UNIVERSITÉ PARIS-EST

École doctorale MSTIC

Thèse de doctorat

Pour obtenir le titre de

Docteur de l'Université Paris-Est

Spécialité : Informatique

On the Power and Universality of Biologically-inspired Models of Computation

défendue par

Sergiu Ivanov

Directeur de thèse : Serghei Verlan

préparée au LACL

Acknowledgements

It is with enormous pleasure and most wholehearted warmth that I would to thank all the people who helped and supported me during my work on the present thesis and without whom this endeavour would have been entirely inconceivable.

I would like to thank before all my supervisor Serghei Verlan who had gone beyond being an excellent tutor: he became my close working partner and amiable mentor in scientific and life matters. I can barely imagine the amount of effort he had invested in setting up the scientific and social backdrop highly propitious for the emergence of my professional experience, and I am profoundly grateful.

It is my deep sorrow that I cannot meaningfully thank the late professor Yurii Rogozhin for having laid down the foundations of my career, without me even realising it. The brilliance of professor Rogozhin's ideas combined with the jovial and lighthearted manner in which he conveyed them had always impressed me and I can only hope that my own humble work be accepted as a continuation of his fundamental contributions.

I am very grateful to my many colleagues whose ideas gave valuable input to my research and from whom I learned the good practices of scientific work. I would like to thank Ion Petre for helping me enlarge my scientific horizon by suggesting work on various highly interesting topics not directly related with the results exposed in the present thesis. Ion Petre taught me, by his own example, how a team of researchers was to be managed in order to maximise their productivity.

I would like to address a big and cordial "Danke schön!" to Rudolf Freund, who has shown me the ways of abstract and very abstract thinking and who taught me to generalise generously. I owe a hearty "Merci beaucoup !" to Antoine Spicher who, besides having always provided me highly gourmet food for thought, directly contributed to the constitution of the scientifically social me. I am addressing another "Merci beaucoup !" to Luidnel Maignan who showed me a marvellous example of scientific curiosity that I desire to follow. Finally, I say a plain but utterly sincere "Mulţumesc!" to Cristian Gratie, who retains one of the top places in my rating of the most cool-headed and composed researchers I have ever had the pleasure of collaborating with.

I would also like to thank my wonderful friends for their invaluable input and support. I would like to thank Katya and Daniel for the moments of childish joy we have spent together, Vladimir for having let me discover the stunning beauty of the Finnish nature, Ravi for having laughed at my weird jokes and for having then invented other, even weirder ones. I am profoundly grateful to Quentin for his patience with my slow digestion of French and for having taught me to put my tongue out for better social interaction. I would like to thank Prince for the great pains he had gone to in order to have me feel at home.

Finally, I would like to express my profound gratitude to some very special people whose contribution to the present work is difficult to measure. I would like to thank Martin for his forbearance to deal with my basic level of French, for his willingness to share his culture, and for his participation in the many side projects which helped me contain and contour the ideas appearing in the present thesis. I would have also liked to express the astronomical degree of my thankfulness to my dear parents who have very actively supported me during my entire doctoral studies, but the confines of formality shall leave this desire unaccomplished.

I conclude this section by thanking you, my dear reader, for setting your mental feet at the boundary of the world of my thesis, and by wishing you a most pleasurable and sunny journey on the plains of computational completeness and in the forest of universality.

Abstract

The present thesis considers the problems of computational completeness and universality for several biologically-inspired models of computation: insertion-deletion systems, networks of evolutionary processors, and multiset rewriting systems. The presented results fall into two major categories: study of expressive power of the operations of insertion and deletion with and without control, and construction of universal multiset rewriting systems of low descriptional complexity.

Insertion and deletion operations consist in adding or removing a subword from a given string if this subword is surrounded by some given contexts. The motivation for studying these operations comes from biology, as well as from linguistics and the theory of formal languages. In the first part of the present work we focus on insertion-deletion systems closely related to RNA editing, which essentially consists in inserting or deleting fragments of RNA molecules. An important feature of RNA editing is the fact that the locus the operations are carried at is determined by certain sequences of nucleotides, which are always situated to the same side of the editing site. In terms of formal insertion and deletion, this phenomenon is modelled by rules which can only check their context on one side and not on the other. We show that allowing one-symbol insertion and deletion rules to check a two-symbol left context enables them to generate all regular languages. Moreover, we prove that allowing longer insertion and deletion contexts does not increase the computational power. We further consider insertion-deletion systems with additional control over rule applications and show that the computational completeness can be achieved by systems with very small rules.

The motivation for studying insertion-deletion systems also comes from the domain of computer security, for the purposes of which a special kind of insertion-deletion systems called leftist grammars was introduced. In this work we propose a novel graphical instrument for visual analysis of the dynamics of such systems.

The second part of the present thesis is concerned with the universality problem, which consists in finding a fixed element able to simulate the work any other computing device. We start by considering networks of evolutionary processors (NEPs), a computational model inspired by the way genetic information is processed in the living cell, and construct universal NEPs with very few rules. We then focus on multiset rewriting systems, which model the chemical processes running in the biological cell. For historical reasons, we formulate our results in terms of Petri nets. We construct a series of universal Petri nets and give several techniques for reducing the numbers of places, transitions, inhibitor arcs, and the maximal transition degree. Some of these techniques rely on a generalisation of conventional register machines, proposed in this thesis, which allows multiple register checks and operations to be performed in a single state transition.

Résumé

L'objectif de cette thèse est d'étudier la puissance d'expression des modèles de calcul qui ont été inspirés par la biologie. Ces modèles représentent de manière formelle les points définitoires des phénomènes biologiques et décrivent au niveau abstrait les interactions entre leurs entités centrales. Nous nous intéressons principalement à des processus qui ont lieu dans la cellule vivante et nous étudions la complexité des systèmes formels inspirés par son activité.

Les travaux présentés se divisent en deux parties. Dans la première nous examinons les règles d'insertion et d'effacement — il a été montré que le modèle ayant des règles qui ne vérifient le contexte que d'un seul côté de l'endroit de modification correspond à l'édition que certains protozoaires réalisent sur leur ARN. Nous étudions également les systèmes d'insertion/effacement munis de mécanismes de contrôle d'application de règles, ce qui peut représenter les phases des processus modélisés pendant lesquelles des actions différentes s'effectuent.

Dans la deuxième partie nous nous concentrons sur le problème d'universalité pour les systèmes de réécriture de multiensembles — des systèmes formels qui permettent de représenter les réactions chimiques. Pour des raisons historiques, nous décrivons nos résultats sous la forme des réseaux de Petri avec des arcs inhibiteurs, un modèle équivalent à la réécriture de multiensembles. Afin de construire des réseaux de Petri universels de petite taille, nous décrivons également des machines à registres universelles avec un petit nombre de registres, ainsi qu'une généralisation de ce modèle.

Nous rappelons que le problème d'universalité pour une classe de modèles de calcul consiste à trouver un objet, dit universel, qui peut répliquer l'action de n'importe quel autre objet de cette classe, la simulation pouvant éventuellement se faire à un codage près. D'une façon plus formelle, si A_0 est un élément universel dans la classe $\mathfrak{C}$, alors, pour tout autre élément $A \in \mathfrak{C}$, il est vrai que $A(x) = f\big(A_0\big(\langle g(A), h(x)\rangle\big)\big)$, où h est la fonction d'encodage de l'entrée, f est la fonction de décodage de la sortie, g est la fonction qui énumère les éléments de $\mathfrak{C}$ (par exemple, l'énumération de Gödel), et $\langle x, y\rangle$ est une fonction d'appariement, c'est-à-dire une fonction qui associe un nombre unique à toute paire (x, y). Les fonctions d'encodage et de décodage ne doivent pas être trop complexes, car sinon tout le travail de simulation pourrait être fait par ces fonctions et non pas par A_0.

Nous précisons la différence entre l'universalité et la complétude computationnelle. La complétude computationnelle est la propriété d'une classe vérifiée lorsque celle-ci contient, pour tout langage récursivement énumérable, un objet qui l'engendre. Une classe vérifiant cette propriété dispose donc d'une puissance d'expression équivalente à celle des machines de Turing. Puisqu'il existe des machines de Turing universelles, la complétude computationnelle d'une classe implique aussi l'existence

d'un élément universel. L'implication inverse n'est généralement pas vraie : dans une classe à un seul élément cet élément est universel, alors que la classe elle-même n'atteint pas nécessairement la complétude computationnelle.

Le mémoire se compose de cinq chapitres. Le premier chapitre décrit l'état de l'art dans les domaines de référence de la thèse. Le deuxième chapitre recueille les définitions de base et les notations de la théorie des langages formels. Le troisième chapitre étudie la puissance d'expression des systèmes d'insertion/effacement avec tous les contextes du même côté, tout en proposant des outils originaux d'analyse de leur comportement. Dans le quatrième chapitre il s'agit des machines à registres universelles, ainsi que d'une généralisation de ce modèle ; nous décrivons des objets universels dans ces deux classes de modèles de calcul. Finalement, le cinquième chapitre présente plusieurs réseaux de Petri avec des arcs inhibiteurs universels.

Chapitre 3

Ce chapitre étudie la puissance d'expression des opérations d'insertion et d'effacement sans mécanisme de contrôle. Ces deux opérations agissant sur des chaînes de caractères sont bien connues pour leur capacité à engendrer des familles de langages complexes incomparables avec la hiérarchie de Chomsky. De manière intuitive, une règle d'insertion rajoute une sous-chaîne à une chaîne de caractères dans un contexte donné. Une règle d'effacement agit de la façon duale : elle supprime une sous-chaîne d'une chaîne de caractères, dans un contexte donné. L'effet de ces opérations correspond donc aux règles de réécriture de la forme $uv \to uxv$ et $uxv \to uv$. Un système d'insertion/effacement contient un ensemble de règles d'insertion et d'effacement ; il engendre un langage en appliquant ces règles itérativement à partir d'un ensemble fini de mots dits axiomes.

La taille d'un système d'insertion/effacement est décrite par le 6-uplet $(m, n, n'; p, q, q')$, où les trois premiers composants représentent la longueur maximale de la sous-chaîne insérée et la taille maximale des contextes à gauche et à droite, alors que les trois derniers composants décrivent les mêmes paramètres pour les règles d'effacement.

L'inspiration qui a motivé l'introduction de ces opérations vient de la linguistique, car elles semblent modéliser assez précisément les procédés de construction de phrases dans une langue vivante. Une autre source d'inspiration se place dans le domaine de la théorie des langages formels : l'insertion et l'effacement peuvent être vus comme des généralisations de la concaténation et du quotient de langages. De plus, il a été montré récemment que ces opérations formalisent l'hybridation erronée des brins d'ADN *(mismatched DNA annealing)*. D'un autre côté, l'édition de l'ARN *(RNA editing)* réalisée par certains protozoaires consiste généralement en des ajouts et des suppressions dans des brins d'ARN. Il est intéressant de noter que l'édition de l'ARN est guidée par un fragment du brin qui se situe toujours du même côté de l'endroit modifié. Du point de vue formel, cet effet peut être représenté par un système d'insertion/effacement dont toutes les règles ont le contexte d'un seul côté.

L'étude des systèmes d'insertion/effacement puise encore son inspiration dans le domaine de la sécurité informatique. Les grammaires gauches *(leftist grammars)* sont l'un des outils théoriques utilisés pour l'étude des interactions entre les entités et notamment pour l'analyse des propriétés d'accessibilité dans des systèmes informatiques critiques. Ces grammaires contiennent des règles d'insertion et de suppression de taille $(1,0,1;1,0,1)$; puisque les règles de cette taille regroupent deux caractères chacune, elle peuvent représenter des relations binaires.

La puissance d'expression de systèmes d'insertion/effacement a fait l'objet de nombreuses études. Il a été démontré que les systèmes de tailles assez petites — $(2,0,0;1,1,1)$ et $(1,1,1;2,0,0)$, par exemple — engendrent tous les langages récursivement énumérables. Les systèmes des tailles $(2,0,0;3,0,0)$ et $(3,0,0;2,0,0)$ atteignent également la complétude computationnelle. D'un autre côté, il existe des langages récursivement énumérables que les systèmes des tailles $(2,0,0;2,0,0)$ et $(1,1,0;1,1,0)$ (et, par symétrie, les grammaires gauches) ne peuvent pas engendrer. On observe que les systèmes de tailles $(1,1,0;1,1,0)$ ne sont pas capables d'engendrer même certains langages rationnels. Toutefois, il existe des langages non algébriques qui sont engendrés par les systèmes de cette taille.

La section 3.3 se concentre sur les systèmes d'insertion/effacement ayant des règles qui possèdent uniquement le contexte à gauche et qui n'insèrent et n'effacent qu'un seul caractère à la fois, c'est-à-dire les systèmes de taille $(1,n,0;1,q,0)$. Nous considérons d'abord les systèmes de tailles $(1,1,0;1,2,0)$ et $(1,2,0;1,1,0)$ et nous montrons qu'ils engendrent tous les langages rationnels. Nous montrons ensuite que les systèmes de cette taille peuvent engendrer l'intersection du langage produit par un système de taille $(1,1,0;1,1,0)$ avec un langage rationnel. Comme conséquence nous obtenons que les systèmes de tailles $(1,1,0;1,2,0)$ et $(1,2,0;1,1,0)$ peuvent engendrer des langages exponentiels.

Ensuite nous nous intéressons à la relation entre les langages engendrés par les systèmes de tailles $(1,1,0;1,k,0)$ et $(1,k,0;1,1,0)$. Nous démontrons que les familles de langages correspondantes coïncident ; de plus, elles sont identiques à la famille engendrée par les systèmes de taille $(1,k,0;1,k,0)$. Finalement, nous prouvons que tout langage appartenant à cette famille peut être engendré par un système de taille $(1,1,0;1,2,0)$ ou $(1,2,0;1,1,0)$. Ce résultat met en évidence la complexité inhérente des systèmes de cette taille et explique pourquoi l'analyse de leur puissance d'expression est une tâche complexe. Néanmoins, nous supposons que ces systèmes ne peuvent pas engendrer tous les langages récursivement énumérables, puisque l'évolution d'un préfixe est indépendante du suffixe qui lui correspond. Autrement dit, la transmission de l'information dans la chaîne de caractères ne peut se faire que dans un seul sens : de gauche à droite.

Dans la section 3.4 nous reconsidérons les règles d'insertion et d'effacement de taille $(1,1,0;1,1,0)$. Nous proposons un nouvel outil d'analyse graphique des évolutions de ce type de systèmes : les graphes de dérivation. Nous remarquons d'abord que toute règle de taille $(1,1,0)$ inclut deux symboles. Une dérivation d'une grammaire gauche peut donc être vue comme une séquence de paires de symboles représentant les insertions et les effacements effectués lors de cette dérivation. Le graphe de cette dérivation est le graphe qui contient toutes ces paires comme arêtes de deux types : correspondant aux insertions et aux effacements.

Nous remarquons tout de suite que plusieurs dérivations peuvent correspondre au même graphe, car l'ordre d'application des règles n'est représenté que partiellement dans celui-ci. Néanmoins, toutes les dérivations qui correspondent à un seul graphe sont équivalentes, car elles engendrent le même mot. Les graphes de dérivations permettent ainsi de décrire de façon compacte les classes d'équivalence des dérivations d'un système de taille $(1,1,0;1,1,0)$, en mettant de côté les comportements dynamiques insignifiants.

Comme ils constituent un outil de représentation convenable, les graphes de dérivation permettent de raisonner davantage sur les interactions locales de caractères. Nous les utilisons pour illustrer plusieurs constructions connues qui décrivent la puissance d'expression de grammaires gauches, ainsi que pour tirer de nouvelles conclusions concernant la manière par laquelle ces systèmes engendrent des langages situés aux niveaux élevés de la hiérarchie de Chomsky.

Chapitre 4

Le quatrième chapitre poursuit l'étude des opérations d'insertion et d'effacement et considère des systèmes munis de mécanismes de contrôle d'application des règles. Un mécanisme de contrôle est une façon de spécifier les pré-conditions pour l'applicabilité d'une règle et peut être vu comme une modélisation de phases de vie d'un organisme biologique, ou tout simplement d'un changement d'activité en fonction de la période du jour. Dans une perspective théorique, les mécanismes de contrôle sont un moyen d'augmenter la puissance de calcul d'un système sans changer essentiellement de type de règle utilisé. Les mécanismes de contrôle ont été largement étudiés dans le cadre de la réécriture formelle contrôlée *(regulated rewriting).* Dans ce domaine, on se concentre sur l'augmentation de la puissance d'expression des règles non contextuelles équipées de contrôle supplémentaire.

L'un des mécanismes de contrôle les plus généraux est le contrôle par graphe *(graph control)*, qui consiste à étiqueter les règles et à donner ensuite le graphe des étiquettes définissant l'ordonnancement correct de leurs applications. L'étiquetage peut ne pas être bijectif ; le cas échéant, les règles peuvent être groupées en ensembles des règles ayant la même étiquette.

Un autre mécanisme de contrôle bien connu est le contrôle matriciel *(matrix control).* Dans le cadre de ce type de contrôle les règles sont réunies en séquences dites matrices ; appliquer une telle matrice revient à appliquer toutes les règles qu'elle contient, dans l'ordre précisé. Le contrôle matriciel est un cas particulier du contrôle par graphe, car celui-ci permet également d'imposer des séquences d'application de règles.

Un mécanisme de contrôle qui se fonde sur des tests d'occurrence est le contrôle semi-conditionnel *(semi-conditional control).* Ce contrôle consiste à rajouter deux ensembles de mots à chaque règle ; l'un de ces ensembles, appelé le contexte promoteur, contient les mots qui doivent être des sous-chaînes de la chaîne de caractères pour que la règle y soit applicable. De façon symétrique, l'autre ensemble, le contexte inhibiteur, contient les mots qui ne doivent pas être présents dans la chaîne pour que la règle y soit applicable.

Un cas particulier du contrôle semi-conditionnel est le contrôle par contextes aléatoires *(random context control).* Les systèmes avec des contextes aléatoires sont des systèmes avec du contrôle semi-conditionnel dans lesquels les contextes promoteurs et inhibiteurs contiennent des caractères à part, et non pas des mots.

Le mécanisme de contrôle par graphe a été adapté pour les systèmes d'insertion/effacement et il a été montré que les systèmes de taille $(1, 1, 0; 1, 1, 0)$ engendrent tous les langages récursivement énumérables avec quatre étiquettes (groupes de règles) seulement. Les systèmes d'insertion/effacement avec le mécanisme de contrôle matriciel ont été étudiés aussi ; il a été prouvé que les systèmes matriciels de petites tailles ($(1, 1, 0; 1, 1, 0)$, par exemple) atteignent la complétude computationnelle.

Dans la section 4.2 nous revoyons le contrôle par graphe pour les systèmes d'insertion/effacement de tailles $(1,1,0;1,2,0)$ et $(1,2,0;1,1,0)$ et nous montrons que trois étiquettes suffisent pour engendrer tous les langages récursivement énumérables. Nous rappelons que nous utilisons un étiquetage non bijectif, donc chaque étiquette correspond à un groupe de règles.

Dans la section 4.3 nous adaptons le contrôle semi-conditionnel aux systèmes d'insertion/effacement et nous montrons que ce mécanisme augmente considérablement leur puissance d'expression. Nous montrons notamment que tous les langages récursivement énumérables peuvent être engendrés par des systèmes ayant des règles de taille $(1,0,0;1,0,0)$. Bien que les insertions et les effacements dans ce genre de systèmes puissent intervenir à n'importe quelle position dans la chaîne, les contextes promoteurs et inhibiteurs peuvent être utilisés pour rejeter les cas où l'opération ne s'est pas produite au bon endroit. Nous montrons également que la présence de règles d'effacement est nécessaire pour atteindre la complétude computationnelle, car la famille des langages engendrés par les systèmes d'insertion/effacement semi-conditionnels de taille $(1,0,0;0,0,0)$, c'est-à-dire sans règles d'effacement, est incluse dans la famille de langages contextuels.

Finalement, dans la section 4.4 nous adaptons le contrôle par contextes aléatoires aux systèmes d'insertion/effacement et nous montrons que la puissance d'expression de ces systèmes est augmentée. Nous prouvons que les systèmes de taille $(2,0,0;1,1,0)$ munis de ce type de contrôle engendrent tous les langages récursivement énumérables. En revanche, il existe des langages rationnels que les systèmes de taille symétrique $(1,1,0;2,0,0)$, et même plus généralement de taille $(1,1,0;p,1,1)$, ne peuvent pas engendrer. Le fait que les systèmes de tailles $(2,0,0;1,1,0)$ et $(1,1,0;2,0,0)$ munis de contrôle par contextes aléatoires n'atteignent pas la complétude computationnelle simultanément est assez remarquable, car dans tous les cas connus les familles symétriques possèdent les mêmes propriétés par rapport à la complétude computationnelle.

Vers la fin du chapitre, dans la section 4.5, nous considérons un modèle de calcul très semblable aux systèmes d'insertion/effacement avec un mécanisme de contrôle distribué : les réseaux de processeurs évolutionnaires *(networks of evolutionary processors)*. Un tel réseau consiste en des unités de calcul élémentaires (processeurs évolutionnaires) pouvant effectuer des insertions, des effacements et des substitutions d'un seul caractère, sans contexte. Les processeurs alternent entre des phases de calcul et des phases de communication. Lors d'une phase de calcul chaque processeur modifie les chaînes de caractères qu'il contient ; pendant une phase de communication les chaînes sont redistribuées entre les processeurs. Un processeur évolutionnaire possède un filtre à l'entrée et en sortie ; toute chaîne destinée à un processeur mais qui ne passe pas son filtre d'entrée sera rejetée. Les chaînes produites par un processeur et ne passant pas le filtre de sortie seront bloquées à l'intérieur du processeur pour un traitement ultérieur. Les réseaux de processeurs évolutionnaires sont un modèle de calcul inspiré par l'activité des organites d'une cellule biologique et par la collaboration des cellules d'un tissu ; ce modèle peut être vu comme une mise en réseau des systèmes de Lindenmayer simples.

La complétude computationnelle des réseaux de processeurs évolutionnaires de différentes topologies a été prouvée dès leur introduction. De plus, il a été montré que ces réseaux peuvent résoudre des problèmes NP-complets en temps linéaire.

Dans la section 4.5 de ce manuscrit nous nous intéressons aux réseaux universels, c'est-à-dire aux réseaux qui peuvent simuler n'importe quel autre réseau. Nous construisons des réseaux universels à 4, 5 ou 7 règles seulement, selon la fonction de codage. Notre résultat montre que le degré de finesse auquel les filtres peuvent contrôler l'application de règles d'insertion, d'effacement et de substitution très simples est assez élevé.

Chapitre 5

Dans le cinquième chapitre de ce manuscrit nous considérons le problème de l'universalité pour les machines à registres. Une telle machine est composée d'un ensemble fixe de registres et d'un programme fini dont les instructions peuvent accéder aux registres par leurs noms. Les registres contiennent des valeurs entières non négatives et non bornées. Plusieurs types d'instructions peuvent être considérés, mais dans la plupart des cas ce sont soit des opérations simples sur les registres, comme l'incrément ou le décrément, soit des vérifications de conditions, comme le test d'un registre à zéro.

Le concept de registre est apparu après l'invention des machines de Turing universelles à deux symboles, dont un symbole vide. Ces machines manipulant des blocs de symboles non vides séparés par des symboles vides, il est possible de les voir comme traitant des nombres et non pas des chaînes de caractères.

Malgré le lien fort entre les machines de Turing et les machines à registres, le problème de l'universalité a été beaucoup moins abordé pour celles-ci. En effet, l'existence des machines à registres universelles a été montrée par Marvin Minsky dans le travail où il les a introduites, alors que les machines universelles concrètes n'ont été construites qu'en 1996 par Ivan Korec. Dans son travail, Korec s'est proposé de réduire le nombre d'états des machines à registres universelles, et il en a décrit plusieurs, utilisant de différents types d'instruction.

Il existe deux motivations principales pour la recherche des petites machines à registres universelles. D'un côté, ces machines mettent en évidence les caractéristiques nécessaires pour atteindre la complétude computationnelle, puisque les petites machines universelles le font avec peu de redondance. D'un autre côté, simuler les machines à registres est souvent la méthode la plus directe pour montrer la complétude computationnelle d'une classe de machines qui manipulent des vecteurs de nombres. Par conséquent, fournir de petites machines à registres universelles permet de trouver de petits éléments universels dans ces autres classes aussi.

Dans la section 5.2 nous continuons la recherche des petites machines à registres universelles et nous nous concentrons sur la minimisation du nombre de registres. Bien que Marvin Minsky ait proposé une technique de simulation d'une machine à un nombre arbitraire de registres par une machine à deux registres seulement, nous n'avons pas trouvé de description concrète de telles machines dans la littérature. Nous utilisons donc l'approche proposée par Minsky afin de simuler les machines à registres décrites par Korec et de construire ainsi une machine universelle à trois registres, dont un servant au stockage de l'entrée et de la sortie, et une machine universelle à deux registres dont l'entrée et la sortie sont encodées exponentiellement.

Un des avantages des machines à registres et des machines de Turing est la simplicité des instructions qu'elles peuvent avoir. Cela rend leur simulation plus simple en utilisant des moyens minimaux. Néanmoins, il arrive souvent que les outils disponibles pour la simulation soient plus expressifs et soient capables de simuler plusieurs instructions à la fois. Ceci est bien le cas des règles de réécriture de multiensembles et des réseaux de Petri qui permettent de simuler les incréments de plusieurs registres en un seul pas d'évolution. N'étant pas gênant un soi, ce phénomène devient un problème lorsque l'on est à la recherche de petits éléments universels et que l'on considère d'autres modèles de calcul dont les unités d'activité atomiques sont plus expressives que les instructions des machines à registres. Il arrive souvent que les techniques de simulation optimisée soient similaires pour des modèles de calcul différents, ce qui met en évidence une redondance descriptive importante.

Dans la section 5.1 nous proposons une généralisation de machines à registres dont le but est de fournir un langage de description des opérations plus puissant. Une machine à registres généralisée est essentiellement un multigraphe dont les nœuds sont les états et dont les arêtes sont annotées des opérations et des conditions de franchissement. Une telle arête peut être franchie seulement si toutes les conditions qui y sont associées sont vraies ; le franchissement d'une arête entraîne l'exécution de toutes les opérations associées. Les arêtes des machines à registres généralisées sont ainsi fortement similaires aux transitions des réseaux de Petri et aux règles de réécriture de multiensembles.

Les machines à registres habituelles sont elles aussi souvent conçues comme des graphes d'états, mais comme dans ce modèle de calcul la correspondance entre les états et les opérations est biunivoque, ce sont toujours les nœuds du graphe qui sont annotés des conditions et des opérations. Il est néanmoins clair que toute machine à registres habituelle est une machine à registres généralisée. L'inverse n'est pas vrai puisqu'une machine à registres généralisée peut vérifier plusieurs conditions et effectuer plusieurs opérations en un seul pas.

Grâce à la possibilité d'associer plus d'une opération ou d'un test à une arête, il est souvent possible de construire une machine à registres généralisée qui fait le même calcul qu'une machine à registres habituelle, mais en utilisant moins d'états. Par exemple, plusieurs incréments qui auraient nécessité plusieurs états dans le modèle classique peuvent être exécutes par une seule arête généralisée. Dans la section 5.1 de ce mémoire nous définissons de façon formelle les cas dans lesquels il est possible d'éliminer des états en plaçant les opérations qui leur correspondent sur une seule arête. Nous allons ensuite construire des machines à registres généralisées universelles ayant sept états seulement, dont l'état d'arrêt.

Chapitre 6

Le sixième chapitre de ce manuscrit contient l'étude du problème de l'universalité pour des systèmes de réécriture de multiensembles. Un multiensemble est une collection non ordonnée d'objets qui admet des répétitions ; les multiensembles sont donc une généralisation des ensembles.

Nous remarquons tout de suite qu'un multiensemble peut être représenté par un vecteur d'entiers naturels dont le n-ième composant est la multiplicité du n-ième symbole de l'alphabet. D'un autre côté, une configuration d'une machine à registres est représentée également par un vecteur d'entiers non négatifs. Une règle de réécriture de multiensembles correspond dans ce cas à une suite de décréments et d'incréments des registres. Cependant, les règles de réécriture de multiensembles ne sont pas capables de vérifier l'absence d'un symbole, ce qui correspondrait à la vérification du fait que la valeur d'un registre soit zéro. Un des moyens d'augmenter la puissance de calcul de la réécriture de multiensembles est de rajouter aux règles un ensemble d'objets dits inhibiteurs, dont aucun ne doit apparaître dans un multiensemble pour que la règle y soit applicable. Cette extension permet de simuler un test de registre à zéro et rend la puissance d'expression de la réécriture de multiensembles équivalente à celle de machines à registres.

Dans ce chapitre nous présentons les systèmes de réécriture de multiensembles universels sous la forme des réseaux de Petri avec des arcs inhibiteurs. Un réseau de Petri est un multigraphe qui contient deux types de nœuds : les places et les transitions. Les places peuvent contenir des jetons ; un arc allant d'une place P vers une transition T indique que T consomme un jeton de P au déclenchement. Symétriquement, un arc allant de T vers une place Q indique que T rajoute un jeton à Q au déclenchement. L'état d'un réseau de Petri est généralement décrit par une fonction, dite marquage, qui associe à chaque place le nombre de jetons qu'elle contient ; l'état d'un réseau est donc un multiensemble et les transition sont des règles de réécriture de multiensembles.

Les réseaux de Petri, de même que les systèmes de réécriture de multiensembles sans inhibiteurs, ont le problème d'accessibilité décidable. Une extension qui permet d'étendre la puissance d'expression des réseaux de Petri consiste à rajouter des arcs inhibiteurs. Un tel arc entre une place et une transition empêche celle-ci de se déclencher lorsque la place est vide. Graphiquement les arcs inhibiteurs sont représentés par un cercle du côté de la transition.

Nous remarquons qu'il existe d'autre modèles de calcul qui s'inscrivent dans la même famille que les machines à registres, les systèmes de réécriture de multiensembles et les réseaux de Petri. Les systèmes d'addition de vecteurs *(vector addition systems)* sont un exemple d'un tel modèle. Ces systèmes évoluent en additionnant des vecteurs d'un ensemble fini, dits vecteurs d'addition *(addition vectors)*, à un vecteur de départ. Un vecteur d'addition w peut être appliqué à un vecteur x seulement si tous les composants du vecteur $x+w$ sont non négatifs. Un vecteur d'addition correspond ainsi à une règle de réécriture de multiensembles dont les membres gauche et droit contiennent des symboles différents, ou à une transition de réseaux de Petri qui ne remet jamais de jetons dans une place de laquelle elle en consomme.

Nous construisons de petits réseaux de Petri universels avec des arcs inhibiteurs. Nous définissons la taille d'un réseau comme un vecteur comprenant le nombre de places, de transitions, d'arcs inhibiteurs, ainsi que le nombre maximal d'arcs incidents à une transition (le degré maximal de transitions). Nous proposons ensuite des techniques de minimisation de chacun de ces paramètres, tout en mettant en évidence certains compromis.

Dans la section 6.2 nous décrivons l'une des façons les plus directes de construire un réseau de Petri universel par simulation d'une machine à registres universelle. Malgré la simplicité de l'approche, les réseaux obtenus par cette voie ont des transitions du degré minimal ; nous montrons que les réseaux de Petri avec des transitions de degré encore plus petit ne possèdent pas la complétude computationnelle.

La section 6.3 porte sur la minimisation du nombre de transitions dans les réseaux de Petri universels. Nous construisons des réseaux simulant les machines à registres généralisées, ce qui permet d'atteindre la complétude computationnelle avec moins de transitions que dans le cas des simulations des machines à registres habituelles. Nous proposons ensuite une technique de codage binaire des états de la machine simulée, ce qui réduit le nombre de places dans les réseaux universels, tout en gardant faible le nombre de transitions.

Dans la section 6.4 nous attaquons le problème de minimisation du nombre de places et nous montrons deux méthodes différentes de simuler n'importe quelle machine à registres avec un réseau de Petri ayant quatre places seulement. Une place supplémentaire est nécessaire si le codage de l'entrée et le décodage de la sortie se fait par le réseau lui-même. Les réseaux obtenus par la première méthode sont non déterministes, alors que ceux obtenus par la deuxième sont déterministes si la machine simulée l'est. Toutefois, ceux-ci utilisent des transitions d'un degré plus important que les réseaux construits d'après la première méthode.

Finalement, la section 6.5 est dédiée à la minimisation du nombre d'arcs inhibiteurs. Nous proposons une méthode de simulation de machines à registres qui ne nécessite qu'un seul arc inhibiteur par registre. Nous appliquons cette approche à la simulation des machines universelles à deux registres et nous arrivons à des réseaux de Petri universels avec deux arcs inhibiteurs seulement, ce qui est le nombre minimal d'arcs inhibiteurs nécessaire pour la complétude computationnelle — il a été montré que le problème d'accessibilité pour les réseaux de Petri avec un seul arc inhibiteur est décidable.

Contents

Introduction

Models of computation, as well as the theory of computation and formal languages which is built around them, could seem to be too abstract to ever have any influence on the study and manipulation of any real-life situations. Nevertheless, it is often by sublimation of empirical experience into formal knowledge that impressive advances in the understanding of natural phenomena are achieved. Meaningfully plodding through profuse, but not immediately categorised, data about the world requires solid theoretical background if a complete picture is to be drawn. This is when the theory of computation and formal languages come into play, along with other abstract domains, and help detect and isolate the core components of the complex systems under study.

At the bases of formal examination of computation lie the Turing machine and a series of devices equivalent in power, such as formal grammars, partial recursive functions, or lambda calculus. These constructions contour the outer bounds of what can be computed without resorting to infinite resources with immediate availability and originate in endeavours to comprehend the foundations of mathematics and the functioning of the human brain. Besides being a deep introspection exercise, the research into the formal notion of computation did help found the domain of practical computer science, which is now a ubiquitous, universal, and largely irreplaceable tool preferred by engineers, scientists, and casual users alike. In other words, some abstract considerations ended up shaping the world as we know it today in some rather concrete ways.

While it is true that, historically, the principal applications of the theory of computation and formal languages lie within or are at least connected with computer science, in the recent years these domains started offering curious and useful insights in biology and related disciplines. The advent of natural computing towards the end of the twentieth century established solid links between some biological phenomena and the formal notion of computing. Notably, the famous experiment by Leonard Adleman described in [1], pointed out that some of the transformations DNA molecules may undergo can be interpreted as computation. This ingenious experimental setup essentially created the whole new field of DNA computing, whose main goal is using biological molecules as a direct substrate for computation (see [103] for an overview).

One of the explanations of the possibility of using DNA molecules as support for computing is in the fact that the theory of computation extensively uses the formal language theory. On the one hand, the two Watson-Crick base-pairs building up the DNA allow an easy interpretation of these molecules as formal strings, so biological systems can be seen as systems manipulating strings. On the other hand, the theory of computation naturally considers string-based representations of problems and data, which means that biological systems can actually be seen as computing devices.

An important difference between biological systems and formal models is that the majority of classical models of computing are inherently modular, on many levels. For example, a Turing machine has a separate program, a tape for storing and manipulating data. Furthermore, its tape consists of singular cells, its program – of atomic instructions. Biological processes, on the other hand, do not usually lend themselves to such clear structurisation easily: even DNA molecules which are often conceived as read-only and almost invariable information, interact rather actively with other entities in the cell and, in some organisms, undergo numerous modifications. Because of the fundamental dissimilarity between the character of models of computation and biological systems, the computational approach to biology starts by pointing out how to split the system into components and how to formally interpret their actions as steps of computation. Often, and especially in the case of processes involving genetic information, modelling based on formal languages is preferred, because formal grammars have remarkably few structural elements – the rules and the manipulated string – but are still, in their most general form, equivalent in power to Turing machines. For example, the work [115] analyses the expressive power of the operation of splicing seen as an operation on strings, while [107] focuses on the process of gene assembly happening in certain bacteria.

Establishing the parallel between Turing machines and a subsystem of the living cell, or any other biological structure, has two broad types of consequences. On the one hand, it immediately reveals that the complexity of the subsystem under study is rather elevated and that the majority of questions concerning its dynamic behaviour are not solvable with a conventional computing device. This opens up a whole new perspective on the biological systems, which is unifying and generalising in the sense that it treats them as systems transforming information, and then gives levers to estimating the complexity of the transformations. On the other hand, stating that some of the operations a biological system carries out are sufficiently complete to simulate a Turing machine, makes it possible to consider (may it be mostly theoretically at present) biological computers, that is, biological systems programmed to solve some concrete problems.

In this thesis we contribute to both facets of cognition the computational approach to biology opens up. In the first part, we focus on the operations of insertion and deletion, which were originally introduced with linguistic and formal-language motivation, but were then discovered to have numerous counterparts in the world of DNA and RNA manipulation. In the most general case, these two operations insert or delete substrings of a string if the insertion or deletion sites are delimited by some fixed contexts. In this work we specifically consider one-sided versions of these operations, in which the insertion or deletion may only depend either on left or right context of the site, but not both. This restriction corresponds directly to how the process of editing of RNA strands is organised, and thus presents interest for understanding the complexity of these manipulations of RNA. We show that one-sided insertion and deletion are quite sophisticated, even if only one symbol is allowed to be inserted or deleted at a time. We do conjecture though that this sophistication is still not enough to achieve the power of Turing machines.

We also discuss insertion and deletion operations endowed with additional control, which reflects the fact that the cell may change between different states and thus perform different sets of transformations on DNA or RNA strands. We show that adding control almost always boosts the power of very small insertion and deletion rules to computational completeness.

The second part of the thesis is concerned with the synthetic perspectives induced by the computational approach to biology, and focuses on universality. A universal element in a class of computing devices is a device which can simulate any other device of this class. We consider the problem of constructing universal computing devices which should also admit compact description. Giving such a device in a class of models of computing inspired by some biological phenomena essentially lays the foundations of using the corresponding biological systems as computers. Indeed, small and universal devices point out, in a way, the amount of complexity a concrete biological system must achieve in order to be able to reproduce any computation of a Turing machine. We pick the class of Petri nets with inhibitor arcs to showcase our constructions, but the obtained results are directly translatable to multiset rewriting systems, and, in particular, to membrane systems, designed to follow the structure of the living cells.

Our universality constructions are loosely based on universal register machines, and we exploit the connection between these computing devices and Petri nets in the reverse direction, too: we introduce generalised register machines which essentially allow more complex operations during a state change, to match the expressiveness of Petri net transitions. We also describe small universal constructions in this new class of computing devices.

Outline

Chapter 1

This chapter provides a overview of the domains this thesis falls within. It first recalls the historic evolution of insertion and deletion as formal operations, and then discusses two different biological motivations for study of these operations: mismatched annealing of DNA strands and RNA editing. The second part of this chapter recalls the origins of Petri nets and some results on reachability, and hence expressive power, of basic Petri nets and Petri nets with inhibitor arcs. A tight connection with register machines and multiset rewriting systems is also pointed out.

Chapter 2

This chapter recalls some of the standard definitions from the theory of formal languages and computation. It briefly discusses formal grammars, finite automata, Turing machines, register machines, as well as the notions of computational completeness and universality.

Chapter 3

This chapter studies the expressive power of insertion-deletion systems in which all rules insert or erase exactly one symbol, and should also verify the context on one side of the site only. It is shown that, when at least insertion or deletion rules are allowed to use two-symbol contexts, the resulting insertion-deletion systems generate all regular languages. Furthermore, we show that further increasing the lengths of the contexts does not increase the computational power. Finally, we introduce a novel tool for visual analysis of the derivations of one-sided insertion-deletion systems with contexts of length 1.

Chapter 4

This chapter focuses on insertion-deletion systems equipped with additional control. Three control mechanisms are considered, and in all cases computational completeness is shown for very small rule sizes. The second part of this chapter is concerned with networks of evolutionary processors, and gives two universal networks with 4, 5, and 7 rules only. The results shown in this chapter are based on the works [57, 58, 59].

Chapter 5

In this chapter we consider the questions of universality for register machines. In the first part of this chapter a generalisation of the conventional model is introduced, in which multiple register tests and operations can be performed in a single state change. We then give a concrete universal generalised register machine with 7 states only. In the second part of this chapter we recall Marvin Minsky's exponential simulation technique [92], allowing machines with arbitrarily registers to be simulated by machines with two registers. We then apply this approach to actually construct universal 3- and 2-register machines. We remark that we did not find any similar explicit constructions in the literature on register machines. The results from this chapter are based on [55, 56].

Chapter 6

This chapter discusses universal devices in the class of Petri nets with inhibitor arcs. We define the size of such a Petri net as a tuple comprising the number of places, transitions, inhibitor arcs, and the maximal transition degree, and then describe techniques for minimising each of these parameters. Some of these techniques allow attaining the theoretical minimum required for universality. The results from this chapter are based on [55, 56] as well.

Chapter 1

State of the Art

This chapter provides a brief overview of the domain of the present thesis, giving a historical outline and relating the results from the following chapters to the research conducted previously. In Section 1.1 we discuss the operations of insertion and deletion in a historical perspective, Section 1.2 considers networks of evolutionary processors, while Section 1.3 focuses on Petri nets and register machines, but also touches upon the questions of computational completeness and universality.

1.1 Insertion-deletion Systems

Insertion and deletion are one of the simplest and yet powerful operations on strings studied in the theory of formal languages. Quite intuitively, inserting a substring z in between x and y in the string $\alpha xy\beta$ results in $\alpha xzy\beta$. Dually, deleting the substring z from $\alpha xzy\beta$ produces the string $\alpha xy\beta$.

The history of insertion starts in 1969 with Solomon Marcus's seminal paper on contextual grammars [82], in which he introduced a new type of generative devices not directly compatible with the Chomsky hierarchy. Contextual grammars start with a set of words over an alphabet V and generate languages by iteratively adjoining contexts to these words. A context is a pair (u, v) of words over V and adjoining it to a word x yields the word uxv.

One can immediately see that the context-free language $\{a^n b^n \mid n \in \mathbb{N}\}$ can be easily generated by a contextual grammar. The power of contextual grammars does not extend beyond context-free languages, however [82, Proposition 7]. Moreover, not all context-free languages or even regular languages can be generated by contextual grammars [82, Propositions 8 and 9].

The article [8] extends the construct of contextual grammar by adding control over the words a context $C = (u, v)$ can be adjoined to by associating with it a language of allowed words (selectors) D. Then, the word uxv can be obtained from x only if $x \in D$. Furthermore, the authors introduce internal derivations, in which contexts are not necessarily adjoined to the ends: whenever x belongs to the set of selectors of the context $C = (u, v)$, C can be adjoined to x within the string $\alpha x\beta$, yielding $\alpha uxv\beta$. Accordingly, the original derivation mode in which contexts are adjoined to the ends of the string is referred to as external. Just as external derivations, internal derivations generate families of languages fully contained in the class of context sensitive languages and incomparable with context-free languages [85]. Yet the degree to which contextual languages are context sensitive seems to be sufficient to capture some fundamental structures in natural languages [83, Introduction].

In [45], a counterpart of Marcus's contextual grammars – semicontextual grammars – is introduced. The working principle of such grammars is insertion of a substring in a given context, rather than adjoining a context to a substring. Thus, a semicontextual grammar contains rules of the form $xy \to xzy$, which effectively result in the insertion of the substring z in between x and y in the string $\alpha xy\beta$. The work [81] studies the expressive power of semicontextual grammars with additional control mechanisms and highlights a series of interesting relationships between various classes of grammars obtained in this way, including a number of infinite hierarchies.

The first mention of insertion under this name is usually attributed to David Haussler, who introduced this operation as a generalisation of Kleene's concatenation [69] in his doctoral thesis [50]. Indeed, insertion can be seen as concatenation, which is not restricted to happening at the ends of strings. In his paper [51], Haussler shows that insertion systems are able to generate non-regular Dyck languages, and proves that regularity of an insertion language is undecidable.

Although deletion seems to be quite a natural counterpart of insertion, it was first introduced several years later in Lila Kari's paper [110], in an attempt to bring over the basic operations of arithmetic to formal languages. Deletion is defined as a generalisation of the quotient operation, in a way which parallels the definition of insertion as a generalisation of concatenation. Kari continued the study of insertion and deletion in her doctoral thesis [65], where she described variants of the two operations, including parallel versions, and investigated their computational power.

The work [66] is one of the first to introduce the now wide-used notations for insertion-deletion systems. Namely, an insertion-deletion system is defined as the tuple (V, T, A, I, D), where V is the alphabet, $T \subseteq V$ is the alphabet of terminal symbols, A is the finite set of axioms, I is the set of insertion rules, and D is the set of deletion rules. The sets of rules contain triples of the form (u, x, v), where u and v are the left and right contexts respectively, while x is the substring to be inserted or deleted. For convenience, the notation $(u, x, v) \in I$ is often replaced by $(u, x, v)_{ins}$, and $(u, x, v) \in D$ by $(u, x, v)_{del}$.

Throughout this thesis we will usually specify the maximal allowed lengths of all components of insertion and deletion rules directly to define a family of insertion-deletion systems of the same expressive power. Thus, we will say that the size of an insertion-deletion system with the set of insertion rules I and the set of deletion rules D is $(n, m, m'; p, q, q')$, where

$$\begin{aligned} n &= \max\{|x| : (u,x,v) \in I\}, & p &= \max\{|x| : (u,x,v) \in D\}, \\ m &= \max\{|u| : (u,x,v) \in I\}, & q &= \max\{|u| : (u,x,v) \in D\}, \\ m' &= \max\{|v| : (u,x,v) \in I\}, & q' &= \max\{|v| : (u,x,v) \in D\}. \end{aligned}$$

The paper [66] also points out an exciting new motivation for the study of insertion and deletion operations: DNA computing or, even more generally, molecular computing – domains vividly illustrated by Leonard Adleman in his seminal report about solving the Hamiltonian path problem with DNA molecules [1]. The authors of [66] highlight the fact that insertion and deletion suffice for modelling DNA computation and, moreover, that actually performing the operations on real DNA molecules is quite doable in laboratory conditions. The cited article shows that insertion-deletion systems with rather strong restrictions on size are capable of characterising the whole family of recursively enumerable languages and are thus comparable in power to Turing machines.

The theoretically-conceived way of performing computations on DNA molecules is by mismatched annealing, which is essentially mismatched gluing of DNA strands resulting in omissions or additions of DNA fragments [103]. In normal conditions, DNA comes in two strands, each of which is composed out of four types of nucleotides: adenine, guanine, thymine, and cytosine, often abbreviated as A, G, T, and C, respectively. The nucleotides A and T exhibit electrostatic attraction to each other and are referred to as complementary. Similarly, G and C form another complementary pair of nucleotides. The entire sequences of nucleotides in the two strands of a DNA molecule are complementary, and it is the attraction between nucleotides that keeps the two strands together.

The attractive force between complementary nucleotides is relatively weak, however, and can be broken by special proteins (enzymes) preparing the DNA molecule for reading the genetic information, or even by heating, resulting in single separate strands of DNA. Annealing is the process in which such single strands reconnect to complementary strands, or even form the necessary complementary strand out of the nucleotides available around. Mismatched annealing, on the other hand, is a theorised process during which two not fully complementary strands do connect in complementary positions, a situation which can be subsequently used to induce insertions or deletions of subsequences of nucleotides. The stages of mismatched DNA annealing are shown in Figure 1.1.

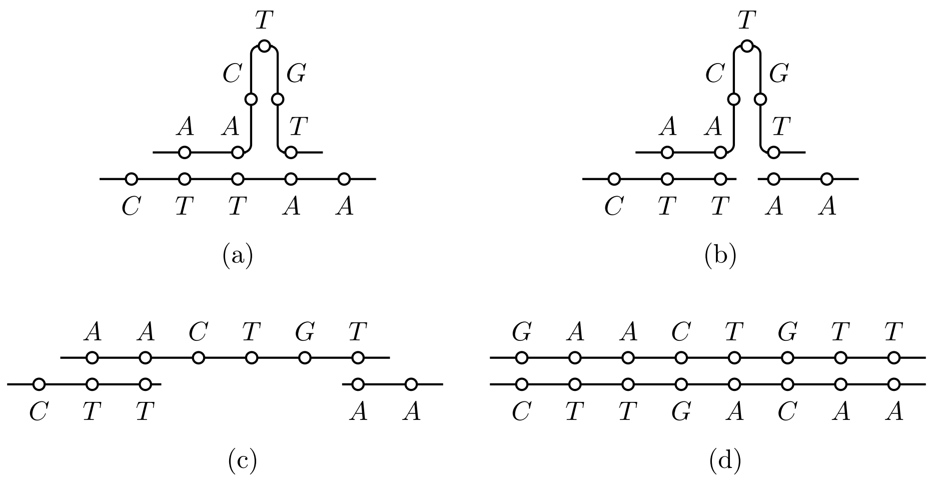


Figure 1.1: The stages of mismatched DNA annealing: (a) formation of complementary nucleotide pairs, (b) cleaving of the shorter strand, (c) unfolding of the longer strand, (d) addition of missing nucleotides to both strands.

Consider the two strands of DNA shown in Figure 1.1a. They could form the nucleotide pairs $A - T$, $A - T$, and $T - A$, as shown in the figure. One could then cleave the second (lower) strand (Figure 1.1b) and unfold the first (upper) strand open (Figure 1.1c). The gap in the second strand could then be filled in with the nucleotides corresponding to the triple CTG on the first strand, and the first strand itself would be completed to match the nucleotide sequence on the second strand. After separating the resulting two strands by heating the solution, one could obtain a new lower strand which corresponds to the original strand with the subsequence

GAC inserted between *T* and *A*. On the other hand, the new upper strand will correspond to the original upper strand with a *G* inserted at its beginning and a *T* at its end.

By a similar theoretical process one could achieve deletion of subsequences of nucleotides. In this case, it would be the upper strand shown on Figure 1.1a that would be cleft between *A* and *C*, and between *G* and the second *T*. This would effectively lead to cutting out of the subsequence *CTG*.

Interestingly, a process which is closely related to the theoretically conjectured mismatched annealing was discovered for RNA molecules in 1993 [12]. This process, referred to as RNA editing, deviates from the central dogma of molecular biology that information from the DNA is solely instanced onto RNA molecules, which then serve as matrices for protein production. The editing concerns messenger RNA molecules (mRNA), which carry a replica of the genetic information, and is performed according to the pattern given by a guide RNA molecule (gRNA).

Similarly to DNA, RNA molecules are built up out of four nucleotides, three of which are the same as the ones appearing in DNA: adenine, cytosine, and guanine, and one which is different: uracil, denoted by *U*. Similarly to thymine (T), uracil exhibits affinity for adenine, and thus the nucleotides forming the RNA can also be grouped into two complementary pairs: *A* and *U*, and *C* and *G*. As different from DNA molecules, RNA usually comes in singular strands which are more active chemically.

