not aware of a way to adapt the techniques of Chapter 7 to $\mathbb{N}$-polyregular functions. A major obstacle lies in the construction of the k-residual transducer: even if the input function is $f \in \mathbb{N}\mathsf{poly}_k$, the transition labels have no reason to be in $\mathbb{N}\mathsf{poly}_{k-1}$. Indeed, such functions are obtained in Algorithm 7.40 by doing subtractions between $\sim_{k-1}$-equivalent residuals of f, which produces functions of $\mathbb{Z}\mathsf{poly}_{k-1}$, but not necessarily of $\mathbb{N}\mathsf{poly}_{k-1}$. The overall intuition is that making subtractions to "correct errors" is relevant in group such as $\mathbb{Z}$, but not in the case of a monoid like $\mathbb{N}$.

As mentioned above, a radically different proof strategy for Conjecture 7.61 would be to forget about the construction of a canonical model, and show instead that any counting transducer which computes a star-free $\mathbb{N}$-polyregular function verifies a decidable structural property. This is exactly what we have done in Chapter 5 with pumpability and in Chapter 6 with permutability. Such a property for characterizing star-free functions is presented in Example 7.62 in a very simple case.

Example 7.62 (Structural property for 1-counting transducers)

Consider a 1-counting transducer $\mathscr{T}$ with transition morphism is $\mu\colon \{a\}^* \to \mathbb{Z}/2\mathbb{Z}$ which computes a 1-smooth $\mathbb{N}$-polyregular function $f\colon \{a\}^* \to \mathbb{N}$. There exists $\alpha \in \mathbb{N}$ such that $\mathsf{prod}_{\mathscr{T}}(0 \lfloor a \rfloor 1) + \mathsf{prod}_{\mathscr{T}}(1 \lfloor a \rfloor 0) = 2\alpha$ and $\mathsf{prod}_{\mathscr{T}}(0 \lfloor a \rfloor 0) = \mathsf{prod}_{\mathscr{T}}(1 \lfloor a \rfloor 1) = \alpha$.

As a consequence, we have $f(a^X) = \alpha X$ for all $X \geqslant 0$ and $\mathscr{T}$ can be simulated by an aperiodic 1-counting transducer which always outputs α when processing a letter.

Proof. Since f is 1-smooth we have $f(a^X) = \alpha X + \beta$ for X large enough. By applying Claim 5.27 on productions we get $f(a^{2X}) = X \times \mathsf{prod}_{\mathscr{T}}(0 \lfloor a \rfloor 1) + X \times \mathsf{prod}_{\mathscr{T}}(1 \lfloor a \rfloor 0)$ and $f(a^{2X+1}) = (X+1) \times \mathsf{prod}_{\mathscr{T}}(0 \lfloor a \rfloor 0) + X \times \mathsf{prod}_{\mathscr{T}}(1 \lfloor a \rfloor 1)$. Hence we obtain:

$$\mathsf{prod}_{\mathscr{T}}(0 \lfloor a \rfloor 1) + \mathsf{prod}_{\mathscr{T}}(1 \lfloor a \rfloor 0) = 2\alpha = \mathsf{prod}_{\mathscr{T}}(0 \lfloor a \rfloor 0) + \mathsf{prod}_{\mathscr{T}}(1 \lfloor a \rfloor 1). \tag{7.63}$$

Now for X large enough we have $f(a^{2X}) = \alpha \times (2X) + \beta = 2\alpha X$ thus $\beta = 0$. Therefore we obtain $f(2^{X+1}) = \alpha \times (2X+1) = 2\alpha + \mathsf{prod}_{\mathscr{T}}(0 \lfloor a \rfloor 0)$ hence $\mathsf{prod}_{\mathscr{T}}(0 \lfloor a \rfloor 0) = \alpha$ and finally $\mathsf{prod}_{\mathscr{T}}(1 \lfloor a \rfloor 1) = \alpha$ thanks to Equation (7.63). ◀

Conversely, one would have to show that when the structural property holds, the function computed by the counting transducer is (effectively) star-free. This is where we needed factorization forests in Chapters 5 and 6, but recall from Footnote 9 that such forests cannot be built in a star-free fashion. Nevertheless Colcombet et al. have shown in [CvGM22, Section 5] that weakened forms of μ-factorization forests called *first-order approximants* can be built in a star-free fashion even when $\mu(A^*)$ is not aperiodic. The author believes that this tool can be helpful to deal with Conjecture 7.61.

7.7.2 Star-free regular functions

Now let us briefly deal with word-to-word star-free regular functions (Open question 7.5). It is easy to see (e.g. by adapting the proof of Proposition 2.16) that if $f\colon A^* \to B^*$ is regular, then there exists $\Omega \geqslant 1$ such that for all $v, u, w \in A^*$, $f(vu^{\Omega(X+1)}w)$ has shape $\alpha_0 \beta_1^X \alpha_1 \cdots \beta_n^X \alpha_n$ for all $X \geqslant 0$.

We suggest in Conjecture 7.64 that star-free regular functions can be characterized by an according adaptation of 1-smoothness, in the setting of non-commutative outputs.

Conjecture 7.64 (Star-free regular functions)

A regular function is star-free if and only if there exists $\Omega \geqslant 0$ such that the following holds. For all $v, u, w \in A^*$, there exist $n \geqslant 0$, $\alpha_0, \dots, \alpha_n \in B^*$ and $\beta_1, \dots, \beta_n \in B^+$ such that $f(vu^{X+\Omega}w) = \alpha_0 \beta_1^X \alpha_1 \cdots \beta_n^X \alpha_n$ for all $X \geqslant 0$.

By adapting once more the proof of Proposition 2.16, it is easy to see that the condition of Conjecture 7.64 is necessary for being star-free. The author believes that such a semantic property can be decided by transforming it into an equivalent decidable structural property on 2DT.

Part III

Streaming computability over infinite words

Chapter 8

Background on transductions of infinite words

Là-bas, c'est le pays de l'étrange et du rêve,
C'est l'horizon perdu par delà les sommets,
C'est le bleu paradis, c'est la lointaine grève
Où votre espoir banal n'abordera jamais.

Jean Richepin, « Les oiseaux de passage », *La chanson des gueux*

Automata over infinite words have been studied since the early days of automata theory, following the seminal work of Büchi [Büc62], whose goal was to decide fragments of second-order arithmetic. They are roughly defined as automata over finite words, while modifying the acceptance conditions in order to take into account the infiniteness of the input. Such machines enable to lift the notion of *regular languages* to infinite words, leading to the celebrated concept of *ω-regular languages* (see e.g. [PP04]).

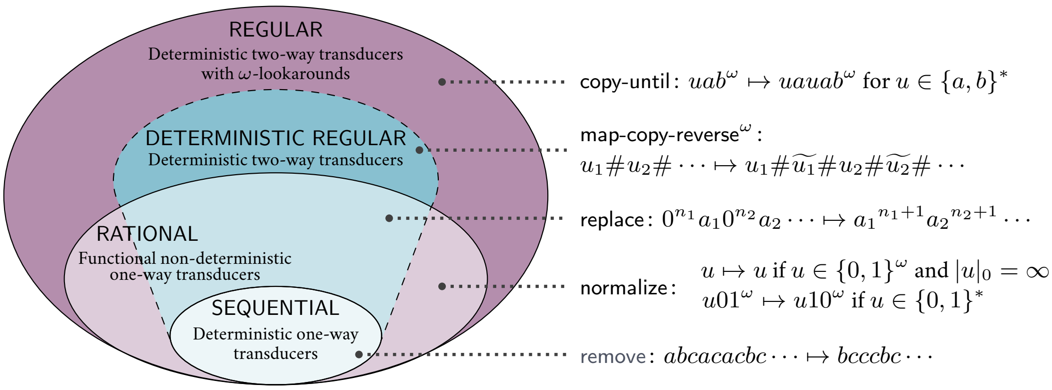


Figure 8.1: Classes of functions over infinite words described in Chapter 8.

This chapter can be seen as the counterpart of Chapter 1 for infinite inputs. More precisely, we shall define in Sections 8.1 and 8.2 the following machine models over infinite words:

- *one-way deterministic transducers*, which define the class of *sequential functions*;
- *one-way non-deterministic transducers*, which define the class of *rational functions*;

- *two-way deterministic transducers*, which define the class of *deterministic regular functions*;
- *two-way deterministic transducers* with *ω-lookarounds* (an adaptation of lookarounds in the setting of infinite words and ω-regular languages), which define the class of *regular functions*.

These various classes are compared in Figure 1.1, In Sections 8.1 and 8.2, we explain that both sequential, rational and regular functions describe robust classes of transductions of infinite words, as witnessed by various characterizations and algorithmic properties. We also discuss which class membership problems are known to be decidable. The class of deterministic regular functions is briefly presented in this chapter, but the detailed study of its properties is deliberately postponed to Chapter 9.

When dealing with practical applications, transducers of infinite words can be seen as a model of *streaming* algorithms to process arbitrary long inputs. However, the classes of rational and regular functions both suffer from a severe downside when it comes to *computability*, which is a major difference with the case of finite words. Indeed, the reader should be convinced that all the functions of finite words studied in Parts I and II are *computable*, in the sense that they can be written in any programming language, or equivalently, computed by a *deterministic Turing machine*. This is no longer the case over infinite words: intuitively, the use of non-determinism or *ω-lookarounds* enables to build the output depending on an ω-regular property of the input (for instance depending on whether it contains infinitely many times a given letter). However, such properties cannot be verified by a deterministic device.

In Section 8.3, we thus formalize the notion of *computability* for functions of infinite words. We recall that one can decide if a given regular function is computable, i.e. if it can effectively be implemented by a streaming algorithm, and build a *Turing machine of infinite words* which computes it. It is conjectured that the computable regular functions are in fact the deterministic regular ones.

8.1 One-way transductions

The goal of this part is to introduce the notions of sequential and rational functions over infinite words. If A is an alphabet, recall that A^ω denotes the set of infinite words over A. We let $A^\infty := A^\omega \cup A^*$. Transducers of infinite words are built by adding outputs to *finite automata of infinite words*, which are classical finite automata with a modified notion of final states. Two notions are well-known:

- *Büchi final conditions*, where a set of *final states* is given in the description. In this case, an infinite run labelled by som infinite input is *final* if it visits infinitely many often a final state;
- *Muller final conditions*, where a set of *final sets of states* is given. In this case, an infinite run labelled by some infinite input is *final* if the set of states visited infinitely often along this run is final.

Non-deterministic or deterministic automata with Muller conditions, or non-deterministic automata with Büchi conditions, describe the same class of languages of infinite words, called *ω-regular languages* and denoted $\omega\mathsf{RegLang}(A)$ (see e.g. [Tho90] for an introduction to their theory).

We say that an ω-regular language is *Büchi deterministic* if it can be recognized by a deterministic automaton with Büchi final conditions. Not every ω-regular language is Büchi deterministic.

Example 8.2 (Non Büchi deterministic language)

Let $A = \{a, b\}$, the language $\{ua^\omega \mid u \in A^*\}$ is ω-regular but not Büchi deterministic.

The next result follows from [Tho90, Theorem 5.3c] and [Tho90, Lemma 5.4].

Proposition 8.3 (Deciding Büchi determinism)

One can decide if an ω-regular language is Büchi deterministic. If this property holds, one can build a Büchi deterministic automaton which computes its.

These two final conditions will also be compared when defining transducer models. Büchi deterministic languages will naturally arise in Chapter 9 when dealing with deterministic regular functions.

8.1.1 Sequential functions

As a first class of functions of infinite words, let us describe the counterpart of sequential functions of finite words. To the knowledge of the author, the model of *one-way deterministic transducer of infinite words* was first investigated in detail in the series of papers [BC00, BC02, BC04].

Definition 8.4 (One-way deterministic transducer of infinite words)

A *one-way deterministic transducer of infinite words* ($1\mathsf{DT}^\omega$) $\mathscr{T} = (A, B, Q, q_0, F, \delta, \lambda)$ consists of:

- an input alphabet A and an output alphabet B;
- a finite set of states Q with an initial state $q_0 \in Q$ and final states $F \subseteq Q$;
- a transition function $\delta\colon Q \times A \rightharpoonup Q$;
- an output function $\lambda\colon Q \times A \rightharpoonup B^*$.

The transition relation $\rightarrow$ and the notion of *run* labelled by a finite or infinite word are defined as for 1DT (over finite words) after Definition 1.3. We say that a run is *initial* if it starts in q_0 and *final* if it visits infinitely often a final state (Büchi final conditions). A run is *accepting* if it is both initial and final. The function $[\![\mathscr{T}]\!]\colon A^\omega \rightharpoonup B^\omega$ computed by $\mathscr{T}$ is defined as follows. Let $u \in A^*$ be the input, then $[\![\mathscr{T}]\!](u)$ is defined if and only if there exists an accepting run $p_0 \xrightarrow{a_1|\alpha_1} p_1 \xrightarrow{a_2|\alpha_2} \cdots$ of $\mathscr{T}$ labelled by u (it has to be unique) whose *output* $\alpha_1\alpha_2\cdots$ is infinite. In this case we let $[\![\mathscr{T}]\!](u) \coloneqq \alpha_1\alpha_2\cdots \in B^\omega$.

Example 8.5 (Removing a letter)

Let $A = \{a, b, c\}$, the function remove: $A^\omega \rightharpoonup \{b, c\}^\omega$, which removes the a in the input with domain $\{u \in A^\omega \mid |u|_b = \infty\}$, is computed by the $1\mathsf{DT}^\omega$ depicted in Figure 8.7a. Final states are denoted by a double circle (recall that in Figure 1.5 we used instead an outing arrow).

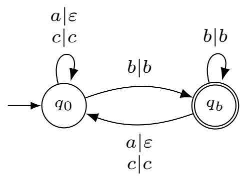
Example 8.6 (Division by 3 in base 2)

Let $A = \{0, 1\}$, then any word of $u \in A^\omega$ can be seen as the binary expansion of some real number $0 \leqslant \gamma \leqslant 1$. The function divide computes the division by 3 on such representations. It is computed by the $1\mathsf{DT}^\omega$ depicted in Figure 8.7b.

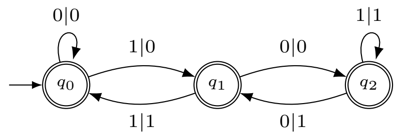
Definition 8.8 (Sequential function of infinite words)

The class of *sequential functions* is the class of functions computed by $1\mathsf{DT}^\omega$.

8.1.1.1 Domains and variants of final conditions. Beware that the output produced along the accepting run of a $1\mathsf{NT}^\omega$ is only taken into account when it is infinite. This restriction forces to define



(a) $1DT^\omega$ computing the function remove.



(b) $1DT^\omega$ computing the function divide.

Figure 8.7: Functions computed by $1DT^\omega$.

functions of type $A^\omega \rightharpoonup B^\omega$ and not $A^\omega \rightharpoonup B^\infty$. In fact, this condition can be encoded syntactically: given a $1DT^\omega$, it is easy to build (by doing a product construction) an equivalent $1DT^\omega$ such that the output along an accepting run is always infinite. As a consequence, one can observe that the domain of a sequential function is (effectively) a Büchi deterministic language.

Conversely, the Büchi conditions cannot be encoded within the condition on infinite inputs. Indeed, one cannot systematically make all states final, as observed in Claim 8.9.

Claim 8.9 (Final states matter for the domain)

The function remove from Example 8.5 cannot be computed by a $1DT^\omega$ with all states final.

Proof idea. If a $1DT^\omega$ computes remove, it has to produce more or less c^n when reading c^n (since otherwise it would not be correct on inputs of shape $c^n b^\omega$). Thus it outputs c^ω on input c^ω, and since this output is infinite, it means that c^ω would belong to $\mathsf{Dom}(\mathsf{remove})$. ◀

In [BC04, Section 2], our sequential functions are called *Büchi sequential functions*. This paper also defines the class of *Muller sequential functions*, by using Muller final conditions instead of Büchi. As observed in [BC04], a Muller sequential function is simply the restriction of a Büchi sequential function to an ω-regular language. Hence the difference between them is a simple matter of domains.

8.1.1.2 Basic properties of sequential functions. It easy to see (using a product construction) that if $f\colon A^\omega \rightharpoonup B^\omega$ is sequential and $L \subseteq \mathsf{Dom}(f)$ is Büchi deterministic, then $f|_L$ is (effectively) a sequential function. Similarly, if $L \subseteq B^\omega$ is Büchi deterministic (resp. ω-regular), then $f^{-1}(L) \subseteq B^\omega$ is (effectively) Büchi deterministic (resp. ω-regular). Using yet another product construction, one can show that sequential functions are (effectively) closed under composition.

As a major difference with finite words, we do lose generalities when restricting our attention to total functions. In other words, sequential functions cannot be "completed". Given $f, g\colon A^\omega \rightharpoonup B^\omega$, we say that g is an *extension* of f if $\mathsf{Dom}(f) \subseteq \mathsf{Dom}(g)$ and for all $u \in \mathsf{Dom}(f)$, $f(u) = g(u)$. The somehow *ad hoc* proof of Claim 8.10 will be re-explained in a more comprehensive fashion through the lens of *continuity* (with respect to a well-chosen topology) in Section 8.3.

Claim 8.10 (Sequential functions cannot be extended)

There is no sequential function which extends the function remove to $\{a, b, c\}^\omega$.

Proof idea. If a $1DT^\omega$ computes an extension of remove, it has to produce ε after reading a^n (since otherwise it would not be correct on inputs of shape $a^n u$ with $u \in \{b, c\}^\omega$ and $|u|_b = \infty$). Thus it must output ε on input a^ω, which means that the extension cannot be total. ◀

Finally, let us recall that given a sequential function computed by a $1DT^{\omega}$, one can effectively compute a canonical $1DT^{\omega}$ which computes it [FGLM18, Section 2].

8.1.2 Rational functions

The study of *rational functions of infinite words* originates from [CP81]. They are obtained by adding Büchi final conditions to the model of one-way non-deterministic transducer of finite words.

Definition 8.11 (One-way non-deterministic transducer of infinite words)

A *one-way non-deterministic transducer of infinite words* ($1NT^{\omega}$) $\mathscr{N} = (A, B, Q, I, F, \Delta, \lambda)$ is:

- an input alphabet A and an output alphabet B;
- a finite set of states Q with initial states $I \subseteq Q$ and final states $F \subseteq Q$;
- a transition relation $\Delta \subseteq Q \times (A \cup \{\varepsilon\}) \times Q$;
- an output function $\lambda \colon \Delta \to B^*$.

The semantics of a $1NT^{\omega}$ is similar to that of a $1NT$ (see Definition 1.7). We write $q \xrightarrow{u|\alpha} q'$ whenever $(q, u, q') \in \Delta$ (beware that here $u \in A \cup \{\varepsilon\}$) and $\alpha = \lambda(q, u, q')$. A *run* labelled by an input word $u_1 \cdots u_n \in A^*$ is a sequence $p_0 \xrightarrow{u_1|\alpha_1} q_1 \cdots \xrightarrow{u_n|\alpha_n} p_n$. The word $\alpha_1 \cdots \alpha_n \in B^*$ is said to be the *output* along the run. We say that the run is *initial* if $p_0 \in I$, and *final* if it visits infinitely often an accepting state. It is *accepting* if it is both initial and final. The relation $[\![\mathscr{N}]\!] \subseteq A^{\omega} \times B^{\omega}$ computed by $\mathscr{N}$, is defined as follows (beware that, here again, we only take infinite outputs into account):

$$[\![\mathscr{N}]\!] \coloneqq \{(u, \alpha) \mid \alpha \in B^{\omega} \text{ is output along some accepting run labelled by } u\}.$$

Example 8.12 (Suffixes)

The relation $\mathsf{suffixes} \subseteq A^{\omega} \times A^{\omega}$ defined by $(u, \alpha) \in \mathsf{suffixes}$ if and only if α is an (infinite) suffix of u can be computed by a $1NT^{\omega}$ inspired by the $1NT$ for factors from Figure 1.5b.

In this manuscript, we shall only focus on functions. The notions of *real-time*, *functional* and *unambiguous* $1NT^{\omega}$ are defined as before (see Definition 1.9). It follows from [Gir86] and [CG99, Corollary 3] that a functional $1NT^{\omega}$ can be transformed in an equivalent real-time and unambiguous $1NT^{\omega}$.

Definition 8.13 (Rational function of infinite words)

The class of *rational functions* is the class of functions computed by functional $1NT^{\omega}$.

Unsurprisingly, rational functions are more expressive than the sequential ones. Indeed, it is easy to show that none of the functions from Examples 8.14 to 8.16 is sequential.

Example 8.14 (Normalization in base 2)

The function $\mathsf{normalize} \colon \{0,1\}^{\omega} \rightharpoonup \{0,1\}^{\omega}$ with domain $\{0,1\}^{\omega} \smallsetminus \{1^{\omega}\}$ which maps $u \mapsto u$ if $|u|_0 = \infty$ and $u01^{\omega} \mapsto u10^{\omega}$ if $u \in \{0,1\}^*$ is computed by the $1NT^{\omega}$ from Figure 8.17a.

Example 8.15 (Replacing factors)

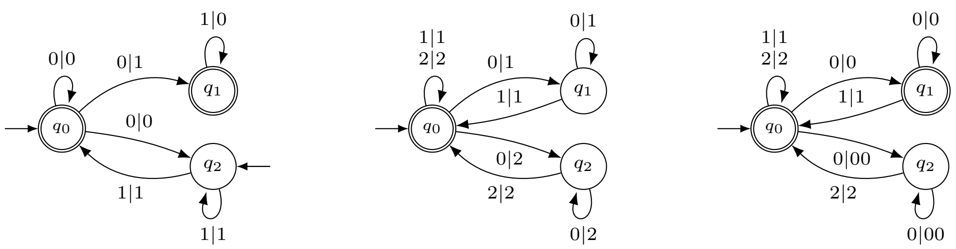
The function replace: $\{0,1,2\}^\omega \rightharpoonup \{1,2\}^\omega$ with domain $\{x \mid |x|_1 = \infty$ or $|x|_2 = \infty\}$ which maps $0^{n_1}a_1 0^{n_2}a_2 \cdots \mapsto {a_1}^{n_1+1}{a_2}^{n_2+1}\cdots$ with $a_i \in \{1,2\}$ and $n_i \geqslant 0$ for $i \geqslant 0$ is computed by the $1\mathsf{NT}^\omega$ from Figure 8.17b.

Example 8.16 (Doubling factors)

The total function double: $\{0,1,2\}^\omega \to \{0,1,2\}^\omega$ which maps for $a_i \in \{1,2\}$:

- $0^{n_1}a_1 0^{n_2}a_2 \cdots \mapsto 0^{a_1 n_1}a_1 0^{a_2 n_2}a_2 \cdots$ if the input has infinitely many 1 or 2;
- $0^{n_1}a_1 \cdots 0^{n_m}a_m 0^\omega \mapsto 0^{a_1 n_1}a_1 \cdots 0^{a_m n_m}a_m 0^\omega$ if the input has finitely many 1 or 2.

is computed by the $1\mathsf{NT}^\omega$ from Figure 8.17c.



(a) $1\mathsf{NT}^\omega$ computing normalize. (b) $1\mathsf{NT}^\omega$ computing replace. (c) $1\mathsf{NT}^\omega$ computing double.

Figure 8.17: Functions computed by unambiguous and real-time $1\mathsf{NT}^\omega$.

8.1.2.1 Domains and final conditions. As for $1\mathsf{DT}^\omega$, it is easy to transform any $1\mathsf{NT}^\omega$ into an equivalent $1\mathsf{NT}^\omega$ such that the output along an accepting run is always infinite. Therefore the domains of rational functions are ω-regular languages (and not necessarily Büchi deterministic languages). As observed in [BC04, Section 2], using Muller final conditions instead of the Büchi conditions for non-deterministic machines would define exactly the same class of functions of infinite words[1].

8.1.2.2 Basic properties of rational functions. It easy to see (using a product construction) that if $f\colon A^\omega \rightharpoonup B^\omega$ is rational and $L \subseteq \mathsf{Dom}(f)$ is ω-regular then $f|_L$ is (effectively) a rational function. Similarly, if $L \subseteq B^\omega$ is ω-regular, then $f^{-1}(L) \subseteq B^\omega$ is (effectively) ω-regular. Using yet another product construction, one can show that rational functions are (effectively) closed under composition.

Since ω-regular languages are closed under taking unions and complements, one can always "complete" a partial rational function into a total one, which outputs a distinguished infinite word when the input is not in the original domain. This differs from the case of sequential functions.

8.1.2.3 Equivalent formalims. The landscape of equivalent formalisms is roughly the same as for rational functions of finite words. From a logical point of view, this class is captured by a (semantic) extension of order-preserving MSO transductions to infinite words [FGLM18, Theorem 30].

[1] This is mainly due to the fact that a non-deterministic automaton with Muller conditions can (rather simply) be transformed in an equivalent non-deterministic automaton with Büchi conditions.

The notion of bimachine can also be extended to infinite words, at the cost of using ω-regular languages for the "right" part of the machine. This idea originates from [Wil16, Section 3] in the case of letter-to-letter transductions. A formal definition was given e.g. in [FGLM18, Section 3].

Definition 8.18 (Bimachine of infinite words)

An *ω-bimachine* $\mathscr{B} = (A, B, \lambda)$ consists of:

- an input alphabet A and an output alphabet B;
- an output function $\lambda\colon \mathsf{RegLang}(A) \times A \times \omega\mathsf{RegLang}(A) \rightharpoonup B^*$ such that:
  (1) $\mathsf{Dom}(\lambda)$ is finite;
  (2) for all $(L, a, R) \neq (L', a, R') \in \mathsf{Dom}(\lambda)$, $RaL \cap R'aL' = \varnothing$.

The semantics of an ω-bimachine is defined in a similar fashion to that of a bimachine of finite words, with the only difference that the suffix starting in some position is now infinite. Furthermore, we exclude input words whose output is finite. The next result originates from [FGLM18, Theorem 13].

Proposition 8.19 (Rational = Bimachine)

A function of infinite words is rational if and only if it can be computed by an ω-bimachine.

It follows from [FGLM18, Theorem 29] that given a rational function, one can in fact build a canonical ω-bimachine which computes it. This result was used in [FGLM18, Theorem 31] to decide whether a rational function can be described by an *order-preserving first-order transduction*.

In the case of finite words, we have shown in Proposition 1.12 that rational functions are compositions of sequential functions (computed by *deterministic* 1NT) and sequential functions from right to left (computed by *co-deterministic* 1NT). Over infinite words, co-determinism is not well-behaved since it does not imply unambiguity. Indeed, since there is no right end in the input, a co-deterministic machine can have several final or accepting runs labelled by same input. [CM03] introduces the concept of *prophetic*[2] automata in order to obtain a better notion of co-determinism. Formally, an automaton with Büchi final conditions is said to be prophetic if it has at most one final run labelled by any given infinite input. The main result of [CM03] states that prophetic automata with Büchi final conditions effectively capture the class of all ω-regular languages. We say that a $1\mathsf{NT}^{\omega}$ is *prophetic* if its underlying automaton is so. The class of functions computed by such machines was studied in [Car10][3].

Definition 8.20 (Prophetic functions of infinite words)

The class of *prophetic functions* is the class of functions computed by prophetic $1\mathsf{NT}^{\omega}$.

Example 8.21 (Normalization in base 2)

The $1\mathsf{NT}^{\omega}$ presented in Figure 8.17a is prophetic, hence so is the function $\mathsf{normalize}$.

The author believes that the class of prophetic functions is worth being studied, since they act as the dual of sequential functions. Furthermore, it provides the analogue of Proposition 1.12, as stated in the next result which originates from [Car10] (see also [FGLM18, Corollary 18]).

[2] Those machines are called *unambiguous* in [CM03]. This terminology is no longer used, since the prophetic condition on final runs is more restrictive than classical unambiguity which only deals with accepting runs.

[3] The terminology of [Car10] for this class is *right-sequential functions* whereas we call them *prophetic functions* instead.

Proposition 8.22 (Rational = sequential ∘ prophetic)

A function of infinite words is rational if and only if it can be written as a composition $f \circ g$ where where f is a sequential function and g is a prophetic function.

8.1.2.4 Decision problems. The landscape of equivalence problems for rational functions of infinite words is similar to the case of finite words. Indeed, it is known since [CP81, Corollary 3] that equivalence of rational functions is decidable, while the same problem for relations computed by $1\mathrm{NT}^\omega$ is undecidable [CP81, Theorem 6]. Furthermore, one can decide if a $1\mathrm{NT}^\omega$ is functional [Gir86, Corollary 3.4].

Now, let us focus on the class membership problem from rational to sequential functions. The two next results originate from [BC02, Section 3][4] and [BC04, Section 3]. The proof consists in adapting the twinning properties used for showing Theorem 1.17 to the case of finite words.

The author is convinced that the most interesting statement is Theorem 8.23, which decides whether a rational function can be extended to a sequential one. Indeed, once this result is shown, being exactly a sequential function is just a matter of domains, as illustrated below in the (easy) proof of Corollary 8.24. Furthermore, computing an extension is trouble-free for practical applications, if we assume that the environment always provides "correct" inputs, i.e. words which belong to the domain. This is the basic idea behind the notion of *good-enough synthesis* introduced in[5] [AK20, Section 2].

Theorem 8.23 (Rational → Sequential extension)

One can decide if a rational function of infinite words has an extension which is sequential. If this property holds, one can build a $1\mathrm{DT}^\omega$ which computes an extension.

Corollary 8.24 (Rational → Sequential)

One can decide if a rational function of infinite words is sequential. If this property holds, one can build a $1\mathrm{DT}^\omega$ which computes it.

Proof. Observe that a rational function f is (effectively) sequential if and only if it can be extended to a sequential function and its domain is Büchi deterministic. Indeed, a sequential function can be restricted to any Büchi deterministic language, as claimed in Section 8.1.1.2. We conclude thanks to Proposition 8.3 and Theorem 8.23. ◀

We conclude this section by conjecturing that another class membership problem for rational functions can be decided. Conjecture 8.25 seems to be less meaningful for practical applications than Corollary 8.24, since prophetic functions are computed by non deterministic devices.

Conjecture 8.25 (Rational → Prophetic)

One can decide if a rational function is prophetic, by adapting the techniques of [BC04].

[4] In the particular case where all states of the $1\mathrm{NT}^\omega$ are final, and in this case they build a $1\mathrm{DT}^\omega$ with all states final.

[5] Curiously, good-enough synthesis is not defined in the same fashion in [FLW20, Section 1]: here they explain that the domain of the specification must be preserved, which is not explicitly required in [AK20].

8.2 Regular and deterministic regular functions

In this section, we study the generalization of two-way deterministic transducers to infinite words. The most striking phenomenon is that, contrary to the case of finite words, *two-way deterministic transducers of infinite words* cannot in general simulate $\mathsf{1NT}^\omega$. Indeed, a $\mathsf{1NT}^\omega$ can e.g. build its output depending on whether the input contains infinitely many times a given letter, which is not possible for a deterministic machine. Therefore, the class of *regular functions of infinite words* was instead defined in [AFT12] as the class of functions computed by two-way transducers with *ω-lookarounds* (which generalize lookarounds to infinite words). Thanks to this *ω-lookaround* feature, $\mathsf{1NT}^\omega$ can be simulated.

8.2.1 Two-way transducers

A *two-way transducer of infinite words* is roughly defined as a two-way transducer of finite words. The only syntactical difference is that the symbol $\dashv$ is no longer needed, since the input has no right border. This model was first mentioned in [AFT12], even if it is not the core of this paper.

Definition 8.26 (Two-way deterministic transducer of infinite words)

A *two-way deterministic transducer of infinite words* ($\mathsf{2DT}^\omega$) $\mathscr{T} = (A, B, Q, q_0, F, \delta, \lambda)$ consists of:

- an input alphabet A and an output alphabet B;
- a finite set of states Q with an initial state $q_0 \in Q$ and a set $F \subseteq Q$ of final states;
- a transition function $\delta \colon Q \times (A \uplus \{\vdash\}) \rightharpoonup Q \times \{\triangleleft, \triangleright\}$;
- an output function $\lambda \colon Q \times (A \uplus \{\vdash\}) \rightharpoonup B^*$ with same domain as δ.

Given a finite or infinite word $u \in (A \cup \{\vdash\})^\infty$, the notions of *configuration* and *transition relation* are defined as in the case of finite words (see Definition 1.19). A *run* of $\mathscr{T}$ labelled by u is a finite or infinite sequence of configurations $(q_1, i_1) \to (q_2, i_2) \to \cdots$. We say that a run is *initial* if it starts in $(q_0, 1)$, and *final* if it is infinite, $i_j \to \infty$ and $q_j \in F$ occurs infinitely often (Büchi conditions[6]). The condition $i_j \to \infty$ means that u is infinite and that $\mathscr{T}$ visits arbitrary large positions of u. Therefore it forbids looping behaviors on a prefix of u. The run is *accepting* if it is both initial and final.

The partial function $[\![\mathscr{T}]\!] \colon A^\omega \rightharpoonup B^\omega$ computed by $\mathscr{T}$ is defined as follows. Let $u \in A^\omega$ be the input, then $[\![\mathscr{T}]\!](u)$ is defined if and only if there exists a (necessarily unique) accepting run $(q_1, i_1) \to \cdots$ labelled by $\vdash u$, whose *output* $\lambda(q_1, \vdash u[i_1]) \cdots$ is infinite. In this case, we let $[\![\mathscr{T}]\!](u)$ be this output.

Example 8.27 (Replacing factors)

The function $\mathsf{replace}$ from Example 8.15 can be computed by $\mathsf{2DT}^\omega$. For each $i \geqslant 1$, this $\mathsf{2DT}^\omega$ crosses the block 0^{n_i} to determine a_i, and then crosses it again to output ${a_i}^{n_i+1}$.

Example 8.28 (Map copy reverse)

Let us extend the function $\mathsf{map\text{-}copy\text{-}reverse}$ to infinite words. Let A be an alphabet, we define $\mathsf{map\text{-}copy\text{-}reverse}^\omega \colon (A \uplus \{\#\})^\omega \to (A \uplus \{\#\})^\omega$ as follows:

- $\mathsf{map\text{-}copy\text{-}reverse}^\omega(u_1 \# u_2 \# \cdots) := u_1 \# \widetilde{u_1} \# u_2 \# \widetilde{u_2} \# \cdots$ with $u_i \in A^*$ for all $i \geqslant 0$;
- $\mathsf{map\text{-}copy\text{-}reverse}^\omega(u_1 \# \cdots \# u_n \# u) := u_1 \# \widetilde{u_1} \# \cdots \# u_n \# \widetilde{u_n} \# u$ with $u \in A^\omega$.

This function can be computed $\mathsf{2DT}^\omega$ which makes several passes on each $\#$-free factor (but only

[6]One could also use Muller conditions, which would only affect the domains of the computed functions. The (seemingly arbitrary) choice of Büchi conditions is motivated by the fact that they can be removed (Proposition-Definition 9.16).

once for the last infinite factor whenever it exists).

The class of functions computed by $2\mathrm{DT}^\omega$ was named in [CD22, Section 3].

Definition 8.29 (Deterministic regular functions of infinite words)

The class of *deterministic regular functions* is the class of functions computed by $2\mathrm{DT}^\omega$.

Chapter 9 is devoted to a rather extensive study of deterministic regular functions, by presenting the contributions of the author published in [CD22, CDFW23][7]. In order not to overlap with them, we do deliberately not state results on deterministic regular functions in the current chapter.

For the moment, let us simply observe that deterministic regular and rational functions are not included in each other. This statement was already claimed informally in [AFT12]. Intuitively, the argument is that deterministic regular functions cannot check ω-regular properties of their input, while conversely rational functions fail to duplicate arbitrarily large portions of their input.

Proposition 8.30 (Rational and deterministic regular are not comparable)

The classes of deterministic regular functions of infinite words and rational functions of infinite words are not comparable (even up to extension). In particular:

- the function normalize has no deterministic regular extension;
- the (total) function $\mathsf{map\text{-}copy\text{-}reverse}^\omega$ is not rational.

Proof sketch. Assume that an extension of normalize is computed by a $2\mathrm{DT}^\omega$. By leveraging the techniques of Section 2.2.2, there exist $\alpha, \beta \in \{0,1\}^*$ and $M, N \geqslant 1$ such that for all input $01^{M+NX}u$ with $X \geqslant 0$ and $u \in \{0,1\}^\omega$, the accepting run of $\mathscr{T}$ has produced $\alpha\beta^X$ at the time it visits the first position of u for the first time. Since $\mathsf{normalize}(01^\omega) = 10^\omega$, one has $\beta \in \{0\}^+$, which contradicts the fact that $\mathsf{normalize}(01^{M+NX}0^\omega) = 01^{M+NX}0^\omega$ for all $X \geqslant 0$.

Now let $a \in A$ and assume that $\mathsf{map\text{-}copy\text{-}reverse}^\omega$ is computed by a functional $1\mathrm{NT}^\omega$. By using similar pumping arguments, there exist $M_0, M_1, N \geqslant 1$, $\alpha, \beta \in A^*$ and $\gamma \in A^\omega$ such that $\mathsf{map\text{-}copy\text{-}reverse}^\omega(a^{M_0}a^{NX}a^{M_1}\#a^\omega) = \alpha\beta^X\gamma^\omega$ for all $X \geqslant 0$, yielding a contradiction. ◀

The case of normalize will be re-explained thanks to continuity in Section 8.3.

8.2.2 Two-way transducers with ω-lookaround

In this section, we extend the model of $2\mathrm{DT}^\omega$ with an extra feature called ω-lookarounds, inspired by the lookarounds over finite words. Informally, a $2\mathrm{DT}^\omega$ with ω-lookarounds is able to select its transitions depending on a ω-regular property of its input where the current position is distinguished.

Definition 8.31 (Two-way transducer with ω-lookarounds)

A two-way deterministic transducer ($2\mathrm{DT}^\omega$) with *ω-lookarounds* consists of a modified two-way deterministic transducer $\mathscr{T} = (A, B, Q, q_0, F, \delta, \lambda)$ such that:

- the transition function δ has type $(Q \times \mathsf{RegLang}(A) \times A \times \omega\mathsf{RegLang}(A)) \rightharpoonup Q$;
- the output function λ has type $(Q \times \mathsf{RegLang}(A) \times A \times \omega\mathsf{RegLang}(A)) \rightharpoonup B^*$;

[7] The model of $2\mathrm{DT}^\omega$ was also used by the author in [Dou18] for very specific purposes.

- $\mathsf{Dom}(\delta) = \mathsf{Dom}(\lambda)$ and this set is finite;
- for all $(q, L, a, R) \neq (q, L', a, R') \in \mathsf{Dom}(\delta)$, we have $LaR \cap L'aR' = \varnothing$.

The semantics of a $\mathsf{2DT}^\omega$ with ω-lookarounds is similar to that of $\mathsf{2DT}$ with lookarounds. Formally, given $u \in A^\omega$ and (q, i) a configuration of $\mathscr{T}$ over u, then by the last item of Definition 8.31 (which ensures determinism) there exists at most one tuple $(q, L, u[i], R) \in \mathsf{Dom}(\delta)$ such that $u[1{:}i{-}1] \in L$ and $u[i{+}1{:}] \in R$. The transition from (q, i) is chosen in accordance with $\delta(q, L, u[i], R)$.

As for finite words, observe that having a symbol $\vdash$ is no longer useful with ω-lookarounds.

Example 8.32 (Copy until)

Let $A = \{a, b\}$ and consider the function $\mathsf{copy\text{-}until} \colon A^\omega \rightharpoonup A^\omega$ which maps $uab^\omega \mapsto uauab^\omega$ for $u \in A^*$. It can be computed by a $\mathsf{2DT}^\omega$ with ω-lookarounds which uses the ω-lookaround in each position labelled by a to check if the suffix starting in this position is b^ω.

Example 8.33 (Rational functions using an ω-lookaround)

It is easy to see that any ω-bimachine can be simulated by a $\mathsf{2DT}^\omega$ with ω-lookarounds which has a single state. As a consequence, any rational function can be computed by this model.

Since $\mathsf{2DT}^\omega$ do not compute all rational functions (Proposition 8.30), it follows from Example 8.33 that ω-lookarounds cannot be removed over infinite words. As mentioned above, this major difference with finite words follows from the fact that deterministic machines cannot check ω-regular properties of infinite suffixes. We shall see in Section 9.1.2 that the *lookbehind* part (i.e. the component in $\mathsf{RegLang}(A)$ of the transition function) can however be removed using the classical tree construction of [HU67] (which was the key for showing Theorem 1.30), at the cost of adding a $\vdash$ symbol.

Definition 8.34 (Regular functions of infinite words)

The class of *regular functions* is the class of functions computed by $\mathsf{2DT}^\omega$ with ω-lookarounds.

Note that regular functions subsumes both rational and deterministic regular functions, because they are able at the same time check ω-regular properties and duplicate large portions of their input.

Proposition 8.35

The function $\mathsf{copy\text{-}until}$ has no extension which is rational or deterministic regular.

Proof idea. The arguments are more or less the same as those of Proposition 8.30. To show by contradiciton that the function is not deterministic regular, we determine some $M, N \geqslant 1$ and study the behavior of a $\mathsf{2DT}^\omega$ on inputs which begin with ab^{M+NX} for $X \geqslant 0$. ◀

8.2.2.1 Domains and acceptance conditions. We first claim that one can always transform a $\mathsf{2DT}^\omega$ with ω-lookarounds in an equivalent machine whose states are all final. Indeed, the acceptance condition can be encoded within the ω-lookarounds. The proof would consist in adapting the classical transformation from two-way to one-way automata [She59] to infinite words (see also Proposition-Definition 9.16 for the case of deterministic regular functions) in order to roughly show that two-way automata of infinite words with Büchi conditions only compute ω-regular languages.

One can similarly show that the domain of a regular function is an ω-regular language. Furthermore, for rational functions, we do not lose generalities if we assume that regular functions are total.

As mentioned in [DFKL20], the model of functional *non-deterministic two-way transducer of infinite words*, with either Muller or Büchi final conditions, would describe a strict subclass of regular functions. Informally, the reason is that for computing the function copy-until, one needs a non-deterministic guess to identify the last occurrence of letter a and after this guess, check that there are only b symbols on the input. However, two passes on the input are necessary, and the same non-deterministic guesses must be done at the same positions, which is impossible to ensure for a finite state machine.

8.2.2.2 Basic properties and equivalent models. We first claim in Theorem 8.36 that regular functions are closed under composition. This result is implicit in [AFT12], using the correspondence with MSO transductions. It can be shown directly by adapting the proof of a Theorem 1.31 over finite words, and relying on ω-lookarounds (which cannot be removed). The reader is invited to consult Section 9.5 for a similar proof in the setting of deterministic regular functions.

Theorem 8.36 (Composition of regular functions)

The class of regular functions of infinite words is (effectively) closed under composition.

As an easy consequence of this result, one can show that if f is regular and $L \subseteq B^\omega$ is ω-regular, then $f^{-1}(L) \subseteq B^\omega$ is (effectively) an ω-regular language.