Even though RNA editing is based on the same principle as DNA annealing: electrostatic attraction between complementary nucleotides, the phases and the result of the process do differ. One difference is that, in RNA editing, only mRNA is modified, while gRNA is left intact. Another important specificity of RNA editing is that only sequences of uracil (denoted by *U*) can be inserted or added to the modified mRNA molecule.

A high-level overview of the phases of RNA editing is shown in Figure 1.2. The process starts with the attaching of a guide RNA to the messenger RNA to be modified. This system of two molecules is then wrapped by a protein complex which cuts open the mRNA, unfolds the gRNA molecule, and then inserts instances of *U*

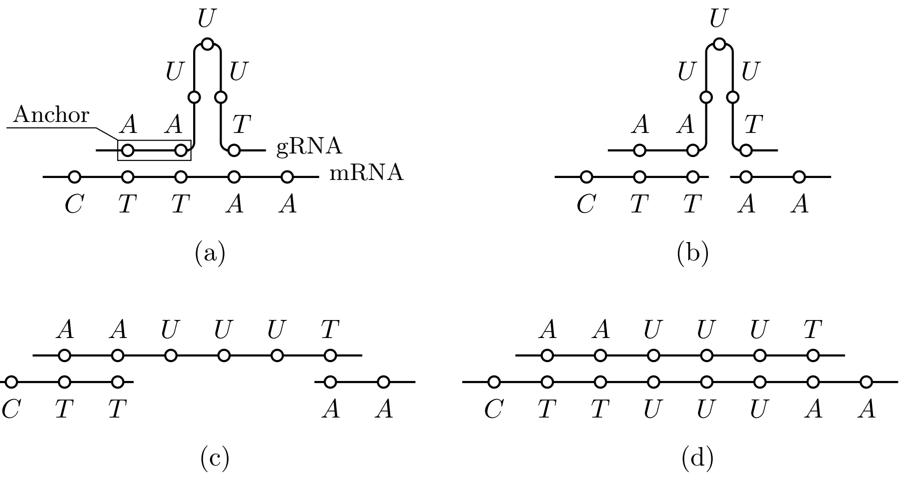


Figure 1.2: The stages of RNA editing: (a) attachment of gRNA (top) to mRNA (bottom), (b) cleaving of mRNA, (c) unfolding of mRNA, (d) addition of U to mRNA.

into mRNA to match the corresponding sequence on the guide. Deletion of uracil segments can be performed symmetrically: if, for example, the mRNA has the form $ACGUUGA$, "applying" the guide $CGGA$ will result in erasing of the two instances of uracil in the original mRNA. Once again, remark that gRNA is not modified in the process, as different from mismatched DNA annealing, where the shorter strand is completed to match the length of its complement.

RNA editing alters the genetic information after it has been read from the DNA, but before it is implemented in proteins, which opens up interesting perspectives. In fact, it may be that RNA editing is a better substrate for biological in vitro computations [112]. This motivated the extension of the formalism of insertion-deletion systems to model the function of gRNA molecules in [17]. Guided insertion-deletion systems, introduced in this work, start their computations with an initial string, but also with a language of control words, called guides. All operations the system performs must be paralleled by the corresponding operation on one of the guides. The paper [13] focuses on guided insertions and shows that the families of languages generated by systems with such rules contain the regular languages, but are essentially incomparable with the other levels of the Chomsky hierarchy.

Further study of the mechanism of guides was conducted in [124], where it was shown that, for a fixed set of guides, guided insertion-deletion systems can generate strings which are exponential in size with respect to the initial words (axioms). The author further considers guided insertion-deletion systems as language transformers, and shows that, if the language of the axioms is regular (respectively context-free), the language generated by the system is not necessarily regular (respectively context-free).

The article [30] goes back to the biological origins of the model and points out that a single guide RNA molecule does not necessarily induce only an insertion or only a deletion. For example, a gRNA molecule $ACUUUG$ could also induce the transformation of $GUAUCUGG$ into $GUACUUUGG$, thereby erasing one uracil molecule between A and C, and also adding two between C and G.

A remarkable feature of gRNA molecules is that they have a distinguished part, the anchor, which is always situated to the same side of the section actually leading to modifications of the mRNA [11, p. 223]. For example, in Figure 1.2, the anchor is represented by two molecules of adenine (A) at the left end of the gRNA molecule. In a formal representation by insertion and deletion, this feature would be best modelled by rules which are only allowed to have a context on one of the sides, but not both. Such models are referred to as *one-sided insertion-deletion systems* (e.g. [73]), and Chapter 3 of the present work focuses on the analysis of such systems having rules of small size.

Abundant biological and linguistic motivation (e.g. [73]) for the operations of insertion and deletion lead to a thorough study of their expressive power. A summary of some important results relating the languages generated or recognised by pure insertion and pure deletion systems to well-known classes of languages is given in Table 1.1.

It turns out, for example, that any recursively enumerable language can be obtained by applying an inverse morphism and a weak coding to an insertion language (Table 1.1, line 1), and that insertion rules of maximal size $(3, 7, 6)$ suffice to this end. The main idea of the paper [86] is that symbols which are meant to be erased can instead be marked by inserting a flag next to it. Then, the inverse morphism

Table 1.1: A summary of results about pure insertion and pure deletion systems. The notation *wc* stands for “weak coding”, *im* for “inverse morphism”, *proj* for “projection”, $\cap Dyck$ for “intersection with a Dyck language”, $\cap loc_2$ for “intersection with a strictly 2-testable language, and $\cap REG$ for “intersection with a regular language”.

	Size	Power	Ingredients	Reference
1	$(3,7,6;0,0,0)$	$= RE$	wc, im	[86, Theorem 1]
2	$(3,3,3;0,0,0)$	$= RE$	wc, im	[67, Theorem 6]
3	$(3,0,0;0,0,0)$	$= RE$	$proj$, $\cap Dyck$	[102, Theorem 1]
4	$(3,3,3;0,0,0)$	$= RE$	$proj$, $\cap loc_2$	[43, Theorem 7]
5	$(2,0,0;0,0,0)$	$= CF$	$proj$, $\cap loc_2$	[43, Theorem 5]
6	$(1,1,1;0,0,0)$	$= CF$	$proj$, $\cap loc_2$	[44, Theorem 8]
7	iterated deletion	$= RE$	$\cap REG$, $proj$	[31, Theorem 1]

will be used to replace each such group with a special symbol, which will be subsequently removed by the weak coding. The work [67] (Table 1.1, line 2) improves on the previous result by showing that insertion systems of size at most $(3,3,3)$ are sufficient to characterise all recursively enumerable languages in the same way.

The article [102] (Table 1.1, line 3) gives yet another characterisation of recursively enumerable languages by insertion systems. The authors show that any such language can be obtained by intersecting the language generated by context-free insertion rules (i.e. rules without context) of size at most $(3,0,0)$ with a Dyck language.

This line of research was continued in the work [43], where the author gives a new characterisation of recursively enumerable languages by insertion systems of size $(3,3,3;0,0,0)$, intersection with a strictly 2-testable language, and a projection (Table 1.1, line 4). Strictly 2-testable languages are a subset of regular languages for which the membership of a word w can be decided by looking at its subwords of length 2. The paper [43] also gives characterisations of context-free and regular languages using insertion systems. In particular, it shows that context-free two-symbol insertion rules together with intersection with a strictly 2-testable language and a projection characterise CF (Table 1.1, line 5). In another work [44], the same author shows that insertion systems with rules of size $(1,1,1;0,0,0)$ combined with the same ingredients characterise the family of context-free languages, too (Table 1.1, line 6).

Due to symmetry between insertion and deletion, any characterisation of RE by insertion languages can be directly transformed into a characterisation of RE by deletion languages. The paper [31] (Table 1.1, line 7) takes a different approach and shows how this family of languages can be characterised by iterating a variant of deletion which corresponds to removal of substrings of a certain form from all words of a language [65, 110].

While insertion and deletion are quite powerful operations on their own, using both in a single system yields a more versatile instrument. Table 1.2 gives a summary of results on the computational power of such combined insertion-deletion systems. Lines 1, 2, and 3 of Table 1.2 summarise some of the results of [84], which show that context-free insertion-deletion systems with small-size rules without contexts are able to generate the family of recursively enumerable languages. Such systems can directly simulate the application of an arbitrary rewriting rule $R : u \to v$ by inserting vR and by then erasing Ru, where R is a special non-terminal symbol.

This result is also optimal in the context-free case, because systems where both insertion and deletion rules are limited to inserting or erasing at most two symbols at a time are not computationally complete, and moreover every such system can be simulated by a context-free pure insertion system of weight 2 [116]. Furthermore, the paper [116] considers context-free insertion-deletion systems in which insertions of arbitrarily long strings are allowed, but all deletion rules are restricted to erasing at most one symbol, and shows that such systems can only generate context-free languages [116, Theorem 4.1]. Systems with symmetric restrictions, i.e. systems in which deletion rules can erase arbitrarily many symbols at a time, but in which only one symbol can be inserted in a step, are even less powerful: they can only generate some regular languages [116, Theorem 4.2].

Small context-free insertion and deletion rules are equivalent in power to arbitrary rewriting grammars, which intuitively suggests that using contexts may allow achieving computational completeness with even smaller sizes of inserted and deleted strings. Indeed, insertion-deletion systems in which at most one symbol is allowed in all components of the rules generate all recursively enumerable languages (Table 1.2, line 9). Moreover, as shown in [73, Theorem 4.3.2] and [74, Theorem 3], context-free deletion rules with insertion rules of size $(1,1,1)$ and the symmetric variant are both computationally complete (lines 8 and 7 of Table 1.2).

Table 1.2: A summary of results on the power of insertion and deletion operations

	Size	Power	Reference
1	$(3,0,0;3,0,0)$	$= RE$	[84, Corollary 2]
2	$(3,0,0;2,0,0)$	$= RE$	[84, Theorem 3]
3	$(2,0,0;3,0,0)$	$= RE$	[84, Theorem 3]
4	$(2,0,0;2,0,0)$	$\subsetneq CF$	[116, Theorem 3.5]
5	$(m,0,0;1,0,0)$	$\subsetneq CF$	[116, Theorem 4.1]
6	$(1,0,0;p,0,0)$	$\subsetneq REG$	[116, Theorem 4.1]
7	$(1,1,1;2,0,0)$	$= RE$	[73, Theorem 4.3.2]
8	$(2,0,0;1,1,1)$	$= RE$	[74, Theorem 3]
9	$(1,1,1;1,1,0)$	$= RE$	[113, Theorem 2]

In what follows we will look more attentively into the types of insertion-deletion systems which we study in Chapters 3 and 4, and will give a detailed overview of the known results.

One-sided Insertion-deletion Systems. The article [87] introduces a restriction on the complexity of the contexts of insertion and deletion rules: one-sided contexts. An insertion-deletion system of size $(n,m,m';p,q,q')$ is said to be with one-sided contexts (or just one-sided) if either $m+m' > 0$ and $m \cdot m' = 0$, or $q+q' > 0$ and $q \cdot q' = 0$, or both. In other words, in one-sided insertion-deletion systems the rules are only allowed to have a context on one side, but not on the other. In [87] it is shown that this restriction does not considerably impair the generative power, and systems of sizes shown in lines 1–3 of Table 1.3 are computationally complete. Theorem 7 of [87] also shows that insertion-deletion systems of size $(1,1,1;1,1,0)$ are not computationally complete, and in fact are not even able to generate the context-free language $\{a^n b^n \mid n > 0\}$.

The paper [76] considers the cases symmetric to the ones shown in [87] (lines 6–9 vs. lines 1–4 of Table 1.3). It turns out that, at least for the considered families, swapping the sizes of insertion and deletion rules does not affect the property of being or not computationally complete, which is rather curious given that the proofs only share the general approach of simulating other computationally complete insertion-deletion systems. Remarkably, while both families of systems of size $(1, 1, 1; 1, 1, 0)$ and $(1, 1, 0; 1, 1, 1)$ are not computationally completely, the proofs of the two facts give little insight into the relationship between their power. Thus, while systems of size $(1, 1, 1; 1, 1, 0)$ are shown to be incapable of generating the language $\{a^n b^n \mid n > 0\}$, Theorem 8 of [76] proves that systems of size $(1, 1, 0; 1, 1, 1)$ cannot generate even the regular language $(ba)^+$.

For further reference, we cite the doctoral thesis [73] and the habilitation thesis [117], which give a thorough overview of the results on one-sided insertion-deletion systems (and on insertion-deletion systems in general) which had been achieved by 2011.

Table 1.3: A summary of results about one-sided insertion-deletion systems

	Size	Power	Reference
1	$(1, 1, 2; 1, 1, 0)$	$= RE$	[87, Theorem 3]
2	$(2, 0, 2; 1, 1, 0)$	$= RE$	[87, Theorem 4]
3	$(2, 0, 1; 2, 0, 0)$	$= RE$	[87, Theorem 5]
4	$(1, 1, 1; 1, 1, 0)$	$\subsetneq RE$	[87, Theorem 7]
5	$(2, 0, 0; 1, 1, 1)$	$= RE$	[76, Theorem 3]
6	$(1, 1, 0; 1, 1, 2)$	$= RE$	[76, Theorem 4]
7	$(1, 1, 0; 2, 0, 2)$	$= RE$	[76, Theorem 5]
8	$(2, 0, 0; 2, 0, 1)$	$= RE$	[76, Theorem 6]
9	$(1, 1, 0; 1, 1, 1)$	$\subsetneq RE$	[76, Theorem 7]

In Chapter 3 of the present work we discuss one-sided insertion-deletion systems of sizes $(1, 1, 0; 1, 2, 0)$ and $(1, 2, 0; 1, 1, 0)$ and show that these systems can generate all regular languages. We then prove that swapping insertion and deletion sizes in such systems does not change the expressive power, and thus the families of languages generated by systems of sizes $(1, 1, 0; 1, 2, 0)$ and $(1, 2, 0; 1, 1, 0)$ are equal. Finally, we show that considering longer left contexts for insertion and deletion does not change the expressive power either, because all of the language generated by systems of size $(1, m, 0; 1, q, 0)$ can also be generated by both systems of size $(1, 1, 0; 1, 2, 0)$ and of size $(1, 2, 0; 1, 1, 0)$.

Even though one-sided insertion-deletion systems were originally introduced with the mainly theoretical goal of exploring the generative power of the two operations, several other important sources of motivation were found for one-sided contexts. One of them is the fact that, as we have already stated above, guide RNA molecules start their action by "checking" a certain section of messenger RNA, which is always located on the same side of the region to be modified. Another source of motivation originates from computer security, from protection systems and trust management problems [93]. In the setup of these problems, the description of possible interactions between a series of objects is given, and the goal is to verify that undesired sequences of actions may not happen. A particular case is the accessibility problem, which deals with a graph of objects with edges encoding the relation "has access to", together with a set of operations reflecting the intuitive properties of this relation. The question is whether performing a sequence of such operations on a given accessibility graph can lead to some objects getting illegal access to other objects.

Before addressing the general accessibility problem, Section 2 of [93] defines a restriction of it and shows that solving the restricted variant is equivalent to deciding the membership of a word in the language recognised by a certain type of grammars introduced in the same paper and referred to as *leftist grammars.* A leftist grammar contains two types of rules: delete rules $ab \to b$ and insert rules $c \to dc$. A derivation of a leftist grammar is a sequence of applications of insert and delete rules, starting with the initial word of the form wx, where w is a string and x is a symbol. A word w belongs to the language recognised by a leftist grammar if the one-symbol string x can be derived from wx.

Because of the employed rule types, leftist grammars are clearly one-sided insertion-deletion-like systems, because symbols are always inserted or deleted on the left (which explains the name "leftist"). A single big divergence from the classical definition of an insertion-deletion systems is the direction of the derivation. While conventional insertion-deletion systems start from an axiom and generate words, leftist grammars rather function in a recognising mode. This difference can be easily overcome though by swapping rule types (i.e. transforming an insertion rule $(\lambda, x, y)_{ins}$ into the deletion rule $(\lambda, x, y)_{del}$ and vice versa) and by considering all the words which can then be derived from x. The insertion-deletion system obtained in this way will be of size $(1, 0, 1; 1, 0, 1)$. For historical reasons, when interpreting leftist grammars as insertion-deletion systems, we will also prefer swapping left and right contexts, thus transforming the rules of the form $(\lambda, x, y)_t$ into $(y, x, \lambda)_t$, $t \in \{ins, del\}$. By this procedure, we establish a direct correspondence between leftist grammars and insertion-deletion systems of size $(1, 1, 0; 1, 1, 0)$.

Besides introducing leftist grammars as a tool for addressing accessibility issues, the work [93] discusses the membership problem for these grammars and shows that it is decidable [93, Theorem 3.1]. The authors give a constructive proof by providing an algorithm for deciding the membership of a word in the language recognised by a leftist grammar.

The paper [64] establishes a series of correspondences between arbitrary and restricted versions of leftist grammars, and the levels of the Chomsky hierarchy. One of the most interesting results of this work is that leftist grammars can recognise non-context-free languages [64, Theorem 6]. The proof is constructive and shows a rather sophisticated grammar which, essentially, recognises the language $\{(a_1a_0)^m(F_0F_1)^n \mid n \geq 2^{2m-2}\}$. This result is particularly interesting given that insertion-deletion systems of size $(1, 1, 0; 1, 1, 0)$, and hence leftist grammars, cannot generate some very simple regular languages like $(ab)^+$.

The paper [62] continues the quest for a better apprehension of time complexity of the membership problem for leftist grammars and gives a lower bound: deciding the membership for this class of recognising devices is PSPACE-hard. In the proof of the main result (Theorem 2 and Corollary 1) the author gives a polynomial-time equivalence of the membership problems for leftist grammars and for linear-bounded automata. In a later article [63], the same author shows that membership problem for leftist grammars is not primitive recursive. A constructive proof is given: the paper describes a leftist grammar indirectly computing a function related to Ackermann's function, which dominates any primitive recursive function.

A novel take on leftist grammars was adopted in [21]. The authors remark that earlier results concerned with the complexity of membership relied on very sophisticated constructions, which were apparently based on modular approaches, but ultimately appeared as a very complex whole. In their work, they define leftist transformers which represent generalised leftist grammars transforming languages. The paper also defines how and when such transformers can be composed, and uses the compositional approach to model 3SAT and to thus show that bounded reachability is NP-complete for leftist grammars.

In Section 3.4 of this thesis we introduce a graphical instrument for visual analysis of derivations of leftist grammars and insertion-deletion systems of size $(1, 1, 0; 1, 1, 0)$: derivation graphs. We then use this tool to illustrate some of the constructions from [62, 63, 64]. It turns out that graphical representation of derivations gives interesting insight into the interactions between the rules, and also allows formulating certain local properties clearly and concisely. Furthermore, derivation graphs help perform non-trivial reasoning and adjustment of the rules to assure that the insertion-deletion system has the required behaviour.

Controlled Insertion-deletion Systems. Even though insertion-deletion systems alone are powerful enough to generate all recursively enumerable languages, adding some control over how such rules are applied presents interest in view of further reduction of minimal rule size required for computational completeness. Often considered control mechanisms for insertion-deletion systems are graph control, in which rules are assigned labels (states) prescribing the possible sequences of application, and P systems.

P systems, or membrane systems, are a biologically inspired model of computing originally introduced by Gheorghe Păun in [99] with focus on the compartmentalised structure of the living cell and on the membranes that separate the compartments. The cell is represented by a hierarchical structure of regions containing multisets of objects and multiset rewriting rules. Many variations to the basic model are possible, of which the greater part can be captured by the general formalism of networks of cells [41]. One of such variations are insertion-deletion P systems, in which regions contain strings and insertion-deletion rules capable of moving strings across the membrane structure. Because grouping rules by membranes can be seen as assigning labels to rules, insertion-deletion P systems can be seen as a variant of graph-controlled insertion-deletion systems, in which the graph of labels is a tree.

Rather unsurprisingly, adding graph control strictly increases the expressive power of insertion and deletion rules. Thus, the article [3] shows that graph-controlled insertion-deletion systems of size $(2, 0, 0; 1, 1, 0)$, $(1, 1, 0; 2, 0, 0)$, and $(1, 1, 0; 1, 1, 0)$ are computationally complete, while standalone insertion-deletion systems of the same sizes are not. Moreover, only five labels arranged in a linear control graph are necessary to achieve the same power as that of arbitrary grammars. Interestingly, graph-controlled insertion-deletion systems of size $(2, 0, 0; 2, 0, 0)$ are still not computationally complete, and cannot even generate the regular language a^*b, because no control over the location of insertion and deletion sites can be exercised [75, Theorem 17].

It turns out that graph control plus some simple ingredients can substantially boost the power even of insertion and deletion of minimal size, $(1, 0, 0; 1, 0, 0)$. The work [4] shows that graph-controlled insertion-deletion systems with rules of this size and priority of deletion over insertion can generate Parikh images of recursively enumerable languages. Priority of deletion is a necessary prerequisite, since graph-controlled insertion-deletion systems of size $(1,0,0;1,0,0)$ without priorities only generate Parikh images of matrix languages [4, Theorem 1].

Another classical control mechanism in the theory of formal languages which was also considered for insertion-deletion systems is matrix control. In matrix controlled insertion-deletion systems, introduced in [97], the rules are grouped into ordered sequences, called matrices. A matrix can be applied to a string w if all of its rules can be applied to w, in order. It turns out that insertion and deletion rules of sizes $(1,1,0;1,1,0)$ and $(1,1,0;1,0,1)$ achieve computational completeness with matrices containing at most three rules (Theorems 2 and 3 of [97]), while uncontrolled versions of such systems are not computationally complete (e.g. [73, Section 4.5]). Moreover, even systems of sizes $(1,0,0;1,1,1)$ and $(1,1,1;1,0,0)$ can generate all recursively enumerable languages, but with larger matrices of length at most eight (Theorems 5 and 6 of [97]). Finally, the authors of [97] also show that already matrices containing at most two rules suffice to boost the power of insertion-deletion systems of sizes $(1,1,0;2,0,0)$ and $(2,0,0;1,1,0)$ to computational completeness.

Remark that matrix control can be seen as a weaker form of graph control. This means in particular that matrix insertion-deletion systems of size $(2,0,0;2,0,0)$ are not computationally complete, because the corresponding graph-controlled variants are not either [75].

In Chapter 4 we focus on one-sided insertion-deletion systems equipped with additional control mechanisms imported from the theory of formal languages and investigate their computing power. In Section 4.2 we reconsider the fact that graph-controlled insertion-deletion systems with 5 labels and with rules of size $(1,1,0;1,1,0)$ are computationally complete, and point out two symmetric trade-offs between the size of one-sided rules and the number of labels: we show that, with rules of size $(1,1,0;1,2,0)$ or $(1,2,0;1,1,0)$, computational completeness can be achieved using 3 labels only.

We then consider semi-conditional control, in which case two sets of words E and F, the permitting and the forbidding context conditions respectively, are assigned to the rules. A rule may be applicable to a string w if all elements of E are subwords of w, and none of the elements of F are subwords of w. The pair (i,j) is called the degree of a semi-conditional insertion-deletion system Γ, where i is the maximal length of a word in a permitting context of Γ and j is the maximal length of a word in a forbidding context of Γ. We prove that context-free semi-conditional insertion-deletion systems of size $(1,0,0;1,0,0)$ and degree $(2,2)$ are computationally complete; moreover, we show that this result is optimal in rule size, because semi-conditional insertion-deletion systems of the same degree and of size $(1,0,0;0,0,0)$ generate a subset of context-sensitive languages.

In Section 4.4 we investigate a weaker control mechanism: random context control. A random context insertion-deletion system is a semi-conditional insertion-deletion system of degree at most $(1,1)$. We show that such systems with rules of size $(2,0,0;1,1,0)$ are able to generate all recursively enumerable languages. We also point out a remarkable asymmetry of random context control with respect to swapping the sizes of insertion and deletion: we prove that random-context systems of size $(1,1,0;2,0,0)$ and, more generally, of size $(1,1,0;p,1,1)$, are not computationally complete. This is a clear deviation from the tendency to symmetry with respect to size swapping observable in Table 1.3.

The results on semi-conditional and random context insertion-deletion systems were published in [58], while those on insertion-deletion P systems in [59].

1.2 Networks of Evolutionary Processors

Networks of evolutionary processors (NEPs) are a computing device consisting of some simple string processors connected in a network. Table 1.4 describes the computational power of NEPs with various ingredients.

Table 1.4: A summary of results about NEPs. The notations *ins*, *del*, and *subs* stand for "insertion", "deletion", and "substitution", respectively; ins_R and del_R refer to insertions and deletions at the right end of the string; *rc* and *reg* stand for "random context" and "regular", respectively; "fast NP" stands for "solves NP-complete problems in linear time".

	Rules	Filters	Semantics	Power	Reference
1	$0L$	X	*copy*	$= EXT0L$	[25, Theorem 4.1]
2	ins_R, del_R, $subs$	*rc*	*move*	fast NP	[18, Theorem 1]
3	*ins*, *del*, *subs*	*reg*	*move*	$= RE$	[19, Theorem 1]
4	*ins*, *del*, *subs*	*rc*	*move*	$\ni L_{abc}$, $\not\ni a^*b^*$	[19, Theorem 4]
5	*splicing*	*rc*	*move*	$= RE$	[22, Lemma 1]

The history of NEPs starts with the introduction of networks of parallel language processors in [25]. These networks represent collections of $0L$ Lindenmayer systems (language processors), each equipped with input and output filters. The filters may be random context, specifying which symbols the string must or must not contain, or regular, specifying which form the string must or must not have. The processors contain sets of strings which evolve according to the corresponding $0L$ rules during rewriting steps. During communication steps, every processor broadcasts a copy of those strings which pass its output filter and, from all the strings broadcast by the other processors, takes in those which pass its input filter (copy semantics). A computation of a network of language processors alternates between communication steps and rewriting steps, and the result is taken to be the language of strings appearing in a designated output node. Networks of language processors with filters of type X (random context or regular) are equivalent in power to conditional $ET0L$ systems relying on conditions of the same type X [25, Theorem 4.1].

The work [18] simplifies the operations which can be carried out by a single processor to one-symbol insertion $\lambda \to b$, deletion $a \to \lambda$, and substitution $a \to b$, each processor being only allowed to contain rules of the same type. Moreover, insertions and deletions must happen at the right end of the string. Language processors which such simple operations are called evolutionary processors, because one-symbol modifications of strings can be seen as corresponding to point mutations in DNA. Furthermore, a different communication semantics is considered: the strings are not copied around, but are moved instead (move semantics). Thus, if a string passes the output filter of a processor N, it is sent out and removed from N. As simple as they are, such networks are capable of solving some intractable problems efficiently. The paper [18] constructs a network solving an instance of the bounded Post correspondence problem (bounded PCP), known to be NP-complete, in linear time.

The article [19] considers yet another variation to NEPs, in which insertions and deletions may happen anywhere in the string. A fixed communication graph is also introduced, giving the topology of interconnections between the processors of the network; the strings emitted by a processor will thus only reach the nodes it is connected to, instead of being broadcast to all nodes. The authors then show that the generative power of such NEPs equipped with regular filters instead of random context filters is equal to that of arbitrary grammars, and that computational completeness can be achieved with five nodes already and with a complete communication graph [19, Theorem 1].

The work [19] also considers networks of evolutionary processors with random context filters and gives a partial characterisation of their power by showing how such a NEP can generate the non-context-free language $L_{abc} = \{w \in \{a, b, c\}^* \mid |w|_a = |w|_b = |w|_c\}$, where $|w|_x$ is the number of occurrences of x in the string w. Yet, even the regular language a^*b^* cannot be generated by such NEPs because random context filters cannot be used to check the local structure of a string and cannot thus have any control over the sites insertions happen at [19, Theorem 4].

The article [5] improves on the result from [19] by showing how NEPs with only four nodes can generate all recursively enumerable languages. The paper also considers a convenient extension of NEPs, called mixed NEPs (mNEPs), in which a processor is not limited to performing only one type of operation. In the case of mixed NEPs, considerable computational power can already be achieved with one node: Theorem 1 of [5] shows that, by picking only the terminal strings from the language generated by a one-node mNEP, one can obtain any given recursively enumerable language [5, Theorem 1]. Therefore, adding one more node which will only accept strings of terminals thereby filtering out the intermediate sentential forms makes it possible to achieve computational completeness with a two-node mixed NEP [5, Corollary 2].

In [22], evolutionary processors carrying out the splicing operation are considered. A splicing rule over the alphabet V is the quadruple $r : (u_1, u_2; u_3, u_4)$ of strings over V. Applying r to a pair of strings $(x_1u_1u_2x_2, y_1u_3u_4y_2)$ results in the pair $(x_1u_1u_4y_2, y_1u_3u_2x_2)$. Lemma 1 of [22] shows that networks of evolutionary processors with such splicing rules and random context filters achieve computational completeness with two nodes only.

In Section 4.5 of the present work we explore the possibilities of constructing small universal networks of evolutionary processors, targeting the reduction of the number of rules. We show that there exists a universal mixed NEP which generates all recursively enumerable sets of numbers with only 4 rules, and a universal mixed NEP which generates all recursively enumerable languages with 7 rules. These results were published in [57].

1.3 Universal Petri Nets

Petri nets are a modelling language originally introduced by Carl Adam Petri in 1966 in his doctoral thesis [98]. A distinctive feature of Petri nets as compared to other modelling languages is a simple and intuitive graphical representation, which explains in part the considerable amount of attention Petri nets have received ever since their introduction [95, 105, 121]. Petri nets have been especially attractive to researchers focusing on parallel and concurrent systems [26, 105].

A *Petri net* is usually defined as a tuple $N = (P, T, W, M_0)$, where P is the set of places possibly containing tokens, T is the set of transitions, W describes how arcs connect places to transitions, and M_0 is a vector giving the initial contents of the places. Arcs are only allowed to connect places to transitions, i.e. the underlying multigraph of the Petri net is bipartite. Graphically, places are represented as hollow circles, tokens as small bullets, transitions as filled rectangles or squares, and arcs as arrows. A configuration of a Petri net, i.e. the vector M giving the number of tokens in each place of the net, is called a *marking* of the net. Thus M_0 is also referred to as the initial marking.

Consider, for example, the Petri net shown in Figure 1.3. The places of this net are $P = \{Q_0, Q_1, Q_2, R_1, R_2\}$ and the transitions are $T = \{T_1, T_2\}$. Place Q_0 contains two tokens, and R_2 contains one token.

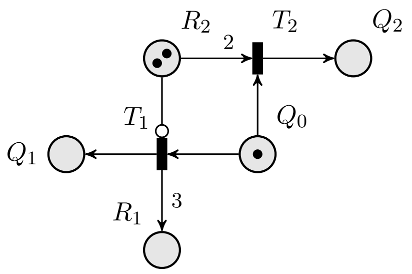


Figure 1.3: A Petri net with an inhibitor arc

Petri nets evolve by sequentially firing transitions. When a transition T fires, it consumes tokens from all the places P for which there is an arc from P to T, and adds tokens to all the places Q for which there is an arc from T to Q. Each arc corresponds to the production or consumption of single token, but multiple arcs are allowed between a place and a transition. For example, when transition T_2 in Figure 1.3 fires, it will consume two tokens from place Q_0 and one token from place R_2, and will add one token to place Q_2. If the places the transition is supposed to consume tokens from do not have enough tokens for the transition to fire, the transition is not enabled and cannot fire. For example, if place R_2 contained only one token instead of two, transition T_2 would not be enabled.

Sometimes the kind of net we define here is referred to as "place-transition net" or "PT net", while the term "Petri net" is used to refer to the class of PT nets which are not allowed to contain more than one token in a place [42]. In this work, we will use all three terms interchangeably.

A model which is very close to Petri nets are *vector addition systems* (often abbreviated as VAS), originally introduced in [68]. A VAS of dimension $n \in \mathbb{N}$ is defined to be the pair (w_0, W), where $w_0 \in \mathbb{N}^n$ is the start vector, and W is a finite set of vectors from $\mathbb{Z}^n$, called addition vectors [36, 121]. An addition vector $w \in W$ is said to be enabled in a vector $x \in \mathbb{N}^n$ if $x + w \in \mathbb{N}^n$, i.e. all the components of the vector $x + w$ are non-negative. A VAS evolves from the start vector w_0 by sequentially iterating the addition of vectors from W.

Vector addition systems are very similar to Petri nets, because a configuration of a Petri net is described by a vector (the marking), and the transitions add to or subtract values from the components of this vector. The translation of Petri net transitions to addition vectors is not immediate in the general case, however, as addition vectors cannot adequately capture self-loops, i.e. situations in which a transition T and a place P are connected by two arcs going in opposite directions. Nevertheless self-loops can be pretty straightforwardly avoided by decomposing the place P into two places and the transition T into two transitions and carrying out the action of the original transition T in two steps [48, Figure 3]. The paper [48] also gives additional details to the correspondence between VAS and Petri nets (which are referred to as "generalised Petri nets").

A model which at first looks quite different from vector addition systems and Petri nets, but is easily shown to be in the same family, is *multiset rewriting systems*. A multiset is essentially an unordered container with repetitions of elements. A multiset rewriting rule $w \to v$ substitutes the submultiset w for the submultiset v. (see Section 2.1 for a formal definition). Because a multiset can be seen as a numeric function associating the number of repetitions (multiplicities) to elements, any marking of a Petri net can be interpreted as a multiset over the alphabet of its places, in other words, as a function which associates to each place the number of tokens it contains. Furthermore, the semantics of multiset rewriting rules and Petri net transitions are the same: symbols (tokens) are first removed, then added. This similarity renders translations between the two models straightforward.

A lot of effort has gone into the analysis of properties of Petri nets, and some of them, like boundedness [68] and reachability [88], were proved to be decidable. While this is often very useful and makes it possible to algorithmically deduce important information about the modelled system, it also limits the class of systems which can be modelled by Petri nets [2]. As an example, the work [80] focuses on the class of functions which can be weakly computed by Petri nets (and hence VAS and VASS), i.e. functions $f : \mathbb{N} \to \mathbb{N}$ for which there exists a Petri net N such that, for any $n \in \mathbb{N}$, there exists a computation of N with input n and output $f(n)$, and, for any computation with input n, the output r satisfies $r \leq f(n)$. The authors of [80] show that such weakly computable functions cannot be sublinear, which means, for example, that the square root or the logarithm are not weakly computable.

To overcome intrinsic modelling limitations, a number of extensions were introduced to Petri nets. One of them is called "coloured Petri nets" and consists, in its simplest form, in assigning "colours" to individual tokens. Then, transitions are only able to fire when the places they are to consume tokens from have tokens of certain colours and in certain quantities. Coloured Petri nets were originally introduced in [60], and the author immediately remarks that just allowing different species of tokens does not augment modelling power, because, essentially, a place containing one red token and two blue tokens can be seen as two places containing one and two uncoloured tokens [60, p. 22]. However, more recent definitions of coloured Petri nets include the possibility of assigning conditions (guards) to transitions [61, 77], which allow a more fine-grained control over the circumstances under which transitions may fire, and thus boost the expressive power of these devices to computational completeness (because guards can be used to simulate inhibitor arcs).

Another extension introduced in the doctoral thesis [49] is the so-called prioritised Petri net, in which the transitions are equipped with a partial order ($<$), and, when two transitions T_1 and T_2 are enabled for a given marking of the net, T_2 is allowed to fire only if $T_1 \not< T_2$. A thorough analysis of the properties of prioritised Petri nets is conducted in [10].

An extension which does not immediately seem to lead to undecidability of reachability is reset arcs. If a transition T is connected to a place P by a reset arc,

P will be emptied whenever T fires. It turns out that reachability is undecidable for Petri nets with such arcs [7]. Moreover, boundedness for such nets is not decidable either: the work [32], for example, proves this fact by showing how to construct a Petri net with reset arcs which is bounded if and only if the register machine it is related to is bounded.

A yet another extension of Petri nets are the inhibitor arcs, originally suggested in [95]. If an inhibitor arc connects place P to transition T, then T may only be enabled when P is empty. Graphically inhibitor arcs are represented by a small hollow circle in place of a normal arrowhead. For example, transition T_1 in Figure 1.3 is connected to place R_2 with an inhibitor arc, which means that T_1 cannot fire in the marking shown in the figure, even though all other conditions for T_1 to be enabled are satisfied (i.e. place Q_0 contains enough tokens). Inhibitor arcs boost the power of Petri nets to computational completeness, because such nets can simulate register machines (e.g. [106]).

It follows from the equivalence of VAS and Petri nets that the two models have the same limitations; indeed, the set of vectors a VAS can generate is decidable [88]. Therefore, somewhat in parallel to extensions of Petri nets, several variations to the VAS model were considered. One of the simplest extensions are vector addition systems with states (VASS) introduced in [52], which add finite state control to VAS. It turns out that adding states to vector addition systems is only a matter of convenience and does not increase the computing power. Indeed, [52, Lemma 2.1] shows that any n-dimensional VASS can be simulated by a $(n+3)$-dimensional VAS. The proof idea consists in storing numbers coding the current state of the simulated VASS in the three additional components and using them to guide the application of addition vectors.

Another extension considered for VAS is the possibility of testing certain vector components for zero. In a VAS with zero tests (a VAS_0), each addition vector w is assigned a (possibly empty) set of indices of the components of x which should be zero for w to be applicable to x. Such zero tests are the exact counterpart of inhibitor arcs in Petri nets. The paper [14] studies some important properties of vector addition systems with one zero test only. Other properties of these systems can be directly derived from the results formulated in the language of Petri nets with inhibitor arcs (e.g. [106]).

Just as in the case of Petri nets and VAS, multiset rewriting systems can be augmented with the possibility of testing for "zero", i.e. requiring that certain elements should *not* be present in the multiset. Similarly to VAS_0, a set of inhibitors F is associated with every multiset rewriting rule, which is sometimes written as $w \to v|_{\neg F}$. Such a rule can only be applied to a multiset x if w is a submultiset of x and, additionally, none of the objects from the set F appear in x. Furthermore, inhibitors may be taken to be multisets instead of sets. A rule $w \to v|_{\neg z}$ with such a multiset inhibitor z can be applied to x if w is a submultiset of x and z is not a submultiset of x. In Petri nets, multiset inhibitors would correspond to "inhibitor" arcs testing that a certain place contains more than a fixed number of tokens. The paper [15] discusses the idea of inhibitors and some other extensions to multiset rewriting rules in the context of P systems.

In many P system variants, and especially in those in which the membrane structure is fixed throughout the whole computation, the hierarchical disposition of regions can be discarded by attaching region labels to objects and transforming region-specific actions into label-specific ones, in particular sending objects across membranes can be modelled by rewriting the labels on objects [39]. Such a flattening of the membrane structure effectively reduces the corresponding P system models to multiset rewriting, which means that many results can be freely brought over from one of these models to the other, but also translated to Petri nets and VAS.

We have already seen that multiset rewriting systems are very closely related to Petri nets and vector addition systems. The same conclusion can be drawn about the relationship of P systems to the latter two models, but with an important notice. While in Petri nets and VAS sequential semantics is often preferred, in P systems maximal parallelism is more frequently found. A number of different evolution modes were also considered for P systems, among which the minimal parallelism (at least one rule, if possible, must be applied in each region [23]), the asynchronous mode (no restrictions on the rule choice is imposed [34]), the clock-free mode (rules are assigned random real-valued duration [20]), etc. Furthermore, a considerable number of extensions (ingredients) have been proposed for P systems [35, 104], among which P systems with active membranes, whose membrane structure may be modified by rules [100]. A formal framework capturing many of these ingredients is given in [41]. These observations lead to the conclusion that, in the area of P systems, the focus is principally on studying the computational power of variations to the basic model, while in the case of Petri nets more attention is given to applications to modelling distributed systems. VAS on the other hand have served as a convenient alternative representation of Petri nets for studying some formal properties like reachability or boundedness [72, 79, 88, 121].

We will conclude the series of introduced models with *register machines*. Register machines spun off the famous Turing machines introduced by Alan Turing in [114]. In his paper [118], Hao Wang showed that it is possible to formulate an even simpler computing model by reducing the tape alphabet to two symbols (including the blank symbol) and relying only on four instructions: move head left, move head right, mark the current cell, and conditional jump, depending on whether the cell the head is currently on is empty or not. In [90], Marvin Minsky essentially cut the tape of Wang's machine into two by proposing a Turing machine with two completely blank tapes, except for the left end markers, and with heads which cannot modify the contents of the cells. The two heads of Minsky's two-tape Turing machines can be effectively used as counters: moving the head corresponds to incrementing or decrementing the counter, while sensing the left end corresponds to zero test.

A different, though ultimately equivalent, approach was taken by Zdzislaw Melzak in [89] and Joachim Lambek in [78]. Their constructions are based on a finite set of distinguishable locations (holes) and an infinite supply of indistinguishable counters (pebbles). Melzak used relatively complex instructions; for example, he allowed subtraction and addition of any number of pebbles. Lambek, on the other hand, simplified the instruction set to atomic increments and decrements with zero test, which reminds Minsky's machines.

A yet another way to introduce register machines comes from the generalisation of push-down automata to machines with multiple stacks [9]. This point of view allows defining a register machine as a language-accepting device (sometimes also called a counter automaton) whose push-down stores may only contain one type of symbol. Since two stacks suffice for achieving the recognising power of Turing machines [9, 92], considering restricted versions of such multi-push-down devices is of interest. Thus in [47] two such restrictions are discussed. The stronger of them, blindness, allows the registers to contain negative integers, but no information as to their contents is made accessible to the machine. A blind counter automaton accepts the input string if, when it reaches its right end, all of its counters are zero. The other restricted construction, the partially blind counter automaton, is a blind counter automaton whose registers are not allowed to contain negative values. Thus a partially blind counter automaton has somewhat more information about the contents of its registers: it knows they always contain non-negative numbers. It was shown in [47] that blind counter automata are strictly less powerful than partially blind ones, which in their turn are strictly less powerful than Turing machines.

A modern definition of a register machine (as found, for example, in [71]), includes a set of registers, a set of states (or instruction labels), and a set of instructions associated with each state. The most widely used are the increment and the decrement-and-zero-test instructions. Sometimes the latter instruction type is decomposed into separate decrement instruction and zero test instruction [71].

Although the historical origins of register machines are completely distinct from those of Petri nets, VAS, or multiset rewriting systems, all these four models are fundamentally similar: indeed, the registers of a register machine can be seen as forming a vector of non-negative integers. Any instruction of a register machine can be easily simulated by a Petri net transition with inhibitor arcs, or a multiset rewriting rule with inhibitors. The converse statement is true modulo "state expansion": a register machine can simulate a Petri net transition (respectively, a multiset rewriting rule) using a sequence of instructions whose cumulative effect corresponds to the action of the transition (respectively, multiset rewriting rule).

Even though the usual semantics of both register machines and Petri nets is sequential, register machines are most often defined as deterministic devices, as different from Petri nets which do not usually prohibit non-deterministic choice. To achieve non-determinism nevertheless, some authors (e.g. [36]) define two possible exit states for an increment instruction, allowing the machine to chooses between them non-deterministically.

One can relate counter automata as language-recognising devices to register machines as number-manipulating devices by restricting the former to one-letter languages and treating the number of symbols on the input tape as the input value in a specially designated register. Under such a convention, Petri nets *without* inhibitor arcs and multiset rewriting systems *without* inhibitors correspond to partially blind counter automata.

An important question in the study of register machines is universality. A register machine is called universal, if, given the code of any other register machine M, it can reproduce the output of M for any of its inputs. The problem of universality was originally posed by Alan Turing himself for Turing machines in [114]. Later, in [111], Claude Shannon launched the quest for the smallest possible Turing machine which would still retain universality. In his article, Shannon actually constructs a two-state universal Turing machine. A series of important results in this direction were obtained later [91, 108, 119].

Even though register machines are computational devices which branched directly out of Turing machines, and in spite of the fact that their computational completeness was shown at the moment of introduction [78, 89, 92], a lot less research has gone into finding small and universal representatives of this class until recently. In [71], Ivan Korec constructed a series of such small universal register machines, including one with only 8 registers and 22 commands.

Because of the strong connection between register machines and P systems, most of the important universality results in the latter domain were obtained by relating certain P system models to Korec's universal register machines. For example, in [24], the authors construct three universal P systems by simulating Korec's machines, and highlight a trade-off between the maximal number of objects a rule is allowed to involve and the total number of rules. An enhancement is suggested in [40], where an even smaller universal P system is described. Another universality result, this time formulated in terms of multiset rewriting, is given in [6]. In this paper, a number of atomic minimisation strategies are introduced, which are then used to construct a universal multiset rewriting system with 23 rules only.

In Petri nets with inhibitor arcs, the first universality construction was carried out by Dmitry Zaitsev in [122]. This paper describes a universal Petri net with inhibitor arcs, which directly simulates any given Petri net (with inhibitor arcs). The author devises a way to encode the structure of the simulated net into a number and then describes the simulation algorithm, which is subsequently implemented in the universal Petri net. This construction uses 500 places and as many transitions.

Later, the same author builds a much smaller universal Petri net with 14 places and 29 transitions [123]. The new net simulates a universal Turing machine instead of directly reproducing the activity of an arbitrary Petri net. The described universal net uses inhibitor arcs, but also implicitly assumes a total order of priorities on transitions [123, p. 2].

In Chapter 6, we continue the research of universality of Petri nets with inhibitor arcs by relating universal Petri nets to universal register machines, instead of simulating other nets directly or relying on Turing machines. To capture the differences in structural complexity between the described universal Petri nets, we define a measure of size of a Petri net as a 4-tuple including the number of places, transitions, inhibitor arcs, and the maximal transition degree. We then give a series of strategies for reducing each of the size parameters, as well as characterisations of concrete universal Petri nets.

To this end, in Chapter 5, we introduce generalised register machines, which can be seen as bridging register machines and Petri nets. Essentially, such generalised constructs represent graphs of states, with multiple conditions and operations on registers attached to the edges. An arc of this graph can therefore check whether a register is empty or not, as well as increment or decrement a register. Because such an arc can perform several register checks or operations at once, generalised register machines achieve universality with less states than conventional register machines. We describe universal generalised register machines with seven states only, and then show Petri nets simulating them.

We then show that further room for improvement exists in non-deterministic simulation approaches and describe a non-deterministic simulation strategy which allows cutting down dramatically on the number of places of the simulating Petri net. In fact, only as many places are required to reproduce the activity of any register machine as there are registers, plus two additional places. We then show how the same reduction of the number of places can be achieved without obliterating the determinism inherited from Korec's machines [71].

We remark that the Petri nets we describe in Sections 6.2 and 6.5 achieve universality with the minimal possible values for the transition degree and the number of inhibitor arcs. We also conjecture that the values for other size parameters achieved by the Petri nets from the other subsections cannot be significantly improved upon because of inherent limitations of the model.

The work on deterministic universal Petri nets was published as [56], while the questions concerning non-determinism were treated in [55]. Chapters 5 and 6 summarise and improve upon the results presented in the mentioned publications.

Chapter 2

Preliminaries

In this chapter, some basic definitions and notations used throughout the thesis are given. Section 2.1 provides the basics of the terminology from the domain of formal languages, Section 2.2 discusses finite automata, register machines, and Turing machines, while Section 2.3 recalls the notions of computational completeness and universality.

2.1 Formal Languages

We denote the set of natural numbers by $\mathbb{N}$ and the set of non-zero natural numbers by $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$. The empty set is written as $\varnothing$. The set of all subsets of X is denoted by 2^X. The number of elements in X (cardinality) will be written as $|X|$.

Any non-empty finite set of symbols V is called an *alphabet*. A finite word over V is any finite sequence of symbols from V. The length of a word is written as $|w|$; the notation $|w|_a$ refers to the number of occurrences of the symbol a in w. The set of all words over V of length n is denoted by V^n. By definition, the only word of zero length is the empty word λ: $V^0 = \{\lambda\}$. The set of all finite words over V is denoted by $V^* = \cup_{n\in\mathbb{N}} V^n$, and the set of non-empty finite words by $V^+ = \cup_{n\in\mathbb{N}^+} V^n = V \setminus \{\lambda\}$. Any subset of V^* is called a *language* over V. We will sometimes identify a singleton language $L = \{w\} \subseteq V^*$ with its only representative.

The *reverse* of the word $w = x_1 \dots x_n$ is $rev(w) = x_n \dots x_1$. The reverse of the language L is naturally defined as $rev(L) = \{x_n \dots x_1 \mid x_1 \dots x_n \in L\}$.

The *shuffle product* of two words over V is defined as the set of all possible ways of interleaving their letters (rifle shuffling them). Formally, for $w, v \in V^*$ and $a, b \in V$, the operation ($⧢$) is defined inductively as follows:

$$\begin{aligned} \lambda ⧢ v &= \{v\}, \\ w ⧢ \lambda &= \{w\}, \\ wa ⧢ vb &= \{ua \mid u \in w ⧢ vb\} \cup \{zb \mid z \in wa ⧢ v\}. \end{aligned}$$

Thus, $a ⧢ x = \{ax, xa\}$, $a ⧢ xy = \{axy, xay, xya\}$, and $ab ⧢ xy = \{abxy, axby, xaby, xayb, axyb, xyab\}$. The shuffle operation can be naturally extended to languages: given $L_1, L_2 \subseteq V^*$, we define

$$L_1 ⧢ L_2 = \bigcup_{\substack{w_1 \in L_1 \\ w_2 \in L_2}} w_1 ⧢ w_2.$$

For an alphabet $V = \{a_1, \dots, a_k\}$ and a word $w \in V^*$, the vector of natural numbers $Ps(w) = (|w|_{a_1}, \dots, |w|_{a_k}) \in \mathbb{N}^k$ is called the *Parikh vector* of w. Given a language $L \in V^*$, the *Parikh image* of L is defined as $Ps(L) = \{Ps(w) \mid w \in L\}$. For a family of languages $\mathcal{X}$, its Parikh image is defined as $Ps\mathcal{X} = \{Ps(L) \mid L \in \mathcal{X}\}$.

A language of vectors of natural numbers $L \subseteq \mathbb{N}^n$ is called *linear*, if there exists such a finite subset $B \subsetneq \mathbb{N}^n$ that every element of L can be represented as a linear combination of the elements from B:

$$L = \Big\{ \sum_{b \in B} k_b b \mid k_b \in \mathbb{N} \Big\}.$$

A language $L \subseteq \mathbb{N}^n$ is called *semilinear* if it is a finite union of linear languages.