Several equivalent descriptions of regular functions of infinite words have been studied. It follows from [AFT12, Proposition 1] that this class coincides with the functions computed by the (semantic) extension of monadic second-order transductions to infinite words[8]. This paper also provides an equivalent model of *streaming string transducers*, generalizing the constructions of [AC10]. A formalism which uses combinators, in the spirit of regular expressions, is described in [DGK18].

8.2.2.3 Decision problems. The next result originates from [AFT12, Theorem 3].

Theorem 8.37 (Equivalence of regular functions)

Given two regular functions of infinite words $f, g \colon A^\omega \rightharpoonup B^\omega$, one can decide if $f = g$.

To the knowledge of the author, the decision problem from regular functions to rational functions is open. We conjecture that this result could be tackled by adapting similar techniques over finite words.

Conjecture 8.38 (Regular $\to$ Rational)

One can decide if a regular function of infinite words (or at least a deterministic regular one) is rational, by adapting e.g. techniques of [FGRS13] to infinite words.

8.3 Computability and continuity

Neither rational nor regular functions are meaningful for building *streaming algorithms*. Indeed, they can e.g. check if the input contains infinitely many times the same symbol, which is irrelevant in practice[9].

[8] This logical correspondence was in fact the initial motivation for introducing regular functions in [AFT12].
[9] For practical streaming applications, the input would never be really infinite, but only arbitrarily long.

To explain what is meant by “algorithm” over infinite words, we describe the model of *Turing machine of infinite words*, which has been used for long to study computability over real numbers [Wei00].

Formally, a *deterministic Turing machine of infinite words* (TM^ω) computing a function $f\colon A^\omega \rightharpoonup B^\omega$, consists in a classical Turing machine which uses 3 distinct tapes:

- a two-way read-only *input tape*, which contains the input $u \in A^\omega$;
- a two-way read-write *working tape* which is used to do internal computations;
- a one-way (write-only) *output tape*, where the output $f(u) \in B^\omega$ (when $u \in \mathsf{Dom}(f)$) is produced in a streaming fashion (since left moves are not allowed, one cannot rewrite the output).

The behavior of such a Turing machine is depicted in Figure 8.39. It can be seen as a $\mathsf{2DT}^\omega$ enhanced with a read-write working tape. Its semantics is defined in a similar fashion.

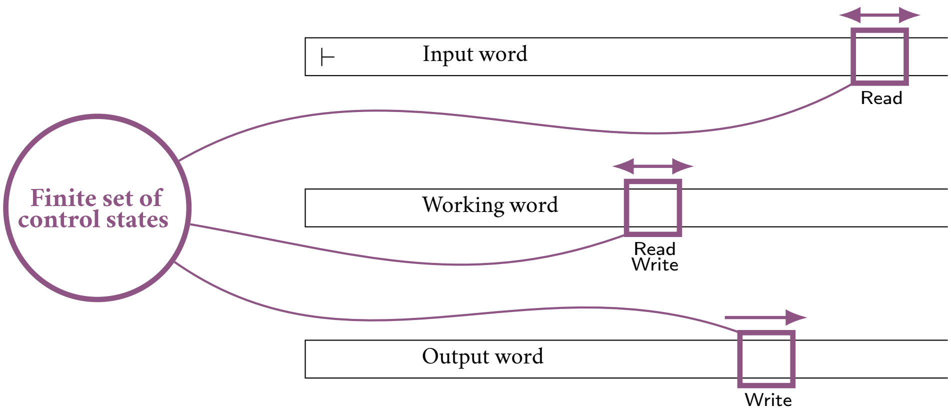


Figure 8.39: Behavior of a Turing machine of infinite words.

Definition 8.40 (Computable functions of infinite words)

The class of *computable functions* is the class of functions computed by TM^ω.

The goal of this section is to recall known results on computable regular functions. For this purpose, we also introduce the notion of *continuity* for functions over infinite words. Let A be an alphabet. Given $u, v \in A^\infty$ we denote by $u \wedge v \in A^\infty$ the longest common prefix of u and v.

Proposition-Definition 8.41 (Topology over infinite words)

The function $\mathsf{d}\colon A^\omega \times A^\omega \to \mathbb{Q}$ such that $\mathsf{d}(u,v) := 2^{-|u \wedge v|}$ defines a distance on A^ω.

The *continuity* of a function $f\colon A^\omega \rightharpoonup B^\omega$ is defined with respect to the topology induced by the distance d on A^ω and B^ω. In an explicit fashion, this means that the function f is *continuous* in a given word $u \in \mathsf{Dom}(f)$ if for all $N \geqslant 0$, there exists $M \geqslant 0$ such that if $v \in \mathsf{Dom}(f)$ with $|u \wedge v| \geqslant M$, then $|f(u) \wedge f(v)| \geqslant N$. Furthermore f is *continuous* when continuous in any word of its domain[10].

Example 8.42 (Points of non-continuity)

The function $\mathsf{normalize}$ is not continuous in 01^ω. Indeed $\mathsf{normalize}(01^\omega) \wedge \mathsf{normalize}(01^X 0^\omega) = \varepsilon$ for all $X \geqslant 0$. The function $\mathsf{copy\text{-}until}$ is not continuous in any word of its domain. Indeed, let

[10] This notion of continuity does not coincide with the *continuity* over finite words studied in [CCP17]. Indeed, the latter defines topologies through varieties of languages (i.e. they ask whether the function preserves certain languages by inverse images).

uab^ω be such a word with $u \in \{a,b\}^*$. We have $\text{copy-until}(uab^X ab^\omega) = uab^X auab^X ab^\omega$ for all $X \geqslant 0$, thus $|\text{copy-until}(uab^X ab^\omega) \wedge \text{copy-until}(uab^\omega)| \leqslant 2|ua|$.

It is known since [Pri01, Proposition 4][11] that one can decide if a rational function is continuous. This result was extended to regular functions in [DFKL20, Theorem 16]. This recent paper also provides improved complexity bounds for the case of rational functions. The basic idea is that the runs of a 2DT^ω with ω-lookarounds which computes a continuous function have specific patterns, inspired of the twinning properties used in the historical proofs of Theorems 1.17 and 8.23 (see Lemma 10.8 for 1NT^ω).

Theorem 8.43 (Deciding continuity)

One can decide if a regular function of infinite words $f\colon A^\omega \rightharpoonup B^\omega$ is continuous.

It is easy to observe that if a function $f\colon A^\omega \rightharpoonup B^\omega$ is computable (even up to extension), then it has to be continuous. Indeed $u \in \text{Dom}(f)$ and $M \geqslant 0$, there exists a position $N \geqslant 0$ such that the TM^ω computing f produces $f(u)[1{:}M]$ while visiting only the positions of $u[1{:}N]$ in its input. Hence $f(u)[1{:}M]$ is a also a prefix of $f(v)$ whenever of $|u \wedge v| \geqslant N$. Thus, one can simply re-prove that neither normalize nor copy-until have a deterministic regular extension (cf. Propositions 8.30 and 8.35). Conversely, a continuous function has no reason to be computable, as illustrated in Example 8.44.

Example 8.44 (Continuous non-computable function)

Let $A := \{0,1\}$. For all $u \in A^\omega$, the function $1^\omega \mapsto u$ with singleton domain is continuous. However, it is not computable as long as u is not computable.

Nevertheless, continuity and computability coincide within the class of regular functions, as claimed in Theorem 8.45. This result originates from [DFKL20, Theorem 6][12], and it also enables to build a TM^ω computing the function. The latter is meaningful in practice for *program synthesis*: given a specification (e.g. using an MSO transduction which describes a regular function), one can automatically build an algorithm which realizes it, whenever it exists, and say that is does not exist otherwise.

Theorem 8.45 (Regular $\rightarrow$ Computable extension)

A regular function of infinite words can be extended to a computable function if and only if it is continuous. If this property holds, one can build a TM^ω which computes an extension.

Starting from a "simple" 2DT^ω with ω-lookarounds to obtain a "complex" Turing machine is somehow disappointing and may be inefficient for implementing the function. Following [DFKL20, Section 6], we conjecture that the TM^ω can always be replaced by a 2DT^ω in Theorem 8.45.

Conjecture 8.46 (Continuous regular functions are deterministic regular)

A regular function of infinite words can be extended to a deterministic regular function if and only if it is continuous. If this property holds, one can build a 2DT^ω which computes an extension.

This conjecture is believed to be rather difficult. The goal of Chapter 10 is to provide a partial answer, by showing that a rational function of infinite words can be extended to a deterministic regular function if and only if it is continuous. This theorem is the most original result of Part III.

[11] This paper contains light mistakes which are fixed in [Pri02].

[12] Their result is even stronger: they show that continuity and computability coincide within the class of functions which preserve ω-regular languages by inverse images (which is the case of regular functions, as mentioned in Section 8.2.2.2).

Chapter 9

Deterministic regular functions of infinite words

UN COQ, *au Paon*
Maître, lequel de nous mettrez-vous à la mode ?

UN PADOUE, *s'avançant en hâte*
Moi ! – J'ai l'air d'un palmier !

UN CHINOIS, *repoussant le Padoue*
Et moi, d'une pagode !

Edmond Rostand, *Chantecler*

In Chapter 8, we have defined the class of deterministic regular functions. The goal of the current chapter is to study its properties in detail and demonstrate that it is a robust and natural class of functions computable by simple devices. To that extent, deterministic regular functions turn out to be more relevant than the regular ones, at least when dealing with practical computable applications.

We shall describe several formalisms which capture deterministic regular functions. The equivalence proofs between these models are somehow entangled, as depicted in Figure 9.1. Solid arrows denote the proofs presented in Chapter 9. Those which require a large amount of additional work with respect to the case of finite words are highlighted in bold. Dashed arrows denote syntactic restrictions.

Over infinite words, $\mathsf{2DT}^{\omega}$ with ω-lookarounds are able to check infinite properties of their input, and therefore have non-computable behaviors. We describe in Section 9.1 a weaker feature called *finite lookarounds*, which enables to check a property of a finite prefix of the input. Thus it accounts for properties which are not local, but still finite. We show that finite lookarounds can effectively be removed for $\mathsf{2DT}^{\omega}$. The author is not aware of a direct proof of this result (similar to the proof over finite words), and our proof instead uses streaming string transducers of infinite words as an intermediate model. The ability to remove finite lookarounds will be used for showing the main result of Chapter 10.

In Section 9.2, we describe a generalization of *streaming string transducers* to infinite words. This model has a distinguished output register which is updated in an append-only fashion. We show that deterministic regular functions are exactly the functions computed by streaming string transducers of infinite words which are copyless, or equivalently K-bounded for some $K \geqslant 0$.

We then show in Section 9.5 that deterministic regular functions are closed under composition, by adapting the classical proof over finite words, and crucially relying on finite lookarounds. In Section 9.6

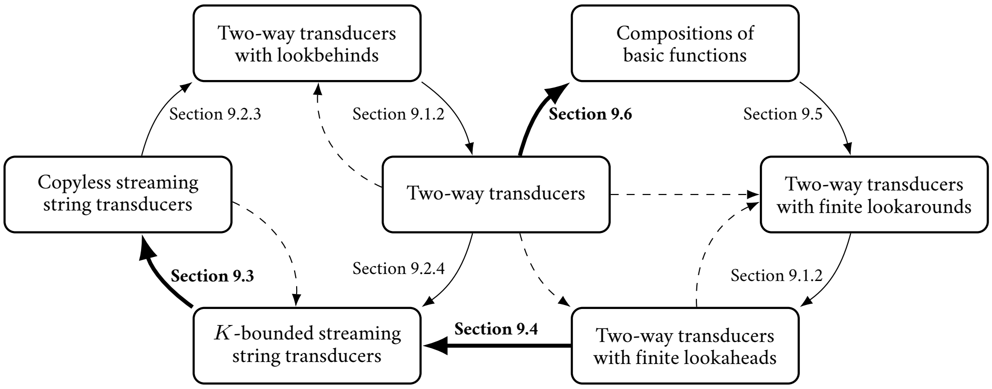


Figure 9.1: Equivalent models presented in Chapter 9 for deterministic regular functions.

we claim that, conversely, deterministic regular functions can be decomposed, i.e. written as compositions of simpler basic functions (which include the sequential ones).

Finally, we discuss in Section 9.7 the generalizations of pebble transducers or marble transducers to infinite words. In this context, we conjecture that obtaining optimization results (even for streaming string transducers, recall Chapter 4) is far more complex than over finite words.

The contributions presented in this chapter are based on part of the results of [CDFW23] and on the theorems of [CD22] which deal with streaming string transducers of infinite words. [CDFW23] also provides a model of *guarded* logical transductions which is equivalent to deterministic regular functions, but we chose not to present it here since we have never dealt with logic in this manuscript[1].

9.1 Two-way transducers with finite lookarounds

The class of deterministic regular functions is built by prohibiting the use of ω-lookarounds for two-way transducers of infinite words (contrary to regular functions). The goal of Section 9.1 is to introduce a weaker feature called *finite lookarounds*, which enable to test non-local but still finite properties of the input. We show that finite lookarounds can effectively be removed, which provides a satisfying analogue of the situation over finite words (even if the proof techniques are more involved).

9.1.1 Finite lookarounds

Intuitively, a $2\mathrm{DT}^\omega$ with *finite lookarounds* is able to check a regular property of a *finite* prefix of its input, in which the current position is distinguished. For instance, this machine can choose its transition depending on whether letter 1 or 2 will occur in the future (see Example 9.3). However, it cannot check that neither a 1 nor a 2 occurs, since this property does not depends on a finite prefix of the input.

Definition 9.2 (Two-way transducer with finite lookarounds)

A two-way deterministic transducer ($2\mathrm{DT}^\omega$) with *finite lookarounds* consists of a modified two-way deterministic transducer $\mathscr{T} = (A, B, Q, q_0, F, \delta, \lambda)$ such that:

- ▶ the transition function δ has type $(Q \times \mathsf{RegLang}(A) \times A \times \mathsf{RegLang}(A)) \rightharpoonup Q$;

[1] Furthermore, the proof of this logic-transducer equivalence is really similar to the historic proof of [EH01] over finite words, once the question of removing finite lookarounds over infinite words is settled.

- the output function λ has type $(Q \times \mathsf{RegLang}(A) \times A \times \mathsf{RegLang}(A)) \rightharpoonup B^*$;
- $\mathsf{Dom}(\delta) = \mathsf{Dom}(\lambda)$ and this set is finite;
- for all $(q, L, a, R) \neq (q, L', a, R') \in \mathsf{Dom}(\delta)$, we have $LaR \cap L'aR' = \varnothing$.

The semantics of a $2\mathsf{DT}^\omega$ with finite lookarounds is built upon that of $2\mathsf{DT}^\omega$ with ω-lookarounds, with the essential difference that the transitions only depend on a finite prefix of the input. Formally, given a configuration (q, i), we say that $(q, L, a, R) \in \mathsf{Dom}(\delta)$ is *admissible* if $u[1{:}i{-}1] \in L$ and there exists some $i \leqslant j$ such that $u[i{+}1{:}j] \in R$. In this case, we say that $u[i{+}1{:}j]$ is a *witness of admissibility*. In order to ensure determinism, the transition which is triggered is the (unique thanks to the last item of Definition 9.2) one which has the shortest witness of admissibility.

Note that $2\mathsf{DT}^\omega$ with finite lookarounds can be seen as a particular case of $2\mathsf{DT}^\omega$ with ω-lookarounds.

Example 9.3 (Replace)

The function replace from Example 8.15 can be computed by a $2\mathsf{DT}^\omega$ with finite lookarounds. The latter uses its finite lookarounds to determine whether the current factor 0^{n_i} will end with letter 1 or letter 2 and chooses its output accordingly. However, if the suffix starting in the current position is 0^ω (i.e. there is no 1 nor 2 in the future), no transition can be enabled.

Now, we claim that finite lookarounds can be removed. As mentioned before Theorem 1.30, over finite words this task can be handled using the tree construction from [HU67, Lemma 3]. However, this construction crucially relies on the fact that the input is finite, thus when moving right we are ensured to meet a $\dashv$ symbol at some point. The proof of Theorem 9.4 is substantially different: it uses a detour through *streaming string transducers of infinite words* and goes over Sections 9.1 to 9.4.

Theorem 9.4 (Finite lookarounds removal)

Given a $2\mathsf{DT}^\omega$ with finite lookarounds, one can build an equivalent $2\mathsf{DT}^\omega$.

Proof. We first transform the $2\mathsf{DT}^\omega$ with finite lookarounds in a $2\mathsf{DT}^\omega$ with finite lookaheads by Theorem 9.9. Then we use Theorem 9.30 to build an equivalent 1-bounded streaming string transducer of infinite words. The latter is transformed in an equivalent $2\mathsf{DT}^\omega$ by Theorem 9.13. ◀

As a side notion, one can define a $1\mathsf{DT}^\omega$ with *finite lookarounds* as a $2\mathsf{DT}^\omega$ with finite lookarounds which only uses right moves. Observe that the $2\mathsf{DT}^\omega$ with finite lookarounds described in Example 9.3 is in fact a $1\mathsf{DT}^\omega$ with finite lookarounds. In this setting, finite lookarounds cannot be removed.

Claim 9.5 (Non-lookaround removal for $1\mathsf{DT}^\omega$)

The function replace can be computed by a $1\mathsf{DT}^\omega$ with finite lookarounds, but it is not sequential.

9.1.2 Lookbehinds and finite lookaheads

The finite lookarounds of a $2\mathsf{DT}^\omega$ can roughly be decomposed in two parts. First, they consist of *look-behinds* which check properties of the prefix ending in the current position (this case is very similar to finite words). Second, they use *finite lookaheads* which check properties of a finite prefix of the suffix starting in the current position. This motivates the definition of two variants of finite lookaheads.

Definition 9.6 (Two-way transducer with lookbehinds)

A two-way deterministic transducer ($2DT^\omega$) with *lookbehinds* consists of a modified two-way deterministic transducer $\mathscr{T} = (A, B, Q, q_0, F, \delta, \lambda)$ such that:

- the transition function δ has type $(Q \times \mathsf{RegLang}(A) \times A) \rightharpoonup Q$;
- the output function λ has type $(Q \times \mathsf{RegLang}(A) \times A) \rightharpoonup B^*$;
- $\mathsf{Dom}(\delta) = \mathsf{Dom}(\lambda)$ and this set is finite;
- for all $(q, L, a) \neq (q, L', a) \in \mathsf{Dom}(\delta)$, we have $L \cap L' = \varnothing$.

Intuitively, this machine only checks a property of the prefix. The semantics of a $2DT^\omega$ with lookbehinds is defined as that of a $2DT^\omega$ with finite lookarounds whose right component in $\mathsf{RegLang}(A)$ is always A^*, which means that it only checks a trivial property of the infinite suffix.

Theorem 9.7 (Lookbehind removal)

Given a $2DT^\omega$ with lookbehinds, one can build $2DT^\omega$ which computes the same function. Furthermore, if a $2DT^\omega$ with lookbehinds with all states final (which means $F = Q$ in Definition 8.26) is given as input, the construction builds a $2DT^\omega$ with all states final.

Proof idea. The aforementioned construction from [HU67, Lemma 3] for removing lookarounds over finite words can directly be adapted in this setting. Indeed, since the machine only checks a property of the prefix, it can act "as if" the input was finite and use the marker $\vdash$. ◀

The difficulty for removing finite lookarounds unsurprisingly lies in the ability to check properties of the suffix starting in the current position. This ability is summarized in the notion of *finite lookaheads*.

Definition 9.8 (Two-way transducer with finite lookaheads)

A two-way deterministic transducer ($2DT^\omega$) with *finite lookaheads* consists of a modified two-way deterministic transducer $\mathscr{T} = (A, B, Q, q_0, F, \delta, \lambda)$ such that:

- the transition function δ has type $(Q \times A \uplus \{\vdash\} \times \mathsf{RegLang}(A)) \rightharpoonup Q$;
- the output function λ has type $(Q \times A \uplus \{\vdash\} \times \mathsf{RegLang}(A)) \rightharpoonup B^*$;
- $\mathsf{Dom}(\delta) = \mathsf{Dom}(\lambda)$ and this set is finite;
- for all $(q, a, R) \neq (q, a, R') \in \mathsf{Dom}(\delta)$, we have $R \cap R' = \varnothing$.

This machine can be seen as the dual of a $2DT^\omega$ with lookbehinds: it can only check properties of the suffix which starts in the current position (hence $\vdash$ is now necessary to detect the border when doing left moves). Its semantics is defined as that of a $2DT^\omega$ with finite lookarounds whose left component in $\mathsf{RegLang}(A)$ is always A^*, which means that it only checks a trivial property on the prefix.

Now, we claim that to remove finite lookarounds, it is in fact sufficient to remove finite lookaheads.

Theorem 9.9 (From finite lookarounds to finite lookaheads)

Given a $2DT^\omega$ with finite lookarounds, one can build a $2DT^\omega$ with finite lookaheads which computes the same function. Furthermore, if a $2DT^\omega$ with finite lookarounds with all states final is given, the construction builds a $2DT^\omega$ with finite lookaheads with all states final.

Proof idea. Adapt the proof of Theorem 9.7. ◀

9.2 Streaming string transducers of infinite words

Generalizations of streaming string transducers to infinite words were first introduced in [AFT12, Definition 3] for regular functions. In this section, we present the model of *streaming string transducers of infinite words* introduced in [CD22, Definition 3.4] to capture deterministic regular functions.

This model is interesting in itself since it provides a "streaming" implementation of deterministic regular functions. Furthermore, it will be used in Section 9.4 as a key tool to remove finite lookaheads for $2\mathrm{DT}^\omega$, and in Chapter 10 to deal with continuous rational functions.

9.2.1 Streaming string transducers of infinite words

Intuitively, a *streaming string transducer of infinite words* consists of a usual streaming string transducer which uses a distinguished register out to collect the output produced when reading an infinite word. We shall ensure syntactically that this output converges to either a finite or infinite word.

Definition 9.10 (Streaming string transducer of infinite words)

A *deterministic streaming string transducer of infinite words* (DSST^ω) $\mathscr{S} = (A, B, Q, q_0, F, \delta, \mathfrak{R}, \mathsf{out}, \iota, \lambda)$ consists of:

- ▶ an input alphabet A and an output alphabet B;
- ▶ a finite set of states Q with $q_0 \in Q$ initial and $F \subseteq Q$ final;
- ▶ a transition function $\delta : Q \times A \rightharpoonup Q$;
- ▶ a finite set of registers $\mathfrak{R}$ with a distinguished output register $\mathsf{out} \in \mathfrak{R}$;
- ▶ an initial function $\iota \colon \mathfrak{R} \to B^*$;
- ▶ an update function $\lambda : Q \times A \rightharpoonup \mathcal{S}^B_{\mathfrak{R}}$ such that for all $(q, a) \in \mathsf{Dom}(\lambda) = \mathsf{Dom}(\delta)$:
  - ▶ $\lambda(q, a)(\mathsf{out}) = \mathsf{out} \cdots$;
  - ▶ there is no other occurrence of out among the $\lambda(q, a)(\mathfrak{r})$ for $\mathfrak{r} \in \mathfrak{R}$.

The *extended transition function* δ^* and *extended output function* λ^* are defined as for finite words after Definition 4.14. For all $\mathfrak{r} \in \mathfrak{R}$ and $u \in A^*$, we define the substitution $[\![\cdot]\!]_u : \mathfrak{R} \to B^*$ which provides "the values of the registers after reading u" in the same fashion. By construction, if $u \in A^\omega$ then $[\![\mathsf{out}]\!]_{u[1:i]}$ is a prefix of $[\![\mathsf{out}]\!]_{u[1:i+1]}$ for all $i \geqslant 0$. The function $[\![\mathscr{S}]\!] : A^\omega \rightharpoonup B^\omega$ is defined as follows. Given $u \in A^\omega$, $[\![\mathscr{S}]\!](u)$ is defined if and only if $\delta^*(q_0, u[1{:}i])$ belongs to F infinitely often (Büchi conditions) and $|[\![\mathsf{out}]\!]_{u[1:i]}| \to \infty$. In this case, we let $[\![\mathscr{S}]\!](u) \coloneqq \bigvee_i [\![\mathsf{out}]\!]_{u[1:i]}$ (where the symbol $\bigvee$ is used to denote the unique $\alpha \in B^\omega$ such that $[\![\mathsf{out}]\!]_{u[1:i]}$ is a prefix of α for all $i \geqslant 0$).

Example 9.11 (Replacing factors)

The function replace can be computed by a DSST^ω which crosses each block 0^{n_i} while computing 1^{n_i} and 2^{n_i} in two registers. When it sees $a_i \in \{1, 2\}$, it sends the appropriate register in out.

Example 9.12 (Map copy reverse)

The function map-copy-reverse$^\omega$ can be computed by a DSST^ω. When reading a factor u_i, it writes u_i in out and $\widetilde{u_i}$ in another register. This register is sent into out when reading $\#$.

The notions of *copyless* and *K-bounded* DSST^ω are directly adapted to the context of infinite words from Definitions 4.26 and 4.33. Now, we give an analogue of Corollary 4.35 by showing that such machines capture deterministic regular functions. The next result originates from [CD22, Theorem 3.7].

Theorem 9.13 (Two-way transducers = copyless/bounded DSST$^\omega$)

Given $f : A^\omega \rightharpoonup B^\omega$, the following conditions are equivalent:

(1) f is computed by a 2DT$^\omega$ (i.e. f is deterministic regular);
(2) f is computed by a K-bounded DSST$^\omega$ for some $K \geqslant 0$;
(3) f is computed by a copyless DSST$^\omega$.

The conversions are effective. If either a copyless DSST$^\omega$, or a K-bounded DSST$^\omega$, or a 2DT$^\omega$ with all states final is given, the construction builds another machine with all states final.

Proof. Item (3) ⇒ Item (1) is shown in Section 9.2.3. Item (1) ⇒ Item (2) is shown in Section 9.2.4. Both conversions are obtained by easily adapting the techniques used over finite words. In Section 9.3, we show Item (2) ⇒ Item (3). This case is more complex than the similar proof over finite words. Finally, all conversions are effective and preserve the property of accepting states. ◀

9.2.2 Domains and final conditions

The goal of this section is to provide low-hanging consequences of Theorem 9.13. We first observe that the domains of deterministic regular functions are Büchi deterministic languages. We also show that such languages are preserved by deterministic regular functions under inverse images. As particular case of regular functions, deterministic regular functions also preserve ω-regular languages by inverse images (but none of these two results imply the other).

Proposition 9.14 (Relations with deterministic Büchi languages)

Let $f : A^\omega \rightharpoonup B^\omega$ be a deterministic regular functions, then:

(1) $\mathsf{Dom}(f)$ is (effectively) Büchi deterministic;
(2) if $L \subseteq B^\omega$ is Büchi deterministic, then $f^{-1}(L) \subseteq A^\omega$ is (effectively) Büchi deterministic;
(3) if $L \subseteq \mathsf{Dom}(f)$ is Büchi deterministic, then $f|_L$ is (effectively) deterministic regular.

Proof ideas. For Item (1), we start from a (copyless) DSST$^\omega$ which computes f, and we build a deterministic Büchi automaton which computes $\mathsf{Dom}(f)$. The idea is to keep track of whether the registers are empty or not (it is a bounded information which can be stored in the states), and reach a final state when the DSST$^\omega$ visits a final state and later adds a non-empty register in out.

For Item (2), we start from a 2DT$^\omega$ which computes f. We perform a (wreath) product construction with a deterministic finite automaton with Büchi conditions which recognizes L to build a 2DT$^\omega$ which computes f restricted to the language $f^{-1}(L) = \{u \in \mathsf{Dom}(f) \mid f(u) \in L\}$. By Item (1) the domain of this deterministic regular function is Büchi deterministic.

For Item (3), we start from a copyless DSST$^\omega$ which computes f. We perform a product construction of its underlying one-way automaton (with Büchi conditions) with a one-way automaton (with Büchi conditions) which recognizes L, to build a copyless DSST$^\omega$ which computes $f|_L$. ◀

More interestingly, we show that for DSST$^\omega$, the Büchi final conditions can be encoded within the fact that the output is infinite. This result relies on the ability of DSST$^\omega$ to "wait" an unbounded time before producing an output. Recall that it does not hold for 1DT$^\omega$ and sequential functions (see Claim 8.9).

Lemma 9.15 (Removing final states in DSST$^\omega$)

Given a copyless (resp. K-bounded) DSST$^\omega$, one can build a copyless (resp. K-bounded) DSST$^\omega$ which computes the same function and with all states final.

Proof idea. Let $\mathscr{S}$ be the original DSST$^\omega$. We build a new DSST$^\omega$ $\mathscr{S}'$ with all states final, which consists of $\mathscr{S}$ with all states final together with a new register out'. Whenever $\mathscr{S}$ would update out, then $\mathscr{S}'$ adds this value in out' instead. Furthermore, each time $\mathscr{S}$ reaches a final state, then $\mathscr{S}'$ empties out' and adds its value in out. It is clear that the output of $\mathscr{S}'$ is infinite if and only if $\mathscr{S}$ infinitely often visits an accepting state and produces an infinite output. ◀

As a consequence, final states are also useless in 2DT$^\omega$. In Sections 9.5 and 9.6, we shall freely assume that our 2DT$^\omega$ are *normalized* (this will avoid technicalities when dealing with final states). We define the notion of *n-run* as we did over finite words in Section 1.2.2.1.

Proposition-Definition 9.16 (Normalization of two-way transducers)

We say that a 2DT$^\omega$ $\mathscr{T} = (A, B, Q, q_0, F, \delta, \lambda)$ is *normalized* the following holds:

- $F = Q$ (all states are final);
- for all $q \in Q$ and $a \in A$, $\lambda(q, a) \in B \cup \{\varepsilon\}$ (at most one letter);
- for all $q \in Q$, $\lambda(q, \vdash) = \varepsilon$ (no output on the border).

Given a 2DT$^\omega$, one can build an equivalent normalized 2DT$^\omega$.

Proof. We first convert the 2DT$^\omega$ into a DSST$^\omega$ by Theorem 9.13. Then we build an equivalent DSST$^\omega$ with all states final by Lemma 9.15, and we further convert it into a 2DT$^\omega$ with all states final by Theorem 9.13. Finally, we shift all productions on $\vdash$ to the first letter of the input. ◀

9.2.3 From copyless streaming string transducers to two-way transducers

The goal of this section is to show Item (3) $\Rightarrow$ Item (1) in Theorem 9.13. Given a copyless DSST$^\omega$, we describe how to build an equivalent 2DT$^\omega$. Furthermore, if all states are final in the DSST$^\omega$, it will also be the case in the 2DT$^\omega$. The proof goes over the proof of Section 4.3.3.1 in the case of finite words.

Consider a DSST$^\omega$ $\mathscr{S} = (A, B, Q, q_0, F, \delta, \mathfrak{R}, \mathsf{out}, \iota, \lambda)$. Without losing generalities, we assume that the updates of the register out in $\mathscr{S}$ always have shape $\mathsf{out} \mapsto \mathsf{out}\,\mathsf{out}'$ for a specific register out' (this can be done by using out' as a buffer to store the values which should be added in out in position i, and then adding them into out in position $i{+}1$ by doing $\mathsf{out} \mapsto \mathsf{out}\,\mathsf{out}'$).

Thanks to Lemma 4.36, one can build a 2DT with lookarounds $\mathscr{T}$ with designated states $p_\mathfrak{r}$ and $r_\mathfrak{r}$ for $\mathfrak{r} \in \mathfrak{R}$, such that the following holds. For all $u \in A^*$, $1 \leqslant i \leqslant |u|$ and $\mathfrak{r} \in \mathfrak{R}$, the longest run of $\mathscr{T}$ labelled by $\vdash u \dashv$ which starts in configuration $(p_\mathfrak{r}, |\vdash u[1{:}i]|)$ and only moves on $\vdash u[1{:}i]$ has the following property: it outputs $[\![\mathfrak{r}]\!]_{u[1:i]}$ and it ends in configuration $(r_\mathfrak{r}, |\vdash u[1{:}i]|{+}1)$. Recall that the construction of Lemma 4.36 follows Algorithm 4.22 and only uses the lookarounds to determine the current state of $\mathscr{S}$, i.e. $q := \delta^*(q_0, u)$. Thus no finite lookaheads (= no informations about the future) are needed, and $\mathscr{T}$ can in fact be seen as a 2DT$^\omega$ with lookbehinds. In this setting, for all $u \in A^\omega$, $i \geqslant 1$ and $\mathfrak{r} \in \mathfrak{R}$, the longest run of $\mathscr{T}$ labelled by $\vdash u$ which starts in configuration $(p_\mathfrak{r}, |\vdash u[1{:}i]|)$ and moves on $\vdash u[1{:}i]$ has the following property: it outputs $[\![\mathfrak{r}]\!]_{u[1:i]}$ and it ends in configuration $(r_\mathfrak{r}, |\vdash u[1{:}i{+}1]|)$.

We are ready to describe a 2DT$^\omega$ with lookbehinds which simulates $\mathscr{S}$. It behaves as follows on input $u \in A^\omega$: for all $i \geqslant 1$, it simulates the 2DT$^\omega$ with lookbehinds $\mathscr{T}$ when starting in configuration $(r_{\mathsf{out}'}, |\vdash u[1{:}i]|)$ until it reaches $(p_{\mathsf{out}'}, |\vdash u[1{:}i{+}1]|$ (in the construction of Lemma 4.36, $p_{\mathsf{out}'}$ cannot be reached before coming back to position i, since out' is only used to update out). It is clear that this 2DT$^\omega$ produces an infinite output (resp. a finite output, resp. gets blocked) if $\mathscr{S}$ produces an infinite output (resp. a finite output, resp. gets blocked). If all the states of $\mathscr{S}$ were final, our construction is correct when setting all states final. If $\mathscr{S}$ has non-final states, we make our machine visit a final state only when it starts to simulate $\mathscr{T}$ in a position $i \geqslant 0$ such that $\delta^*(q_0, u[1{:}i]) \in F$. Finally, we remove the lookbehinds of the 2DT$^\omega$ thanks to Theorem 9.7.

9.2.4 From two-way transducers to bounded streaming string transducers

The goal of this section is to show Item (1) ⇒ Item (2) in Theorem 9.13. Given a 2DT$^\omega$, we show how to build an equivalent 1-bounded DSST$^\omega$. Furthermore, if all states are final in the 2DT$^\omega$, it will also be the case in the DSST$^\omega$. The proof is a variant of Section 4.2.4 which studies recursive marble transducers. Even if a similar construction is well-known over finite words, we describe it in detail since it plays the role of a warm-up for the proof of Section 9.4 which deals with finite lookaheads.

Let $\mathscr{T} = (A, B, Q, q_0, F, \delta, \lambda)$ be a 2DT$^\omega$ which computes a function $f : A^\omega \rightharpoonup B^\omega$. One can define the *extended transition function* δ^* and the *extended output function* λ^* as we did for 2DT, together with the function maxi-run which builds the longest run staying in a word (see Section 2.1.1). The notions of *transition morphism* and *transition monoid* are defined exactly as in Proposition-Definition 2.2. Furthermore, we extend δ^* and maxi-run to infinite words by defining $\delta^*(\overrightarrow{q}, u) := \overleftarrow{p}$ for $u \in A^\omega$ if the longest (finite) run leaves u "on the left" and $\delta^*(\overrightarrow{q}, u) := \omega$ if this run is infinite and visits arbitrary large positions (otherwise $\delta^*(\overrightarrow{q}, u)$ and maxi-run$(\overrightarrow{q}, u)$ are undefined). Note that $\delta^*(\overleftarrow{q}, u)$ for $u \in A^\omega$ would not make sense (there is no right border). The function λ^* is extended accordingly.

Now, let us describe a 1-bounded DSST$^\omega$ $\mathscr{S}$ which computes the function f.

9.2.4.1 Information stored by $\mathscr{S}$. After reading $u \in A^*$, $\mathscr{S}$ will store:

(1) informations about the right-to-right runs of $\mathscr{T}$ labelled by $\vdash u$:
   (a) for all $p \in Q$ such that $\delta^*(\overleftarrow{p}, \vdash u)$ has shape $\overleftarrow{q}$, the state q, stored in the states of $\mathscr{S}$;
   (b) for all $p \in Q$ such that $\delta^*(\overleftarrow{p}, \vdash u)$ has shape $\overleftarrow{q}$, $\lambda^*(\overleftarrow{p}, \vdash u)$ stored in a register right_p;

(2) informations about the beginning of the initial run labelled by $\vdash u$:
   (a) if $\delta^*(\overrightarrow{q_0}, \vdash u)$ has shape $\overrightarrow{q}$, the state q, stored within the states of $\mathscr{S}$;
   (b) if $\delta^*(\overrightarrow{q_0}, \vdash u)$ has shape $\overrightarrow{q}$, $\lambda^*(\overrightarrow{q_0}, \vdash u)$ stored in the register out.

9.2.4.2 Updating the right-to-right and initial runs. Assume that $\mathscr{S}$ has computed the elements of Items (1) and (2) for some $u \in A^*$. Let $a \in A$ and $p_0 \in Q$, we show how maxi-run$(\overleftarrow{p_0}, \vdash ua)$ can be described by recombining the informations about $\vdash u$. Claim 9.17 is roughly a reformulation of Claim 4.23 in the absence of recursive calls. The reader is invited to consult back Figure 4.24 if needed.

Claim 9.17 (Updating right-to-right runs)

$\delta^*(\overleftarrow{p_0}, \vdash ua) = \overrightarrow{q}$ if and only there exist $0 \leqslant n < |Q|$ and $q_1, p_1, \dots, q_n, p_n \in Q$ such that:

- $\delta(p_n, a) = (\triangleright, q)$ and for all $0 \leqslant i < n$, $\delta(p_i, a) = (\triangleleft, q_{i+1})$;
- for all $1 \leqslant i \leqslant n$, $\delta^*(\overleftarrow{q_i}, u) = \overrightarrow{p_i}$.

In this case, we have:

$$\lambda^*(\overleftarrow{p_0}, \vdash ua) = \lambda(p_0, a)\, \lambda^*(\overleftarrow{q_1}, \vdash u)\, \lambda(p_1, a) \cdots \lambda^*(\overleftarrow{q_n}, \vdash u)\, \lambda(p_n, a).$$

Thanks to Claim 9.17, $\mathscr{S}$ can compute the states $q_1, p_1, \dots, q_n, p_n, q \in Q$ whenever they exist. For the registers, we update $\mathsf{right}_q \mapsto \lambda(p_0, a)\ \mathsf{right}_{q_1}\ \lambda(p_1, a) \cdots \mathsf{right}_{q_n}\ \lambda(p_n, a)$.

The updates of Item (2) are done by a similar construction for maxi-run$(\overrightarrow{q_0}, \vdash u)$.

9.2.4.3 Correctness of the construction. We first observe that the register out is only used to update itself. Indeed, there is no need to use the output produced along the initial run when building right-to-right runs. Since it contains exactly the output produced along an initial run, we have $[\![\mathsf{out}]\!]_{v[1:i]} \to f(v)$

whenever $v \in \mathsf{Dom}(f)$. If $F = Q$ in $\mathscr{T}$, it is easy to see that when $v \notin \mathsf{Dom}(f)$, either $\mathscr{S}$ gets blocked, or it produces a finite output. Hence we can set all its states as final.

If $F \neq Q$, one needs to strengthen the information stored by $\mathscr{S}$. For all $p \in Q$ such that $\delta^*(\overleftarrow{p}, \vdash u)$ has shape $\overrightarrow{p}$, we make $\mathscr{S}$ keep track of whether $\mathsf{maxi\text{-}run}(\overleftarrow{p}, \vdash u)$ meets a final state or not. This information is then used to determine when the initial run $\mathsf{maxi\text{-}run}(\overrightarrow{q_0}, \vdash u)$ meets accepting states when doing right-to-right runs, and the accepting states of $\mathscr{S}$ are built accordingly.

9.2.4.4 1-boundedness of the streaming string transducer.

We finally justify that $\mathscr{S}$ is 1-bounded. Intuitively, sending two copies of a given register into another one would mean that the same piece of run is used twice to build another run (which is not possible).

Claim 9.18 (Copies are loops)

Let $u \in A^*$ and $0 \leqslant i \leqslant |u|$. Let s be the substitution applied by $\mathscr{S}$ when reading $u[i{+}1{:}|u|]$, after having read $u[1{:}i]$. For all $p \in Q$, if right_p occurs $k \geqslant 1$ times in $s(\mathsf{right}_q)$, then $\delta^*(\overleftarrow{q}, u)$ has shape $\overrightarrow{r}$ and $\mathsf{maxi\text{-}run}(\overleftarrow{q}, u)$ visits k times the configuration $(|\vdash u[1{:}i]|, p)$.

Proof idea. By induction on $|u|$. ◀

Claim 9.18 immediately gives a contradiction if $k \geqslant 2$, since $\mathsf{maxi\text{-}run}(\overleftarrow{q}, u)$ cannot visit twice the same configuration without looping. Similar results can be shown when dealing with out. Overall, $\mathscr{S}$ is 1-bounded (assuming that it is trim, i.e. that any of its states is accessible).

9.3 From bounded to copyless streaming string transducers

The goal of this section is to show Item (2) ⇒ Item (3) in Theorem 9.13. Given a K-bounded DSST^ω, we describe how to build an equivalent copyless DSST^ω. Furthermore, if all states are final in the first DSST^ω, it will also be the case in the second one. The construction is adapted from those over finite words [DJR18, DFG20][2]. However, it is not possible to directly re-use them since they add e.g. non-determinism or lookarounds to DSST, and we are precisely trying to avoid such features here.

The first idea for transforming a K-bounded DSST^ω into a copyless one is to maintain K copies of each register. However, this information cannot be updated during a computation: if $\mathfrak{r}$ is used to update both $\mathfrak{r}_1$ and $\mathfrak{r}_2$, one cannot build K copies of $\mathfrak{r}_1$ and K copies of $\mathfrak{r}_2$ in a copyless fashion. A solution is to maintain exacly the number of copies of $\mathfrak{r}$ that will be used in the output at some point in the future. However, this value cannot be determined before reading the whole infinite input. Therefore, we shall keep track of a tree which contains all the consistent non-deterministic guesses of these values. The second difficulty of the proof is to ensure in this setting that the output is infinite.

In Section 9.3.1, we first study some properties of the copies in a K-bounded DSST^ω. In Section 9.3.3 we describe how to transform a K-bounded DSST^ω into a copyless one by doing a tree construction, under the assumption that it can manipulate K-bounded substitutions. In Section 9.3.2, we explain how these manipulations can be done in a copyless fashion. From now on, let us fix a K-bounded DSST^ω $\mathscr{S} = (A, B, Q, q_0, F, \delta, \mathfrak{R}, \mathsf{out}, \iota, \lambda)$. We let $\mathfrak{T} := \mathfrak{R} \smallsetminus \{\mathsf{out}\}$. Given $u \in A^\omega$ and $i \geqslant 1$, we denote by λ_i^u the substitution applied when reading $u[i]$, i.e. $\lambda_i^u := \lambda(\delta^*(q_0, u[1{:}i{-}1], u[i]))$.

[2] It is however not exactly the same construction as in [AFT12].

9.3.1 Properties of copies

This section studies properties of register copies in $\mathscr{S}$. We first define formally in Definition 9.19 what is meant by "the number of copies of $\mathfrak{r}$ that will be used in the output at some point in the future".

Definition 9.19 (Copies)

Let $u \in A^\omega$ and $i \geqslant 0$ be such that λ^u_{i+1} is defined. Given $\mathfrak{r} \in \mathfrak{T}$, we let:

$$\mathsf{copies}^u_i(\mathfrak{r}) := \max\{|\lambda^u_{i+1} \circ \cdots \circ \lambda^u_j(\mathsf{out})|_\mathfrak{r} \mid j \geqslant i \text{ and } \lambda^u_j \text{ is defined}\}.$$

Observe that $\mathsf{copies}^u_i(\mathfrak{r}) \leqslant K$ since $\mathscr{S}$ is K-bounded. Now, we describe an inductive relation that $\mathsf{copies}^u_i(\mathfrak{r})$ verifies. Intuitively, Lemma 9.20 means that if $\mathsf{copies}^u_i(\mathfrak{r})$ copies of $\mathfrak{r}$ will be needed, then in the next transition these copies can be distributed in a consistent way among the registers.

Lemma 9.20 (Distributing copies)

Let $u \in A^\omega$ and $i \geqslant 0$ be such that λ^u_{i+1} is defined. Then for all $\mathfrak{r} \in \mathfrak{T}$:

$$\mathsf{copies}^u_i(\mathfrak{r}) = |\lambda^u_{i+1}(\mathsf{out})|_\mathfrak{r} + \sum_{\mathfrak{s} \in \mathfrak{T}} |\lambda^u_{i+1}(\mathfrak{s})|_\mathfrak{r} \times \mathsf{copies}^u_{i+1}(\mathfrak{s}).$$

The rest of Section 9.3.1 is devoted to showing Lemma 9.20. We first provide a way to count the copies obtained when composing two substitutions with Claim 9.21.

Claim 9.21 (Counting copies in compositions)

Let $s, s' \in \mathcal{S}^B_\mathfrak{R}$, then for all $\mathfrak{r}, \mathfrak{s} \in \mathfrak{R}$, $|s \circ s'(\mathfrak{r})|_\mathfrak{s} = \sum_{\mathfrak{t} \in \mathfrak{R}} |s(\mathfrak{t})|_\mathfrak{s} \times |s'(\mathfrak{r})|_\mathfrak{t}$.

From now on, we fix $u \in A^\omega$. We note that since out is always updated with out α, the number of copies of a given register which are used out can only grow when reading the input.

Claim 9.22 (More copies in the outputs)

Given $\mathfrak{r} \in \mathfrak{T}$ and $i \geqslant 0$ such that λ^u_{i+1} is defined, the function which maps $j \geqslant i$ such that λ^u_j is defined to $|\lambda^u_{i+1} \circ \cdots \circ \lambda^u_j(\mathsf{out})|_\mathfrak{r}$ is non-decreasing.

Proof. By Claim 9.21, $|\lambda^u_{i+1} \circ \cdots \circ \lambda^u_{j+1}(\mathsf{out})|_\mathfrak{r} \geqslant |\lambda^u_{i+1} \circ \cdots \circ \lambda^u_j(\mathsf{out})|_\mathfrak{r} \times |\lambda^u_{j+1}(\mathsf{out})|_\mathsf{out}$. ◀

By Claim 9.21, we have for all $j \geqslant i{+}1$ such that λ^u_j is defined and for all $\mathfrak{r} \in \mathfrak{T}$:

$$|\lambda^u_{i+1} \circ \cdots \circ \lambda^u_j(\mathsf{out})|_\mathfrak{r} = |\lambda^u_{i+1}(\mathsf{out})|_\mathfrak{r} \times 1 + \sum_{\mathfrak{s} \in \mathfrak{T}} |\lambda^u_{i+1}(\mathfrak{s})|_\mathfrak{r} \times |\lambda^u_{i+2} \circ \cdots \circ \lambda^u_j(\mathsf{out})|_\mathfrak{s}.$$

Now let $j_0 \geqslant i{+}1$ be such that $|\lambda^u_{i+1} \circ \cdots \circ \lambda^u_{j_0}(\mathsf{out})|_\mathfrak{r}$ is maximal (this value exists since copies^u_i is finite). Since $j \mapsto |\lambda^u_{i+1} \circ \cdots \circ \lambda^u_j(\mathsf{out})|_\mathfrak{r}$ is non-decreasing, we get for all $j \geqslant j_0$ (with λ^u_i defined):

$$\mathsf{copies}^u_i(\mathfrak{r}) = |\lambda^u_{i+1} \circ \cdots \circ \lambda^u_j(\mathsf{out})|_\mathfrak{r} = |\lambda^u_{i+1}(\mathsf{out})|_\mathfrak{r} + \sum_{\mathfrak{s} \in \mathfrak{T}} |\lambda^u_{i+1}(\mathfrak{s})|_\mathfrak{r} \times |\lambda^u_{i+2} \circ \cdots \circ \lambda^u_j(\mathsf{out})|_\mathfrak{s}.$$

Since the function $j \mapsto |\lambda^u_{i+2} \circ \cdots \circ \lambda^u_j(\mathsf{out})|_\mathfrak{s}$ is constant to copies^u_{i+1} for j large enough, we conclude.

Jump to contents

9.3.2 Toolbox: manipulating bounded substitutions

In Section 9.3.3, we shall describe a copyless DSST$^\omega$ which simulates the K-bounded DSST$^\omega$ $\mathscr{S}$. This machine will manipulate some substitutions applied by $\mathscr{S}$ along portions of its input. In this section, we explain as a first step how a copyless DSST$^\omega$ can manipulate K-bounded substitutions.

9.3.2.1 Hoarding bounded substitutions. We describe how a copyless DSST$^\omega$ can store K-bounded substitutions (the ideas are mainly based on those of [DFG20, Section E.2]). Let $s \in \mathcal{S}^B_{\mathfrak{T}}$ be a K-bounded substitution, then for all $\mathfrak{r} \in \mathfrak{T}$ there exists $n \leqslant K|\mathfrak{T}|$ such that $s(\mathfrak{r}) = \alpha_0 \mathfrak{r}_1 \alpha_1 \cdots \mathfrak{r}_n \alpha_n$ with $\alpha_i \in B^*$ and $\mathfrak{r}_i \in \mathfrak{T}$. We mainly have two informations in this expression:

(1) the sequence $\mathfrak{r}_1, \cdots, \mathfrak{r}_n$ with $n \leqslant K|\mathfrak{T}|$ which describes where the registers are used;
(2) the sequence $\alpha_0, \ldots, \alpha_n$ of (possibly unbounded) words from B^*.

Definition 9.23 (Hoarding bounded substitutions)

Let $k \geqslant 0$. We say that a DSST$^\omega$ $\mathscr{S}'$ *hoards* k copies of $s(\mathfrak{r})$ if it stores:

(1) the bounded sequence $\mathfrak{r}_1, \cdots, \mathfrak{r}_n$ in its bounded memory;
(2) for all $0 \leqslant j \leqslant n$, k copies of the word $\alpha_j \in B^*$ in k distinct registers.

In the rest of Section 9.3, we use the term hoard with the formal meaning of Definition 9.23.

9.3.2.2 Composing bounded substitutions. Thanks to the representation of substitutions described in Section 9.3.2.1, we show how to simulate the composition of two K-bounded substitutions, whose composition is itself K-bounded, without making copies.

Claim 9.24 (Copyless composition of K-bounded substitutions)

Let $s, s' \in \mathcal{S}^B_{\mathfrak{T}}$ be K-bounded and such that $s \circ s'$ is so. Let $g, g' \colon \mathfrak{T} \to [0{:}K]$ be such that $g(\mathfrak{r}) = \sum_{\mathfrak{s} \in \mathfrak{T}} |s'(\mathfrak{s})|_{\mathfrak{r}} \times g'(\mathfrak{s})$ for all $\mathfrak{r} \in \mathfrak{T}$. Assume that some DSST$^\omega$ $\mathscr{S}'$ hoards:

- $g(\mathfrak{r})$ copies of $s(\mathfrak{r})$ for all $\mathfrak{r} \in \mathfrak{T}$;
- $g'(\mathfrak{s})$ copies of $s'(\mathfrak{s})$ for all $\mathfrak{s} \in \mathfrak{T}$.

Then one can describe a copyless update of $\mathscr{S}'$ so that the following holds after this update: $\mathscr{S}'$ hoards $g'(\mathfrak{s})$ copies of $s \circ s'(\mathfrak{s})$ for $\mathfrak{s} \in \mathfrak{T}$.

Proof. In order to hoard $g'(\mathfrak{s})$ copies of $s \circ s'(\mathfrak{s})$, we exactly need to use $|s'(\mathfrak{s})|_{\mathfrak{r}} \times g'(\mathfrak{s})$ copies of each hoarded element $s(\mathfrak{r})$. The result follows by summing over all $\mathfrak{s} \in \mathfrak{T}$. ◀

Thanks to Claim 9.24, the machine that we build will be able to compose the substitutions that it hoards, assuming that it has a correct number of copies.

9.3.3 Construction of the copyless streaming string transducer

Now our goal is to build a copyless DSST$^\omega$ $\mathscr{S}'$ which, when given $u \in A^\omega$ as input, hoards $\mathsf{copies}^u_i(\mathfrak{r})$ copies of $[\![\mathfrak{r}]\!]_{u[1:i]}$ after reading $u[1{:}i]$. However, one needs informations about $u[i{+}1{:}]$ in order to determine $\mathsf{copies}^u_i(\mathfrak{r})$ (this is typically where we would need an ω-lookaround). Thus our copyless DSST$^\omega$ cannot exactly compute $\mathsf{copies}^u_i(\mathfrak{r})$. It will instead memorize a finite forest of substitutions.

Formally, we introduce the notion of *decomposition* of a substitution s. We recall that the *depth* of a node in a tree is defined inductively by starting from *root* which has depth 1.

Definition 9.26 (Decomposition)

Given a K-bounded substitution $s \in \mathcal{S}^B_{\mathfrak{T}}$, a *decomposition* of s of *depth* $n \geqslant 1$ consists of:

- a sequence $s_1, \ldots, s_n$ of K-bounded substitutions such that $s = s_1 \circ \cdots \circ s_n$;
- a finite set of trees of depth n, whose nodes are labelled by functions $\mathfrak{T} \to [0{:}K]$ such that:
  - all leaves of all trees have depth n and distinct labels;
  - for all $1 \leqslant j \leqslant n{-}1$, there exist a tree and a node of depth j in this tree which has at least two children (i.e. there is no linear branching simultaneously in all trees);
  - for all $2 \leqslant j \leqslant n$, if h is the label of a node of depth j in a tree and g labels its parent, then for all $\mathfrak{r} \in \mathfrak{T}$:

$$g(\mathfrak{r}) = \sum_{\mathfrak{s} \in \mathfrak{T}} |s_j(\mathfrak{s})|_{\mathfrak{r}} \times h(\mathfrak{s}). \tag{9.26}$$

Observe that the number of decompositions is finite when forgetting the $s_1, \ldots, s_n$ part (which contains an unbounded information). Indeed, the total number of leaves is bounded (since all leaves have to be distinct). Furthermore, the depth n of the trees has to be bounded, since otherwise by the pigeonhole principle one could find $1 \leqslant j \leqslant n{-}1$ such that all nodes of depth j have a single child.

9.3.3.1 Information stored by the copyless DSST$^\omega$. Let $u \in A^\omega$ and $i \geqslant 0$ be such that λ^i_u is defined. We want $\mathscr{S}'$ to preserve the following invariants after reading $u[1{:}i]$:

1. it has produced $[\![\mathsf{out}]\!]_{u[1:i]}$ in its output register;
2. it keeps track of a decomposition of the K-bounded substitution $\iota \circ \lambda^u_1 \circ \cdots \lambda^u_i|_{\mathfrak{T}}$ such that one of the trees has a leaf whose label is copies^u_i. More precisely if $s_1, \ldots, s_n$ are the substitutions:
   (a) $\mathscr{S}'$ stores the (bounded) trees in its states;
   (b) for all node of a tree of depth $1 \leqslant j \leqslant n$ whose label is $g\colon \mathfrak{T} \to [0{:}K]$, $\mathscr{S}'$ hoards $g(\mathfrak{r})$ copies of $s_j(\mathfrak{r})$ for all $\mathfrak{r} \in \mathfrak{T}$.

The rest of Section 9.3.3 is devoted to explaining, how $\mathscr{S}'$ performs its initialization and updates. We rely on the ability to compose the hoarded substitutions, as explained in Claim 9.24.

9.3.3.2 Initialization of the decomposition. For $i = 0$, $\mathscr{S}'$ stores a decomposition of height 1 which consists of the substitution $s_1 := \iota|_{\mathfrak{T}}$ and a set of $(K+1)^{|\mathfrak{T}|}$ trees of depth 1 whose root labels describe all the possible functions $\mathfrak{T} \to [0{:}K]$ (one of them is copies^u_0). Furthermore, $\mathscr{T}'$ outputs $\iota(\mathsf{out})$.

9.3.3.3 Updates of the decomposition. Assume that λ^u_{i+1} is defined (otherwise we make $\mathscr{S}'$ get blocked), and that $\mathscr{S}'$ stores a decomposition of $\iota \circ \lambda^u_1 \circ \cdots \lambda^u_i$ of depth n, whose substitutions are $s_1, \ldots, s_n$, which verifies Invariant (2). Assume that Invariant (1) also holds. The update of $\mathscr{S}'$ in position $i{+}1$ is performed in two steps. The first one is described in the current Section 9.3.3.3 and consists in adding the substitution λ^u_{i+1} to the decomposition. The second step is described in Section 9.3.3.4 and consists in reducing the depth of the substitution if some nodes with a single child are met.

Recall that $\lambda^u_{i+1}(\mathsf{out})$ has shape $\mathsf{out}\,\alpha$ for some $\alpha \in (B \uplus \mathfrak{T})^*$. We want $\mathscr{S}'$ to output $s_1 \circ \cdots \circ s_n(\alpha)$. For this purpose, it needs to compute the value $s_1 \circ \cdots \circ s_n(\mathfrak{r})$ for all $\mathfrak{r} \in \mathfrak{T}$ which occurs in α. We thus define by decreasing induction the functions $\mathsf{used}_j\colon \mathfrak{T} \to [0{:}K]$ for $1 \leqslant j \leqslant n$, which describe how many copies of the $s_j(\mathfrak{r})$ are necessary to compute $s_1 \circ \cdots \circ s_n(\alpha)$, following Claim 9.24:

- $\mathsf{used}_n(\mathfrak{r}) := |\alpha|_{\mathfrak{r}}$ for all $\mathfrak{r} \in \mathfrak{T}$;
- $\mathsf{used}_j(\mathfrak{r}) := \sum_{\mathfrak{s} \in \mathfrak{T}} |s_{j+1}(\mathfrak{s})|_{\mathfrak{r}} \times \mathsf{used}_{j+1}(\mathfrak{s})$ for all $\mathfrak{r} \in \mathfrak{T}$ and $1 \leqslant j \leqslant n{-}1$.

Intuitively, the $\mathsf{used}_j(\mathfrak{r})$ represent the number of hoarded copies of $s_j(\mathfrak{r})$ which will be "consumed" to output $s_1 \circ \cdots \circ s_n(\alpha)$. Therefore, we want to subtract these values to the labels of the nodes, since the latter describe the number of copies of the $s_j(\mathfrak{r})$ which are stored by $\mathscr{S}'$.

Claim 9.28 (Consuming values)

Assume that h labels a node of depth $2 \leqslant j \leqslant n$ in some tree of the decomposition and that g labels its parent. We define $\overline{h} := h - \mathsf{used}_j$ and $\overline{g} := g - \mathsf{used}_{j-1}$. If $\overline{h} \geqslant 0$, then $\overline{g} \geqslant 0$ and:

$$\overline{g}(\mathfrak{r}) = \sum_{\mathfrak{s} \in \mathfrak{T}} |s_j(\mathfrak{s})|_{\mathfrak{r}} \times \overline{h}(\mathfrak{s}). \tag{9.28}$$

Proof. Recall that $g(\mathfrak{r}) = \sum_{\mathfrak{s} \in \mathfrak{T}} |s_j(\mathfrak{s})|_{\mathfrak{r}} \times h(\mathfrak{s})$, therefore we get by definition of used:

$$g(\mathfrak{r}) - \mathsf{used}_{j-1}(\mathfrak{r}) = \sum_{\mathfrak{s} \in \mathfrak{T}} |s_j(\mathfrak{s})|_{\mathfrak{r}} \times (h(\mathfrak{s}) - \mathsf{used}_j(\mathfrak{s})). \quad \blacktriangleleft$$

We are ready to update the decomposition and the substitutions hoarded, and to produce the output:

(1) We first replace each label g in the decomposition by $\overline{g}$ from Claim 9.27. This operation may create negative labels, but since one leaf is labelled by copies_i^u, we get $\overline{\mathsf{copies}_i^u} \geqslant 0$ by Lemma 9.20;
(2) Hence by Claim 9.27, there is a root-to-leaf branch in a tree whose labels are nonnegative. We choose such a branch and consume $\mathsf{used}_j(\mathfrak{r})$ copies of the hoarded $s_j(\mathfrak{r})$ which correspond to the nodes along this branch. We then output $s_1 \circ \cdots \circ s_n(\alpha)$ thanks to Claim 9.24;
(3) Then we add a layer $n+1$ to the decomposition. We define $s_{n+1} := \lambda_{i+1}^u\big|_{\mathfrak{T}}$ (this substitution is K-bounded since $\mathscr{T}$ is so). For each leaf of a tree which is labelled by $\overline{g}$, we create several children labelled by the possible $h : \mathfrak{T} \to [0{:}K]$ such that for all $\mathfrak{r} \in \mathfrak{T}$ we have:

$$\overline{g}(\mathfrak{r}) = \sum_{\mathfrak{s} \in \mathfrak{T}} |s_{n+1}(\mathfrak{s})|_{\mathfrak{r}} \times h(\mathfrak{s}).$$

For all $\mathfrak{r} \in \mathfrak{T}$ and all created leaf labelled by h, $\mathscr{S}'$ hoards $h(\mathfrak{r})$ copies of $s_{n+1}(\mathfrak{r})$ (which is a bounded information). Note that two created leaves cannot have the same label (otherwise it would be the case for their parents, which is impossible thanks to the invariants). Finally by Lemma 9.20 the node labelled by $\overline{\mathsf{copies}_i^u}$ necessarily has a leaf labelled by copies_{i+1}^u;
(4) Now it remains to deal with the fact that some nodes may have negative labels, and some leaves may have depth $\ell < n+1$. We thus remove all the nodes labelled by functions which take negative values, together with their descendants and the according hoarded copies of substitutions. Finally, we trim the resulting forest by removing all nodes which are not ancestors of some leaf of depth $n+1$ (i.e. a leaf which has just been created).

9.3.3.4 Merging operation. It remains to perform a last operation if there exists $1 \leqslant j \leqslant n$ such that all nodes of depth j have a single child. In this case, we want to "merge" s_j and s_{j+1}, i.e. to replace the sequence $s_1, \dots, s_j, s_{j+1}, \dots, s_{n+1}$ by $s_1, \dots, s_j \circ s_{j+1}, \dots, s_{n+1}$.

One has to guarantee that $s_j \circ s_{j+1}$ is K-bounded. This property follows by observing that we must have $s_j = \lambda_\ell^u \circ \cdots \circ \lambda_{\ell'}^u\big|_{\mathfrak{T}}$ and $s_{j+1} = \lambda_{\ell'+1}^u \circ \cdots \circ \lambda_{\ell''}^u\big|_{\mathfrak{T}}$ for some $\ell < \ell' < \ell''$.

Before explaining how the trees are updated, let us show Claim 9.29.

Claim 9.29 (Composition preserves the structure)

Assume that h labels a node of depth $j+1$ in a tree of the decomposition and let g be the label of its grandparent. Then for all $\mathfrak{r} \in \mathfrak{T}$:

$$g(\mathfrak{r}) = \sum_{\mathfrak{s} \in \mathfrak{T}} |s_j \circ s_{j+1}(\mathfrak{s})|_{\mathfrak{r}} \times h(\mathfrak{s}).$$

Proof. By Claim 9.21, we have:

$$\sum_{\mathfrak{s} \in \mathfrak{T}} |s_j \circ s_{j+1}(\mathfrak{s})|_{\mathfrak{r}} \times h(\mathfrak{s}) = \sum_{\mathfrak{s} \in \mathfrak{T}} \sum_{\mathfrak{t} \in \mathfrak{T}} |s_j(\mathfrak{t})|_{\mathfrak{r}} \times |s_{j+1}(\mathfrak{s})|_{\mathfrak{t}} \times h(\mathfrak{s})$$

The result follows by permuting the sums and using the hypothesis between depths j and $j+1$. ◀

The merging operation is done as follows. We transform the trees by merging each node of depth j with its single child of depth $j+1$ labelled by some g, and labelling the resulting node by g. For this node, $\mathscr{S}'$ has to hoard $g(\mathfrak{r})$ copies of $s_j \circ s_{j+1}(\mathfrak{r})$ for all $\mathfrak{r} \in \mathfrak{R}$, which is possible thanks to the invariants and Claim 9.24. The properties of the labels in decompositions still hold because of Claim 9.29.

9.3.4 Correctness of the construction

We finally justify that the previous construction is correct.

9.3.4.1 Final states and correctness. Given $u \in A^\omega$, it is clear that $\mathscr{S}'$ produces an infinite output (resp. a finite output, resp. gets blocked) if $\mathscr{S}$ produces an infinite output (resp. a finite output, resp. gets blocked). If all the states of $\mathscr{S}$ were final, our construction is correct when setting all states final in $\mathscr{S}'$. If $\mathscr{S}$ has non-final states, we make $\mathscr{S}'$ visit a final state in position $i \geqslant 0$ only when $\delta^*(q_0, u[1{:}i]) \in F$.

9.3.4.2 Origin semantics. In Chapter 4, we skipped the proof of Lemma 4.40 in order not to duplicate the proof of the current section. Indeed, it is easy to see that it can directly be transferred to finite words. In this setting, we also observe that origin semantics is preserved: given $u \in A^\omega$ and $i \geqslant 0$ such that λ_i^u is defined, if a letter $b \in B$ is created by λ_i^u, then it is also created in position i by $\mathscr{S}'$.

9.4 Removing finite lookaheads via streaming string transducers

The goal of this section is to show Theorem 9.30, which is the key technical ingredient for obtaining the finite lookarounds removal for 2DT$^\omega$ (Theorem 9.4). The proof is a simplification of the original proof presented in [CDFW23, Section C] (the latter builds tree structures, we do not).

Theorem 9.30 (From finite lookaheads to bounded DSST$^\omega$)

Given a 2DT$^\omega$ with finite lookaheads, one can effectively build a 1-bounded DSST$^\omega$ which computes the same function.

Proof sketch. The main idea is to leverage the proof of Section 9.2.4, which converts a $2DT^\omega$ into a 1-bounded $DSST^\omega$, in order to take the finite lookarounds into account. The main difficulty is that the behavior of a $2DT^\omega$ with finite lookarounds on a prefix of the input also depends on the future, which has not been read to far. However, the number of possible behaviors is bounded, since we only look for a regular property of the suffix. Thus we extend the construction by taking the possible choices induced by finite lookaheads into account. As usual over infinite words, an additional difficulty is to ensure that we produce an infinite output.

In Section 9.4.1, we first introduce the notions of extended transition function and extended output function for $2DT^\omega$ with finite lookaheads (now these functions have to take the future into account). Then, we describe in Section 9.4.2 a $DSST^\omega$ which builds an abstraction of initial and right-to-right runs of a $2DT^\omega$ with finite lookaheads while processing its input. Finally, we justify in Section 9.4.3 that the $DSST^\omega$ previously built is correct and 1-bounded. ◀

From now on, we fix a $2DT^\omega$ with finite lookaheads $\mathscr{T} = (A, B, Q, q_0, F, \delta, \lambda)$ which computes a function $f : A^\omega \rightharpoonup B^\omega$. Without loss of generalities, we assume that $\mathscr{T}$ verifies the following structural properties (which are used to simplify the construction):

(1) given a state $q \in Q$, all transitions leaving q go in the same direction, and left moves only use a trivial lookahead A^*. Formally, for any $q \in Q$, one of the following cases occur
- ▶ either for all a, R such that $(q, a, R) \in \mathsf{Dom}(\delta)$, we have $\delta(q, a, R) = (\_, \triangleright)$;
- ▶ or for all a, R such that $(q, a, R) \in \mathsf{Dom}(\delta)$, we have $\delta(q, a, R) = (\_, \triangleleft)$ and $R = A^*$.

It is always possible to get this property by adding several transitions to $\mathscr{T}$.

(2) given $q \in Q$ and $a \in A$ such that $(q, a, R), (q, a, R') \in \mathsf{Dom}(\delta)$ with $R \neq R'$, then $R \perp R'$, where this symbol means that for all $u \in R$ and $u' \in R'$, we have $u' \not\sqsubseteq u$ and $u \not\sqsubseteq u'$ (hence there is always a single witness). We obtain this property as follows. Let $R_1, \dots, R_n$ be the distinct languages such that (q, a, R) is defined if and only if $R = R_i$ for some $1 \leqslant i \leqslant n$. We replace each R_i in the transition by $R_i \smallsetminus \bigcup_{j \neq i} R_j A^*$. Due to the semantics which considers the shortest witness, such a modification does not affect the behavior of $\mathscr{T}$.

(3) for all $(q, a, R) \in \mathsf{Dom}(\delta)$, R is *suffix closed*, i.e. $RA^* = R$. We obtain this property by replacing each such R by RA^*. Observe that it does not affect the behavior of $\mathscr{T}$. Furthermore, it does not affect the property of Item (2). Indeed, assume that $u \sqsubseteq u'$ with $u \in RA^*$ and $u' \in R'A^*$, then $u = vw$ and $u' = v'w'$ for some $v \in R$ and $v' \in R'$. Thus either $v \sqsubseteq v'$ or $v' \sqsubseteq v$.

9.4.1 Lookahead informations

Let us first define the notions of *extended transition function* and *extended output function* for the $2DT^\omega$ with finite lookaheads $\mathscr{T}$. Since these lookaheads depend on the future, it no longer makes sense to define $\delta^*(q, u)$ for $u \in A^*$. The solution is to add a "future" component to the extended transition function: given $u \in A^+$ and $v \in A^\omega$, we let $\delta^*(\overrightarrow{p}, u, v) = \overrightarrow{q}$ whenever the run labelled by uv which starts in $(1, p)$ reaches position $|u|+1$ for the first time in state q. The output produced along this run is denoted by $\lambda^*(\overrightarrow{p}, u, v)$. We define $\delta^*(\overleftarrow{p}, u, v)$ and $\lambda^*(\overleftarrow{p}, u, v)$ in the same fashion.

We observe that this definition is consistent with the elementary transitions.

Claim 9.31 (Extended transitions on letters)

If $\delta(p, a, R) = (\triangleright, q)$, then $\delta^*(\overrightarrow{p}, a, v) = \delta^*(\overleftarrow{p}, a, v) = \overrightarrow{q}$ if and only $v \in RA^\omega$.

Let $C \subseteq \mathsf{RegLang}(A)$ be the closure of $\{R \mid (q, a, R) \in \mathsf{Dom}(\delta), q \in Q, a \in A\}$ under taking residuals and intersections. Intuitively, this set describes all combinations of lookahead conditions, possibly after reading some letters. The set $\mathsf{Look} \subseteq \mathsf{RegLang}(A)$ is defined as $(C \cup \{A^*\}) \smallsetminus \varnothing$ (we remove the empty language since it accounts for contradictory combinations of conditions).

Claim 9.32 (Future languages)

The set Look is finite. For all $L \in \mathsf{Look}$, L is suffix closed (i.e. $LA^* = L$).

Proof. Let $R_1, \dots, R_n \coloneqq \{R \mid (q, a, R) \in \mathsf{Dom}(\delta), q \in Q, a \in A\}$. Let $M_1, \dots, M_n$ be finite monoids which recognize respectively $R_1, \dots, R_n$. The monoid $M_1 \times \cdots \times M_n$ recognizes any language of Look, hence this set is finite. Furthermore, we have assumed in Item (3) that R_i was suffix closed, and this property is preserved under taking intersections and residuals. ◀

9.4.2 Construction of the streaming string transducer

Now we are ready to build a DSST^ω $\mathscr{S}$ which simulates the runs of $\mathscr{T}$, as we did in Section 9.2.4 without finite lookaheads. The main difficulty to perform such a simulation is that the DSST^ω has *a priori* no access to the information of which finite lookahead is admissible. Therefore, it will follow several possible runs in parallel and verify whether the assumed choices for finite lookaheads are indeed valid.

9.4.2.1 Information stored by $\mathscr{S}$. After reading $u \in A^*$, the machine $\mathscr{S}$ will keep track of:

(1) a function $\delta\text{-}\mathsf{right}_u : Q \times \mathsf{Look} \rightharpoonup Q$ and a finite set of registers $\mathsf{right}_{p,L}$ for $p, L \in Q \times \mathsf{Look}$. These elements will describe the right-to-right runs of $\mathscr{T}$ labelled by $\vdash u$;
(2) a function $\delta\text{-}\mathsf{initial}_u : \mathsf{Look} \times \mathsf{Look} \rightharpoonup Q$ and a finite set of registers frozen_F and $\mathsf{next}_{F,L}$ for $F, L \in \mathsf{Look}$. These elements will describe the initial runs of $\mathscr{T}$ labelled by $\vdash u$. In more detail, F and L describe the (regular) conditions[3] which have to be verified in the future for the state of the initial run to be $\delta\text{-}\mathsf{initial}_u(F, L)$ when the current position is reached for the first time.

The main idea is that $\delta\text{-}\mathsf{right}_u$ and $\delta\text{-}\mathsf{initial}_u$ are finite abstractions of δ^*, where the word $v \in A^\omega$ is replaced by a language $L \in \mathsf{Look}$. Formally, we want $\mathscr{S}$ to maintain the following invariants:

(1) (a) if $\delta\text{-}\mathsf{right}_u(p, L) = q$, then for all $v \in LA^\omega$, we have $\delta^*(\overleftarrow{p}, \vdash u, v) = \overrightarrow{q}$ and $\lambda^*(\overleftarrow{p}, \vdash u, v) = [\![\mathsf{right}_{p,L}]\!]_u$ (where $[\![\cdot]\!]$ denotes the values of registers in $\mathscr{S}$);
(b) if $v \in A^\omega$ is such that $\delta^*(\overleftarrow{p}, \vdash u, v) = \overrightarrow{q}$, then there exists $L \in \mathsf{Look}$ such that $v \in LA^\omega$ and $\delta\text{-}\mathsf{right}_u(p, L)$ is defined;
(c) if $\delta\text{-}\mathsf{right}_u(p, L_1)$ and $\delta\text{-}\mathsf{right}_u(p, L_2)$ are defined, then either $L_1 \perp L_2$ or $L_1 = L_2$.

(2) (a) if $\delta\text{-}\mathsf{initial}_u(F, L) = q$, then $F \cap L \neq \varnothing$ and for all $v \in (F \cap L)A^\omega$, we have $\delta^*(\overrightarrow{q_0}, \vdash u, v) = \overrightarrow{q}$ and $\lambda^*(\overrightarrow{q_0}, \vdash u, v) = [\![\mathsf{out}]\!]_u [\![\mathsf{frozen}_F]\!]_u [\![\mathsf{next}_{F,L}]\!]_u$;
(b) if $v \in A^\omega$ is such that $\delta^*(\overrightarrow{q_0}, \vdash u, v) = \overrightarrow{q}$, then there exist $L, T \in \mathsf{Look}$ such that $v \in (L \cap T)A^\omega$ and $\delta\text{-}\mathsf{initial}_u(L, T)$ is defined;
(c) if $\delta\text{-}\mathsf{initial}_u(F_1, L_1)$ and $\delta\text{-}\mathsf{initial}_u(F_2, L_2)$ are defined then either $F_1 \perp F_2$, or $F_1 = F_2$ and ($L_1 \perp L_2$ or $L_1 = L_2$).

9.4.2.2 Updating the right-to-right runs. Assume that $\mathscr{S}$ has computed $\delta\text{-}\mathsf{right}_u$ and $\delta\text{-}\mathsf{initial}_u$ after reading $u \in A^*$, so that Invariants (1) and (2) hold. Given $a \in A$, we explain how $\mathscr{S}$ builds $\delta\text{-}\mathsf{right}_{ua}$. Given $p_0 \in Q$ and $L \in \mathsf{Look}$, the value $\delta\text{-}\mathsf{right}_{ua}(p_0, L)$ is defined if and only if there exist $0 \leqslant n < |Q|$, $q_1, p_1, \dots, q_n, p_n, q \in Q$, $L_1, \dots, L_n, R \in \mathsf{Look}$ such that:

[3] The reader may ask why having a single component (either F or L) would not be sufficient. The reason is that F will contain "frozen" conditions, while the new conditions when moving forward on the input will be added to L. If the guess was correct, then F is finally verified at some point (by definition of finite lookaheads). At this point the machine is certain that frozen_F is a portion of the output and therefore it can produce it. This way, we shall guarantee that the output is infinite.

- $\delta(p_n, a, R) = (\rhd, q)$ and for all $0 \leqslant i < n$, $\delta(p_i, a, A^*) = (\lhd, q_{i+1})$;
- for all $1 \leqslant i \leqslant n$, $\delta\text{-right}_u(q_i, L_i) = p_i$;
- $L = R \cap \bigcap_{i=1}^{n}(a^{-1}L_i)$.

The intuition behind these conditions follows from Section 9.2.4.2 (see also Figure 4.24). The main difference is that we also update the language L, which corresponds to the lookahead conditions that still have to be checked: we have to see a word in R and words in $a^{-1}L_i$ for a run to be valid.

Claim 9.33 (Uniqueness in the construction)

Given $p_0 \in Q, L \in \mathsf{Look}$, if such $q_1, p_1, \dots, q_n, p_n, q$ and $L_1, \dots, L_n, R$ exist, they are unique.

Proof. Assume that one can find other sequences $q'_1, p'_1, \dots, q'_{n'}, p'_{n'}, q$ and $L'_1, \dots, L'_{n'}, R'$ such that the result holds. First observe that if $n = 0$ then $n' = 0$ (because all transitions leaving q must go in the same direction), and then we must have $R = R'$, therefore $q = q'$. Assume that $n, n' > 0$ and let $j \geqslant 1$ be the smallest index such that $(q_j, p_j, L_j) \neq (q'_j, p'_j, L'_j)$. Since left moves use A^* as a lookahead, we must have $q_j = q'_j$. Now if $L_j \perp L'_j$, then $a^{-1}L_j \cap a^{-1}L'_j = \varnothing$, hence they would not give the same $L \neq \varnothing$, thus $L_j = L'_j$ and $q_j = q'_j$. Finally $n = n'$ and $q = q'$. ◀

Thanks to Claim 9.33, one can set $\delta\text{-right}_{ua}(p, L) := q$ in a well-defined fashion. Furthermore, we update the registers by $\mathsf{right}_{p_0,L} \mapsto \lambda(p_0, a, A^*)\mathsf{right}_{q_1,L_1}\lambda(p_1, a, A^*) \cdots \mathsf{right}_{q_n,L_n}\lambda(p_n, a, R)$.

Claim 9.34 (Invariant preservation)

After this operation, Invariant (1) holds for $\delta\text{-right}_{ua}$.

Proof idea. The ideas are similar to that of Claims 4.23 and 9.17. Observe that having $n < |Q|$ is sufficient, since otherwise one would have $p_i = p_j$ for some $0 \leqslant i < j < n$, and such a run never occurs with some suffix $v \in A^\omega$. Let us justify that if $\delta\text{-right}_{ua}(p, L_1)$ and $\delta\text{-right}_{ua}(p, L_2)$ are defined, then $L_1 \perp L_2$. Assume the converse and let $w \in L_1$ which has a prefix in L_2. One obtains a contradiction by considering the languages used at the first time the sequences defining $\delta\text{-right}_u(p, L_1)$ and $\delta\text{-right}_u(p, L_2)$ differ, as we did in Claim 9.33. ◀

9.4.2.3 Updating the initial runs. Given $a \in A$, we explain how $\mathscr{S}$ builds $\delta\text{-initial}_{ua}$. The construction is close to that of Section 9.4.2.2, but we have to deal with the two components of $\delta\text{-initial}_{ua}$. Given $F, L \in \mathsf{Look}$ with $F \cap L \neq \varnothing$, the value $\delta\text{-initial}_{ua}(F, L)$ is defined if and only if there exist $0 \leqslant n < |Q|$, $p_0, q_1, p_1, \dots, q_n, p_n, q \in Q$ and $F', L', L_1, \dots, L_n, R \in \mathsf{Look}$ such that:

- $\delta\text{-initial}_u(F', L') = p_0$;
- $\delta(p_n, a, R) = (\rhd, q)$ and for all $0 \leqslant i < n$, $\delta(p_i, a, A^*) = (\lhd, q_{i+1})$;
- for all $1 \leqslant i \leqslant n$, $\delta\text{-right}_u(q_i, L_i) = p_i$;
- $L = R \cap L' \cap \bigcap_{i=1}^{n}(a^{-1}L_i)$ and $F = a^{-1}F'$.

The intuition behind this construction is depicted in Figure 9.35. Observe that the language F roughly reproduces the conditions of F', while all the new conditions are added in L. This way, we shall ensure that the unchanged F can be verified after reading a finite amount of letters.

Claim 9.36 (Uniqueness in the construction)

If such $p_0, q_1, p_1, \dots, q_n, p_n, q$ and $F', L', L_1, \dots, L_n, R$ exist, they must be unique.

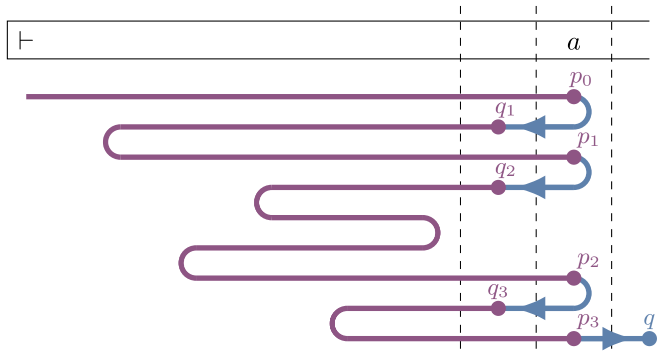


Figure 9.35: Structure of an initial run.

Thanks to Claim 9.36, one can set $\delta\text{-initial}_{ua}(F, L) := q$ in a well-defined fashion. Furthermore, we update the registers $\mathsf{next}_{F,L} \mapsto \mathsf{next}_{F',L'}\lambda(p_0, a, A^*)\mathsf{right}_{q_1,L_1}\lambda(p_1, a, A^*)\cdots\mathsf{right}_{q_n,L_n}\lambda(p_n, a, R)$.

For the case of frozen_F which does not depend on L, we observe that there exists at most one F' such that $\delta\text{-initial}_u(F', L')$ is defined for some $L' \in \mathsf{Look}$ and $F = a^{-1}F'$. Indeed, the converse would contradict Item (2)(c). Therefore, we update $\mathsf{frozen}_F \mapsto \mathsf{frozen}_{F'}$ in a well-defined fashion.

Claim 9.37 (Invariant preservation)

After this operation, Invariant (2) holds for $\delta\text{-initial}_{ua}$.

9.4.2.4 Producing an output. Once the two previous operations are performed, $\mathscr{S}$ tries to add a value in out. The latter is done as follows. First, we check if there exists $L \in \mathsf{Look}$ such that $\delta\text{-initial}_{ua}(A^*, L)$ is defined. Due to Item (2)(c), it means that we have $F = A^*$ whenever $\delta\text{-initial}_{ua}(F, L)$ is defined. In this case we send frozen_{A^*} in out and replace the frozen by the next, formally:

- we update $\mathsf{out} \mapsto \mathsf{out}\ \mathsf{frozen}_{A^*}$;
- for all $L \in \mathsf{Look}$ such that the value $\delta\text{-initial}_{ua}(A^*, L)$ is defined, we update $\mathsf{frozen}_L \mapsto \mathsf{next}_{A^*,L}$ and $\mathsf{next}_{A^*,L} \mapsto \varepsilon$;
- we replace $\delta\text{-initial}_{ua}$ by the function $g : \mathsf{Look} \times \mathsf{Look} \rightharpoonup Q$ defined as follows: $g(L, A^*) := q$ whenever $\delta\text{-initial}_{ua}(A^*, L) = q$.

It is easy to see that this operation preserves the Invariant (2) for $\delta\text{-right}_{ua}$.

9.4.3 Correctness of the construction

In this section, we justify that the machine $\mathscr{S}$ computes f and is 1-bounded.

9.4.3.1 Correctness on the domain. Let $v \in \mathsf{Dom}(f)$, we want to justify that $\mathscr{S}$ produces an infinite output which is $f(v)$. First, it follows from Invariant (2) that for all $i \geqslant 0$, there exist unique F_i, L_i with $v[i{+}1{:}] \in (F_i \cap L_i)A^\omega$. Then $\lambda^*(\overrightarrow{q_0}, \vdash v[1{:}i], v[i{+}1{:}]) = [\![\mathsf{out}]\!]_{v[1:i]}[\![\mathsf{frozen}_{F_i}]\!]_{v[1:i]}[\![\mathsf{frozen}_{F_i,L_i}]\!]_{v[1:i]}$. Therefore we obtain $[\![\mathsf{out}]\!]_{v[1:i]}[\![\mathsf{frozen}_{F_i}]\!]_{v[1:i]}[\![\mathsf{frozen}_{F_i,L_i}]\!]_{v[1:i]} \to f(v)$.

To conclude, it is thus sufficient to show that $|[\![\mathsf{out}]\!]_{v[1:i]}| \to \infty$. For this, we first claim that the operation described in Section 9.4.2.4 is applied infinitely often[4]. Indeed, given $i \geqslant 0$, there exists $j \geqslant i$ such

[4] This is the reason why we needed the function δ-initial to start from $\mathsf{Look} \times \mathsf{Look}$ and not only Look.

that $v[i{:}j] \in F_i \cap L_i$. If the operation of Section 9.4.2.4 is not applied before reading position j, one can show $F_j = (v[i{:}j])^{-1} F_i$, thus $\varepsilon \in F_j$. Since $F_j \in \mathsf{Look}$ is suffix closed by Claim 9.32, then $F_j = A^*$ and the operation of Section 9.4.2.4 is applied in position j. As a consequence, we have $[\![\mathsf{next}_{F_i,L_i}]\!]_{v[1:i]} = \varepsilon$ infinitely often. Furthermore, one can show that the function $i \mapsto |[\![\mathsf{out}]\!]_{v[1:i]}[\![\mathsf{frozen}_{F_i}]\!]_{v[1:i]}|$ is non-decreasing. Thus we must have $|[\![\mathsf{out}]\!]_{v[1:i]}[\![\mathsf{frozen}_{F_i}]\!]_{v[1:i]}| \to \infty$. But since $[\![\mathsf{frozen}_{F_i}]\!]_{v[1:i]}$ is added into out infinitely often, this implies that $|[\![\mathsf{out}]\!]_{v[1:i]}| \to \infty$.