Given two finite alphabets V and U, a mapping $h : V \to U^*$, extended to $h : V^* \to U^*$ by $h(\lambda) = \lambda$ and $h(wv) = h(w)h(v)$, $w, v \in V^*$, is called a *morphism*. If $h(a) \neq \lambda$, for all $a \in V$, h is a λ-free morphism. If $|h(a)| = 1$, for all $a \in V$, h is called a *coding*; if $|h(a)| \leq 1$, for all $a \in V$, h is called a *weak coding*. The morphism $h : V \cup U \to V$ such that $h(a) = a$ for $a \in V$ and $h(b) = \lambda$ for $b \in U$ is called a *projection* of $V \cup U$ on V. Given a morphism $h : V^* \to U^*$, the mapping $h^{-1} : U^* \to 2^{V^*}$, defined as $h^{-1}(w) = \{v \in V^* \mid w = h(v)\}$ is called the *inverse* of h.

Given an alphabet $V = \{a_1, \dots, a_k\}$, a mapping $w : V \to \mathbb{N}$ is called a *multiset* over V. The value $w(a)$, for $a \in V$, is the multiplicity of a in w. The size of w is the sum of all multiplicities of elements in w: $|w| = \sum_{a \in V} w(a)$. The support of w is the set of elements which appear in it: *supp*$(w) = \{a \in V \mid w(a) > 0\}$. Note that any set can be seen as a multiset in which all the elements have multiplicity 1.

A finite multiset w over the alphabet V is often represented as the string $a_1^{w(a_1)} \dots a_k^{w(a_k)}$. For this reason, we will often abuse notation and use V^* to refer to the set of all finite multisets over V, and V^+ to refer to the set of all finite non-empty multisets over V.

A multiset w_1 over an alphabet V is a submultiset of another multiset w_2 over the same alphabet, written as $w_1 \leq w_2$, if $w_1(a) \leq w_2(a)$, for all $a \in V$. The union of w_1 and w_2 is the multiset $w = w_1 + w_2$ with the property that $w(a) = w_1(a) + w_2(a)$, for all $a \in V$. If $w_1 \leq w_2$, the difference of w_2 and w_1 is the multiset $w = w_2 - w_1$ with the property that $w(a) = w_2(a) - w_1(a)$, for all $a \in V$.

A *sequential multiset rewriting system* is the three-tuple $\gamma = (O, T, w_0, P)$, where O is an alphabet, $T \subseteq O$ is the terminal alphabet, $w_0 \in O^*$ is the initial multiset, and P is a set of multiset rewriting rules of the form $w \to v$, with $w \in O^+$ and $v \in O^*$. A rule $r : w \to v$ is applicable to a multiset $x \in O^*$ if $w \leq v$. An application of this rule to x yields the new multiset $y = x - w + v$; this fact is denoted by $x \overset{r}{\Rightarrow}_\gamma y$. In general, the multiset x is said to derive y, written as $x \Rightarrow_\gamma y$, if there exists such a rule $r \in P$ that $x \Rightarrow_\gamma y$. The notation $(\Rightarrow_\gamma^*)$ refers to the reflexive and transitive closure of $(\Rightarrow_\gamma)$. The language of multisets generated by γ is defined as $L(\gamma) = \{w \in T^* \mid w_0 \Rightarrow_\gamma^* w\}$.

A *string rewriting grammar* is a construct $G = (N, T, S, P)$, where N is the alphabet of non-terminal symbols, T is the alphabet of terminal symbols, S is the axiom, and P is a set of string rewriting rules of the form $\alpha \to \beta$, where $\alpha \in (N \cup T)^+$ and $\beta \in (N \cup T)^*$. A string $w\alpha v$ is said to derive $w\beta v$ by the rewriting rule $\alpha \to \beta$, which is denoted by $w\alpha v \Rightarrow w\beta v$. The language generated by G is defined by $L(G) = \{w \in T^* \mid S \Rightarrow w^*\}$, where $(\Rightarrow^*)$ is the reflexive and transitive closure of $(\Rightarrow)$.

If no restrictions are imposed on the rules in P, G is called is type-0 grammar; the family of languages generated by such grammars is called recursively enumerable languages and is denoted by RE. Grammars with non-retracting rules, i.e. rules $\alpha \to \beta$, where $|\alpha| \leq |\beta|$, are called type-1 or context-sensitive grammars; the family of languages generated by such grammars is called context-sensitive languages, and is denoted by CS. Grammars with rules of the form $A \to \beta$, with $A \in N$ and $\beta \in (N \cup T)^*$, are called context-free; the family of languages they generate is called context-free languages and is denoted by CF. Finally, grammars with rules of the form $A \to bB$, where $A \in N$, $B \in N \cup \{\lambda\}$, and $b \in T$, are called regular grammars; the languages they generate are called regular, denoted by REG.

A special subfamily of context-free grammars are semilinear grammars, in which all rules have the form $A \to uBv$, with $u, v \in T^*$, $A \in N$, and $B \in N \cup \{\lambda\}$. The family of languages generated by such grammars is called semilinear languages and is denoted by $SLIN$. It is well-known that $REG \subsetneq SLIN \subsetneq CF$, but $PsREG = PsSLIN = PsCF$ [104, 109].

A type-0 grammar $G = (N, T, S, P)$ is said to be in *Geffert normal form* [46], if its alphabet of non-terminals N is defined as $N = \{S, A, B, C, D\}$, T is a terminal alphabet, and P only contains context-free rules $S \to uSv$, with $u \in \{A, C\}^+$ and $v \in (T \cup \{B, D\})^+$, as well as $S \to \lambda$, and two (non-context-free) erasing rules $AB \to \lambda$ and $CD \to \lambda$.

According to [46], the generation of a string using a grammar in Geffert normal form is performed in two stages. During the first stage, only context-free rules $S \to uSv$ can be applied; this follows from the fact that $u \in \{A, C\}^+$ and $v \in (\{B, D\} \cup T)^+$. During the second stage, only non-context-free rules can be applied, because the symbol S is no longer present in the string. The transition between the stages is done by the rule $S \to \lambda$ (in [46] a set of rules of the form $S \to uv$ is used instead, leading to an equivalent result). Note that the symbols A, B, C, D are treated like terminals during the first stage and so each rule $S \to uSv$ is, in a sense, "linear".

It is possible to decompose the context-free rules of a grammar in Geffert normal form into rules with right-hand sides containing at most two symbols: the result of an application of a rule $r : S \to a_1 \dots a_n \, S \, b_1 \dots b_m$ can be reproduced by the following shorter rules:

$$\begin{aligned} S &\to a_1 X_1^{(r)}, \quad X_{i-1}^{(r)} \to a_i X_i^{(r)}, \quad 2 \leq i \leq n, \\ X_n^{(r)} &\to Y_m^{(r)} b_m, \quad Y_{j+1}^{(r)} \to Y_j^{(r)} b_j, \quad 2 \leq j \leq m-1, \\ Y_2^{(r)} &\to S b_1. \end{aligned}$$

Thus, for any type-0 grammar $G = (N, T, S, P)$, it is possible to construct an equivalent grammar $G_{nf} = (N_{nf}, T, S, P_{nf})$, where $N = N' \cup N''$, $N' \cap N'' = \varnothing$, $N'' = \{A, B, C, D\}$, N' contains S and the symbols $X_i^{(r)}$ and $Y_j^{(r)}$ for each rule of G, and P contains rules of the following four forms:

- $X \to bY$, with $X, Y \in N'$, $X \neq Y$, $b \in N''$,
- $X \to Yb$, with $X, Y \in N'$, $X \neq Y$, $b \in N'' \cup T$,
- $UV \to \lambda$, with $(U, V) \in \{(A, B), (C, D)\}$,
- $S \to \lambda$.

The grammar G_{nf} is said to be in *special Geffert normal form.* Remark that even the deletion rule $S \to \lambda$ could be avoided in G_{nf} by considering rules of the form $Y_2^{(r)} \to b_1$.

2.2 Computing Devices

A *finite automaton* is the tuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is an alphabet of symbols, $\delta : Q \times \Sigma \to 2^Q$ is the transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. A configuration of A is the tuple (q, w), where $s \in Q$ and $w \in \Sigma^*$. A transition of A is guided by δ: if $p \in \delta(q, a)$, then $(q, ax) \vdash (p, x)$. The reflexive and transitive closure of the relation $(\vdash)$ is denoted by $(\vdash^*)$. A word is w accepted A if $(q_0, w) \vdash^* (q_f, \lambda)$, where $q_f \in F$ is a final state. Thus the language accepted by A is defined as $L(A) = \{w \in V^* \mid (q_0, w) \vdash^* (q_f, \lambda)\}$. Finite automata accept exactly the class of regular languages.

A Turing machine is the construct $T = (Q, \Sigma, a_0, q_0, F, \delta)$, where Q is a finite set of states, Σ is the tape alphabet, $a_0 \in \Sigma$ is the blank symbol, q_0 is the initial state, $F \subseteq Q$ is the set of final (halting) states, and δ is a transition function. A way of defining the transition function is by listing tuples of the form (q_i, a_k, q_j, a_l, D), which are interpreted as follows: if the head of T being in state q_i is scanning a cell which contains a_k, then the contents of the scanned cell is replaced by a_l, the head moves to the left ($D = L$) or to the right ($D = R$) and the state of the machine changes to q_j. We only consider deterministic non-stationary Turing machines in this thesis, i.e. deterministic machines which move their head at every step of the computation; it is known that these devices are computationally complete [108].

A configuration of T in which the machine is in state q, the contents of the tape is $w_1 a w_2$, where $w_1, w_2 \in \Sigma^*$ and $a \in \Sigma$, and the head is positioned on a, can be represented as the string $w_1 q a w_2$. In the initial configuration of a Turing machine, the input occupies a finite region of the tape. The situation in which the tape is empty is represented by the string $q a_0$ (a_0 is the blank symbol). When T is in the configuration $w_1 q_i a_k$ and the definition of δ contains a tuple (q_i, a_k, q_j, a_l, R), then the machine transitions into state $w_1 a_l q_j a_0$, that is a new empty tape cell is "added". The case in which the machine is on the leftmost non-empty cell of the tape and needs to move to the left is treated similarly. A halting configuration of a Turing machine is a configuration in which no transition rule is defined for the current state $q \in F$ and the symbol a the head is positioned upon.

A computation of a Turing machine T on the input word w is a sequence of configurations $\mathcal{C} = (C_i)_{0 \le i \le t}$, where $C_0 = q_0 w_0$ is the initial configuration and $w_0 \in \Sigma^*$ is the input word, $C_t = w_1 q_f w_2$ is a halting configuration, and $C_i \vdash C_{i+1}$, for all $1 \le i < t$. Since we only discuss deterministic Turing machines, the computation $\mathcal{C}$ is entirely determined by the initial configuration C_0. We denote the fact that T halts with the word $w_1 w_2$ on the tape when it starts with the input w_0 by $T(w_0) = w_1 w_2$. This notation is naturally extended to non-deterministic Turing machines by taking $T(w_0)$ to be the set of all possible tape contents in halting configurations of T in computations starting with w_0.

A (deterministic) *register machine* is defined as a 5-tuple $M = (Q, R, q_0, q_f, P)$, where Q is a set of states, $R = \{R_1, \ldots, R_k\}$ is the set of registers, $q_0 \in Q$ is the initial state, $q_f \in Q$ is the final state, and P is a set of instructions of the following three forms:

– *(increment)* $(p, A(i), q)$, where $p, q \in Q$, $R_i \in R$: being in state p, increment register R_i and go to state q;
– *(zero-check-and-decrement)* $(p, S(i), q, s)$, where $p, q, s \in Q, R_i \in R$: being in state p, try decrementing register R_i and go to q if successful or to s otherwise;
– *(stop)* $(q_f, Stop)$: halt the execution; may only be associated with the final state q_f.

In some sources (e.g., [55, 56, 71]), the increment is written as RiP, and the zero-check-and-decrement instruction as $RiZM$. The work [71] also uses other instruction types, like a separate decrement (p, RiM, q), which corresponds to $(p, S(i), q, q)$, or a separate zero-check (p, Ri, q, s), which works like zero-check-and-decrement, but does not modify the register: it corresponds to the pair of instructions $(p, S(i), q', s)$ and $(q', A(i), q)$.

Register machines are often represented graphically in flowchart notation; in this case the instructions checking a condition on a register are drawn as diamonds, while non-conditional instructions as rectangles. An example of such a graphical representation is shown in Figure 2.1. This figure reproduces the machine U_{22} from [71] and uses conventional instruction names.

A configuration of a register machine is given by $(q, n_1, \dots, n_k)$, where $q \in Q$ and $n_i \in \mathbb{N}, 1 \leq i \leq k$, describe the current state of the machine as well as the contents of all of its registers. A transition of the register machine consists in updating or checking the value of a register according to an instruction of one of the types above and in changing the current state to another one. We say that the machine stops if it reaches the state q_f. We say that M computes a value $y \in \mathbb{N}$ on the input $x_1, \dots, x_n$, $x_i \in \mathbb{N}$, $1 \leq i \leq n \leq k$, if, starting from the initial configuration $(q_0, x_1, \dots, x_n, 0, \dots, 0)$, it reaches the final configuration $(q_f, y, 0, \dots, 0)$.

2.3 Computational Completeness and Universality

Consider a class of number-generating devices $\mathfrak{C}_N$ and a class of string-generating devices $\mathfrak{C}_S$. For an element $A \in \mathfrak{C}_N$, we will denote the set of numbers produced by A by $N(A)$. Similarly, for an element $B \in \mathfrak{C}_S$, we will denote the language generated by B by $L(B)$. $\mathfrak{C}_N$ is called *computationally complete* if, for every recursively enumerable subset of natural numbers $X \in \mathbb{N}$, there exists an element $A_X \in \mathfrak{C}_N$ such that $X = N(A_X)$. $\mathfrak{C}_S$ is called computationally complete if, for every recursively enumerable language Y there exists an element $B_Y \in \mathfrak{C}_S$ such that $L(B_Y) = Y$.

We denote the fact that a number-manipulating computing device M_N produces the outputs from the set $N_x \subseteq \mathbb{N}$ for the input vector $x \in \mathbb{N}^k$, $k \in \mathbb{N}$, by $M_N(x) = N_x$. Similarly, the fact that a string-manipulating computing device M produces the output language L_x for the input word x will be denoted by $M(x) = L_x$.

An element $A_0 \in \mathfrak{C}$, for a class of computing devices $\mathfrak{C}$, is called *(weakly) universal* if it is capable of simulating any other device A in $\mathfrak{C}$ using an appropriate encoding. More precisely, if $A(x) = y$, then $A(x) = f(A_0(g(A), h(x)))$, where h and f are the encoding and decoding functions respectively, and g is the function assigning numbers to devices in $\mathfrak{C}$, according to some fixed enumeration (e.g. Gödel numbering). The encoding and decoding functions should not be too complicated, otherwise the universal element would be trivial. For example, if f is partial recursive, the entire simulation could be done by f and not by M_0. It is commonly

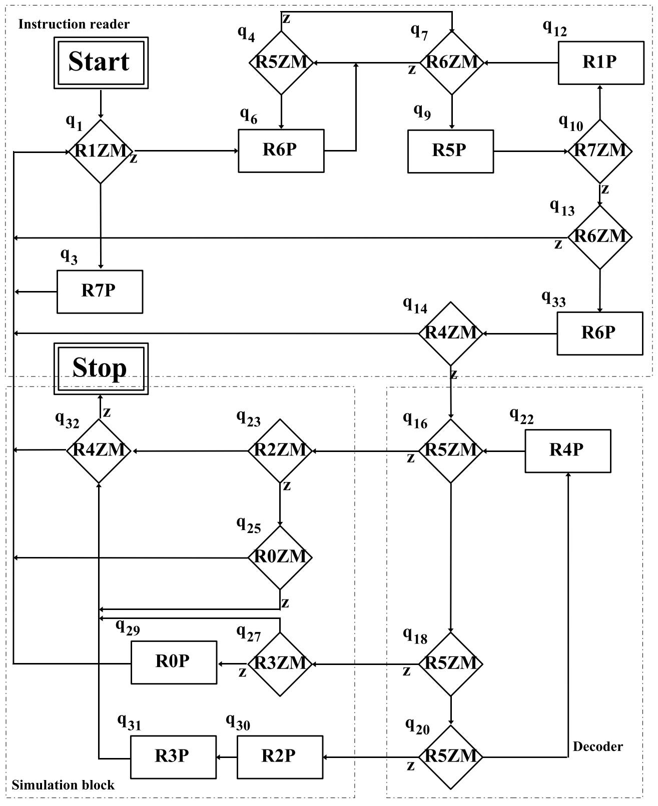


Figure 2.1: The flowchart of the strongly universal machine U_{22}. The symbol "z" indicates the transitions which happen when the checked register is zero.

admitted that recursive functions can be used for encoding and decoding. In the case of register machines, for example, the typically used functions are $f(x) = \log_2(x)$ and $h(x) = 2^x$. If the functions f and h are identities, the element A_0 is called *strongly universal.*

In our definition, the universal element A_0 appears to use two input values, while the other devices of $\mathfrak{C}$ use only one. This is mainly a matter of notation, however, because any pair of numbers or strings can be encoded in a single number or a single string respectively, by using a pairing function. To stress this fact, we could rewrite the definition of universality for A_0 in this way: $A(x) = f(A_0(\langle g(A), h(x) \rangle))$, where $\langle a, b \rangle$ is the pairing function.

The classes of register machines and Turing machines are known to contain universal elements [92, 111], and the same is true for any computationally complete class of computing devices. The converse is not true, because any singleton class of computing devices trivially contains a universal element, but is not necessarily computationally complete.

In this work we will rely heavily on universal register machines constructed by Ivan Korec in [71]. The flowchart of the strongly universal register machine with 22 commands is shown in Figure 2.1; its complete program follows.

$$\begin{array}{lll}
(q_1, R1ZM, q_3, q_6) & (q_3, R7P, q_1) & (q_4, R5ZM, q_6, q_7) \\
(q_6, R6P, q_4) & (q_7, R6ZM, q_9, q_4) & (q_9, R5P, q_{10}) \\
(q_{10}, R7ZM, q_{12}, q_{13}) & (q_{12}, R1P, q_7) & (q_{13}, R6ZM, q_{33}, q_1) \\
(q_{33}, R6P, q_{14}) & (q_{14}, R4ZM, q_1, q_{16}) & (q_{16}, R5ZM, q_{18}, q_{23}) \\
(q_{18}, R5ZM, q_{20}, q_{27}) & (q_{20}, R5ZM, q_{22}, q_{30}) & (q_{22}, R4P, q_{16}) \\
(q_{23}, R2ZM, q_{32}, q_{25}) & (q_{25}, R0ZM, q_1, q_{32}) & (q_{27}, R3ZM, q_{32}, q_1) \\
(q_{29}, R0P, q_1) & (q_{30}, R2P, q_{31}) & (q_{31}, R3P, q_{32}) \\
(q_{32}, R4ZM, q_1, q_f) & (q_f, STOP) &
\end{array}$$

Chapter 3

One-sided Insertion-deletion Systems

In this chapter we focus on the operations of insertion and deletion and, in particular, on one-sided versions of these operations with small context. In Section 3.2 we consider leftist systems and formally show how they are related to insertion-deletion systems of size $(1, 1, 0; 1, 1, 0)$. Then, in Section 3.3, we first remind the proof of the fact that systems of this size are not computationally complete and then show that systems with slightly bigger contexts – of sizes $(1, 1, 0; 1, 2, 0)$ and $(1, 2, 0; 1, 1, 0)$ – can generate all regular languages, and even some non-context-free ones. Furthermore, we prove that considering longer left contexts does not increase the expressive power of the insertion-deletion systems.

In Section 3.4, we introduce a novel approach to dynamic analysis of the derivations of systems of size $(1, 1, 0; 1, 1, 0)$ – derivation graphs – and show how it can be used to better understand the functioning of leftist grammars generating non-semilinear languages. We also give some ideas of how this tool could be extended to deal with larger deletion contexts.

3.1 Definitions

In this section we will formally introduce insertion-deletion systems and leftist grammars, as well as some related notions, such as leftmost and greedy derivations.

An *insertion-deletion system* is a construct $\Gamma = (V, T, A, I, D)$, where V is an alphabet, $T \subseteq V$ is the terminal alphabet, $A \subseteq V^*$ is a set of axioms, and I and D are finite sets of triples of the form (u, α, v), with $u, \alpha, v \in V^*$ and $\alpha \neq \lambda$. The symbols from $V \setminus T$ are called non-terminal symbols. The triples in I are called insertion rules, and those in D are called deletion rules. Instead of writing $(u, \alpha, v) \in I$ or $(u, \alpha, v) \in D$, we will often prefer the notations $(u, \alpha, v)_{ins}$ and $(u, \alpha, v)_{del}$ respectively.

An application of the insertion rule $r_i : (u, \alpha, v)_{ins}$ to a string of the form $yu\,vz$, $y, z \in V^*$, yields $yu\,\alpha\,vz$. Symmetrically, an application of the deletion rule $r_d : (u, \alpha, v)_{del}$ to the string $yu\,\alpha\,vz$ yields $yu\,vz$. Thus, the semantics of the insertion rule r_i corresponds to that of the rewriting rule $uv \to u\alpha v$, while the semantics of the deletion rule r_d to that of the rewriting rule $u\alpha v \to uv$.

By acting on strings, the insertion and deletion rules of Γ give rise to two relations on words: "derives by insertion rule r_i" $(\overset{r_i}{\Rightarrow}_{ins}) \subseteq V^* \times V^*$ and "derives by deletion rule r_d" $(\overset{r_d}{\Rightarrow}_{del}) \subseteq V^* \times V^*$. Thus, for r_i we can write $yu\,vz \overset{r_i}{\Rightarrow} yu\,\alpha\,vz$, and for r_d we can write $yu\,\alpha\,vz \overset{r_d}{\Rightarrow} yu\,vz$. We will often use the following less fine-grained relations:

$$(\Rightarrow_{ins}) = \bigcup_{r\in I}(\overset{r}{\Rightarrow}_{ins}), \quad (\Rightarrow_{del}) = \bigcup_{r\in D}(\overset{r}{\Rightarrow}_{del}), \quad \text{and} \quad (\Rightarrow) = (\Rightarrow_{ins} \cup \Rightarrow_{del}).$$

We will also use the notation $(\Rightarrow^*)$ to refer to the reflexive and transitive closure of $(\Rightarrow)$. We will sometimes use the notation $(\Rightarrow_\Gamma)$ to explicitly refer to the insertion-deletion system Γ whose rules induce the "derives" relation.

A derivation of the insertion-deletion system Γ is a sequence of consecutive insertion or deletion steps: $w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n$. Each word w_i, $1 \le i \le n$, is called a sentential form of this derivation. The language generated by Γ is defined in the following way:

$$L(\Gamma) = \{w \in T^* \mid w_0 \Rightarrow^* w, w_0 \in A\}.$$

The descriptional complexity of Γ is given by the vector $(n, m, m'; p, q, q')$, called *size*, with components given by the following:

$$\begin{aligned} n &= \max\{|\alpha| : (u,\alpha,v) \in I\}, & p &= \max\{|\alpha| : (u,\alpha,v) \in D\},\\ m &= \max\{|u| : (u,\alpha,v) \in I\}, & q &= \max\{|u| : (u,\alpha,v) \in D\},\\ m' &= \max\{|v| : (u,\alpha,v) \in I\}, & q' &= \max\{|v| : (u,\alpha,v) \in D\}. \end{aligned}$$

We use the notation $INS_n^{m,m'}DEL_p^{q,q'}$ to refer to the family of languages generated by insertion-deletion systems of size $(n, m, m'; p, q, q')$. An insertion-deletion system is called *one-sided* if either $m + m' > 0$ and $m \cdot m' = 0$, or $q + q' > 0$ and $q \cdot q' = 0$, or both.

We immediately remark that, according to Corollary 3.3.2, an insertion-deletion system of a given size can be considered to be of any other larger size.

Consider a one-sided insertion rule $r : (u,\, \alpha,\, \lambda)_{ins}$, $u \neq \lambda$, and the insertion derivation step a $yuz \overset{r}{\Rightarrow} yu\alpha z$. We will call the substring u of yuz the *active context* in this step and will sometimes underline it for readability: $y\underline{u}z \overset{r}{\Rightarrow} yu\alpha z$. The active context for a deletion derivation step is defined symmetrically. We will often abuse terminology and say that u inserted (respectively, deleted) α, instead of saying that r inserted (respectively, deleted) α.

Example 3.1.1. *Consider the insertion-deletion system $\Gamma = (V, T, A, I, D)$ defined as follows:*

$$\begin{aligned} V &= \{a, b, S_a, S_b\},\\ T &= \{a, b\},\\ A &= \{S_aS_b\},\\ I &= \{(S_a,\, a,\, \lambda)_{ins}, (S_b,\, b,\, \lambda)_{ins}\},\\ D &= \{(\lambda,\, S_a,\, \lambda)_{del}, (\lambda,\, S_b,\, \lambda)_{del}\}. \end{aligned}$$

The size of Γ is $(1, 1, 0; 1, 0, 0)$ and it is a one-sided insertion-deletion system. The symbols S_a and S_b insert the terminals a and b respectively, until erased. Given that the only axiom of Γ is S_aS_b, this means that $L(\Gamma) = a^*b^*$. The following is an example of a terminal derivation of Γ:

$$S_aS_b \Rightarrow S_aaS_b \Rightarrow S_aaS_bb \Rightarrow aS_bb \Rightarrow ab.$$

A *leftist grammar* is a triple $G = (V, R, x)$, where V is a finite alphabet, $x \in V$ is the final symbol, and R is a set of rewriting rules of the following two types: insertion rules $a \to ba$ and deletion rules $cd \to d$, $a, b, c, d \in V$. A string $w \in V^*$ is said to derive $v \in V^*$ if either $w = yaz$, $v = ybaz$, and $r_i : a \to ba \in R$ (written as $w \overset{r_i}{\Rightarrow} v$), or $w = ycdz$, $v = ydz$, and $r_d : cd \to d \in R$ (written as $w \overset{r_d}{\Rightarrow} v$). Without losing generality, we can suppose that the set R does not contain rules erasing or inserting x, that is $\{a \to xa, xa \to a\} \not\subseteq R$ [64, Proposition 3].

Similarly to the case of insertion-deletion systems, we define $(\Rightarrow) = \bigcup_{r\in P}(\overset{r}{\Rightarrow})$, and use $(\Rightarrow^*)$ to refer to the reflexive and transitive closure of $(\Rightarrow)$. A derivation of a leftist grammar is a sequence of derivation steps $w_1x \Rightarrow w_2x \Rightarrow \ldots \Rightarrow w_nx$, where $w_i \in V^*$, $1 \le i \le n$. The language $L(G)$ recognised by the leftist grammar $G = (V, R, x)$ is

$$L(G) = \{w \in V^* \mid wx \Rightarrow^* x\}.$$

We will denote the family of languages recognised by leftist grammars by LFT.

Example 3.1.2. *Consider the leftist grammar $G = (V, R, x)$ defined as follows:*

$$V = \{a, b, x\}, \qquad R = \{ab \to b, bx \to x\}.$$

In this grammar, b can erase any number of a's to the left of it, and x can erase any number of b's. Therefore, the language recognised by this grammar is $L(G) = (a^*b)^*$.

Note that, even though the definition of a leftist grammar parallels the definition of a string rewriting grammar, the former is not a special case of the latter because of the way in which the recognised language is defined. Still, leftist grammars are closely related to string rewriting devices, and namely to one-sided insertion-deletion systems of sizes $(1, 1, 0; 1, 1, 0)$ and $(1, 0, 1; 1, 0, 1)$. The relationship between the two computing devices is formally captured in Section 3.2.

Similarly to insertion-deletion systems, we say that, in the derivation step $yaz \overset{r}{\Rightarrow} ybaz$, for $r : a \to ba$, the letter a is the active context; we will often underline it for readability: $y\underline{a}z \overset{r}{\Rightarrow} ybaz$. The active context for a leftist deletion rule is defined symmetrically.

Consider a one-sided insertion-deletion system Γ of size $(1, 1, 0; 1, 1, 0)$ and a derivation $C : w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n$ of it. C is called *leftmost* if the leftmost possible context is active in any derivation step. Formally, C is leftmost if, in any derivation step $w_i = u\underline{a}y \Rightarrow w_{i+1}$ with the active context a, no rule of Γ is applicable to u. Similarly, C is called *rightmost*, if the rightmost possible context is active in any derivation step, i.e. no rule of Γ is applicable to y.

A symbol is useful in C if it belongs either to w_1 or w_n, or if it inserts or deletes a useful symbol. The definition is well founded, because in insertion-deletion systems of size $(1, 1, 0; 1, 1, 0)$, symbols only insert or delete other symbols on one side only, so the relation "inserts-or-deletes" is acyclic [21]. A derivation is *pure* if all symbols appearing in it are useful.

The derivation C is *eager* if, informally, deletions occur as soon as possible. Formally, C is not eager if it contains a sentential form w_i, $1 \le i < n$, such that a deletion rule is applicable to w_i, but the rule applied in the derivation step $w_i \Rightarrow w_{i+1}$ is an insertion rule.

We define leftmost, rightmost, pure, and eager derivations for insertion-deletion systems of size $(1, 0, 1; 1, 0, 1)$ and for leftist grammars in the same manner.

A derivation of an insertion-deletion system of size $(1,1,0;1,1,0)$ is *greedy* if it is rightmost, pure, and eager. A derivation of an insertion-deletion system of size $(1,0,1;1,0,1)$ or a leftist grammar is greedy if it is leftmost, pure, and eager.

Example 3.1.3. *Consider the insertion-deletion system $\Gamma = (V, T, A, I, D)$ defined as follows:*

$$\begin{aligned} V &= \{S, A, a\},\\ T &= \{a\},\\ A &= \{aS\},\\ I &= \{(S,\ A,\ \lambda)_{ins}, (A,\ a,\ \lambda)_{ins}\},\\ D &= \{(S,\ A,\ \lambda)_{del}, (a,\ S,\ \lambda)_{del}, (a,\ A,\ \lambda)_{del}\}. \end{aligned}$$

The derivation $C_1 : a\underline{S} \Rightarrow aS\underline{A} \Rightarrow a\underline{S}Aa \Rightarrow \underline{a}Sa \Rightarrow aa$ is a rightmost derivation. The derivation $C_2 : \underline{a}S \Rightarrow a$ is a leftmost derivation, and the only leftmost derivation possible in Γ. The derivations C_1 and C_2 are also pure, while the derivation C_3 : $a\underline{S} \Rightarrow a\underline{S}A \Rightarrow \underline{a}S \Rightarrow a$ is not, because A is not useful. The derivation C_1 is also eager, and thus greedy, while the derivation $C_4 : a\underline{S} \Rightarrow aS\underline{A} \Rightarrow a\underline{S}Aa \Rightarrow aS\underline{A}Aa \Rightarrow aSA\underline{a}Aa \Rightarrow a\underline{S}Aaa \Rightarrow \underline{a}Aaa \Rightarrow aaa$ is pure, rightmost, but not eager, because the first instance of A to be inserted could have been deleted by S before the insertion of the second A.

3.2 Systems of Size $(1,1,0;1,1,0)$ and Leftist Grammars

In this section we discuss the relationship between leftist grammars and one-sided insertion-deletion systems of sizes $(1,0,1;1,0,1)$ and $(1,1,0;1,1,0)$. Even though the difference between the rules of a leftist grammar and those of an insertion-deletion system is purely syntactical – the rules $a \to ba$ and $cd \to d$ have exactly the same semantics as the rules $(\lambda, b, a)_{ins}$ and $(\lambda, c, d)_{del}$ – the languages associated with these devices are defined differently, which induces a number of important divergences. Firstly, the direction of derivation is flipped: while leftist grammars start with a word and erase it completely, insertion-deletion systems start with an axiom and generate a word. Secondly, the last sentential form of any valid derivation of a leftist grammar should be x, while an insertion-deletion system can start its derivation with any element of a finite set of axioms. Thus, leftist grammars work in recognising mode, while insertion-deletion systems operate in generating mode. This means, in particular, that taking the words satisfying a certain property as initial words for a leftist grammar (as done in [62], for example) corresponds in insertion-deletion systems to intersection of the generated language with a different language with given properties.

The following statement shows formally how a leftist grammar can be transformed into an equivalent insertion-deletion system generating the language recognised by the leftist grammar.

Proposition 3.2.1. *Consider the leftist grammar $G = (V, R, x)$ and construct the insertion-deletion system $\Gamma = (V, T, A, I, D \cup D_x)$ of size $(1,0,1;1,0,1)$ with $T = V \setminus \{x\}$, $A = \{x\}$, and the set of rules defined in the following way:*

$$\begin{aligned} I &= \{(\lambda,\ c,\ d)_{ins} \mid cd \to d \in R\},\\ D &= \{(\lambda,\ b,\ a)_{del} \mid a \to ba \in R\},\\ D_x &= \{(\lambda,\ x,\ \lambda)_{del}\}. \end{aligned}$$

We claim that $L(G) = L(\Gamma)$.

Proof. Consider the following derivation of G:

$$C : wx \Rightarrow_G^* w_i x \overset{r}{\Rightarrow}_G w_j x \Rightarrow_G^* x,$$

where $w, w_i, w_j \in V^*$. Then, by construction, $w_j x \overset{r'}{\Rightarrow}_\Gamma w_i x$, where $r' = (\lambda,\, c,\, d)_{ins}$ if $r = cd \to d$, and $r' = (\lambda,\, b,\, a)_{del}$ if $r = a \to ba$. By applying this observation to every step of C, we conclude that $x \Rightarrow_\Gamma^* wx$, and since $wx \Rightarrow_\Gamma w$ by the context-free rule $(\lambda,\, x,\, \lambda)_{del}$, we conclude that $x \Rightarrow_\Gamma^* w$. Therefore $L(G) \subseteq L(\Gamma)$.

To prove the converse inclusion, remark that all derivations of Γ are of the following form:

$$x \Rightarrow_\Gamma^* w_i x \Rightarrow_\Gamma w_i \Rightarrow_\Gamma^* w,$$

with $w_i, w \in V^*$. The subderivation $w_i \Rightarrow_\Gamma^* w$ of Γ can be stepwise translated into the derivation $wx \Rightarrow_G^* w_i x$ of G in which no rules employing x are applied. On the other hand, $x \Rightarrow_\Gamma^* w_i x$ can directly be translated into $w_i x \Rightarrow_G^* x$. Therefore $wx \Rightarrow_G^* w_i x \Rightarrow_G^* x$ and $w \in L(G)$. This means that $L(\Gamma) \subseteq L(G)$ which concludes the proof. □

It is turns out, quite unsurprisingly, that the class of languages generated by insertion-deletion systems of size $(1,0,1;1,0,1)$ is larger than that recognised by leftist grammars.

Proposition 3.2.2. $INS_1^{0,1}DEL_1^{0,1} \setminus LFT \neq \varnothing$

Proof. Consider the insertion-deletion system $\Gamma = (V, V, A, I, \varnothing)$ with rules of size $(1,1,0;1,1,0)$ defined as follows:

$$V = \{a, b\}, \qquad A = \{a, b\}, \qquad I = \{(\lambda,\, a,\, a)_{ins}, (\lambda,\, b,\, b)_{ins}\}.$$

The language generated by this system is $L(\Gamma) = a^* \cup b^*$, and $a^* \cup b^* \notin LFT$. Indeed, suppose that there exists a leftist grammar G such that $L(G) = a^* \cup b^*$, and consider two derivations $C_1 : a^n x \Rightarrow_G^* x$ and $C_2 : b^m x \Rightarrow_G^* x$, for some $n, m \in \mathbb{N}$. But then we can construct the derivation $C_{12} : a^n b^m x \Rightarrow_G^* x$ by first applying the rule applications of C_2 and then those of C_1. This implies that $a^n b^m \in L(G)$, which is contradiction. □

The following statement brings together the conclusions of the previous two propositions.

Corollary 3.2.3. $LFT \subsetneq INS_1^{0,1}DEL_1^{0,1} = \{rev(L) \mid L \in INS_1^{1,0}DEL_1^{1,0}\}$.

Even though there exist insertion-deletion systems of size $(1,0,1;1,0,1)$ which generate languages that no leftist grammar can recognise, for some of the insertion-deletion systems of this size equivalent leftist grammars can be constructed in a rather straightforward manner. The following statement describes precisely which insertion-deletion systems of size $(1,0,1;1,0,1)$ have this property.

Proposition 3.2.4. *Consider the insertion-deletion system* $\Gamma = (V, T, A, I, D)$ *of size* $(1,0,1;1,0,1)$ *such that* $T = V \setminus \{x\}$, $A = \{x\}$, $D = D' \cup \{(\lambda,\, x,\, \lambda)_{del}\}$, *and $I \cup D'$ contains no context-free rules, nor rules inserting or deleting x. Let $G = (V, R, x)$ be the leftist grammar with the following set of rules*

$$R = \{cd \to d \mid (\lambda, c, d)_{ins} \in I\} \cup \{a \to ba \mid (\lambda, b, a)_{del} \in D'\}.$$

Then $L(\Gamma) = L(G)$.

Proof. Since Γ has no context-free insertion rules, no symbols can be inserted to the right of x, which means that any derivation of Γ has the form

$$C : x \Rightarrow_\Gamma^* w_i x \Rightarrow_\Gamma w_i \Rightarrow_\Gamma^* w,$$

where $w_i, w \in V^*$. But then, by the construction of R, it is possible to use the same approach as in the proof of Proposition 3.2.1 and stepwise translate C into $wx \Rightarrow_G x$, therefore $L(\Gamma) \subseteq L(G)$. The converse inclusion follows from a direct stepwise translation of any derivation $wx \Rightarrow_G x$ into $x \Rightarrow_\Gamma^* wx$ and from the fact that $wx \Rightarrow_\Gamma w$. $\square$

For historical reasons, we prefer using rules of size $(1, 1, 0; 1, 1, 0)$ instead of $(1, 0, 1; 1, 0, 1)$ in the rest of this work. The languages generated by insertion-deletion systems of the former size are the reverse of those generated by insertion-deletion systems of the latter size. We will call the family of insertion-deletion systems with the properties described in the above proposition, but with the contexts on the left, *leftist insertion-deletion system.*

We can now aggregate the statements of Propositions 3.2.1 and 3.2.4 into a single corollary which formally shows which families of insertion-deletion systems exactly correspond to leftist grammars.

Corollary 3.2.5. *For every leftist grammar G there exists a leftist insertion-deletion system Γ such that $L(G) = rev(L(\Gamma))$, and conversely, for every leftist insertion-deletion system Γ there exists a leftist grammar G such that $L(\Gamma) = rev(L(G))$.*

Remark that leftist insertion-deletion systems are only one of the subfamilies of insertion-deletion systems of size $(1, 1, 0; 1, 1, 0)$ which are equivalent in power to leftist grammars (modulo a reverse operation). For example, lifting the restriction that all rules must use a non-empty context yields an equivalent computing device. Indeed, according to [87, Lemma 2] (recalled in this thesis as Lemma 3.3.1), the effect of context-free rules can be reproduced by replacing them with rules using any symbol of the alphabet as a context, and by adding a marker symbol to the left end of the string. In the case of leftist insertion-deletion systems, x readily serves as such a marker, which means that context-free operations can be directly simulated.

We will now recall an important statement about greedy derivations, which are a very useful instrument for the analysis of behaviour of leftist grammars: as is shown in [62, Theorem 1] and then generalised in [21, Proposition 3.1], any derivation $C : w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n$ of a leftist grammar has an equivalent greedy derivation $w_1 \Rightarrow^* w_n$. The generalised proposition does not require that $w_n = x$ or that any sentential form of the derivation contain an x at all, which means that the statement can be directly translated to insertion-deletion systems of size $(1, 1, 0; 1, 1, 0)$ by simply flipping the positions of the contexts.

Proposition 3.2.6 ([21, Proposition 3.1])**.** *Given an insertion-deletion system $\Gamma = (V, T, A, I, D)$ of size $(1, 1, 0; 1, 1, 0)$, and any derivation $C : w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n$*

(not necessarily terminal), there exists an equivalent derivation $C' : w_1 \Rightarrow w_2' \Rightarrow \ldots \Rightarrow w_n$ *which is greedy (rightmost, pure, and eager).*

3.3 Systems of Sizes $(1, m, 0; 1, q, 0)$

In this section we remind that insertion-deletion systems of size $(1, 1, 0; 1, 1, 0)$ (and thus leftist grammars) are not computationally complete and cannot generate even some simple regular languages. We will show, on the other hand, that allowing slightly larger contexts either for insertions or deletions and considering rules of size $(1, 1, 0; 1, 2, 0)$ or $(1, 2, 0; 1, 1, 0)$ is enough to generate all regular languages, and therefore renders such systems more difficult to reason about. A summary of the results presented in this section is given in Table 3.1 on page 48. In this table, L_{2^k} stands for $\{(F_1F_0)^n(a_1a_0)^m \mid n \geq 2^{2m-2}\}$, where $m, n \in \mathbb{N}$ and $k \geq 2$.

We start by reminding that any insertion-deletion system can be brought into a normal form in which all rules, but two, correspond exactly to the total size of the system.

Lemma 3.3.1 ([73, Lemma 4.2.1])**.** *For any insertion-deletion system* $\Gamma = (V, T, A, I, D)$ *of size* $(n, m, m'; p, q, q')$ *it is possible to construct the padded insertion-deletion system* $\Gamma_p = (V \cup \{X, Y\}, T, A_p, I_p, D_p \cup D_p')$ *of the same size with the following properties:*

- $L(\Gamma) = L(\Gamma_p)$,
- $D_p' = \{(\lambda,\ X,\ \lambda)_{del}, (\lambda,\ Y,\ \lambda)_{del}\}$,
- *for any insertion rule* $(u,\ \alpha,\ v)_{ins} \in I_p$, $|\alpha| = n$, $|u| = m$, *and* $|v| = m'$, *and*
- *for any deletion rule* $(u,\ \alpha,\ v)_{del} \in D_p$, $|\alpha| = p$, $|u| = q$, *and* $|v| = q'$.

Proof sketch. The idea of the proof is based on the fact that shorter contexts, inserted or deleted strings, and, correspondingly, the axioms, can be padded by instances of X and Y, which are eventually erased by the rules from D_p'.

We will first define the function padding a string w to a given length k with the symbol U:

$$pad_U(w, k) = w \sqcup\!\sqcup \{U\}^{k \dot{-} |w|},$$

where $k \dot{-} |w| = \max(k - |w|, 0)$. The insertion rules of Γ_p can now be given as $I_p = I_p' \cup I_p''$, with

$$\begin{aligned} I_p' &= \{\, (z_1,\ xY^k,\ z_2)_{ins} \mid (a,\ x,\ b)_{ins} \in I,\ z_1 \in pad_X(a, m), \\ &\qquad z_2 \in pad_Y(b, m'),\ k = n \dot{-} |x| \,\}, \\ I_p'' &= \{\, (z_1,\ \ Y^n,\ z_2)_{ins} \mid z_1 \in (V \cup \{X, Y\})^m,\ z_2 \in (V \cup \{X, Y\})^{m'} \}. \end{aligned}$$

The set I_p' essentially contains the padded versions of insertion rules from Γ, while the rules from I_p'' add padding to the string itself whenever necessary. The deletion rules are defined similarly:

$$\begin{aligned} D_p = \{\, (z_1,\ d,\ z_2)_{del} \mid\ & (a,\ x,\ b)_{del} \in D,\ z_1 \in pad_X(a, q), \\ & z_2 \in pad_Y(b, q'),\ d \in pad_Y(x, p) \}. \end{aligned}$$

Finally, we pad the axioms of Γ:

$$A_p = \{X^i w Y^t Y^j \mid w \in A,\ i = \max(m, q),\ j = \max(m', q'),\ t = \max(n, p) \dot{-} |w|\}.$$

All the rules of Γ_p, except those in D'_p, are exactly of the size $(n, m, m'; p, q, q')$ and are defined in such a way that the padding symbols X and Y are "transparent": any derivation of Γ_p can be transformed into a derivation of Γ by "discarding" the padding. On the other hand, since the counterparts of all rules of Γ exist in Γ_p as well, any derivation of Γ can be simulated by Γ_p. □

The practical implication of the previous statement to the case of insertion-deletion systems of sizes $(1, 1, 0; 1, 1, 0)$, $(1, 1, 0; 1, 2, 0)$, and $(1, 2, 0; 1, 1, 0)$ is that we can consider without losing generality that all rules are of the same size. Furthermore, the same technique can be used to pad any insertion-deletion system to any given larger size.

Corollary 3.3.2. *$INS_n^{m,m'} DEL_p^{q,q'} \subseteq INS_y^{x,x'} DEL_t^{z,z'}$, for $m \leq x$, $m' \leq x'$, $n \leq y$, and $q \leq z$, $q' \leq z'$, $p \leq t$.*

Next, we quote the result stating that one can safely assume that terminal symbols are never deleted in a derivation of an insertion-deletion system.

Lemma 3.3.3 ([73, Lemma 4.2.2]). *For any insertion-deletion system $\Gamma = (V, T, A, I, D)$ it is possible to construct an equivalent insertion-deletion system $\Gamma' = (V', T, A', I', D')$ of the same size such that $L(\Gamma) = L(\Gamma')$ and Γ' does not contain any rules deleting terminal symbols.*

Proof. Let $V' = V \cup \{N_x \mid x \in T\}$ and consider the mappings $f : V \to V'$ given by $f(x) = N_x$ if $x \in T$ and $f(x) = x$ if $x \in V \setminus T$, as well as $id : V \to V'$, $id(x) = x$, for any $x \in V$. We now define the function $F : V \to 2^{(V')^*}$ as

$$F(x_1 \dots x_n) = \{g(x_1) \dots g(x_n) \mid g \in \{f, id\}\}.$$

$F(w)$ is the set of words obtained from w by replacing some (or none) of the terminals in w by the corresponding non-terminals of the form N_x. We apply F to the axioms and insertion rules of Γ to obtain $A' = \{w' \mid w \in A, w' \in F(w)\}$ and

$$I' = \{(a',\, x',\, b')_{ins} \mid (a,\, x,\, b)_{ins} \in I,\ a' \in F(a),\ b' \in F(b),\ x' \in F(x)\}.$$

We cannot construct D' in exactly the same way, because we need to avoid the rules erasing terminals. Therefore, instead of applying F to deleted strings, we will use the morphism induced by the mapping f:

$$D' = \{(a',\, x',\, b')_{del} \mid (a,\, x,\, b)_{del} \in I,\ a' \in F(a),\ b' \in F(b),\ x' = f(x)\}.$$

It follows from the way in which the rules of Γ' are defined that any derivation of Γ' can be transformed into a derivation of Γ by substituting the symbols N_x in every sentential form for the corresponding terminal x. On the other hand, any derivation of Γ can be transformed into a derivation of Γ' by replacing insertions of terminals that are eventually deleted by insertions of the corresponding symbols of the N_x family. We conclude therefore that $L(\Gamma') = L(\Gamma)$. □

We now remind that insertion-deletion systems of size $(1,1,0;1,1,0)$ cannot generate the regular language $(ba)^+$.

Theorem 3.3.4 ([76, Theorem 8])**.** $(ba)^+ \in REG \setminus INS_1^{1,0}DEL_1^{1,0}$.

Proof. Suppose that there exists an insertion-deletion system Γ of size $(1,1,0;1,1,0)$ such that $L(\Gamma) = (ba)^+$. Since Γ has a finite number of axioms, there must exist a derivation of Γ in which a letter a is inserted:

$$w \Rightarrow^* w_1 z w_2 \Rightarrow w_1 z a w_2 \Rightarrow^* \alpha ba \beta \in (ba)^+,$$

where $w \in A$, $w_1, w_2, \alpha, \beta \in V^*$, and $z \in V$. It follows from the size of Γ that $w_1 z \Rightarrow^* \alpha b$ and that $w_2 \Rightarrow^* \beta$. But then we can insert the highlighted instance of a twice and get the following terminal derivation of Γ:

$$w \Rightarrow^* w_1 z w_2 \Rightarrow w_1 z a w_2 \Rightarrow w_1 z a a w_2 \Rightarrow^* \alpha baa \beta \notin (ba)^+,$$

which is a contradiction and concludes the proof. □

Although insertion-deletion systems of size $(1,1,0;1,1,0)$ cannot generate some simple regular languages, even pure insertion systems with insertion rules of size $(1,1,0)$ are capable of producing non-regular context-free languages.

Example 3.3.5 ([117, Example 2.4.1])**.** *Consider the insertion-deletion system* $\Gamma = (T, T, \{a\}, I, \varnothing)$*, with* $T = \{a, b, c, d\}$ *and*

$$I = \{(a,\ b,\ \lambda)_{ins}, (b,\ c,\ \lambda)_{ins}, (c,\ d,\ \lambda)_{ins}, (d,\ a,\ \lambda)_{ins}\}.$$

One of the possible derivations of Γ *looks as follows (we underline the active contexts):*

$$C : \underline{a} \Rightarrow \underline{a}b \Rightarrow a\underline{b}b \Rightarrow a\underline{b}cb \Rightarrow ab\underline{c}cb \Rightarrow ab\underline{c}dcb \Rightarrow abc\underline{d}dcb \Rightarrow abcdadcb.$$

Remark that similar rule applications will have to be carried out in any derivation of a word ending in dcb. Γ *can also generate words ending in* $(dcb)^*$ *by carrying out the rule applications from* C *near the rightmost instance of a, but the insertion of each subword dcb of the suffix will require the insertion of at least an instance of each of the symbols a, b, c, and d, in this order, in the prefix of the string. On the other hand, multiple repetitions of the word abcd can be generated in the prefix without adding any words dcb to the suffix. In other words,* $L(\Gamma) \cap (abcd)^*(dcb)^* = \{(abcd)^m(dcb)^n \mid m \geq n\}$*. Given that this language is context-free and that context-free languages are closed under intersection with regular languages, this means that* $L(\Gamma)$ *is a context-free language as well.*

The argument in the proof of Theorem 3.3.4 points out that terminal symbols hinder any transmission of information along the string, and so extra insertions cannot be detected. The following theorem shows that allowing insertion contexts of length 2 suffices to enable at least some information to get across.

Theorem 3.3.6. $REG \subseteq INS_1^{2,0}DEL_1^{1,0}$.