9.4.3.2 Correctness out of the domain. We first assume that all the states of $\mathscr{T}$ are final, i.e. $F = Q$. By a similar argument, one can show that if $v \notin \mathsf{Dom}(f)$, then either $\mathscr{S}$ gets blocked at some point, or it produces a finite output. If $F \neq Q$, one would have to adapt the construction by adding a component to the output of δ-$\mathsf{initial}_u$ for $u \in A^*$, in order to keep track of the fact that a right-to-right run has visited an accepting state or not, as mentioned in Section 9.2.4.3.

9.4.3.3 1-boundedness of the streaming string transducer. We finally justify that $\mathscr{S}$ is 1-bounded. For this, we first observe that an analogue of Claim 9.18 holds.

Claim 9.38 (Copies are loops)

Let $u \in A^*$ and $0 \leqslant i \leqslant |u|$. Let s be the substitution applied by $\mathscr{S}$ when reading $u[i{+}1{:}|u|]$, after having read $u[1{:}i]$. For all $p \in Q$, if right_p occurs $k \geqslant 1$ times in $s(\mathsf{right}_q)$, there exists $v \in A^\omega$ such that $\delta^*(\overleftarrow{q}, u, v)$ has shape $\overrightarrow{r}$ and $\mathsf{maxi\text{-}run}(\overleftarrow{q}, u, v)$ visits k times $(|\vdash u[1{:}i]|, p)$.

Proof idea. By induction on $|u|$. ◀

Claim 9.38 immediately gives a contradiction if $k \geqslant 2$, since $\mathsf{maxi\text{-}run}(\overleftarrow{q}, u, v)$ cannot visit twice the same configuration without looping. Similar results can be shown when dealing with frozen, next and out. All in all, $\mathscr{S}$ is 1-bounded (assuming that it was trim, i.e. that any of its states was accessible).

9.5 Composition of deterministic regular functions

The goal of this section is to show closure under composition of deterministic regular functions, which was first claimed in [CDFW23, Theorem 3.1]. The proof is inspired by the historical proof of [CJ77] for regular functions of finite words, which relies on the fact that lookarounds can be removed for 2DT .

Theorem 9.39 (Composition of deterministic functions)

The class of deterministic regular functions is (effectively) closed under composition.

Proof. Let $f : A^\omega \rightharpoonup B^\omega$ (resp. $f' : B^\omega \rightharpoonup C^\omega$) be a deterministic regular function computed by the normalized $2\mathsf{DT}^\omega$ $\mathscr{T} = (A, B, Q, q_0, F, \delta, \lambda)$ (resp. $\mathscr{T}' = (B, C, Q', q'_0, F', \delta', \lambda')$). By Theorem 9.4, we only need to build a $2\mathsf{DT}^\omega$ with finite lookarounds which computes $f' \circ f$. Since the machines are normalized, we do not have to deal with final conditions. Furthermore, we assume without losing generalities that $\mathscr{T}$ produces exactly one letter at each transition, i.e. $\lambda(q, a) \in B$ for all $(q, a) \in \mathsf{Dom}(\lambda)$ with $a \in A$. Indeed, one can replace outputs ε by an extra fresh letter $\#$ which is systematically ignored by $\mathscr{T}'$. Formally, when $\mathscr{T}'$ moves $\triangleleft$ (resp. $\triangleright$) on its tape and meets letter $\#$, it goes on moving $\triangleleft$ (resp. $\triangleright$) until it meets another letter.

Let $u \in \mathsf{Dom}(f)$ be such that $f(u) \in \mathsf{Dom}(f')$. Let $(q_1, i_1) \to (q_2, i_2) \to \cdots$ (resp. $(q'_1, i'_1) \to (q_2, i'_2) \to \cdots$) be the accepting n-run of $\mathscr{T}$ (resp. of $\mathscr{T}'$) labelled by u (resp. by $f(u)$). Observe that for all $i' \geqslant 1$ we have $f(u)[i'] = \lambda(q_{i'}, u[i_{i'}])$ because $\mathscr{T}$ produces exactly one letter at each transition. Thus for all $\ell \geqslant 1$, we have $f(u)[i'_\ell] = \lambda(q_{i'_\ell}, u[i_{i'_\ell}])$.

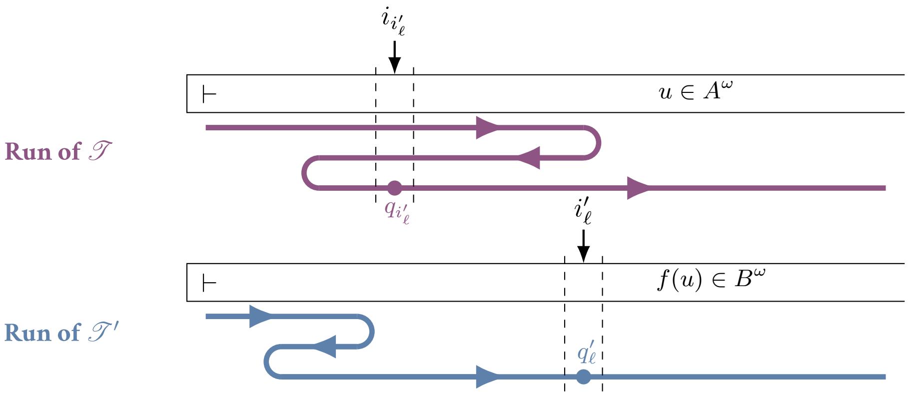


Figure 9.40: Product construction for the composition of 2DT$^\omega$.

The main idea is to build $\mathscr{U}$ by doing a product construction of $\mathscr{T}$ and $\mathscr{T}'$. When its input is u, we want $\mathscr{U}$ to successively build the pairs of configurations $(q'_\ell, i'_\ell), (q_{i'_\ell}, i_{i'_\ell})$ for $\ell \geqslant 1$, as depicted in Figure 9.40. Formally, the states q'_ℓ and $q_{i'_\ell}$ can be maintained in the bounded memory of $\mathscr{U}$. The position $i_{i'_\ell}$ will be the current position of $\mathscr{U}$. However, i'_ℓ can only be maintained implicitly, because the input of $\mathscr{U}$ is $u \in A^\omega$ and not $f(u) \in B^\omega$.

Now, let us explain the updates of $\mathscr{U}$. Let $\ell \geqslant 1$ and assume that $\mathscr{U}$ is in position $i_{i'_\ell}$, that it stores q'_ℓ, and $q_{i'_\ell}$ and that it has output $\lambda'(q'_1, f(u)[i'_1]) \cdots \lambda'(q'_{\ell-1}, f(u)[i'_{\ell-1}])$ so far. We want $\mathscr{U}$ to preserve this invariant at step $\ell{+}1$. First note that $b := f(u)[i'_\ell] = \lambda(q_{i'_\ell}, u[i_{i'_\ell}])$ can be determined by $\mathscr{U}$. Thus $\mathscr{U}$ can compute $\delta'(q'_\ell, b) = (q'_\ell, \star)$ and output $\lambda'(q'_\ell, b)$. It remains to determine $q_{i'_{\ell+1}}$ and move to $i_{i'_{\ell+1}}$. Two cases occur (beware that they are *not* symmetrical):

- either $\mathscr{T}'$ moves forward, i.e. $\star = \triangleright$. In this case, one has to determine the next configuration of $\mathscr{T}$, which is possible by computing $\delta(q_{i'_\ell}, u[i_{i'_\ell}])$ and moving accordingly;
- or $\mathscr{T}'$ moves backward, i.e. $\star = \triangleleft$. In this case, one has to determine the *previous* configuration of $\mathscr{T}$. However, since $\mathscr{T}$ is not co-deterministic, the current configuration may have several possible predecessors. In order to detect which is the correct one, finite lookarounds comes in handy. The basic idea is to use them to rewind the initial n-run of $\mathscr{T}$, as justified in Claim 9.41. One only needs to check a "finite" regular property, since only a prefix of the input has been visited by the computation of the two-way transducer.

Claim 9.41 (Finite lookaheads for rewinding initial runs)

For all $p, q \in Q$ and $a \in A$, one can build a regular languages $L, R \subseteq A^*$ such that the following conditions are equivalent for all $v \in A^*$ and $1 \leqslant i \leqslant |v|$:
- $v[1{:}i] \in L$, $v[i] = a$ and $v[i{+}1{:}|v|]$;
- the longest initial n-run of $\mathscr{T}$ labelled by v contains the sequence $(p, i{-}1) \to (q, i)$.

Proof idea. The behavior of a 2DT$^\omega$ (or simply a 2DT) over a finite input can be described using regular languages (recall the notion of transition monoid). ◀

By using a bunch of finite lookarounds of the previous form, one can determine the previous configuration $(i_{i'_{\ell+1}}, q_{i'_{\ell+1}})$ of $\mathscr{T}$. Only one of the finite lookarounds will be a admissible, since the configuration $(i_{i'_\ell}, q_{i'_\ell})$ is visited only once in the accepting n-run of $\mathscr{T}$.

Our machine $\mathscr{U}$ produces the same output as $\mathscr{T}'$ on input $f(u)$, that is $f'(f(u)) \in C^\omega$. Conversely, if we have $u \notin \mathsf{Dom}(f' \circ f)$, the computation will either fail or produce a finite output. ◀

To conclude Section 9.5, let us discuss a low hanging consequence for DSST$^\omega$. Informally, a DSST$^\omega$ with *finite lookarounds* can be defined as a DSST$^\omega$ which selects its transitions depending on disjoint sets of regular languages, as explained for 2DT$^\omega$ (or even 1DT$^\omega$) with finite lookarounds in Definition 9.2.

Corollary 9.42 (Finite lookaround removal for DSST$^\omega$)

The functions computed by a copyless (or K-bounded) DSST$^\omega$ with finite lookarounds are exactly the deterministic regular functions. Furthermore, the conversions are effective.

Proof sketch. A function computed by a copyless or a K-bounded DSST$^\omega$ with finite lookarounds $\mathscr{S}$ can (effectively) be written as the composition of a deterministic regular function (computed by a 1DT$^\omega$ with finite lookarounds) which precomputes the run of $\mathscr{S}$ and determines which finite lookarounds are admissible, and a deterministic regular function (computed by a copyless or a K-bounded DSST$^\omega$) which simulates the updates of $\mathscr{S}$. We conclude by Theorem 9.39. ◀

9.6 Decomposition of deterministic regular functions

In this section, we show that deterministic regular functions can be obtained as a composition of basic functions. This result can be seen as an analogue of Theorem 1.32 over infinite words. It illustrates the fact that deterministic regular functions mainly differ from the sequential ones due to their abitiliy to copy or reverse finite factors of the input (using map-copy-reverse$^\omega$ from Example 8.28).

Theorem 9.43 originates from [CDFW23, Theorem 3.6]. Its proof is somehow technical and it confronts once more the main difficulty of this chapter: contrary to regular functions, deterministic regular functions cannot check properties which concern the "infinite" future of the input.

Theorem 9.43 (Decomposition of deterministic regular functions)

A function of infinite words is deterministic regular if and only if it can be written as a composition of sequential functions and map-copy-reverse$^\omega$ functions. The conversions are effective.

Proof sketch. The right-to-left implication is clear by Theorem 9.39 and since the sequential and map-copy-reverse$^\omega$ functions are deterministic regular. It remains to show the decomposition result. Analogue results over finite words (Theorems 1.32 and 1.45) were shown in [Boj18, BS20] by using factorization forests: they first build a forest with a rational function (Theorem 2.21), and then use its structure to simulate the runs of a transducer using basic functions.

We intend to follow a similar proof sketch for infinite words. Even if factorization forests can be generalized in this setting (see e.g. [Col10]), it is not known whether they can be computed using compositions of sequential and map-copy-reverse$^\omega$ functions. Therefore we use instead a weakened object named *forward factorization forests*, introduced by Colcombet in [Col07]. We claim in Section 9.6.1 that forward factorization forests of bounded height can be computed by a sequential function. Then, we introduce in Section 9.6.2 a class of functions of both finite and infinite words $\mathfrak{C}$ which is closed under composition, and show by induction in Section 9.6.3 that this class enables to "simulate" the runs of a 2DT$^\omega$ when a forward factorization forest of the input is given (using both finite and infinite words is necessary for the induction step). ◀

Before giving the detail of this proof, let us discuss an easy consequence for regular functions. To the knowledge of the author, Corollary 9.44 was not explicitly known in the literature.

Corollary 9.44 (Decomposition of regular functions)

A function of infinite words is regular if and only if can be written as a composition of rational functions and map-copy-reverse$^\omega$ functions. The conversions are effective.

Proof. The right-to-left implication follows since regular functions are closed under composition (Theorem 8.36). For the converse implication, a function computed by a 2DT$^\omega$ with ω-lookarounds can (effectively) be written as the composition of a rational function (computed by an ω-bimachine which precomputes which ω-lookarounds) succeed in each position of the input, and a deterministic regular function which simulates the 2DT$^\omega$ once the ω-lookarounds are known. We then apply Theorem 9.43 to decompose this deterministic regular function. ◀

The rest of this section is devoted to a detailed proof of Theorem 9.43.

9.6.1 Forward factorization forests

First, we study the notion of *forward factorization forest* introduced in [Col07][5]. Recall that $\langle t_1\rangle \cdots \langle t_n\rangle$ denotes a finite tree whose root is not labelled, and whose subtrees are $t_1, \dots, t_n$. We extend this notation to infinitely branching trees, by writing $\langle t_1\rangle\langle t_2\rangle \cdots$ for such a branching.

Definition 9.45 (Forward factorization forest)

Let $\mu\colon A^* \to \mathbb{M}$ be a monoid morphism and $u \in A^\infty$. We say that $\mathcal{F}$ is a *forward μ-forest* of u if:

- either $u = a \in A$ and $\mathcal{F} = a$;
- or $\mathcal{F} = \langle \mathcal{F}_1\rangle \cdots \langle \mathcal{F}_n\rangle$, $u = u_1 \cdots u_n$ and for all $1 \leqslant i \leqslant n$, $\mathcal{F}_i$ is a forward μ-forest of $u_i \in A^\infty \smallsetminus \{\varepsilon\}$ and for all $1 < i,j < n$, $\mu(u_i)\mu(u_j) = \mu(u_i)$;
- or $\mathcal{F} = \langle \mathcal{F}_1\rangle\langle \mathcal{F}_2\rangle \cdots$ where $u = u_1 u_2 \cdots \in A^\omega$, for all $1 \leqslant i$, $\mathcal{F}_i$ is forward μ-forest of $u_i \in A^+$, and for all $1 < i,j$, $\mu(u_i)\mu(u_j) = \mu(u_i)$;

Observe that the second rule is a weakening of the idempotent rule for factorization forests presented in Definition 2.17. For $n = 2$, it does not provide any constraint on the factorization. For $n \geqslant 3$, it implies that all the inner factors are idempotent (since we have $\mu(u_i)\mu(u_i) = \mu(u_i)$ for $1 < i < n$) and "absorbing on the left" (hence *$\mathcal{L}$-equivalent* in the sense of *Greene's relations*, see e.g. [Col11]), but not necessarily equal. Furthermore, there is no assumption on the first and the last factor (the latter can even be an infinite word). Finally, the third rule is an infinitely branching version of the second rule.

Given $d \geqslant 1$ and $u \in A^\infty$, we let $\mathsf{f\text{-}Forests}^d_\mu(u)$ be the set of all forward μ-forests of u of height at most d (defined by induction). For $u \in A^+$, a tree $\mathcal{F} \in \mathsf{f\text{-}Forests}^d_\mu(u)$ can be seen as a finite word over the alphabet $A \uplus \{\langle, \rangle\}$. For $u \in A^\omega$, $\mathcal{F} \in \mathsf{f\text{-}Forests}^d_\mu(u)$ can be seen as an infinite word. In this case, some of the opening brackets may remain open forever, since e.g. $\langle \mathcal{F}\rangle$ has to be encoded by $\langle \mathcal{F}$ when $\mathcal{F}$ is infinite. As for factorization forests, we let the function $\mathsf{word}^d_\mu\colon \mathsf{f\text{-}Forests}^d_\mu \to A^\infty$ be the morphism which removes the letters in $\{\langle, \rangle\}$, i.e. which maps $\mathcal{F} \in \mathsf{f\text{-}Forests}^d_\mu(u)$ to $u \in A^\infty$.

The next result originates from [Col07, Theorem 1]. However, it is not directly stated under this formalism since the author uses *forward Ramseyan splits* instead of forward factorization forests. An explicit reformulation in terms of forward factorization forests is available in [Boj09, Theorem 7] which we follow in Theorem 9.46. We directly instantiate the result in the case of infinite words.

[5] Formally, this paper describes the equivalent notion of *forward Ramseyan splits*.

Theorem 9.46 (Sequential Simon)

Given a morphism $\mu \colon A^* \to \mathbb{M}$ into a finite monoid, one can build a sequential function $\text{f-forest}_\mu \colon A^\omega \to \text{f-Forests}_\mu^{|\mathbb{M}|}$ such that $\text{word}_\mu^{|\mathbb{M}|} \circ \text{f-forest}_\mu$ is the identity function over A^ω.

In Lemma 2.15, we have seen that the runs of a 2DT along a block of factors having the same idempotent value under the transition monoid of the 2DT enjoy a particular shape. Recall that we have extended the notion of transition monoid to 2DT^ω in Section 9.2.4. Lemma 9.48 adapts Lemma 2.15 for forward factorization forests, i.e. with different idempotents which are "absorbing on the left".

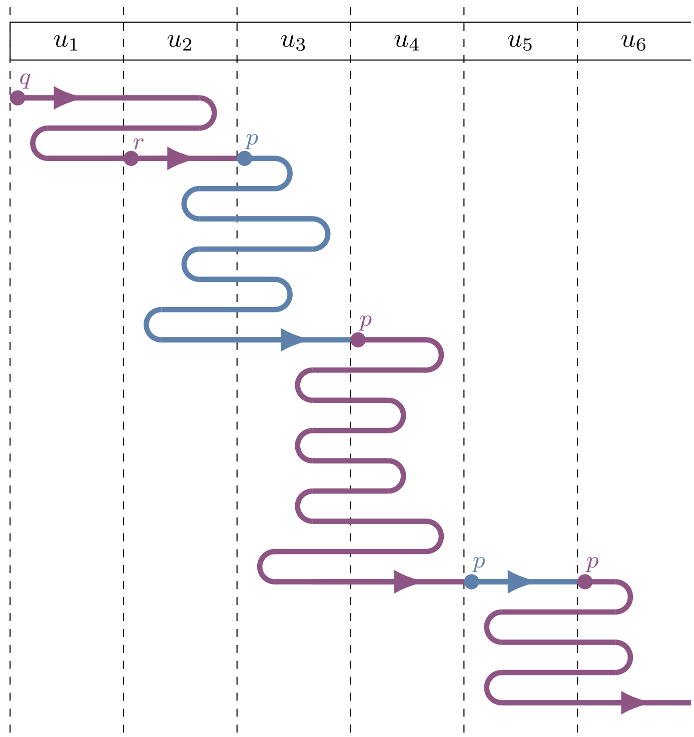


Figure 9.47: Shape of a run along a block in a forward factorization forest.

Lemma 9.48 (Runs in forward factorization forests)

Let $\mathscr{T} = (A, B, Q, q_0, F, \delta, \lambda)$ be a 2DT^ω with transition monoid $\mu \colon A^* \to \mathbb{T}$. Let $u = u_1 u_2 \cdots \in A^\infty$ be such that $\mu(u_i)\mu(u_j) = \mu(u_i)$ for all $1 < i, j$ such that u_i and u_j are defined and are not the last factor (i.e. u_{i+1} and u_{j+1} are also defined). If $\delta^*(\overrightarrow{q}, u_1 u_2) = \overrightarrow{p}$, then $\text{maxi-run}(\overrightarrow{q}, u)$ has shape $\text{maxi-run}(\overrightarrow{q}, u_1 u_2) \to \rho_3 \to \rho_4 \to \cdots$ where:

(1) for all $i \geqslant 3$, ρ_i starts in the first configuration of ρ which visits u_i;
(2) for all $i \geqslant 3$, ρ_i begins with a configuration of shape $(p, \_)$ (i.e. it starts in p);
(3) for all $i \geqslant 3$ such that u_i is not the last factor, ρ_i only visits the positions of u_i and u_{i-1} (it cannot go back to u_{i-2}).

Proof. We have $\delta^*(\overrightarrow{q}, u_1 u_2) = \overrightarrow{p}$, thus $\delta^*(\overrightarrow{q}, u_1 u_2 \cdots u_{i-1}) = \overrightarrow{p}$ for all $i \geqslant 2$ such that u_i is defined, since $\mu(u_2 \cdots u_{i-1}) = \mu(u_2)$. This means that u_i is visited by $\text{maxi-run}(\overrightarrow{q}, u_1 u_2)$, and furthermore that this visit starts in state p, giving Items (1) and (2) by defining ρ_i accordingly. For Item (3), let $i \geqslant 3$ be such that u_i is defined and not the last factor. We show that ρ_i only visits u_i and u_{i-1}. First, observe that this run does not visit u_{i+1} by construction of ρ_{i+1}. Second, let us consider the state r seen in the last visit of the first position of u_{i-1} in ρ_{i-1} (or in $\text{maxi-run}(\overrightarrow{q}, u_1u_2)$ if $i = 3$). Since $\mu(u_{i-1}u_i) = \mu(u_{i-1})$ because u_i is not the last factor, we have $\delta^*(\overrightarrow{r}, u_{i-1}u_i) = \delta^*(\overrightarrow{r}, u_{i-1}) = \overrightarrow{p}$ (this last equality follows from Item (2), because it describes the beginning of ρ_i). This means that when starting from r in the first position of u_{i-1}, $\mathscr{T}$ will execute the end of ρ_{i-1}, then ρ_i, and it will eventually leave $u_{i-1}u_i$ "by the right". Hence the run ρ_i stays in $u_{i-1}u_i$, until it goes to u_{i+1} in state p (and this is by construction the beginning of ρ_{i+1}). ◀

The shape of $\text{maxi-run}(\overrightarrow{q}, u)$ in Lemma 9.48 is depicted in Figure 9.47. This figure is roughly similar of Figure 2.14, but let us highlight the key differences between them. First, we have no control on the runs starting in the first and the last (if it exists) factors. The run starting in the last factor may even go back to u_1. Second, since the $\mu(u_i)$ are not all the same, the runs ρ_i for $i \geqslant 3$ may not cross the border between each u_{i-1} and u_i in the same fashion. The only information we get is that they begin in state p.

Another difference between factorization forests and forward factorization forests is that the case of Lemma 2.15 is symmetrical (i.e. it also holds when studying $\text{maxi-run}(\overleftarrow{q}, u)$) while Lemma 9.48 is not. This is first because in our case the input word may be infinite, thus it does not make sense to enter it "from the right". More interestingly, the reader is invited to note that the conditions $\mu(u_i)\mu(u_j) = \mu(u_i)$ are not symmetrical and provide no information on the runs which start on the right.

9.6.2 A class of functions closed under composition

Now, let us describe a class of functions $\mathfrak{C}$ which goes both from finite words to finite words and from infinite words to infinite words (i.e. the functions of $\mathfrak{C}$ has type $(A^* \rightharpoonup B^*) \cup (A^\omega \rightharpoonup B^\omega)$). Our goal is to show that any deterministic regular function can be computed as the restriction to infinite words of a function from $\mathfrak{C}$. However, this proof will be done by induction both over finite and infinite words, and we shall not know *a priori* if the current input is finite or not[6].

A *one-way deterministic transducer of finite and infinite words* consists of a 1DT $(A, B, Q, q_0, F, \delta, \lambda)$ and a 1DT^ω $(A, B, Q, q_0, F_\omega, \delta, \lambda)$, which share the same A, B, Q, q_0, δ and λ. Such a machine therefore describes a function of type $(A^* \rightharpoonup B^*) \cup (A^\omega \rightharpoonup B^\omega)$. The key property is that its runs over finite and infinite words have the same structure. We let the class of *sequential functions of finite and infinite words* be the class of functions computed by these machines.

Given an alphabet A and a fresh symbol $\# \notin A$, we also build the function basic-copy-reverse which has type $((A \uplus \{\#\})^* \to (A \uplus \{\#\})^*) \cup ((A \uplus \{\#\})^\omega \to (A \uplus \{\#\})^\omega)$ and is defined by:

- $\text{basic-copy-reverse}(u) = \text{map-copy-reverse}(u)$ for $u \in (A \uplus \{\#\})^*$, see Example 1.23;
- $\text{basic-copy-reverse}(u) = \text{map-copy-reverse}^\omega(u)$ for $u \in (A \uplus \{\#\})^\omega$, see Example 8.28.

Since we may use several distinct separating symbols, we shall say that basic-copy-reverse has *separator* $\#$ to say explicitly that the letter $\#$ is the one used to separate the factors of the input.

Definition 9.49 (Class $\mathfrak{C}$)

The class $\mathfrak{C}$ is the smallest class of functions which is closed under composition and contains both sequential functions of finite and infinite words and the basic-copy-reverse functions.

Before coming to the main proof in Section 9.6.3, we describe useful properties of this class $\mathfrak{C}$.

[6] Intuitively, checking if something is infinite or not requires ω-lookarounds, which are forbidden in our setting.

Example 9.50 (Map copy)

The function basic-copy: $(A \uplus \{\#\})^\infty \to (A \uplus \{\#\})^\infty$ is obtained from basic-copy-reverse by replacing each copy mirror factor $\widetilde{u_i}$ by u_i. This function belongs to $\mathfrak{C}$. Indeed, we apply basic-copy-reverse twice, which outputs a word of shape $u_1\#\widetilde{u_1}\#\widetilde{u_1}\#u_1\#u_2\cdots$, and the one can easily remove the useless pieces thanks to a sequential function of finite and infinite words.

More interestingly, we show that the class $\mathfrak{C}$ is closed under a "map" operator, which applies a given function to a sequence of finite or infinite words separated by a specific symbol. We even claim in Lemma 9.51 one can apply distinguished functions on the first n factors.

Lemma 9.51 (Map operator)

Let $f_1, f_2, \dots, f_n \colon A^\infty \rightharpoonup B^\infty \in \mathfrak{C}$ and $\#$ be a fresh symbol. One can build in $\mathfrak{C}$ a function $f_1\#f_2\#\cdots\#f_n\# \colon (A \uplus \{\#\})^\infty \rightharpoonup (B \uplus \{\#\})^\infty$ such that:

$$\begin{aligned} & f_1\#f_2\#\cdots\#f_n\#(u_1\#u_2\cdots) \\ & = f_1(u_1)\#f_2(u_2)\#\cdots\#f_n(u_n)\#f_n(u_{n+1})\#f_n(u_{n+2})\cdots \end{aligned}$$

whenever $u_1 \in \mathsf{Dom}(f_1), u_2 \in \mathsf{Dom}(f_2), \dots$.

Remark 9.52 (Map operator)

A few elements are left implicit in Lemma 9.51. First, if there are $k \leqslant n$ factors in the input, then $f_1\#\cdots\#f_n\#$ only applies the first functions $f_1, \dots, f_k$. Second, if the input is infinite, we must have $f_1(u_1)\#f_2(u_2)\#\cdots \in (B \uplus \{\#\})^\omega$ for the output to be defined.

Proof. We only deal with the case $n = 1$. The other cases can be treated in a similar fashion, using sequential functions to drop specific marks on the n first pieces. Let $f \colon A^\infty \rightharpoonup B^\infty \in \mathfrak{C}$, we show how to build $f\#$ by induction on the construction of f. If f is sequential then we build a sequential $f\#$ described by a one-way deterministic transducer similar to that of f, except if a $\#$ is seen, in which case it produces the (finite) final output of the transducer in the current state, and goes back to the initial state to pursue its computation. If $f = g \circ h$ the result is obvious by induction hypothesis. If f is basic-copy-reverse with separator $\$$ (thus $\$ \neq \#$ since $\#$ is fresh), we first apply the sequential function which turns each $\#$ into $\#\$$. Then we apply basic-copy-reverse with separator $\$$ on the whole input. We conclude by applying a sequential function which replaces each factor $\#\$\#$ by a single $\$$, and in this case replaces the next occurrence of $\$$ by $\#$. ◀

We conclude this section by giving one last property of $\mathfrak{C}$. Observe that Theorem 1.32 exactly states that any regular function of finite words can be written as the restriction of a function of $\mathfrak{C}$ to finite words. Using this result, we claim in Lemma 9.53 that the runs of a $\mathsf{2DT}^\omega$ which start on the *right* of a *finite* input can be simulated by a function of $\mathfrak{C}$. Using this result will be necessary, since Lemma 9.48 provides no information to describe runs which start on the right, as mentioned above. To homogenize the forthcoming proofs, we assume anyway that a forward factorization forest is given as input.

Lemma 9.53 (Left runs)

Let $\mathscr{T} = (A, B, Q, q_0, F, \delta, \lambda)$ be a $\mathsf{2DT}^\omega$ with transition morphism $\mu : A^* \to \mathbb{T}$. For all $d \geqslant 0$, one can build a function $\overleftarrow{\mathsf{simul}\text{-}\mathsf{d}} \colon (A \cup \{\langle, \rangle\} \cup \overleftarrow{Q})^* \rightharpoonup (A \cup \{\langle, \rangle\} \cup B \cup \overleftarrow{Q} \cup \overrightarrow{Q})^*$ which belongs to $\mathfrak{C}$, such that for all $u \in A^+$, $\mathcal{F} \in \mathsf{f}\text{-}\mathsf{Forests}^\mu_d(u)$ and $q \in Q$:

- if $\delta^*(\overleftarrow{q}, u) = \overrightarrow{p}$ and $\lambda^*(\overleftarrow{q}, u) = \alpha$ then $\overleftarrow{\text{simul-d}}(\mathcal{F}\overleftarrow{q}) = \alpha\mathcal{F}\overrightarrow{p}$;
- if $\delta^*(\overleftarrow{q}, u) = \overrightarrow{p}$ and $\lambda^*(\overleftarrow{q}, u) = \alpha$ then $\overleftarrow{\text{simul-d}}(\mathcal{F}\overleftarrow{q}) = \alpha\overleftarrow{p}\mathcal{F}$.

Proof. Such a function (from finite words to finite words) can be computed by a 2DT which ignores the symbols $\langle$ and $\rangle$. This function can be decomposed as composition of functions from $\mathfrak{C}$ thanks to Theorem 1.32. Recall that having a forward forest is not useful at that stage. ◀

9.6.3 Inductive construction of the runs

The core of the proof of Theorem 9.43 consists in showing Lemma 9.54 by induction on $d \geqslant 1$. It is an analogue of Lemma 9.53 when the runs start on the left of a finite or infinite word.

Lemma 9.54 (Key induction step)

Let $\mathscr{T} = (A, B, Q, q_0, F, \delta, \lambda)$ be a $2\mathrm{DT}^\omega$ with transition morphism $\mu : A^* \to \mathbb{T}$. For all $d \geqslant 0$, one can build a function:

$$\overrightarrow{\text{simul-d}} : (A \cup \{\langle, \rangle\} \cup \overrightarrow{Q})^\infty \rightharpoonup (A \cup \{\langle, \rangle\} \cup B \cup \overleftarrow{Q} \cup \overrightarrow{Q})^\infty$$

which belongs to $\mathfrak{C}$ such that for all $q \in Q$, $u \in A^\infty$ and $\mathcal{F} \in \text{f-Forests}^d_\mu(u)$:

- if $\delta^*(\overrightarrow{q}, u) = \overleftarrow{p}$ and $\lambda^*(\overrightarrow{q}, u) = \alpha \in B^*$ then $\overrightarrow{\text{simul-d}}(\overrightarrow{q}\mathcal{F}) = \alpha\overleftarrow{p}\mathcal{F}$;
- if $\delta^*(\overrightarrow{q}, u) = \omega$ and $\lambda^*(\overrightarrow{q}, u) = \alpha \in B^\omega$ then $\overrightarrow{\text{simul-d}}(\overrightarrow{q}\mathcal{F}) = \alpha$;
- if $u \in A^+$, $\delta^*(\overrightarrow{q}, u) = \overrightarrow{p}$ and $\lambda^*(\overrightarrow{q}, u) = \alpha \in B^*$ then $\overrightarrow{\text{simul-d}}(\overrightarrow{q}\mathcal{F}) = \alpha\mathcal{F}\overrightarrow{p}$.

Proof sketch. We build the function $\overrightarrow{\text{simul-d}}$ by induction on $d \geqslant 1$. For $d = 1$ the result is obvious since necessarily $u = \mathcal{F} = a$. For the induction step with $d+1$, Lemma 9.48 shows that the run maxi-run$(\overrightarrow{q}, u)$ can be decomposed by following the structure of $\mathcal{F}$. We then rely on $\overrightarrow{\text{simul-d}}$ and $\overleftarrow{\text{simul-d}}$ to build pieces of this run and we re-combine them together. ◀

The rest of Section 9.6.3 is devoted to the detailed proof of Lemma 9.54 by induction on $d \geqslant 1$.

Assume that the function $\overrightarrow{\text{simul-d}}$ in $\mathfrak{C}$ is built for some $d \geqslant 1$. We describe how to build the function $\overrightarrow{\text{simul-(d+1)}}$ in $\mathfrak{C}$. We first create a function from $\mathfrak{C}$ which checks if the input has a correct shape, applies $\overrightarrow{\text{simul-d}}$ if it is the case, and otherwise behaves as the identity function.

Claim 9.55 (Try right)

One can build a function $\overrightarrow{\text{try-simul}}$ in $\mathfrak{C}$, which behaves:

- as $\overrightarrow{\text{simul-d}}$ if the input begins with some letter $\overrightarrow{q} \in \overrightarrow{Q}$;
- as the identity function if it contains no $\overrightarrow{q} \in \overrightarrow{Q}$.

Proof. We first apply a sequential function which writes letter \$ before any $\overrightarrow{q}$ of the input. Then we apply id\$$\overrightarrow{\text{simul-d}}$\$ from Lemma 9.51, where id denotes the identity function. ◀

Let $u \in A^\infty$, $\mathcal{F} \in \text{f-Forests}^{d+1}_\mu(u)$ and $q_1 \in Q$. Up to first applying a sequential function which removes the first $\langle$ and replaces the appropriate factors $\rangle\langle$ by $\#$, we can assume that $\overrightarrow{q_1}\mathcal{F}$ has shape

$\overrightarrow{q_1}\mathcal{F}_1\#\mathcal{F}_2\#\mathcal{F}_3\#\cdots$ where $\mathcal{F}_i \in \text{f-Forests}^d_\mu(u_i)$, $u_i \in A^\infty$, $u = u_1u_2\cdots$ and $\mu(u_i)\mu(u_j) = \mu(u_i)$ for all $2 \leqslant i$ such that u_i is not the last factor of u.

Our goal is to simulate the run $\text{maxi-run}(\overrightarrow{q_1}, u)$, for this we use the slicing given by Lemma 9.48. We first simulate the run $\text{maxi-run}(\overrightarrow{q_1}, u_1u_2)$ in Section 9.6.3.1 (this case is specific since Lemma 9.48 provides no properties of this run). For this purpose, we use alternatively the functions $\overrightarrow{\text{simul-d}}$ and $\overleftarrow{\text{simul-d}}$. Then, we show in Section 9.6.3.2 how to build the runs $\rho_3, \rho_4, \dots$. The main idea is to build all these runs in parallel, while crucially relying on the fact that they all begin in the same state.

9.6.3.1 Building the run in u_1u_2. We first deal with the run $\text{maxi-run}(\overrightarrow{q_1}, u_1u_2)$ which is not controlled by Lemma 9.48. The enumeration below describes the operations performed.

(1) We first apply $\overrightarrow{\text{try-simul}}\#$ which outputs the following (with $\alpha_1 := \lambda^*(\overrightarrow{q_1}, u_1)$):
  - (a) if $\delta^*(\overrightarrow{q_1}, u_1) = \omega$ (necessarily $u = u_1 \in A^\omega$) and $\alpha_1 \in B^\omega$, then α_1;
  - (b) if $\delta^*(\overrightarrow{q_1}, u_1) = \overleftarrow{q_2}$, then $\alpha_1\overleftarrow{q_2}\mathcal{F}_1\#\mathcal{F}_2\cdots$;
  - (c) if $\delta^*(\overrightarrow{q_1}, u_1) = \overrightarrow{q_2}$ (necessarily $u_1 \in A^+$), then $\alpha_1\mathcal{F}_1\overrightarrow{q_2}\#\mathcal{F}_2\cdots$.

(2) We apply a sequential function which replaces the first $\overrightarrow{q}\#$ with $q \in Q$ by $\#\overrightarrow{q}$. Once this is done, we apply once more $\overrightarrow{\text{try-simul}}\#$, which yields the following cases:
  - (a) if $\delta^*(\overrightarrow{q_1}, u_1) = \omega$ and $\alpha_1 \in B^\omega$, then α_1;
  - (b) if $\delta^*(\overrightarrow{q_1}, u_1) = \overleftarrow{q_2}$, then $\alpha_1\overleftarrow{q_2}\mathcal{F}_1\#\mathcal{F}_2\cdots$;
  - (c) if $\delta^*(\overrightarrow{q_1}, u_1) = \overrightarrow{q_2}$ and (with $\alpha_2 := \lambda^*(\overrightarrow{q_2}, u_2)$):
    - (i) $u = u_1$, then $\alpha_1\mathcal{F}_1\overrightarrow{q_2}$;
    - (ii) $\delta^*(\overrightarrow{q_2}, u_2) = \omega$ and $\alpha_2 \in B^\omega$, then $\alpha_1\mathcal{F}_1\#\alpha_2$;
    - (iii) if $\delta^*(\overrightarrow{q_2}, u_2) = \overleftarrow{q_3}$, then $\alpha_1\mathcal{F}_1\#\alpha_2\overleftarrow{q_3}\mathcal{F}_2\cdots$;
    - (iv) if $\delta^*(\overrightarrow{q_2}, u_2) = \overrightarrow{p}$, then $\alpha_1\mathcal{F}_1\#\alpha_2\mathcal{F}_2\overrightarrow{p}\cdots$.

(3) In Item (2)(c)(iii), we need to compute $\overleftarrow{\text{simul-d}}(\mathcal{F}_1\overleftarrow{q_3})$, thus to create a factor $\mathcal{F}_1\overleftarrow{q_3}$. For this, we add a fresh symbol \$ before $\mathcal{F}_2$ (if it exists) and apply basic-copy with separator \$, which yields:
  - (a) if $\delta^*(\overrightarrow{q_1}, u_1) = \omega$ and $\alpha_1 \in B^\omega$, then α_1;
  - (b) if $\delta^*(\overrightarrow{q_1}, u_1) = \overleftarrow{q_2}$, then $\alpha_1\overleftarrow{q_2}\mathcal{F}_1\#\$\alpha_1\overleftarrow{q_2}\mathcal{F}_1\#\$\mathcal{F}_2\cdots$;
  - (c) if $\delta^*(\overrightarrow{q_1}, u_1) = \overrightarrow{q_2}$ and:
    - (i) $u = u_1$, then $\alpha_1\mathcal{F}_1\overrightarrow{q_2}\$\alpha_1\mathcal{F}_1\overrightarrow{q_2}$;
    - (ii) $\delta^*(\overrightarrow{q_2}, u_2) = \omega$ and $\alpha_2 \in B^\omega$, then $\alpha_1\mathcal{F}_1\#\alpha_2$;
    - (iii) if $\delta^*(\overrightarrow{q_2}, u_2) = \overleftarrow{q_3}$, then $\alpha_1\mathcal{F}_1\#\alpha_2\overleftarrow{q_3}\$\alpha_1\mathcal{F}_1\#\alpha_2\overleftarrow{q_3}\$\mathcal{F}_2\cdots$;
    - (iv) if $\delta^*(\overrightarrow{q_2}, u_2) = \overrightarrow{p}$, then $\alpha_1\mathcal{F}_1\#\alpha_2\$\alpha_1\mathcal{F}_1\#\alpha_2\$\mathcal{F}_2\overrightarrow{p}\cdots$.

  Then we use a well-chosen sequential function to remove the useless symbols, which yields:
  - (a) if $\delta^*(\overrightarrow{q_1}, u_1) = \omega$ and $\alpha_1 \in B^\omega$, then α_1;
  - (b) if $\delta^*(\overrightarrow{q_1}, u_1) = \overleftarrow{q_2}$, then $\alpha_1\overleftarrow{q_2}\mathcal{F}_1\#\mathcal{F}_2\cdots$;
  - (c) if $\delta^*(\overrightarrow{q_1}, u_1) = \overrightarrow{q_2}$ and:
    - (i) $u = u_1$, then $\alpha_1\mathcal{F}_1\overrightarrow{q_2}$;
    - (ii) $\delta^*(\overrightarrow{q_2}, u_2) = \omega$ and $\alpha_2 \in B^\omega$, then $\alpha_1\alpha_2$;
    - (iii) if $\delta^*(\overrightarrow{q_2}, u_2) = \overleftarrow{q_3}$, then $\alpha_1\alpha_2\$\mathcal{F}_1\overleftarrow{q_3}\$\#\mathcal{F}_2\cdots$;
    - (iv) if $\delta^*(\overrightarrow{q_2}, u_2) = \overrightarrow{p}$, then $\alpha_1\alpha_2\mathcal{F}_1\#\mathcal{F}_2\overrightarrow{p}\cdots$.

(4) In order to compute $\text{maxi-run}(\overleftarrow{q_3}, u_1)$, we build the function $\overleftarrow{\text{try-simul}} := \text{id}\$\overleftarrow{\text{simul-d}}\$\text{id}\$$. This function is inspired by $\overrightarrow{\text{try-simul}}$ from Claim 9.55. Observe that if it meets not \$, it will just behave as the identity function id. Therefore we apply the function $\overleftarrow{\text{try-simul}}\#\text{id}$.

(5) By iterating $|Q|$ times the previous steps, one can simulate the whole run $\text{maxi-run}(\overrightarrow{q_1}, u_1u_2)$. We obtain the following (where $\alpha := \lambda^*(\overrightarrow{q_1}, u_1u_2)$ if u_2 exists and $\alpha := \lambda^*(\overrightarrow{q_1}, u_1)$ otherwise):
  - (a) if $\delta^*(\overrightarrow{q_1}, u_1) = \omega$ or $\delta^*(\overrightarrow{q_1}, u_1u_2) = \omega$ and $\alpha \in B^\omega$, then α;
  - (b) if $\delta^*(\overrightarrow{q_1}, u_1) = \overleftarrow{q_2}$ or $\delta^*(\overrightarrow{q_1}, u_1u_2) = \overleftarrow{q_2}$, then $\alpha\overleftarrow{q_2}\mathcal{F}_1\#\mathcal{F}_2\cdots$;
  - (c) if $\delta^*(\overrightarrow{q_1}, u_1) = \overrightarrow{p}$ and $u = u_1$, then $\alpha\mathcal{F}_1\overrightarrow{p}$;

(d) if $\delta^*(\overrightarrow{q_1}, u_1u_2) = \overrightarrow{p}$ then $\alpha\mathcal{F}_1\#\mathcal{F}_2\overrightarrow{p}\cdots$.