Proof. Consider an arbitrary finite automaton $FA = (Q, T, q_0, F, \delta)$. We define the insertion-deletion system $\Gamma = (V, T, A, I, D)$ in the following way:

$$\begin{aligned} V &= \{Q_i \mid q_i \in Q\} \cup \{B, E\} \cup T, \\ A &= \{Q_0 BE\}, \\ I &= \{(B,\, a,\, \lambda)_{ins} \mid a \in T\} \\ &\cup \{(Q_i a,\, Q_j,\, \lambda)_{ins} \mid q_j \in \delta(q_i, a)\}, \\ D &= \{(Q_f,\, E,\, \lambda)_{del} \mid q_f \in F\} \cup \{(\lambda,\, B,\, \lambda)_{del}\} \\ &\cup \{(\lambda,\, Q_i,\, \lambda)_{del} \mid q_i \in Q\}. \end{aligned}$$

The system works in two phases. In the first phase, B inserts a string of terminals, resulting in a word of the form $Q_0 BT^* E$. The second phase starts with a deletion of B, which results in a string of the form $Q_0 T^* E$ and effectively disables all rules inserting terminals. The only rules applicable from now on are those inserting and erasing the state symbols Q_i. If Q_i is to the left of such a terminal a that $\delta(q_i, a) \neq \varnothing$, then a new state symbol Q_j, $q_j \in \delta(q_i, a)$, can be inserted. A chain of these insertions starts with Q_0 and continues while possible or until the end of the string is reached. If the state symbol inserted after the last terminal is a Q_f, $q_f \in F$, then the end marker E can be erased. This eventually results in a terminal string, after all state symbols are deleted by context-free rules. The deletion of E is therefore only possible if the terminals inserted by B in the first phase form a word accepted by FA, which means that $L(\Gamma) = L(FA)$. $\square$

It turns out that insertion-deletion systems of symmetric size, $(1, 2, 0; 1, 1, 0)$, are also capable of simulating any finite automaton. We will prove this fact by using a similar approach: we will use deletions with two-symbol contexts to verify that a correct trajectory of a finite automaton was generated.

Theorem 3.3.7. *$REG \subseteq INS_1^{1,0} DEL_1^{2,0}$.*

Proof. Consider an arbitrary finite automaton $FA = (Q, T, q_0, F, \delta)$. We define the insertion-deletion system $\Gamma = (V, T, A, I, D)$ in the following way:

$$\begin{aligned} V &= \{Q_i \mid q_i \in Q\} \cup \{A, B, E\} \cup T, \\ A &= \{ABE\}, \\ I &= \{(B,\, Q_i,\, \lambda)_{ins} \mid q_i \in Q\} \\ &\cup \{(B,\, a,\, \lambda)_{ins} \mid a \in T\}, \\ D &= \{(Q_f,\, E,\, \lambda)_{del} \mid q_f \in F\} \\ &\cup \{(Q_i a,\, Q_j,\, \lambda)_{del} \mid q_j \in \delta(q_i, a)\} \\ &\cup \{(A,\, B,\, \lambda)_{del}, (A,\, Q_0,\, \lambda)_{del}, (\lambda,\, A,\, \lambda)_{del}\}. \end{aligned}$$

We claim that the only way for Γ to generate a terminal string is to correctly simulate a trajectory of FA. Indeed, in order to erase E, B has to insert at least one Q_f, for $q_f \in F$. Then, for any Q_j, $j > 0$, to be erased, there must be such a subword $Q_i a$ to its left that $q_j \in \delta(q_i, a)$. Note that, while several instances of a state symbol Q_j can be inserted one next to the other, the contexts of deletion rules assure that there is exactly one instance of a terminal symbol between two state symbols. Since Q_0 is the only state symbol whose deletion does not depend on other state symbols, B is guaranteed to insert a sequence of the form $Q_0^* a_{i_0} Q_{i_1}^* a_{i_1} \ldots Q_{i_n}^* a_{i_n} Q_f^*$ which corresponds to a trajectory of FA accepting the word $a_{i_0} a_{i_1} \ldots a_{i_n}$, because otherwise A will not be able to erase the leftmost state symbol Q_j.

We finish the proof by pointing out that, if A or B is deleted before a complete trajectory of the automaton is generated, some of the state symbols will never be erased. Therefore, deleting A or B too early yields derivations which do not produce terminal strings. $\square$

Theorem 6 of [64] shows such an insertion-deletion system Γ of size $(1,1,0;1,1,0)$ that $L(\Gamma) \cap (F_1F_0)^+(a_1a_0)^+ = L_{2^k} = \{(F_1F_0)^n(a_1a_0)^m \mid n \geq 2^{2m-2}\}$, which trivially implies that systems in which contexts of length 2 are allowed can generate non-context-free languages as well. It turns out, however, that such systems can generate the language L_{2^k} directly, without any additional control. We start by showing how the normal form for insertion-deletion systems induced by Lemmas 3.3.1 and 3.3.3 can be further refined for rules of size $(1,1,0;1,1,0)$ by requiring that no terminal inserts or deletes any symbols.

Lemma 3.3.8. *For an arbitrary insertion-deletion system $\Gamma = (V, T, A, I, D)$ of size $(1,1,0;1,1,0)$ there exists an equivalent system $\Gamma' = (V', T, A', I', D')$ of the same size such that $L(\Gamma') = L(\Gamma)$ and no insertion rule of Γ is of the form $(a,\, x,\, \lambda)_{ins}$, with $a \in T$, $x \in V$.*

Proof. We define the alphabet of Γ' to contain additional non-terminal symbols per each terminal: $V' = V \cup \{N_a, N'_a\}$. The new set of axioms is defined as $A' = \{h(w_0) \mid w_0 \in A\}$, where $h : V' \to V'$ is a morphism given by the following:

$$h(x) = \begin{cases} xN'_x, & \text{if } x \in T, \\ x, & \text{otherwise.} \end{cases}$$

We will use the notation I_T to refer to those insertion rules of Γ which have a terminal symbol in their context: $I_T = \{(a,\, x,\, \lambda)_{ins} \in I \mid a \in T, x \in V\}$. The insertion and deletion rules of Γ' are

$$\begin{aligned} I' &= \{(N_a,\, x,\, \lambda)_{ins}, (N'_a,\, x,\, \lambda)_{ins} \mid (a,\, x,\, \lambda)_{ins} \in I\} \\ &\cup \{(x,\, N_a,\, \lambda)_{ins} \mid (x,\, a,\, \lambda)_{ins} \in I\} \\ &\cup I \setminus I_T, \\ D' &= \{(N_a,\, x,\, \lambda)_{del}, (N'_a,\, x,\, \lambda)_{del} \mid (a,\, x,\, \lambda)_{del} \in D\} \\ &\cup \{(a,\, N_a,\, \lambda)_{del}, (a,\, N'_a,\, \lambda)_{del} \mid a \in T\} \\ &\cup D. \end{aligned}$$

Remember that it is with no loss of generality that we can consider that Γ does not ever delete terminal symbols [73, Lemma 4.2.2].

Γ' directly simulates a derivation of Γ by applying the rule $(x,\, N_a,\, \lambda)_{ins}$ right before any application of the rule $(x,\, a,\, \lambda)_{ins}$, and by replacing all insertions and deletions happening in the context of a by insertions and deletions happening in the context of the corresponding N_a or N'_a. At the very end, the rules $(a,\, N_a,\, \lambda)_{del}$ and $(a,\, N'_a,\, \lambda)_{del}$ are applied to finalise the clean-up of the string.

To see how Γ can simulate any derivation of Γ', remark that, to be erased, a symbol N_a requires the presence of a to its left, and consider the derivation

$$C : wv \Rightarrow wN_av \Rightarrow^* w'aN_av' \Rightarrow w'av',$$

where $w, v, w', v' \in V^*$. Because all contexts are of length at most 1 and are all to the left, it holds that $N_av \Rightarrow^* N_av'$, and $w \Rightarrow^* w'a$. But then it is possible to reorder C in the following way:

$$wv \Rightarrow wN_av \Rightarrow^* w'aN_av \Rightarrow^* w'aN_av' \Rightarrow w'av'.$$

Therefore, it suffices to consider those derivations of Γ' in which all symbols N_a carry out operations only when there is a symbol a to the left of them. Remember also that, by the construction of Γ', all symbols N'_a, are guaranteed to always be located immediately to the right of an instance of a, too. This means that, to simulate any derivation of Γ', Γ would just need to directly repeat applications of the rules which do not involve symbols N_a and N'_a, and perform the operations happening in the context of N_a or N'_a in the context of the corresponding instances of a.

We have therefore established that any terminal derivation of Γ can be simulated by Γ' and conversely, which implies that $L(\Gamma) = L(\Gamma')$ and proves the statement of the lemma. □

The previous lemma together with Lemma 3.3.3 effectively state that, in the case of insertion-deletion systems of size $(1, 1, 0; 1, 1, 0)$, any contiguous region of terminals is guaranteed to never change. This allows us to formulate the following lower bound for the power of systems of size $(1, 1, 0; 1, 2, 0)$.

Theorem 3.3.9. *Consider an insertion-deletion system Γ_1 of size $(1, 1, 0; 1, 1, 0)$ and a regular language L. Then there exists an insertion-deletion system Γ_2 of size $(1, 1, 0; 1, 2, 0)$ such that $L(\Gamma_2) = L(\Gamma_1) \cap L$.*

Proof. Consider an insertion-deletion system $\Gamma_1 = (V, T, A, I, D)$ with rules of size $(1, 1, 0; 1, 1, 0)$ and the finite automaton $FA = (Q, T, q_0, F, \delta)$ recognising the language L. Without losing generality, we may suppose that Γ_1 has no context-free rules (Lemma 3.3.1), never deletes terminals (Lemma 3.3.3), and never inserts anything in the context of a terminal (Lemma 3.3.8). We will construct the insertion-deletion system $\Gamma_2 = (V_2, T, A_2, I_2, D_2)$ in the following way:

$$\begin{aligned}
V_2 &= \{Q_i \mid q_i \in Q\} \cup \{B, E\} \cup V, \\
A_2 &= \{BQ_0w_0E \mid w_0 \in A\}, \\
I_2 &= \{(a,\, Q_i,\, \lambda)_{ins} \mid a \in T, q_i \in Q\} \cup I, \\
D_2 &= \{(Q_ia,\, Q_j,\, \lambda)_{del} \mid q_j \in \delta(q_i, a)\} \\
&\quad \cup \{(Q_f,\, E,\, \lambda)_{del} \mid q_f \in F\} \\
&\quad \cup \{(B,\, Q_0,\, \lambda)_{del}, (\lambda,\, B,\, \lambda)_{del}\} \cup D.
\end{aligned}$$

A terminal derivation of Γ_2 consists of two phases. In the first phase the rules from $I \cup D$ are applied, possibly resulting in a word of the form BQ_0wE, where $w \in T^*$. In the second phase, every terminal inserts a state symbol, and the deletion rules of the form $(Q_ia,\, Q_j,\, \lambda)_{del}$ check that a trajectory of FA is correctly simulated. The rules of the form $(Q_f,\, E,\, \lambda)_{del}$ and $(B,\, Q_0,\, \lambda)_{del}$ assure that complete accepting trajectories of FA are generated (cf. the proof of Theorem 3.3.7).

To see that Γ_2 cannot essentially deviate from this evolution scheme, consider the word wQ_ju, in which $w \in V_2^*$ and $u \in T^*$. According to our initial assumptions about Γ_1, the only symbols that may be inserted in u are state symbols Q_i. However, since these symbols do not insert anything, they will only eventually get erased without influencing the form of the terminal word u. Remark now that the only way to erase E is to insert Q_f to the left of it. We can now inductively apply our observation about the words of the form wQ_ju to conclude that, whenever Γ_2 succeeds in erasing all non-terminal symbols, the resulting terminal word belongs to the language recognised by the finite automaton FA. □

The following statement can now be directly deduced from [64, Theorem 6].

Corollary 3.3.10. $L_{2^k} = \{(F_1F_0)^n(a_1a_0)^m \mid n \geq 2^{2m-2}\} \in INS_1^{1,0}DEL_1^{2,0}$.

The proofs above give the intuitive impression that every contraption devised for insertion-deletion systems of size $(1, 1, 0; 1, 2, 0)$ can be translated in a rather straightforward manner to systems of size $(1, 2, 0; 1, 1, 0)$. In fact, this statement is true even in a much more general one-sided case.

Lemma 3.3.11. $INS_1^{k,0}DEL_1^{k,0} \subseteq INS_1^{k,0}DEL_1^{1,0}$, $k \geq 1$.

Proof. Consider an insertion-deletion system $\Gamma = (V, T, A, I, D)$ with rules of size $(1, k, 0; 1, k, 0)$. We construct the insertion-deletion system $\Gamma = (V', T, A, I', D')$ of size $(1, k, 0; 1, 1, 0)$ in the following way:

$$\begin{aligned}
V' &= \{X_r \mid r \in D\} \cup V,\\
I' &= \{(u,\ X_r,\ \lambda)_{ins} \mid r : (u,\ x,\ \lambda)_{del} \in D\} \cup I,\\
D' &= \{(X_r,\ x,\ \lambda)_{del}, (t,\ X_r,\ \lambda)_{del} \mid r : (u,\ x,\ \lambda)_{del} \in D, u = u't\}.
\end{aligned}$$

We immediately see that $L(\Gamma) \subseteq L(\Gamma')$, because any derivation step $w_i \overset{r}{\Rightarrow}_\Gamma w_{i+1}$ for $r \in I$ can be directly reproduced in Γ', while a step with $r = (u,\ x,\ \lambda)_{del}$ can be simulated in three steps of Γ' (the active contexts are underlined):

$$w_i'\underline{u}xw_i'' \Rightarrow_{\Gamma'} w_i'u\underline{X_r}xw_i'' \Rightarrow_{\Gamma'} w_i'\underline{u}X_rw_i'' \Rightarrow_{\Gamma'} w_i'uw_i'',$$

To see that $L(\Gamma') \subseteq L(\Gamma)$, consider the derivation

$$w_1\underline{u}v_1 \Rightarrow w_1uX_rv_1 \Rightarrow^* w_2X_rxv_2,$$

where $w_1, v_1, w_2, v_2 \in V^*$. Remember that one may require, without losing generality, that all rules in Γ have contexts of length exactly k. This, together with the fact that X_r is not included in the context of any insertion rule, implies that x can appear to the right of X_r in $w_2X_rxv_2$ only if it was already there in $w_1uX_rv_1$; in other words, $v_1 = xv_1'$. But then, to simulate a derivation of Γ', Γ has to directly reproduce the applications of the rules not involving X_r, and, instead of carrying out deletions in the context of X_r, perform them in the context of u directly. This means that any terminal word produced by Γ' can be also produced by Γ, which concludes the proof. □

The following lemma captures a similar inclusion for rules of size $(1, 1, 0; 1, k, 0)$.

Lemma 3.3.12. $INS_1^{k,0}DEL_1^{k,0} \subseteq INS_1^{1,0}DEL_1^{k,0}$, $k \geq 1$.

Proof. Consider an insertion-deletion system $\Gamma = (V, T, A, I, D)$ with rules of size $(1, k, 0; 1, k, 0)$. We construct the system $\Gamma' = (V', T, A, I', D')$ of size $(1, 1, 0; 1, k, 0)$ in the following way:

$$\begin{aligned}
V' &= \{X_r \mid r \in I\} \cup V,\\
I' &= \{(t,\ X_r,\ \lambda)_{ins}, (X_r,\ x,\ \lambda)_{ins} \mid r : (u,\ x,\ \lambda)_{ins} \in I, u = u't\},\\
D' &= \{(u,\ X_r,\ \lambda)_{del} \mid r : (u,\ x,\ \lambda)_{ins} \in I\} \cup D.
\end{aligned}$$

It is immediately clear that $L(\Gamma) \subseteq L(\Gamma')$, because all the rules from D are included in D' and the application of a rule $r = (u,\ x,\ \lambda)_{ins} \in I$ can be simulated as follows:

$$w\underline{u}v \Rightarrow_{\Gamma'} wu\underline{X_r}v \Rightarrow_{\Gamma'} w\underline{u}X_rxv \Rightarrow_{\Gamma'} wuxv.$$

Consider now the following derivation of Γ':

$$C : w_1\underline{t}v_1 \Rightarrow_{\Gamma'} w_1tX_rv_1 \Rightarrow^*_{\Gamma'} w_2\underline{X_r}v_2 \Rightarrow_{\Gamma'} w_2X_rxv_2 \Rightarrow^*_{\Gamma'} w_3\underline{u}X_rv_3 \Rightarrow_{\Gamma} w_3uv_3,$$

where $w_i, v_i \in V^*$, $1 \leq i \leq 3$, and $w_2X_rv_2$ is the first sentential form in which X_r inserts an x. In this derivation the applications of the rules $(t,\ X_r,\ \lambda)_{ins}$, $(X_r,\ x,\ \lambda)_{ins}$, and $(u,\ X_r,\ \lambda)_{del}$ are interleaved with other operations. Yet, since X_r only appears in two other rules of Γ' and because all rules in Γ' only have left contexts, the following hold: $w_1t \Rightarrow^* w_2 \Rightarrow^* w_3u$, $v_1 \Rightarrow^* v_2$, and $X_rxv_2 \Rightarrow^* X_rv_3$. Therefore, C can be reordered as follows:

$$w_1tv_1 \Rightarrow^* w_3\underline{u}v_2 \Rightarrow w_3u\underline{X_r}v_2 \Rightarrow w_3uX_rxv_2 \Rightarrow^* w_3\underline{u}X_rv_3 \Rightarrow w_3uv_3.$$

The possibility of such a reordering of any derivation involving X_r and the fact that this symbol can only be erased in the context of a substring u leads us to the conclusion that any such derivation can be reproduced in Γ by performing the insertions of x directly in the context of the corresponding substring u, and by carrying over the applications of other rules. This indicates that $L(\Gamma') \subseteq L(\Gamma)$ and concludes the proof. □

Lemmas 3.3.11 and 3.3.12 immediately imply the following statement.

Theorem 3.3.13. $INS_1^{1,0}DEL_1^{k,0} = INS_1^{k,0}DEL_1^{1,0} = INS_1^{k,0}DEL_1^{k,0}$, $k \geq 1$.

Combined with Corollary 3.3.10 stating that insertion-deletion systems of size $(1, 1, 0; 1, 2, 0)$ can generate non-context-free languages, this theorem implies that the same is true for insertion-deletion systems of size $(1, 2, 0; 1, 1, 0)$.

Corollary 3.3.14. $L_{2^k} = \{(F_1F_0)^n(a_1a_0)^m \mid n \geq 2^{2m-2}\} \in INS_1^{2,0}DEL_1^{1,0}$.

Contrary to what one might expect, increasing the length of the left context in one-sided one-symbol insertion and deletion rules beyond 2 does not add expressive power. To prove this statement, we will first show that insertion-deletion systems of size $(1, k, 0; 1, k, 0)$ can simulate systems of size $(1, k, 0; 1, k + 1, 0)$, for $k \geq 2$, and will then inductively apply this observation.

Lemma 3.3.15. $INS_1^{k,0}DEL_1^{k+1,0} \subseteq INS_1^{k,0}DEL_1^{k,0}$, $k \geq 2$.

Proof. Consider the insertion-deletion system $\Gamma = (V, T, A, I, D)$ with rules of size $(1, k, 0; 1, k + 1, 0)$. One can require without losing generality that all deletion contexts in Γ are exactly of size $k + 1$ and all insertion contexts are of size k. We will construct the system $\Gamma' = (V', T, A, I', D')$ of size $(1, k, 0; 1, k, 0)$ in the following way:

$$\begin{aligned} V' &= \{X_r \mid r \in D\} \cup V, \\ I' &= \{(u,\ X_r,\ \lambda)_{ins} \mid r : (ux,\ t,\ \lambda)_{del} \in D\} \cup I, \\ D' &= \{(u,\ X_r,\ \lambda)_{del}, (X_rx,\ t,\ \lambda)_{del} \mid r : (ux,\ t,\ \lambda)_{del} \in D\}, \end{aligned}$$

where $u \in V^k$ and $x \in V$. $L(\Gamma) \subseteq L(\Gamma')$ because Γ' can directly reproduce any application of an insertion rule from I, while any application of a deletion rule $(ux,\, t,\, \lambda)_{del} \in D$ can be simulated as follows:

$$w\,\underline{u}xt\,v \Rightarrow w\,u\underline{X_r x}t\,v \Rightarrow w\,\underline{u}X_r xv \Rightarrow w\,ux\,v.$$

Remark now that, since X_r inserts no symbols, whenever it is inserted, the symbol x must already be present to the right of the insertion site. Moreover, the same is true for the erased instance of t, because we require that all insertion rules in Γ have contexts of length $k \geq 2$, and, on the other hand, X_r does not appear in the context of any insertion rule and can only participate in the deletion of t. Therefore any derivation of Γ' in which X_r is inserted and triggers the deletion of at least a t has the following form:

$$\begin{aligned} C : w_1\,\underline{u}xt\,v_1 &\Rightarrow w_1\,uX_r xt\,v_1 \Rightarrow^* w_2\,\underline{X_r x}t\,v_2 \\ &\Rightarrow w_2\,X_r x\,v_2 \Rightarrow^* w_3\,\underline{u}X_r x\,v_3 \Rightarrow w_3\,ux\,v_3. \end{aligned}$$

But then, because of the fact that the rules of Γ' are one-sided and that X_r only appears in the context of one rule, we know that $w_1 u \Rightarrow^* w_2 \Rightarrow^* w_3 u$, $X_r xt\,v_1 \Rightarrow^* X_r xt\,v_2$, and $X_r x\,v_2 \Rightarrow^* X_r x\,v_3$. With this in mind, one can reorder C in the following way:

$$\begin{aligned} w_1\,\underline{u}xt\,v_1 &\Rightarrow w_1\,uX_r xt\,v_1 \Rightarrow^* w_1\,u\underline{X_r x}t\,v_2 \Rightarrow w_1\,uX_r x\,v_2 \\ \Rightarrow^* w_1\,\underline{u}X_r x\,v_3 &\Rightarrow w_1\,ux\,v_3 \Rightarrow^* w_2\,x\,v_3 \Rightarrow^* w_3\,ux\,v_3. \end{aligned}$$

In this new derivation X_r is inserted and deleted within the same substring ux, which means that Γ can simulate a subderivation of Γ' employing X_r by directly applying the deletion rule $(ux,\, t,\, \lambda)_{del}$ in the corresponding context ux. Together with the fact that the applications of all insertion rules from I can carried over to Γ directly, this implies that Γ can generate any terminal word Γ' can produce and concludes the argument. □

Note that, due to Corollary 3.3.2, we know that $INS_1^{k,0}DEL_1^{k+1,0}$ contains $INS_1^{k,0}DEL_1^{k,0}$, so the previous lemma actually shows equality between the two classes of languages, rather than inclusion. Inductively applying that statement and using Theorem 3.3.13 yields the following result.

Theorem 3.3.16. $INS_1^{k,0}DEL_1^{k,0} = INS_1^{2,0}DEL_1^{2,0}$, $k \geq 2$.

Proof. According to Lemma 3.3.15, insertion-deletion systems of size $(1, k, 0; 1, k, 0)$ are equivalent in power to those of size $(1, k, 0; 1, k+1, 0)$. On the other hand, we know from Theorem 3.3.13 that $INS_1^{1,0}DEL_1^{k+1,0} = INS_1^{k+1,0}DEL_1^{k+1,0}$. But, since Corollary 3.3.2 implies the inclusions $INS_1^{1,0}DEL_1^{k+1,0} \subseteq INS_1^{k,0}DEL_1^{k+1,0} \subseteq INS_1^{k+1,0}DEL_1^{k+1,0}$, we conclude that $INS_1^{k,0}DEL_1^{k+1,0} = INS_1^{k+1,0}DEL_1^{k+1,0}$ and therefore that systems of size $(1, k, 0; 1, k, 0)$ generate the same languages as those with rules of size $(1, k+1, 0; 1, k+1, 0)$. It suffices now to inductively apply this observation and to remark that the construction from the proof of Lemma 3.3.15 works for the base case $k = 2$. □

The following is an immediate corollary of Theorems 3.3.13 and 3.3.16.

Table 3.1: A summary of new results on one-sided insertion-deletion systems

Result	Reference
$INS_1^{1,0}DEL_1^{1,0} \not\ni (ba^*)$	[76, Theorem 8]
$INS_1^{1,0}DEL_1^{1,0} \cap CF \setminus REG \neq \varnothing$	[117, Example 2.4.1]
$INS_1^{1,0}DEL_1^{2,0} \supseteq REG$	Theorem 3.3.7
$INS_1^{2,0}DEL_1^{1,0} \supseteq REG$	Theorem 3.3.6
$INS_1^{1,0}DEL_1^{2,0} \ni L_{2^k}$	Corollary 3.3.10
$INS_1^{1,0}DEL_1^{k,0} = INS_1^{k,0}DEL_1^{1,0},\ k \geq 1$	Theorem 3.3.13
$INS_1^{k,0}DEL_1^{k,0} = INS_1^{2,0}DEL_1^{2,0},\ k > 1$	Theorem 3.3.16
$INS_1^{1,0}DEL_1^{2,0} = INS_1^{m,0}DEL_1^{q,0}$, $m \cdot q \neq 0$, $m + q > 2$	Corollary 3.3.17
$INS_1^{2,0}DEL_1^{1,0} = INS_1^{m,0}DEL_1^{q,0}$, $m \cdot q \neq 0$, $m + q > 2$	Corollary 3.3.17

Corollary 3.3.17. $INS_1^{1,0}DEL_1^{2,0} = INS_1^{2,0}DEL_1^{1,0} = INS_1^{m,0}DEL_1^{q,0}$, *where* $m \cdot q \neq 0$, $m + q > 2$.

Table 3.1 summarises the results about one-sided insertion-deletion systems given in this section. We remind that L_{2^k} stands for $\{(F_1F_0)^n(a_1a_0)^m \mid n \geq 2^{2m-2}\}$, where $m, n \in \mathbb{N}$ and $k \geq 2$.

Even though insertion-deletion systems of size $(1, 1, 0; 1, 2, 0)$, and hence all systems of size $(1, m, 0; 1, q, 0)$, $m, q \in \mathbb{N}$, are sufficiently powerful to generate non-context-free languages, the fact that they can only check contexts on one side and that one symbol can be inserted or deleted at a time leads us to the supposition that such systems cannot generate all recursively enumerable languages.

Conjecture 3.3.18. $INS_1^{m,0}DEL_1^{q,0} \subsetneq RE$, *for all* $m, n \in \mathbb{N}$.

3.4 Derivation Graphs

3.4.1 Definition and Motivation

In spite of the apparent simplicity of insertion-deletion systems of size $(1, 1, 0; 1, 1, 0)$, Theorem 6 of [64] shows that rules of this size are capable of generating rather complex languages. In this section we introduce a graphical analysis tool aimed at capturing the complex interplay between the rules of such size and simplifying the design and understanding of such systems. Our construction is based on derivation trees introduced in [75, Definition 4].

Derivation trees are an alternative graphical representation of a derivation of an insertion-deletion system of size $(1, 1, 0; 1, 1, 0)$. Consider such a system $\Gamma = (V, T, A, I, D)$ and a derivation $C : w_0 \Rightarrow^* z$, $w_0 \in A$, $z \in T^*$. The derivation tree τ of C is an ordered tree iteratively constructed as follows:

- the root of τ is λ,
- the letters of w_0 are the children of λ, in order,
- for any insertion $wav \Rightarrow wabv$, add b as the new leftmost child of the letter a, and
- for any deletion $wabv \Rightarrow wav$, strike out the node corresponding to the letter b.

Example 3.4.1 ([75, Example 1]). *Let $\Gamma = (V = \{a, b, c\}, T = V, A = \{a\}, I, D)$ be an insertion-deletion system with the following rules:*

$$\begin{aligned} I &= \{(a,\, b,\, \lambda)_{ins}, (a,\, a,\, \lambda)_{ins}, (b,\, c,\, \lambda)_{ins}, (a,\, c,\, \lambda)_{ins}\},\\ D &= \{(c,\, b,\, \lambda)_{del}\}, \end{aligned}$$

and consider the following derivation of it (we underline the active contexts):

$$\underline{a} \Rightarrow \underline{a}a \Rightarrow \underline{a}ba \Rightarrow a\underline{a}ba \Rightarrow aac\underline{b}a \Rightarrow aac\underline{b}ca \Rightarrow aa\underline{c}bcca \Rightarrow aaccca.$$

The sequence of derivation trees corresponding to each sentential form of this derivation is shown in Figure 3.1.

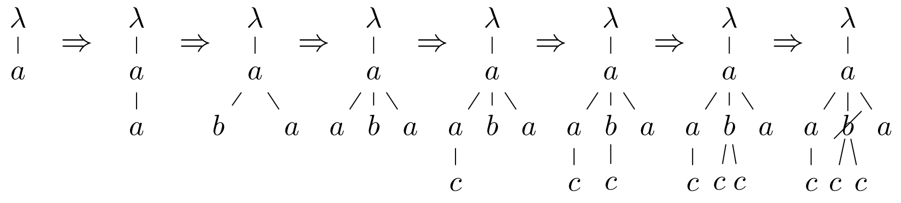


Figure 3.1: The sequence of derivation trees corresponding to the sentential forms of the derivation shown in Example 3.4.1.

By construction, the word corresponding to a derivation tree can be obtained by collecting the nodes in pre-order, and by ignoring the nodes which are struck through.

Derivation graphs essentially improve on derivation trees by storing more information about deletion and by adopting a more convenient style of graphical presentation. Before introducing this structure, however, we need to formally capture the notion of a letter, informally discussed in [21].

Definition 3.4.2. *Given a finite alphabet of symbols V and an infinite alphabet of markers M, a* letter over V *is a pair $(a, m) \in V \times M$. A* well-formed string of letters *over V is any string from $(V \times M)^*$ without repetitions. The* canonical projection *of any well-formed string of letters $\bar{w} = (a_1, m_1)(a_2, m_2) \ldots (a_n, m_n)$ is the string $\pi(\bar{w}) = a_1 a_2 \ldots a_n$.*

The role of markers is helping distinguish between different instances of the same symbol in one string. Thus, any string over the alphabet V can be transformed into a well-formed string of letters by picking distinct markers for every instance of a symbol. For example, for $M = \mathbb{N}$ and $V = \{a, b\}$, one well-formed string of letters corresponding to *abab* could be $(a, 1)(b, 3)(a, 3)(b, 7)$.

Definition 3.4.3. *A well-formed string $\bar{w}$ over the alphabet V is said to* derive *another well-formed string $\bar{v}$* by the insertion rule *$(u,\, \alpha,\, v)_{ins}$ if $\bar{w} = \bar{w}'\bar{u}\bar{v}\bar{w}''$ and $\bar{v} = \bar{w}'\bar{u}\bar{\alpha}\bar{v}\bar{w}''$, where $\pi(\bar{u}) = u$, $\pi(\bar{v}) = v$, and $\pi(\bar{\alpha}) = \alpha$. Symmetrically, $\bar{w}$ is said to* derive *$\bar{v}$* by the deletion rule *$(u,\, \alpha,\, v)_{del}$ if $\bar{w} = \bar{w}'\bar{u}\bar{\alpha}\bar{v}\bar{w}''$ and $\bar{v} = \bar{w}'\bar{u}\bar{v}\bar{w}''$, with $\pi(\bar{u}) = u$, $\pi(\bar{v}) = v$, and $\pi(\bar{\alpha}) = \alpha$.*

The derivation relations induced by each of the insertion and deletion rules over the set of well-formed strings are easily extendable to sets of rules in the usual way (cf. Section 3.1), yielding constructions which parallel exactly the relations $(\Rightarrow_{ins})$, $(\Rightarrow_{del})$, $(\Rightarrow)$, and $(\Rightarrow^*)$. Moreover, remark that for a fixed insertion-deletion system and any two well-formed strings of letters $\bar{w}$ and $\bar{v}$ over its alphabet, $\bar{w} \Rightarrow \bar{v}$ if and only if $\pi(\bar{w}) \Rightarrow \pi(\bar{v})$. We will therefore often identify well-formed strings of letters and their canonical projections, as well as the operations performed on these two classes of objects.

The following definition introduce the main object of study of this section: derivation graphs.

Definition 3.4.4. *Consider a insertion-deletion system $\Gamma = (V, T, A, I, D)$ of size $(1, 1, 0; 1, 1, 0)$ and a derivation $C : w \Rightarrow_\Gamma^* v$, $w, v \in V^*$. A derivation graph corresponding to C is the construct $G_C = (G, h)$, where $G = (\bar{V}, E)$, $\bar{V}$ is the (finite) set of letters appearing in C, $E \subseteq \bar{V} \times \bar{V}$ is the set of edges, and $h : E \to \{ins, del\}$. The edges E and the mapping h are given by the following:*

- *if the letter x inserts y in C, then $(x, y) \in E$ and $h((x, y)) = ins$,*
- *if the letter x deletes y in C, then $(x, y) \in E$ and $h((x, y)) = del$.*

Remark that derivation graphs are conceived to represent derivations and not words. This means that different derivation graphs may correspond to one and the same word, just because the same word may be derived in several different ways.

The immediate advantage of derivation graphs over derivation trees is that the information about which symbol carried a deletion operation is also stored, while in deletion trees deleted symbols are simply struck through. We also introduce the convention that, when visually representing a derivation graph, the relative horizontal position of letters in the graph should correspond to relative positions of these letters in the string. Thus the derivation graph corresponding to the last derivation tree shown in Figure 3.1 will be drawn as shown in Figure 3.2.

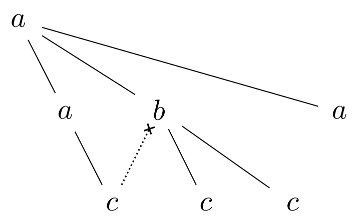


Figure 3.2: The derivation graph corresponding to the rightmost derivation tree from Figure 3.1

Instead of verbosely specifying edge labels, we use different styles. Edges marked with *ins* are drawn as the edges of insertion trees, while edges marked with *del* are drawn as dotted lines with a cross at the end.

Remark that while respecting the relative positions of letters in the string makes it possible to immediately find the word the derivation of which a graph represents by projecting all the nodes on a horizontal axis and discarding those which were deleted, it is only a convention of visual representation and is not an inherent (or necessary) part of the concept of a derivation graph.

One can clearly construct a derivation graph for any derivation of a given insertion-deletion system. However, since the derivation graph does not completely capture the information about the order in which the operations were carried out, the same derivation graph can be obtained for different derivations. For example, the following two different derivations correspond to the derivation graph shown in Figure 3.2:

$$a \Rightarrow aa \Rightarrow aba \Rightarrow aaba \Rightarrow aacba \Rightarrow aacbca \Rightarrow aacbcca \Rightarrow aaccca,$$
$$a \Rightarrow aa \Rightarrow aba \Rightarrow abca \Rightarrow abcca \Rightarrow aabcca \Rightarrow aacbcca \Rightarrow aaccca.$$

Nevertheless, if one only considers rightmost derivations, a one-to-one correspondence with derivation graphs can be established. Indeed, it can be seen from the construction of derivation graphs, that picking the rightmost operation is always possible and deterministic, so only one rightmost derivation can be constructed from a given graph, and different graphs will result in different derivations. On the other hand, Proposition 3.2.6 shows that an equivalent rightmost derivation exists for any derivation of an insertion-deletion system of size $(1, 1, 0; 1, 1, 0)$, so the class of derivation graphs is sufficiently fine-grained to serve as a tool for analysing the dynamics of such insertion-deletion systems.

In the following subsections we will show how derivation graphs can be used to illustrate and analyse some of the constructions shown in [62, 63, 64]. In all of the cases we will rewrite the leftist grammars to fit the format of insertion and deletion rules we use in this work.

3.4.2 Multiplication and Division by Two

The work [63] gives rather complicated constructions, but they are based on combinations of fairly simple structures. Since one-sided one-symbol insertion and deletion rules cannot distinguish between multiple instances of the same symbol, the author relies on strings of a special alternating structure and uses the number of alternations to encode values. To that end, they introduce partitions of the alphabet and define an alternation as a pair of neighbouring symbols which belong to different members of the partition [63, Section 2]. For example, for the alphabet $V = \{a, b, c\}$ and the subsets $A_1 = \{a, c\}$ and $A_2 = \{b\}$, the string $abacaab$ has 3 alternations: the first group ab, the immediately following ba, and the second group ab. The author interprets the number of alternations plus one as the numerical value stored in the string.

Due to the very limited amount of information insertion and deletion rules of size $(1, 1, 0; 1, 1, 0)$ have about their application sites, exactly calculating a function does not appear possible. For this reason, the author of [63] defines a weaker version of "computing" with leftist grammars: starting from a valid input word w_i with $n - 1$ alternations, where n is the input value, the leftist transformer is expected to generate a word w_o with *at least* $f(n) - 1$ alternations, where f is the function we would like to calculate.

The first construction we will discuss is the one shown in [63, Section 3]; it is a leftist system which essentially performs a multiplication by two on the input language (cf. leftist transformers of [21]). We will take the alphabet of the input language to consist of the actual input symbols $I = \{b_0, b_1\}$, partitioned into $I_0 = \{b_0\}$ and $I_1 = \{b_1\}$, plus one service symbol $B = \{b\}$. The alphabet of the output language consists of the output symbols themselves: $O = \{a_0, a_0', a_1, a_2, a_3\}$, partitioned into $O_1 = \{a_0, a_0', a_2\}$ and $O_2 = \{a_1 a_3\}$, plus the service symbols $F = \{f_0, f_1, f\}$. As we will see in more details in the following analysis, the leftist transformer from [63, Section 3] always leaves a trail of service symbols at the end of the string. The fact that in Lemma 2 of [63] the authors combine different instances of this transformer with itself explains the necessity of allowing for and taking care of service symbols in the input alphabet.

We will now formally present the discussed leftist transformer as the insertion-deletion transformer $\mathcal{T}_1 = (V_1, R_1)$, with $V_1 = I \cup O \cup F$ and the following rules in R_1 (for readability, we do not consider separate sets of insertion and deletion rules):

$$\begin{array}{llll}
(g, & a_i, & \lambda)_{del}, & i \in \{0,2\}, \\
(g, & a_0', & \lambda)_{del}, & \\
(a_0', & f, & \lambda)_{del}, & \\
(a_i, & a_{i\oplus_4 1}, & \lambda)_{del}, & i \in \{0,1,2,3\}, \\
(a_1, & f_0, & \lambda)_{del} &
\end{array}
\qquad
\begin{array}{llll}
(a_3, & f_1, & \lambda)_{del}, & \\
(f_i, & f, & \lambda)_{del}, & i \in \{0,1\}, \\
(f_i, & f_{1-i}, & \lambda)_{ins}, & i \in \{0,1\}, \\
(f_i, & b_i, & \lambda)_{ins}, & i \in \{0,1\}, b_i \in I_i, \\
(f, & b, & \lambda)_{ins}, & b \in B,
\end{array}$$

where $i \oplus_4 1 = (i+1) \bmod 4$ is the operation of increment modulo 4. Note that the original deletion rules correspond to insertion rules in this version and vice versa, in full agreement with the conversion procedure outlined in Proposition 3.2.1. Due to this conversion, the function of the transformer is reversed as well: the version we have just described will "divide" by two, instead of multiplying. Thus, $\mathcal{T}_1$ starts with a string $w = w_1w_2$, where w_1 is an alternating word over the O (with respect to partitioning into O_1 and O_2) and $w_2 \in F^*$, while its output is taken to be any string $v = v_1v_2$ such that v_1 is an alternating word over I (with respect to partitioning into I_0 and I_1) and $v_2 \in B^*$.

A direct translation of the interpretation of derivations of leftist grammars as computations to insertion-deletion systems of size $(1,1,0;1,1,0)$ yields the requirement that an output word of the insertion-deletion transformer $\mathcal{T}_1$ should contain *at most* $\lfloor n/2 \rfloor - 1$ alternations, where $n-1$ is the number of alternations in the corresponding input word, and $\lfloor n/2 \rfloor$ is the integer part of $n/2$. A rightmost derivation in which the transformer we have just shown "computes" the value $\lfloor n/2 \rfloor$ for $n = 4$ may look as follows:

$$\begin{aligned}
&ga_0a_1a_2a_3f_1\underline{f} \Rightarrow ga_0a_1a_2a_3\underline{f_1}fb \Rightarrow ga_0a_1a_2a_3\underline{f_1}b \Rightarrow ga_0a_1a_2a_3\underline{f_1}b_1b \\
\Rightarrow\ &ga_0a_1a_2a_3f_1\underline{f_0}b_1b \Rightarrow ga_0a_1a_2\underline{a_3}f_1f_0b_0b_1b \Rightarrow ga_0a_1\underline{a_2}a_3f_0b_0b_1b \\
\Rightarrow\ &ga_0\underline{a_1}a_2f_0b_0b_1b \Rightarrow ga_0\underline{a_1}f_0b_0b_1b \Rightarrow g\underline{a_0}a_1b_0b_1b \Rightarrow \underline{g}a_0b_0b_1b \Rightarrow gb_0b_1b,
\end{aligned}$$

where the symbols active in each step are underlined. The graph corresponding to this derivation is shown in Figure 3.3.

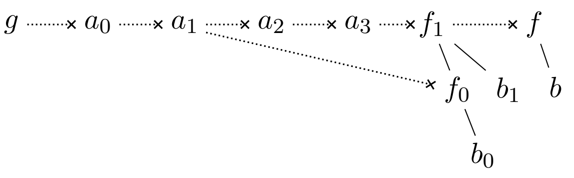


Figure 3.3: A derivation graph of the leftist transformer from [63, Section 3]

The system starts with the word $a_0a_1a_2a_3f_1f$ which contains 3 alternations of the input symbols $\{a_0, a_1, a_2, a_3\}$, corresponding to the input value 4, and two worker symbols f_1 and f. In order to produce an alternating word over the alphabet $\{b_0, b_1\}$, f_1 must insert f_0, so that both symbols can both insert b_1 and b_0. However, to subsequently erase an instance of f_1 or f_0, two symbols of the input string need to be erased, because only a_1 and a_3 can erase an indexed f-symbol. This leads to the transformation of the input word with 3 alternations into the output word b_0b_1b with one alternation, coding the value $2 = \lfloor 4/2 \rfloor$. The insertion of b by f is inessential to the division process and we include it to showcase the corresponding insertion rule.

A similar construction is given in Section 4 of the same paper [63]. This time the leftist transformer "performs" multiplication by 2, which, after conversion to the conventional semantics of insertion-deletion systems, translates to a transformer doubling the input value. We take the input alphabet to be $I = \{b_0, b_1, b_2, b_3\}$, partitioned into four subsets $I_i = \{b_i\}$, $0 \le i \le 3$, and the output alphabet $O = \{a_0, a_1, a_2, a_3\}$, partitioned in a similar way into $O_i = \{a_i\}$, $0 \le i \le 3$, plus the service symbols $F = \{f_i, f'_i, f''_i \mid 0 \le i \le 3\}$.

The insertion-deletion transformer is defined as $\mathcal{T}_2 = (V_2, R_2)$, where $V_2 = I \cup O \cup F$ and the set R_2 contains the following rules:

$$\begin{array}{llll}
(g,\ a_i,\ \lambda)_{del}, & i \in \{0,1\}, & (f_{i\oplus_4 1},\ f_i,\ \lambda)_{ins}, & i \in \{0,1,2,3\}, \\
(a_i,\ a_{i\oplus_4 1},\ \lambda)_{del}, & i \in \{0,1,2,3\}, & (f'_{i\oplus_4 1},\ f'_i,\ \lambda)_{ins}, & i \in \{0,1,2,3\}, \\
(a_i,\ f_i,\ \lambda)_{del}, & i \in \{0,1,2,3\}, & (f_i,\ b_j,\ \lambda)_{ins}, & j \in \{0,1,2,3\}, \\
 & & & i \bmod 2 = \lfloor j/2 \rfloor, \\
(a_i,\ f'_i,\ \lambda)_{del}, & i \in \{0,1,2,3\}, & (f'_i,\ b_j,\ \lambda)_{ins} & j \in \{0,1,2,3\}, \\
 & & & i \bmod 3 = |\lfloor (j-1)/2 \rfloor|, \\
(g,\ f''_i,\ \lambda)_{del}, & i \in \{0,1,2,3\}, & (f''_i,\ b_i,\ \lambda)_{ins} & i \in \{0,1,2,3\},
\end{array}$$

where $i \oplus_4 1 = (i+1) \bmod 4$ and $|n|$ is the absolute value of n. $\mathcal{T}_2$ is applied to a string $w = w_1w_2$, where w_1 is an alternating word over O (with respect to partitioning into O_i, $0 \le i \le 3$) and $w_2 \in F^*$, while its output is taken to be any alternating word over I (with respect to partitioning into I_i, $0 \le i \le 3$).

A derivation in which the value $2 \cdot 2$ is computed, may look as follows (the corresponding derivation graph is shown in Figure 3.4):

$$\begin{aligned}
&ga_0a_1\underline{f_1} \Rightarrow ga_0a_1\underline{f_1}b_3 \Rightarrow ga_0a_1\underline{f_1}b_2b_3 \Rightarrow ga_0a_1\underline{f_1}f_0b_2b_3 \Rightarrow ga_0a_1f_1\underline{f_0}b_2b_3 \\
&\Rightarrow ga_0a_1f_1\underline{f_0}b_1b_2b_3 \Rightarrow ga_0\underline{a_1}f_1f_0b_0b_1b_2b_3 \Rightarrow g\underline{a_0}a_1f_0b_0b_1b_2b_3 \Rightarrow g\underline{a_0}f_0b_0b_1b_2b_3 \\
&\Rightarrow \underline{g}a_0b_0b_1b_2b_3 \Rightarrow gb_0b_1b_2b_3.
\end{aligned}$$

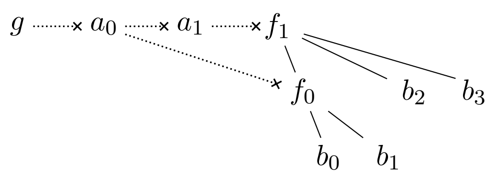


Figure 3.4: A derivation graph of the leftist transformer from [63, Section 4]

The explanation of the activity of $\mathcal{T}_2$ is symmetric to that of $\mathcal{T}_1$. Here, in order to generate a language of the form $(b_0b_1b_2b_3)^*$, for example, the system has to insert a series of indexed f-symbols, each of which will contribute two b-symbols. Because each indexed f-symbol should be erased by an a-symbol, a one-to-two correspondence is established between the number of a-symbols in the input string and the number of b-symbols in the output string.

Remark that the two derivation graphs we saw in this section are considerably more compact and less redundant than the corresponding derivations. Moreover, the graphical representations we showed in Figures 3.3 and 3.4 revealed some details of the interaction between the symbols which were not easily visible from the formal definition of the rules of the corresponding transformer. Thus, derivation graphs prove useful even for relatively small leftist grammars carrying out rather simple computations.

3.4.3 Simulation of an LBA

The next example we will consider is the main construction of [62]. The paper investigates the time complexity of the membership problem for leftist grammars and shows that this problem is PSPACE-hard. The proof is done by constructing a leftist grammar some derivations of which simulate runs of a linear bounded automaton (LBA).

A linear bounded automaton is a non-deterministic Turing machine $L = (Q, \Sigma \cup \{\rhd, \lhd\}, a_0, q_0, F, \delta)$ (cf. Section 2.2) with the additional requirement that, initially, the head is positioned between the left end marker $\rhd$ and the right end marker $\lhd$, and L is not allowed to leave the region delimited by this markers, nor to erase them. Thus, the workspace L may use is statically limited as a function of the size of the input.

In the definition of the rules from [62], the symbol Γ is used to refer to the alphabet of the LBA, Q to refer to the set of states, and Φ represents the rewriting rules of the form $a_1a_2 \to b_1b_2$, $a_1, a_2, b_1, b_2 \in \Gamma \cup Q$, defining the behaviour of the automaton (for example, the rule $q_1a \to bq_2$ moves the head of the LBA to the right, rewrites a to b, and changes the state from q_1 to q_2). The blank symbol is denoted by $\flat$. The rules of the grammar simulating this LBA, written in the conventional notation for insertion and deletion rules, are the following:

$$\begin{array}{llllll}
(\langle G,i,j,b\rangle, & \langle K,i,j,b\rangle, & \lambda)_{del}, & (H, & \langle G,j,0,\flat\rangle, & \lambda)_{del}, \\
(\langle G,i,j,b\rangle, & \langle G,i,\bar{j},b'\rangle, & \lambda)_{del}, & (\langle R,0,\flat\rangle, & H, & \lambda)_{ins}, \\
(\langle K,i,j,b\rangle, & \langle K,i,\bar{j},b'\rangle, & \lambda)_{ins}, & (\langle G,i,j,b\rangle, & \langle G_{\alpha,2},i,\bar{j},b_2\rangle, & \lambda)_{del}, \\
(\langle K,i,j,b\rangle, & \langle Y,\bar{i},j,b\rangle, & \lambda)_{ins}, & (\langle G_{\alpha,2},i,j,b_2\rangle, & \langle K,i,j,a_2\rangle, & \lambda)_{del}, \\
(\langle R,j,b\rangle, & \langle R,\bar{j},c\rangle, & \lambda)_{ins}, & (\langle G_{\alpha,2},i,j,b_2\rangle, & \langle G_{\alpha,1},i,\bar{j},b_1\rangle, & \lambda)_{del}, \\
(\langle R,j,b\rangle, & \langle Y,1,j,b\rangle, & \lambda)_{ins}, & (\langle G_{\alpha,1},i,j,b_1\rangle, & \langle K,i,j,a_1\rangle, & \lambda)_{del}, \\
(x, & \langle R,i,\flat\rangle, & \lambda)_{ins}, & (\langle G_{\alpha,1},i,j,b\rangle, & \langle G,i,\bar{j},b'\rangle, & \lambda)_{del}, \\
(x, & \langle K,i,j,b\rangle, & \lambda)_{ins}, & & &
\end{array}$$

where $i, j \in \{0,1\}$, $\bar{j} = 1 - j$, $b, b', c \in \Gamma \cup Q$, $\alpha = a_1a_2 \to b_1b_2 \in \Phi$, and $Y \in \{G\} \cup \{G_{\alpha,k} \mid \alpha \in \Phi, k \in \{1,2\}\}$. The alphabet of the system is taken to include all symbols appearing in the rules above.

According to [62, Subsection 3.1], a valid sentential form of this system looks like $vHwu$, where w is the reversed current contents of the tape between the end markers, encoded in the fourth components of a sequence of G-symbols (that is, symbols represented by tuples having G or G with subscripts as the first component), v stores the accepting configuration of the automaton encoded in the third components of a sequence of R-symbols (symbols with R in the first component), and u is a tail of K-symbols (symbols with K in the first component). The grammar moves from one configuration of the automaton to the other by having H produce a new tape, each cell of which (represented by a G-symbol) inserts a K-symbol capable of erasing the corresponding cell of the old tape. Whenever the old tape is completely erased, the last K-symbol will stay in the string and contribute to the tail of K-symbols.

The part v of every sentential form never changes in a derivation and is used to compare the current configuration with the accepting configuration of the LBA. Figure 3.5 shows the fragment of the derivation graph which corresponds to moving from configuration $\triangleright\, aq_1\flat \triangleleft$ to $\triangleright\, q_2b\flat \triangleleft$.

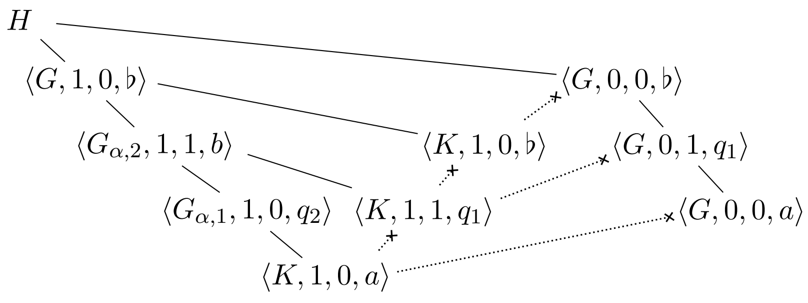

H
$\langle G, 1, 0, \flat\rangle$
$\langle G, 0, 0, \flat\rangle$
$\langle G_{\alpha,2}, 1, 1, b\rangle$
$\langle K, 1, 0, \flat\rangle$
$\langle G, 0, 1, q_1\rangle$
$\langle G_{\alpha,1}, 1, 0, q_2\rangle$
$\langle K, 1, 1, q_1\rangle$
$\langle G, 0, 0, a\rangle$
$\langle K, 1, 0, a\rangle$
+
+
×
×
×

Figure 3.5: An excerpt of the derivation graph of the insertion-deletion system some derivations of which simulate an LBA [62]

Remember that, when converting leftist grammars to insertion-deletion systems, the direction of the derivation is reversed, which means that the insertion-deletion system we have just shown will simulate the reverse evolution of the LBA by starting in its accepting configuration and successively going back through all of its configurations up to the starting configuration.

It follows from the way in which configurations of the LBA are represented in sentential forms that only *context-free* subsets of the language recognised by the leftist grammar are considered in [62]. Indeed, final words include the encoding of an accepting configuration of the LBA and its initial configuration, both of which must be of the same length. This variant of control is even more powerful than intersection with regular languages used in [63], and, while being helpful in the assessment of complexity of the membership problem for leftist grammars (and insertion-deletion systems of size $(1, 1, 0; 1, 1, 0)$), represents little interest from the point of view of the theory of formal languages and computability theory.

By looking at the derivation graph in Figure 3.5, one can remark that the third components of G- and K-symbols assure alternating sequences and help avoid accidental deletions of several instances of the same symbol. The second components on the other hand allow distinguishing between two successive tapes. Furthermore, considering the derivation graph makes it clear that the evolution of the insertion-deletion system (leftist grammar) consists of super-steps, in which the new tape is generated while the old one is erased. Therefore, in the case of this example, relying on derivation graphs helped us attain a clear understanding of the functioning of the fairly intricate construction shown [62].

3.4.4 Generation of an Exponential Language

The last example we will analyse is the one shown in Section 8 of [64]. The authors construct a leftist grammar which recognises such a non-context-free language L in which all the words of the form $(a_1a_0)^m(F_0F_1)^n$ have the property $n \geq 2^{2m-2}$. This parallels the way in which multiplications and divisions are carried out by leftist transformers in [63] in the sense that no exact correspondence, but only a certain inequality between the multiplicities of symbols, can be assured. Furthermore, the strings under consideration have a similar alternating structure. When translated into insertion-deletion systems according to Proposition 3.2.1, this leftist grammar will have the following form:

$$\begin{array}{lll}
(a_i, \; B_i, \; \lambda)_{del}, & (B_i, \; a_{1-i}, \; \lambda)_{ins}, & (F_0, \; a_0, \; \lambda)_{ins}, \\
(a_i, \; X_{i,0}, \; \lambda)_{del}, & (D_{i,j}, \; B_i, \; \lambda)_{ins}, & (F_0, \; X_{0,j}, \; \lambda)_{ins}, \\
(X_{i,j}, \; Y_{i,j}, \; \lambda)_{del} & (D_{i,1-j}, \; D_{i,j}, \; \lambda)_{ins} & (F_1, \; Y_{0,j}, \; \lambda)_{ins}, \\
(Y_{i,j}, \; D_{i,j}, \; \lambda)_{del} & (D_{i,0}, \; X_{1-i,k}, \; \lambda)_{ins} & (F_i, \; F_{1-i}, \; \lambda)_{ins}, \\
(Y_{i,j}, \; X_{i,1-j}, \; \lambda)_{del} & (D_{i,1}, \; Y_{1-i,k}, \; \lambda)_{ins} & (x, \; F_1, \; \lambda)_{ins}, \\
 & & (x, \; D_{i,j}, \; \lambda)_{ins},
\end{array}$$

where $i, j, k \in \{0, 1\}$. The language generated by this insertion-deletion system is L' with the following property:

$$L' \cap (F_1F_0)^*(a_0a_1)^* = \{(F_1F_0)^n(a_0a_1)^m \mid n \geq 2^{2m-2}\}.$$

A possible way for the above system to derive the word $(F_1F_0)^4(a_0a_1)^2$ is shown in Figure 3.6. The following is the fragment of the derivation involving the insertion paths from x to the rightmost a_1 and to the rightmost a_0, as well as the symbols $X_{0,0}$ and $Y_{0,0}$ situated in between:

$$\begin{aligned}
&\underline{x} \Rightarrow x\underline{D_{0,0}} \Rightarrow xD_{0,0}\underline{B_0} \Rightarrow \underline{x}D_{0,0}B_0a_1 \Rightarrow x\underline{D_{1,1}}D_{0,0}B_0a_1 \\
&\Rightarrow xD_{1,1}\underline{Y_{0,0}}D_{0,0}B_0a_1 \Rightarrow x\underline{D_{1,1}}Y_{0,0}B_0a_1 \Rightarrow xD_{1,1}\underline{D_{1,0}}Y_{0,0}B_0a_1 \\
&\Rightarrow xD_{1,1}D_{1,0}\underline{X_{0,0}}Y_{0,0}B_0a_1 \Rightarrow xD_{1,1}\underline{D_{1,0}}X_{0,0}B_0a_1 \Rightarrow xD_{1,1}D_{1,0}\underline{B_1}X_{0,0}B_0a_1 \\
&\Rightarrow xD_{1,1}D_{1,0}B_1\underline{a_0}X_{0,0}B_0a_1 \Rightarrow xD_{1,1}D_{1,0}B_1\underline{a_0}B_0a_1 \Rightarrow xD_{1,1}D_{1,0}B_1a_0a_1.
\end{aligned}$$

We do not give the full derivation corresponding to the graph from Figure 3.6 because just spelling out all of its steps would require a lot of space without offering additional

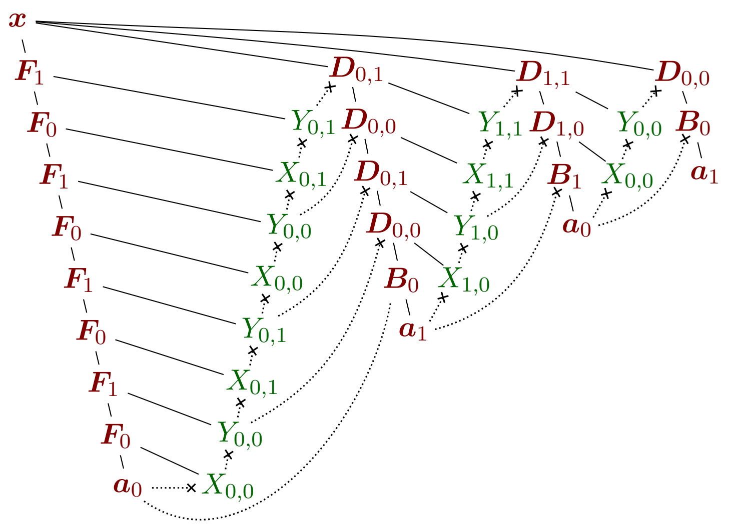


Figure 3.6: A derivation graph showing how exponential growth can be achieved in insertion-deletion systems of size $(1, 1, 0; 1, 1, 0)$ [64, Section 8]

information on the dynamics of the system.

In this graphical representation we extend the notion of the derivation graph by assigning colours to its vertices according to their position relative to terminal symbols. Thus all symbols which are connected by an insertion path to a terminal symbol are called *red*, while those which are not are called *green*. In the black-and-white version of this thesis red symbols are also shown in bold font, while green ones in light font.

Previously, in Subsections 3.4.2 and 3.4.3, we did not introduce colours, because we only focused on some parts of the derivation graphs, and also because we mainly used them for illustrative purposes. In this section, however, we will see how coloured derivation graphs can be used to actually generate interesting conclusions in a fairly straightforward manner, despite the complexity of the underlying insertion and deletion rules.

A series of properties of red and green symbols can be deduced immediately from their definition, like for example, that all red symbols together with the corresponding insertion edges form a set of subtrees of the derivation graph, or that all green symbols are eventually deleted, or that a green symbol cannot insert a red symbol. Because red symbols form trees, we will refer to insertion paths starting at a red symbol and ending with a terminal as to *red* branches.

One other property which is just as immediate but gives important insight as to how far insertion and deletion rules can carry information in a string is that, whenever a red symbol appears in a sentential form, it cannot be erased unless it has inserted another red symbol. Given that we can require that terminals are never deleted (Lemma 3.3.3), this means that red symbols have a behaviour similar to that of terminals in the sense that a red branch cannot "disappear" from the string. The implication of this for rules of size $(1,1,0;1,1,0)$ is that no rule can ever "see" what happens to the left or to the right of a red branch, and that the evolution of a green subgraph depends exclusively on the enclosing red branches.

Unfortunately, this observation is only applicable in an a posteriori fashion, because a symbol being red or green is usually decided rather late, sometimes at the very end of the derivation. A static separation between red and green symbols is not possible: while one can design a system in which symbols statically marked as green never end up producing terminal symbols, there is no way of guaranteeing that a symbol statically marked as red will actually yield a terminal. For example, if the rightmost letter B_0 in the graph from Figure 3.6 does not insert a_1, the whole branch $x - D_{0,0} - B_0$ becomes green.

Distinguishing between red and green symbols further helps analyse the dynamics of insertion-deletion systems of size $(1,1,0;1,1,0)$. For example, one can see from Figure 3.6 that every red branch has twice as many D-symbols than the next one to its right. This is because of the fact that two D-symbols have to insert a Y-symbol and then an X-symbol to delete a single D-symbol and then see both Y- and X-symbols erased. F-symbols have the same behaviour as D-symbols in what concerns interaction with the red branch immediately to the right, but since F-symbols are terminals, they do not need to be erased. Remark now that all red branches, except the leftmost one, are finally represented by a single terminal a-symbol. The leftmost branch on the other hand must contain at least 2^k F-symbols, where k is the number of red branches not containing F-symbols. Indeed, if this is not the case, there will be D-symbols left in the string with no more letters to erase them. This constraint explains how insertion-deletion systems of size $(1, 1, 0; 1, 1, 0)$ are capable of performing a series of multiplications by two.

While it is a trivial observation that red symbols form subtrees of the derivation graph, no suppositions as to the kinds of structures green symbols may arrange in can be immediately made. Our conjecture is, however, that the graph from Figure 3.6 showcases the possible functionality of green structures comprehensively. This supposition is based on the fact that the main role of green symbols is assuring a relationship between the numbers of certain symbols on two neighbouring red branches, and that the induced relations are essentially linear (e.g. there have to be twice as many D-symbols on a red branch than there are on the branch which is immediately to the right). This conjecture implies therefore that green structures can be considerably "flattened" down to the kind of up-going deletion paths formed by X- and Y-symbols in Figure 3.6, and that the derivation graphs of any insertion-deletion system can be brought into a kind of a "wave" normal form, consisting of normal red branches interspersed with up-going green deletion paths.

Another non-trivial conclusion the observation of a derivation graph in Figure 3.6 suggests is that, at least in this case, green symbols can be done away with. Indeed, consider the situation in which F_1 inserts $Y_{0,1}$, which then erases $D_{0,1}$; this deletion operation can instead be associated with F_1 directly. Remark however, that F_1 can also insert $Y_{0,0}$ which erases $D_{0,0}$; if we associate this other deletion with the same symbol F_1, a single instance of it will be sufficient to erase all D-symbols on the second leftmost red branch. To avoid such a collapse, we would need to introduce two different species of F_1: F_1' and F_1'', one of which would be able to erase $D_{0,1}$, and the other would delete $D_{0,0}$. Of course, the rules inserting F_1 would have to be altered accordingly. Furthermore, the same procedure would have to be carried out for D-symbols. All these operations clearly result in a substantial increase in the number of rules, but the resulting system will produce derivation graphs consisting of red branches only (whenever a terminal word of the form $(F_1F_0)^*(a_0a_1)^*$ is derived). The important consequence of such a transformation would be that any derivation of the new system could be simulated by non-contracting (context-sensitive) string rewriting rules. While this simulation comes with little surprise for the system we are discussing, if the conjecture about the existence of the "wave" normal form is true, the possibility of eliminating green symbols would imply that any insertion-deletion system of size $(1, 1, 0; 1, 1, 0)$ can be simulated by a context-sensitive string rewriting grammar.

Conjecture 3.4.5. *$INS_1^{1,0}DEL_1^{1,0} \subseteq CS$.*

Differentiating between red and green symbols also allows giving an easy explanation of what pure derivations are. In a pure derivation, all inserted symbols must either insert or delete useful symbols, i.e. symbols which eventually contribute to the insertion of a terminal. It is immediate that red symbols are useful. On the other hand, a green symbol is useful either if it erases a red symbol or if it inserts or deletes another useful green symbol. Thus, in the derivation graph of a pure derivation, there exists a path from every green symbol to at least one red symbol. Expressing eagerness in derivation graph parlance is more difficult and gives no additional insight as compared to the original definition, essentially because eagerness discusses the time moment at which a symbol is deleted, and time is not directly captured in derivation graphs.

We remark that derivations graphs proved very useful in analysing the functioning of the leftist grammar introduced in [64, Section 8]. We saw that, first of all, that Figure 3.6 compactly represents a derivation which would take about a page to spell out completely, and does it without losing any essential details. Moreover, colouring the letters helped discern even more structural elements right away. Further, we used some properties of red branches to explain the way in which exponentiation is performed in a clear and concise manner. Finally, we formulated two non-trivial conjectures in terms of local properties of derivation graphs: the one about the existence of the "wave" normal form, and the one suggesting that the family of languages generated by insertion-deletion systems of size $(1, 1, 0; 1, 1, 0)$ is completely contained in the family of context-sensitive languages.

We conclude this section by considering the possibilities of extending the idea of derivation graphs to insertion-deletion systems of size $(1, 1, 0; 1, 2, 0)$. Because these systems rely on insertion rules of size $(1, 1, 0)$, we can start with derivation trees, just as we did in the case of insertion-deletion systems of size $(1, 1, 0; 1, 1, 0)$ and leftist grammars. Unfortunately, elegantly representing deletions of size $(1, 2, 0)$ poses a greater challenge, mostly because such rules involve three symbols, possibly situated on three different branches. While describing this relation graphically can be as simple as connecting the three nodes with a special kind of line, reasoning about the resulting structure (which, incidentally, may be seen as a hypergraph) is considerably more difficult than about derivation graphs for insertion-deletion systems of size $(1, 1, 0; 1, 1, 0)$. Yet, a couple interesting observations can be made right away. However we choose to represent deletions, red branches can still be defined, because they only rely on insertions. It turns out that deletion rules of size $(1, 2, 0)$ can "look over" a red branch and interact with the green structure situated to the left of it. Such rules cannot involve letters separated by more than one red branch, but it follows from Corollary 3.3.17 that looking over one red branch suffices to indirectly connect letters separated by any bounded number of red branches.

Chapter 4

Insertion-deletion Systems with Control

In this chapter, we consider controlled variants of insertion-deletion systems and discuss their expressive power. We show that adding graph, semi-conditional, or random context control allows achieving computational completeness with very small rules. These results are summarised in Table 4.1 on page 86. In Section 4.5, we consider mixed networks of evolutionary processors and show how universal networks with 4, 5, and 7 rules only can be constructed.

4.1 Definitions

In this section we will introduce the three control mechanisms we will employ in this work: semi-conditional control, random context control, and graph control.

4.1.1 Sequential Rewriting Systems

Following the ideas in [37], we start by generalising the notion of a rewriting system to define control mechanisms in their most general form.

Definition 4.1.1. *A* sequential grammar *is a construct $G = (O, O_T, A, P, (\Rightarrow_G))$, where O is a set of objects, $O_T \subseteq O$ is a set of terminal objects, $A \subseteq O$ is a finite set of axioms, P is a finite set of rules, and $(\Rightarrow_G) \in O \times O$ is the derivation relation of G. Given an object $w \in O$, its membership in the set of terminal objects $w \in O_T$ is required to be decidable.*

Each rule $p \in P$ is assigned a derivation relation $(\overset{p}{\Rightarrow}_G) \subseteq O \times O$ which should verify the following conditions:

- for any $x \in O$, the number of objects $y \in O$ which satisfy $x \overset{p}{\Rightarrow}_G y$ is finite, and
- there exists a Turing machine which, given an object $x \in O$, computes all such $y \in O$ that $x \overset{p}{\Rightarrow}_G y$.

The derivation relation of the grammar is defined as the union of all derivation relations associated with every rewriting rule:

$$(\Rightarrow_G) = \bigcup_{p \in P} (\overset{p}{\Rightarrow}_G).$$

The symbol ($\Rightarrow_G^*$) will be used to refer to the reflexive and transitive closure of ($\Rightarrow_G$). We will often omit the index in ($\Rightarrow_G$) when the grammar to which we refer is clear from the context.

A derivation of G is a sequence of derivation steps $w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n$, $w_i \in O$, $1 \leq i \leq n$. The objects w_i are sometimes referred to as *configurations.* The language generated by a sequential rewriting grammar G is defined as

$$L(G) = \{v \in O_T \mid w \Rightarrow_G^* v, w \in A\}.$$

Definition 4.1.2. *A* string rewriting grammar *is defined as the following tuple: $G_s = (V^*, T^*, A, P, (\Rightarrow_{G_s}))$, where V is a finite alphabet of symbols, $T \subseteq V$ is the alphabet of terminal objects, $A \subseteq V^*$ is a finite set of axioms (start strings), and P is a set of string rewriting rules with or without additional control mechanisms.*

A string rewriting grammar is an instantiation of the notion of a sequential rewriting grammar for the case of strings. String rewriting rules without control mechanisms have the form $p : w \to v$, $w \in V^+$, $v \in V^*$, and the relations ($\overset{p}{\Rightarrow}_{G_s}$) and ($\Rightarrow_{G_s}$) are defined to capture the usual semantics of string rewriting rules. Configurations of string rewriting systems are sometimes referred to as sentential forms. Insertion-deletion systems without context-free insertion rules (i.e. rules of size $(k, 0, 0)$, $k \in \mathbb{N}$) belong to the class of string rewriting grammars without control mechanisms. Context-free insertion rules can be represented by extending the semantics of string rewriting to allow rules of the form $\lambda \to v$, $v \in V^*$, performing a context-free insertion of v.

4.1.2 Graph Control

In this subsection we first introduce the graph control mechanism for sequential rewriting systems, and then instantiate the concept for the case of conventional insertion-deletion systems.

Definition 4.1.3. *A* graph-controlled sequential rewriting system *with objects O and rules of type X, where X may stand for "string rewriting", "multiset rewriting", etc., is a sequential rewriting system $G_{gc} = (O', O'_T, A, P, (\Rightarrow_{G_{gc}}))$, with the alphabet $O' \subseteq O \times H$, where H is a finite set of state labels, and the set of terminal objects $O'_T \subseteq O_T \times H$, $O_T \subseteq O$. The set of axioms is a finite set of labelled objects: $A \subseteq O \times H$.*

The rules of G_{gc} have the form $p : (i, r, S, F)$, where $i \in H$, r is a rule of type X, and $S, F \subseteq H$ are the success and failure fields of p, respectively. For such a rule, $(i, x) \overset{p}{\Rightarrow} (j, y)$ holds if either $x \overset{r}{\Rightarrow} y$ and $j \in S$, or $x \overset{r}{\not\Rightarrow} y$ and $j \in F$. If all the rules of G have empty failure fields, then G is said to be *without appearance checking.* Otherwise, it is called a graph-controlled sequential rewriting system *with appearance checking.*

Observe that the rules of a graph-controlled system induce a graph of labels with two types of edges, corresponding to the success and failure fields. Some authors prefer defining the graph up front, e.g. [37].

Often, when defining the output of a graph-controlled sequential rewriting system, we will discard state labels in terminal configurations:

$$L(G) = \{v \in O_T \mid w \Rightarrow_{G_{gc}}^* (h, v),\ (h, v) \in O'_T, w \in A\}.$$

Depending on the correspondence between rules and state labels, as well as on the number of elements allowed in success and failure fields of rules, different equivalent ways of defining graph-controlled systems exist. Thus, if state labels are required to be in bijective correspondence with the rules, one obtains Definition 2.2.4 of [73]. If, on the other hand, multiple rules are allowed to correspond to the same label, but success and failure fields are required to be singletons sets, Definition 2.2.5 of [73] is obtained.

A graph-controlled insertion-deletion system is a graph-controlled sequential rewriting system with strings over a finite alphabet V as objects and insertion and deletion rules. We introduce the following equivalent definition, which can be found more often in the literature.

Definition 4.1.4. *A* graph-controlled insertion-deletion system *is a tuple of the form* $\Gamma = (V, T, A, H, H_0, H_f, R)$*, where* V *is a finite alphabet of symbols,* $T \subseteq V$ *is the terminal alphabet,* H *is a set of labels,* $H_0 \subseteq H$ *is the set of initial labels,* $H_f \subseteq H$ *is the set of final labels,* $A \subseteq V^* \times H_0$ *is a set of axioms, and* R *contains rules of the form* $\big(h, (u,\, \alpha,\, v)_t, S, F\big)$*, with* $t \in \{ins, del\}$*,* $h \in H$*, and* $S, F \subseteq H$*.*

According to the definition of a graph-controlled sequential rewriting system, the language generated by Γ is

$$L(\Gamma) = \{v \in T^* \mid (w_0, h_0) \Rightarrow_\Gamma^* (v, h_f), (w_0, h_0) \in A, h_f \in H_f\}.$$

We will use the symbol $ELSP_k(ins_n^{m,m'}, del_p^{q,q'})$ to refer to the family of languages generated by graph-controlled insertion-deletion systems with k states and insertion and deletion rules of size $(n, m, m'; p, q, q')$.

Example 4.1.5. *Consider the following graph-controlled insertion-deletion system* $\Gamma = (V, T, A, H, H_0, H_f, R)$*:*

$$\begin{aligned} V = T &= \{a, b\},\\ A &= \{\lambda\},\\ H &= \{1, 2\},\\ H_0 = H_f &= \{1\},\\ R &= \big\{r_1 : \big(1, (\lambda,\, a,\, \lambda)_{ins}, \{2\}, \varnothing\big), r_2 : \big(2, (\lambda,\, b,\, \lambda)_{ins}, \{1\}, \varnothing\big)\big\}. \end{aligned}$$

The rules in R *induce the following graph of states:*

$$\begin{tikzcd} 1 \arrow[r, bend left, "r_1"] & 2 \arrow[l, bend left, "r_2"] \end{tikzcd}$$

Γ *starts in the configuration* $(\lambda, 1)$ *in which its only option is inserting* a *and moving into state 2. In state 2, the only possibility* Γ *has is to insert* b *and move back into state 1. This, together with the fact that the final label is 1, implies that* $L(\Gamma) = \{v \in V^* \mid |v|_a = |v|_b\}$*.*

Graph-controlled insertion-deletion systems without appearance checking and with all success fields required to be singletons correspond to insertion-deletion P systems (e.g. [59]). Indeed, even though multiple strings may be present in a P system, the fact that they do not interact means that it suffices to trace the evolution of each of them individually.

4.1.3 Semi-conditional and Random Context Control

Like in the previous subsection, we will first define semi-conditional and random context control for the general case of sequential rewriting systems, and will then show how these definitions apply to the case of insertion-deletion systems.

Definition 4.1.6. *A* semi-conditional grammar *is a string rewriting grammar with the rules of the form $p : (\alpha \to \beta, P, Q)$, where $\alpha \in V^+$, $\beta \in V^*$, and P and Q are finite subsets of V^* called the permitting and forbidding context conditions respectively. For such a rule p, $w \overset{p}{\Rightarrow} v$, $w \in V^+$, $v \in V^*$, if $w = w'\alpha w''$ and $v = w'\beta w''$, and, additionally, all elements of P are subwords (i.e. factors) of w and no elements of Q are subwords of w.*

Definition 4.1.7. *A* random context grammar *is a semi-conditional grammar in which the permitting and forbidding context conditions are only allowed to contain single symbols.*

Semi-conditional insertion-deletion systems are defined to be those semi-condittional grammars allowing rules of the form $\lambda \to v$ in which all rules correspond to insertion or deletion rules [58].

Definition 4.1.8. *A* semi-conditional insertion-deletion system *is the construct $\Gamma = (V, T, A, R)$, where V is a finite alphabet, $T \subseteq V$ is the alphabet of terminal symbols, $A \subseteq V^*$ is a finite set of axioms, and R is finite set of rules of the form $\big((u,\ \alpha,\ v)_t, P, Q\big)$, where $t \in \{ins, del\}$, and P and Q are the permitting and forbidding context conditions correspondingly.*

The degree of the semi-conditional insertion-deletion system $\Gamma = (V, T, A, R)$ is the pair (i, j), where

$$i = \max\{|w| : \big(p, P, Q\big) \in R, w \in P\} \quad \text{and} \quad j = \max\{|v| : \big(p, P, Q\big) \in R, v \in Q\}.$$

We will use the notation $SC_{i,j}INS_n^{m,m'}DEL_p^{q,q'}$ to refer to the family of languages generated by semi-conditional insertion-deletion systems of degree (i, j) and of size $(n, m, m'; p, q, q')$.

Definition 4.1.9. *A* random context insertion-deletion system *is a semi-conditional insertion-deletion system of degree $(1, 1)$.*

We will use the following shortcut notation for the family of languages generated by such systems: $RC\,INS_n^{m,m'}DEL_p^{q,q'} = SC_{1,1}INS_n^{m,m'}DEL_p^{q,q'}$.

The following example illustrates the power of semi-conditional control, which may be used to assure complex interactions between the symbols in the string. Section 4.3 conducts further study of the power of this control mechanism.

Example 4.1.10. *We will construct a semi-conditional insertion-deletion system $\Gamma = (V, T, A, R)$ of size $(1, 0, 0; 1, 0, 0)$ and degree $(2, 2)$ such that $L(\Gamma) = \{a^n b^n \mid n \in \mathbb{N}\}$. The alphabets of Γ are defined as follows:*

$$V = \{a, \bar{a}, b, \bar{b}, S_1, S_2, S_3, S_4, B, E\}, \quad T = \{a, b\},$$

and the single axiom of Γ is BS_1E. We will assure that all rules of Γ can only operate on a string in which clean and barred versions of terminal symbols alternate, *and in which the beginning and end markers B and E are on the corresponding ends of the string, by adding the following set to the forbidding conditions of every rule:*

$$\mathcal{N} = \{xy, \bar{x}\bar{y} \mid x, y \in T\} \cup \{B\bar{a}, bE\} \cup \{XB, EX \mid X \in V\}.$$

We will call normalised the strings which do not have any of the words of $\mathcal{N}$ as subwords.

Γ will work in cycles, during each of which it will insert the words $a\bar{a}$ and $b\bar{b}$ to the left and to the right of the center of the string, marked by one of the symbols S_i. The pair $a\bar{a}$ will be inserted by the following rules:

$$\begin{array}{llll}
\big((\lambda,\, a,\, \lambda)_{ins}, & \{S_1\}, & \{S_2, S_3, S_4\} & \cup\, \mathcal{N}\big),\\
\big((\lambda,\, S_2,\, \lambda)_{ins}, & \{aS_1\}, & \{S_2, S_3, S_4\} & \cup\, \mathcal{N}\big),\\
\big((\lambda,\, \bar{a},\, \lambda)_{ins}, & \{S_1S_2\}, & \{S_3, S_4\} & \cup\, \mathcal{N}\big),\\
\big((\lambda,\, S_1,\, \lambda)_{del}, & \{\bar{a}S_1, S_1S_2\}, & \{S_2\bar{a}, S_3, S_4\} & \cup\, \mathcal{N}\big).
\end{array}$$

Consider a string $B\alpha S_1\beta E$, with $\alpha \in (a\bar{a})^*$ and $\beta \in (b\bar{b})^*$. If a is not inserted to the left of S_1, the string will not be normalised any more (will denormalise), and Γ will halt. When a is inserted, S_2 can be inserted, and it has to be inserted to the right of S_1 in order for the insertion of $\bar{a}$ to happen. The insertion of $\bar{a}$ can only occur to the left of S_1, since otherwise the contexts of the rule erasing S_1 will never be satisfied. Γ will therefore effect the following sequence of transformations (the inserted symbols are in bold):

$$B\alpha S_1\beta E \Rightarrow B\alpha\,\boldsymbol{a}\,S_1\beta E \Rightarrow B\alpha aS_1\,\boldsymbol{S_2}\,\beta E \Rightarrow B\alpha a\,\boldsymbol{\bar{a}}\,S_1S_2\beta E \Rightarrow B\alpha a\bar{a}\,S_2\beta E.$$

When S_1 is erased, Γ will switch to the second half of the cycle, during which it will insert $b\bar{b}$ by the following rules:

$$\begin{array}{llll}
\big((\lambda,\, S_3,\, \lambda)_{ins}, & \{\bar{a}S_2, S_2b\}, & \{S_1, S_3, S_4\} & \cup\, \mathcal{N}\big),\\
\big((\lambda,\, \bar{b},\, \lambda)_{ins}, & \{S_2S_3\}, & \{S_1, S_4\} & \cup\, \mathcal{N}\big),\\
\big((\lambda,\, S_2,\, \lambda)_{del}, & \{S_3\bar{b}, S_2S_3\}, & \{\bar{b}S_2, S_1, S_4\} & \cup\, \mathcal{N}\big),\\
\big((\lambda,\, S_4,\, \lambda)_{ins}, & \{\bar{a}S_3, S_3\bar{b}\}, & \{S_1, S_2, S_4\} & \cup\, \mathcal{N}\big),\\
\big((\lambda,\, b,\, \lambda)_{ins}, & \{S_3S_4\}, & \{S_1, S_2\} & \cup\, \mathcal{N}\big).
\end{array}$$

As before, permitting and forbidding contexts are used to assure that the insertions happen at the desired sites, and that the pair of symbols S_iS_j in the center of the string follows the desired evolution orbit. Applications of these rules yield the following derivation:

$$\begin{aligned}
&B\alpha a\bar{a}S_2\beta E \Rightarrow B\alpha a\bar{a}S_2\,\boldsymbol{S_3}\,\beta E \Rightarrow B\alpha a\bar{a}S_2S_3\,\boldsymbol{\bar{b}}\,\beta E \Rightarrow B\alpha a\bar{a}\,S_3\bar{b}\beta E\\
\Rightarrow\, &B\alpha a\bar{a}S_3\,\boldsymbol{S_4}\,\bar{b}\beta E \Rightarrow B\alpha a\bar{a}S_3S_4\,\boldsymbol{b}\,\bar{b}\beta E.
\end{aligned}$$

To finalise a cycle of insertions, Γ will use the following rules:

$$\begin{array}{llll}
\big((\lambda,\, S_3,\, \lambda)_{del}, & \{S_4b, S_3S_4\}, & \{bS_3, S_1, S_2\} & \cup\, \mathcal{N}\big),\\
\big((\lambda,\, S_1,\, \lambda)_{ins}, & \{\bar{a}S_4, S_4b\}, & \{S_1, S_2, S_3\} & \cup\, \mathcal{N}\big),\\
\big((\lambda,\, S_4,\, \lambda)_{del}, & \{S_1S_4\}, & \{S_1, S_2\} & \cup\, \mathcal{N}\big),
\end{array}$$

which effectively replace the word S_3S_4 by S_1 and transform the string $B\alpha a\bar{a}S_3S_4b\bar{b}\beta E$ into $B\alpha a\bar{a}S_3S_4b\bar{b}\beta E$, thus permitting Γ to carry out the insertion *cycle once again.*

In the end, to stop the production of symbols, Γ *will just erase* S_1 *instead of starting a new cycle:*

$$\big((\lambda,\, S_1,\, \lambda)_{del}, \{\bar{S}_1, S_1 b\}, \{S_2, S_3, S_4\} \cup \mathcal{N}\big),$$

which will enable the following rule to remove all barred terminals:

$$\big((\lambda,\, \bar{x},\, \lambda)_{del}, \varnothing, \{S_1, S_2, S_3, S_4\} \cup \mathcal{N}\big),\ \textit{for } x \in T.$$

Finally, the end markers will be removed:

$$\big((\lambda,\, X,\, \lambda)_{del}, \varnothing, \{\bar{x} \mid x \in T\} \cup \{S_1, S_2, S_3, S_4\} \cup \mathcal{N}\big),\ \textit{for } X \in \{B, E\}.$$

4.2 Graph-controlled Insertion-deletion Systems

In this subsection we consider graph-controlled insertion-deletion systems with rules of size $(1,2,0;1,1,0)$ and $(1,1,0;1,2,0)$ and show that three states are sufficient for achieving computational completeness. This contrasts Conjecture 3.3.18 according to which systems with rules of such size but without control would not be computationally complete.

We first prove computational completeness for graph-controlled insertion-deletion systems of size $(1,2,0;1,1,0)$.

Theorem 4.2.1. $ELSP_3(ins_1^{2,0}, del_1^{1,0}) = RE$.

Proof. Consider a type-0 grammar $G = (N, T, P, S)$ in special Geffert normal form and let $N'' = \{A, B, C, D\} \subseteq N$. We will now construct a graph-controlled insertion-deletion system $\Gamma = (V, T, A, H, H_0, H_f, R)$ simulating G. The alphabet of Γ is $V = N \cup T \cup \{M_i \mid i : X \to \alpha \in P\} \cup \{K, K'\}$. The set of labels is defined as $H = \{1, 2, 3\}$, and the initial and the final sets of labels are $H_0 = H_f = \{1\}$. The set of rules R of Γ is constructed in the following way:

– for every $i : X \to bY \in P$ and for all $\mathbf{a} \in N''$ we add to R the following rules:

$$\begin{aligned}
&i.1 : \big(1, (X,\, M_i,\, \lambda)_{ins}, \{2\}, \varnothing\big), && i.2 : \big(2, (XM_i,\, Y,\, \lambda)_{ins}, \{3\}, \varnothing\big),\\
&i.3 : \big(2, (\mathbf{a},\, M_i,\, \lambda)_{del}, \{1\}, \varnothing\big), && i.4 : \big(3, (\mathbf{a},\, X,\, \lambda)_{del}, \{3\}, \varnothing\big),\\
&i.5 : \big(3, (\mathbf{a}M_i,\, b,\, \lambda)_{ins}, \{2\}, \varnothing\big);
\end{aligned}$$

– for every $i : X \to Yb \in P$ and for all $\mathbf{a} \in N''$ we add to R the following rules:

$$\begin{aligned}
&i.1 : \big(1, (X,\, M_i,\, \lambda)_{ins}, \{2\}, \varnothing\big), && i.2 : \big(2, (XM_i,\, b,\, \lambda)_{ins}, \{3\}, \varnothing\big),\\
&i.3 : \big(2, (\mathbf{a},\, M_i,\, \lambda)_{del}, \{1\}, \varnothing\big), && i.4 : \big(3, (\mathbf{a},\, X,\, \lambda)_{del}, \{3\}, \varnothing\big),\\
&i.5 : \big(3, (\mathbf{a}M_i,\, Y,\, \lambda)_{ins}, \{2\}, \varnothing\big);
\end{aligned}$$

– for the erasing rule $i_1 : AB \to \lambda \in P$ we add to R the following rules:

$$\begin{aligned}
&i_1.1 : \big(1, (\lambda,\, K,\, \lambda)_{ins}, \{2\}, \varnothing\big), && i_1.2 : \big(2, (K,\, A,\, \lambda)_{del}, \{3\}, \varnothing\big),\\
&i_1.3 : \big(2, (\lambda,\, K,\, \lambda)_{del}, \{1\}, \varnothing\big), && i_1.4 : \big(3, (K,\, B,\, \lambda)_{del}, \{2\}, \varnothing\big);
\end{aligned}$$

– for the erasing rule $i_2 : CD \to \lambda \in P$ we add to R the following rules:

$$\begin{array}{ll} i_2.1 : \big(1, (\lambda,\, K',\, \lambda)_{ins}, \{2\}, \varnothing\big), & i_2.2 : \big(2, (K',\, C,\, \lambda)_{del}, \{3\}, \varnothing\big), \\ i_2.3 : \big(2, (\lambda,\, K',\, \lambda)_{del}, \{1\}, \varnothing\big), & i_2.4 : \big(3, (K',\, D,\, \lambda)_{del}, \{2\}, \varnothing\big); \end{array}$$

– finally for $i_3 : S \to \lambda \in P$, we add to R the following rule:

$$i_3.1 : \big(1, (\lambda,\, S,\, \lambda)_{del}, \{1\}, \varnothing\big).$$

The rules of Π induce the following graph of state labels:

$$\begin{tikzcd} 1 \arrow[loop left, "R_{11}"] \arrow[r, bend left, "R_{12}"] & 2 \arrow[l, bend left, "R_{21}"] \arrow[r, bend left, "R_{23}"] & 3 \arrow[l, bend left, "R_{32}"] \arrow[loop right, "R_{33}"] \end{tikzcd}$$

In this figure, the symbol R_{ij} refers to the set of rules assuring the transition from state i to state j. These sets are defined as follows:

$$\begin{array}{ll} R_{11} = \{i_3.1\}, & R_{12} = \{i.1, i_1.1, i_2.1\}, \\ R_{23} = \{i.2, i_1.2, i_2.2\}, & R_{33} = \{i.4\}, \\ R_{32} = \{i.5, i_1.4, i_2.4\}, & R_{21} = \{i.3, i_2.3, i_3.3\}. \end{array}$$

In this listing, the labels $i.1$ through $i.5$ refer to both the group of rules simulating $X \to bY$ and the group simulating $X \to Yb$.

We state that $L(\Pi) = L(G)$. For this we show how each rule of G can be simulated in Π. Consider the configuration $(wXw', 1)$ and suppose that there is a rule $i : X \to bY$ in P. Then the following unique evolution can happen:

$$\begin{array}{l} (wXw', 1) \overset{i.1}{\Rightarrow} (wXM_iw', 2) \overset{i.2}{\Rightarrow} (wXM_iYw', 3) \\ \overset{i.4}{\Rightarrow} (wM_iYw', 3) \overset{i.5}{\Rightarrow} (wM_ibYw', 2) \overset{i.3}{\Rightarrow} (wbYw', 1). \end{array}$$

In the second step it was also possible to apply $i.3$, yielding $(wXw', 1)$, but this would just reset the simulation to the original configuration.

The rule $X \to Yb$ is simulated in a similar manner:

$$\begin{array}{l} (wXw', 1) \overset{i.1}{\Rightarrow} (wXM_iw', 2) \overset{i.2}{\Rightarrow} (wXM_ibw', 3) \\ \overset{i.4}{\Rightarrow} (wM_ibw', 3) \overset{i.5}{\Rightarrow} (wM_iYbw', 2) \overset{i.3}{\Rightarrow} (wYbw', 1). \end{array}$$

The rule $i_1 : AB \to \lambda$ is simulated as follows (the case of rule $i_2 : CD \to \lambda$ is treated in an analogous way). First a symbol K is inserted in a context-free manner into the string ww' by using rule $i_1.1$, yielding $(wKw', 2)$. If the symbol to the right of K is not an A, then the only possibility is to apply rule $i_1.3$ which deletes K and returns into state 1. If K is inserted in front of a symbol A ($w' = Aw''$) then rule $i_1.2$ can be applied and the system moves to the configuration $(wKw'', 3)$. Now if w'' does not start with B, then the evolution of this word is stopped and does not yield a result. Otherwise ($w'' = Bw'''$), rule $i_1.4$ is applied yielding $(wKw''', 2)$. Now the computation may continue in the same manner and K either eliminates another pair of symbols AB if this is possible, or the system returns to state 1 and to a string without K and is then ready for further evolution.

Now in order to complete the proof, we observe that the only sequences of rules leading to a terminal derivation in Π correspond to the simulation sequences shown above. Indeed, we pointed out that any subderivation diverging from these sequences either leads to Π halting with no output, or makes it return to one of the previous configurations, thus having no effect on the result. Hence, a derivation in G can be reconstructed from any derivation in Π, and therefore $L(G) = L(\Pi)$. $\square$

We will now show that graph-controlled insertion-deletion systems with rules of symmetric size $(1, 1, 0; 1, 2, 0)$ are computationally complete as well. We will use a technique similar to the one shown in the proof above to prove the result for rules of this size.

Theorem 4.2.2. *$ELSP_3(ins_1^{1,0}del_1^{2,0}) = RE$.*

Proof. Consider the type-0 grammar $G = (N, T, S, P)$ in special Geffert normal form and denote $N'' = \{A, B, C, D\} \subseteq N$. We will now construct a graph-controlled insertion-deletion system $\Gamma = (V, T, A, H, h_0, h_f, R)$ simulating G. The set of labels of Γ is $H = \{1, 2, 3\}$, the set of initial labels is $H_0 = \{1\}$ and the set of final labels is $H_f = \{1\}$ as well. The alphabet of Γ contains new special symbols per each rule of G and is defined in the following way:

$$\begin{aligned} V = &\{M_i, \bar{Y}_i, M_i' \mid i : X \to bY \in P\} \\ &\cup \{M_i, N_i, \bar{Y}_i, M_i' \mid i : X \to Yb \in P\} \\ &\cup \{K, K'\} \cup N \cup T. \end{aligned}$$

The set of rules R of Γ is constructed in the following way:

– for every $i : X \to bY \in P$ we add to R the following rules:

$$\begin{aligned} &i.1 : \big(1, (\lambda,\ M_i,\ \lambda)_{ins}, \{2\}, \varnothing\big), && i.2 : \big(2, (M_i,\ \bar{Y}_i,\ \lambda)_{ins}, \{2\}, \varnothing\big), \\ &i.3 : \big(2, (M_i,\ b,\ \lambda)_{ins}, \{3\}, \varnothing\big), && i.4 : \big(2, (b\bar{Y}_i,\ X,\ \lambda)_{del}, \{1\}, \varnothing\big), \\ &i.5 : \big(3, (\lambda,\ M_i,\ \lambda)_{del}, \{2\}, \varnothing\big); \end{aligned}$$

– for every $i : X \to Yb \in P$ and for all $\mathbf{a} \in N''$ we add to R the following rules:

$$\begin{aligned} &i.1 : \big(1, (\lambda,\ M_i,\ \lambda)_{ins}, \{2\}, \varnothing\big), && i.2 : \big(2, (M_i,\ N_i,\ \lambda)_{ins}, \{2\}, \varnothing\big), \\ &i.3 : \big(2, (N_i,\ b,\ \lambda)_{ins}, \{3\}, \varnothing\big), && i.4 : \big(2, (\bar{Y}_i b,\ X,\ \lambda)_{del}, \{1\}, \varnothing\big), \\ &i.5 : \big(3, (M_i,\ \bar{Y}_i,\ \lambda)_{ins}, \{3\}, \varnothing\big), && i.6 : \big(3, (\lambda,\ M_i,\ \lambda)_{del}, \{3\}, \varnothing\big), \\ &i.7 : \big(3, (\mathbf{a}\bar{Y}_i,\ N_i,\ \lambda)_{del}, \{2\}, \varnothing\big); \end{aligned}$$

– moreover, for every $i : X \to bY \in P$ or $i : X \to Yb \in P$, we add to R the following rules:

$$\begin{aligned} &i'.1 : \big(1, (\lambda,\ M_i',\ \lambda)_{ins}, \{2\}, \varnothing\big), && i'.2 : \big(2, (M_i',\ Y,\ \lambda)_{ins}, \{3\}, \varnothing\big), \\ &i'.3 : \big(2, (\lambda,\ M_i',\ \lambda)_{del}, \{1\}, \varnothing\big), && i'.4 : \big(3, (M_i'Y,\ \bar{Y}_i,\ \lambda)_{del}, \{2\}, \varnothing\big); \end{aligned}$$

– for the erasing rule $i_1 : AB \to \lambda \in P$ we add to R the following rules:

$$\begin{aligned} &i_1.1 : \big(1, (\lambda,\ K,\ \lambda)_{ins}, \{2\}, \varnothing\big), && i_1.2 : \big(2, (K,\ A,\ \lambda)_{del}, \{3\}, \varnothing\big), \\ &i_1.3 : \big(2, (\lambda,\ K,\ \lambda)_{del}, \{1\}, \varnothing\big), && i_1.4 : \big(3, (K,\ B,\ \lambda)_{del}, \{2\}, \varnothing\big); \end{aligned}$$

– for the erasing rule $i_2 : CD \to \lambda \in P$ we add to R the following rules:

$$\begin{array}{ll} i_2.1 : \big(1, (\lambda,\, K',\, \lambda)_{ins}, \{2\}, \varnothing\big), & i_2.2 : \big(2, (K',\, C,\, \lambda)_{del}, \{3\}, \varnothing\big), \\ i_2.3 : \big(2, (\lambda,\, K',\, \lambda)_{del}, \{1\}, \varnothing\big), & i_2.4 : \big(3, (K',\, D,\, \lambda)_{del}, \{2\}, \varnothing\big); \end{array}$$

– finally, for $i_3 : S' \to \lambda \in P$, we add to R the following rule:

$$i_3.1 : \big(1, (\lambda,\, S',\, \lambda)_{del}, \{1\}, \varnothing\big).$$

The rules of Π induce the following graph of state labels:

$$\begin{tikzcd}
1 \arrow[loop left, "R_{11}"] \arrow[r, bend left, "R_{12}"] & 2 \arrow[loop above, "R_{22}"] \arrow[l, bend left, "R_{21}"] \arrow[r, bend left, "R_{23}"] & 3 \arrow[loop right, "R_{33}"] \arrow[l, bend left, "R_{32}"]
\end{tikzcd}$$

In this figure, the symbol R_{ij} refers to the set of rules assuring the transition from state i to state j. The set R_{11} only contains $i_3.1$. The set R_{12} contains rules $i.1$ from the first two groups, as well as $i'.1$, $i_1.1$, and $i_2.1$. The set R_{22} contains rules $i.2$ from the first two groups. R_{23} contains $i.3$ from the first two groups, as well as $i'.2$, $i_1.2$, and $i_2.2$. R_{33} contains rules $i.5$ and $i.6$ of the second group. R_{32} contains rules $i.5$ of the first group, $i.7$ of the second group, as well as $i'.4$, $i_1.4$, and $i_2.4$. Finally, R_{21} contains rules $i.4$ of the first and the second groups, as well $i'.3$, $i_1.3$, and $i_2.3$.

We claim that $L(\Pi) = L(G)$. For this we show how each rule of G can be simulated in Π. We immediately remark that the simulation of the rules $AB \to \lambda$ and $CD \to \lambda$ is done in exactly the same way as in the proof of Theorem 4.2.1.

Rule $i : X \to bY$**.** Consider the configuration $(wXw', 1)$ and suppose that there is a rule $i : X \to bY$ in P. The simulation of this rule occurs in two phases: in the first phase we rewrite X to $b\bar{Y}_i$, while in the second one we substitute $\bar{Y}_i$ with Y. The following is the valid first-phase simulation sequence in Π:

$$\begin{aligned} &(wXw', 1) \overset{i.1}{\Rightarrow} (wM_iXw', 2) \overset{i.2}{\Rightarrow} (wM_i\bar{Y}_iXw', 2) \\ \overset{i.3}{\Rightarrow}\ &(wM_ib\bar{Y}_iXw', 3) \overset{i.5}{\Rightarrow} (wb\bar{Y}_iXw, 2) \overset{i.4}{\Rightarrow} (wb\bar{Y}_iw', 1). \end{aligned}$$

The second phase happens due to rules $i'.1$ through $i'.4$ and consists of the following steps:

$$\begin{aligned} &(w\bar{Y}_ibw', 1) \overset{i'.1}{\Rightarrow} (wM_i'\bar{Y}_ibw', 2) \overset{i'.2}{\Rightarrow} (wM_i'Y\bar{Y}_ibw', 3) \\ \overset{i'.4}{\Rightarrow}\ &(wM_i'Ybw', 2) \overset{i'.3}{\Rightarrow} (wYbw', 1). \end{aligned}$$

We claim the both the first phase and the second phase simulation sequences are the only ones which can happen in valid derivations of Π. Indeed, consider the string wXw' into which $i.1$ has inserted an instance of M_i. By inspecting the symbol requirements of the rules associated with state 2, we conclude that only rules $i.2$ and $i.3$ may become applicable. Suppose that rule $i.3$ is applied directly. If, for example, M_i was inserted to the right of X, this would result in the configuration $(\gamma M_ib\gamma''Xw', 3)$. The case when $i.1$ inserts M_i to the right of X is treated similarly. Now, the only way to further move the computation out of state 3 is by applying

$i.5$ which will erase the instance of M_i and move the system into state 2. However, no more rules will be applicable from now on, because the string contains no service symbols at all, but the system is in state 2.

Suppose now that, after the application of $i.1$, rule $i.2$ is applied $k > 1$ times. A subsequent application of $i.3$ will insert an instance of b after M_i, thus yielding a substring of the form $M_i b(\bar{Y}_i)^k$. Again, the only way to move the system out of state 3 is to erase the symbol M_i which results in a string with a substring of k instances of $\bar{Y}_i$ in state 2. If X is situated to the left of $(\bar{Y}_i)^k$, the string cannot contain $\bar{Y}_i X$, which is required by $i.3$. On the other hand, if X is to the right of $(\bar{Y}_i)^k$, it will not be possible to apply $i.3$ again, because the string does not contain the substring $\bar{Y}_i X$ preceded by a symbol from N''. Finally, if $i.1$ does not insert M_i just to the left of X, Γ will not be able to move out of state 2, thus blocking without producing any meaningful result.

We will focus on the second-phase simulation sequence now. The application of rule $i'.1$ inserts an instance of M_i' somewhere and moves the system into state 2. There are only two rules that may become applicable: $i'.2$ and $i'.3$. Suppose that $i'.3$ is applied directly after $i'.1$. In this case the system will come back into the configuration it was in before the application of $i'.1$ without doing any changes to the string whatsoever. Therefore, to actually modify the string, rule $i'.2$ must be applied.