9.6.3.2 Building the run in $u_3u_4\cdots$. Once maxi-run$(\overrightarrow{q_1}, u_1u_2)$ is treated, we are ready to build the runs ρ_i from Lemma 9.48, for $i \geqslant 3$. The steps described below roughly follow the steps from Section 9.6.3.1, but since we have an unbounded (possibly infinite) number of runs ρ_i, we cannot build them sequentially. The key idea is to build them in parallel, relying on the fact that they all start in p. We shall only focus on the case of Item (5)(d). One can show that other cases Items (5)(a) to (5)(c) are not be modified when applying the functions described below.

(6) We first apply a sequential function which checks if there is a $\overrightarrow{q}\#$ with $q \in Q$ in the input, and in this case replaces each subsequent symbol $\#$ by $\#\overrightarrow{q}$. After this operation, the output of Item (5)(d) becomes $\alpha\mathcal{F}_1\#\mathcal{F}_2\#\overrightarrow{p}\mathcal{F}_3\#\overrightarrow{p}\mathcal{F}_4\cdots$, i.e. it describes the first state of each ρ_i for $i \geqslant 3$;

(7) Then we apply $\overrightarrow{\text{try-simul}}\#$ on the whole input. The output is now $\alpha\mathcal{F}_1\#\mathcal{F}_2\#w_3\#w_4\cdots$ where w_i has of the following shapes for all $i \geqslant 3$ (where $\beta_i := \lambda^*(\overrightarrow{p}, u_i)$):
   (a) either $\delta^*(\overrightarrow{p}, u_i) = \omega$ and $\beta_i \in B^\omega$, then β_i;
   (b) or $\delta^*(\overrightarrow{p}, u_i) = \overrightarrow{p}$ (thus ρ_i never visits the last position of u_{i-1}), then $\beta_i\mathcal{F}_i\overrightarrow{p}$;
   (c) or $\delta^*(\overrightarrow{p}, u_i) = \overleftarrow{p_i}$ (thus ρ_i visits the last position of u_{i-1}), then $\beta_i\overleftarrow{p_i}\mathcal{F}_i$.

   By Lemma 9.48, Item (7)(b) describes the beginning of ρ_{i+1}, hence the state is necessarily p.

(8) Then we replace each subword $\overrightarrow{q}\#$ with $q \in Q$ by $\#$. This enables to remove the $\overrightarrow{p}\#$ in the w_i of Item (7)(b), since the corresponding ρ_i have been fully simulated.

(9) Now, our goal is to deal with the w_i of Item (7)(c), which correspond to the runs ρ_i which are not fully simulated. In this case, we need to compute $\overleftarrow{\text{simul-d}}(\mathcal{F}_{i-1}\overleftarrow{p_i})$. We first build a function which implements an operation similar to that of Item (3).

Claim 9.56 (Behind)

Let \$ be a fresh symbol. One can build a function behind in $\mathfrak{C}$ which takes as input a word $w_1\#w_2\#\cdots$ where each factor w_i has shape either $\beta_i\overleftarrow{r_i}\mathcal{F}_i$, or $\beta_i\mathcal{F}_i$, or $\beta_i\mathcal{F}_i\overrightarrow{r_i}$ or β_i. It outputs a word where each factor w_i of shape $\beta_i\overleftarrow{r_i}\mathcal{F}_i$ for $i \geqslant 2$ is replaced by $\beta_i r_i\$\mathcal{F}_{i-1}\overleftarrow{r_i}\$\mathcal{F}_i$, and the other w_i are unchanged.

Proof. We can first apply a sequential function which adds a \$ symbol just before reach $\mathcal{F}$. Then we apply basic-copy with separator \$, giving factors of shape:

$$\$\mathcal{F}_{i-1}(\overrightarrow{r_{i-1}}?)\#\beta_i(\overleftarrow{r_i}?)\$\mathcal{F}_{i-1}\#\beta_i(\overleftarrow{r_i}?)\$.$$

where $?$ denotes the possibility of having or not a letter. Then we apply a sequential function which uses the first $\mathcal{F}_{i-1}(\overrightarrow{r_{i-1}}?)$ to complete the $(i-1)$-th factor, then outputs β_i, then $r_i\$\mathcal{F}_{i-1}$ if there is a $\overleftarrow{r_i}$, ignores the next β_i and ends with $\overleftarrow{r_i}$. It is easy to see that this function behaves as expected if β_i or $\mathcal{F}_{i-1}$ is infinite. ◀

We thus apply the function behind from Claim 9.56 to the whole input.

(10) It remains to apply $\overleftarrow{\text{simul-d}}(\mathcal{F}_{i-1}\overleftarrow{p_i})$ on the appropriate factors, as we did in Item (4). For this, we apply the function $\overleftarrow{\text{try-simul}}\#$. After this operation, the factors without \$ are not modified, and the factors of shape $\beta_i p_i\$\mathcal{F}_{i-1}\overleftarrow{p_i}\$\mathcal{F}_i$ are transformed in (with $\beta_i' := \lambda^*(\overleftarrow{p_i}, u_{i-1})$):
   (a) if $\delta^*(\overleftarrow{p_i}, u_{i-1}) = \overrightarrow{p_i'}$, then $\beta_i p_i\$\beta_i'\mathcal{F}_{i-1}\overrightarrow{p_i'}\$\mathcal{F}_i$;
   (b) if $\delta^*(\overleftarrow{p_i}, u_{i-1}) = \overleftarrow{p_i'}$, then $\beta_i p_i\$\beta_i'\overleftarrow{p_i'}\mathcal{F}_{i-1}\$\mathcal{F}_i$. In this case, it means that ρ_i will visit u_{i-2}. According to Lemma 9.48, this is only possible if u_i is the last factor. Here the forward factorization forest cannot help us to control the end of ρ_i, but this very particular case can occur only once in the whole process and will be treated in Item (13).

(11) Now, let us remove the \$ and the useless copies of factors.

Claim 9.57 (Cleaning)

One can build in $\mathfrak{C}$ a function clean which behaves as follows:

- if its input does not contain $\$$, it is not modified;
- if its input has shape $\beta_i p_i \$ \beta'_i \mathcal{F}_{i-1} \overrightarrow{p'_i} \$ \mathcal{F}_i$, the output is $\beta_i \beta'_i \overrightarrow{p'_i} \mathcal{F}_i$;
- if its input has shape $\beta_i p_i \$ \beta'_i \overleftarrow{p'_i} \mathcal{F}_{i-1} \$ \mathcal{F}_i$, it is mapped to $\beta_i p_i \mathcal{F}_i$.

Proof. We first replace the second $\$$ (if it exists) by a $\&$ and then apply a basic-copy with separator $\&$. We finally apply a sequential function which outputs what it sees until a factor $q\$$ with $q \in Q$. In this case it reads the next factor between $\$$ and $\&$ (without writing) to determine whether the input has the second or the third shape, and then it behaves accordingly on the last piece. This can be done without modifying the words without $\$$. ◀

We then apply the function clean$\#$ to our whole input. Observe that the last case of Claim 9.57, we have undone the computation of $\overleftarrow{\text{simul-d}}(\mathcal{F}_{i-1} \overleftarrow{p_i})$ and the state p_i no longer has an overarrow. We say that this state is *frozen*[7] so that it does not interfere with the remaining parallel computations of the ρ_i. Recall from Item (10)(b) that in this case p_i marks the beginning of the last factor, which is a very rich information in a situation where ω-lookarounds are not permitted.

(12) By iterating $|Q|$ times the previous steps, and then applying functions of $\mathfrak{C}$ to clean the output, one can ensure that the result is one of the following:
   (a) $\beta \# \mathcal{F}_1 \cdots \# \mathcal{F}_n \overrightarrow{r}$ and in this case $\delta^*(\overrightarrow{q_1}, u) = \overrightarrow{r}$ and $\beta = \lambda^*(\overrightarrow{q_1}, u)$;
   (b) $\beta \in B^\omega$ and in this case $\beta = \lambda^*(\overrightarrow{q_1}, u)$;
   (c) $\beta \# \mathcal{F}_1 \cdots \# r \mathcal{F}_n$ where r was frozen during the computation and β is the output along maxi-run$(\overrightarrow{q_1}, u)$ until it visits r in the first position of u_n.

(13) Finally, let us explain briefly how to deal with Item (12)(c). The key argument is that the last factor is now marked. We first transform the $\# \overleftarrow{r}$ into $\overleftarrow{r}\$$. By applying successively $\overleftarrow{\text{simul-(d+1)}}$ on $\mathcal{F}_1 \# \cdots \# \mathcal{F}_{i-1}$ and $\overrightarrow{\text{simul-d}}$ on $\mathcal{F}_i$, as we did in Section 9.6.3.1 for a concatenation[8] of two forward factorization forests, one can build the end of the run maxi-run$(\overrightarrow{q_1}, u)$.

9.6.4 Decomposing deterministic regular functions

Thanks to the results presented in Sections 9.6.1 to 9.6.3, now we are ready to conclude the proof of Theorem 9.43. Let $\mathscr{T} = (A, B, Q, q_0, F, \delta, \lambda)$ be a $2\mathsf{DT}^\omega$ whose transition morphism is $\mu : A^* \to \mathbb{T}$, which computes a deterministic regular function $f : A^\omega \rightharpoonup B^\omega$. It follows from Lemma 9.54 that one can build a function f' from the class $\mathfrak{C}$ such that $f'(\overrightarrow{q_0}\mathcal{F}) = f(u)$ whenever $\mathcal{F} \in \text{f-Forests}_\mu^{|\mathbb{T}|}(u)$ for some $u \in \text{Dom}(f)$. Thus the composition $f' \circ \text{f-forest}_\mu$ is an extension of f. Thanks to Theorem 9.46 and the definition of $\mathfrak{C}$, $f' \circ \text{f-forest}_\mu$ is a composition of sequential and map-copy-reverse$^\omega$ functions. To remove the words which are not in the domain (which is Büchi deterministic by Proposition 9.14), it suffices to pre-compose this function by an appropriate sequential function.

9.7 Discussion: pebbles and marbles of infinite words

The reader may ask why *(deterministic) polyregular functions of infinite words* have never been defined in the literature. A first answer is that nested $2\mathsf{DT}^\omega$ (or nested infinite “for” loops) are less meaningful than

[7] But in Item (13) we shall “let it got, let it go”.

[8] Beware that here we really need to use $\overleftarrow{\text{simul-(d+1)}}$, and that $\overleftarrow{\text{simul-d}}$ would not suffice. This is why $\overleftarrow{\text{simul-(d+1)}}$ was previously created in Lemma 9.53, and not built in our induction.

nested 2DT of finite words. Indeed, in order to produce an infinite word (and not word indexed by a more complex ordinal[9]), one has to ensure that any *submachine* only produces a finite output.

Therefore it seems more natural to define *(recursive) marble transducers of infinite words*, whose *submachines* process a finite prefix of the input, thus necessarily produce a finite output. Furthermore, one can conjecture that this model is equivalent to DSST^ω without restrictions on the register copies, which is still meaningful over infinite words, since it processes its input in a streaming fashion.

Over finite words, we have seen in Theorem 4.41 how to decide if a DSST can be transformed into an equivalent k-layered DSST. Recall that for $k = 1$, it shows how to decide if a function computed by a DSST is regular. Now, we discuss this problem over infinite words in Open question 9.58.

Open question 9.58 ($\mathsf{DSST}^\omega \to$ Deterministic regular)

Given a DSST^ω, can we decide if it computes a deterministic regular function?

First, we note that comparing the size of the input and the output no longer makes sense over infinite words. Furthermore, making exponential copies of shape $\mathfrak{r} \mapsto \mathfrak{r}\mathfrak{r}$ no longer prevents from being deterministic regular (for instance, if this register is used to produce a unary output of shape 1^ω).

Let us try to build alternative insights on this problem, by relying on a possible semantics characterization. Recall that an infinite word is *ultimately periodic* if it has shape uv^ω for some $u, v \in A^+$. By adapting the techniques of Section 2.2.2, it is easy to show that if f is deterministic regular, then for all ultimately periodic $u \in \mathsf{Dom}(f)$, $f(u)$ is ultimately periodic. This condition is however not sufficient to characterize the functions computed by DSST^ω which are deterministic regular.

Example 9.59 (Ultimately periodic output)

The function which maps 0^n1^ω to $0^{n^2}1^\omega$ is computable by a DSST^ω. Furthermore, the output over any word is ultimately periodic, but this function is not deterministic regular.

To avoid this issue, one can formulate the following candidate for a characterization: a function f computed by a DSST^ω is deterministic regular if and only if there exists $K \geqslant 0$ such that for all $uv^\omega \in \mathsf{Dom}(f)$, $f(uv^\omega) = \alpha\beta^\omega$ for some $|\alpha| \leqslant K|u|$ and $|\beta| \leqslant K|v|$. This condition somehow introduces uniformity in the ultimate periodicity. It is necessary, but we do not know if it is sufficient.

In the rest of Section 9.7, we discuss the case when the input alphabet is unary, i.e. it is $\{0\}$. This restriction may seem completely dumb, since the domain is now the singleton $\{0^\omega\}$. We shall observe that it is not the case, since a large variety of sequences can be obtained by iterating a substitution.

Example 9.60 (Linear blocks)

Let $f\colon 0^\omega \to \{0,1\}^\omega$ be such that $f(0^\omega) = 10100100010\cdots$. This function can be computed by a copyful DSST^ω with a single state, registers $\{\mathfrak{r}, \mathsf{out}\}$ and updates $\mathfrak{r} \mapsto \mathfrak{r}\,0, \mathsf{out} \mapsto \mathsf{out}\,\mathfrak{r}\,1$.

It is easy to see that in this particular setting, f is deterministic regular if and only if $f(0^\omega)$ is ultimately periodic. Let us observe that our question is related to a well-known word combinatorics problem. A *morphic word* is an infinite word given by a tuple (B, C, c, φ, ψ) such that B and C are alphabets, $\psi\colon C^* \to C^*$ and $\varphi\colon C^* \to B^*$ are morphisms and $c \in C$ is such that $\psi(c) = c\,u$ for some $u \in C^+$. Thanks to this last condition, $\psi^n(c) := \psi \circ \cdots \circ \psi(c)$ (with n compositions) converges to some $\psi^\omega(c) \in C^\omega$. The *morphic word* is formally $\varphi(\psi^\omega(c)) \in B^\omega$ when this value is infinite.

[9]In this case, one could imagine that a *k-pebble transducer* outputs a word indexed by the ordinal ω^k.

Proposition 9.61 (Unary input alphabet)

The following problems are effectively equivalent:

(1) given a DSST^ω computing $f\colon 0^\omega \mapsto u$, deciding whether f is deterministic regular;
(2) given a DSST^ω computing $f\colon 0^\omega \mapsto u$, deciding whether u is ultimately periodic;
(3) given a morphic word $u \in A^\omega$, deciding whether u is ultimately periodic.

Proof. We treat equivalence between Items (2) and (3). This result follows by observing that a DSST^ω with input alphabet $\{0\}$ can always be transformed in a simple one (see Section 4.4.1). A simple DSST^ω is $(B, \mathfrak{R}, \mathsf{out}, \iota, \lambda)$ where $\lambda\colon \mathfrak{R} \to \mathfrak{R}^*$ and $\iota\colon \mathfrak{R} \to B^*$. Observe that the semantics of such a machine exactly matches with the definition of a morphic word. ◀

Thanks to Theorem 9.62 from [Dur13], our problem becomes decidable over unary input alphabets.

Theorem 9.62 (Ultimate periodicity of morphic words)

One can decide if a morphic word is ultimately periodic.

On the negative side, Proposition 9.61 means that deciding deterministic regularity over arbitrary input alphabets is at least as technical as showing Theorem 9.62, and probably much more. However, Theorem 9.62 is already known to be a difficult result in word combinatorics, which had been open for at least 30 years before Durand's proof. All in all, we believe that Open question 9.58 is quite difficult, and that it cannot be solved by using the techniques of this manuscript.

Chapter 10

Determinization of continuous rational functions

L'idée de l'avenir, grosse d'une infinité de possibles, est donc plus féconde que l'avenir lui-même, et c'est pourquoi l'on trouve plus de charme à l'espérance qu'à la possession, au rêve qu'à la réalité.

Henri Bergson, *Essai sur les données immédiates de la conscience*

We have conjectured in Chapter 8 that the class of continuous (or, equivalently, computable) regular functions is (up to extensions) exactly the class of deterministic regular functions. The various results of Chapter 9 tend to support this conjecture, since they show that the class of deterministic regular functions is especially robust. In particular, it is closed under composition, and so is the class of continuous regular functions, since continuity is preserved under composition.

The goal of the current chapter is to partially solve the aforementioned conjecture by showing that a continuous rational function can effectively be extended to a deterministic regular one. This main result is stated in Section 10.1. Given a rational function, it enables to build a deterministic machine with bounded memory which computes it whenever it is possible (recall that continuity is known to be decidable). As such, it can be seen as a way to synthesize a simple program from a specification.

The proof of this statement is rather complex and goes over Sections 10.2 to 10.7. A key obstacle is that a deterministic machine cannot choose which run of a $\mathsf{1NT}^\omega$ is accepting, since final Büchi conditions deal with events happening infinitely often. Therefore, a $\mathsf{1DT}^\omega$ which simulates this $\mathsf{1NT}^\omega$ has to manipulate several runs in parallel. This intuition motivates our key definition of *compatible sets of states* which are the sets of states of a $\mathsf{1NT}^\omega$ having a "common infinite future". We show that when the function f computed by the $\mathsf{1NT}^\omega$ is continuous, the outputs produced along finite runs which end in a compatible set enjoy handy combinatorial properties. Finally, we leverage these properties in order to build a deterministic regular extension of f. To the knowledge of the author, the techniques used in this proof are completely original. This result probably is the most involved of this manuscript.

We believe that the path is still long towards generalizing this result to regular functions. In Section 10.8, we nevertheless conjecture that the proof can be adapted to study *uniform continuity* of rational functions, and to capture this subclass by a dedicated computation model.

This chapter is mainly based on [CD22].

10.1 Continuity of rational functions

In Section 10.1.1, we claim that a continuous rational function can be extended to a deterministic regular one. We also argue that this result is tight, in the sense that even $1\mathsf{DT}^\omega$ with finite lookarounds are not powerful enough to capture continuous rational functions. We then recall in Section 10.1.2 a well-known decidable characterization of continuity for the functions computed by $1\mathsf{NT}^\omega$.

10.1.1 Two-way determinization of continuous rational functions

We provide in Theorem 10.1 a partial answer to Conjecture 8.46. This result is the main statement of Chapter 10. It was first stated in [CD22, Theorem 4.2]. Its proof is especially long and involved: it nearly requires a whole chapter and ranges over Sections 10.2 to 10.7.

Theorem 10.1 (Rational → Deterministic regular extension)

A rational function of infinite words has an extension which is deterministic regular if and only if it is continuous. If this property holds, one can build a $2\mathsf{DT}^\omega$ which computes an extension.

Proof sketch. The "only if" direction is obvious. Conversely, let $\mathscr{T}$ be a real-time unambiguous $1\mathsf{NT}^\omega$ which computes a continuous function $f : A^\omega \rightharpoonup B^\omega$. The main obstacle for giving a deterministic regular extension of f is that one cannot compute the accepting run of $\mathscr{T}$ in a deterministic fashion. Indeed, there may exist several infinite runs labelled by the same input, and the accepting one can only be detected by using Büchi conditions.

An extension of f is built as the composition (recall from Theorem 9.39 that deterministic regular functions are closed under composition) of three deterministic regular functions buildSteps (Theorem 10.22), buildTrees (Theorem 10.26) and buildOutput (Theorem 10.28) where:

(1) the function buildSteps first computes an over-approximation of the accepting run of $\mathscr{T}$ in terms of subsets of Q which are called *compatible*. Intuitively, this construction captures all possible infinite runs labelled by the input and removes irrelevant finite runs;
(2) by leveraging the continuity hypothesis one can show that the runs which visit compatible sets enjoy several combinatorial properties. The function buildTrees uses these properties to build a sequence of trees (encoded as words) which describe the output of f. In these trees, branching behaviors (which correspond to the remaining non-determinism) are only allowed when the outputs commute. Hence this construction is another step towards determinism;
(3) finally, the function buildOutput removes the branching behaviors of the trees previously built. It is obtained by describing a DSST^ω which manipulates its registers in tree-like fashion. ◀

Example 10.2 (Doubling factors)

The total function double: $\{0,1,2\}^\omega \to \{0,1,2\}^\omega$ from Example 8.16 can be computed by a $2\mathsf{DT}^\omega$ which does a first left-to-right pass on each block 0^{n_i} (or 0^ω) while outputting 0^{n_i} (or 0^ω). If it reads a 1, it outputs it and moves to the next block. If it reads a 2, it does a right-to-left pass on 0^{n_i}, and then a last left-to-right pass while outputting 0^{n_i} again.

As a low-hanging consequence, one can decide in Corollary 10.3 if a rational function is deterministic regular. As observed for Corollary 8.24, obtaining such a corollary is just a matter of domains.

Corollary 10.3 (Rational $\to$ deterministic regular)

One can decide if a rational function of infinite words is deterministic regular. If this property holds, one can build a $2\mathsf{DT}^\omega$ which computes it.

Proof. Observe that a rational function f is (effectively) deterministic regular if and only if it can be extended to a deterministic regular function and its domain is Büchi deterministic. Indeed, a deterministic regular function can be restricted to any Büchi deterministic language by Proposition 9.14. We conclude thanks to Proposition 8.3 and Theorem 10.1. ◀

Recall that $1\mathsf{DT}^\omega$ are not sufficient to capture continuous rational functions, since e.g. the functions replace or double are not sequential. In Section 9.1.1, we introduced the model of $1\mathsf{DT}^\omega$ with finite lookarounds, which lies in between $2\mathsf{DT}^\omega$ and $1\mathsf{DT}^\omega$, since it has the ability to check a property of a finite prefix of the input. We say that a function is *deterministic rational* if it can be computed by such a machine. It is easy to show that deterministic rational functions are closed under composition.

A natural question[1] is whether continuous rational functions are deterministic rational. We show in Proposition 10.4 that it is not the case[2]. This result means that our Theorem 10.1 is tight, i.e. that two-way moves are absolutely unavoidable when determinizing rational functions.

Proposition 10.4 ($1\mathsf{DT}^\omega$ with finite lookarounds are not sufficient)

The total function $\mathsf{double}\colon \{0,1,2\}^\omega \to \{0,1,2\}^\omega$ is not deterministic rational.

Proof. Assume that the function double is computed by a $1\mathsf{DT}^\omega$ with finite lookarounds of shape $\mathscr{T} = (A, B, Q, q_0, F, \delta, \lambda)$ where $\delta\colon Q \times \mathsf{RegLang}(A) \to Q$. Let $q_0 \to q_1 \to \cdots$ be the accepting run of $\mathscr{T}$ labelled by 0^ω. Let $L_0, L_1, \dots$ be such that $\delta(q_i, L_i) = q_{i+1}$ for $i \geqslant 0$. There exists $N \geqslant 0$ such that 0^N has a prefix in L_i for all $i \geqslant 0$. Thus if $p_{0,n} \to p_{1,n} \to \cdots$ (resp. $r_0 \to r_1^n \to \cdots$) is the accepting run labelled by $0^{n+N}10^\omega$ (resp. $0^{n+N}20^\omega$), then $p_i^n = r_i^n = q_i$ for all $0 \leqslant i \leqslant n$. Hence there exist $K \geqslant 0$, $\alpha, \beta, \gamma, \gamma', \delta, \delta' \in \{0,1,2\}^+$ such that $\mathsf{double}(0^{Kn+N}10^\omega) = \alpha\beta^n\gamma\delta^\omega$ and $\mathsf{double}(0^{Kn+N}20^\omega) = \alpha\beta^n\gamma'\delta'^\omega$, a contradiction. ◀

Nevertheless, the author believes that deterministic rational functions are a robust class of functions which is worth being studied in detail and characterized among the deterministic regular ones.

Conjecture 10.5 (Rational $\to$ Deterministic rational)

One can decide if a rational function of infinite words is deterministic rational.

10.1.2 Continuity and twinning property

The goal of Section 10.1.2 is to recall the well-known characterization of continuity for rational functions in terms of twinning properties for $1\mathsf{NT}^\omega$ (Lemma 10.8). This result enables to normalize a $1\mathsf{NT}^\omega$ computing a continuous function, which is a first easy step towards Theorem 10.1.

[1] The author is grateful to Lhote for asking this question.

[2] Intuitively, a $1\mathsf{DT}^\omega$ with finite lookarounds computing the function double may have to check that the current suffix is 0^ω, which is not possible since this is not a property of a finite word.

Let $\mathscr{T} = (A, B, Q, q_0, \Delta, \lambda)$ be a 1NT$^\omega$. If $u \in A^*$ and $p, q \in Q$, we write $p \xrightarrow{u|\alpha} q$ to denote the existence of a run from p to q labelled by u which outputs $\alpha \in B^*$. If $u \in A^\omega$, we write $p \xrightarrow{u|\alpha} \infty$ for the existence an infinite (but not necessarily final) run labelled by u which outputs $\alpha \in B^\infty$.

Definition 10.6 (Trim, clean)

We say that a 1NT$^\omega$ is *trim* if any state occurs in some accepting run[3]. The 1NT$^\omega$ is said to be *clean* whenever the production along any accepting run is infinite.

Observe that a trim 1NT$^\omega$ is clean if and only if it has no run of shape $q \xrightarrow{u|\varepsilon} q$ for some final state q and $u \in A^+$. Recall from Section 8.1.2 that rational functions are computed by unambiguous and real-time 1NT$^\omega$. It is easy to show that such a machine can always be made trim and clean.

Claim 10.7 (Trim and clean 1NT$^\omega$)

Given an unambiguous and real-time 1NT$^\omega$, one can build an unambiguous, real-time, clean and trim 1NT$^\omega$ which computes the same function.

Proof. Given an unambiguous and real-time 1NT$^\omega$ $\mathscr{T}$, one can build an equivalent unambiguous, real-time and clean 1NT$^\omega$. The latter consists of two disjoint copies of $\mathscr{T}$, where accepting states are only taken in the second copy, which is visited only when producing a non-empty output. This way, we ensure the absence of loops with empty output in a final state. Finally, we trim this machine. ◀

Now we are ready to recall the characterization of continuous functions in terms of run patterns for 1NT$^\omega$ (also called *twinning properties*). These patterns are presented in Figure 10.9 and Lemma 10.8. In this manuscript, we only need the "only if" direction of Lemma 10.8, whose proof is easy using continuity. The converse direction (shown e.g. in [DFKL20, Lemma 11]) was used to give tight complexity bounds for deciding the continuity of a function computed by a 1NT$^\omega$ [DFKL20, Theorem 12].

Lemma 10.8 (Characterization of continuity)

Let $\mathscr{T} = (A, B, Q, I, F, \Delta, \lambda)$ be an unambiguous, real-time, clean and trim 1NT$^\omega$which computes a function $f : A^\omega \rightharpoonup B^\omega$. Then f is continuous if and only if the following holds.

For all $q_1, q_2 \in I$, $q_1' \in F$, $q_2' \in Q$, $u \in A^*$, $u' \in A^+$, $\alpha_1, \alpha_1', \alpha_2, \alpha_2' \in B^*$ such that $q_i \xrightarrow{u|\alpha_i} q_i' \xrightarrow{u'|\alpha_i'} q_i'$ for $i \in \{1, 2\}$, we have (recall that $\alpha_1' \neq \varepsilon$ since $\mathscr{T}$ is clean):

- ▶ if $\alpha_2' \neq \varepsilon$, then $\alpha_1 {\alpha_1'}^\omega = \alpha_2 {\alpha_2'}^\omega$;
- ▶ if $\alpha_2' = \varepsilon$, then for all $v \in A^\omega, \beta \in B^\omega$ such that $q_2' \xrightarrow{v|\beta} \infty$ is final, $\alpha_1 {\alpha_1'}^\omega = \alpha_2 \beta$.

Proof of "only if". Let $v \in A^\omega$ and $\beta \in B^\omega$ be such that $q_2' \xrightarrow{v|\beta}$ is final (such a run exists since the transducer is trim and clean). Therefore, for all $n \geqslant 0$ we have $f(u{u'}^n v) = \alpha_2 {\alpha_2'}^n \beta$. On the other hand $f(u{u'}^\omega) = \alpha_1 {\alpha'}_1^\omega$ because $q_1 \in F$. By continuity in $u{u'}^\omega \in \mathrm{Dom}(f)$, for all $p \geqslant 0$ we have $|f(u{u'}^n v) \wedge f(u{u'}^\omega)| \geqslant p$ for n large enough. The result directly follows. ◀

In particular, if $\alpha_2' = \varepsilon$, then for any final run $q_2' \xrightarrow{v|\beta} \infty$ we have $\beta = \alpha_2^{-1} \alpha_1 {\alpha_1'}^\omega$, i.e. the output along this run does not depend on v. Using this observation, we show in Claim 10.11 how to ensure that the case $\alpha_2' = \varepsilon$ never occurs. Avoiding such loops with empty output will be useful in Section 10.5.3, in order to ensure that all infinite runs (even the non-accepting ones) produce an infinite outputs.

[3] Equivalently, there exist both a (finite) initial run which ends in this state and an (infinite) final which starts in it.

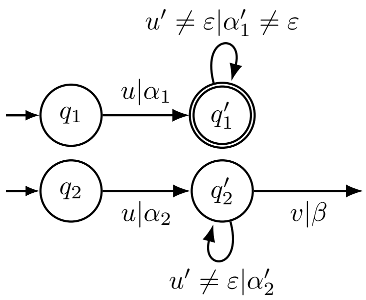


Figure 10.9: Twinning property described in Lemma 10.8.

Definition 10.10 (Parallel productivity)

We say that a $1\mathsf{NT}^\omega$ is *parallel productive* if the hypotheses of Lemma 10.8 imply $\alpha'_2 \neq \varepsilon$.

Claim 10.11 (Parallel productive $1\mathsf{NT}^\omega$)

Given an unambiguous, real-time, trim and clean $1\mathsf{NT}^\omega$ computing a continuous function, one can build an unambiguous, real-time, trim, clean and parallel productive $1\mathsf{NT}^\omega$ computing it.

Proof idea. Let $\mathscr{T} := (A, B, Q, I, F, \Delta, \lambda)$ be such a $1\mathsf{NT}^\omega$ computing a continuous function. We say that $q'_2 \in Q \smallsetminus F$ is *constant* if the conditions of Lemma 10.8 hold and $\alpha'_2 = \varepsilon$. One can decide if a state $q \in Q$ is constant (by using a pumping argument, one can enforce $|u|, |u'| \leqslant |Q|^{|Q|}$ in Lemma 10.8) and in this case one can effectively compute $\alpha_q \in B^*, \alpha'_q \in B^+$ such that for all final run $q \xrightarrow{v|\beta} \infty$, $\beta = \alpha_q {\alpha'_q}^\omega$ (as observed right before Definition 10.10). For all such constant state $q \in Q$, one can build an unambiguous, clean and parallel productive $1\mathsf{NT}^\omega$ $\mathscr{T}_q$ which computes the constant partial function $v \mapsto \alpha_q \alpha'_q$, with domain $\{v \mid v$ labels a final run of $\mathscr{T}$ starting in $q\}$. Finally, we replace each constant state q of $\mathscr{T}$ by a disjoint copy of $\mathscr{T}_q$ (i.e. we remove q and send all ingoing transitions to the initial states of $\mathscr{T}_q$). We finally trim this machine. ◀

We say that a $1\mathsf{NT}^\omega$ is *productive* if for all $q \in Q$ and $u \in A^+$, if $q \xrightarrow{u|\alpha} q$ then $\alpha \neq \varepsilon$. As observed in Claim 10.12, it is not possible to ensure that a $1\mathsf{NT}^\omega$ computing a continuous function is productive. Therefore Claim 10.11 is the best simplification we can get. We conjecture in Section 10.8 that productivity can be reached in the setting of uniformly continuous rational functions.

Claim 10.12 (Non-productivity)

The sequential function $\mathsf{remove}\colon \{a, b, c\}^\omega \rightharpoonup \{b, c\}^\omega$ is not computable by a productive $1\mathsf{NT}^\omega$.

Proof idea. Assume that remove is computed by a productive $1\mathsf{NT}^\omega$. By relying on classical pumping arguments, one can show the existence of $n \geqslant 1$, $\alpha \in B^+$, $\beta, \gamma \in B^\omega$, such that $b^\omega = f(a^n b^\omega) = \alpha\beta$ and $c^\omega = f(a^n c^\omega) = \alpha\gamma$. This yields a contradiction. ◀

10.2 Overall description of the determinization process

In the rest of Chapter 10, $\mathscr{T} = (A, B, Q, I, F, \Delta, \lambda)$ denotes an unambiguous, real-time, clean, trim, and parallel productive $1\mathsf{NT}^\omega$ which computes a continuous function $f\colon A^\omega \rightharpoonup B^\omega$. The goal of Section 10.2 is to describe the main steps of the construction of a deterministic regular function which extends f, which proves Theorem 10.1. The structure of proof presented in this manuscript substantially differs from the original proof of [CD22], even if the underlying ideas are the same. The author believes that the current presentation is more modular and easier to follow[4].

Formally, we shall build three deterministic regular functions buildSteps (Theorem 10.22), buildTrees (Theorem 10.26), buildOutput (Theorem 10.28) such that buildOutput∘buildTrees∘buildSteps : $A^\omega \rightharpoonup B^\omega$ is an extension of the function f computed by $\mathscr{T}$. This function is deterministic regular as a composition of deterministic regular functions (Theorem 9.39), which concludes the proof of Theorem 10.1. The three aforementioned functions intend to capture distinct difficulties of the construction.

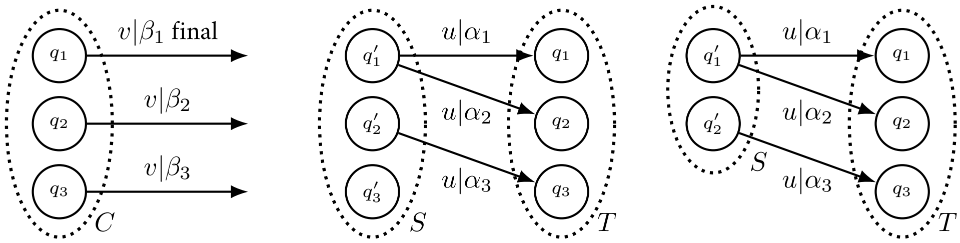
10.2.1 Computing compatible sets

The goal of this section is to state Theorem 10.22, which builds the function buildSteps for computing an over-approximation of the accepting run of $\mathscr{T}$ in terms of *compatible sets*. Informally, they are sets of states from Q which have a "common infinite future" and such that one of the future runs is final, as depicted in Figure 10.14a. In terms of computability, they capture a form of non-determinism which cannot be solved by a deterministic machine, even when finite lookarounds are allowed.

Definition 10.13 (Compatible set)

We say that a subset $C \subseteq Q$ is *compatible* whenever there exists $v \in A^\omega$ such that for all $q \in C$, there exists an infinite run ρ_q labelled by v for which the following holds:

- for all $q \in C$, ρ_q starts in state q;
- there exists $q \in C$ such that ρ_q is final.



(a) A compatible set C. (b) A pre-step S, u, T. (c) A step S, u, T.

Figure 10.14: Compatible sets, pre-steps and steps.

Observe that singletons are always compatible sets since the $1\mathsf{NT}^\omega$ is trim. However, a subset of a compatible set has no reason to be compatible itself (we may have lost the accepting run).

Example 10.15 (Compatible sets)

In the $1\mathsf{NT}^\omega$ of Figure 8.17b, the compatible sets are the singletons, $\{q_0, q_1\}$ and $\{q_0, q_2\}$. However $\{q_1, q_2\}$ is not compatible. In Figure 8.17c, all pairs of states are compatible.

We denote by Comp (resp. Comp(S)) the set of compatible sets of states of $\mathscr{T}$ (resp. the set of compatible sets of which are included in a given $S \subseteq Q$). By using the pigeonhole principle, one can easily

[4]Due to the fact that neither closure under composition of deterministic regular functions (Theorem 9.39) nor finite lookarounds removal (Theorem 9.4) for $2\mathsf{DT}^\omega$ were known at the publication of [CD22], the original proof is not divided in three distinct steps like the current one. Hence its various constructions are entangled and less easy to understand.

characterize compatible sets in terms of loops, as detailed in Claim 10.16. As a direct consequence, one can effectively determine if some $C \subseteq Q$ is compatible or not.

Claim 10.16 (Characterization of compatible sets)

The set $C \subseteq Q$ is compatible if and only if there exists a function $d : C \to Q$, and words $u, u' \in A^*$ such that the following holds:

- for all $q \in C$, $q \xrightarrow{u} d(q) \xrightarrow{u'} d(q)$;
- there exists $q \in C$ such that $d(q) \in F$,
- $u' \neq \varepsilon$ and $|u|, |u'| \leqslant |Q|^{|Q|}$.

Now let us introduce the notions of *pre-step* and *step*, which describe how one can move from a compatible set to another by reading letters. This intuition is depicted in Figures 10.14b and 10.14c.

Definition 10.17 (Pre-step, step)

Given $C, D \in \mathsf{Comp}$, we say that C, u, D is a *pre-step* if $u \in A^*$ and for all $q \in D$, there is a unique state of C denoted $\mathsf{pred}^u_{C,D}(q)$, such that $\mathsf{pred}^u_{C,D}(q) \xrightarrow{u} q$.

We say that C, u, D is a *step* if it is a pre-step and the function $\mathsf{pred}^u_{C,D} : C \to D$ is surjective.

Given $q \in D$, let $\mathsf{prod}^u_{C,D}(q)$ be the output $\alpha \in B^*$ produced along the run $\mathsf{pred}^u_{C,D}(q) \xrightarrow{u|\alpha} q$. We shall mainly be interested in pre-steps or steps of shape J, u, C where $J \subseteq I$ and $J, C \in \mathsf{Comp}$, which are called *initial*. Indeed, they describe the execution of several initial runs of $\mathscr{T}$.

Example 10.18 (Pre-steps, steps)

In Figure 8.17b, the initial steps are $\{q_0\}, u, \{q_i\}$ for some $i \in \{0, 1, 2\}$. In Figure 8.17c, observe that $\{q_0\}, 0^n, \{q_1, q_2\}$ is also a step for all $n \geqslant 0$.

We observe in Claim 10.19 that initial pre-steps naturally emerge in the proofs, due to the unambiguity of $\mathscr{T}$. If $u \in A^*$ and $S \subseteq Q$, we define $u \triangleright S := \{q \mid p \xrightarrow{u} q \text{ for some } p \in S\}$.

Claim 10.19 (Construction of pre-steps)

Let $u, v \in A^*$ and $C, D \in \mathsf{Comp}$ be such that $C \subseteq v \triangleright I$ and $D \subseteq u \triangleright C$, then C, u, D is a pre-step.

Proof idea. Uniqueness follows from the fact that $\mathscr{T}$ is trim and unambiguous. ◀

Now we justify in Lemma 10.20 why the study of compatible sets is especially relevant in our setting. This result originates from [CD22, Lemma 4.8] and shows that the initial runs labelled by some $u \in A^*$ which end in a compatible set produce the same output, up to taking prefixes. Formally, we say that words $u_1, \dots, u_n$ are *mutual prefixes* if for all $1 \leqslant i, j \leqslant n$, either $u_i \sqsubseteq u_j$ or $u_j \sqsubseteq u_i$ holds.

Lemma 10.20 (Mutual prefixes)

Let J, u, C be an initial pre-step, then the $\mathsf{prod}^u_{J,C}(q)$ for $q \in C$ are mutual prefixes.

Proof. We first show a stronger result that will be re-used in Section 10.4. Claim 10.21 provides an equation which is verified by the outputs of initial runs ending in a compatible set. The proof of this result crucially relies on the continuity of the function f computed by $\mathscr{T}$.

Claim 10.21 (Ends)

For all $C \in \mathsf{Comp}$, there exists a function $\mathsf{end}_C : C \to B^\omega$ such that for all[5] initial pre-step J, u, C and $p, q \in C$, we have $\mathsf{prod}^u_{J,C}(p)\mathsf{end}_C(p) = \mathsf{prod}^u_{J,C}(q)\mathsf{end}_C(q)$.

Proof. Since C is compatible, we get by Claim 10.16 words $v \in A^*, v' \in A^+$ and $d : C \to Q$ such that for all $q \in C$, $q \xrightarrow{v|\alpha(q)} d(q) \xrightarrow{v'|\alpha'(q)} d(q)$ with $\alpha(q), \alpha'(q) \in B^*$. Since $\mathscr{T}$ is clean and parallel productive and $d(p)$ is final for some $p \in C$, then $\alpha'(q) \neq \varepsilon$ for all $q \in C$. We define $\mathsf{end}_C(q) := \alpha(q)\alpha'(q)^\omega \in B^\omega$ and the result follows from Lemma 10.8. ◀

Lemma 10.20 directly follows from Claim 10.21. ◀

Now we are ready to state Theorem 10.22, which builds a deterministic regular function for computing an over-approximation of the accepting run of $\mathscr{T}$ in terms of compatible sets. In other words, this result only keeps initial runs whose outputs are prefixes of each other. It is therefore a first step towards computing f by a deterministic regular function since it removes several irrelevant behaviors. Theorem 10.22 originates from[6] [CD22, Lemma 4.16]. Its proof is detailed in Section 10.3.

Theorem 10.22 (Computing pre-steps)

One can build a deterministic regular function[7] $\mathsf{buildSteps} : A^\omega \rightharpoonup (A \uplus \mathsf{Comp})^\omega$ such that for all $u \in \mathsf{Dom}(f)$, $\mathsf{buildSteps}(u)$ is defined and has shape $S_0 u[1] S_1 u[2] S_2 \cdots$ where:

- $S_0 \subseteq I$ and for all $i \geqslant 0$, $S_i, u[i{+}1], S_{i+1}$ is a pre-step;
- for all $i \geqslant 0$, $q_i \in S_i$, where $q_0 \xrightarrow{u[1]} q_1 \xrightarrow{u[2]} \cdots$ is the accepting run of $\mathscr{T}$ labelled by u.

Proof sketch. If one performs a classical subset construction on $\mathscr{T}$, there is unfortunately no reason why the current set of states should be compatible. The main idea is to compute the function buildSteps by a $1\mathsf{DT}^\omega$ with finite lookarounds which performs an improved subset construction. At each stage, the machine uses the finite lookarounds to determine a subset of the current set of states which is compatible and contains the current state of the accepting run of $\mathscr{T}$. ◀

Remark 10.23 (Relation with the results of [FW21])

It is known since [FW21, Corollary 13] that one can build a $2\mathsf{DT}^\omega$ which computes f whenever $\mathscr{T}$ verifies a specific structural property called $(\mathscr{P})$. Formally, $(\mathscr{P})$ asks that for all $p, q, q' \in Q$ and $u \in A^*$, if $p \xrightarrow{u|\alpha} q$ and $p \xrightarrow{u|\alpha'} q'$ with $\alpha \sqsubseteq \alpha'$, then $q = q'$. Observe that if $(\mathscr{P})$ holds, then initial steps are necessarily of shape $\{p\}, u, \{q\}$ and thus S_i is a singleton for all $i \geqslant 0$. In this very restricted case, we immediately recover the result of [FW21] with Theorem 10.22: indeed, since the S_i are singletons, they describe a single run which is the accepting one (and it is trivial to produce its output). The main difficulties for proving Theorem 10.1 in general arise from the fact that the S_i may not be singletons. We cope with this obstacle in the next sections.

10.2.2 Computing trees

We have built in Theorem 10.22 an over-approximation $\mathsf{buildSteps}(u)$ of the accepting run of $\mathscr{T}$ labelled by $u \in \mathsf{Dom}(f)$ in terms of compatible sets. The runs described by $\mathsf{buildSteps}(u)$ still contains a form

[5] Observe that the function end_C does not depend on the initial pre-step chosen.
[6] The formulation of this original result is in fact (needlessly) more complex.
[7] We shall in fact build a deterministic rational function (but this precise statement is not useful in our proof).

of non-determinism, but the latter is restricted to the case when the outputs are mutual prefixes by Lemma 10.20. The goal of Section 10.2.2 is to show Theorem 10.28 which goes one step further: non-determinism is only allowed when all the outputs belong to θ^* for some $\theta \in B^*$ (i.e. they commute).

This construction is achieved by forgetting about the runs of $\mathscr{T}$ and building an intermediate model of tree sequences. We shall re-use the classical notions of *depth* (the root having depth 1), *ancestor*, *descendant*, etc. in a tree. We define the *width* of a (finite or infinite) tree as the (finite or infinite) maximal number of nodes of a given depth. A *deepest* leaf of a finite tree is defined as a leaf of maximal *depth*.

Definition 10.24 (θ-tree)

Let $\theta \in B^*$. A *θ-tree* is a tree of width bounded by $2^{|Q|}$, whose nodes are labelled by either ε or θ.

We say that a finite non-empty tree with node labels in B^* is *pointy* if it has a single deepest leaf. We let the *value* of such a tree be the concatenation of the node labels along the (unique) branch which goes from the root to this deepest leaf. In particular, the value of a finite pointy θ-tree has shape θ^m for some $m \geqslant 0$. An example of pointy θ-tree is depicted horizontally[8] in Figure 10.25.

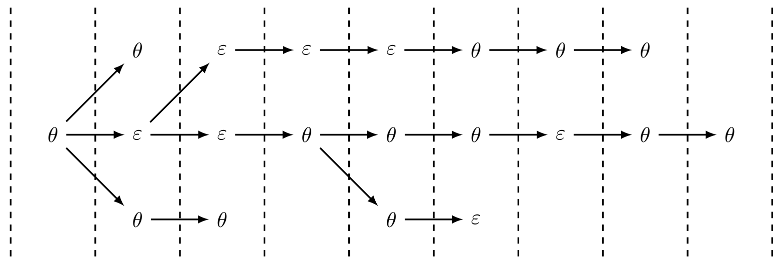


Figure 10.25: A pointy θ-tree of value θ^6 (vertical slices are dashed).

We say that an infinite tree with node labels in B^* is *fertile* if the concatenation of the node labels along any infinite branch which starts in the root is an infinite word. This word is said to be the *value* of the tree if it is the same along all branches. In particular, the value of an infinite fertile θ-tree is θ^ω.

Let us fix $M := \max(4, \max_{q,q' \in Q, a \in A} |\lambda(q,a,q')|)$ and $\Omega := M|Q|^{|Q|}$.

For $|\theta| \leqslant \Omega!$, it is easy to see that a (finite or infinite) θ-tree can be encoded as a (finite or infinite) word over some alphabet Slices. Indeed, since the tree has bounded width, the idea is to make the letters of Slices describe all possible *vertical slices*, i.e. the labels of nodes which have the same depth, together with the according parent relationship (see the dashed slices in Figure 10.25). From now on, we therefore identify finite (resp. infinite) θ-trees for $|\theta| \leqslant \Omega!$ with words of Slices^* (resp. Slices^ω).

Now we are ready to state Theorem 10.26, which shows how to leverage $\mathsf{buildSteps}(u)$ in order to abstract the runs of $\mathscr{T}$ as a (finite or infinite) sequence $\mathsf{buildTrees}(\mathsf{buildSteps}(u))$ of (finite or infinite) θ-trees for various $|\theta| \leqslant \Omega!$. A θ-tree can roughly be understood as a form of non-deterministic computation where all outputs belong to θ^*. We use a fresh symbol $\#$ as a separator between the elements of a sequence. The proof of Theorem 10.26 is presented in Sections 10.5 and 10.6 and it crucially relies on the properties of compatible sets which are presented in Section 10.4.

Theorem 10.26 (Computing θ-trees)

One can build a deterministic regular function[9] $\mathsf{buildTrees}\colon (A \uplus \mathsf{Comp})^\omega \rightharpoonup (\mathsf{Slices} \uplus \{\#\})^\omega$ such that for all $u \in \mathsf{Dom}(f)$, $\mathsf{buildTrees}(\mathsf{buildSteps}(u))$ is:

[8] Since θ-trees are meant to abstract computations of $\mathscr{T}$, we shall represent them in a horizontal fashion.

- either an infinite sequence $t_1 \# t_2 \# \cdots$ where:
  - for all $i \geqslant 1$ t_i is a finite pointy θ_i-tree of value $\theta_i^{m_i}$ with $|\theta_i| \leqslant \Omega!$;
  - $\theta_1^{m_1} \theta_2^{m_2} \cdots = f(u)$;
- or a finite sequence $t_1 \# t_2 \# \cdots \# t_n \# t$ where:
  - for all $1 \leqslant i \leqslant n$, t_i is a finite pointy θ_i-tree of value θ_i^m with $|\theta_i| \leqslant \Omega!$;
  - t is an infinite fertile θ-tree (of value θ^ω) with $|\theta| \leqslant \Omega!$;
  - $\theta_1^{m_1} \cdots \theta_n^{m_n} \theta^\omega = f(u)$.

Proof sketch. Given initial runs which end in a compatible set, recall from Lemma 10.20 that their outputs are mutual prefixes. We prove a stronger result in Section 10.4 (Lemma 10.36):
- either the difference of lengths between these various outputs is "small";
- or the difference of lengths is "big", in which case the ends of these outputs have to be prefixes of θ^ω for some $\theta \in B^*$ with $|\theta| = \Omega!$ (i.e. they commute).

In the first case, the greatest common prefix of these outputs can roughly be produced, and the bounded remainders can be stored in buffers. In the second case, we shall produce a θ-tree which describes the various runs of $\mathscr{T}$. The detailed construction is rather technical. ◀

An example of $\mathsf{buildTrees}(\mathsf{buildSteps}(u))$ is depicted in Figure 10.27 when $f(u) = \theta_1\theta_2\theta_2\theta_3\cdots$. Recall that the vertical slices are encoded by the letters of $\mathsf{Slices} \uplus \{\#\}$.

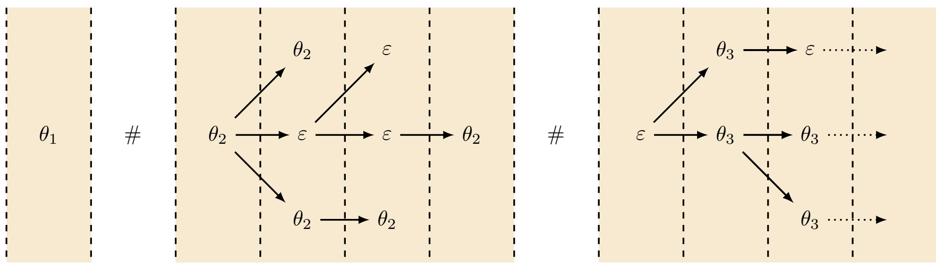


Figure 10.27: A possible value of $\mathsf{buildTrees}(\mathsf{buildSteps}(u))$ when $f(u) = \theta_1\theta_2\theta_2\theta_3\cdots$.

10.2.3 Computing the output

Thanks to Theorem 10.26, we obtain a sequence of trees $\mathsf{buildTrees}(\mathsf{buildSteps}(u))$ which describes the output $f(u)$ whenever $u \in \mathsf{Dom}(f)$. In these trees, branching is only allowed when the outputs belong to θ^* for some $\theta \in B^*$. We finally explain in Theorem 10.28 how a deterministic regular function $\mathsf{buildOutput}$ can produce $f(u)$ when given $\mathsf{buildTrees}(\mathsf{buildSteps}(u))$ as input. The detailed proof of this result is given in Section 10.7 which relies on the construction of a 1-bounded DSST^ω.

Theorem 10.28 (Computing the output)

One can build a deterministic regular function[10] $\mathsf{buildOutput}\colon (\mathsf{Slices} \cup \{\#\})^\omega \rightharpoonup B^\omega$ such that for all $u \in A^\omega$, $\mathsf{buildOutput}(\mathsf{buildTrees}(\mathsf{buildSteps}(u))) = f(u)$.

[9] We shall in fact build a deterministic rational function (but this precise statement is not useful in our proof).

[10] This function is deterministic regular but has no reason to be deterministic rational, contrary to what happened in the constructions of Theorems 10.22 and 10.26.

Proof sketch. We build a 1-bounded DSST^ω (recall from Theorem 9.13 that such a machine computes a deterministic regular function) which computes such a function buildOutput. Indeed, registers offer a flexible way to manipulate the outputs produced along branches of θ-trees.

The main difficulty of the construction is that the DSST^ω cannot know[11] if the θ-tree that it is currently reading is finite (and thus pointy) or infinite. Thus, it has to ensure at the same time that:

- if the current θ-tree is infinite, then the output produced when reading this tree is θ^ω;
- it can recompute the concatenation of the node labels along any branch of the tree starting in the root. Indeed, if the current θ-tree is finite, the DSST^ω has to output exactly its value.

In order to ensure these two properties simultaneously, we devise an original algorithm which manipulates various registers of the DSST^ω to encode portions of the output. ◀

Sections 10.3 to 10.7 are devoted to the detailed proofs of Theorems 10.22, 10.26 and 10.28.

10.3 Computing compatible sets

The goal of this section is to show Theorem 10.22. Given an input word $u \in \mathsf{Dom}(f)$, we explain how a deterministic regular function called buildSteps can compute a sequence of pre-steps which over-approximates the accepting run of $\mathscr{T}$ labelled by u. For this purpose, we shall build a $\mathsf{1DT}^\omega$ with finite lookarounds (recall from Theorem 9.4 that such a machine computes a deterministic regular function).

We first show in Claim 10.30 that using a compatible set is sufficient to describe all the runs which start in a given set and are labelled by a given infinite word. This situation is depicted in Figure 10.29. Recall that if $u \in A^*$ and $S \subseteq Q$, we have defined $u \triangleright S := \{q \mid p \xrightarrow{u} q \text{ for some } p \in S\}$.

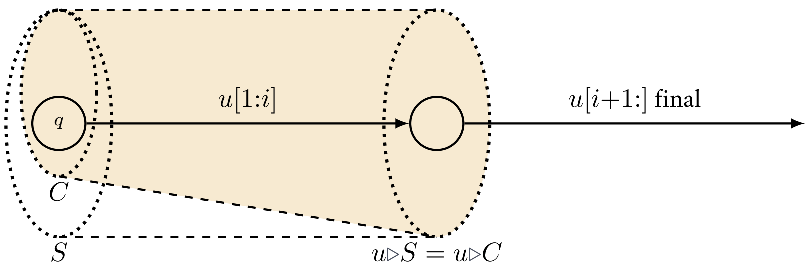


Figure 10.29: Covering the future with a compatible set.

Claim 10.30 (Compatible sets cover the future)

Let $S \subseteq Q$ and $u \in A^\omega$ be such that there exists a final run $q \xrightarrow{u} \infty$ for some $q \in S$. There exist $C \in \mathsf{Comp}(S)$ and $i \geqslant 0$ such that $u[1{:}i] \triangleright S = u[1{:}i] \triangleright C$ (and therefore $q \in C$).

Proof. Assume by contradiction that the property does not hold. Let P be the set of subsets $C \subseteq S$ such that $q \in C$, and such that for all $p \in C$ there exists an infinite run $p \xrightarrow{u} \infty$ (not necessarily final). Observe that $P \subseteq \mathsf{Comp}(S)$ and that $\{q\} \in P$, thus $P \neq \varnothing$.

Now consider a set $C \in P$ such that $|C| = \max_{C' \in P} |C'|$. Since $C \in \mathsf{Comp}(S)$, then by assumption for all $i \geqslant 0$ we have $u[1{:}i] \triangleright C \neq u[1{:}i] \triangleright S$, thus $u[1{:}i] \triangleright (S \smallsetminus C) \neq \varnothing$ (because $u[1{:}i] \triangleright S = (u[1{:}i] \triangleright C) \cup (u[1{:}i] \triangleright (S \smallsetminus C))$). Hence the tree of all runs starting from $S \smallsetminus C$ and labelled by u is infinite, thus by König's lemma it has an infinite branch, i.e. there exists a state $p \in S \smallsetminus C$ such that $p \xrightarrow{u} \infty$. Thus $C \uplus \{p\} \in P$, which contradicts the maximality of $|C|$. ◀

[11] Observe that even if finite lookarounds were allowed, it would not be possible to determine this information.

Now we are ready to describe a $1\mathsf{DT}^\omega$ with finite lookarounds which computes the desired function buildSteps. The main idea is to perform an one-the-fly subset construction, using finite lookaheads and Claim 10.30 to remove useless states and ensure that the current set is compatible.

Let us describe how the $1\mathsf{DT}^\omega$ with finite lookarounds is able to produce $S_0 u[1] S_1 \cdots u[i] S_i$ when reading $u[1{:}i]$, for all $i \geqslant 0$. For $i = 0$, by Claim 10.30 there exist $C \in \mathsf{Comp}(I)$ and $i' \geqslant 0$ such that $u[1{:}i'] \triangleright I = u[1{:}i'] \triangleright C$. The $1\mathsf{DT}^\omega$ uses its finite lookarounds (one for each candidate $C \in \mathsf{Comp}(I)$) to determine some C such that this property holds for the smallest possible $i' \geqslant 0$. It lets S_0 be this set. We have $q_0 \in S_0$. Now for $i \geqslant 1$, assume that S_{i-1} has been computed and let $S := u[i] \triangleright S_{i-1}$. By Claim 10.30 there exist $C \in \mathsf{Comp}(S)$ and $i' \geqslant i$ such that $u[i{:}i'] \triangleright S = u[1{:}i'] \triangleright C$. As before, finite lookarounds can be used to determine some $S_i \in \mathsf{Comp}(S)$ which verifies this property. Furthermore, $S_{i-1}, u[i], S_i$ is a pre-step since by induction $S_{i-1} \subseteq u[1{:}i{-}1] \triangleright I$ and $\mathscr{T}$ is unambiguous.

10.4 Properties of compatible sets

The goal of Section 10.4 is to describe several properties of compatible sets of states (Definition 10.13) which will be useful for the proof of Theorem 10.26 in Sections 10.5 and 10.6. In particular, Lemma 10.36 shows that some productions are prefixes of θ^ω and it is thus the key result for building θ-trees.

10.4.1 Common part and advances

Recall from Lemma 10.20 that the productions of initial runs which end in a compatible set are mutual prefixes. In Section 10.4.1, we therefore introduce several notations to describe how the $\mathsf{prod}^u_{J,C}(q)$ are related when J, u, C is a pre-step. Recall that if the words α, β are mutual prefixes, then $\alpha \wedge \beta$ (resp. $\alpha \vee \beta$) denotes the shortest (resp. the longest) word between α and β.

Definition 10.31 (Common part and advances)

Let J, u, C be an initial pre-step, we define:

- the ***common part*** $\mathsf{com}^u_{J,C} \in B^*$ as the longest common prefix $\bigwedge_{q \in C} \mathsf{prod}^u_{J,C}(q)$;
- for all $q \in C$, its ***advance*** $\mathsf{adv}^u_{J,C}(q) \in B^*$ as $(\mathsf{com}^u_{J,C})^{-1} \mathsf{prod}^u_{J,C}(q)$;
- the ***maximal advance*** $\mathsf{advm}^u_{J,C}$ as the longest advance, i.e. $\bigvee_{q \in C} \mathsf{adv}^u_{J,C}(q)$.

Observe that $\mathsf{prod}^u_{J,C}(q) = \mathsf{com}^u_{J,C} \mathsf{adv}^u_{J,C}(q)$ for all $q \in C$. Furthermore, there exists $p, q \in C$ such that $\mathsf{prod}^u_{J,C}(p) = \mathsf{com}^u_{J,C}$ and $\mathsf{adv}^u_{J,C}(q) = \mathsf{advm}^u_{J,C}$ by definition of the longest common prefix.

Example 10.32 (Common part and avances)

In Figure 8.17c, we get $\mathsf{com}^{0^n}_{\{q_0\},\{q_1,q_2\}} = 0^n$, $\mathsf{adv}^{0^n}_{\{q_0\},\{q_1,q_2\}}(q_1) = \varepsilon$, $\mathsf{adv}^{0^n}_{\{q_0\},\{q_1,q_2\}}(q_2) = 0^n$.

Remark 10.33 (Common part is not regular)

The reader may believe[12] from Example 10.32 that given $J, C \in \mathsf{Comp}$, then $u \in A^* \mapsto \mathsf{com}^u_{J,C}$ (whenever J, u, C is a step) is always a sequential function of finite words. However, this function may not even be regular. Let us justify informally this statement by considering a $1\mathsf{NT}^\omega$ with two possible runs: one performs transitions $a|1, b|\varepsilon$ and the other performs $a|\varepsilon, b|1$. After reading $u \in \{a,b\}^*$, the common part of these runs is $1^{\min(|u|_a, |u|_b)}$ which is not a regular function.

[12] The reader is grateful to Lhote and Passemard for this observation.

10.4.2 Separable compatible sets

The goal of Section 10.4.2 is to state Lemma 10.36, which claims that when the advances are not bounded, they must have a periodic structure. Let us introduce the notion of *separable set*. Intuitively, a compatible set $S \subseteq Q$ is separable if there exists a way to reach S by doing an initial step whose maximal advance is long enough. Recall that $M := \mathsf{max}(4, \mathsf{max}_{q,q' \in Q, a \in A} |\lambda(q, a, q')|)$ and $\Omega := M|Q|^{|Q|}$.

Definition 10.34 (Separable set)

We say that a set $C \subseteq Q$ is *separable* if there exists an initial step J, u, C and $p, q \in C$ such that $\Big| |\mathsf{adv}^u_{J,C}(p)| - |\mathsf{adv}^u_{J,C}(q)| \Big| > \Omega$ (or equivalently, $|\mathsf{advm}^u_{J,C}| > \Omega$).

It is easy to characterize separable sets in terms of loops, as explained in Claim 10.35. As a direct consequence, one can effectively determine if some $C \subseteq Q$ is separable or not. This result also shows that one can build initial steps with arbitrarily large maximal advances.

Claim 10.35 (Characterization of separable sets)

A set $S \in \mathsf{Comp}$ is separable if and only if there exists two functions $i : S \to I$ and $\ell : S \to Q$, $u, u', u'' \in A^*$ and three functions $\alpha, \alpha', \alpha'' : S \to B^*$ such that:

- for all $q \in S$, $i(q) \xrightarrow{u|\alpha(q)} \ell(q) \xrightarrow{u'|\alpha'(q)} \ell(q) \xrightarrow{u''|\alpha''(q)} q$;
- $u' \neq \varepsilon$ and $|u|, |u'|, |u''| \leqslant |Q|^{|Q|}$;
- there exists $p, q \in S$ such that $0 \leqslant |\alpha'(q)| < |\alpha'(p)| \leqslant \Omega$.

Proof. If the conditions holds, then for all $n \geqslant 0$, $i(S), u(u')^n u'', S$ is a step by Claim 10.19. Furthermore, the maximal advance of this step can be made arbitrarily large when $n \to \infty$, thus in particular the compatible set S is separable. Conversely, let J, v, S be a step and $p, q \in S$ be such that $||\mathsf{prod}^v_{J,S}(p)| - |\mathsf{prod}^v_{J,S}(q)|| > \Omega$. Suppose by symmetry that $|\mathsf{prod}^v_{J,S}(p)| > |\mathsf{prod}^v_{J,S}(q)| + \Omega$. Thus $|v| > \Omega/M = |Q|^{|Q|}$. By the pigeonhole principle and since $|S| \leqslant |Q|$, we can factor $v = uu'u''$ with $0 < |u'| \leqslant |Q|^{|Q|}$ such that $i(r) \xrightarrow{u|\alpha(r)} \ell(r) \xrightarrow{u'|\alpha'(r)} \ell(r) \xrightarrow{u''|\alpha''(r)} r$ for all $r \in Q$. Observe that $0 \leqslant |\alpha'(p)|, |\alpha'(q)| \leqslant M|Q|^{|Q|} = \Omega$. Now, if $|\alpha'(p)| = |\alpha'(q)|$, we can remove the loop and get the result by induction since $|uu''| < |v|$ and we preserve $|\alpha(p)\alpha''(p)| > |\alpha(q)\alpha''(q)| + \Omega$. Otherwise $|\alpha'(p)| \neq |\alpha'(q)|$ (thus $|\alpha'(q)| < |\alpha'(p)|$ up to permutation) and we enforce $|u|, |u''| \leqslant |Q|^{|Q|}$ by using once more the pigeonhole principle. ◀

Now we claim that the productions along an initial step which ends in a separable and compatible set necessarily "repeat" some output word $\theta \in B^+$ when the step is pursued, as depicted in Figure 10.37. Lemma 10.36 is a therefore a key ingredient for showing Theorem 10.26 in Sections 10.5 and 10.6. This result originates from [CD22, Lemma 4.13].

Lemma 10.36 (Looping futures)

Let $C \in \mathsf{Comp}$ be separable and J, u, C be an initial step (not necessarily the one which makes C separable). There exists $\tau, \theta \in B^*$ with $|\tau| \leqslant \Omega!$ and $|\theta| = \Omega!$, which can be uniquely determined from C and $\mathsf{adv}^u_{J,C}(p)$ for $p \in C$, such that for all step C, v, D and $q \in D$:

$$\mathsf{adv}^u_{J,C}(p)\mathsf{prod}^v_{C,D}(q) \sqsubseteq \tau\theta^\omega \text{ for } p := \mathsf{pred}^v_{C,D}(q).$$

Proof. The result follows from the stronger Lemma 10.39. ◀

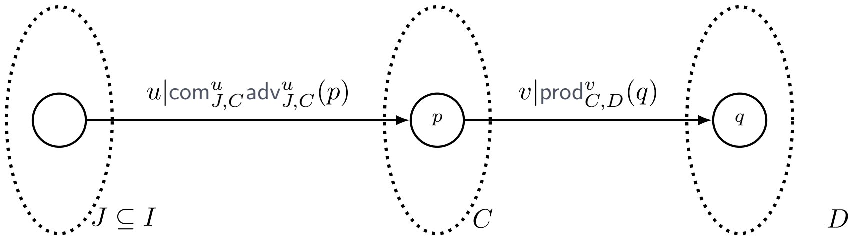


Figure 10.37: Situation of Lemma 10.36 with $\mathsf{adv}^u_{J,C}(p)\mathsf{prod}^v_{C,D}(q) \sqsubseteq \tau\theta^\omega$.

Since C, ε, C is always a step, in particular we have $\mathsf{adv}^u_{J,C}(p) \sqsubseteq \mathsf{advm}^u_{J,C} \sqsubseteq \tau\theta^\omega$ for all $p \in C$.

Example 10.38 (Looping futures)

In Figure 8.17c, the set $C := \{q_1, q_2\}$ is separable. For all step C, v, D we have $D = C, v = 0^n$, $\mathsf{prod}^{0^n}_{C,D}(q_1) = 0^n$ and $\mathsf{prod}^{0^n}_{C,D}(q_2) = 0^{2n}$. Both are prefixes of 0^ω.

10.4.3 Looping futures in separable sets

The goal of this section is to show Lemma 10.39. We shall in fact show a stronger result with Lemma 10.39.

Lemma 10.39 (Looping futures - strong version)

Let $C \in \mathsf{Comp}$ be separable and J, u, C be an initial step (not necessarily the one which makes C separable). For all step C, v, D and for all state $\overline{z} \in D$ with $z := \mathsf{pred}^v_{C,D}(\overline{z})$, we have:

$$\mathsf{adv}^u_{J,C}(z)\mathsf{prod}^v_{C,D}(\overline{z})\mathsf{end}_D(\overline{z}) = \tau\theta^\omega$$

for some τ and θ which only depend on C and on the $\mathsf{adv}^u_{J,C}(t)$ for $t \in C$.

Now we show Lemma 10.39. Since C is separable, we get $i : C \to I, \ell : C \to Q, w, w', w'' \in A^*$, three functions $\alpha, \alpha', \alpha'' : C \to B^*$ and $p, q \in C$ which verify the conditions of Claim 10.35. Recall that we have $|\alpha'(q)| < |\alpha'(p)| \leqslant \Omega$. Since $i(C), ww'^n w'', C$ is a step for all $n \geqslant 0$ by Claim 10.19, then by Lemma 10.20 the $\mathsf{prod}^{ww'^n w''}_{i(C),C}$ are mutual prefixes. Now, we show in Claim 10.40 that the difference of output between p and q necessarily has a looping behavior for n large enough.

Claim 10.41 (Differences are looping)

There exists $\beta, \theta \in B^*$ and $P, N \geqslant 0$, such that $|\beta| \leqslant \Omega$, $|\theta| = \Omega!$, and for all $n \geqslant P$:

$$\beta\theta^{n-P} \sqsubseteq \left(\mathsf{prod}^{ww'^{nN}w''}_{i(C),C}(q)\right)^{-1} \mathsf{prod}^{ww'^{nN}w''}_{i(C),C}(p). \tag{10.41}$$

Furthermore, the values β and θ can effectively be computed and only depend on C.

Proof. Since $|\alpha'(p)| > |\alpha'(q)|$, we can define for n large enough:

$$\pi_n := \left(\mathsf{prod}^{ww'^n w''}_{i(C),C}(q)\right)^{-1} \mathsf{prod}^{ww'^n w''}_{i(C),C}(p) = \alpha(p)\alpha'(p)^n\alpha''(p)[t_n:]$$

where $t_n := |\alpha(q)| + n|\alpha'(q)| + |\alpha''(q)|$. We assume that $|\alpha'(q)| > 0$ (the case $|\alpha'(q)| = 0$ is somehow simpler since then t_n is constant). For n large enough, consider:

$$\begin{aligned}\pi_{n|\alpha'(p)|} &= \left(\alpha(p)\alpha'(p)^{n|\alpha'(p)|}\alpha''(p)\right)[t_{n|\alpha'(p)|}:] \\ &= \left(\alpha'(p)^{n(|\alpha'(p)|-|\alpha'(q)|)+K}\alpha''(p)\right)[t:]\end{aligned}$$

where t is (constant and) defined below and K is chosen in a way which ensures $t \geqslant 0$:

$$\begin{aligned}t &:= t_{n|\alpha'(p)|} - n|\alpha'(p)||\alpha'(q)| + K|\alpha'(p)| - |\alpha(p)| \\ &= |\alpha(q)| + |\alpha''(q)| - |\alpha(p)| + K|\alpha'(p)|.\end{aligned}$$

We let $\theta := \alpha'(p)^{\Omega!/|\alpha'(p)|}$ (thus we get $|\theta| = \Omega!$), β as a suffix of $\alpha'(p)$ which depends on t, $N := \Omega!/(|\alpha'(p)| - |\alpha'(q)|) = |\alpha'(p)| \underbrace{\frac{\Omega!}{|\alpha'(p)|(|\alpha'(p)| - |\alpha'(q)|)}}_{\text{integer}}$ and P accordingly. ◀

From this result, now we deduce that the possible future steps have a looping behavior.

Claim 10.42 (Futures are looping)

For all step C, v, D and all $\overline{r} \in D$, if $r := \mathsf{pred}^v_{C,D}(\overline{r})$, we have:

$$\mathsf{prod}^v_{C,D}(\overline{r})\mathsf{end}_D(\overline{r}) = (\mathsf{adv}^{ww''}_{i(C),C}(r))^{-1}(\mathsf{adv}^{ww''}_{i(C),C}(q))\beta\theta^\omega.$$

Proof. Let $\overline{p}, \overline{q} \in D$ be such that $\mathsf{pred}^v_{C,D}(\overline{p}) = p$ and $\mathsf{pred}^v_{C,D}(\overline{q}) = q$. It follows from Claim 10.19 that $i(C), ww'^n w''v, D$ is an initial step for all $n \geqslant 0$, thus by Claim 10.21:

$$\mathsf{prod}^v_{C,D}(\overline{q})\mathsf{end}_D(\overline{q}) = \left(\mathsf{prod}^{ww'^n w''}_{i(C),C}(q)\right)^{-1} \mathsf{prod}^{ww'^n w''}_{i(C),C}(p)\mathsf{prod}^v_{C,D}(\overline{p})\mathsf{end}_D(\overline{p}).$$

For n large enough, Claim 10.40 shows $\beta\theta^{n-P} \sqsubseteq \left(\mathsf{prod}^{ww'^{nN}w''}_{i(C),C}(q)\right)^{-1} \mathsf{prod}^{ww'^{nN}w''}_{i(C),C}(p)$. Therefore $\beta\theta^{n-M} \sqsubseteq \mathsf{prod}^v_{C,D}(\overline{q})\mathsf{end}_D(\overline{q})$. Hence $\mathsf{prod}^v_{C,D}(\overline{q})\mathsf{end}_D(\overline{q}) = \beta\theta^\omega$. Finally, by applying Claim 10.21 once more to the initial step $i(C), ww''v, D$, we get:

$$\mathsf{prod}^v_{C,D}(\overline{r})\mathsf{end}_D(\overline{r}) = (\mathsf{adv}^{ww''}_{i(C),C}(r))^{-1}\mathsf{adv}^{ww''}_{i(C),C}(q)\underbrace{\mathsf{prod}^v_{C,D}(\overline{q})\mathsf{end}_D(\overline{q})}_{=\beta\theta^\omega}. \quad ◀$$

Now, let us consider what happens with the step J, u, C. Let $r \in C$ (resp. $s \in C$) be such that $\mathsf{adv}^u_{J,C}(r) = \varepsilon$ (resp. $\mathsf{adv}^u_{J,C}(s) = \mathsf{advm}^u_{J,C}$), i.e. the run ending in r (resp. in s) has the shortest (resp. the longest) production. Observe that one may have $r = s$.

Let C, v, D be a step and $\overline{s} \in D$ (resp. $\overline{r}$) be such that $s = \mathsf{pred}^v_{C,D}(\overline{s})$ (resp. $r = \mathsf{pred}^v_{C,D}(\overline{r})$). Note that such a step always exists (at least the empty one C, ε, C), and furthermore:

$$\begin{aligned}\mathsf{advm}^u_{J,C}\mathsf{prod}^v_{C,D}(\overline{s})\mathsf{end}_D(\overline{s}) &= \mathsf{adv}^u_{J,C}(s)\mathsf{prod}^v_{C,D}(\overline{s})\mathsf{end}_D(\overline{s}) && \text{by choice of } s; \\ &= \mathsf{adv}^u_{J,C}(r)\mathsf{prod}^v_{C,D}(\overline{r})\mathsf{end}_D(\overline{r}) && \text{by Claim 10.21 for } J, uv, D; \\ &= \varepsilon\ \mathsf{prod}^v_{C,D}(\overline{r})\mathsf{end}_D(\overline{r}) && \text{by choice of } r; \\ &= (\mathsf{adv}^{ww''}_{i(C),C}(r))^{-1}(\mathsf{adv}^{ww''}_{i(C),C}(q))\beta\theta^\omega && \text{by Claim 10.42.}\end{aligned}$$

Let $m := |\mathsf{adv}^{ww''}_{i(C),C}(q)| - |\mathsf{adv}^{ww''}_{i(C),C}(r)| + |\beta|$, then $-2\Omega \leqslant m \leqslant 3\Omega$ (indeed $|\beta| \leqslant \Omega$, and furthermore $|\mathsf{adv}^{ww''}_{i(C),C}(q)| \leqslant 2\Omega$ and $|\mathsf{adv}^{ww''}_{i(C),C}(r)| \leqslant 2\Omega$ because $|ww''| \leqslant 2|Q|^{|Q|}$). Now, recall that $|\theta| = \Omega!$ and observe that $\Omega! \geqslant 3\Omega$ because $\Omega \geqslant 4$ (indeed, $\Omega = M|Q|^{|Q|}$ and $M \geqslant 4$). We build the value $\tau \in B^*$ depending on the sign of m:

- if $m \geqslant 0$, we let $\tau := (\mathsf{adv}^{ww''}_{i(C),C}(r))^{-1}(\mathsf{adv}^{ww''}_{i(C),C}(q))\beta$;
- if $m < 0$, we let $\tau := (\mathsf{adv}^{ww''}_{i(C),C}(r))^{-1}(\mathsf{adv}^{ww''}_{i(C),C}(q))\beta\theta$.

Note that $|\tau| \leqslant \Omega!$ and that this value it only depends on the step $J, ww'w'', C$ (as it is the case of β) and on the $\mathsf{adv}^{u}_{J,C}(t)$ for $t \in C$ (this information is needed to determine r), but *not* on the "future" step C, v, D that we have selected. Hence, for all step C, v, D we have:

$$\mathsf{advm}^{u}_{J,C}\mathsf{prod}^{v}_{C,D}(\overline{s})\mathsf{end}_D(\overline{s}) = \tau\theta^\omega.$$

Finally, for all $\overline{z} \in D$ with $z := \mathsf{pred}^{v}_{C,D}(\overline{z})$, we conclude the proof of Lemma 10.39 follows:

$$\begin{aligned}
\mathsf{adv}^{u}_{J,C}(z)\mathsf{prod}^{v}_{C,D}(\overline{z})\mathsf{end}_D(\overline{z}) &= \mathsf{adv}^{u}_{J,C}(s)\mathsf{prod}^{v}_{C,D}(\overline{s})\mathsf{end}_D(\overline{s}) && \text{by Claim 10.21 for } J, uv, D;\\
&= \mathsf{advm}^{u}_{J,C}\mathsf{prod}^{v}_{C,D}(\overline{s})\mathsf{end}_D(\overline{s}) && \text{by choice of } s;\\
&= \tau\theta^\omega.
\end{aligned}$$

10.5 Computing (τ, θ)-trees from compatible sets

The goal of Section 10.5 is to show a first half of Theorem 10.26. More precisely, we show how to transform a sequence of pre-steps which over-approximates the accepting run of $\mathscr{T}$ into a sequence of *(τ, θ)-trees* which verifies the conditions of Proposition 10.44. Informally, (τ, θ)-trees can be seen as relaxed versions of θ-trees. We shall explain in Section 10.6 how to finally obtain θ-trees.

Definition 10.43 ((τ, θ)-tree)

Let $\tau, \theta \in B^*$. A *(τ, θ)-tree* is a tree of width bounded by $2^{|Q|}$, whose nodes labels belong to $\tau\theta^*$ or θ^*. Furthermore, the two following conditions hold:

- any node with label in θ^+ has an ancestor with label in $\tau\theta^*$
- along a given branch, there is at most one node with label in $\tau\theta^*$.

An example of (τ, θ)-tree is depicted in Figure 10.54a. The value of a finite pointy (τ, θ)-tree is either ε or $\tau\theta^m$ for some $m \geqslant 0$. The value of an infinite fertile (τ, θ)-tree is necessarily $\tau\theta^\omega$. In practice, the node labels of the (τ, θ)-trees that we shall manipulate will always belong to a finite set. Therefore we assume that the alphabet Slices can be used to describe the vertical slices of (τ, θ)-trees (recall that this alphabet was introduced in Section 10.2.2 to describe the vertical slices of θ-trees).

Intuitively, the reason why (τ, θ)-trees occur in our construction is Lemma 10.36, which shows that the productions starting in separable compatible sets are prefixes of $\tau\theta^\omega$ for some $\tau, \theta \in B^*$. Hence the proof of Proposition 10.44 will crucially rely on the properties of compatible sets.

Proposition 10.44 (Computing (τ, θ)-trees)

One can build a sequential function $g\colon (A \uplus \mathsf{Comp})^\omega \rightharpoonup (\mathsf{Slices} \uplus \{\#\})^\omega$ such that for all $u \in \mathsf{Dom}(f)$, $g(\mathsf{buildSteps}(u))$ is:

- either an infinite sequence $t_1 \# t_2 \# \cdots$ where:

  - ▶ for all $i \geqslant 1$ t_i is a finite pointy (τ_i, θ_i)-tree of value α_i with $|\tau_i|, |\theta_i| \leqslant \Omega!$;
  - ▶ $\alpha_1\alpha_2 \cdots = f(u)$;
- ▶ or a finite sequence $t_1\#t_2\# \cdots \#t_n\#t$ where:
  - ▶ for all $1 \leqslant i \leqslant n$, t_i is a finite pointy (τ_i, θ_i)-tree of value α_i with $|\tau_i|, |\theta_i| \leqslant \Omega!$;
  - ▶ t is an infinite fertile (τ, θ)-tree (of value $\tau\theta^\omega$) with $|\theta| \leqslant \Omega!$;
  - ▶ $\alpha_1 \cdots \alpha_n \theta^\omega = f(u)$.

The rest of Section 10.5 is devoted to the detailed proof of Proposition 10.44. For this, we describe a $\mathsf{1DT}^\omega$ which computes a function $g\colon (A \uplus \mathsf{Comp})^\omega \rightharpoonup (\mathsf{Slices} \uplus \{\#\})^\omega$.

In order to simplify the notations, we extend the notions of advances and common part to subsets of pre-steps. Formally, if J, v, C is a pre-step and $\varnothing \neq R \subseteq C$, we let $\mathsf{prod}^v_{J,R}(q) := \mathsf{prod}^v_{J,C}(q)$ for $q \in R$. If J, v, C is an initial pre-step, we define $\mathsf{com}^v_{J,R}$, $\mathsf{adv}^v_{J,R}$ and $\mathsf{advm}^v_{J,R}$ as we did in Definition 10.31 for C. This definition makes sense since the $\mathsf{prod}^v_{J,R}(q)$ for $q \in R$ are still mutual prefixes.

10.5.1 Information stored by the one-way transducer

In this section we present the informations that will be stored in the finite memory of the $\mathsf{1DT}^\omega$ which computes a function g verifying the conditions of Proposition 10.44. More precisely, we shall describe Invariants (1) to (5) which are maintained during the computation of this $\mathsf{1DT}^\omega$.

10.5.1.1 Rigid sets. Let $u \in \mathsf{Dom}(f)$. Assume that $\mathsf{buildSteps}(u)$ has shape $S_0u[1]S_1u[2]S_2\cdots$. For $i \geqslant 0$, we say that $R \subseteq S_i$ is i-rigid if the maximal advance of the initial runs described by $S_0u[1]S_1 \cdots u[i]S_i$ has "always" been "small". This notion is formalized in Definition 10.45.

Definition 10.45 (Rigid set)

Let $i \geqslant 0$ and $R \subseteq S_i$. We say that R is *i-rigid* if $\mathsf{advm}^{u[1:j]}_{S_0, \mathsf{pred}^{u[j+1:i]}_{S_j,S_i}(R)} < 2\Omega!$ for all $0 \leqslant j \leqslant i$.

Observe that the subsets of S_i which are not i-rigid are necessarily separable. However, the converse may not hold: being separable means that initial runs with different lengths can be found, but it is not necessarily those which are described by the sequence $S_0u[1]S_1u[2]S_2\cdots$. Also observe that any subset of an i-rigid set is also i-rigid. We denote by Rigid_i the set of maximal (for inclusion) i-rigid subsets of S_i. Since the sets $\{q\}$ for $q \in S_i$ are i-rigid, we have $\varnothing \notin \mathsf{Rigid}_i$ for all $i \geqslant 0$.

10.5.1.2 Invariants maintained by the one-way transducer. Assume that the $\mathsf{1DT}^\omega$ has read $S_0u[1]S_1 \cdots u[i]S_i$ for some $i \geqslant 0$. It has produced the output $t_1\# \cdots \#t_\ell\#t \in (C \uplus \{\#\})^*$ where:

- ▶ for all $1 \leqslant j \leqslant \ell$, t_j is a pointy (τ_j, θ_j)-tree of value ψ_j, where $|\tau_j| \leqslant \Omega!$, $|\theta_j| = \Omega!$;
- ▶ t is a (τ, θ)-tree where $|\tau| \leqslant \Omega!$, $|\theta| = \Omega!$ (this tree is currently being built).

We let $\psi := \psi_1 \cdots \psi_\ell$. We shall ensure that the following invariants hold:

1. the deepest leaves of t are indexed by the elements of Rigid_i. For all $R \in \mathsf{Rigid}_i$, let α_R be the concatenation of the labels along the branch from the root of t to the deepest leaf indexed by R;
2. the $\mathsf{1DT}^\omega$ stores (in its finite states) the functions[13] $\mathsf{adv}^{u[1:i]}_{S_0,R}\colon R \to B^*$ for $R \in \mathsf{Rigid}_i$;
3. the $\mathsf{1DT}^\omega$ stores (in its finite states) two functions $\mathsf{buffer}_1, \mathsf{buffer}_2\colon \mathsf{Rigid}_i \to B^*$ such that:
   (a) for all $R \in \mathsf{Rigid}_i$, $\mathsf{buffer}_1(R) \sqsubseteq \tau$ and $\mathsf{buffer}_2(R) \sqsubset \theta^2$;

[13] Since the set R is i-rigid, this information is bounded.

   (b) for all $R \in \mathsf{Rigid}_i$, $\psi\, \alpha_R\, \mathsf{buffer}_1(R)\, \mathsf{buffer}_2(R) = \mathsf{com}^{u[1:i]}_{S_0,R}$;
(4) if S_i is i-rigid (then $|\mathsf{Rigid}_i| = 1$), then $\mathsf{buffer}_1(S_i) = \mathsf{buffer}_2(S_i) = \varepsilon$;
(5) if S_i is not i-rigid (then $|\mathsf{Rigid}_i| > 1$):
   (a) for all $R \in \mathsf{Rigid}_i$, if $|\mathsf{buffer}_2(R)| < \Omega!$ then $\alpha_R = \varepsilon$, and $\mathsf{buffer}_2(R) = \varepsilon$ if $\mathsf{buffer}_1(R) \neq \tau$;
   (b) for all $R \in \mathsf{Rigid}_i$, if $|\mathsf{buffer}_2(R)| \geqslant \Omega!$ then $\mathsf{buffer}_1(R) = \varepsilon$ and $\alpha_R \in \tau\theta^*$;
   (c) for all "future" step S_i, v, D and $q \in D$, $\mathsf{prod}^{u[1:i]v}_{S_0,D}(q) \sqsubseteq \psi\, \tau\, \theta^\omega$.