An application of $i'.2$ inserts exactly one instance of Y after M_i' and moves the system in state 3. Now, the only way to exit this state is by applying $i'.4$, which means that, if the application of rule $i'.1$ has not inserted M_i' to the left of $\bar{Y}_i$, the system Γ will unproductively block in the third state. Consequently, after the application of $i'.4$, the string must be of the form $wM_i'Ybw'$. At this point, two rules are still applicable, $i'.2$ and $i'.3$. Suppose that rule $i'.2$ is applied a second time and inserts another instance of Y after M_i', thus yielding the string $wM_i'YYbw'$ and moving the system in state 3. Now, however, rule $i'.4$ is not applicable because the string lacks $\bar{Y}_i$ and Γ will thus block. Therefore, the only productive way to move out of the second state is to apply $i'.3$.

Rule $i : X \to Yb$. Again, the simulation of i happens in two phases: in the first phase we rewrite X to $\bar{Y}_i b$, while in the second phase we substitute $\bar{Y}_i$ with Y. Since the second phase of the simulation happens in exactly the same way as in the case of the rule $X \to bY$, we will only focus on the first-phase simulation sequence:

$$\begin{aligned}(wXw', 1) &\overset{i.1}{\Rightarrow} (wM_iXw', 2) \overset{i.2}{\Rightarrow} (wM_iN_iXw', 2) \overset{i.3}{\Rightarrow} (wM_iN_ibXw', 3)\\ &\overset{i.5}{\Rightarrow} (wM_i\bar{Y}_iN_ibXw', 3) \overset{i.6}{\Rightarrow} (w\bar{Y}_iN_ibXw', 3) \overset{i.7}{\Rightarrow} (w\bar{Y}_ibXw', 2) \overset{i.4}{\Rightarrow} (w\bar{Y}_ibw', 1).\end{aligned}$$

We claim that the first-phase simulation sequence we have just shown is the only possible in a terminal derivation of Γ. We will now consider the variations that can interfere with this subderivation and show that none of them can influence the result of a computation of Γ.

Consider the application of $i.1$ which inserts M_i into the original string wXw' and moves the system into the configuration $(\gamma M_i\gamma' Xw', 2)$, $w = \gamma\gamma'$. The case when M_i is inserted to the right of X is treated in a similar way. In the current situation, the only applicable rule is $i.3$, which may insert k instances of N_i, thus yielding the string $\gamma M_i(N_i)^k\gamma' Xw'$. If one discards the possibility of producing yet more instances of N_i, the only other way to evolve is the application of rule $i.3$ to insert a b after one of the N_i's and thereby move the system into state 3. The new configuration will have the form $(\gamma M_i(N_i)^{k_1}b(N_i)^{k_2}\gamma' Xw', 3)$, where $k_1 \geq 1$ and $k_1 + k_2 = k$. We immediately remark that the only way for Γ to move out of this configuration is to apply rule $i.7$. This rule requires that there is a substring of $\bar{Y}_i N_i$ preceded by a symbol from N''. The string $\gamma M_i(N_i)^{k_1}b(N_i)^{k_2}\gamma' Xw'$, with which the system has just arrived in state 3, does not contain any instances of $\bar{Y}_i$, but rule $i.5$ can introduce them. Suppose this latter rule is applied t times, $t \geq 0$, thus yielding the following result:

$$\gamma M_i(\bar{Y}_i)^t(N_i)^{k_1}b(N_i)^{k_2}\gamma' Xw'.$$

Clearly, $i.7$ is not yet applicable, because there are no instances of $\bar{Y}_i$ preceded by symbols from N''. The only way to reach this situation is to apply $i.6$ to obtain the string

$$\gamma(\bar{Y}_i)^t(N_i)^{k_1}b(N_i)^{k_2}\gamma' Xw'.$$

Rule $i.7$ imposes an even stronger requirement: the instance of $\bar{Y}_i$ which is preceded by a symbol from N'' must be immediately followed by N_i. Since instances of $\bar{Y}_i$ can only be inserted to the right of M_i, and since the process of inserting N_i's has already been completed in state 2, applying $i.7$ actually requires that exactly one instance of $\bar{Y}_i$ should have been inserted by $i.5$ (i.e., it requires that $t = 1$), which means that the string must have the form

$$\gamma\bar{Y}_i(N_i)^{k_1}b(N_i)^{k_2}\gamma' Xw'.$$

An application of $i.7$ will erase the leftmost instance of N_i and will move the system in state 2 with the following string:

$$\gamma\bar{Y}_i(N_i)^{k_1-1}b(N_i)^{k_2}\gamma' Xw'.$$

Rule $i.3$ will still be applicable at this moment. Remark, however, that the string resulting from such an application would contain no instances of M_i, so the rule which might become applicable is $i.7$; it would remove yet another instance of N_i following $\bar{Y}_i$. Applications of rules $i.3$ and $i.7$ in a loop are only possible as long as there are instances of N_i just to the right of $\bar{Y}_i$ and then Γ would either block in state 3 or move in state 1 by an application of $i.4$.

We can now assert that the general form of configurations in which Γ is in state 2 after at least one passage through state 3 is $\gamma\bar{Y}_i(N_i^*(N_ib)^*)^*\gamma' Xw'$. If we discard the possibility of yet again retracing the loop formed by $i.3$ and $i.7$, the only other way for Γ to proceed is to apply $i.4$. However, in order for this rule to be applicable, there has to exist a substring $\bar{Y}_i bX$. The only way to have exactly one b between $\bar{Y}_i$ and X is, firstly, to have $i.1$ insert M_i exactly to the left of X (that is, γ' should be zero) and, secondly, to only apply $i.3$ once during the whole simulation process, thus obtaining the string $\gamma\bar{Y}_i bXw'$ in state 2. The application of $i.4$ will then erase the X and successfully finish the rewriting of X into $\bar{Y}_i b$. □

4.3 Semi-conditional Insertion-deletion Systems

In this subsection we show that semi-conditional insertion-deletion systems of degree $(2, 2)$ and with rules of size $(1, 0, 0; 1, 0, 0)$ are computationally complete. We start by proving that this result is optimal in the sense that semi-conditional insertion-deletion systems of the same degree, but with rules of size $(1, 0, 0; 0, 0, 0)$ (i.e. insertion-only systems), do not generate more than context-free languages.

Proposition 4.3.1. *$SC_{2,2}INS_1^{0,0}DEL_0^{0,0} \subseteq CS$.*

Proof. Let $\Gamma = (V, T, A, I, D)$ be a semi-conditional insertion-deletion system of size $(1, 0, 0; 0, 0, 0)$. We will construct a semi-conditional grammar $G = (N, T, S, P)$ in the following way:

$$\begin{aligned}
N &= \{S\} \cup \{N_a \mid a \in T\},\\
P &= \big\{\big(S \to N_{a_1} \dots N_{a_n}, \varnothing, \varnothing\big) \mid a_1 \dots a_n \in A\big\}\\
&\quad \cup \big\{\big(N_b \to N_b N_a, E, F\big), \big(N_b \to N_a N_b, E, F\big)\\
&\qquad \mid \big((\lambda,\, a,\, \lambda)_{ins}, E, F\big) \in I,\ b \in T\big\}\\
&\quad \cup \big\{\big(N_a \to a, \varnothing, \varnothing\big) \mid a \in T\big\}.
\end{aligned}$$

Observe that G accurately simulates Γ in the sense that $L(\Gamma) = L(G)$. This, combined with the fact that the family of languages generated by semi-conditional grammars without erasing rules is contained in CS [109], proves the statement of the theorem. □

We will now show a normal form for random context string rewriting grammars.

Lemma 4.3.2. *For an arbitrary random context grammar $G = (N, T, S, R)$, there exist an equivalent random context grammar $G' = (N', T, S, R')$ such that $L(G') = L(G)$ and the length of any right-hand side in R' is either 2 or 0.*

Proof. We will define the alphabet of the grammar G' as $N' = Q_W \cup N$, where

$$Q_W = \{W_r^{(i)} \mid r : \big(A \to u_1 u_2 \dots u_n, P, Q\big) \in R, 1 \le i \le n\}.$$

The rules of G' are given by the following set:

$$\begin{aligned}
R' = \big\{&\big(A \to u_1 W_r^{(1)}, P, Q \cup Q_W\big), \big(W_r^{(i)} \to u_{i+1} W_r^{(i+1)}, \varnothing, \varnothing\big),\\
&\big(W_r^{(n)} \to \lambda, \varnothing, \varnothing\big) \mid r : \big(A \to u_1 u_2 \dots u_n, P, Q\big) \in R, 1 \le i < n\big\}\\
\cup \big\{&r \mid r : \big(A \to \lambda, P, Q\big) \in R\big\},
\end{aligned}$$

We claim that G' can correctly simulate any rule r of G. Indeed, the contexts of r are checked in the application of the rule $A \to u_1 W_r^{(1)}$. The presence of Q_W in the forbidding set ensures that no other rule simulations (including r itself) may start before the already running simulation has finished. Thus G' can reproduce any derivation of G, and, moreover, any derivation involving symbols from Q_W ends up simulating a rule of G, which concludes the proof. □

We proceed to showing that semi-conditional insertion-deletion systems of degree $(2, 2)$, with rules of size $(1, 0, 0; 1, 0, 0)$, are computationally complete. The main inconvenience with systems of such a small size is that there is no direct way to check the context of a specific symbol in the string. To address this issue, we will encode every symbol $a \in V$ with a pair of symbols $\hat{a}\bar{a}$ and place special markers B and E at the beginning and the end of the string respectively, similarly to what we did in Example 4.1.10. Every rule will have a forbidding context which will check whether the string has this form. We will call this forbidding context the *normalisation condition*.

To operate at a specific locus in the string, we will insert some "service" symbols and rely on the permitting and forbidding contexts of the subsequent rules to check whether the insertion has happened in the appropriate position. We then further insert and delete symbols as we need, and the normalisation condition included in every rule will ensure that the operations we are performing only happen in the neighbourhood of the service symbols. Whenever the proper organisation of the string is broken, no rules will be further applicable, thus blocking the computation at a string which contains non-terminal symbols.

We will now show the inclusion $\lambda RC_{ac} \subseteq SC_{2,2}INS_1^{0,0}DEL_1^{0,0}$, where λRC_{ac} is the family of languages generated by random context grammars with erasing rules, which is known to be equal to RE [27]. The proof will be split into several propositions formally stating the necessary properties of the construction which will be described in the following several paragraphs.

Consider an arbitrary random context grammar $G = (N, T, S, R)$ and let $V = N \cup T$. We can assume that, for any random context rule $r : \big(A \to u, P, Q\big)$ of G, either $|u| = 2$, or $|u| = 0$ (Lemma 4.3.2). Further consider $\bar{V} = \{\bar{a} \mid a \in V\}$, and let $c : V^* \to (V\bar{V})^*$ be the morphism given by $c(a) = a\bar{a}$. The normalisation condition is the following set:

$$Q_{\mathcal{N}} = \{xy, \bar{x}\bar{y} \mid x, y \in V\} \cup \{B\bar{x}, xE \mid x \in V\} \cup \{uB, Eu \mid u \in V \cup \bar{V}\}.$$

Those words over the extended alphabet $V' = V \cup \bar{V} \cup \{B, E\}$ which do *not* have strings from $Q_{\mathcal{N}}$ as substrings are said to satisfy the normalisation condition, or to be normalised. A normalised string thus has the form $B(V\bar{V})^*E$.

We will now show how to construct the semi-conditional insertion-deletion system $\Gamma = (V_\Gamma, T, A_\Gamma, I_\Gamma, D_\Gamma)$ which simulates G. The alphabet V_Γ contains the service symbols, as well as the clean and barred versions of every symbol in the alphabet of G:

$$\begin{aligned} V_\Gamma &= Q_\# \cup Q_\$ \cup \{a, \bar{a} \mid a \in V\}, \text{ where} \\ Q_\# &= \{\#_p \mid p : \big(A \to \lambda, P, Q\big) \in R\}, \\ Q_\$ &= \{\$_q^{(i)} \mid q : \big(A \to \alpha, P, Q\big) \in R, |\alpha| = 2, 1 \le i \le 5\}. \end{aligned}$$

The only axiom of the constructed insertion-deletion system is $BS\bar{S}E$. According to the strategy briefly described above, Γ operates by pinpointing a locus with service symbols, erasing the left-hand side of the simulated rule, and then inserting the right-hand side. The correct positioning of inserted non-service symbols is guaranteed by the normalisation condition which is part of the forbidding context of every rule of Γ.

The set of rules of Γ is constructed in the following way:

– for every rule $p : \big(X \to \lambda, P, Q\big) \in R$, we add to I_Γ and D_Γ the rules

$$\begin{aligned} &p.1 : \big((\lambda,\ \#_p,\ \lambda)_{ins}, \{X\bar{X}\} \cup P, Q \cup Q_\# \cup Q_\$ \cup Q_{\mathcal{N}}\big), \\ &p.2 : \big((\lambda,\ X,\ \lambda)_{del}, \{\#_p X\}, Q_{\mathcal{N}}\big), \\ &p.3 : \big((\lambda,\ \bar{X},\ \lambda)_{del}, \{\#_p \bar{X}\}, \{X\#_p\} \cup Q_{\mathcal{N}}\big), \\ &p.4 : \big((\lambda,\ \#_p,\ \lambda)_{del}, \varnothing, Q_{\mathcal{N}}\big); \end{aligned}$$

– for every rule $q : \big(X \to YZ, P, Q\big) \in R$, $X, Y, Z \in V$, we add to I_Γ and D_Γ the rules

$$\begin{aligned}
q.1 &: \big((\lambda,\, \$_q^{(1)},\, \lambda)_{ins}, \{X\bar{X}\} \cup P, Q \cup Q_\# \cup Q_\$ \cup Q_\mathcal{N}\big),\\
q.2 &: \big((\lambda,\, \$_q^{(2)},\, \lambda)_{ins}, \{\$_q^{(1)}X\}, \{\$_q^{(2)}\} \cup Q_\mathcal{N}\big),\\
q.3 &: \big((\lambda,\, X,\, \lambda)_{del}, \{\$_q^{(1)}X, \bar{X}\$_q^{(2)}\}, Q_\mathcal{N}\big),\\
q.4 &: \big((\lambda,\, \$_q^{(3)},\, \lambda)_{ins}, \{\$_q^{(1)}\bar{X}\}, \{\$_q^{(2)}\bar{X}, \$_q^{(3)}, X\$_q^{(1)}\} \cup Q_\mathcal{N}\big),\\
q.5 &: \big((\lambda,\, \$_q^{(1)},\, \lambda)_{del}, \{\$_q^{(3)}\$_q^{(1)}\}, Q_\mathcal{N}\big),\\
q.6 &: \big((\lambda,\, \bar{X},\, \lambda)_{del}, \{\$_q^{(3)}\bar{X}\}, \{\$_q^{(1)}\} \cup Q_\mathcal{N}\big),\\
q.7 &: \big((\lambda,\, \$_q^{(4)},\, \lambda)_{ins}, \{\$_q^{(3)}\$_q^{(2)}\}, \{\$_q^{(1)}, \$_q^{(4)}, X\$_q^{(3)}\} \cup Q_\mathcal{N}\big),\\
q.8 &: \big((\lambda,\, \$_q^{(2)},\, \lambda)_{del}, \{\$_q^{(4)}\$_q^{(3)}\}, Q_\mathcal{N}\big),\\
q.9 &: \big((\lambda,\, \$_q^{(5)},\, \lambda)_{ins}, \{\$_q^{(4)}\}, \{\$_q^{(2)}, \$_q^{(5)}\} \cup Q_\mathcal{N}\big),\\
q.10 &: \big((\lambda,\, Y,\, \lambda)_{ins}, \{\$_q^{(4)}\$_q^{(3)}, \$_q^{(3)}\$_q^{(5)}\}, \{Y\$_q^{(4)}\} \cup Q_\mathcal{N}\big),\\
q.11 &: \big((\lambda,\, \bar{Y},\, \lambda)_{ins}, \{\$_q^{(4)}Y, Y\$_q^{(3)}, \$_q^{(3)}\$_q^{(5)}\}, \{\$_q^{(5)}\bar{Y}\} \cup Q_\mathcal{N}\big),\\
q.12 &: \big((\lambda,\, Z,\, \lambda)_{ins}, \{\$_q^{(4)}Y, \bar{Y}\$_q^{(3)}, \$_q^{(3)}\$_q^{(5)}\}, \{Z\$_q^{(4)}\} \cup Q_\mathcal{N}\big),\\
q.13 &: \big((\lambda,\, \bar{Z},\, \lambda)_{ins}, \{\$_q^{(4)}Y, \bar{Y}\$_q^{(3)}, \$_q^{(3)}Z, Z\$_q^{(5)}\}, \{\$_q^{(5)}\bar{Z}\} \cup Q_\mathcal{N}\big),\\
q.14 &: \big((\lambda,\, \$_q^{(3)},\, \lambda)_{del}, \{\$_q^{(4)}Y, \bar{Y}\$_q^{(3)}, \$_q^{(3)}Z, \bar{Z}\$_q^{(5)}\}, Q_\mathcal{N}\big),\\
q.15 &: \big((\lambda,\, \$_q^{(4)},\, \lambda)_{del}, \{\$_q^{(4)}Y, \bar{Z}\$_q^{(5)}\}, \{\$_q^{(3)}\} \cup Q_\mathcal{N}\big),\\
q.16 &: \big((\lambda,\, \$_q^{(5)},\, \lambda)_{del}, \varnothing, \{\$_q^{(3)}, \$_q^{(4)}\} \cup Q_\mathcal{N}\big);
\end{aligned}$$

– we also add the rules

$$\begin{aligned}
z.1 &: \big((\lambda,\, B,\, \lambda)_{del}, \varnothing, Q_\mathcal{N} \cup (V_\Gamma \setminus \{\{B, E\} \cup \{a, \bar{a} \mid a \in T\}\})\big),\\
z.2 &: \big((\lambda,\, E,\, \lambda)_{del}, \varnothing, \{B\})\big\},\\
z.3 &: \big((\lambda,\, \bar{a},\, \lambda)_{del}, \varnothing, \{B, E\}\big),\ \text{for all } a \in V \setminus T.
\end{aligned}$$

We will start by analyzing the third and simplest group of rules. The goal of these rules is to consecutively erase the symbols B, E, and $\bar{a}$, for $a \in T$, in the clean-up phase of the simulation.

Lemma 4.3.3. *Rules $z.1$ through $z.3$ transform any string of the form $w = B\beta E$, where $\beta = c(\alpha)$, $\alpha \in T^*$, into α.*

Proof. Remark that w is a normalised string consisting of the start and end markers B and E, and of the images of the original terminals under the morphism c.

The forbidding condition of the first rule in the third group of rules ensures that the string is in the normalised form and that it only contains the symbols B and E, and the images of the original terminals under c. At this stage, no rules from the first two groups are applicable (the actual simulation of a derivation of the original random context grammar has already been finished).

The first rule in the third group of rules erases B. This enables the second rule in this group to erase E. This in its turn enables the third rule in this group, which eventually removes all barred symbols from the string, leaving only terminals. □

We will now turn to simulating the erasing rule $p : (X \to \lambda, P, Q)$ and will show that the first group of rules works correctly. Before discussing the actual proof, though, we remark that the simulation of any rule starts by introducing a service symbol from $Q_\# \cup Q_\$$. During the simulation, there are always service symbols in the string, and the simulation of a rule application ends by removing all service symbols. On the other hand, the forbidding contexts of the rules of Γ which start simulations of rule applications all include $Q_\# \cup Q_\$$. This means that, after a simulation of a rule application has started, no other simulations can start.

Lemma 4.3.4. *Rules* $p.1$ *through* $p.4$ *correctly simulate an application of the random context rule* $p : (X \to \lambda, P, Q)$.

Proof. The correct simulation sequence has the following form:

$$w_1 X \bar{X} w_2 \overset{p.1}{\Rightarrow} w_1 \#_p X \bar{X} w_2 \overset{p.2}{\Rightarrow} w_1 \#_p \bar{X} w_2 \overset{p.3}{\Rightarrow} w_1 \#_p w_2 \overset{p.4}{\Rightarrow} w_1 w_2,$$

where $w_1, w_2 \in V_\Gamma^*$ and all sentential forms are normalised.

To prove that, in a terminal derivation, Γ cannot modify the string in a way different from what we have just shown, for each rule $p.i$, $1 \le i \le 4$, we will describe the set of strings $in(p.i)$ it is applicable to and the set of strings $out(p.i)$ that it can produce. We will then show that all derivations which result in a normalised string different from $w_1 X \bar{X} w_2$ correctly simulate the original rule.

The set $in(p.1)$ contains the strings which are in the normalised form (because the forbidding context includes $Q_\mathcal{N}$), which do not have traces of other simulations (because the forbidding context includes $Q_\# \cup Q_\$$), which contain a substring $X\bar{X}$, and which contain all the permitting contexts P of the original rule p and do not contain any of the forbidding contexts Q of the original rule. In other words, any string in $in(p.1)$ can be written as $Bc(\alpha)E$, where $\alpha \in in(p)$ is a string to which the original rule p could be applied.

Rule $p.1$ inserts $\#_p$ somewhere in the string. Consider a (normalised) string $w = w_1 X \bar{X} w_2 \in in(p.1)$. After an application of $p.1$ we can have either of the following cases: $w_1 \#_p X \bar{X} w_2$, $w_1 X \#_p \bar{X} w_2$, $w_1 X \bar{X} \#_p w_2$, or, eventually, $\alpha \#_p \beta$, where α does not end in X or $\bar{X}$, and β and does not start with X or $\bar{X}$. It turns out that only the first case is possible in terminal derivations, or else the string is not modified. Indeed, in the sentential form $w_1 X \#_p \bar{X} w_2$ rule $p.2$ is not applicable because the string does not contain $\#_p X$, while rule $p.3$ is not applicable, because the string includes $X \#_p$. Rule $p.4$ is applicable, however, but it will simply remove $\#_p$ from the string, thus resetting the simulation to the starting string w. Using the same reasoning for the other three cases, we conclude that the only way in which Γ can continue evolving is producing the string $w_1 \#_p X \bar{X} w_2$.

Consider now rule $p.2$. The set $in(p.2)$ contains normalised strings with $\#_p X$ as substrings, the application of $p.2$ removes an instance of X from the string. Note that, if it removes an instance of X which is not to the right $\#_p$, the two barred symbols which surrounded X will be one next to the other, which will denormalise the string, so $p.2$ has to delete the correct instance of X for the system not to block.

The set $in(p.3)$ consists of the normalised strings containing $\#_p \bar{X}$ and not containing $X \#_p$. As we have seen above, $p.3$ is never applicable after $p.1$ has been applied. On other hand, $p.2$ is only applicable after $p.1$ has inserted the $\#_p$ at the correct location. Therefore, $p.3$ is only applicable after $p.2$ has been applied. The application of $p.3$ either removes the desired instance of $\bar{X}$, thus transforming

$w_1\#_p\bar{X}w_2$ into $w_1\#_p w_2$, or removes a different instance of $\bar{X}$ and denormalises the string. Consequently, in a terminal derivation, an application of $p.3$ always results in the step $w_1\#_p\bar{X}w_2 \overset{p.3}{\Rightarrow} w_1\#_p w_2$.

Rule $p.4$ is less deterministic, since it can be applied to any normalized string containing $\#_p$. In a terminal derivation, this symbol can appear in the string only after the application of $p.1$. If rule $p.4$ gets applied immediately after $p.1$, the derivation returns to the original string w. If rule $p.4$ gets applied after $p.2$, the system will arrive at the string $w_1\bar{X}w_2$, which is not normalized (the string $w_1\bar{X}$ contains two successive barred symbols). Finally, if rule $p.4$ is applied after rule $p.3$, it transforms the string $w_1\#_p w_2$ into $w_1 w_2$, thereby finalising a correct simulation of the erasing rule $p : \big(X \to \lambda, P, Q\big)$. $\square$

We will now focus on the simulation of the generic context-free rule $q : \big(X \to YZ, P, Q\big)$. The simulation is based on ideas similar to what we have just exposed. It starts by enclosing the site of X by two markers which further delete X and insert Y and Z.

Lemma 4.3.5. *Rules $q.1$ through $q.16$ correctly simulate an application of the random context rule $q : \big(X \to YZ, P, Q\big)$.*

Proof. The correct simulation sequence consists of two phases. In the first phase, the *deletion phase*, Γ selects a word $X\bar{X}$ and replaces it with three service symbols, thereby delimiting the sites at which the words $Y\bar{Y}$ and $Z\bar{Z}$ will be inserted later. In the second phase, the *insertion phase*, Γ uses the markers which have been inserted in the deletion phase to insert $Y\bar{Y}$ and $Z\bar{Z}$ to thereby complete the simulation. A correct first-phase simulation sequence is as follows:

$$\begin{aligned}
& w_1X\bar{X}w_2 \overset{q.1}{\Rightarrow} w_1\$_q^{(1)}X\bar{X}w_2 \overset{q.2}{\Rightarrow} w_1\$_q^{(1)}X\bar{X}\$_q^{(2)}w_2 \overset{q.3}{\Rightarrow} w_1\$_q^{(1)}\bar{X}\$_q^{(2)}w_2 \\
\overset{q.4}{\Rightarrow}\ & w_1\$_q^{(3)}\$_q^{(1)}\bar{X}\$_q^{(2)}w_2 \overset{q.5}{\Rightarrow} w_1\$_q^{(3)}\bar{X}\$_q^{(2)}w_2 \overset{q.6}{\Rightarrow} w_1\$_q^{(3)}\$_q^{(2)}w_2 \overset{q.7}{\Rightarrow} w_1\$_q^{(4)}\$_q^{(3)}\$_q^{(2)}w_2 \\
\overset{q.8}{\Rightarrow}\ & w_1\$_q^{(4)}\$_q^{(3)}w_2 \overset{q.9}{\Rightarrow} w_1\$_q^{(4)}\$_q^{(3)}\$_q^{(5)}w_2.
\end{aligned}$$

A correct second-phase simulation sequence is as follows:

$$\begin{aligned}
& w_1\$_q^{(4)}\$_q^{(3)}\$_q^{(5)}w_2 \overset{q.10}{\Rightarrow} w_1\$_q^{(4)}Y\$_q^{(3)}\$_q^{(5)}w_2 \overset{q.11}{\Rightarrow} w_1\$_q^{(4)}Y\bar{Y}\$_q^{(3)}\$_q^{(5)}w_2 \\
\overset{q.12}{\Rightarrow}\ & w_1\$_q^{(4)}Y\bar{Y}\$_q^{(3)}Z\$_q^{(5)}w_2 \overset{q.13}{\Rightarrow} w_1\$_q^{(4)}Y\bar{Y}\$_q^{(3)}Z\bar{Z}\$_q^{(5)}w_2 \overset{q.14}{\Rightarrow} w_1\$_q^{(4)}Y\bar{Y}Z\bar{Z}\$_q^{(5)}w_2 \\
\overset{q.15}{\Rightarrow}\ & w_1Y\bar{Y}Z\bar{Z}\$_q^{(5)}w_2 \overset{q.16}{\Rightarrow} w_1Y\bar{Y}Z\bar{Z}w_2.
\end{aligned}$$

We remark that all the strings shown in this derivation sequence satisfy the normalisation condition. We will now prove that the only possible sequence of events in a terminal derivation is the one we have just described.

Deletion phase. Consider the set of strings $in(q.1)$ to which $q.1$ can be applied. This set consists of those normalised strings without traces of other simulations which include a substring $X\bar{X}$, all strings from the permitting context P of the original random context rule q, and which do not contain any element of the forbidding context Q of rule q. Therefore the strings in $in(q.1)$ correspond to those strings to which q could be applied in a derivation of the original random context grammar G.

Per each string in $w \in in(q.1)$, the set $out(q.1)$ contains all the strings which can be obtained by inserting an instance of $\$_q^{(1)}$ into w. Notice that only rules of the second group may be applicable to such strings. Among those, only $q.2$ might be applicable to a string in $out(q.1)$. Remark that $q.4$ cannot be applicable right after $q.1$. Indeed, the permitting context of this rule demands that $\$_q^{(1)}$ be just to the left of $\bar{X}$, while the forbidding context requires that there be no X to the left of $\$_q^{(1)}$. However, the only rule that has been applied up to now is $q.1$, which has only inserted a $\$_q^{(1)}$. If this symbol were inserted right before a substring $X\bar{X}$, then the permitting context of $q.4$ would not be satisfied. On the other hand, if $\$_q^{(1)}$ was inserted in between X and $\bar{X}$, the forbidding contexts of $q.4$ will again render it inapplicable. Finally, if $\$_q^{(1)}$ is inserted at any other place in the string, the permitting context of $q.4$ will not be satisfied once more.

The contexts of rule $q.2$ guarantee that Γ will proceed with a terminal computation only if $\$_q^{(1)}$ was inserted by $q.1$ immediately to the left of an X. An application of $q.2$ adds an instance of $\$_q^{(2)}$ to the string. This application may render rule $q.3$ applicable, but no other rule. Remark that if $\$_q^{(2)}$ is inserted in such a way that the contexts of $q.3$ are not satisfied, i.e. $\$_q^{(2)}$ is not to the right of an $\bar{X}$, Γ will halt with a string with non-terminal symbols.

An application of $q.3$ removes an X from the string. As we have already seen in other situations, if this application removes an instance of X which is not near $\$_q^{(1)}$ or $\$_q^{(2)}$, the string will denormalise and Γ will halt prematurely. However, remark that if there is an X near $\$_q^{(2)}$, then it is necessarily to the right of $\$_q^{(2)}$, because the contexts of $q.3$ are satisfied only when there is an $\bar{X}$ to the left of $\$_q^{(2)}$. Consequently, if $q.3$ deletes this X, the string will contain $\$_q^{(2)}\bar{X}$, because, before this deletion, the string contained $\$_q^{(2)}X$, and any X must be followed by an $\bar{X}$. rule $q.4$ (and all other rules) will not be applicable in these circumstances, and Γ will thus halt at a string with non-terminals. Therefore, in a terminal derivation, $q.3$ can only remove the X which is just to the right of $\$_q^{(1)}$.

After the application of $q.3$, the system arrives at a string which contains a substring $\$_q^{(1)}\bar{X}$ and a substring $\bar{X}\$_q^{(2)}$. Note that, while we do want to see the substring $\$_q^{(1)}\bar{X}\$_q^{(2)}$, we have not ensured this as of yet.

Once $q.3$ has been applied, only rule $q.4$ may become applicable; this rule inserts $\$_q^{(3)}$ somewhere in the string. The only rule that may now become applicable is rule $q.5$. The contexts of $q.5$ ensure that $\$_q^{(3)}$ was inserted right before $\$_q^{(1)}$ (or else Γ halts with error). An application of rule $q.5$ removes the $\$_q^{(1)}$, thus assuring that the string contains the substrings $\$_q^{(3)}\bar{X}$ and $\bar{X}\$_q^{(2)}$, and enabling $q.6$. Rule $q.6$ removes an instance of $\bar{X}$. Once again, this instance should be either near $\$_q^{(3)}$ or near $\$_q^{(2)}$. Remark, however, that the contexts of $q.3$ and $q.4$ ensure that there is no $\bar{X}$ near $\$_q^{(2)}$, so there is nothing for $q.6$ to remove in the neighbourhood of $\$_q^{(2)}$. On the other hand, if there is an $\bar{X}$ to the left of $\$_q^{(3)}$ and it is removed, the only applicable rule will again be $q.6$. If it is applied a second time and finally removes the $\bar{X}$ after $\$_q^{(3)}$, no more rules will be applicable: after $q.6$ deletes an $\bar{X}$ to the left of $\$_q^{(3)}$, the string will include the substring $X\$_q^{(3)}$, which is listed in the forbidding context of $q.7$.

Observe now that the permitting context of $q.7$ imposes a strong condition: it requires that the string contain $\$_q^{(3)}\$_q^{(2)}$. This condition ensures that Γ will halt if

$q.2$ did not insert $\$_q^{(2)}$ just to the right of the group $X\bar{X}$ already marked from the left by $\$_q^{(1)}$ (and subsequently by $\$_q^{(3)}$).

The insertion of $\$_q^{(4)}$ by $q.7$ marks the end of the deletion phase. The only rule that may become applicable after $q.7$ is $q.8$. The contexts of this rule ensure that $q.7$ has inserted the $\$_q^{(4)}$ right before the $\$_q^{(3)}$, while the application of the rule removes the $\$_q^{(2)}$. Once $\$_q^{(2)}$ is removed, rule $q.9$ becomes applicable and inserts an instance of $\$_q^{(5)}$ somewhere into the string.

Insertion phase. The insertion of $\$_q^{(5)}$ marks the beginning of the insertion phase of the simulation of the non-erasing random context rule q. The only rule which may become applicable after the application of $q.9$ is $q.10$. The contexts of this rule require that $q.9$ has inserted the $\$_q^{(5)}$ immediately to the right of $\$_q^{(3)}$. This guarantees that the string has now the aspect $w_1\$_q^{(4)}\$_q^{(3)}\$_q^{(5)}w_2$.

The actual insertion starts with an application of rule $q.10$, which results in the addition of Y to the string. This insertion can occur at any locus in the string; however, it has to happen in the immediate neighbourhood of the symbols $\$_q^{(4)}$, $\$_q^{(3)}$, or $\$_q^{(5)}$ to not denormalise the string and halt the system. Remark, however, that we cannot control the number of times Y is inserted in the same way as we did with the service symbols of the $\$_q^{(i)}$ family, i.e. by adding Y to the forbidding context of the rule which inserts it. The reason is that Y is not a service symbol and is therefore not necessarily unique in the string we are working on. To exercise the desired control, we utilise both the contexts of the rules and the normalisation condition in the following way. First of all, note that the only rule that may become applicable after the insertion of Y is $q.11$ (rules $q.12$ through $q.15$ require more symbols to be present near $\$_q^{(4)}$, $\$_q^{(3)}$, and $\$_q^{(5)}$). The permitting context of $q.11$ ensures that there is a Y between $\$_q^{(4)}$ and $\$_q^{(3)}$, and that there is nothing between $\$_q^{(3)}$ and $\$_q^{(5)}$. The other two possible loci where $q.10$ could insert Y are to the left of $\$_q^{(4)}$ and to the right of $\$_q^{(5)}$. Note, however, that if Y is inserted to the right of $\$_q^{(5)}$, the string will become denormalised, because, in a normalised string, $\$_q^{(5)}$ is already followed by a non-barred symbol. On the other hand, if $q.10$ performs the insertion to the left of $\$_q^{(4)}$, all rules will be disabled (including $q.10$) and so Y will never appear between $\$_q^{(4)}$ and $\$_q^{(3)}$, forcing Γ to halt on a string with non-terminals. Yet further, if $q.10$ is applied more than once to insert more than one instance of Y between $\$_q^{(4)}$ and $\$_q^{(3)}$, the string will become denormalized as well. Therefore, in a terminal derivation, Y is necessarily inserted only once and only between $\$_q^{(4)}$ and $\$_q^{(3)}$.

When $q.11$ is enabled, it will insert a $\bar{Y}$. The only rule that may become applicable after this is $q.12$. Similarly to the case with $q.10$, if $\bar{Y}$ is inserted to the left of $\$_q^{(4)}$, the string will become denormalized, because, in a normalized string, $\$_q^{(4)}$ is already preceded by a barred symbol. On the other hand, if $\bar{Y}$ is inserted to the right of $\$_q^{(5)}$, the forbidding context of $q.11$ will prevent the insertion from happening again, $\bar{Y}$ will never appear to the left of $\$_q^{(3)}$, and Γ will halt with error. The contexts of $q.12$ ensure that $\bar{Y}$ is not inserted between $\$_q^{(3)}$ and $\$_q^{(5)}$, and the normalisation condition guarantees that, if $\bar{Y}$ does get inserted to the left of $\$_q^{(3)}$, then it is inserted exactly once.

The next applicable rule, $q.12$, inserts a Z. This may only render applicable rule

$q.13$ and, as with Y and $\bar{Y}$ before, the contexts of $q.12$ and $q.13$ ensure that Z is inserted between $\$_q^{(3)}$ and $\$_q^{(5)}$. Rule $q.13$ inserts a $\bar{Z}$, this may only enable $q.14$, and yet again the contexts of $q.13$ and $q.14$ ensure the correct positioning of the inserted symbol and limit the number of insertions. Therefore, in a valid derivation, the string $w_1\$_q^{(4)}Y\bar{Y}\$_q^{(3)}\$_q^{(5)}w_2$ is always transformed into $w_1\$_q^{(4)}Y\bar{Y}\$_q^{(3)}Z\bar{Z}\$_q^{(5)}w_2$.

The several remaining rules finalise the simulation by erasing the service symbols. Rule $q.14$ removes $\$_q^{(3)}$, enabling $q.15$. Rule $q.15$ removes $\$_q^{(4)}$, enabling $q.16$, which finally removes $\$_q^{(5)}$ and concludes the overall transformation of $w_1X\bar{X}w_2$ into $w_1Y\bar{Y}Z\bar{Z}w_2$. □

The three lemmas we have just proved imply that $SC_{2,2}INS_1^{0,0}DEL_1^{0,0} \supseteq \lambda RC_{ac} = RE$. To show the actual equality between the two classes of languages, we recall that $INS_1^{0,0}DEL_1^{0,0} \subseteq CF$ [116]; therefore, $SC_{2,2}INS_1^{0,0}DEL_1^{0,0} \subseteq SC_{2,2} \subseteq RE$. Hence the following statement is true.

Theorem 4.3.6. *$SC_{2,2}INS_1^{0,0}DEL_1^{0,0} = RE$.*

4.4 Random Context Insertion-deletion Systems

In this section we investigate the computational power of random context insertion-deletion systems. We show that such systems with rules of size $(2,0,0;1,1,0)$ are computationally complete, while the systems of size $(1,1,0;p,1,1)$, with $p > 0$, are not. A comparison between the results shown in this section and those from the previous one stresses that limiting the permitting and forbidding contexts of rules to only include symbols (instead of words) is a considerable restriction.

We start by showing the computational completeness of the family of random context insertion-deletion systems of size $(2,0,0;1,1,0)$. The proof will be based on the simulation of an arbitrary grammar $G = (N,T,S,R)$ in special Geffert normal form. We suppose that $N = N' \cup F$, with $F = \{A,B,C,D\}$, and $N' \cap F = \varnothing$. We will use an approach similar to the one shown in Section 4.3: given a string $w \in V^*$ which can be derived from the axiom of G, the corresponding string w_Γ of the system Γ will not directly contain any symbols from F. Instead, for any occurrence of a symbol $U \in F$ in w, w_Γ will include the substring $\hat{U}\bar{U}$ in the corresponding position. Although this approach is indeed similar to the one used in Section 4.3, at this time we only apply it to the symbols in F, and neither will we set up special markers at the beginning or the end of the string. This is because, in this proof, we only use the pairs of hatted and barred symbols to control the number of times a deletion rule is applied

Recall that, since the grammar G is in special Geffert normal form, a non-terminal symbol $X \in N'$ cannot appear more than once in any derivable string. This means that any deletion rule of Γ which erases such a non-terminal X will never be applied more than once in a row. On the other hand, the symbols from F can appear any number of times and in arbitrary succession, and we need to simulate the erasing rules $AB \to \lambda$ and $CD \to \lambda$. However, since we cannot check more than a one-symbol context to the left of the deletion site, and the permitting and forbidding contexts may only include single symbols, it is impossible to control the number of times a rule erasing a symbol from F is applied when there are runs of such symbols, i.e. substrings from $\{U\}^*$, where $U \in F$. The goal of having hatted and barred symbols is to avoid precisely this kind of substrings.

We will rewrite the rules of the original grammar G in the following way. For any rule $X \to UY$ (or $X \to YU$), $U \in F$, we will take the rule $X \to \hat{U}\bar{U}Y$ (or $X \to Y\hat{U}\bar{U}$ respectively). Further, instead of a rule $UV \to \lambda$, we will take $\hat{U}\bar{U}\hat{V}\bar{V} \to \lambda$, where $U = A$ and $V = B$, or $U = C$ and $V = D$. To avoid simulating longer right-hand sides, we further rewrite these new productions into $X \to \hat{U}Y'$ and $Y' \to \bar{U}Y$ ($X \to Y'\bar{U}$ and $Y' \to Y\hat{U}$, respectively) and define the modified grammar $G' = (N' \cup \bar{F} \cup \hat{F}, T, S, R')$ to contain the new rules.

We will now construct the random context insertion-deletion system $\Gamma = (V_\Gamma, T, \{S\}, I_\Gamma, D_\Gamma)$ simulating the modified G. The alphabet of the new system is defined as $V_\Gamma = T \cup N' \cup (\bar{F} \cup \hat{F}) \cup N_P$, where $\hat{F} = \{\hat{X} \mid X \in F\}$, $\bar{F} = \{\bar{X} \mid X \in F\}$ and N_P is defined as follows:

$$\begin{aligned} N_P = {} & \{\#_q^{(1)}, \#_q^{(2)} \mid q : X \to YU \in R'\} \cup \{m, m', f, f'\} \\ & \cup \{\$_r^{(i)} \mid r : \hat{U}\bar{U}\hat{V}\bar{V} \to \lambda \in R', 1 \le i \le 7\}. \end{aligned}$$

Γ can simulate the rules $X \to UY$ almost directly by inserting UY and having Y erase X. The same approach does not work with $X \to YU$ because the non-terminal Y cannot be used to erase X directly any more. We will therefore need to mark the instance of X with service symbols and use them as reference points. The simulation of the erasing rules is the more complex part of the proof. Γ starts by marking the substring to erase with special symbols and uses them to perform deletions and assure that, if something goes wrong, no result is ever produced. We remark up front that deletions of $\hat{A}$ and $\hat{C}$ are delegated either to subsequent simulations or to the final cleanup phase.

The sets of rules I_Γ and D_Γ is constructed in the following way:

– for every rule $p : X \to UY$, we add the following rules to I_Γ and D_Γ:

$$p.1 : \big((\lambda,\, UY,\, \lambda)_{ins}, \{X\}, (N' \cup N_P) \setminus \{X\}\big),\ p.2 : \big((Y,\, X,\, \lambda)_{del}, \varnothing, \varnothing\big);$$

– for every rule $q : X \to YU$, we add the following rules to I_Γ and D_Γ:

$$\begin{array}{ll} q.1 : \big((\lambda,\, \#_q^{(1)}\#_q^{(2)},\, \lambda)_{ins}, \{X\}, (N' \cup N_P) \setminus \{X\}\big), & q.2 : \big((\#_q^{(2)},\, X,\, \lambda)_{del}, \varnothing, \varnothing\big), \\ q.3 : \big((\lambda,\, YU,\, \lambda)_{ins}, \{\#_q^{(1)}\}, \{X, Y\}\big), & q.4 : \big((U,\, \#_q^{(2)},\, \lambda)_{del}, \varnothing, \varnothing\big), \\ q.5 : \big((\lambda,\, \#_q^{(1)},\, \lambda)_{del}, \varnothing, \{\#_q^{(2)}\}\big); & \end{array}$$

– for every rule $r : \hat{U}\bar{U}\hat{V}\bar{V} \to \lambda$, $(U, V) \in \{(A, B), (C, D)\}$, we add the following rules to R_Γ:

$$\begin{array}{ll} r.1{:}\big((\lambda,\, \$_r^{(1)}\$_r^{(2)},\, \lambda)_{ins}, \{\bar{U}\}, N' \cup N_P\big), & r.2{:}\big((\hat{U},\, \$_r^{(1)},\, \lambda)_{del}, \varnothing, \varnothing\big), \\ r.3{:}\big((\lambda,\, f'f,\, \lambda)_{ins}, \{\$_r^{(2)}\}, \{\$_r^{(1)}, f\}\big), & r.4{:}\big((\bar{V},\, f',\, \lambda)_{del}, \varnothing, \varnothing\big), \\ r.5{:}\big((\lambda,\, m'm,\, \lambda)_{ins}, \{\$_r^{(2)}, f\}, \{f', m\}\big), & r.6{:}\big((\bar{U},\, m',\, \lambda)_{del}, \varnothing, \varnothing\big), \\ r.7{:}\big((\lambda,\, \$_r^{(3)},\, \lambda)_{ins}, \{m\}, \{m', \$_r^{(3)}\}\big), & r.8{:}\big((\$_r^{(3)},\, \$_r^{(2)},\, \lambda)_{del}, \varnothing, \varnothing\big), \\ r.9{:}\big((\$_r^{(3)},\, \bar{U},\, \lambda)_{del}, \varnothing, \{\$_r^{(2)}\}\big), & r.10{:}\big((\$_r^{(3)},\, m,\, \lambda)_{del}, \varnothing, \{\$_r^{(2)}\}\big), \\ r.11{:}\big((\lambda,\, \$_r^{(4)},\, \lambda)_{ins}, \{\$_r^{(3)}\}, \{\$_q^{(2)}, \$_q^{(4)}, m\}\big), & r.12{:}\big((\$_r^{(4)},\, \$_r^{(3)},\, \lambda)_{del}, \varnothing, \varnothing\big), \\ r.13{:}\big((\$_r^{(4)},\, \hat{A},\, \lambda)_{del}, \varnothing, \{\$_r^{(3)}, \$_r^{(5)}\}\big), & r.14{:}\big((\$_r^{(4)},\, \hat{C},\, \lambda)_{del}, \varnothing, \{\$_r^{(3)}, \$_r^{(5)}\}\big), \\ r.15{:}\big((\lambda,\, \$_r^{(5)},\, \lambda)_{ins}, \{\$_r^{(4)}\}, \{\$_r^{(3)}, \$_r^{(5)}\}\big), & r.16{:}\big((\$_r^{(5)},\, \$_r^{(4)},\, \lambda)_{del}, \varnothing, \varnothing\big), \\ r.17{:}\big((\$_r^{(5)},\, \hat{V},\, \lambda)_{del}, \varnothing, \{\$_r^{(4)}, \$_r^{(6)}\}\big), & r.18{:}\big((\lambda,\, \$_r^{(6)},\, \lambda)_{ins}, \{\$_r^{(5)}\}, \{\$_r^{(4)}, \$_r^{(6)}\}\big), \\ r.19{:}\big((\$_r^{(6)},\, \$_r^{(5)},\, \lambda)_{del}, \varnothing, \varnothing\big), & r.20{:}\big((\$_r^{(6)},\, \bar{V},\, \lambda)_{del}, \varnothing, \{\$_r^{(5)}, \$_p^{(7)}\}\big), \\ r.21{:}\big((\lambda,\, \$_r^{(7)},\, \lambda)_{ins}, \{\$_r^{(6)}\}, \{\$_q^{(5)}, \$_r^{(7)}\}\big), & r.22{:}\big((\$_r^{(7)},\, \$_r^{(6)},\, \lambda)_{del}, \varnothing, \varnothing\big), \\ r.23{:}\big((\$_r^{(7)},\, f,\, \lambda)_{del}, \varnothing, \{\$_r^{(6)}\}\big), & r.24{:}\big((\lambda,\, \$_r^{(7)},\, \lambda)_{del}, \varnothing, \{f\}\big); \end{array}$$

– finally, we add the following two rules:

$$\begin{aligned} s.1 &: \big((\lambda,\, \hat{A},\, \lambda)_{del}, \varnothing, V_\Gamma \setminus \big(T \cup \{\hat{A}, \hat{C}\}\big)\big),\\ s.2 &: \big((\lambda,\, \hat{C},\, \lambda)_{del}, \varnothing, V_\Gamma \setminus \big(T \cup \{\hat{A}, \hat{C}\}\big)\big). \end{aligned}$$

We will now prove that each of the described groups of rules carries out the corresponding simulations correctly. We start with rules of the form $X \to UY$.

Lemma 4.4.1. *Rules $p.1$ and $p.2$ always correctly simulate an application of a rule $p : X \to UY$, with $X, Y \in N_P$, $U \in \hat{F} \cup \bar{F}$.*

Proof. The correct simulation sequence is as follows:

$$w_1 X w_2 \overset{p.1}{\Rightarrow} w_1 UYX w_2 \overset{p.2}{\Rightarrow} w_1 UY w_2,$$

where $w_1, w_2 \in V_\Gamma^*$ are such strings over the alphabet of Γ that $S \Rightarrow_\Gamma^* w_1 X w_2$.

Γ starts by inserting the pair UY somewhere in the string, given that the only symbol from $N' \cup N_P$ present in the string is X. The presence of other symbols from $N' \cup N_P$ would mean that there is an on-going simulation of another rule, so Γ is not allowed to start a new simulation.

The insertion performed by $p.1$ can happen anywhere in the string; however, rule $p.2$ serves two goals at the same time: it ensures that the pair UY was inserted just to the left of X and it also removes the original instance of X. Indeed, if UY was not inserted before X, $p.2$ would not be applicable, but the string would contain two non-terminals from N_P: X and Y. By checking the contexts of other rules, one can verify that none will be applicable and Γ will therefore halt with a string with non-terminals. Consequently, in a terminal derivation of Γ, rule $p : X \to UY$ is always simulated correctly. □

The simulation of the rule $q : X \to YU$ is slightly more sophisticated than that of $p : X \to UY$, and it requires two service symbols, because this time the unique symbol Y is to the left of U.

Lemma 4.4.2. *Rules $q.1$ through $q.5$ correctly simulate the application of a rule $q : X \to YU$, with $X, Y \in N_P$, $U \in \hat{F} \cup \hat{F} \cup T$.*

Proof. The correct simulation sequence has the following form:

$$\begin{aligned} & w_1 X w_2 \overset{q.1}{\Rightarrow} w_1 \#_q^{(1)} \#_q^{(2)} X w_2 \overset{q.2}{\Rightarrow} w_1 \#_q^{(1)} \#_q^{(2)} w_2\\ \overset{q.3}{\Rightarrow}\ & w_1 \#_q^{(1)} YU \#_q^{(2)} w_2 \overset{q.4}{\Rightarrow} w_1 \#_q^{(1)} YU w_2 \overset{q.5}{\Rightarrow} w_1 YU w_2, \end{aligned}$$

where w_1 and w_2 are defined as in the case of the rule $p : X \to UY$. We will prove that this simulation sequence is the only one possible in a terminal derivation of Γ.

First of all, observe that $q.1$ is only applicable to those strings which contain an instance of X and do not contain other symbols from $N' \cup N_P$. Just like in the case of the rule $p : X \to UY$, this guarantees that there are no other simulations already running. After the application of $q.1$, only $q.2$ may become applicable. Indeed, $q.3$ is not applicable because the string still contains an instance of X, $q.4$ is not applicable because $\#_q^{(2)}$ is preceded by $\#_q^{(1)}$, while $q.5$ is not applicable because the string contains an instance of $\#_q^{(2)}$.