The main role of buffer_2 is to temporarily keep track of part of the current production, until a value θ is completely produced and therefore can be added to the tree. The role of buffer_1 is similar for τ. Furthermore, these buffers will play a role to ensure that the productions are "long enough" in Section 10.5.2.3 when dealing with the most technical case of update (when $S_i, u[i{+}1], S_{i+1}$ is not a step).

10.5.2 Updates of the one-way transducer

The goal of Section 10.5.2 it to describe the updates of the $\mathsf{1DT}^\omega$ which computes the function g, while preserving the invariants of Section 10.5.1.2 at each step of its computation.

For $i = 0$, observe that S_0 is 0-rigid and $\mathsf{com}^{\varepsilon}_{S_0,S_0} = \varepsilon$. Therefore it is sufficient to output a tree which consists in a single node labelled by ε and indexed by S_0. The advances are also empty and we let $\mathsf{buffer}_1(S_0) := \mathsf{buffer}_2(S_0) := \varepsilon$. It is easy to see that Invariants (1) to (4) hold.

In the rest of Section 10.5.2, we assume that the $\mathsf{1DT}^\omega$ has read the input $S_0 u[1] S_1 \cdots u[i] S_i$ so far for some $i \geqslant 1$ and that the invariants of Section 10.5.1.2 hold (we shall re-use the notations of this section). We explain the updates of the $\mathsf{1DT}^\omega$ when reading $u[i{+}1]S_{i+1}$, which on depends the various cases presented in Sections 10.5.2.1 to 10.5.2.3. Observe that the $\mathsf{1DT}^\omega$ can determine which case holds, thanks to the bounded information that it has stored in its finite states.

10.5.2.1 Update when $\mathsf{pred}^{u[i+1]}_{S_i,S_{i+1}}(S_{i+1})$ is not i-rigid and $S_i, u[i{+}1], S_{i+1}$ is a step. In this case, S_i cannot be i-rigid and therefore Invariant (5) holds. Furthermore, S_{i+1} cannot be $(i{+}1)$-rigid. For all $S \in \mathsf{Rigid}_{i+1}$, let[14] $R_S \in \mathsf{Rigid}_i$ be such that $\mathsf{pred}^{u[i+1]}_{S_i,S_{i+1}}(S) \subseteq R_S$.

Recall that by Invariant (1) the deepest leaves of the (τ,θ)-tree t are indexed by the elements or Rigid_i. The main idea is to pursue the construction on this tree by relying Invariant (5)(c).

Let $S \in \mathsf{Rigid}_{i+1}$. By recombining the values $\mathsf{adv}^{u[1:i]}_{S_0,R_S}(q)$ for $q \in R_S$ and $\mathsf{prod}^{u[i+1]}_{S_i,S_{i+1}}(q)$ for $q \in S$, the $\mathsf{1DT}^\omega$ can determine[15] the values $\mathsf{adv}^{u[1:i+1]}_{S_0,S}(q)$ for $q \in S$ and $\beta_S \in B^*$ such that:

$$\psi\, \alpha_{R_S}\, \mathsf{buffer}_1(R)\, \mathsf{buffer}_2(R)\, \beta_S = \mathsf{com}^{u[1:i+1]}_{S_0,S}.$$

Since $S_i, u[i], S_{i+1}$ is a step, we get by Invariant (5):

- if $\mathsf{buffer}_2(R_S) \geqslant \Omega!$, then $\tau \sqsubseteq \alpha_R$, $\mathsf{buffer}_1(R_S) = \varepsilon$ and $\mathsf{buffer}_2(R)\beta_S \sqsubseteq \theta^\omega$. Therefore the $\mathsf{1DT}^\omega$ can determine $m \geqslant 0$ and $\theta \sqsubseteq \theta' \sqsubset \theta^2$ such that $\mathsf{buffer}_2(R_S)\beta_S = \theta^m\theta'$.
  In this case, the $\mathsf{1DT}^\omega$ defines $\mathsf{buffer}_1(S) := \varepsilon$ and $\mathsf{buffer}_2(S) := \theta'$. Concerning the output, it adds a child to the node of t indexed by R_S. This child is indexed by S and labelled with θ^m.
- if $\mathsf{buffer}_2(R_S) < \Omega!$, then $\mathsf{buffer}_1(R_S)\, \mathsf{buffer}_2(R_S)\beta_S \sqsubseteq \tau\, \theta^\omega$. The $\mathsf{1DT}^\omega$ can determine $\tau' \sqsubseteq \tau$ and $\theta' \sqsubseteq \theta^\omega$ such that $\mathsf{buffer}_1(R_S)\, \mathsf{buffer}_2(R_S)\beta_S = \tau'\theta'$ and $\theta' = \varepsilon$ if $\tau' \neq \tau$. Now:
  - if $\theta' \sqsubset \theta$, the $\mathsf{1DT}^\omega$ updates $\mathsf{buffer}_1(S) := \tau'$ and $\mathsf{buffer}_2(S) := \theta'$. For the output, it adds a child to the node of t indexed by R_S. This child is indexed by S and labelled with ε;

[14] There may exist several such R_S, but choosing any of them is enough.
[15] Formally, this computation is hardcoded in its states and transitions, since it only manipulates bounded values.

- if $\theta \sqsubseteq \theta'$, the $1\mathsf{DT}^\omega$ determines $m \geqslant 0$ and $\theta \sqsubseteq \theta'' \sqsubset \theta^2$ such that $\theta^m\theta'' = \theta'$. It updates $\mathsf{buffer}_1(S) := \varepsilon$ and $\mathsf{buffer}_2(S) := \theta''$. Concerning the output, it adds a child to the node of t indexed by R_S. This child is indexed by S and labelled with $\tau\theta^m$;

It is easy to show that the invariants of Section 10.5.1.2 hold after applying this operation for all $S \in \mathsf{Rigid}_{i+1}$. The fact that $S_i, u[i{+}1], S_{i+1}$ is a step is crucial for maintaining Invariant (5)(c).

10.5.2.2 Update when $\mathsf{pred}^{u[i+1]}_{S_i,S_{i+1}}(S_{i+1})$ is i-rigid. In this case, there exists $R \in \mathsf{Rigid}_i$ such that $\mathsf{pred}^{u[i+1]}_{S_i,S_{i+1}}(S_{i+1}) \subseteq R$. Furthermore by Invariant (1) the (τ,θ)-tree t has a deepest leaf indexed by R and by Invariant (3)(b) we obtain $\psi\, \alpha_R\, \mathsf{buffer}_1(R)\, \mathsf{buffer}_2(R) = \mathsf{com}^{u[1:i]}_{S_0,R}$.

By recombining the values $\mathsf{buffer}_1(R)$, $\mathsf{buffer}_2(R)$, $\mathsf{adv}^{u[1:i]}_{S_0,R}(q)$ for $q \in C$ and $\mathsf{prod}^{u[i+1]}_{S_i,S_{i+1}}(q)$ for $q \in S_{i+1}$, the $1\mathsf{DT}^\omega$ can determine the values $\mathsf{adv}^{u[1:i+1]}_{S_0,S_{i+1}}(q)$ for $q \in S_{i+1}$ and $\beta \in B^*$ such that $\psi\, \alpha_R\, \beta = \mathsf{com}^{u[1:i+1]}_{S_0,S_{i+1}}$. It also factors $\beta = \beta_1 \cdots \beta_p$ with $|\beta_j| \leqslant \Omega!$ for all $1 \leqslant j \leqslant p$.

For the output, $1\mathsf{DT}^\omega$ first adds a child labelled by ε to the deepest leaf of t which is indexed by R. This way, the last (τ,θ)-tree of the output is now pointy and has value α_R. After this tree, the $1\mathsf{DT}^\omega$ adds p new trees consisting of single nodes labelled by $\beta_1, \dots, \beta_p$. The rest depends on S_{i+1}:

- if S_{i+1} is $(i{+}1)$-rigid, then the $1\mathsf{DT}^\omega$ simply lets $\mathsf{buffer}_1(S_{i+1}) := \varepsilon$ and $\mathsf{buffer}_2(S_{i+1}) := \varepsilon$;
- if S_{i+1} is not $(i{+}1)$-rigid, then in particular S_{i+1} is separable. Thanks to Lemma 10.36, there exist $\gamma, \pi \in B^*$ with $|\gamma| \leqslant \Omega!$, and $|\pi| = \Omega!$ (which can be determined by the $1\mathsf{DT}^\omega$ thanks to the $\mathsf{adv}^{u}_{S_0,S_{i+1}}(q)$ for $q \in C$) such that for all step S_{i+1}, v, D and $q \in D$ we have:

$$\mathsf{adv}^{u[1:i+1]}_{S_0,S_{i+1}}(p)\mathsf{prod}^{v}_{S_{i+1},D}(q) \sqsubseteq \gamma\pi^\omega \text{ for } p := \mathsf{pred}^{v}_{S_{i+1},D}(q). \tag{10.46}$$

  For all $S \in \mathsf{Rigid}_{i+1}$, the $1\mathsf{DT}^\omega$ can determine the values $\mathsf{adv}^{u[1:i+1]}_{S_0,S}(q)$ for $q \in S$ and β_S such that $\psi\, \alpha_R\, \beta\, \beta_S = \mathsf{com}^{u[1:i+1]}_{S_0,S}$. Furthermore since $\psi\, \alpha_R\, \beta = \mathsf{com}^{u[1:i+1]}_{S_0,S_{i+1}}$ and thanks to Equation (10.46), we get $\beta_S \sqsubseteq \gamma\pi^\omega$. The $1\mathsf{DT}^\omega$ then initiates a (γ,π)-tree which consists of a root labelled by ε and children indexed by the $S \in \mathsf{Rigid}_{i+1}$. The labels of these children and the according $\mathsf{buffer}_1(S)$ and $\mathsf{buffer}_2(S)$ are built as we did in Section 10.5.2.1.

It is easy to show that the invariants of Section 10.5.1.2 hold after this operation.

10.5.2.3 Update when $\mathsf{pred}^{u[i+1]}_{S_i,S_{i+1}}(S_{i+1})$ is not i-rigid and $S_i, u[i{+}1], S_{i+1}$ is not a step. The key difference with Section 10.5.2.1 is that Invariant (5)(c) cannot be used since $S_i, u[i{+}1], S_{i+1}$ is not a step. Therefore one cannot ensure that the $\mathsf{prod}^{u[i+1]}_{S_i,S_{i+1}}(q)$ for $q \in S_{i+1}$ are factors of $\tau\theta^\omega$.

Let $S'_i := \mathsf{pred}^{u[i+1]}_{S_i,S_{i+1}}(S_{i+1})$, observe that $S'_i, u[i{+}1], S_{i+1}$ is a now a step. Let Rigid'_i be the set of maximal i-rigid subsets of S'_i, observe that $\mathsf{Rigid}'_i = \{S \cap S'_i \mid S \in \mathsf{Rigid}_i\}$. By recombining the values of the buffers and the advances as done in Section 10.5.2.1, the $1\mathsf{DT}^\omega$ can add slices to the (τ,θ)-tree and update the buffers so that now the invariants of Section 10.5.1.2 hold when S_i is replaced by S'_i, except for Invariant (5)(c) which cannot be preserved. However, we still have that $\mathsf{prod}^{u[1:i]}_{S_0,S'_i}(q) \sqsubseteq \psi\, \tau\, \theta^\omega$ for all $q \in S'_i$ since this result was true for all $q \in S_i$. With such a preparation, we assume that the deepest leaves of the (τ,θ)-tree are henceforth indexed by the sets of Rigid'_i.

The rest of the proof distinguishes two cases, depending on whether the (τ,θ)-tree has non-empty values on the branches which go from the root to its deepest leaves or not:

- if $|\mathsf{buffer}_2(R)| < \Omega!$ for all $R \in \mathsf{Rigid}'_i$, then $\alpha_R = \varepsilon$ for all $R \in \mathsf{Rigid}'_i$ thanks to Invariant (5)(a). In other words, the (τ,θ)-tree t only contains empty information along its useful branches. In this case, the main idea is to finish this tree and to create a new tree.
Observe that $\psi \sqsubseteq \mathsf{com}^{u[1:i]}_{S_0,S'_i}$ thanks to Invariant (3)(b). Thus by recombining the values stored in the buffers and advances, the $1\mathrm{DT}^\omega$ can determine $\beta \in B^*$ such that $\psi\beta = \mathsf{com}^{u[1:i+1]}_{S_0,S_{i+1}}$. It factors $\beta = \beta_1 \cdots \beta_p$ with $|\beta_j| \leqslant \Omega!$ for all $1 \leqslant j \leqslant p$. The $1\mathrm{DT}^\omega$ thus adds an extra leaf with label ε to any of the deepest leaves of the (τ,θ)-tree, which becomes pointy and with value ε. After this tree, the $1\mathrm{DT}^\omega$ adds p new trees consisting of single nodes labelled by $\beta_1,\dots,\beta_p$.
Since S_{i+1} is separable, then by Lemma 10.36 the $1\mathrm{DT}^\omega$ can compute values $\gamma,\pi \in B^*$ with $|\gamma| \leqslant \Omega!$, and $|\pi| = \Omega!$. The end of the construction consists in creating a new (γ,π)-tree, as we did in Section 10.5.2.2. After this operation, all the invariants of Section 10.5.1.2 hold;
- if $|\mathsf{buffer}_2(R)| \geqslant \Omega!$ for some $R \in \mathsf{Rigid}'_i$. In this case, we intend to show that Invariant (5)(c) still holds with the current τ and θ, by relying on the fact that $\mathsf{prod}^{u[1:i]}_{S_0,S'_i}(q)$ for some $q \in S'_i$ is “long enough” to exhibit repetitions of θ. More precisely, we show Claim 10.47.

Claim 10.47 (Preservation of Invariant (5)(c))

For all step S'_i, v, D and $q \in D$, $\mathsf{prod}^{u[1:i]v}_{S_0,D}(q) \sqsubseteq \psi\,\tau\,\theta^\omega$.

Proof. Since S'_i is not i-rigid, there exists $0 \leqslant j \leqslant i$ such that $\mathsf{advm}^{u[1:j]}_{S_0,E} \geqslant 2\Omega!$ where E is defined as $\mathsf{pred}^{u[j+1:i]}_{S_j,S_i}(C)$. In particular E is separable and by applying Lemma 10.36 to $S_0, u[1:j], E$, there exists[16] values $\gamma,\pi \in B^*$ with $|\gamma| \leqslant \Omega!$ and $|\pi| = \Omega!$, which describe loops in the productions. In particular, there exists $\eta \sqsubseteq \pi^\omega$ such that $\mathsf{advm}^{u[1:j]}_{S_0,E} = \gamma\eta$. Since $\mathsf{advm}^{u[1:j]}_{S_0,E} \geqslant 2\Omega!$, one has $|\eta| = |\mathsf{advm}^{u[1:j]}_{S_0,E}| - |\gamma| \geqslant 2\Omega! - \Omega! \geqslant \Omega!$.
For all step C, v, D, we obtain a step $E, u[j{+}1{:}i]v, D$. Therefore, for all $q \in D$:

$$\mathsf{adv}^{u[1:j]}_{S_0,E}(\mathsf{pred}^{u[j+1:i]v}_{E,D}(q))\mathsf{prod}^{u[j+1:i]v}_{E,D}(q) \sqsubseteq \gamma\pi^\omega$$

and thus, by adding $\mathsf{com}^{u[1:j]}_{S_0,E}$ on both sides:

$$\mathsf{prod}^{u[1:i]v}_{S_0,D}(q) \sqsubseteq \mathsf{com}^{u[1:j]}_{S_0,E}\gamma\pi^\omega. \tag{10.48}$$

To obtain Claim 10.47, it is thus sufficient to show that $\mathsf{com}^{u[1:j]}_{S_0,E}\gamma\pi^\omega = \psi\,\tau\,\theta^\omega$.
For showing this statement, we consider $q \in S'_i$ such that $\mathsf{prod}^{u[1:i]}_{S_0,S'_i}(q)$ has maximal length, then for all $p \in S'_i$ we have $\mathsf{prod}^{u[1:i]}_{S_0,S'_i}(p) \sqsubseteq \mathsf{prod}^{u[1:i]}_{S_0,S'_i}(q)$. Since there exists $R \subseteq S'_i$ such that $|\mathsf{buffer}_2(R)| \geqslant \Omega!$ and $\mathsf{com}^{u[1:i]}_{S_0,R} \sqsubseteq \mathsf{prod}^{u[1:i]}_{S_0,S'_i}(q)$, we conclude thanks to the invariants that $\mathsf{prod}^{u[1:i]}_{S_0,S'_i}(q)$ has shape $\psi\tau\,\theta^m\,\theta'$ for some $m \geqslant 1$ and $\theta' \sqsubset \theta$. Furthermore:

$$\begin{aligned}\mathsf{com}^{u[1:j]}_{S_0,E}\gamma\eta &= \mathsf{com}^{u[1:j]}_{S_0,E}\mathsf{advm}^{u[1:j]}_{S_0,E}\\ &\sqsubseteq \psi\tau\,\theta^m\,\theta' = \mathsf{prod}^{u[1:i]}_{S_0,S'_i}(q) \sqsubseteq \mathsf{com}^{u[1:j]}_{S_0,E}\gamma\pi^\omega.\end{aligned} \tag{10.49}$$

by using Equation (10.48) to obtain the rightmost hand-side.
Finally, two cases can occur depending on the sign of $\ell := |\psi\tau| - |\mathsf{com}^{u[1:j]}_{S_0,E}\gamma|$:

- if $\ell \geqslant 0$, then $\psi\tau = \mathsf{com}^{u[1:j]}_{S_0,E}\gamma(\pi^\omega[1{:}\ell])$ and $\theta = \pi^\omega[\ell{+}1{:}\ell{+}1{+}\Omega!]$;

[16] Since our goal is to pursue the current (τ,θ)-tree, the $1\mathrm{DT}^\omega$ will have no need to determine these values.

- if $\ell < 0$, then $\psi\tau\ ((\theta^m\theta')[|\ell|:]) = \mathsf{com}^{u[1:j]}_{S_0,E}\gamma$ and $\eta \sqsubseteq (\theta^m\theta')[|\ell|{+}1:] \sqsubseteq \theta^\omega[\ell{+}1:]$. But since $\eta \sqsubseteq \pi^\omega$ and $|\eta| \geqslant \Omega!$ (it is long enough to exhibit a repetition of π), we conclude that $\pi = \theta^\omega[|\ell|{+}1 : |\ell|{+}1{+}\Omega!]$.

In both cases, we conclude that $\mathsf{com}^{u[1:j]}_{S_0,E}\gamma\pi^\omega = \psi\ \tau\ \theta^\omega$. ◀

Thanks to Claim 10.47, all the invariants of Section 10.5.2.1 hold when S_i is replaced by S'_i. Therefore the end of the construction for dealing with the step $S'_i, u[i{+}1], S_{i+1}$ (i.e. adding extra vertical slices and modifying the buffers) can be done as in Section 10.5.2.1.

10.5.3 Correctness of the construction

The goal of this section is to show that the output produced by the $1\mathsf{DT}^\omega$ verifies the properties required for Proposition 10.44. This output is indeed a sequence of (τ, θ)-tree. It remains to show that the concatenation of the values is indeed $f(u)$ and that if an infinite tree was built, it is fertile.

We first reformulate the fact that $\mathscr{T}$ was parallel productive (Definition 10.10) to obtain Claim 10.50. This key result roughly states that all the infinite initial runs of $\mathscr{T}$ have an infinite output.

Claim 10.50 (Infinite output)

Let $u \in \mathsf{Dom}(f)$ and $q_0 \xrightarrow{u[1]|\alpha_1} q_1 \xrightarrow{u[1]|\alpha_2} \cdots$ be an infinite initial run of $\mathscr{T}$ labelled by u (which is not necessarily accepting). Then $\alpha_1\alpha_2\cdots = f(u)$. In particular, this output is infinite.

Proof. Let $p_0 \xrightarrow{u[1]|\beta_1} p_1 \xrightarrow{u[1]|\beta_2} \cdots$ be the (necessarily unique) accepting run of $\mathscr{T}$ labelled by u. Observe that $I, u[1{:}i], \{p_i, q_i\}$ is a pre-step for all $i \geqslant 0$ and therefore $\alpha_1 \cdots \alpha_i$ and $\beta_1 \cdots \beta_i$ are mutual prefixes by Lemma 10.20. It remains to show that $\alpha_1\alpha_2 \cdots \in B^\omega$. By the pigeonhole principle, there exist an infinite sequence $0 \leqslant i_1 < i_2 < \ldots$ and states $p \in F$ and $q \in Q$ such that $p_{i_j} = p$ and $q_{i_j} = q$ for all $j \geqslant 0$. Since $\mathscr{T}$ is parallel productive (recall Definition 10.10), then $\alpha_{i_j+1} \cdots \alpha_{i_{j+1}} \neq \varepsilon$ for all $j \geqslant 0$. The result follows immediately. ◀

As a consequence, we observe that well-chosen common parts always converge to the output of $\mathscr{T}$.

Claim 10.51 (Increasing common part)

Let $u \in \mathsf{Dom}(f)$ and $S_0u[1]S_1 \cdots := \mathsf{buildSteps}(u)$. Then $\mathsf{com}^{u[1:i]}_{S_0,S_i} \to f(u)$.

Proof. For all $i \geqslant 0$ let $R_i := \bigcap_{j \geqslant i} \mathsf{pred}^{u[i:j]}_{S_i,S_j}(S_j)$. Intuitively, R_i contains the states of S_i which have a future run reaching S_j for all $j \geqslant i$. It follows from König's lemma that $q \in R_i$ if and only if there exists an infinite run $q \xrightarrow{u[i+1]} p_{i+1} \xrightarrow{u[i+2]} \cdots$ such that $p_j \in S_j$ for all $j \geqslant i{+}1$.

From this characterization, we deduce that $\mathsf{pred}^{u[i+1]}_{S_i,S_{i+1}}(R_{i+1}) = R_i$ for all $i \geqslant 0$. Therefore the pred relation over the R_i for $i \geqslant 0$ describes an infinite tree of bounded width, where each node has at least one child. In other words, it describes a finite number of infinite initial runs of $\mathscr{T}$. Therefore by Claim 10.50 we deduce that $\mathsf{com}^{u[1:i]}_{S_0,R_i} \to f(u)$.

We observe that for all $i \geqslant 0$, there exists $j \geqslant i$ such that $\mathsf{pred}^{u[i+1:j]}_{S_i,S_j}(S_j) = R_i$. Indeed, since $S_j, u[j{+}1], S_{j+1}$ is a pre-step for all $j \geqslant i$, then $\mathsf{pred}^{u[i+1:j]}_{S_i,S_j}(S_j) \supseteq \mathsf{pred}^{u[i+1:j+1]}_{S_i,S_{j+1}}(S_{j+1})$. Hence $(\mathsf{pred}^{u[i+1:j]}_{S_i,S_j}(S_j))_{j \geqslant i}$ is ultimately constant, and its limit is R_i.

Now if $j \geqslant i$ is such that $\mathsf{pred}^{u[i+1:j]}_{S_i,S_j}(S_j) = R_i$, we have $\mathsf{com}^{u[1:i]}_{S_0,R_i} \sqsubseteq \mathsf{com}^{u[1:j]}_{S_0,S_j}$. Therefore Claim 10.51 follows from the statements of the two previous paragraphs. ◀

Jump to contents

Now we are ready to show the correctness of our construction, by distinguishing two cases:

- if the output is an infinite sequence of finite trees. In the construction of Section 10.5.2, each time a new (τ, θ)-tree is created, the concatenation ψ of the values of the trees produced so far is (up to a bounded difference) $\mathsf{com}^{u[1:i]}_{S_0,S_i}$ for some $i \geqslant 0$. The result follows from Claim 10.51;
- if the output is a finite sequence which ends with an infinite tree. Let us consider this last infinite (τ, θ)-tree. In the construction of Section 10.5.2, each time an operation is performed on this tree, the current deepest leaves are indexed by the $R \in \mathsf{Rigid}_i$ for some $i \geqslant 0$. Furthermore (with the notations of this section), we maintain the invariant $\psi\alpha_R = \mathsf{com}^{u[1:i]}_{S_0,R}$. In particular, we have $\mathsf{com}^{u[1:i]}_{S_0,S_i} \sqsubseteq \psi\alpha_R$ for all $R \in \mathsf{Rigid}_i$. Since these values tend to $f(u)$, the result follows from Claim 10.51 and from Claim 10.52 which provides an easy reformulation of fertility.

Claim 10.52 (Finite branches in a fertile tree)

An infinite tree whose nodes are labelled by B^* is fertile if and only if for all $N \geqslant 1$, there exists $d \geqslant 1$ such that for all nodes of depth $\geqslant d$, the concatenation of the node labels along the branch which goes from the root to this node is a word of length $\geqslant N$.

Proof. The "if" direction is obvious. Conversely, assume that the tree is fertile, let $N \geqslant 1$ and consider the set S of nodes such that the output along the branch which goes from the root to this node is a word of length $< N$. Since this property is preserved under taking ancestors, S is a sub-tree of the original tree. Observe that S has no infinite branch by definition of fertility, thus it is finite by König's lemma. The result follows. ◀

10.6 Computing θ-trees from (τ, θ)-trees

In Section 10.5 we have shown a first half of Theorem 10.26, by building a sequence of (τ, θ)-trees (recall Definition 10.43) from a sequence of pre-steps. The goal of Section 10.6 is to conclude this proof by building in Proposition 10.53 a sequence of θ-trees from the sequence of (τ, θ)-trees. Theorem 10.26 directly follows since deterministic regular functions are closed under composition (Theorem 9.39).

Proposition 10.53 (Computing θ-trees)

One can build a deterministic regular function[17] $h \colon (\mathsf{Slices} \uplus \{\#\})^\omega \rightharpoonup (\mathsf{Slices} \uplus \{\#\})^\omega$ such that if g is the function of Proposition 10.44, then $\mathsf{buildTrees} := h \circ g \colon (A \uplus \mathsf{Comp})^\omega \rightharpoonup (\mathsf{Slices} \uplus \{\#\})^\omega$ verifies the conditions required for Theorem 10.26.

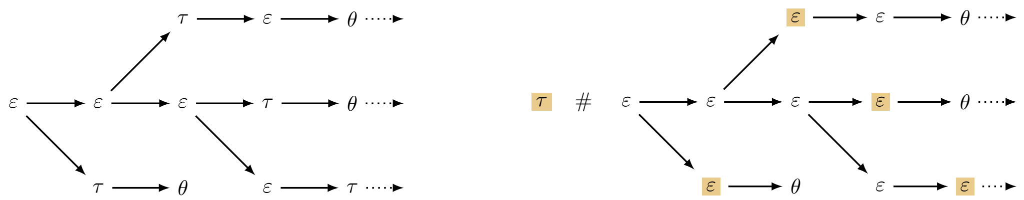
Proof. First, we shall assume that labels of (τ, θ)-tree built by g are only τ, θ or ε (and no longer $\tau\theta^m$ with $m \geqslant 1$ or θ^m with $m \geqslant 2$). This simplification can be achieved by first applying a sequential function which pre-processes the sequence of trees by dividing each vertical slice containing a label $\tau\theta^m$ with $m \geqslant 1$ or θ^m with $m \geqslant 2$ into several vertical slices.

Let $u \in \mathsf{Dom}(f)$. We are ready to describe a $1\mathsf{DT}^\omega$ with finite lookarounds which transforms the sequence $g(\mathsf{buildSteps}(u))$ of (τ, θ)-trees into a sequence of θ-trees. We let h be the function computed by this machine, thanks to Theorem 9.4 it will be deterministic regular. Without loss of generalities (this information could have been encoded in the alphabet Slices in Section 10.5), we assume that when starting to read a (τ, θ)-tree, the machine has access to the values of τ and θ.

[17] We shall in fact build a deterministic rational function (but this precise statement is not useful in our proof).

If $\tau = \varepsilon$, then the current (τ, θ)-tree is already a θ-tree and the $1\mathsf{DT}^\omega$ can output it directly. Now assume that the $1\mathsf{DT}^\omega$ starts reading a (τ, θ)-tree with $\tau \neq \varepsilon$. Because of Claim 10.52 and the properties of $g(\mathsf{buildSteps}(u))$ given by Proposition 10.44, exactly one of the following occurs:

(1) either there exists a depth such that all branches of the (τ, θ)-tree going to this depth meet a label τ. In this case, the $1\mathsf{DT}^\omega$ first outputs a τ-tree with a single node and then transforms the (τ, θ)-tree by removing the τ but keeping the θ, as depicted in Figure 10.54;



(a) Original (τ, θ)-tree.

(b) Equivalent τ-tree and θ-tree (changes are highlighted).

Figure 10.54: Transformation when all branches produce a τ at some point.

(2) or the (τ, θ)-tree is finite, pointy and has value ε. In this case, the $1\mathsf{DT}^\omega$ does not recopy the forthcoming (τ, θ)-tree since it is useless.

The key argument is that the $1\mathsf{DT}^\omega$ can determine whether Item (1) or Item (2) holds by using a finite lookaround. Indeed, it can detect if Item (1) is verified or if the current (τ, θ)-tree is finite. ◀

10.7 Computing the output from θ-trees

The goal of this section is to show Theorem 10.28. Given a sequence of θ-trees which verifies the conditions of Theorem 10.26, we explain how to compute the concatenation of the tree values by a deterministic regular function denoted $\mathsf{buildOutput}$. For this purpose, we shall build a 1-bounded DSST^ω (recall from Theorem 9.13 that such a machine computes a deterministic regular function).

If the input always consists of an infinite sequence of finite trees, the result is easy. Indeed, in this case the DSST^ω can e.g. maintain in its registers the concatenation of the labels along the branches, and output the according value each time a tree is ended (observe that having θ-trees would not be useful). However, this algorithm no longer works when some tree can be infinite, since it would not produce an infinite output in this case. Therefore, we devise a more complex DSST^ω which ensures that θ^ω is produced when reading an infinite fertile θ-tree, while being able to compute the values of the branches.

10.7.1 Information stored by the streaming string transducer

Assume that the $1\mathsf{DT}^\omega$ is reading a vertical slice in Slices which represents a set of nodes N in the current θ-tree. Observe that N is a set of nodes which have the same depth. We define *set chains* as decreasing sequences of subsets of N. This key notion[18] is formalized in Definition 10.55.

Definition 10.55 (Set chain)

A *set chain* is a finite word $C_1 \cdots C_n$ over the alphabet $2^N \smallsetminus \varnothing$ such that $C_1 = N$ and for all $1 \leqslant i \leqslant n{-}1$ we have $\varnothing \neq C_{i+1} \subset C_i$.

We let Chains be the set of all set chains. Since N has bounded size, so does Chains.

[18] Set chains are inspired by the (more complex) *trees of compatibles* in the original proof of [CD22].

10.7.1.1 Information stored by the streaming string transducer. After reading the vertical slice describing N, the DSST^ω keeps track of the following elements:

- the content of its output register out;
- for all $C_1 \cdots C_n \in \mathsf{Chains}$ (recall from Definition 10.55 that $C_1 = N$):
  - a function $\mathsf{buffer}_{C_1 \cdots C_n} : C_n \to \{\varepsilon, \theta\}$ (stored in the finite states);
  - the content of a register $\mathsf{out}_{C_1 \cdots C_n}$. For $n = 1$, we identify the registers out_N and out.

Observe that the information stored in the buffers is bounded since $|\mathsf{Chains}|$ and θ are so. Whenever a configuration of the DSST^ω is clearly fixed, we abuse notations and denote by $\mathsf{out}_{C_1 \cdots C_n}$ the value contained in the register $\mathsf{out}_{C_1 \cdots C_n}$ in this configuration.

10.7.1.2 Invariants maintained by the streaming string transducer. Let $t_1, \dots, t_\ell$ be the finite pointy trees read so far by the DSST^ω and let t be the current θ-tree. Let $\psi_1, \dots, \psi_\ell \in B^*$ be the values of $t_1, \dots, t_\ell$ and $\psi := \psi_1 \cdots \psi_\ell$. For all $\mathfrak{t} \in N$, let $\alpha_{\mathfrak{t}} \in \theta^*$ be the concatenation of the node labels along the branch which goes from the root of t to $\mathfrak{t}$. The following invariants will be preserved:

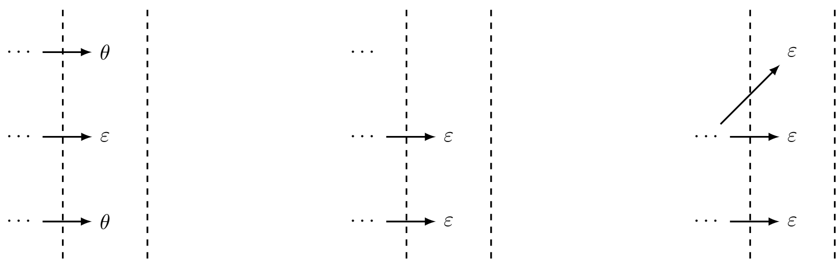
(1) for all $C_1 \cdots C_n \in \mathsf{Chains}$ with $n \geqslant 2$, $\mathsf{out}_{C_1 \cdots C_n} = \theta^m$ for some $m \geqslant 0$;
(2) for all $C_1 \cdots C_n \in \mathsf{Chains}$, there exists $\mathfrak{t} \in C_n$ such that $\mathsf{buffer}_{C_1 \cdots C_n}(\mathfrak{t}) = \varepsilon$;
(3) for all $\mathfrak{t} \in N$ and all $C_1 \cdots C_n \in \mathsf{Chains}$ such that $C_n = \{\mathfrak{t}\}$, we have

$$\psi\, \alpha_{\mathfrak{t}} = \prod_{i=1}^{n} \mathsf{out}_{C_1 \cdots C_i}\ \mathsf{buffer}_{C_1 \cdots C_i}(\mathfrak{t}). \tag{10.56}$$

The main intuition behind Invariant (3) is that the DSST^ω is able to recover $\alpha_{\mathfrak{t}}$ for all $\mathfrak{t} \in N$. Furthermore, for all $N' \subseteq N$, it stores a common prefix of the $\alpha_{\mathfrak{t}}$ for $\mathfrak{t} \in N'$. Observe that all the terms of Equation (10.56) belong to θ^*, except possibly the first one which is $\mathsf{out}_{C_1} = \mathsf{out}_N = \mathsf{out}$. The buffers will be used as a key feature to ensure that the output along an infinite fertile θ-tree is infinite.

10.7.2 Updates of the streaming string transducer

Without loss of generalities (up to first preprocessing the input by applying a sequential function which duplicates each vertical slice), we assume that exactly one of the following holds for all vertical slice:



(a) Bijective parent function. (b) Injective parent function. (c) Surjective parent function.

Figure 10.57: The three possible cases of vertical slices.

(1) either the parent function is bijective (between the current nodes and those of previous depth);
(2) or all nodes have label ε and the parent function is injective;
(3) or all nodes have label ε and the parent function is surjective.

The three cases are depicted in Figures 10.57a to 10.57c. By the same argument, we shall also assume that the root of a θ-tree is always labelled with ε. These assumptions aim at simplifying the description of the transitions of the DSST^{ω}, since it they enable to separately deal with various behaviors.

Let N be the set of nodes which is currently being read by the DSST^{ω} (i.e. the nodes described by the current vertical slice) and Chains be the according set of set chains. We denote by $\overline{N}$ (resp. $\overline{\mathsf{Chains}}$) the set of nodes (resp. of set chains) which was seen in the previous vertical slice.

10.7.2.1 Updates when the parent function is bijective. In this case, the set N can be seen identified with the set $\overline{N}$. For all $\mathfrak{t} \in N$, let $\theta_{\mathfrak{t}} \in \{\varepsilon, \theta\}$ denote the label of $\mathfrak{t}$ in the θ-tree. The main idea is to add this value to $\mathsf{buffer}_N(\mathfrak{t})$. Formally, the DSST^{ω} applies the following updates:

- $\mathsf{buffer}_N(\mathfrak{t}) := \mathsf{buffer}_N(\mathfrak{t})\ \theta_{\mathfrak{t}}$ for all $\mathfrak{t} \in N$;
- $\mathsf{out}_\pi \mapsto \mathsf{out}_\pi$ for all $\pi \in \mathsf{Chains}$ and $\mathsf{buffer}_\pi := \mathsf{buffer}_\pi$ for all $N \neq \pi \in \mathsf{Chains}$.

It is clear that Invariants (1) and (3) hold. However, now we may have $\mathsf{buffer}_N : N \to \{\varepsilon, \theta, \theta^2\}$ (and not necessarily $\mathsf{buffer}_N : N \to \{\varepsilon, \theta\}$) since we have $\theta_{\mathfrak{t}} \in \{\varepsilon, \theta\}$.

To ensure $\mathsf{buffer}_{C_1 \cdots C_n} : C_n \to \{\varepsilon, \theta\}$ for all $C_1 \cdots C_n \in \mathsf{Chains}$, the DSST^{ω} applies the function $\mathsf{spread}(N)$ from Algorithm 10.58. The main intuition concerning this function is that it "sends down" the excessively long buffer_π along the set chains, while using the registers out_π to store greatest common prefixes. Formally, the function $\mathsf{spread}(N)$ adds to out the value $\theta^m := \bigwedge_{\mathfrak{t} \in N} \mathsf{buffer}_N(\mathfrak{t})$ and removes θ^m to each $\mathsf{buffer}_N(\mathfrak{t})$. Then it sends down the values which are still too long. Producing as soon as possible this value θ^m is a key argument to ensure that the output is infinite along an infinite fertile θ-tree.

Algorithm 10.58: Spreading down the values of the buffer_π.

Function $\mathsf{spread}(C_1 \cdots C_n)$
  `/* 1. Add the common prefix` θ^m `to the local output */`
  $\theta^m := \bigwedge_{\mathfrak{t} \in N} \mathsf{buffer}_{C_1 \cdots C_n}(\mathfrak{t})$
  $\mathsf{out}_{C_1 \cdots C_n} \mapsto \mathsf{out}_{C_1 \cdots C_n}\ \theta^m$
  $\mathsf{buffer}_{C_1 \cdots C_n}(\mathfrak{t}) := (\theta^m)^{-1} \mathsf{buffer}_{C_1 \cdots C_n}(\mathfrak{t})$ for all $\mathfrak{t} \in C_n$
  `/* 2. Reduce to` θ `the` $\mathsf{buffer}_\pi(q)$ `which are still too long */`
  for $\mathfrak{t} \in C_n$ **do**
    if $\mathsf{buffer}_{C_1 \cdots C_N}(\mathfrak{t}) = \theta^p$ **with** $p \geqslant 2$ **then**
      for $C_{n+1} \subset C_n$ **such that** $\mathfrak{t} \in C_{n+1}$ **do**
        $\mathsf{buffer}_{C_1 \cdots C_n C_{n+1}}(\mathfrak{t}) := \mathsf{buffer}_{C_1 \cdots C_n C_{n+1}}(\mathfrak{t})\ \theta^{p-1}$
      end
      $\mathsf{buffer}_{C_1 \cdots C_n}(\mathfrak{t}) := \theta$
    end
  end
  `/* 3. Recursive calls */`
  for $\varnothing \neq C_{n+1} \subset C_n$ **do**
    $\mathsf{spread}(C_1 \cdots C_n C_{n+1})$
  end

It is an easy check that executing the function $\mathsf{spread}(N)$ preserves Invariants (1) and (3) and makes Invariant (2) true. It also ensures that $\mathsf{buffer}_{C_1 \cdots C_n} : C_n \to \{\varepsilon, \theta\}$ for all $C_1 \cdots C_n \in \mathsf{Chains}$.

10.7.2.2 Updates when the parent function is injective and all nodes have label ε. In this case, the set N can be seen as a strict subset of $\overline{N}$. With this identification we get $\mathsf{Chains} = \{N C_2 \cdots C_n \mid$

$\overline{N}NC_2 \cdots C_n \in \overline{\mathsf{Chains}}\}$. The DSST^ω applies the following updates:

- out $\mapsto$ out $\mathsf{out}_{\overline{N}N}$ and $\mathsf{buffer}_N(\mathfrak{t}) := \mathsf{buffer}_{\overline{N}}(\mathfrak{t})\ \mathsf{buffer}_{\overline{N}N}(\mathfrak{t})$ for all $\mathfrak{t} \in N$;
- $\mathsf{out}_\pi \mapsto \mathsf{out}_{\overline{N}\pi}$ and $\mathsf{buffer}_\pi := \mathsf{buffer}_{\overline{N}\pi}$ for all $N \neq \pi \in \mathsf{Chains}$.

It is clear Invariants (1) and (3) hold. However, now we may have $\mathsf{buffer}_N : N \to \{\varepsilon, \theta, \theta^2\}$ since it is defined as a concatenation. Therefore, the DSST^ω finally applies $\mathsf{spread}(N)$ to get $\mathsf{buffer}_{C_1 \cdots C_n} : C_n \to \{\varepsilon, \theta\}$ for all $C_1 \cdots C_n \in \mathsf{Chains}$ and to obtain Invariant (2).

10.7.2.3 Updates when the parent function is surjective and all nodes have label ε. This technical case is not especially enlightening, since it just consists in re-shaping the set chains in order to preserve the invariants of Section 10.7.1.2. Let $\sigma\colon N \to \overline{N}$ denote the (surjective) parent function, which naturally extends to a function $\sigma\colon 2^N \to 2^{\overline{N}}$. Since σ is surjective, we have $\sigma(N) = \overline{N}$.

Let $D_1 \cdots D_n \in \mathsf{Chains}$. Let $C_i := \sigma(D_i)$ for all $1 \leqslant i \leqslant n$, then $C_1 = \overline{N}$ since $D_1 = N$. Furthermore, $C_1 \supseteq \cdots \supseteq C_n$ but we may not have $C_1 \cdots C_n \notin \overline{\mathsf{Chains}}$ due to possible equalities[19]. Let $1 = i_1 < \cdots < i_m \leqslant n$ be such that $C_{i_1} = \cdots = C_{i_2 - 1} \supset C_{i_2}$ and so on until $C_{i_m - 1} \supset C_{i_m} = \cdots = C_n$. Observe that $C_{i_1} \cdots C_{i_m} \in \overline{\mathsf{Chains}}$. The DSST^ω performs the updates:

- if $i_m = n$, we let $\mathsf{buffer}_{D_1 \cdots D_n} := \mathsf{buffer}_{C_1 \cdots C_n} \circ \sigma\colon D_n \to \{\varepsilon, \theta\}$ and $\mathsf{out}_{D_1 \cdots D_n} \mapsto \mathsf{out}_{C_1 \cdots C_n}$;
- if $i_m < n$, we let $\mathsf{buffer}_{D_1 \cdots D_n} := 0$ and $\mathsf{out}_{D_1 \cdots D_n} := \varepsilon$.

Let us check that the invariants of Section 10.7.1.2 are preserved. Invariant (1) is trivial. For Invariant (3), let $\mathfrak{t} \in N$ and $D_1 \cdots D_n \in \mathsf{Chains}$ be such that $D_n = \{\mathfrak{t}\}$. Let $C_i := \sigma(D_i)$ for all $1 \leqslant i \leqslant n$ and let $\rho := C_{i_1} \cdots C_{i_m}$ as above. Observe that $C_{i_m} = \{\mathfrak{t}\}$. It is sufficient to show that:

$$\prod_{j=1}^{m} \mathsf{out}_{C_{i_1} \cdots C_{i_j}}\ \mathsf{buffer}_{C_{i_1} \cdots C_{i_j}}(\sigma(\mathfrak{t})) = \prod_{i=1}^{n} \mathsf{out}_{D_1 \cdots D_i}\ \mathsf{buffer}_{D_1 \cdots D_i}(\mathfrak{t}).$$

This equation follows by observing that for all $1 \leqslant i \leqslant n$, we have:

- if $i \notin \{i_1, \dots, i_m\}$, $\mathsf{out}_{D_1 \cdots D_i} = \varepsilon$ and $\mathsf{buffer}_{D_1 \cdots D_i} = 0$;
- if $i = i_j$ with $1 \leqslant j \leqslant i$, $\mathsf{out}_{D_1 \cdots D_i} = \mathsf{out}_{C_{i_1} \cdots C_{i_j}}$ and $\mathsf{buffer}_{D_1 \cdots D_i}(\mathfrak{t}) = \mathsf{buffer}_{C_{i_1} \cdots C_{i_j}}(\sigma(\mathfrak{t}))$.

We finally apply $\mathsf{spread}(N)$ to enforce Invariant (2)

10.7.2.4 Updates when reading a new tree. If the $\mathsf{1DT}^\omega$ starts reading a new θ-tree, it means that the previous tree was finite and pointy. Therefore $\overline{N}$ was a singleton $\{\mathfrak{t}\}$ and $\mathsf{buffer}_{\overline{N}}(\mathfrak{t}) = \varepsilon$ by Invariant (2). Hence by Invariant (3) we get $\psi\alpha_{\mathfrak{t}} =$ out, i.e. the output concatenates the values of the trees. Finally, since the root of the new tree is labelled with ε, it suffices to create an empty buffer.

10.7.3 Correctness of the construction

To conclude the proof of Theorem 10.28, one has to justify that the DSST^ω described so far is indeed 1-bounded and that it produces the concatenation of the values of the θ-trees.

10.7.3.1 Boundedness of the transducer. To show that the DSST^ω is 1-bounded, we claim that the following invariant are maintained along the computation. Let N be the set of nodes of the vertical slice which is currently being read and Chains the according set of set chains. Then for all previous vertical slice with nodes N' and set chains Chains', if s denotes the substitution applied by the DSST^ω when reading the factor of the input between these two slices, we have the following:

[19]There may exists distinct C_i such that $\sigma(C_i)$ is the same.

- for all $\pi \in \mathsf{Chains}$ and $\pi' \in \mathsf{Chains}'$, $\mathsf{out}_{\pi'}$ occurs at most once in $s(\mathsf{out}_\pi)$;
- for all $\pi \sqsubset \rho \in \mathsf{Chains}$ and $\pi' \in \mathsf{Chains}'$, $\mathsf{out}_{\pi'}$ does not occur both in $s(\mathsf{out}_\pi)$ and $s(\mathsf{out}_\rho)$.

The fact that $\mathscr{S}$ is 1-bounded follows (up to trimming the DSST^ω) from the first item.

10.7.3.2 Output produced. Section 10.7.2.4 ensures that when the end of a finite pointy tree is met, the content of out is the concatenation of the values of the trees seen so far. Thus, if the input consists of an infinite sequence of finite pointy trees, the DSST^ω produces the concatenation of their values.

It remains to justify that when reading an infinite fertile θ-tree, the DSST^ω produces θ^ω. Let us consider such a tree. For all $i \geqslant 1$, we let N_i denote the set of nodes of depth i (recall that the N_i corresponds to the sets of nodes described by the vertical slices). We assume by contradiction that there exists $j \geqslant 1$ such that the DSST^ω no longer modifies out after reading N_j (i.e. the updates are systematically out $:=$ out during the rest of the computation). We show the key Claim 10.59.

Claim 10.59 (Empty output + empty buffer ⇒ empty input + empty buffer)

Let $i \geqslant j$ and $\mathfrak{t} \in N_i$ be such that $\mathsf{buffer}_{N_i}(\mathfrak{t}) = \varepsilon$ after reading N_i. Let $\bar{\mathfrak{t}}$ be the ancestor of $\mathfrak{t}$ which belongs to N_j, then after reading N_j we had $\mathsf{buffer}_{N_j}(\bar{\mathfrak{t}}) = \varepsilon$. Furthermore, the branch of the θ-tree from $\bar{\mathfrak{t}}$ to $\mathfrak{t}$ has empty labels.

Proof. By induction it is sufficient to show the result for $j = i-1$, i.e. for the update described in Section 10.7.2. We re-use the notations of this section and let $\overline{N} := N_{i-1}$ and $N := N_i$.

We first observe that if $\mathsf{buffer}_N(\mathfrak{t}) = \varepsilon$ holds right after applying $\mathsf{spread}(N)$ and if nothing was added to out during its execution, then we also had $\mathsf{buffer}_N(\mathfrak{t}) = \varepsilon$ before applying this function. The rest of the proof thus only depends on the rest of the update:

- for Section 10.7.2.1 (bijective parent function) the update is $\mathsf{buffer}_N(\mathfrak{t}) := \mathsf{buffer}_{\overline{N}}(\bar{\mathfrak{t}})\,\theta_{\mathfrak{t}}$ which means that we had both $\mathsf{buffer}_{\overline{N}}(\bar{\mathfrak{t}}) = \varepsilon$ and $\theta_{\mathfrak{t}} = \varepsilon$;
- for Section 10.7.2.2, the update is $\mathsf{buffer}_N(\mathfrak{t}) := \mathsf{buffer}_{\overline{N}}(\bar{\mathfrak{t}})\,\mathsf{buffer}_{\overline{N}N}(\bar{\mathfrak{t}})$ which means that we had $\mathsf{buffer}_{\overline{N}}(\bar{\mathfrak{t}}) = \varepsilon$ (and the node label is empty by hypothesis);
- for Section 10.7.2.3, the update is $\mathsf{buffer}_N(\mathfrak{t}) := \mathsf{buffer}_{\overline{N}}(\bar{\mathfrak{t}})$ and the argument the same. ◀

Now let $L \geqslant 0$ be the maximal length of the concatenation of the node labels along the branches which go from the root to the nodes of N_j. For all $i \geqslant j$, by Invariant (2) there exists $\mathfrak{t} \in N_i$ such that $\mathsf{buffer}_{N_i}(\mathfrak{t}) = \varepsilon$ after reading N_i. Hence by Claim 10.59 the concatenation of the node labels along the branch which goes from the root to $\mathfrak{t}$ has length at most L. This result contradicts Claim 10.52.

10.8 Discussion: uniformly continuous rational functions

The author is not aware of an easy way to generalize the proof of Theorem 10.1 to show Conjecture 8.46. In this section, we discuss a generalization for studying the subclass of rational functions which are *uniformly continuous*. The latter seems to be more affordable.

Formally, we say that the function f is *uniformly continuous* if for all $N \geqslant 0$, there exists $M \geqslant 0$ such that if $u, v \in \mathsf{Dom}(f)$ with $|u \wedge v| \geqslant M$, then $|f(u) \wedge f(v)| \geqslant N$.

Example 10.60 (Uniformly continuous functions)

The function double is uniformly continuous. More generally, any total continuous function of type $A^\omega \to B^\omega$ is uniformly continuous since (A^ω, d) is a *compact*[20] topological space.

Let us note that not all sequential functions are continuous.

Example 10.61 (Non-uniform continuity)

The function remove is continuous but not uniformly continuous. Indeed, for all $n \geqslant 0$ we have $\mathsf{remove}(a^n b^\omega) \wedge \mathsf{remove}(a^n (cb)^\omega) = \varepsilon$.

It is known since [Pri01, Corollary 7] that uniform continuity of rational functions can be decided. A characterization in terms of twinning properties can be obtained by dropping the condition $q_1' \in F$ in Lemma 10.8. This decidability result was extended to regular functions in [DFKL20, Theorem 16].

In Chapter 10, we have shown that a continuous rational function can be extended to a deterministic regular one, which can be computed by a copyless DSST^ω by Theorem 9.13. In other words, such a function is computed by a copyless DSST^ω which produces a non-empty output *infinitely often*. We suspect in Conjecture 10.63 that uniform continuity can be captured by copyless DSST^ω which produce a non-empty output *periodically often*. This intuition is formalized through the concept of *productivity*.

Definition 10.62 (Productive streaming string transducer)

A copyless DSST^ω $\mathscr{S} = (A, B, Q, q_0, F, \delta, \mathfrak{R}, \mathsf{out}, \iota, \lambda)$ is *productive* if there exists $K \geqslant 0$ such that for all $q \in Q$ and $u \in A^*$ with $|u| \geqslant K$, $|\lambda^*(q,u)(\mathsf{out})|_B > 0$.

Note that productivity can be decided by looking at the loops of the DSST^ω.

Conjecture 10.63 (Uniformly continuous)

A rational function is uniformly continuous if and only if it can be computed by a productive copyless DSST^ω. The conversion is effective.

Proving Conjecture 10.63 may be useful for practical applications, since it would enable to build a machine which does not spend arbitrary long times reading inputs without providing outputs. In other words, it is close to a *Mealy machine*, which is one of the basic ingredients for *reactive synthesis*.

Now, let us briefly substantiate Conjecture 10.63. The main idea is to follow step by step the proof presented in Chapter 10 for continuous functions. First, as mentioned above, one can adapt Lemma 10.8 to the setting of uniform continuity. Then, one can show that a $\mathsf{1NT}^\omega$ computing a uniformly continuous function can be transformed into a productive one (meaning that it has no loops with output ε). For the function buildSteps (Theorem 10.22), we replace the notion of compatible set by the (weaker) notion of *weakly compatible set*. It is obtained by dropping all final conditions about runs.

Definition 10.64 (Weakly compatible set)

We say that a subset $C \subseteq Q$ is *weakly compatible* whenever there exists $v \in A^\omega$ such that for all $q \in C$, there exists an infinite run ρ_q labelled by v which starts in state q.

The author believes that the rest of the proof can be adapted accordingly.

[20] Recall that *metric space* is *compact* if one can extract a convergent sub-sequence from any sequence. Given a sequence $(v_n)_{n\geqslant 0} \in (A^\omega)^\mathbb{N}$, one can extract a convergent sub-sequence by considering the set $\{u \in A^* \mid u \sqsubseteq v_n \text{ infinitely often}\}$ which must be infinite. It is prefix-closed and therefore can be seen as a tree. We conclude by applying König's lemma.

Outlook

> Il faut que tout finisse… J'ai joué comme un enfant autour d'une chose que je ne soupçonnais pas… J'ai joué en rêve autour des pièges de la destinée…
>
> Maurice Maeterlinck, *Pelléas et Mélisande*

The various results of this manuscript provide a deeper understanding of the celebrated two-way transducer model, both over finite and infinite words. Apart from membership procedures in themselves, the proof techniques (from the high-level sketches to the detailed constructions) are also worth being put in the spotlight. Indeed, they provide a large toolbox for studying other problems.

Future research programs. The author believes that several problems such as Conjectures 4.56, 8.25, 8.38, 10.5 and 10.63 are rather easy[1] to solve by adapting the techniques developed of this manuscript. Beyond these low-hanging fruits, broader research questions are raised by this work:

- most of the proofs in this manuscript do not build a canonical object for solving class membership problems. The author believes that it is worth trying to decide star-freeness of regular functions (Open question 7.5) without looking for canonical two-way transducers or canonical streaming string transducers. As suggested in Section 7.7, the strategy would be to describe forbidden combinatorial patterns in any transducer which computes a star-free function, and use star-free definable variants of factorization forests in order to build an aperiodic machine whenever it exists. If this technique proves successful, it will provide a versatile tool for various problems;
- the main result[2] of Chapter 10 strongly substantiates the conjecture that deterministic regular functions are exactly the class of continuous regular functions of infinite words (Conjecture 8.46). The author calls for a substantial research effort in this direction, since proving this conjecture would be a major and meaningful achievement in the theory of transductions of infinite words. In this setting, he hopes that leveraging the techniques of Chapter 10 can be helpful;
- towards a practical use of the membership and optimization procedures presented in this manuscript, it is also interesting to study and improve their computational complexity. Obtaining small complexity bounds is often rather technical (see e.g. the involved constructions of [BGMP18] when studying the membership problem from regular functions of finite words to rational functions), but it is a necessary step towards an implementation of the procedures.

[1] Formally, "rather easy" means that each of these research questions is good for the research internship of a master student.
[2] The fact that continuous rational functions are deterministic regular, up to considering function extensions.

Index

List of Figures

Bibliography

[AC10] Rajeev Alur and Pavol Cerný. Expressiveness of streaming string transducers. In *30th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2010*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010. `https://doi.org/10.4230/LIPIcs.FSTTCS.2010.1`.

[AFR14] Rajeev Alur, Adam Freilich, and Mukund Raghothaman. Regular combinators for string transformations. In *39th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2014*. Association for Computing Machinery, 2014. `http://arxiv.org/abs/1402.3021`.

[AFT12] Rajeev Alur, Emmanuel Filiot, and Ashutosh Trivedi. Regular transformations of infinite strings. In *27th Annual IEEE Symposium on Logic in Computer Science,LICS 2012*. IEEE Computer Society, 2012. `https://doi.org/10.1109/LICS.2012.18`.

[AHU69] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. A general theory of translation. *Mathematical Systems Theory*, 3(3):193–221, 1969. `https://doi.org/10.1007/BF01703920`.

[AK20] Shaull Almagor and Orna Kupferman. Good-enough synthesis. In *32nd International Conference on Computer Aided Verification, CAV 2020*, pages 541–563. Springer, 2020. `https://arxiv.org/abs/2109.03594`.

[Bas17] Félix Baschenis. *Minimizing resources for regular word transductions*. PhD thesis, University of Bordeaux, France, 2017. `https://tel.archives-ouvertes.fr/tel-01680357`.

[BC00] Marie-Pierre Béal and Olivier Carton. Determinization of transducers over infinite words. In *27th International Colloquium on Automata, Languages, and Programming, ICALP 2000*. Springer, 2000. `https://doi.org/10.1007/3-540-45022-X_47`.

[BC02] Marie-Pierre Béal and Olivier Carton. Determinization of transducers over finite and infinite words. *Theoretical computer science*, 289(1):225–251, 2002. `https://doi.org/10.1016/S0304-3975(01)00271-7`.

[BC04] Marie-Pierre Béal and Olivier Carton. Determinization of transducers over infinite words: the general case. *Theory of Computing Systems*, 37(4):483–502, 2004. `https://doi.org/10.1007/s00224-003-1014-9`.

[BCPS03] Marie-Pierre Béal, Olivier Carton, Christophe Prieur, and Jacques Sakarovitch. Squaring transducers: an efficient procedure for deciding functionality and sequentiality. *Theoretical Computer Science*, 292(1):45–63, 2003. `https://doi.org/10.1016/S0304-3975(01)00214-6`.

[BDK18] Mikołaj Bojańczyk, Laure Daviaud, and Shankara Narayanan Krishna. Regular and first-order list functions. In *33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018*. Association for Computing Machinery, 2018. `http://arxiv.org/abs/1803.06168`.

[BDSW17] Michael Benedikt, Timothy Duff, Aditya Sharad, and James Worrell. Polynomial automata: Zeroness and applications. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017*. Association for Computing Machinery, 2017. `https://doi.org/10.1109/LICS.2017.8005101`.

[Bel05] Jason P. Bell. A gap result for the norms of semigroups of matrices. *Linear algebra and its applications*, pages 101–110, 2005. `https://doi.org/10.1016/j.laa.2004.12.007`.

[Ber13] Jean Berstel. *Transductions and context-free languages*. Springer-Verlag, 2013.

[BGMP18] Félix Baschenis, Olivier Gauwin, Anca Muscholl, and Gabriele Puppis. One-way definability of two-way word transducers. *Logical Methods in Computer Science*, 14, 2018. `https://doi.org/10.23638/LMCS-14(4:22)2018`.

[BKL19] Mikolaj Bojańczyk, Sandra Kiefer, and Nathan Lhote. String-to-string interpretations with polynomial-size output. In *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. `http://arxiv.org/abs/1905.13190`.

[Boj09] Mikołaj Bojańczyk. Factorization forests. In *13th International Conference on Developments in Language Theory, DLT 2009*. Springer, 2009. `http://doi.org/10.1007/978-3-642-02737-6_1`.

[Boj14] Mikołaj Bojańczyk. Transducers with origin information. In *41th International Colloquium on Automata, Languages, and Programming, ICALP 2014*. Springer, 2014. `http://arxiv.org/abs/1309.6124`.

[Boj18] Mikołaj Bojańczyk. Polyregular functions. *arXiv preprint*, 2018. `http://arxiv.org/abs/1810.08760`.

[Boj19] Mikołaj Bojańczyk. The hilbert method for transducer equivalence. *ACM SIGLOG News*, 6(1):5–17, 2019. `https://doi.org/10.1145/3313909.3313911`.

[Boj22] Mikołaj Bojańczyk. Transducers of polynomial growth. In *37th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2022*, pages 1–27. Association for Computing Machinery, 2022. `https://doi.org/10.1145/3531130.3533326`.

[Boj23a] Mikołaj Bojańczyk. Folding interpretations. In *38th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2023*. Association for Computing Machinery, 2023. `https://doi.org/10.48550/arXiv.2301.05101`.

[Boj23b] Mikołaj Bojańczyk. On the growth rate of polyregular functions. In *38th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2023*. Association for Computing Machinery, 2023. `https://doi.org/10.48550/arXiv.2212.11631`.

[BP09] Mário JJ Branco and Jean-Éric Pin. Equations defining the polynomial closure of a lattice of regular languages. In *36th International Colloquium on Automata, Languages, and Programming, ICALP 2009*. Springer, 2009. `http://doi.org/10.1007/978-3-642-02930-1_10`.

[BR11] Jean Berstel and Christophe Reutenauer. *Noncommutative rational series with applications*. Cambridge University Press, 2011.

[BR18] Nicolas Baudru and Pierre-Alain Reynier. From two-way transducers to regular function expressions. In *22nd International Conference on Developments in Language Theory, DLT 2018*. Springer, 2018. https://doi.org/10.1007/978-3-319-98654-8_8.

[BS20] Mikołaj Bojańczyk and Rafał Stefański. Single-use automata and transducers for infinite alphabets. In *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020. https://doi.org/10.4230/LIPIcs.ICALP.2020.113.

[BSSS06] Mikołaj Bojańczyk, Mathias Samuelides, Thomas Schwentick, and Luc Segoufin. Expressive power of pebble automata. In *33rd International Colloquium on Automata, Languages, and Programming, ICALP 2006*. Springer, 2006. https://doi.org/10.1007/11786986_15.

[Büc60] Julius Richard Büchi. Weak second-order arithmetic and finite automata. *Mathematical Logic Quarterly*, 6(1-6):66–92, 1960. https://doi.org/10.1002/malq.19600060105.

[Büc62] Julius Richard Büchi. On a decision method in restricted second-order arithmetic. In *International Congress on Logic, Mathematics, and Philosophy of Science*. Standford University Press, 1962. https://doi.org/10.1007/978-1-4613-8928-6_23.

[Car10] Olivier Carton. Right-sequential functions on infinite words. In *5th International Computer Science Symposium in Russia, CSR 2010*. Springer, 2010. https://doi.org/10.1007/978-3-642-13182-0_9.

[Car14] Olivier Carton. *Langages formels: calculabilité et complexité*. Vuibert, 2014.

[CCP17] Michaël Cadilhac, Olivier Carton, and Charles Paperman. Continuity and rational functions. In *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. http://arxiv.org/abs/1802.10555.

[CD15] Olivier Carton and Luc Dartois. Aperiodic two-way transducers and fo-transductions. In *24th EACSL Annual Conference on Computer Science Logic, CSL 2015*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. https://arxiv.org/abs/2103.15651.

[CD22] Olivier Carton and Gaëtan Douéneau-Tabot. Continuous rational functions are deterministic regular. In *47th International Symposium on Mathematical Foundations of Computer Science, MFCS 2022*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. https://doi.org/10.48550/arXiv.2204.11235.

[CDFW23] Olivier Carton, Gaëtan Douéneau-Tabot, Emmanuel Filiot, and Sarah Winter. Deterministic regular functions of infinite words. In *50th International Colloquium on Automata, Languages, and Programming, ICALP 2023*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. https://doi.org/10.48550/arXiv.2302.06672.

[CDL23] Thomas Colcombet, Gaëtan Douéneau-Tabot, and Aliaume Lopez. Z-polyregular functions. In *38th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2023*. Association for Computing Machinery, 2023. https://doi.org/10.48550/arXiv.2207.07450.

[CG99] Christian Choffrut and Serge Grigorieff. Uniformization of rational relations. In *Jewels are Forever: Contributions on Theoretical Computer Science in Honor of Arto Salomaa*, pages 59–71. Springer, 1999. https://doi.org/10.1007/978-3-642-60207-8_6.

[Cho77] Christian Choffrut. Une caractérisation des fonctions séquentielles et des fonctions sous-séquentielles en tant que relations rationnelles. *Theoretical Computer Science*, 5(3):325–337, 1977. https://doi.org/10.1016/0304-3975(77)90049-4.

[Cho03] Christian Choffrut. Minimizing subsequential transducers: a survey. *Theoretical Computer Science*, 292(1):131–144, 2003. https://doi.org/10.1016/S0304-3975(01)00219-5.

[Cho17] Christian Choffrut. An hadamard operation on rational relations. *Theoretical Computer Science*, 664:78–90, 2017. https://doi.org/10.1016/j.tcs.2015.10.042.

[CJ77] Michal P. Chytil and Vojtěch Jákl. Serial composition of 2-way finite-state transducers and simple programs on strings. In *4th International Colloquium on Automata, Languages, and Programming, ICALP 1977*, pages 135–147. Springer, 1977. https://doi.org/10.1007/3-540-08342-1_11.

[CK86] Karel Culík II and Juhani Karhumäki. The equivalence of finite valued transducers (on hdt0l languages) is decidable. *Theoretical Computer Science*, 47:71–84, 1986. https://doi.org/10.1016/0304-3975(86)90134-9.

[CL11] Arnaud Carayol and Christof Löding. Uniformization in automata theory. In *14th Congress of Logic, Methodology and Philosophy of Science, CLMPST 2011*, 2011.

[CM03] Olivier Carton and Max Michel. Unambiguous büchi automata. *Theoretical Computer Science*, 297(1-3):37–81, 2003. https://doi.org/10.1016/S0304-3975(02)00618-7.

[Col07] Thomas Colcombet. A combinatorial theorem for trees. In *34th International Colloquium on Automata, Languages, and Programming, ICALP 2007*. Springer, 2007. https://doi.org/10.1007/978-3-540-73420-8_77.

[Col10] Thomas Colcombet. Factorization forests for infinite words and applications to countable scattered linear orderings. *Theoretical Computer Science*, 411(4-5):751–764, 2010. https://doi.org/10.1016/j.tcs.2009.10.013.

[Col11] Thomas Colcombet. Green's relations and their use in automata theory. In *5th International Conference on Language and Automata Theory and Applications, LATA 2011*. Springer, 2011. https://doi.org/10.1007/978-3-642-21254-3_1.

[CP81] Karel Culík II and Jan K. Pachl. Equivalence problems for mappings on infinite strings. *Information and Control*, 49(1):52–63, 1981. https://doi.org/10.1016/S0019-9958(81)90444-7.

[CP17] Thomas Colcombet and Daniela Petrisan. Automata minimization: a functorial approach. In *7th Conference on Algebra and Coalgebra in Computer Science, CALCO 2017*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. http://arxiv.org/abs/1711.03063.

[CvGM22] Thomas Colcombet, Sam van Gool, and Rémi Morvan. First-order separation over countable ordinals. In *25th International Conference on Foundations of Software Science and Computation Structures, FoSSaCS 2022*. Springer, 2022. https://arxiv.org/abs/2201.03089.

[DFG20] Gaëtan Douéneau-Tabot, Emmanuel Filiot, and Paul Gastin. Register transducers are marble transducers. In *45th International Symposium on Mathematical Foundations of Computer Science, MFCS 2020*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. https://arxiv.org/abs/2005.01342.

[DFJL17] Luc Dartois, Paulin Fournier, Ismaël Jecker, and Nathan Lhote. On reversible transducers. In *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. http://arxiv.org/abs/1702.07157.

[DFKL20] Vrunda Dave, Emmanuel Filiot, Shankara Narayanan Krishna, and Nathan Lhote. Synthesis of computable regular functions of infinite words. In *31st International Conference on Concurrency Theory, CONCUR 2020*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020. `https://doi.org/10.4230/LIPIcs.CONCUR.2020.43`.

[DFL18] Luc Dartois, Emmanuel Filiot, and Nathan Lhote. Logics for word transductions with synthesis. In *33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018*. Association for Computing Machinery, 2018.

[DG19] Manfred Droste and Paul Gastin. Aperiodic weighted automata and weighted first-order logic. In *44th International Symposium on Mathematical Foundations of Computer Science, MFCS 2019*, pages 76:1–76:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. `http://arxiv.org/abs/1902.08149`.

[DGK18] Vrunda Dave, Paul Gastin, and Shankara Narayanan Krishna. Regular transducer expressions for regular transformations. In *33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018*, pages 315–324. Association for Computing Machinery, 2018. `https://doi.org/10.1145/3209108.3209182`.

[DGK21] Luc Dartois, Paul Gastin, and Shankara Narayanan Krishna. Sd-regular transducer expressions for aperiodic transformations. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021*. IEEE, 2021. `https://arxiv.org/abs/2101.07130`.

[DJR18] Luc Dartois, Ismaël Jecker, and Pierre-Alain Reynier. Aperiodic string transducers. *International Journal of Foundations of Computer Science*, 29(5):801–824, 2018. `https://doi.org/10.1142/S0129054118420054`.

[Dou18] Gaëtan Douéneau-Tabot. On the complexity of infinite advice strings. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. `https://doi.org/10.4230/LIPIcs.ICALP.2018.122`.

[Dou21] Gaëtan Douéneau-Tabot. Pebble transducers with unary output. In *46th International Symposium on Mathematical Foundations of Computer Science, MFCS 2021*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. `https://arxiv.org/abs/2104.14019`.

[Dou22] Gaëtan Douéneau-Tabot. Hiding pebbles when the output alphabet is unary. In *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. `https://arxiv.org/abs/2112.10212`.

[Dou23] Gaëtan Douéneau-Tabot. Pebble minimization: the last theorems. In *26th International Conference on Foundations of Software Science and Computation Structures, FoSSaCS 2023*. Springer, 2023. `https://doi.org/10.48550/arXiv.2210.02426`.

[Dur13] Fabien Durand. Decidability of the hd0l ultimate periodicity problem. *RAIRO Theoretical Informatics and Applications*, 47(2):201–214, 2013. `http://arxiv.org/abs/1111.3268`.

[EH99] Joost Engelfriet and Hendrik Jan Hoogeboom. Tree-walking pebble automata. *Jewels are Forever: Contributions on Theoretical Computer Science in Honor of Arto Salomaa*, pages 72–83, 1999. `https://doi.org/10.1007/978-3-642-60207-8_7`.

[EH01] Joost Engelfriet and Hendrik Jan Hoogeboom. MSO definable string transductions and two-way finite-state transducers. *ACM Transactions on Computational Logic (TOCL)*, 2(2):216–254, 2001. `https://arxiv.org/abs/cs/9906007`.

[EH06] Joost Engelfriet and Hendrik Jan Hoogeboom. Nested pebbles and transitive closure. In *23th International Symposium on Theoretical Aspects of Computer Science, STACS 2006*. Springer, 2006. `http://arxiv.org/abs/cs/0703079`.

[EHS07] Joost Engelfriet, Hendrik Jan Hoogeboom, and Bart Samwel. Xml transformation by tree-walking transducers with invisible pebbles. In *26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2007*. Association for Computing Machinery, 2007. `http://arxiv.org/abs/1809.05730`.

[EHV99] Joost Engelfriet, Hendrik Jan Hoogeboom, and Jan-Pascal Van Best. Trips on trees. *Acta Cybernetica*, 14(1):51–64, 1999. `https://cyber.bibl.u-szeged.hu/index.php/actcybern/article/view/3510`.

[Eil74] Samuel Eilenberg. *Automata, languages, and machines (A)*. Academic press, 1974.

[EIM21] Joost Engelfriet, Kazuhiro Inaba, and Sebastian Maneth. Linear-bounded composition of tree-walking tree transducers: linear size increase and complexity. *Acta Informatica*, 58:95–152, 2021. `http://arxiv.org/abs/1904.09203`.

[Elg61] Calvin C Elgot. Decision problems of finite automata design and related arithmetics. *Transactions of the American Mathematical Society*, 98(1):21–51, 1961. `https://doi.org/10.2307/1993511`.

[EM65] Calvin C. Elgot and Jorge E. Mezei. On relations defined by generalized finite automata. *IBM Journal of Research and Development*, 9(1):47–68, 1965. `https://doi.org/10.1147/rd.91.0047`.

[EM02] Joost Engelfriet and Sebastian Maneth. Two-way finite state transducers with nested pebbles. In *27th International Symposium on Mathematical Foundations of Computer Science, MFCS 2002*. Springer, 2002. `https://doi.org/10.1007/3-540-45687-2_19`.

[Eng81] Joost Engelfriet. Three hierarchies of transducers. *Mathematical Systems Theory*, 15(1):95–125, 1981. `https://doi.org/10.1007/BF01786975`.

[Eng15] Joost Engelfriet. Two-way pebble transducers for partial functions and their composition. *Acta Informatica*, 52(7-8):559–571, 2015. `https://doi.org/10.1007/s00236-015-0224-3`.

[ERS78] Joost Engelfriet, Grzegorz Rozenberg, and Giora Slutzki. Tree transducers, l systems and two-way machines. In *10th Annual ACM Symposium on Theory of Computing, STOC 1978*, pages 66–74, 1978. `https://doi.org/10.1145/800133.804333`.

[FCL10] Charles N. Fischer, Ron K. Cytron, and Richard J. Jr. LeBlanc. *Crafting a compiler*. Addison-Wesley, 2010.

[FGL19] Emmanuel Filiot, Olivier Gauwin, and Nathan Lhote. Logical and algebraic characterizations of rational transductions. *Logical methods in computer science*, 15, 2019. `http://arxiv.org/abs/1705.03726`.

[FGLM18] Emmanuel Filiot, Olivier Gauwin, Nathan Lhote, and Anca Muscholl. On canonical models for rational functions over infinite words. In *38th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2018*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. `https://doi.org/10.4230/LIPIcs.FSTTCS.2018.30`.

[FGRS13] Emmanuel Filiot, Olivier Gauwin, Pierre-Alain Reynier, and Frédéric Servais. From two-way to one-way finite state transducers. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013*. Association for Computing Machinery, 2013. `http://arxiv.org/abs/1301.5197`.

[Fil15] Emmanuel Filiot. Logic-automata connections for transformations. In *6th Indian Conference on Logic and Its Applications, ICLA 2015, Mumbai*. Springer, 2015. `https://doi.org/10.1007/978-3-662-45824-2_3`.

[FKT14] Emmanuel Filiot, Shankara Narayanan Krishna, and Ashutosh Trivedi. First-order definable string transformations. In Venkatesh Raman and S. P. Suresh, editors, *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2014. `http://arxiv.org/abs/1406.7824`.

[FLW20] Emmanuel Filiot, Christof Löding, and Sarah Winter. Synthesis from weighted specifications with partial domains over finite words. In *40th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2020*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. `https://arxiv.org/abs/2103.05550`.

[FR17] Emmanuel Filiot and Pierre-Alain Reynier. Copyful streaming string transducers. In *11th International Workshop on Reachability Problems, RP 2017*. Springer, 2017. `https://doi.org/10.1007/978-3-319-67089-8_6`.

[FT08] Emmanuel Filiot and Sophie Tison. Regular n-ary queries in trees and variable independence. In *5th IFIP International Conference On Theoretical Computer Science, TCS 2008*. Springer, 2008. `https://doi.org/10.1007/978-0-387-09680-3_29`.

[FW21] Emmanuel Filiot and Sarah Winter. Synthesizing computable functions from rational specifications over infinite words. In *41st IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2021*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. `https://arxiv.org/abs/2103.05674`.

[GH96] Noa Globerman and David Harel. Complexity results for two-way and multi-pebble automata and their logics. *Theoretical Computer Science*, 169(2):161–184, 1996. `https://doi.org/10.1016/S0304-3975(96)00119-3`.

[Gin62] Seymour Ginsburg. *An Introduction to Mathematical Machine Theory*. Addison-Wesley, 1962.

[Gir86] Françoise Gire. Two decidability problems for infinite words. *Information Processing Letters*, 22(3):135–140, 1986. `https://doi.org/10.1016/0020-0190(86)90058-X`.

[GR66] Seymour Ginsburg and Gene F. Rose. A characterization of machine mappings. *Canadian Journal of Mathematics*, 18:381–388, 1966. `https://doi.org/10.4153/cjm-1966-040-3`.

[Gur80] Eitan M. Gurari. The equivalence problem for deterministic two-way sequential transducers is decidable. In *21st Annual Symposium on Foundations of Computer Science, FOCS 1980*. IEEE Computer Society, 1980. `https://doi.org/10.1109/SFCS.1980.46`.

[Har72] Juris Hartmanis. On non-determinancy in simple computing devices. *Acta Informatica*, 1(4):336–344, 1972. `https://doi.org/10.1007/BF00289513`.

[HMU07] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Addison-Wesley, 2007.

[Hop71] John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computations*, pages 189–196. Academic Press, 1971. https://doi.org/10.1016/B978-0-12-417750-5.50022-1.

[HU67] John E. Hopcroft and Jeffrey D. Ullman. An approach to a unified theory of automata. In *8th Annual Symposium on Switching and Automata Theory, SWAT 1967*. IEEE Computer Society, 1967. https://doi.org/10.1109/FOCS.1967.4.

[Iba71] Oscar H Ibarra. Characterizations of some tape and time complexity classes of turing machines in terms of multihead and auxiliary stack automata. *Journal of Computer and System Sciences*, 5(2):88–117, 1971. https://doi.org/10.1016/S0022-0000(71)80029-6.

[Jec21] Ismaël Jecker. A ramsey theorem for finite monoids. In *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. https://arxiv.org/abs/2101.05895.

[KA10] Ondrej Klima and Jorge Almeida. New decidable upper bound of the second level in the straubing-thérien concatenation hierarchy of star-free languages. *Discrete Mathematics and Theoretical Computer Science*, 12:41–58, 2010. https://doi.org/10.46298/dmtcs.490.

[Kle56] Stephen Cole Kleene. Representation of events in nerve nets and finite automata. *Automata Studies: Annals of Mathematics Studies*, 34:3–42, 1956. https://doi.org/10.1515/9781400882618-002.

[KNP23] Sandra Kiefer, Lê Thành Dung Nguyên, and Cecilia Pradic. Refutations of pebble minimization via output languages. *arXiv preprint*, 2023. https://doi.org/10.48550/arXiv.2301.09234.

[Knu74] Donald E. Knuth. Structured programming with go to statements. *ACM Computing Surveys*, 6(4):261–301, 1974. https://doi.org/10.1145/356635.356640.

[Kuf08] Manfred Kufleitner. The height of factorization forests. In *33rd International Symposium on Mathematical Foundations of Computer Science, MFCS 2008*. Springer, 2008. http://doi.org/10.1007/978-3-540-85238-4_36.

[Leu00] Rainer Leupers. *Code optimization techniques for embedded processors - methods, algorithms, and tools*. Kluwer, 2000.

[LLN+11] Grégoire Laurence, Aurélien Lemay, Joachim Niehren, Sławek Staworko, and Marc Tommasi. Normalization of sequential top-down tree-to-word transducers. In *5th International Conference on Language and Automata Theory and Applications, LATA 2011*. Springer, 2011. https://doi.org/10.1007/978-3-642-21254-3_28.

[Mas87] Henry Massalin. Superoptimizer - A look at the smallest program. In Randy H. Katz and Martin Freeman, editors, *2nd International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 1987*. Association for Computing Machinery, 1987. doi:10.1145/36206.36194.

[Moo56] Edward F. Moore. Gedanken-experiments on sequential machines. *The Journal of Symbolic Logic*, 34(1):129–153, 1956. https://doi.org/10.1515/9781400882618-006.

[MP71] Robert McNaughton and Seymour A Papert. *Counter-Free Automata (MIT research monograph no. 65)*. The MIT Press, 1971.

[MP19] Anca Muscholl and Gabriele Puppis. The many facets of string transducers (invited talk). In *36th International Symposium on Theoretical Aspects of Computer Science, STACS 2019*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019. https://doi.org/10.4230/LIPIcs.STACS.2019.2.

[MPR08] Mehryar Mohri, Fernando Pereira, and Michael Riley. Speech recognition with weighted finite-state transducers. *Springer Handbook of Speech Processing*, pages 559–584, 2008. https://doi.org/10.1007/978-3-540-49127-9_28.

[MS77] Arnaldo Mandel and Imre Simon. On finite semigroups of matrices. *Theoretical Computer Science*, 5(2):101–111, 1977. https://doi.org/10.1016/0304-3975(77)90001-9.

[MSV00] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for xml transformers. In *9th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2000*. Association for Computing Machinery, 2000. https://doi.org/10.1145/335168.335171.

[Ner58] Anil Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958. https://doi.org/10.2307/2033204.

[NNP21] Lê Thành Dung Nguyên, Camille Noûs, and Cecilia Pradic. Comparison-free polyregular functions. In *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. https://arxiv.org/abs/2105.08358.

[NSV04] Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Transactions on Computational Logic (TOCL)*, 5(3):403–435, 2004. https://doi.org/10.1145/1013560.1013562.

[Pin84] Jean Eric Pin. *Variétés de langages formels*. Masson, 1984.

[Pin17] Jean-Éric Pin. The dot-depth hierarchy, 45 years later. *The Role of Theory in Computer Science - Essays Dedicated to Janusz Brzozowski*, pages 177–201, 2017. https://doi.org/10.1142/9789813148208_0008.

[PP04] Dominique Perrin and Jean-Éric Pin. *Infinite words: automata, semigroups, logic and games*. Elsevier, 2004.

[Pri01] Christophe Prieur. How to decide continuity of rational functions on infinite words. *Theoretical Computer Science*, 250(1-2):71–82, 2001. https://doi.org/10.1016/S0304-3975(99)00115-2.

[Pri02] Christophe Prieur. How to decide continuity of rational functions on infinite words. *Theoretical computer science*, 276(1-2):445–447, 2002. Corrective note of [Pri01]. https://doi.org/10.1016/S0304-3975(01)00307-3.

[PW97] Jean-Éric Pin and Pascal Weil. Polynomial closure and unambiguous product. *Theory of computing systems*, 30:383–422, 1997. http://doi.org/10.1007/BF02679467.

[Roz86] Brigitte Rozoy. Outils et résultats pour les transducteurs boustrophédons. *RAIRO Theoretical Informatics and Applications*, 20(3):221–250, 1986. https://doi.org/10.1051/ita/1986200302211.

[RS59] Michael O. Rabin and Dana Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959. https://doi.org/10.1147/rd.32.0114.

[RS91] Christophe Reutenauer and Marcel-Paul Schützenberger. Minimization of rational word functions. *SIAM Journal on Computing*, 20(4):669–685, 1991. https://doi.org/10.1137/0220042.

[Sak09] Jacques Sakarovitch. *Elements of automata theory*. Cambridge University Press, 2009.

[Sch61a] Marcel-Paul Schützenberger. On the definition of a family of automata. *Information and Control*, 4(2-3):245–270, 1961. https://doi.org/10.1016/S0019-9958(61)80020-X.

[Sch61b] Marcel-Paul Schützenberger. A remark on finite transducers. *Information and Control*, 4(2-3):185–196, 1961. https://doi.org/10.1016/S0019-9958(61)80006-5.

[Sch62] Marcel-Paul Schützenberger. Finite counting automata. *Information and Control*, 5(2):91–107, 1962. https://doi.org/10.1016/S0019-9958(62)90244-9.

[Sch65] Marcel-Paul Schützenberger. On finite monoids having only trivial subgroups. *Information and Control*, 8(2):190–194, 1965. https://doi.org/10.1016/S0019-9958(65)90108-7.

[Sch77] Marcel-Paul Schützenberger. Sur une variante des fonctions séquentielles. *Theoretical Computer Science*, 4(1):47–57, 1977. https://doi.org/10.1016/0304-3975(77)90055-X.

[Sco67] Dana Scott. Some definitional suggestions for automata theory. *Journal of Computer and System Sciences*, 1(2):187–212, 1967. https://doi.org/10.1016/S0022-0000(67)80014-X.

[She59] John C. Shepherdson. The reduction of two-way automata to one-way automata. *IBM Journal of Research and Development*, 3(2):198–200, 1959. https://doi.org/10.1147/rd.32.0198.

[Sim90] Imre Simon. Factorization forests of finite height. *Theoretical Computer Science*, 72(1):65–94, 1990. https://doi.org/10.1016/0304-3975(90)90047-L.

[Sip12] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012.

[Smi14] Tim Smith. A pumping lemma for two-way finite transducers. In *39th International Symposium on Mathematical Foundations of Computer Science, MFCS 2014*. Springer, 2014. https://doi.org/10.1007/978-3-662-44522-8_44.

[Ste67] Richard Edwin Stearns. A regularity test for pushdown machines. *Information and Control*, 11(3):323–340, 1967. https://doi.org/10.1016/S0019-9958(67)90591-8.

[Str94] Howard Straubing. *Finite automata, formal logic, and circuit complexity*. Springer, 1994.

[Tho90] Wolfgang Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 133–191. Elsevier, 1990. https://doi.org/10.1016/b978-0-444-88074-1.50009-3.

[Tho97] Wolfgang Thomas. Languages, automata, and logic. In *Handbook of Formal Languages, Volume 3: Beyond Words*, pages 389–455. Springer, 1997. https://doi.org/10.1007/978-3-642-59126-6_7.

[Tra62] Boris Abramovich Trakhtenbrot. Finite automata and the logic of one-place predicates. *Sibirskii Matematicheskii Zhurnal*, 3(1):103–131, 1962.

[VDKT93] Arie Van Deursen, Paul Klint, and Frank Tip. Origin tracking. *Journal of Symbolic Computation*, 15(5-6):523–545, 1993. https://doi.org/10.1016/S0747-7171(06)80004-0.

[Wei00] Klaus Weihrauch. *Computable analysis: an introduction*. Springer, 2000. https://doi.org/10.1007/978-3-642-56999-9.

[Wil16] Thomas Wilke. Past, present, and infinite future. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. https://doi.org/10.4230/LIPIcs.ICALP.2016.95.

[WK95] Andreas Weber and Reinhard Klemm. Economy of description for single-valued transducers. *Information and Computation*, 118(2):327–340, 1995. https://doi.org/10.1006/inco.1995.1071.

[WS91] Andreas Weber and Helmut Seidl. On the degree of ambiguity of finite automata. *Theoretical Computer Science*, 88(2):325–349, 1991. https://doi.org/10.1016/0304-3975(91)90381-B.