The application of $q.2$ after $q.1$ brings the string into the form $w_1\#_q^{(1)}\#_q^{(2)}w_2$. Remember that there was only one instance of X before this application, so there are no more instances of this symbol. The only rule that may now be rendered applicable is rule $q.3$: $q.4$ is not applicable because $\#_q^{(2)}$ is still preceded by $\#_q^{(1)}$, while q_5 is inapplicable because the string still contains $\#_q^{(2)}$. An application of $q.3$ inserts the pair YU into the string. The only rule that may become applicable after this insertion is $q.4$, and if YU was not inserted right in between $\#_q^{(1)}$ and $\#_q^{(2)}$, the system will halt with a string with non-terminals. Therefore, in order for the system to proceed, rule $q.3$ must transform the string into $w_1\#_q^{(1)}YU\#_q^{(2)}w_2$. An application of $q.4$ removes the $\#_q^{(2)}$. This enables $q.5$ which removes the $\#_q^{(1)}$ and finishes the transformation of w_1Xw_2 into w_1YUw_2. □

The simulation of an erasing rule $r : \hat{U}\bar{U}\hat{V}\bar{V} \to \lambda$ is the most sophisticated part of the argument given in this subsection. The reason is that the only way we can mark a certain position in the string is by relying on left contexts of deletion rules, and this means that we cannot reliably place a service symbol to the left of a symbol which may be present in multiple instances, like $\hat{U}$, without erasing it.

Our solution is to delay the removal of $\hat{U}$ to later phases. A part of leftover hatted symbols is erased at the very end of the simulation of a derivation of G by rules $s.1$ and $s.2$. These two rules only get activated when the string contains terminals and $\hat{A}$ and $\hat{C}$, i.e. when Γ has almost finished simulating a derivation of G which resulted in a string of terminals and only needs to clean up the remaining service symbols. The other situation in which $\hat{A}$ and $\hat{C}$ are removed is when the string contains nested quadruples of symbols from $\hat{F} \cup \bar{F}$, i.e. constructions similar to $\hat{A}\bar{A}\,\hat{C}\bar{C}\hat{D}\bar{D}\,\hat{B}\bar{B}$. In this case, a simulation of the rule $\hat{C}\bar{C}\hat{D}\bar{D} \to \lambda$ will erase $\bar{C}$, $\hat{D}$, and $\bar{D}$ thus producing the string $\hat{A}\bar{A}\,\hat{C}\,\hat{B}\bar{B}$. Rules $r.13$ and $r.14$ are meant to handle exactly these kinds of situations and assure that the extra hatted symbols are removed and do not hinder further simulations.

The proof of the following statement gives more details to how Γ erases quadruples of hatted and barred symbols and formally shows that, whenever Γ takes a wrong guess, it may not produce a result.

Lemma 4.4.3. *Rules $r.1$ through $r.24$ correctly remove a substring of the form $\hat{U}\bar{U}\gamma\hat{V}\bar{V}$, where $(U,V) \in \{(A,B),(C,D)\}$ and $\gamma \in \{\hat{A},\hat{C}\}^*$.*

Proof. We will show the correct simulation sequence for the case $\gamma = \lambda$. The case in which $\gamma \neq \lambda$ is handled by applying rules $r.13$ and $r.14$ after $r.12$ and $r.15$ a sufficient number of times.

The simulation happens in two phases. In the *first phase*, Γ inserts some markup symbols and erases $\bar{U}$:

$$\begin{aligned}
&w_1\hat{U}\bar{U}\hat{V}\bar{V}w_2 \overset{r.1}{\Rightarrow} w_1\hat{U}\$_r^{(1)}\$_r^{(2)}\bar{U}\hat{V}\bar{V}w_2 \overset{r.2}{\Rightarrow} w_1\hat{U}\$_r^{(2)}\bar{U}\hat{V}\bar{V}w_2\\
\overset{r.3}{\Rightarrow}\ &w_1\hat{U}\$_r^{(2)}\bar{U}\hat{V}\bar{V}f'fw_2 \overset{r.4}{\Rightarrow} w_1\hat{U}\$_r^{(2)}\bar{U}\hat{V}\bar{V}fw_2 \overset{r.5}{\Rightarrow} w_1\hat{U}\$_r^{(2)}\bar{U}m'm\hat{V}\bar{V}fw_2\\
\overset{r.6}{\Rightarrow}\ &w_1\hat{U}\$_r^{(2)}\bar{U}m\hat{V}\bar{V}fw_2 \overset{r.7}{\Rightarrow} w_1\hat{U}\$_r^{(3)}\$_r^{(2)}\bar{U}m\hat{V}\bar{V}fw_2 \overset{r.8}{\Rightarrow} w_1\hat{U}\$_r^{(3)}\bar{U}m\hat{V}\bar{V}fw_2\\
\overset{r.9}{\Rightarrow}\ &w_1\hat{U}\$_r^{(3)}m\hat{V}\bar{V}fw_2.
\end{aligned}$$

In the *second phase*, Γ erases $\hat{V}$ and $\bar{V}$, and also assures retroactively that the markup symbols were inserted correctly and selected a correct substring $\hat{U}\bar{U}\hat{V}\bar{V}$:

$$\begin{aligned}
& w_1\hat{U}\$_r^{(3)}m\hat{V}\bar{V}fw_2 \overset{r.10}{\Rightarrow} w_1\hat{U}\$_r^{(3)}\hat{V}\bar{V}fw_2 \overset{r.11}{\Rightarrow} w_1\hat{U}\$_r^{(4)}\$_r^{(3)}\hat{V}\bar{V}fw_2 \\
\overset{r.12}{\Rightarrow} & w_1\hat{U}\$_r^{(4)}\hat{V}\bar{V}fw_2 \overset{r.15}{\Rightarrow} w_1\hat{U}\$_r^{(5)}\$_r^{(4)}\hat{V}\bar{V}fw_2 \overset{r.16}{\Rightarrow} w_1\hat{U}\$_r^{(5)}\hat{V}\bar{V}fw_2 \\
\overset{r.17}{\Rightarrow} & w_1\hat{U}\$_r^{(5)}\bar{V}fw_2 \overset{r.18}{\Rightarrow} w_1\hat{U}\$_r^{(6)}\$_r^{(5)}\bar{V}fw_2 \overset{r.19}{\Rightarrow} w_1\hat{U}\$_r^{(6)}\bar{V}fw_2 \\
\overset{r.20}{\Rightarrow} & w_1\hat{U}\$_r^{(6)}fw_2 \overset{r.21}{\Rightarrow} w_1\hat{U}\$_r^{(7)}\$_r^{(6)}fw_2 \overset{r.22}{\Rightarrow} w_1\hat{U}\$_r^{(7)}fw_2 \overset{r.23}{\Rightarrow} w_1\hat{U}\$_r^{(7)}w_2 \\
\overset{r.24}{\Rightarrow} & w_1\hat{U}w_2.
\end{aligned}$$

The symbols m and f are the "middle marker" and the "final marker" respectively, in the sense that, after the application of $r.6$, we expect m to be located between the halves $\hat{U}\bar{U}$ and $\hat{V}\bar{V}$ of a quadruple $\hat{U}\bar{U}\hat{V}\bar{V}$, while f is expected to mark the end of the quadruple. Obviously, there is no guarantee that the insertions have happened at the desired places. Nevertheless, we will later see that, whenever the marker symbols are not positioned at proper sites, Γ halts with a string with non-terminals.

We will now prove that, in a valid derivation, Γ will always simulate the erasing rule $r : \hat{U}\bar{U}\hat{V}\bar{V} \to \lambda$ in the way we have just described. Consider the string $w = w_1\hat{U}\bar{U}\gamma\hat{V}\bar{V}w_2$, where $w_1, w_2 \in V_\Gamma^*$ are some strings over the alphabet of Γ, while $\gamma \in \{\hat{A}, \hat{C}\}^*$ is an arbitrary string containing the hatted symbols left after the incomplete erasure of some quadruples of hatted and barred symbols, as above.

First phase. The simulation starts by the application of $r.1$, which inserts the pair $\$_r^{(1)}\$_r^{(2)}$. The only rule that may become applicable after this insertion is $r.2$, and the only case when it becomes applicable is when $\$_r^{(1)}\$_r^{(2)}$ has been inserted to the right of an instance of $\hat{U}$. The application of rule $r.2$ enables $r.3$ and only this rule, which inserts the pair $f'f$ at some locus in the string. Similarly to the situation with the pair $\$_r^{(1)}\$_r^{(2)}$, the only rule which may become enabled after this insertion is the deletion rule $r.4$, which ensures that the marker symbol f is fixed after an instance of $\bar{V}$. The application of $r.4$ enables $r.5$ and only this rule, which inserts the pair $m'm$. Finally, rule $r.6$, which is the only rule to be enabled after the application of $r.5$, ensures that the marker symbol m is fixed to the right of an instance of $\hat{U}$.

After the applications of rules $r.1$–$r.6$, we would like to have the initial string w transformed into $w_1\hat{U}\$_r^{(2)}\bar{U}\,m\,\gamma\,\hat{V}\bar{V}f$. However, so far we have only ensured the presence of the pairs $\hat{U}\$_r^{(2)}$, $\bar{U}m$, and $\bar{V}f$ in the resulting string. In what follows, we will keep in mind two cases: the "good" case, when Γ produces the desired string $w_1\hat{U}\$_r^{(2)}\bar{U}\,m\,\gamma\,\hat{V}\bar{V}f$, and the "bad" case in which Γ produces a different string, which still includes the substrings $\hat{U}\$_r^{(2)}$, $\bar{U}m$, and $\bar{V}f$.

The application of $r.6$ enables $r.7$, and only this rule. Rule $r.7$ inserts $\$_r^{(3)}$ somewhere in the string. This may only enable $r.8$, which requires that $r.7$ has inserted $\$_r^{(3)}$ exactly before $\$_r^{(2)}$. This means that the pair of rules $r.7$ and $r.8$ has the effect of replacing $\$_r^{(2)}$ with $\$_r^{(3)}$. Remark once again that this is the only possible effect of the application of this pair of rules in a terminal derivation of Γ.

After the application of $r.8$, the string contains the substrings $\hat{U}\$_r^{(3)}$, $\bar{U}m$, and $\bar{V}f$. Since the string does not contain $\$_r^{(2)}$, but contains $\$_r^{(3)}$, there are two rules which may become applicable: rules $r.9$ and $r.10$. Remember, however, that we have assured the presence of $\bar{U}m$ in the string, which means that $r.10$ cannot be applicable, because $\$_r^{(3)}$ is definitely not to the left of m. This means that the only rule that may become applicable is $r.9$, which can only be enabled if $\$_r^{(3)}$ is followed by $\bar{U}$. This means, in its turn, that, for Γ not to halt, the string must include the substring $\$_r^{(3)}\bar{U}$, besides the aforementioned substrings $\hat{U}\$_r^{(3)}$, $\bar{U}m$, and $\bar{V}f$. Knowing that there cannot be more than one instance of $\$_r^{(3)}$ in the string, we conclude that, in a terminal derivation, the string must contain the substring $\hat{U}\$_r^{(3)}\bar{U}$.

The application of $r.9$ removes the instance of $\bar{U}$ which is situated to the right of $\$_r^{(3)}$. Remember that the symbols from $\hat{F} \cup \bar{F}$ are always produced in pairs of the form $\hat{U}\bar{U}$, and therefore, when the erasure of quadruples begins, the string cannot include runs of hatted or barred symbols, i.e. it cannot contain substrings of the form $\alpha \in \hat{F}^k$ or $\beta \in \bar{F}^k$, where $k \geq 2$. On the other hand, as we have already seen, a valid simulation of a rule $r : \hat{U}\bar{U}\hat{V}\bar{V} \to \lambda$ removes all barred symbols. This means that the string cannot include runs of barred symbols at any step of a derivation of Γ. Therefore, an application of $r.9$ (which removes a $\bar{U}$) can only happen once at a time, because $\bar{U}$ is necessarily followed by a non-barred symbol.

Keeping in mind all of the previous observations regarding the structure of the string ensured by the time $r.9$ has been applied, one can see that only $r.10$ may become applicable at this moment. However, this rule may only become applicable if $\$_r^{(3)}$ is followed by m. This means that, before the application of $r.9$, the string had to include the substring $\$_r^{(3)}\bar{U}m$ so that the erasure of $\bar{U}$ could place $\$_r^{(3)}$ right before m; otherwise Γ would have halted on a string with non-terminals.

We remark that, actually, a slightly stronger condition has been assured by the time $r.9$ has been applied: the string has to include the substring $\hat{U}\$_r^{(3)}\bar{U}m$. Whenever this is not the case, Γ halts on a string with non-terminals.

Second phase. An application of rule $r.10$ removes m from the string. The only rule that may now become applicable is $r.11$, which is coupled with $r.12$, and the only possible effect of these two rules is replacing $\$_r^{(3)}$ with $\$_r^{(4)}$. In the "good" case, when all markers have been positioned correctly, the string will look like $w_1\hat{U}\$_r^{(4)}\gamma\hat{V}\bar{V}w_2$, where $\gamma \in \{\hat{A}, \hat{C}\}^*$, as before. We remind that the mission of $\$_r^{(4)}$ is to remove γ (which can also be empty). This is accomplished by successive applications of $r.13$ and $r.14$. However, rule $r.15$ is also applicable after $r.12$, which means that $\$_r^{(5)}$ can be inserted even if not all of the leftover $\hat{A}$ and $\hat{C}$ were removed and $\$_r^{(4)}$ is not immediately followed by $\hat{V}$. Remark, though, that if this does happen, rules $r.13$ and $r.14$ will be rendered inapplicable. If $\$_r^{(5)}$ is not inserted before $\$_r^{(4)}$, the system will halt on a string with non-terminals; otherwise, the only applicable rule will be $r.16$, which will remove $\$_r^{(4)}$. Among the rules which are left, none is capable of removing $\hat{A}$ or $\hat{C}$ ($r.17$ can only erase $\hat{B}$ or $\hat{D}$, depending on the form of the rule r), which means that the only way in which Γ can evolve from this point is by successively applying $r.18$, $r.19$, $r.21$, and $r.22$. Observe that the erasing rules $r.17$ and $r.20$ will not be applied, because all of the dollar symbols will be followed by either $\hat{A}$ or $\hat{C}$. Therefore, after the application of $r.22$, the symbol $\$_r^{(7)}$ will not be followed by f and Γ will halt with a string with non-terminals. The conclusion is that, in the "good" case, the only valid way for Γ to evolve is by erasing the substring γ entirely, up to the moment when $\$_r^{(4)}$ is just to the left of $\hat{V}$.

At the beginning of the simulation, we have assured that f is preceded by a $\bar{V}$; on the other hand, up to the application of rule $r.15$, no instances of $\bar{V}$ have been erased. This means that f is still preceded by $\bar{V}$ by the time $r.15$ is applied. Observe now that, as different from the case of $\hat{U} \in \{\hat{A}, \hat{C}\}$, all pairs $\hat{V}\bar{V}$, $V \in \{B, D\}$, are removed entirely, i.e. in the original string $w = w_1\hat{U}\bar{U}\gamma\hat{V}\bar{V}w_2$, there were no instances of $\bar{V}$ which were not preceded by $\hat{V}$ (whenever this is not so, the string must necessarily contain non-terminals from N_P). Consequently, we have actually assured that f is preceded by the pair $\hat{V}\bar{V}$, which means that, if $\$_r^{(4)}$ is not located to the left of the pair $\hat{V}\bar{V}$ which is followed by f, Γ will never arrive at a string including $\$_r^{(7)}f$ and thus f will never be deleted. Indeed, by looking at rules $r.16$–$r.21$, one can see that the symbols $\$_r^{(5)}$ and $\$_r^{(6)}$ can condition the removal of at most one $\hat{V}$ and one $\bar{V}$ (since there can be no runs of these symbols), in this order. Therefore, if there are different symbols between $\$_r^{(4)}$ and f, it is impossible for Γ to arrive at a string where there is a dollar symbol immediately to the left of f.

Up to now, we have shown that, at the moment of application of $r.15$, $\$_r^{(4)}$ and f should mark the same pair $\hat{V}\bar{V}$ for Γ to continue the evolution. We will now prove that, in a valid derivation, Γ necessarily removes this pair.

As we have already seen above, the only valid effect of rules $r.15$ and $r.16$ is replacing $\$_r^{(4)}$ with $\$_r^{(5)}$. This renders two rules applicable: $r.17$ and $r.18$. Remark, however, that, if $r.18$ is indeed applied at this time, $r.17$ will never become applicable again. Rule $r.19$, coupled with $r.18$, will replace $\$_r^{(5)}$ with $\$_r^{(6)}$. The system Γ will not be able to apply $r.20$, because $\$_r^{(6)}$ will still be followed by $\hat{V}$, and, finally, the applications of $r.21$ and $r.22$ will not place $\$_r^{(7)}$ right before f, thus making impossible the removal of the latter two symbols and blocking the system on a string with non-terminals. Therefore, in a valid derivation, $r.17$ is always applied before $r.18$.

The pair of rules $r.20$ and $r.21$, which becomes applicable after the substitution of $\$_r^{(6)}$ with $\$_r^{(7)}$ effected by $r.18$ and $r.19$, has the same property as the pair $r.17$ and $r.18$, i.e. Γ necessarily halts on a string with non-terminals if $r.21$ is applied before $r.20$. Therefore, in a valid derivation, the pair $\hat{V}\bar{V}$ marked by the symbols $\$_r^{(4)}$ and f is always removed. The general conclusion is that Γ either simulates the rule $r : \hat{U}\bar{U}\hat{V}\bar{V} \to \lambda$ correctly or halts on a string with non-terminals. $\square$

The three lemmas above show that Γ simulates exactly the derivations of the original grammar in special Geffert normal form G, which implies the statement of the following theorem.

Theorem 4.4.4. $RC\,INS_2^{0,0}DEL_1^{1,0} = RE$.

We will now show that random context insertion-deletion systems with rules of size $(1,1,0;p,1,1)$ are not capable of generating all recursively enumerable languages. The proof is based on the observation that it is impossible to control the number of applications of an insertion rule $r : \big((x,\, y,\, \lambda)_{ins}, P, Q\big)$ when y is already present in the string. Indeed, in the case when $x \notin P \cup Q$ and $y \notin P \cup Q$, if r can be applied once, it can also be applied any number of times. Furthermore, including x in either P or Q does not in any way increase control: including x in the permitting context is redundant, while including it in the forbidding context would just make the rule never applicable. Similarly, including y into P contributes nothing in what concerns control over the number of applications of rule r. The remaining possibility is including y into the forbidding context Q. In this case, however, the rule will never be applicable to a string which already contains y.

Proposition 4.4.5. $(ab)^+ \in REG \setminus RC\,INS_1^{1,0}DEL_p^{1,1}$, $p \geq 1$.

Proof. Suppose, by contradiction, that there exists a random context insertion deletion system $\Gamma = (V, T, S, R)$ of the required size such that $L(\Gamma) = (ab)^+$. Since the statements of Lemmas 3.3.1 and 3.3.3 can be directly generalised to random-context insertion-deletion systems, we can require without losing generality that Γ contain no context-free insertion rules and no rules erasing terminals.

Consider a derivation $S \Rightarrow_\Gamma^* w$, with $w \in (ab)^+$, and pick a pair ab by regarding the string as $w = w_1 ab w_2$, where $w_1, w_2 \in (ab)^*$. Since R does not contain rules erasing terminals, the b in our pair was either added by an insertion rule or was a part of the axiom S. Note, however, that we can choose w such that $|w| > |S|$, because the length of the words in $(ab)^+$ is unbounded. Obviously, in such a w, one can always pick a pair ab which does not originate from the axiom.

Suppose that the instance of b in $w = w_1 ab w_2$ was inserted by the rule $p : \big((x, b, \lambda)_{ins}, P, Q\big)$, where $x \in V$. Then, in the derivation of w, there was a step $\gamma x \delta \overset{p}{\Rightarrow} \gamma x b \delta$, with $\gamma, \delta \in V^*$. Remark that, if b belongs to the forbidding context Q of p, this rule will only be able to insert the very first instance of b into the string. However, since the length of the words in $(ab)^+$ is unbounded, R must contain a rule which inserts b and does not include b in its forbidding context. Suppose we have chosen w and the sentential form $\gamma x \delta$ in such a way that p is this rule, i.e. $b \notin Q$; in this case $\gamma x \delta \overset{p}{\Rightarrow} \gamma x b \delta \overset{p}{\Rightarrow} \gamma x b b \delta$. But then it is possible to take the rule applications from the derivation $\gamma x b \delta \Rightarrow^* w_1 ab w_2$ and build the derivation $\gamma x b b \delta \Rightarrow^* w_1 abb w_2$, which means that $S \Rightarrow^* w_1 abb w_2 \notin (ab)^+$. This and the reasoning we showed in the previous paragraphs implies that no random context insertion-deletion system can generate the language $(ab)^+$. □

The results presented in this section are summarised in Table 4.1.

Table 4.1: A summary of results about the computational power of insertion-deletion systems with control

Size	Control	Power	Reference
$(1, 0, 0; 0, 0, 0)$	semi-conditional	$\subseteq CS$	Proposition 4.3.1
$(1, 0, 0; 1, 0, 0)$	semi-conditional	$= RE$	Theorem 4.3.6
$(1, 1, 0; p, 1, 1)$	random-context	$\not\supseteq REG$	Proposition 4.4.5
$(2, 0, 0; 1, 1, 0)$	random context	$= RE$	Theorem 4.4.4
$(1, 2, 0; 1, 1, 0)$	graph	$= RE$	Theorem 4.2.1
$(1, 1, 0; 1, 2, 0)$	graph	$= RE$	Theorem 4.2.2

4.5 Small Universal NEPs

In this section we consider universal networks of evolutionary processors (NEPs) having a small number of rules. We show that it is possible to construct a strongly universal NEP with 5 rules, and a weakly universal NEP with 4 rules. We recall that an element $A_0 \in \mathfrak{C}$, for a class of computing devices $\mathfrak{C}$, is called weakly universal if $A(x) = y$ implies $A(x) = f(A_0(g(A), h(x)))$, where h and f are the encoding and decoding functions respectively, and g is the function assigning numbers to devices in $\mathfrak{C}$, according to some fixed enumeration. If h and f are identities, the element A_0 is called strongly universal. We refer to Section 2.3 for a detailed discussion of computational completeness and universality.

Towards the end of the section, we give a construction of a universal NEP with 7 rules that efficiently (in polynomial time) simulates any Turing machine.

4.5.1 Definitions

An *evolution* rule over an alphabet V is a rule $a \to b$, with $a, b \in V \cup \{\lambda\}$ and where a and b cannot be both empty. We say that an evolution rule is a substitution rule if both a and b are not λ; it is a deletion rule if $a \neq \lambda$ and $b = \lambda$; it is an insertion rule if $a = \lambda$ and $b \neq \lambda$. The set of all evolution, substitution, deletion, and insertion rules over an alphabet V are denoted by Evo_V, Sub_V, Del_V, and Ins_V, respectively.

Given a rule $\sigma : a \to b \in Evo_V$ we define the result of the application of σ to a word $w \in V^*$ as follows:

$$\sigma(w) = \begin{cases} \{ubv \mid w = uav, u, v \in V^*\}, & \text{if } w \text{ contains } a, \\ \{w\}, & \text{otherwise.} \end{cases}$$

We can generalize this notation to languages and write $\sigma(L) = \bigcup\limits_{w \in L} \sigma(w)$.

A *network of evolutionary processors* of size n is the tuple $\Gamma = (V, N_1, \ldots, N_n, k, G)$, where V is an alphabet, $1 \leq k \leq n$ designates the output node and $N_i = (M_i, A_i, IF_i, OF_i)$ is the i-th node (processor), $1 \leq i \leq n$, in which $M_i \subseteq Evo_V$ is a finite set of evolutionary rules, A_i is a finite set of strings over V (initial strings), and IF_i and OF_i are regular languages over V specifying conditions for a string to enter and to exit a node, respectively (input and output filters). Finally, $G = (\{N_1, \ldots, N_n\}, E)$ is the graph specifying the underlying communication network.

Sometimes the NEPs as we define them here are referred to as *mixed* NEPs, in contrast with the classical definition of this computational model in which the processors are only allowed to carry out one type of operation, i.e. a processor is only allowed to execute either insertions, or deletions, or substitutions [5]. As it will be visible later, allowing mixed processors is a convention that simplifies descriptions of networks without essentially modifying their computational power.

The configuration $C = (C_1, \ldots, C_n)$ of a NEP consists of the sets of strings appearing in each node (each string appears in an arbitrary large number of copies). The system evolves from a configuration $C = (C_1, \ldots, C_n)$ to a configuration $C' = (C'_1, \ldots, C'_n)$ in two kinds of alternating steps:

– evolution step ($C \Rightarrow C'$): $C'_i = \bigcup\limits_{\substack{\sigma \in M_i \\ w \in C_i}} \sigma(w)$,

– communication step ($C \vdash C'$): $C'_i = C_i \setminus OF_i \cup \bigcup\limits_{(N_i, N_j) \in E} C_j \cap OF_j \cap IF_i$.

A computation consists of a sequence of configurations, $C^0 = (A_1, \ldots, A_n)$, $C^{2i} \Rightarrow C^{2i+1}$ and $C^{2i+1} \vdash C^{2i+2}$, for $i \geq 0$. Remember that each string is present in an arbitrarily large number of copies, which means that, after the rule applications of an evolution step, we get arbitrarily many copies of each of the resulting strings in each of the nodes of the network. The result of a (possibly infinite) computation is a language collected in a designated node N_k called the output node. Thus, $L(\Gamma) = \bigcup_{t \geq 0} C_k^t$.

For the purposes of the present thesis we define the result of the computation of Γ on an input word w, denoted by $\Gamma(w)$. We assume that Γ has the property $A_i = \varnothing$, $i \geq 1$. Then $\Gamma(w)$ can be computed by setting $A_1 = \{w\}$ and $A_i = \varnothing$, $i \geq 2$, obtaining the NEP Γ', and having Γ' evolve as usual. In this case $\Gamma(w) = L(\Gamma')$.

In this work we only consider NEPs in which all nodes contain at most one string at any step of computation. Therefore $\Gamma(w)$ is always a singleton language, and we say that the only word of $\Gamma(w)$ is the result of the computation of Γ on w.

4.5.2 Simulation of Register Machines

In this subsection we show that any register machine can be simulated by a NEP. Since such machines deal with integer numbers, we need to modify the definitions of the input and output of a NEP in order to accommodate to this property.

Let $\mathcal{M} = (k, Q, q_1, q_0, P)$ be a register machine with k registers (we remark that the initial state is q_1 and the final state is q_0). We will use unary encoding to represent a configuration $(q_i, n_1, \ldots, n_k)$ of $\mathcal{M}$: $1^i01^{n_1}0\ldots1^{n_k}0$. In this encoding, the number of symbols 1 in the first block represents the state number and the size of each next block of 1's corresponds to the value of the respective register. Hence, the initial configuration is represented by the string $101^{n_1}0\ldots1^{n_k}0$, where $n_1, \ldots, n_k$ is the input of $\mathcal{M}$. We assume that $\mathcal{M}$ produces the result in the first register and empties all other registers before halting. For a vector $v = (n_1, \ldots, n_k)$, $n_i \in \mathbb{N}$, $k > 0$, we define the result $\Gamma(v)$ of the computation of Γ on v as the length of the string $\Gamma(101^{n_1}0\ldots1^{n_k}0)$. We will use a similar definition of NEPs with input in Section 4.5.3, where Turing machines are simulated.

For a given register machine $\mathcal{M}$, we will construct the corresponding NEP simulating $\mathcal{M}$, $\Gamma_{\mathcal{M}} = (\{0, 1, \bar{1}, \bar{0}, \bar{\bar{1}}\}, N_1, N_2, N_3, 3, G)$, with the following communication graph G:

$$\begin{tikzcd} N_3 & N_1 \arrow[l] \arrow[r, bend left] & N_2 \arrow[l, bend left] \end{tikzcd}$$

The main idea behind the construction is to have processor N_1 mark the symbols that should be deleted with double bars, and the symbols that should be added with a single bar. Since unary encoding is used, adding and deleting symbols permits rewriting both the state and the values of registers using the same set of rules.

The processors of $\Gamma_{\mathcal{M}}$ are defined as follows:

$$\begin{aligned}
N_1 : IF_1 &= \{0,1\}^*,\\
M_1 &= \{\lambda \to \bar{1}, 1 \to \bar{\bar{1}}\},\\
OF_1 &= \{\bar{1}^j\bar{\bar{1}}^i0(1^*0)^{t-1}1^*\bar{1}0(1^*0)^{k-t} \mid (q_i, A(t), q_j) \in P\}\\
&\cup \{\bar{1}^j\bar{\bar{1}}^i0(1^*0)^{t-1}1^*\bar{\bar{1}}0(1^*0)^{k-t} \mid (q_i, S(t), q_j, q_s) \in P\}\\
&\cup \{\bar{1}^s\bar{\bar{1}}^i0(1^*0)^{t-1}0(1^*0)^{k-t} \mid (q_i, S(t), q_j, q_s) \in P\},\\
A_1 &= \varnothing;
\end{aligned}$$

$$\begin{aligned}
N_2 : IF_2 &= \{0,1,\bar{1},\bar{\bar{1}}\}^*,\\
M_2 &= \{\bar{1} \to 1, \bar{\bar{1}} \to \lambda\},\\
OF_2 &= \{(1^*0)^{k+1}\},\\
A_2 &= \varnothing;
\end{aligned}$$

$$\begin{aligned} N_3 : IF_3 &= \{01^*00^{k-1}\}, \\ M_3 &= \varnothing, \\ OF_3 &= \varnothing, \\ A_3 &= \varnothing. \end{aligned}$$

The NEP $\Gamma_{\mathcal{M}}$ simulates the register machine $\mathcal{M}$ in the following way. Suppose that $\mathcal{M}$ is in configuration $(q_i, n_1, \ldots, n_t, \ldots, n_k)$ and there exists an instruction $(q_i, A(t), q_j) \in P$. As mentioned above, this configuration will be represented by the string $1^i01^{n_1}0 \ldots 1^{n_t}0 \ldots 1^{n_k}0$ in processor N_1. The rules from M_1 can now be repeatedly applied, yielding strings over $\{0, 1, \bar{1}, \bar{\bar{1}}\}^*$. However, only a string of the form $\bar{1}^j\bar{\bar{1}}^i01^{n_1}0 \ldots 1^{n_t}\bar{1}0 \ldots 1^{n_k}0$ can pass the filters OF_1 and IF_2 and go to processor N_2. We remind that the marking of this string instructs to add j symbols and delete i symbols in the first block of 1's, thus referring to the transition from q_i to q_j, and to add an additional symbol 1 to the $(t+1)$-th block of 1's, which corresponds to the increment of the register R_t. Now, in order to pass OF_2, all symbols $\bar{\bar{1}}$ should be erased and all instances of $\bar{1}$ rewritten to 1. The only result of this transformation that can pass through OF_2 and IF_1 is the string $1^j01^{n_1}0 \ldots 1^{n_t+1}0 \ldots 1^{n_k}0$ which corresponds to the configuration $(q_j, n_1, \ldots, n_t + 1, \ldots, n_k)$.

The simulation of a decrement instruction of $\mathcal{M}$ is done in a similar way: the new state is marked and the corresponding register is decremented by marking one of its 1's with a double bar. If there are no instances of 1 in the corresponding block, the state corresponding to the empty register is selected.

Remark that the number of rules in our construction does not depend on the number of instructions in the register machine. This allows us to formulate the following universality result.

Theorem 4.5.1. *There exist a weakly universal NEP with 4 rules and a strongly universal NEP with 5 rules.*

Proof. The proof follows from the fact that there exist strongly universal register machines, e.g. U_{22} from [71]. Hence, $\Gamma_{U_{22}}$ constructed as above will be weakly universal because for any $\mathcal{M}$ it is true that $\Gamma_{\mathcal{M}}(x) = \mathcal{M}(x) + k + 1$. It is possible to achieve strong universality by adding a new output processor connected to N_3, a rule $0 \to \lambda$ to M_3, and a filter 1^* to OF_3. This rule and the corresponding filter will remove symbols 0 and only the value of the first register will be kept. $\square$

4.5.3 Simulation of Turing Machines

Consider the deterministic Turing machine $\mathcal{T} = (Q, \Sigma, a_1, q_1, F, \delta)$. We remind that, if $\mathcal{T}$ halts with input w, then $\mathcal{T}(w)$ is the contents of the tape in the halting configuration (cf. Section 2.2). In this subsection we will construct a NEP $\Gamma_{\mathcal{T}}$ capable of simulating a given Turing machine $\mathcal{T}$: $\Gamma_{\mathcal{T}}$ will halt on input w if and only if $\mathcal{T}$ halts, and moreover, $\mathcal{T}(w) = h'(\Gamma_{\mathcal{T}}(f'(w)))$, where f' and h' are the coding and the decoding functions correspondingly.

Let $\Sigma = \{a_1, \ldots, a_m\}$ and consider the following (unary) coding: $\phi(a_k) = 1^k0$, $k \geq 0$, so the empty symbol is coded as 10. We remark that this coding can be extended to words in a standard way. Similarly, consider the coding $\psi(q_i) = 0^i$ for $Q = \{q_1, \ldots, q_n\}$. We represent a configuration $w'q_ia_kw''$ of $\mathcal{T}$ in the following way: $\phi(w')\psi(q_i)\phi(a_k)\phi(w'')$. Accordingly, we represent the initial configuration of $\mathcal{T}$ as follows: $10\,0\,\phi(w)$, where w is the input of $\mathcal{T}$ and the second symbol 0 is the code of the initial state q_1 of $\mathcal{T}$. If w is empty, it is replaced by an instance of a_1 to assure that there is at least one symbol to the left and to the right of the head.

The function coding the input is defined as $f'(x) = \phi(a_1)\psi(q_1)\phi(x)$ and the function decoding the output as $h'(x) = \pi(z^{-1}(x))$, where $z(x) = \psi(x)$ if $x \in Q$ and $z(x) = \phi(x)$ otherwise, while $\pi(x) = \lambda$ if $x \in Q$ and $\pi(x) = x$ if $x \notin Q$.

We construct the NEP $\Gamma_{\mathcal{T}} = (\{0, 1, \bar{1}, \bar{0}, \bar{\bar{1}}\}, N_1, N_2, N_3, N_4, 4, G)$ with the following communication graph G:

$$\begin{tikzcd}
N_1 \arrow[r] & N_2 \arrow[r] & N_3 \arrow[r] \arrow[ll, bend left] & N_4
\end{tikzcd}$$

The main idea is essentially similar to what we have already shown in the previous subsection: processor N_1 inserts $\bar{1}$'s to mark additions of 1's to the string and rewrites certain 1's into $\bar{\bar{1}}$'s to schedule them for subsequent deletion. It does so in correspondence with the program of the simulated Turing machine.

The processors of $\Gamma_{\mathcal{T}}$ are defined as follows:

$$\begin{aligned}
N_1 : IF_1 &= \{(1^+0)^+0^i(1^+0)^+ \mid q_i \notin F\},\\
M_1 &= \{\lambda \to \bar{1}, 1 \to \bar{\bar{1}}, \lambda \to \bar{0}\},\\
OF_1 &= \{(1^+0)^+\ 0^i\ 1^k\bar{1}^{l-k}0\ \bar{0}^j\ (1^+0)^+ \mid (q_i, a_k, q_j, a_l, R) \in \delta, l \geq k\}\\
&\cup \{(1^+0)^+\ 0^i\ 1^k\bar{1}^{l-k}0\ \bar{0}^j\ \bar{1}\bar{0} \mid (q_i, a_k, q_j, a_l, R) \in \delta, l \geq k\}\\
&\cup \{(1^+0)^+\ 0^i\ 1^l\bar{\bar{1}}^{k-l}0\ \bar{0}^j\ (1^+0)^+ \mid (q_i, a_k, q_j, a_l, R) \in \delta, l < k\}\\
&\cup \{(1^+0)^+\ 0^i\ 1^l\bar{\bar{1}}^{k-l}0\ \bar{0}^j\ \bar{1}\bar{0} \mid (q_i, a_k, q_j, a_l, R) \in \delta, l < k\}\\
&\cup \{(1^+0)^+\ \bar{0}^j\,(1^+0)\ 0^i\ 1^k\bar{1}^{l-k}0\ (1^+0)^* \mid (q_i, a_k, q_j, a_l, L) \in \delta, l \geq k\}\\
&\cup \{\bar{1}\bar{0}\ \bar{0}^j\ (1^+0)\ 0^i\ 1^k\bar{1}^{l-k}0\ (1^+0)^* \mid (q_i, a_k, q_j, a_l, L) \in \delta, l \geq k\}\\
&\cup \{(1^+0)^+\ \bar{0}^j(1^+0)0^i\ 1^l\bar{\bar{1}}^{k-l}0\ (1^+0)^* \mid (q_i, a_k, q_j, a_l, L) \in \delta, l < k\}\\
&\cup \{\bar{1}\bar{0}\ \bar{0}^j\ (1^+0)\ 0^i1^l\ \bar{\bar{1}}^{k-l}\ 0(1^+0)^* \mid (q_i, a_k, q_j, a_l, L) \in \delta, l < k\},\\
A_1 &= \varnothing;\\[1ex]
N_2 : IF_2 &= \{0, 1, \bar{0}, \bar{1}, \bar{\bar{1}}\}^*,\\
M_2 &= \{0 \to \bar{\bar{1}}\},\\
OF_2 &= \{(1^+0)^+\ \bar{\bar{1}}^+\ 1^+(\bar{1}^*|\bar{\bar{1}}^*)0\ \bar{0}^+\ \big((1^+0)^+|\bar{1}\bar{0}\big)\}\\
&\cup \{\big((1^+0)^+|\bar{1}\bar{0}\big)\ \bar{0}^+\ (1^+0)\ \bar{\bar{1}}^+\ 1^+(\bar{1}^*|\bar{\bar{1}}^*)0\ (1^+0)^+\},\\
A_2 &= \varnothing;\\[1ex]
N_3 : IF_3 &= \{0, 1, \bar{0}, \bar{1}, \bar{\bar{1}}\}^*,\\
M_3 &= \{\bar{1} \to 1, \bar{\bar{1}} \to \lambda, \bar{0} \to 0\},\\
OF_3 &= \{0, 1\}^*,\\
A_3 &= \varnothing;\\[1ex]
N_4 : IF_4 &= \{(1^+0)^+0^i(1^+0)^+ \mid q_i \in F\},\\
M_4 &= \varnothing,\\
OF_4 &= \varnothing,\\
A_4 &= \varnothing.
\end{aligned}$$

We will now describe how the network $\Gamma_{\mathcal{T}}$ simulates the Turing machine $\mathcal{T}$. We recall that the result of the computation of $\Gamma_{\mathcal{T}}$ is the (only) string that reaches processor N_4. To rewrite the substring $\phi(a_k)$ into $\phi(a_l)$, we either need to add some 1's to the block of 1's coding a_k if $l \geq k$, or to erase some 1's otherwise. This exactly is the mission of N_1, which schedules the operations to carry out and also inserts a number of $\bar{0}$'s to select the next state of $\mathcal{T}$.

The output filter OF_1 exercises very precise control over which strings are eventually allowed out of N_1. The first line in the definition of OF_1 allows strings corresponding to a transition of $\mathcal{T}$ during which the head moves to the right, a_k is rewritten into a_l, $l \geq k$, and the sequences of 0's and $\bar{0}$'s code the corresponding old state q_i and the new state q_j. The second line in the definition of OF_1 allows N_1 to deal with the situations in which a_k is rewritten into a_l, $l \geq k$, and the head is at the right end of the string: a new empty cell 10 is scheduled for insertion by placing $\bar{1}\bar{0}$ at the right end. The next two lines in the definition of OF_1 have exactly the same role as the first two, but for the case in which $l < k$. Note how the first two lines expect additions of 1's by requiring the presence of $\bar{1}$'s, while the following two lines expect removals of 1's by requiring the presence $\bar{\bar{1}}$'s. The next four lines handle the same special cases for leftward head moves.

The task of N_2 is to mark the previous state for deletion. This is done by replacing the corresponding sequence of 0's by a sequence of $\bar{\bar{1}}$'s. We remark that due to the form of the strings that can reach N_2 (the ones which pass the filter OF_1), only those strings will be allowed out of N_2 in which all the 0's representing the old state have been rewritten.

The role of N_3 is to apply the operations scheduled by N_1 and N_2 by inserting barred symbols and erasing double-barred 1's. N_3 forwards its output to both N_1 and N_4, but only one of these two processors will accept the string, since N_4 only lets in strings representing a configuration of $\mathcal{T}$ in one of its final states, while N_1 will only work on strings which contain the code of a non-final state.

Applying our construction to simulate a concrete universal Turing machine yields the following universality result.

Theorem 4.5.2. *There exists a universal NEP with 7 rules that simulates any Turing machine in polynomial time.*

Proof. The proof follows from the existence of universal Turing machines that simulate the target machine in polynomial time, see [120] for more details. □

For completeness, we remind what polynomial-time simulation means. Consider two computing devices A and B and denote the number of steps A (respectively, B) needs to produce output on input w by $t_A(w)$ (respectively, by $t_B(w)$). Then "A simulates B in polynomial time" means that, for a given input w, A produces the same output as B, and moreover, for any w, $t_A(w)$ is bounded by a polynomial function of $t_B(w)$.

We also remark that, in $\Gamma_{\mathcal{T}}$, the simulation of each step of a Turing machine happens in a fixed number of steps depending on the codes of the implied symbols.

Finally, we remark that, while the universal Turing machines we simulate do not rely on commands which only rewrite symbols without moving the head, it is rather easy to apply the ideas we have just shown to simulate such instructions.

Chapter 5

Small Universal Register Machines

In this chapter we explore the questions of universality for register machines. We recall that an element $A_0 \in \mathfrak{C}$, for a class of computing devices $\mathfrak{C}$, is called weakly universal if $A(x) = y$ implies $A(x) = f(A_0(g(A), h(x)))$, where h and f are the encoding and decoding functions respectively, and g is the function assigning numbers to devices in $\mathfrak{C}$, according to some fixed enumeration. If h and f are identities, the element A_0 is called strongly universal. We refer to Section 2.3 for a detailed discussion of computational completeness and universality.

In Section 5.1 of the present chapter, we introduce a generalisation of the classical model of register machines allowing several tests and operations to be carried out at a single time, after which we describe a universal generalised register machine with seven states only. In Section 5.2 we recall Marvin Minsky's approach to building strongly and weakly universal register machines with 3 and 2 registers respectively [92], and then we actually construct and give the instructions of these machines.

5.1 Generalised Register Machines

5.1.1 Definition and Motivation

In this section we discuss an extension of the construct of a register machine which originates in the fundamental similarities of this computing model with multiset rewriting systems and Petri nets, discussed in more detail in Section 1.3. Conventional register machines are based on a very restricted set of instructions, which simplifies reasoning about them and gives a detailed insight into the complexity of such operations as multiplication and division. However, a reduced instruction set is more of a nuisance in the pursuit of minimal universality constructions in the domains of Petri nets and multiset rewriting systems, which are traditionally shown to simulate one of the universal register machines from Ivan Korec's work [71] (e.g., [6, 24]). Indeed, several operations of a conventional register machine can be carried out in a single evolution step of such a system. Consider multiset rewriting, for example; if the value of the register R_i is represented by the multiplicity of the symbol r_i, the multiset rewriting rule $w \to v\, r_1 r_2 r_3^2$ increments the registers R_1 and R_2, and adds 2 to R_3 all at the same time, while a basic register machine would require 4 separate instructions to achieve the same result.

This disparity between the complexity of an instruction of a register machine and a multiset rewriting rule opens up several possibilities of optimising the simulation of the former by the latter (e.g., [6, 40]). Often these optimisation ideas are not specific to multiset rewriting and can be easily carried over to different computing devices, like, for example, Petri nets. The generalised construction introduced in this section is meant to capture the versatility of such systems in a form closely related to original register machines, so that some important optimisations can be performed on this abstract model and then directly taken over to other models of computing. Essentially, we consider devices with registers, which can modify or test several registers for zero in a single state transition.

A *generalised register machine* is a construct $M = (R, G, q_0, F, z, nz, add, sub)$, where $R = \{1, \dots, k\}$ is a set of register numbers, $G = (Q, E, s, t)$ is a directed multigraph of states Q and arcs (transitions) E, $q_0 \in Q$ is the initial state, $F \subseteq Q$ comprises the final states, while $z, nz : E \to 2^R$, $add : E \to R^*$, and $sub : E \to 2^R$ are functions defining the registers which should be zero or non-zero for a transition to be activated, as well as the registers which should be decremented or incremented during the transition, respectively. We require that, for any $e \in E$, $z(e) \cap nz(e) = \varnothing$, and that $sub(e) \subseteq nz(e)$, i.e. only those registers which are required to be non-zero are decremented. Remark that $add(e)$ is a multiset, which means that multiple increments of the same register can be carried out in a single transition.

A configuration of a generalised register machine is the tuple $(q, n_1, \dots, n_k)$, where, just as in the case of conventional register machines, $q \in Q$ is the current state of M, and $n_i \in \mathbb{N}$, $1 \le i \le k$, are the values of the registers. We say that there is a transition from configuration $C = (q, n_1, \dots, n_k)$ to $C' = (q', n'_1, \dots, n'_k)$ if G contains an edge e such that $s(e) = q$, $t(e) = q'$, the conditions represented by $z(e)$ and $nz(e)$ hold in C:

$$\forall i \in z(e) \,.\, n_i = 0 \quad \text{and} \quad \forall i \in nz(e) \,.\, n_i \neq 0,$$

and the values of registers in C' can be obtained by applying the instructions prescribed by $s_e = sub(e)$ and $a_e = add(e)$:

$$\forall i \in R \,.\, n'_i = n_i - s_e(i) + a_e(i),$$

where $x(i)$ is the number of occurrences of i in the multiset x, and the set s_e is treated as a multiset. The arc e is said to be enabled in configuration C.

A graphical representation of a generalised register machine is shown in Figure 5.1. This machine has two registers numbered 1 and 2, and three states: q_0, q_1, and q_2, the latter state being final: $F = \{q_2\}$. This register machine has four arcs: $E = \{e_1, e_2, e_3, e_4\}$ with the following source and target vertices:

$$\begin{aligned} &s(e_1) = q_0, \ t(e_1) = q_1, \qquad && s(e_2) = (q_0), \ t(e_2) = q_1, \\ &s(e_3) = q_1, \ t(e_3) = q_2, && s(e_4) = (q_2), \ t(e_4) = q_0. \end{aligned}$$

The conditions and operations associated with these edges are the following:

$$\begin{aligned} &z(e_1) = \varnothing, && nz(e_1) = \{1, 2\}, && add(e_1) = \lambda, && sub(e_1) = \{1, 2\}, \\ &z(e_2) = \varnothing, && nz(e_2) = \varnothing, && add(e_2) = 2, && sub(e_2) = \varnothing, \\ &z(e_3) = \{1\}, && nz(e_3) = \varnothing, && add(e_3) = 1, && sub(e_3) = \varnothing, \\ &z(e_4) = \varnothing, && nz(e_4) = \{2\}, && add(e_4) = \lambda, && sub(e_4) = \varnothing. \end{aligned}$$

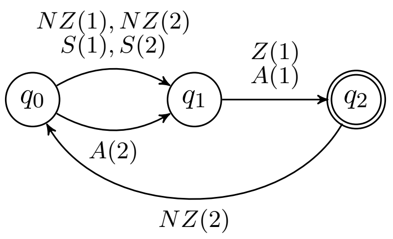


Figure 5.1: An example of a generalised register machine

In the graphical representation of the label of an arc e of a generalised register machine, we use the symbol $Z(i)$ if $i \in z(e)$, $NZ(i)$ if $i \in nz(e)$, $S(i)$ if $i \in sub(e)$, and $A^j(i)$ if the multiplicity of the register number i in the multiset $add(e)$ is j. If $j = 1$, we will just write $A(i)$.

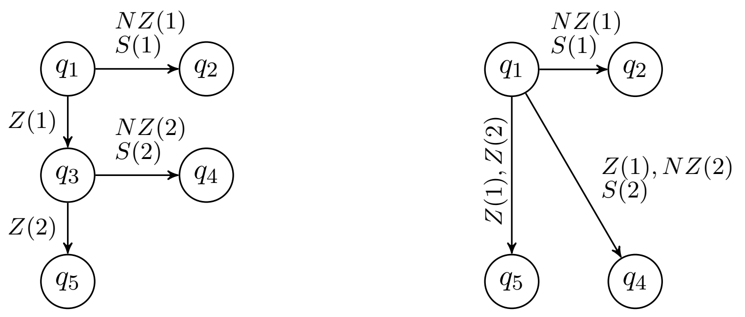
Remark that, if the set of arcs E of a generalised register machine contains two distinct elements e_1 and e_2 such that $z(e_1) = z(e_2)$, $nz(e_1) = nz(e_2)$, $add(e_1) = add(e_2)$, and $sub(e_1) = sub(e_2)$, then these arcs have the same effect and we can safely identify them and only consider machines without such pairs of arcs. This means that an arc is uniquely identified by the values the functions s, t, z, nz, add, and sub assign to it. We will therefore omit specifying arc names in most of the cases. Often we will also use the notation $e : q_1 \to q_2$ when $s(e) = q_1$ and $t(e) = q_2$.

A generalised register machine is not required to be deterministic, and indeed, the machine shown in Figure 5.1 can choose between two different evolution paths in configuration $(q_0, 1, 2)$. We will say that a generalised register machine is *deterministic* if, in every configuration $C = (q, n_1, \dots, n_k)$, $q \in Q \setminus F$, there is precisely one enabled arc, and in any configuration $C = (q_f, n_1, \dots, n_k)$, $q_f \in F$, no arcs are enabled.

It follows from the definition that any register machine based on the increment and decrement-and-zero-check operations can be directly transformed into a generalised register machine; in particular, U_{22} and U_{20} from [71] can be directly seen as generalised register machines. Some more care should be taken in case the pure decrement instruction is used without a prior zero test, because in [71, Section 1] this instruction is defined to not modify the contents of the register if it is already empty. This corresponds to a decrement-and-zero-check instruction $(p, S(i), q, q)$, which moves the machine into state q no matter what the value of the register was, and should thus be simulated using two arcs of a generalised register machine having complementary sets of conditions on the register i.

5.1.2 State Compression and Universality

As we have seen in the previous subsection, generalised register machines are capable of performing several register tests and modifications at a single time. This property can be used to reduce the number of states at the expense of having more complex transitions. Consider for example the generalized register machine shown in Figure 5.2a; it could be described with the instructions $(q_1, S(1), q_2, q_3)$ and $(q_2, S(j), q_4, q_5)$. The generalized register machine in Figure 5.2b performs the same operations, but uses one state less. The idea is that, instead of checking that register 1 is zero on the transition from q_1 to q_3, and then checking that register 2



(a) Two decrements (b) Compressed version

Figure 5.2: State compression for successive $S(1)$ and $S(2)$.

is non-zero and decrementing it while moving to state q_4, the register machine can do all of these checks and operations in a single direct transition from q_1 to q_4. We will refer to the procedure of removing states while conserving the same behaviour as *state compression.*

While many states can be reduced away in this fashion, not all of them can. To formally define the circumstances under which a state may be removed, consider a state q, the set of edges which end in q: $pred(q) = \{e \in E \mid t(e) = q\}$, and another set of edges which start at q: $succ(q) = \{e \in E \mid s(e) = q\}$. Then q is compressible if all of the following conditions hold:

– no arc going into q modifies a register which is involved in the conditions of an arc leaving q, that is, for every $e_{pred} \in pred(q)$ and $e_{succ} \in succ(q)$,

$$\big(supp(add(e_{pred})) \cup sub(e_{pred})\big) \cap \big(z(e_{succ}) \cup nz(e_{succ})\big) = \varnothing,$$

where $supp(w)$ is the support of the multiset w (see Section 2.1);

– q has no loop arcs (i.e. arcs which do not move the machine away from state q):

$$\nexists e \in E \,.\, s(e) = t(e) = q.$$

Note that, since all registers which are decremented by an arc are required to be checked for being non-zero by the same arc, the former condition implies that a state, for which an incoming arc decrements a register and an outgoing arc decrements the same register, is not compressible.

To compress away a state q, we take all pairs of edges $e_{pred} \in pred(q)$ and $e_{succ} \in succ(q)$ and pick the pairs which do not check for conflicting conditions, that is:

$$z(e_{pred}) \cap nz(e_{succ}) = nz(e_{pred}) \cap z(e_{succ}) = \varnothing.$$

Then, for every such pair we add a new arc $e : q_{pred} \to q_{succ}$, putting together all the conditions and operations of the arcs e_{pred} and e_{succ}:

$$\begin{aligned} z(e) &= z(e_{pred}) \cup z(e_{succ}), & nz(e) &= nz(e_{pred}) \cup nz(e_{succ}),\\ add(e) &= add(e_{pred}) + add(e_{succ}), & sub(e) &= sub(e_{pred}) \cup sub(e_{succ}). \end{aligned}$$

The state q and all arcs having it as source or target are then removed.

The *state compression* algorithm is defined as iterative reduction of compressible states until no more such states remain in the generalised register machine. Using this algorithm, it is possible to compress the original register machine U_{22} to a generalized register machine with 7 states, including a *Stop* state. We will refer to this generalized machine as U_7; its program is shown in Table 5.1. Similarly, the register machine U_{20} can be compressed to a weakly universal generalised register machine with 7 states, which we will call U'_7; the program of this machine is given in the appendix. Both machines start in state q_1 with input in register 2 (corresponding to R_2 in U_{22} and U_{20}), and place the result into register 0 (corresponding to R_0 in U_{22} and U_{20}).

Table 5.1: The program of the universal generalised register machine U_7

q_i	q_j	Conditions	Operations
q_1	q_1	NZ(1)	S(1), A(7)
q_1	q_4	Z(1)	A(6)
q_4	q_4	Z(5), Z(6)	
q_4	q_4	NZ(5)	S(5), A(6)
q_4	q_{10}	Z(5), NZ(6)	A(5), S(6)
q_{10}	q_1	Z(6), Z(7)	
q_{10}	q_1	NZ(4), NZ(6), Z(7)	S(4)
q_{10}	q_4	Z(6), NZ(7)	A(1), S(7)
q_{10}	q_{10}	NZ(6), NZ(7)	A(1), A(5), S(6), S(7)
q_{10}	q_{16}	Z(4), NZ(6), Z(7)	
q_{16}	q_1	NZ(0), Z(2), Z(5)	S(0)
q_{16}	q_1	NZ(2), NZ(4), Z(5)	S(2), S(4)
q_{16}	q_1	Z(0), Z(2), NZ(4), Z(5)	S(4)
q_{16}	q_{18}	NZ(5)	S(5)
q_{16}	*Stop*	Z(0), Z(2), Z(4), Z(5)	
q_{16}	*Stop*	NZ(2), Z(4), Z(5)	S(2)
q_{18}	q_1	Z(3), Z(5)	A(0)
q_{18}	q_1	NZ(3), NZ(4), Z(5)	S(3), S(4)
q_{18}	q_{20}	NZ(5)	S(5)
q_{18}	*Stop*	NZ(3), Z(4), Z(5)	S(3)
q_{20}	q_1	NZ(4), Z(5)	A(2), A(3), S(4)
q_{20}	q_{16}	NZ(5)	A(4), S(5)
q_{20}	*Stop*	Z(4), Z(5)	A(2), A(3)

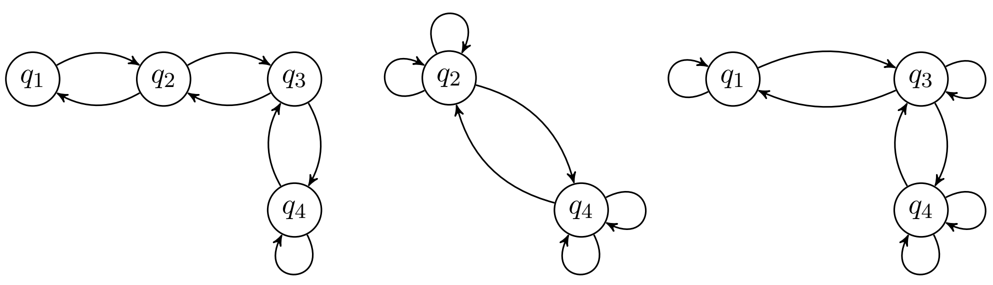
The first three states of the machine U_7 result from the compression of the instruction block of U_{22} shown in Figure 2.1: state q_1 of U_7 corresponds to the cycle involving q_1 and q_3 in U_{22}, state q_4 of U_7 corresponds to the cycle involving q_4 and q_6 in U_{22}, and state q_{10} of U_7 corresponds to the cycle involving the states q_7, q_9, q_{10}, and q_{12} of U_{22}. Clearly, states q_1, q_4, and q_{10} of U_7 cannot be further compressed, because they have self-loop arcs.

The other three non-final states of U_7 correspond to the three decrement operations of the decoder block of U_{22}. The only compressible state in this block is q_{22}; the other three successively decrement the same register, R_5, and thus cannot be reduced.

Finally, the states of the simulation block of U_{22} are completely compressed into the three states of the decoder. This is possible because the simulation block contains no loops and does not interact in any way with R_5.

We remark that, even though state compression always produces equivalent generalised register machines, the state compression algorithm is not confluent: starting with the same machine, structurally different irreducible machines can be obtained. Moreover, even the number of the states in the resulting machines can vary, as illustrated in the following example.

Example 5.1.1. *Consider a generalised register machine whose multigraph has the form shown in Figure 5.3a. Suppose that the conditions and operations assigned to the arcs are so that q_1, q_2, and q_3 are compressible. If we start by compressing the state q_1 first, and then q_3, we reach the two-state irreducible graph shown in Figure 5.3b. If, however, we decide to compress q_2 first, we will have to add loops to q_1 and q_3, thereby rendering the two states incompressible, as shown in Figure 5.3c.*



(a) Original multigraph (b) Compressed q_1 and q_3 (c) Compressed q_2

Figure 5.3: Two irreducible multigraphs with a different number of states resulting from compressing of different states in the same original multigraph

5.2 Universal 2- and 3-Register Machines

In this section we recall that there exist weakly universal register machines with two registers only, and that strong universality for unary functions can be achieved by adding a third input/output register. This result was proved by Marvin Minsky in [92], a work which also gives the actual algorithm for simulating a register machine with any number of registers using only two (respectively, three) registers. In this section we will apply this algorithm to universal register machines constructed by Ivan Korec [71] in order to give a concrete description of 2- and 3-register machines. We remark that, even though Minsky's approach has been known for a long time, we could not find concrete constructions presented anywhere.

Consider a register machine $M = (Q, R, q_0, q_f, P)$. Minsky's construction is based on exponential coding of the values of all registers in R as one number. The values of the k registers in a configuration of M given by the vector $(n_1, \ldots, n_k)$ can be represented as the number $\rho = p_1^{n_1} \times \ldots \times p_k^{n_k}$, where $p_1, \ldots, p_k$ are different prime numbers. Then, incrementing the register R_i in this coding corresponds to multiplying ρ by p_i, while decrementing the same register corresponds to dividing ρ by p_i.

Supposing that the value ρ is stored in register R_0 of a 2-register machine M_2, multiplication of ρ by p_i can be performed by repeatedly decrementing R_0 and adding p_i to register R_1 of M_2. Division can be carried out symmetrically: by repeatedly subtracting p_i from R_0 and incrementing R_1 whenever the subtraction was possible. If, after a certain number of subtractions of p_i, the value of R_0 is zero, we know that ρ was divisible by p_i and that the value stored in R_1 is ρ/p_i. If, on the other hand, the value of R_0 is non-zero, but less than p_i, we conclude that ρ was not divisible by p_i, and we have to restore the original value ρ. Since R_1 contains the value $\lfloor \rho/p_i \rfloor$ at this point, M_2 can rebuild ρ by adding to R_0 the result of multiplication of R_1

by p_i. For more details on this simulation approach we refer the reader to Minsky's work [92].

To achieve strong universality for unary functions, a third input/output register is required to carry out exponentiation and logarithm. Suppose that the input register of the simulated machine M is R_{in}. Then, in order to be able to use exponential encoding, a 3-register machine M_3 would have to repeatedly multiply the value in register R_0 by the corresponding prime number p_{in}, and to decrement its own input/output register R_2 after each iteration. When R_2 is zero, M_3 is sure to have raised p_{in} to the power of the original input value from R_2. In a similar fashion, to obtain the value of the output register R_{out} from the exponential encoding of a halting configuration of M, M_3 would have to perform repeated divisions of R_0 by p_i, incrementing R_2 after each successful iteration.

The strategy for achieving weak universality with two registers can be applied directly to constructing the weakly universal 2-register machine U_2 which simulates Korec's weakly universal U_{20}. This 2-register machine has 112 decrement and 165 increment instructions: 278 states all in all (including a final state). Simulating Korec's strongly universal U_{22} using two registers, and further adding the coding of input and the decoding of output, allows building the strongly universal 3-register machine U_3 with 146 decrement and 221 increment instructions: 368 states in total. Both register machines use register R_0 to store the exponentially-coded values of the simulated machine, and register R_1 to keep the intermediate results. The input of U_2 should thus be provided in coded form in R_0. The register machine U_3, on the other hand, reads its input from and writes its output to the third register, R_2. Full programs of U_2 and U_3 are provided in the appendix.

An important remark with regard to the strong universality of U_3 is due here: since this machine uses one register for input, it is only able to directly simulate unary partial recursive functions. Nevertheless, Section 9 of [71] describes a way to construct register machines simulating n-ary partial recursive functions; the machines use a coding to store the values of the n arguments in one of the working registers. This approach can be naturally adapted to the register machine U_2 to obtain strongly universal register machines with n input registers, read by successive decrements at the start of the computation, and which only have two working registers.

We remark that Section 2 of [70] describes the construction of a universal 3-register machine with 130 states. However in this paper compound instructions are assigned to single states (e.g. any increment of a register by m is treated as a single instruction). Writing out such instructions in terms of elementary register machine commands as we use in U_2 and U_3, would yield more than 450 instructions.

Chapter 6

Small Universal Petri Nets

In this chapter we discuss the problem of universality for Petri nets with inhibitor arcs seen as computing devices, and give several strongly and weakly universal constructions. We define the size of a Petri net as a tuple comprising the number of places, transitions, inhibitor arcs, and the maximal transition degree, and then describe techniques for minimising each of these parameters, which sometimes allow attaining the theoretical minimum required for universality. The sizes of constructed universal Petri nets are given in Table 6.1 on page 110.

We recall that an element $A_0 \in \mathfrak{C}$, for a class of computing devices $\mathfrak{C}$, is called weakly universal if $A(x) = y$ implies $A(x) = f(A_0(g(A), h(x)))$, where h and f are the encoding and decoding functions respectively, and g is the function assigning numbers to devices in $\mathfrak{C}$, according to some fixed enumeration. If h and f are identities, the element A_0 is called strongly universal. We refer to Section 2.3 for a detailed discussion of computational completeness and universality.

The universality of the Petri nets we construct in this chapter is an indirect corollary of the universality of register machines and the fact that, for any Petri net N, it is possible to construct a register machine simulating it by representing each place of N as a register and reproducing token dynamics by incrementing and decrementing the corresponding registers. This means that if we construct a Petri which simulates a universal register machine, this net will be capable of (indirectly) simulating any other Petri net.

6.1 Definitions

A *Petri net with inhibitor arcs* is a construct $N = (\mathcal{P}, \mathcal{T}, W, M_0)$ where $\mathcal{P}$ is a finite set of *places*, $\mathcal{T}$ is a finite set of *transitions*, with $\mathcal{P} \cap \mathcal{T} = \varnothing$, $W : (\mathcal{P} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{P}) \to \mathbb{N} \cup \{-1\}$ is the weight function, and M_0 is a multiset over $\mathcal{P}$ called the initial marking.

Petri nets are usually represented by diagrams in which places are drawn as circles, transitions as squares annotated with their locations, and a directed arc (X, Y) is added between X and Y if $W(X, Y) \geq 1$. The weight of the arc will be explicitly written if it is 2 or more. Arcs of weight -1 are called *inhibitor* arcs and will be drawn with a small circle on the side of the transition. Figure 1.3 in Chapter 1 shows an example of such a representation.

The *degree* of a transition T is defined as the sum of the weights of the incoming and outgoing arcs involved with it plus the number of inhibitor arcs:

$$deg(T) = \sum_{P \in \mathcal{P}} \left|W(P,T)\right| + \left|W(T,P)\right|.$$

Note that the degree is not the number of weighted arcs adjacent to the transition, but rather the number of single arcs they represent.

Given a Petri net N, the pre- and post-multiset of a transition T are respectively the multiset $pre_{N(T)}$ and the multiset $post_{N(T)}$ such that, for all $P \in \mathcal{P}$, for which $W(P,T) \geq 0$, $pre_{N(T)}(P) = W(P,T)$ and $post_{N(T)}(P) = W(T,P)$. A configuration of N, which is called a *marking*, is a multiset M over $\mathcal{P}$; in particular, for every $P \in \mathcal{P}$, $M(P)$ represents the number of tokens inside place P. A transition T is enabled at a marking M if the multiset $pre_{N(T)}$ is contained in the multiset M and all inhibitor places P (such that $W(P,T) = -1$) are empty. A transition T enabled at marking M can fire and produce a new marking M' such that $M' = M - pre_{N(T)} + post_{N(T)}$ (i.e. for every place $P \in \mathcal{P}$, the firing transition T consumes $pre_{N(T)}(P)$ tokens and produces $post_{N(T)}(P)$ tokens). We denote this by $M \xrightarrow{T} M'$.

For the purposes of this work, we define a special subtype of Petri nets which can execute computations (e.g. compute partially recursive functions). In such a net some distinguished places $I_1, \ldots, I_k$ from $\mathcal{P}$ will be called input places and another one, $I_0 \in \mathcal{P}$, will be called the output place. The computation of the net N on the input vector $(n_1, \ldots, n_k)$ starts with the initial marking M'_0 such that $M'_0(I_j) = n_j$, and $M'_0(x) = M_0(x)$, for all $x \neq I_j$, $1 \leq j \leq k$. This net will evolve by firing transitions until deadlock occurs in some marking M_f, i.e. until no transition is enabled in M_f. Thus we have $M'_0 \rightarrow^* M_f$ and there are no M'_f and $T \in \mathcal{T}$ such that $M_f \xrightarrow{T} M'_f$. The result of the computation of N on the vector $(n_1, \ldots, n_k)$, denoted by $\Phi^k_N(n_1, \ldots, n_k)$, is defined as $M_f(I_0)$, i.e. the number of tokens in place I_0 in the final state. Since in the general case Petri nets are non-deterministic, the function Φ^k_N may be considered to compute sets of numbers.

We remark that, with this definition of computation for Petri nets, a parallel with multiset rewriting is directly established. Indeed, transitions are easily represented as multiset rewriting rules sequentially acting on markings of the net until no more rules are applicable.

If, for any reachable marking M of a Petri net N, there is at most one transition T and one marking M' such that $M \xrightarrow{T} M'$, the Petri net is called *deterministic*. This corresponds to labeled deterministic Petri nets in which all transitions are labeled with the same symbol [96]. Otherwise the Petri net is called *non-deterministic*.

We define the *size* of a Petri net to be the vector (p, t, h, d) where p is the number of places, t is the number of transitions, h is the number of inhibitor arcs, and d is the maximal degree of a transition. These parameters provide fundamental information about the structure of the net and can be further used to reason about its other features (e.g., the average number of inhibitor arcs per transition, etc.). Moreover, each of these parameters has a direct equivalent in the multiset rewriting interpretation of Petri nets as the cardinality of the alphabet, number of rules, inhibitors and maximal rule size.

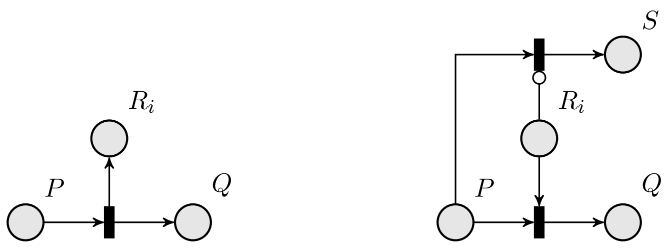
We conclude this section by remarking that the effect of inhibitor arcs can be reproduced by other computationally complete Petri net extensions, like, for example, priorities. Indeed, the fact that there is an inhibitor arc going from place Q

to transition T_1 may be represented by the fact that any other transition T_2 which consumes tokens from Q has priority over T_1. If such a transition T_2 does not exist, a dummy transition taking a token from Q and putting it back can be added and given priority over T_1.

6.2 Minimising the Transition Degree

In this section we will show how to construct a strongly and a weakly universal Petri net with inhibitor arcs, with transitions of degree of most 3, based on the universal register machines from Ivan Korec's work [71]. We will then evaluate the other complexity parameters of the obtained nets: the number of places, the number of transitions, and the number of inhibitor arcs.

We will mainly rely on the strongly universal U_{22} and the weakly universal U_{20} from [71]. Since these machines only use instructions of type increment and zero-check-and-decrement, it suffices to describe how such operations can be carried out in Petri nets. One of the simplest ideas is representing registers as places and also allocating a place per state. The nets carrying out the two types of instructions we are interested in are shown in Figure 6.1. The simulation of the increment is straightforward: the token moves from place P into place Q and adds one token to R_i along the way. The zero-check-and-decrement is simulated using an inhibitor arc connected to R_i: if R_i is not empty, the token can only move from P to Q removing a token from R_i along the way, while if R_i is empty, the token can only move to S.



(a) $(p, A(i), q)$ (b) $(p, S(i), q, s)$

Figure 6.1: Direct simulation of increment and zero-check-and-decrement in Petri nets

We can now directly construct the universal Petri net N_1 simulating the register machine U_{22} by iteratively translating all of its instructions. This Petri net has 22 places for states and 8 places for registers, 30 states all in all. There are 34 transitions in this net, as many as there are arcs in the graph representation of U_{22}. To count the number of inhibitor arcs in N_1, remark that one will be required per each decrement instruction of U_{22}, except for q_{32}, because it suffices to have the net halt with a token in Q_{32} if the place corresponding to register R_4 cannot be decremented. Correspondingly, N_1 has 12 inhibitor arcs. Finally, the maximal transition degree is 3, which means that the size of N_1 is $(30, 34, 12, 3)$.

At the initial marking N_1 will have one token in place Q_1 corresponding to state q_1 of U_{22}, the code of the machine to be simulated in place R_1, and the input value in place R_2. The output of N_1 is to be read from place R_0. Therefore in order to simulate the computation of an arbitrary Petri net N with one input place, N_1 shall be provided with the appropriate coding of N and its input in places R_1 and R_2 respectively. The net N_1 is strongly universal in the sense of the relation

$\Phi_N(x) = \Phi^2_{N_1}(g(N), x)$, where Φ_N is the function computed by the Petri net N and g is a function assigning a number to every Petri net, in some fixed enumeration (like Gödel numbering).

The same approach can be applied to simulating the weakly universal register machine U_{20}, yielding the weakly universal Petri net N_1' of size $(27, 31, 11, 3)$.

If all transitions are required to consume at least one token, nets N_1 and N_1' achieve the minimal value of the transition degree necessary for computational completeness. Indeed, such a Petri net (even with inhibitor arcs) which only has transitions of degree 2 is bounded; even more, the total count of tokens present in the net at any time can never increase. Demanding that all transitions consume at least one token corresponds to the requirement that no rule may have an empty left-hand side in multiset rewriting systems.

6.3 Minimising the Number of Transitions

Because of the similarity of the semantics of Petri net transitions and the arcs of a generalized register machine, Petri nets can be used to directly simulate the former class of computing devices.

Consider a generalized register machine $M = (R, G, q_0, F, z, nz, add, sub)$, with the underlying multigraph $G = (Q, E, s, t)$. We will construct a Petri net N with the weight function W simulating M. As before, we will represent the states Q and the registers R as places, and, for every edge $e \in E$, with $s(e) = q$ and $t(e) = q'$, we will add a Petri net transition T which will have:

– an arc coming from the state place Q (representing the state q) and an arc going into the state place Q' (representing the state q'):

$$W(Q, T) = 1, \quad W(T, Q') = 1;$$

– an arc going into each of the register places representing the registers incremented by e:

$$W(T, R_i) = a_e(i), \quad \text{for } a_e = add(e), \ i \in R,$$

– an inhibitor arc to each of the register places representing the registers which should be zero in order for e to be enabled:

$$W(R_i, T) = -1, \quad \text{for } i \in z(e),$$

– an arc coming from the register places corresponding to registers decremented by e:

$$W(R_i, T) = 1, \quad \text{for } i \in sub(e),$$

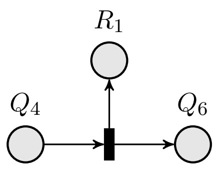
– an arc coming from the register place and an arc going into the same register place for registers which have to be non-zero for e to be enabled, but which are not decremented by e:

$$W(R_i, T) = W(R_i, T) = 1, \quad \text{for } i \in nz(e) \setminus sub(e).$$

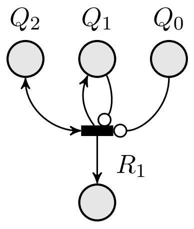
Using this approach to simulate the generalized register machine U_7 from Subsection 5.1.2, we obtain the strongly universal Petri net N_2 of size $(14, 23, 30, 6)$. By simulating the generalized register machine U'_7, we construct the weakly universal Petri net N'_2 of size $(13, 21, 23, 6)$. The initial markings of both Petri nets have one token in place Q_1, which corresponds to the initial state, the code of the simulated register machine in place R_1, and the input value (exponentially coded in the case of N'_2) in place R_2. The output of both networks will be found in the register place R_0.

Remark that, while a Petri net transition can simulate any arc of the generalised register machine, the converse is not true, because an arc of a generalised register machine cannot do more than one decrement of the same register.

We can further reduce the number of places, while keeping the number of transitions low, by coding the current state number in binary. If the simulated machine has n states, we will use $n_p = \lceil \log_2 n \rceil$ places to encode the current state number in the following way: place Q_i, $0 \leq i < n_p$, will contain a token if the i-th bit of the binary representation of n is 1, and will be empty otherwise. All transitions of such a Petri net will thus depend on all the state places Q_i, $0 \leq i < n_p$, and will produce the new marking of the state places corresponding to the next state number. An example of simulation of the increment instruction $(q_4, A(1), q_6)$ using binary coded states is shown in Figure 6.2b. The only transition of this net is enabled only if place Q_2 contains a token, and Q_1 and Q_0 are empty, which corresponds to the number $(100)_2 = 4$. When this transition fires, it adds a token to R_1, replaces the consumed token in Q_2, adds a token to Q_1, and leaves Q_0 empty, thus producing the number $(110)_2 = 6$ in the state places.



(a) $(q_4, A(1), q_6)$



(b) Binary-coded states

Figure 6.2: A simulation of an increment instruction with binary-coded states

Remark that the choice of binary codes for states may influence the total number of inhibitor arcs. Indeed, every transition simulating an arc originating in state q will need to use as many inhibitor arcs as there are zeroes in the binary code assigned to q. Therefore, to keep the number of inhibitor arcs low, we will assign numbers with more non-zero bits to states with more outgoing transitions. This approach yields a strongly universal Petri net N_3 of size $(11, 23, 37, 10)$ and a weakly universal Petri net N'_3 of size $(10, 21, 30, 10)$. In the initial marking, the state places of both nets will contain the binary value $(010)_2$, which is the code of the initial states of both U_7 and U'_7.

Of course, one need not restrict oneself to the simulation of generalized register machines which cannot be further compressed. For example, one could consider only compressing the states corresponding to increment instructions; such partial compression has interesting applications to the construction of small universal multiset rewriting systems (e.g. [6]). In the case of Petri nets, this approach allows building a strongly universal Petri net N_4 of size $(21, 25, 12, 5)$ and a weakly universal Petri net N'_4 of size $(19, 23, 11, 5)$, which use more places and transitions than N_2, N'_2, N_3, or N'_3, but instead have considerably fewer inhibitor arcs.

6.4 Minimising the Number of Places

6.4.1 Non-deterministic Simulation

In this subsection we will construct strongly and weakly universal Petri nets with 5 and 4 places respectively. The constructions will be based on simulations of the universal 3- and 2-register machines U_3 and U_2 described and discussed in Section 5.2. We start by showing that a register machine with n registers can be simulated by a non-deterministic Petri net with only $n + 2$ places. We recall that in the non-deterministic semantics, we only consider those branches of computation which halt. The basic idea is representing a state number of a register machine in unary encoding as the number of tokens in a (single) place of the Petri net, and then using another place to assure that the transformations of state numbers happen correctly.

To illustrate our construction, we consider the register machine $M = (Q, R, q_1, q_4, P)$, where $Q = \{q_1, q_2, q_3, q_4\}$, $R = \{R_0, R_1, R_2\}$, and the set of instructions P is defined as

$$P = \{ (q_1, S(1), q_2, q_3), (q_2, A(0), q_1), (q_3, A(2), q_4), (q_4, \mathit{Stop}) \}.$$

This machine is shown in Figure 6.3 in generalised register machine notation. M adds the contents of register R_1 to R_0, eventually setting R_1 to zero, and then increments R_2 once.

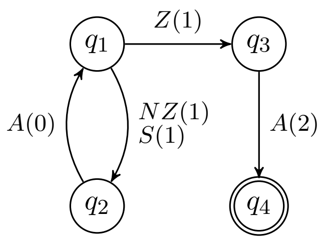


Figure 6.3: The toy register machine M

A non-deterministic Petri net N simulating M is shown in Figure 6.4. It uses a place to store the value of each register (places R_0, R_1, and R_2), and two more places to store state numbers and validate state transitions (places Q_1 and Q_0). The transitions of N read the state number out of Q_0 or Q_1, simulate the corresponding instruction of the register machine, and then put the new state number into Q_1 or Q_0. The inhibitor arcs connecting transitions to state places prevent incomplete, and therefore incorrect, readings of state numbers. We do not need to represent the final state q_4 in the Petri net: it suffices that N carries out the operation associated with q_3 and halts.

The transitions of N fall into the following three groups. The transitions T_0, T_1, T_2, and T_3 simulate the arcs $q_1 \to q_2$, $q_1 \to q_3$, $q_2 \to q_1$, and $q_3 \to q_4$ in the graph of M, respectively. For example, the transition T_0 decrements R_1 if possible, consumes one state token from place Q_0, and puts two tokens into Q_1. The transitions T_0', T_1', T_2', and T_3' simulate the same transitions of M, in the same order, but consume state tokens from Q_1 and add tokens to Q_0. Since transitions are not restricted to consuming all tokens from a state place, it might happen that, for example, T_0 fires when Q_0 contains 2 tokens. If this happens, however, the transitions T_0 through T_3 and T_0' through T_3' become blocked, because they have inhibitor arcs going to one

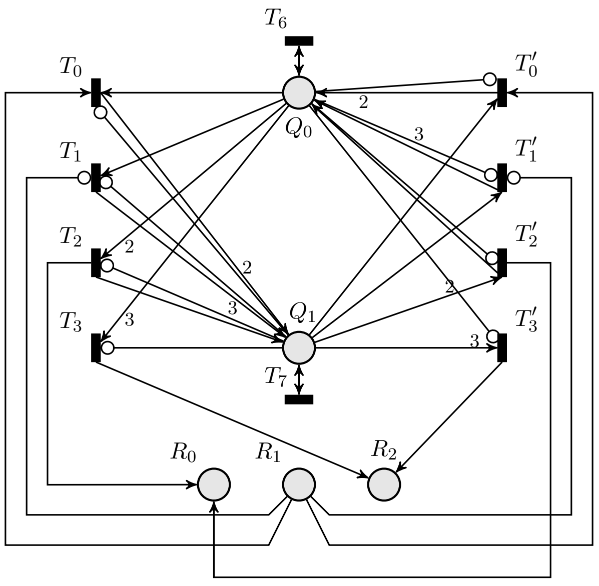


Figure 6.4: A non-deterministic Petri net N simulating M

of the state places, neither of which is empty. In this situation, loop transitions T_6 and T_7 assure that the net never halts. The net halts correctly by firing one of the transitions T_3 or T'_3, which consume, but do not reproduce, state tokens.

Note that, while in the case of the toy register machine M we did not need to explicitly represent the final state of M, this omission is not possible in the case in which several arcs go into the final state of M. Indeed, suppose that M contains two arcs of the form $q \to q_f$ and $q' \to q_f$, and we attempt to simulate both of them with transitions which only consume state tokens from either Q_0 or Q_1, and do not put anything back. The states q and q' must have different codes, $c(q)$ and $c(q')$; suppose that $c(q') > c(q)$. But then, it is possible that, when the code of state q' is put into Q_0 or Q_1, the transition simulating to $q \to q_f$ fires, effectively setting the state code to $c(q') - c(q)$, regardless of whether the corresponding state transition is possible in M. Therefore, in what follows, we will still represent the final states of the simulated register machines explicitly.

The number of places in a Petri net constructed is this way only depends on the number of registers of the simulated register machine, and is in no connection with the number of instructions. We can therefore construct a strongly universal Petri net with 5 places only and a weakly universal net with 4 places by simulating U_3 and U_2. However, before we carry out the actual construction, remark that the maximal degree of a transition in the resulting Petri net depends on how states are represented as numbers of tokens in state places. Indeed, suppose that $c : Q \to \mathbb{N}$ is the function assigning a code to each state. Then the degree of a transition T_{ij} simulating the arc $q_i \to q_j$ is given by $d_{ij} + c(q_i) + c(q_j)$, where d_{ij} is the number of arcs connecting T_{ij} to register places. Our goal is minimising the worst-case transition degree, i.e. the degree of the transition which has the maximal degree in the resulting Petri net. We will now describe how linear programming can be used to find an assignment c that satisfies our requirements.

Remark at first that we may assume without losing generality that $c(q) \leq |Q|$, for any $q \in Q$. We may also assume that $q_f \in Q$ is the only halting state of the simulated machine. Given that both the domain and the codomain of c are bounded, we can represent it as a family of binary variables in the following way:

$$c_{i,i'} = \begin{cases} 1, & \text{if } c(q_i) = i', \\ 0, & \text{otherwise.} \end{cases}$$

In order to assure that these variables represent c correctly, some normalisation conditions are in order:

$$\forall q_i \in Q \,.\, \sum_{1 \le i' \le |Q|} c_{i,i'} = 1 \quad \text{and} \quad \forall 1 \le i' \le |Q| \,.\, \sum_{q_i \in Q} c_{i,i'} = 1.$$

These conditions require that every state have exactly one code assigned, and that each code be picked exactly once. To assure correct halting, we also require that $c_{f,|Q|} = 1$, i.e. that the final state q_f be assigned the maximal code.

The family of variables $c_{i,i'}$ can be used to express the cost of the arc $q_i \to q_j$, i.e. the degree of the transition T_{ij} simulating it, in the following way:

$$deg(T_{ij}) = d_{ij} + c(q_i) + c(q_j) = d_{ij} + \sum_{1 \le i' \le |Q|} i' \cdot c_{i,i'} + \sum_{1 \le j' \le |Q|} j' \cdot c_{j,j'}.$$

Before formulating the optimisation problem itself, we introduce a shortcut notation for the set of pairs of states between which there exists an arc in the graph of the simulated register machine:

$$B = \{(p,q) \mid (p, A(i), q) \in P\} \cup \{(p,q),(p,s) \mid (p, S(i), q, s) \in P\}.$$

We can now write the linear programming problem optimizing the maximal transition degree by putting together the normalisation conditions on the family of variables $c_{i,i'}$ and the expressions for transition degrees:

$$\begin{array}{rrcll} \text{Minimise} & C & & & \\ \text{Subject to} & deg(T_{ij}) & \le & C, & \text{for all } (q_i, q_j) \in B, \\ & \displaystyle\sum_{1 \le i' \le |Q|} c_{i,i'} & = & 1, & \text{for all } q_i \in Q, \\ & \displaystyle\sum_{q_i \in Q} c_{i,i'} & = & 1, & \text{for all } 1 \le i' \le |Q|, \\ & c_{f,|Q|} & = & 1. & \end{array}$$

Remark that both the number of variables and the number and size of the constraints of this problem depend on the number of states and transitions of the simulated register machine. We will therefore consider simulating generalized register machines with compressed increment instructions which have less states than their conventional counterparts. We saw in Section 5.2 that machines U_3 and U_2 have 221 and 165 increment instructions respectively; given that the number of binary variables $c_{i,i'}$ in the linear optimisation problem is quadratic in the number of states, cutting down on increment instructions alone reduces the size of the family of these variables by a factor of four. Furthermore, the number of transitions in the resulting Petri nets will be three times less, roughly.

On the other hand, since compressing an increment instruction essentially corresponds to adding an extra arc to a Petri net transition, the maximal increase of the transition degree when considering partially compressed versions of U_3 and U_2 will be equal to the length of the longest chain of increment instructions. From the construction of these machines we know that such chains arise whenever multiplication is required and that their lengths are defined by the choice of prime numbers used in exponential encoding. Since U_3 simulates an 8-register machine and U_2 a 7-register one [71], we know that the longest chain of increments in U_3 contains 19 instructions, and the longest chain in U_2 contains 17 instructions (eighth and seventh prime numbers, respectively). We conclude therefore that the increase in transition degree due to simulating several increments at once is massively outweighed by the cut in transition degree due to the reduction of the number of states to simulate.

To attack the instances of this linear programming problem for U_3 and U_2 with compressed increments, we used the Gurobi Optimizer [94]. The problem itself was formulated in the AMPL model description language [33]. Based on the results obtained with these tools, we constructed the 5-place strongly universal non-deterministic Petri net N_5 of size $(5, 590, 734, 208)$ and the 4-place weakly universal non-deterministic Petri net N_5' of size $(4, 452, 562, 162)$.

6.4.2 Deterministic Simulations

It turns out that, even without non-determinism, it is possible to simulate any n-register machine with a Petri net with $n + 2$ places only. The idea is to represent state q_i by putting i tokens in state place Q_0 and $|Q| - i$ tokens in place Q_1, and then to have all transitions read the state out of both places Q_0 and Q_1. With this approach, any two multisets $Q_0^i Q_1^{|Q|-i}$ and $Q_0^j Q_1^{|Q|-j}$, representing the markings of state places corresponding to q_i and q_j, $q_i \neq q_j$, are incomparable with respect to the submultiset relation. This means that, if the simulated register machine is deterministic, the resulting Petri net will be deterministic as well. However, in this case all transitions will end up having the degree of at least $2|Q|$, because they need to consume $i + (|Q| - i) = |Q|$ state tokens and produce another $|Q|$ tokens coding the next state. Using this approach to simulate U_3 and U_2 with compressed increment instructions, one can construct a strongly universal Petri net N_6 of size $(5, 293, 146, 314)$ and a weakly universal Petri net N_6' of size $(4, 224, 112, 242)$. Notice the increase in the maximal transition degree by about a hundred with respect to the nets N_5 and N_5'.

6.5 Minimising the Number of Inhibitor Arcs

In the previous subsections we saw that it was possible to construct universal Petri nets with as few as four places by having rather complex transitions and by employing an important number of inhibitor arcs. We will now show that it is possible to construct a Petri net which will only have as many inhibitor arcs as there are registers in the simulated register machine. This can be achieved by "outsourcing" the actual zero-check-and-decrement action to special checker subnets instead of using an inhibitor arc per each $S(i)$ instruction. Figure 6.5 shows how a decrement can be simulated in this way. Essentially, the state token in Q_j is "split" into a token in place C_i activating the checker subnet, and another token which waits in place Q_j'

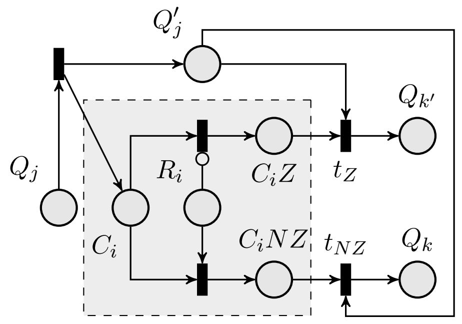


Figure 6.5: A Petri net simulating a *RiZM* instruction using a checker subnet.

for the result of the checker. The checker subnet is the exact copy of the net from Figure 6.1 simulating a decrement instruction, and it has the same function.

It is now possible to use checker subnets to simulate the 3- and 2-register machines U_3 and U_2. In addition to having one place per state and one per register of the simulated machine, the Petri nets will have to include one extra place per decrement instruction (the waiting place Q'_j in Figure 6.5), as well as three more places per register (for the checker subnets); the maximal transition degree is as low as 3 though. This simulation yields a strongly universal Petri net N_7 of size $(525, 659, 3, 3)$ and a weakly universal Petri net N'_7 of size $(397, 504, 2, 3)$. Note that, since reachability is decidable for Petri nets with one inhibitor arc [16, 106], the net N'_7 uses the minimal number of inhibitor arcs to achieve universality.

To considerably reduce the number of places at the expense of an increase in the maximal transition degree, it is possible to turn to representations of U_3 and U_2 as generalized register machines with compressed increment instructions, just as shown in Subsection 6.4.1. By simulating these slightly modified versions, we construct the strongly universal Petri net N_8 of size $(304, 438, 3, 22)$ and the weakly universal Petri net N'_8 of size $(232, 339, 2, 20)$.

6.6 Final Remarks

In this section we considered the question of universality for Petri nets with inhibitor arcs and gave several small universal Petri nets of different descriptional complexity. We remark that the constructions from Sections 6.2, 6.3, and 6.5 simulate Korec's machines without any slowdown, while those from Subsections 6.4.1 and 6.4.2 do so with exponential slowdown. The sizes of constructed universal Petri nets are given in Table 6.1.

Some of our constructions achieve the theoretical minimum for the correspond-

Table 6.1: The sizes of universal Petri nets constructed in this chapter

	Strong universality								Weak universality							
	N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8	N'_1	N'_2	N'_3	N'_4	N'_5	N'_6	N'_7	N'_8
Places	30	14	11	21	5	5	525	304	27	13	10	19	4	4	397	232
Transitions	34	23	23	25	590	293	659	438	31	21	21	23	452	224	504	339
Inhibitor arcs	12	30	37	12	734	146	3	3	11	23	30	11	562	112	2	2
Maximal degree	3	6	10	5	208	314	3	22	3	6	10	5	162	242	3	20

ing parameters of descriptional complexity. Thus, nets N_1 and N_1' described in Section 6.2 attain the minimal possible value for transition degree: 3, while nets N_7, N_7', N_8, and N_8' from Section 6.5 achieve universality with the smallest possible number of inhibitor arcs: 3 for strong universality and 2 for weak. As for the number of places and transitions, we conjecture that the values that we give in the other subsections: 23 and 21 transitions (Section 6.3), and 5 and 4 places (Subsections 6.4.1 and 6.4.2) for strong and weak universality respectively, cannot be significantly improved upon because of inherent limitations of Petri nets with inhibitor arcs.

Our constructions bring out a number of interesting trade-offs. Comparing nets N_1 and N_1' from Section 6.2 with N_2 and N_2', N_3 and N_3', and N_4 and N_4', we remark that a reduction of the number of places leads to an increase in the number of inhibitor arcs and the maximal degree of a transition. Nets N_5, N_5', N_6, and N_6' from Subsections 6.4.1 and 6.4.2 accentuate this trade-off even more: they are universal with 5 and 4 places, but rely on considerably more inhibitor arcs and transitions of big degrees.

Another trade-off brought out in this section is that between the number of inhibitor arcs and the number of places and transitions. Nets N_7 and N_7' from Section 6.5 use 3 and 2 inhibitor arcs respectively, but rely on considerably more places and transitions than nets N_1 and N_1'.

An interesting trade-off is observable between the pairs of nets N_5 and N_5', and N_6 and N_6': the first pair is non-deterministic, while the second pair is deterministic. Switching to deterministic simulation halved the number of transitions and reduced the number of inhibitor arcs by a factor of five roughly, while keeping the number of places at the same value. On the other hand, the degrees of transitions in deterministic nets N_6 and N_6' is about 50% bigger than that in nets N_5 and N_5'.

Conclusions

In this thesis we focused on insertion-deletion systems and on Petri nets with inhibitor arcs, which are two formal devices whose study is, in part, motivated by biological and biochemical considerations. We started by stating that leftist grammars, introduced in [93], correspond to a subclass of insertion-deletion systems of size $(1,1,0;1,1,0)$ with a single one-symbol axiom. Therefore, the majority of results on leftist grammars can be directly taken over to insertion-deletion systems of the said size. We then switched focus to a slightly more general case, in which either the insertion or the deletion rules are allowed to make use of contexts of length 2. We showed that both systems of size $(1,2,0;1,1,0)$ and $(1,1,0;1,2,0)$ can generate all regular languages, and that, moreover, rules of this size allow performing a one-time filtering by a regular language. Given that any system of size $(1,1,0;1,1,0)$ can be trivially considered to be either of size $(1,2,0;1,1,0)$ or $(1,1,0;1,2,0)$, we concluded that the systems of the latter size could simulate those of size $(1,1,0;1,1,0)$ equipped with a regular filter, which means that, when contexts of size 2 are allowed, some rather simple non-context-free languages can be generated.

We continued by looking into contexts of larger sizes and showed that increasing context size does not augment the expressive power of the system. In other words, all languages generated by insertion-deletion systems of size $(1,m,0;1,q,0)$ can be generated by systems of size $(1,2,0;1,1,0)$ or of size $(1,1,0;1,2,0)$. This rather unexpected result shows that the gap in expressiveness between the power of the systems of the latter sizes and those of size $(1,1,0;1,1,0)$ is larger than one would intuitively expect. We do nevertheless conjecture that having contexts longer than one symbol does not lead to universality, because one-sided contexts do not seem to allow sending unbounded information in both directions.

After discussing insertion-deletion systems of size $(1,m,0;1,q,0)$, we turned back to those of size $(1,1,0;1,1,0)$ and to the results as to the complexity of the languages recognised by leftist grammars. Because of the small size of the rules of these systems, the constructions giving the lower bounds of their power turn out rather hefty and difficult to grasp. To facilitate understanding and design of such systems, we introduced a new instrument: derivation graphs. Considering pictures of these graphs instead of naked derivations offers a much more intuitive insight into the function of individual symbols or groups of symbols. We showcased this new tool by illustrating some of the leftist constructions pertaining to the exploration of complexity of the membership problem, and also by giving examples of reasoning that can be more easily conducted on derivation graphs.

After discussing pure one-sided insertion-deletion systems, we moved to controlled insertion and deletion, and considered semi-conditional, random context, and graph control mechanisms. Additional control over insertion and deletion rules allows achieving a strict increase in computational power, which is especially visible in the case of semi-conditional insertion-deletion systems: we showed that these systems are capable of generating the family of recursively enumerable languages with rather minimalistic context-free one-symbol insertion and deletion rules, and at most two-symbol strings in permitting and forbidding contexts. This demonstrates that the semi-conditional control mechanism provides better increase in computational power than the mechanism of graph control, because the latter does not go beyond $PsRE$ with such rule size [4]. The aspect of the rules and the way the control mechanism is specified permit considering semi-conditional insertion-deletion systems as a particular case of networks of evolutionary processors, which means that our result can be transcribed to fit that area as well.

In our study of insertion-deletion systems with random context conditions, we unveiled an interesting asymmetry related to swapping the sizes of insertion and deletion rules: systems with random context rules of size $(2, 0, 0; 1, 1, 0)$ are computationally complete, while those having rules of size $(1, 1, 0; 2, 0, 0)$ (and, more generally, of size $(1, 1, 0; p, 1, 1)$) are not. This asymmetry is intuitively unexpected and surprising, given that, in the cases analysed previously, the computational power was largely insusceptible to such an exchange of sizes.

Finally, we discussed graph-controlled insertion-deletion systems with rules of size $(1, 2, 0; 1, 1, 0)$ and $(1, 1, 0; 1, 2, 0)$, and showed that computational completeness can be achieved with 3 states. Compared to [38] this result points out an interesting trade-off between the sizes of contexts in insertion-deletion rules and the number of states: with 4 states, computational completeness is obtained already with insertion and deletion rules of size $(1, 1, 0; 1, 1, 0)$. Now it remains an open question if the number of states can be further decreased for the investigated systems or for systems having bigger contexts for the insertion or deletion rules.

To conclude our discussion of computing devices based on the operations of insertion and deletion, we turned to networks of evolutionary processors. We described 3 universal networks with a small number of rules, including a universal NEP with 4 rules only. Our constructions clearly show that a large part of the computational power of NEPs is provided by filters. It could therefore be interesting and practical to investigate a similar universality problem for systems in which the filtering components are restricted, e.g., NEPs from [28, 29], which use subclasses of regular languages as filters, or hybrid NEPs that use random context filters.

We also remark that, in NEPs, one traditionally considers a single type of operation (insertion, deletion or substitution) per processor, while we allow mixed nodes. Yet, the constructions we give in the present thesis can be easily adapted to the uniformity condition without modifying the number of rules by simply increasing the number of nodes in the network.

After discussing the computational power of one-sided insertion-deletion systems, we moved on to the problem of universality in the classes of register machines and Petri nets with inhibitor arcs. We started by generalising the concept of a register machine to allow more complex instructions to be carried out at a time, and then gave strongly and a weakly universal generalised register machines with 7 states only. Next we considered universal register machines with a small number of registers and applied Marvin Minsky's algorithm from [92] to actually construct universal 3- and 2-register machines. While Minsky's approach of exponential coding of the values of any number of registers into a single number is rather well-known, we have not been able to find any reference to the actual descriptions of concrete machines built after this technique.

After considering register machines, we turned to Petri nets, and capitalising on the close relation of these two computing devices, we described 8 pairs of strongly and weakly universal Petri nets with inhibitor arcs. The nets are classified into four groups, by the minimised parameter of descriptional complexity. The results exhibit a number of interesting trade-offs, especially between the maximal transition degree on the one hand and the number of transitions and inhibitor arcs on the other hand. In the case of nets N_5 and N_6 (respectively N_5' and N_6') this trade-off is particularly conspicuous and also includes a switch from non-deterministic constructs (N_5 and N_5') to deterministic ones (N_6 and N_6').

We remark that, since we always simulated deterministic (generalised) register machines, and since most of our translation techniques preserve this property, all of the shown Petri nets, except N_5 and N_5', are also deterministic.

In what concerns minimal values for the parameters, we saw that 3 is the lowest possible maximal transition degree and nets N_1, N_1', N_7, and N_7' actually attain that lower bound. Similarly, it is known that at least 2 inhibitor arcs are necessary to render reachability undecidable for a Petri net [16, 106], and we do show nets N_7' and N_8' which have precisely 2 inhibitor arcs. Even though we cannot indicate a lower bound for the other parameters, we conjecture that the number of transitions cannot be under 23. We also believe it to be difficult to substantially decrease the number of places below what was obtained in the paper. It might be possible to achieve strong universality with 4 places, because the input/output register of U_3 is only modified during the initial and final phases of execution of the machine, when input is read and output is produced [91]. We conjecture that 3 places or less are insufficient to achieve strong universality, however.

Finally, we recall that our results on Petri nets can be directly translated to P systems [101] and, more generally, multiset rewriting [6], or vector addition systems [36].

The following list summarises the open problems and promising research directions related to the results from the present thesis.

1. Are insertion-deletion systems of size $(1, m, 0; 1, q, 0)$, $m, q > 1$, computationally complete? If not, are the languages they generate all context-sensivite?
2. Are insertion-deletion systems of size $(n, m, 0; 1, 1, 0)$ or $(1, 1, 0; p, q, 0)$ computationally complete?
3. Can computational completeness be achieved with graph-controlled insertion-deletion systems of sizes $(1, 2, 0; 1, 1, 0)$ or $(1, 1, 0; 1, 2, 0)$ with 2 states only?
4. Do there exist strongly universal Petri nets with inhibitor arcs and with 4 places (computing unary functions)?
5. Do there exist universal Petri nets with inhibitor arcs and with 3 places?
6. What is the minimal number of transitions in a universal Petri net with inhibitor arcs?
7. Are semi-conditional insertion-deletion systems of size $(1, 0, 0; 1, 0, 0)$ and of degrees $(1, 2)$ and $(2, 1)$ computationally complete?
8. What is the minimal number of instructions in strongly universal 3-register machines and weakly universal 2-register machines?

Bibliography

[1] Leonard M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266:1021–1024, November 11, 1994.

[2] Tilak Agerwala and Mike Flynn. Comments on capabilities, limitations and "correctness" of Petri nets. *SIGARCH Computer Architecture News*, 2(4):81–86, December 1973.

[3] Artiom Alhazov, Alexander Krassovitskiy, Yurii Rogozhin, and Sergey Verlan. *Small Size Insertion and Deletion Systems*, volume 228 of *Mathematics, Computing, Language, and Life: Frontiers in Mathematical Linguistics and Language Theory*, chapter 1, pages 459–524. World Scientific, 2010.

[4] Artiom Alhazov, Alexander Krassovitskiy, Yurii Rogozhin, and Sergey Verlan. P systems with minimal insertion and deletion. *Theoretical Computer Science*, 412(1–2):136 – 144, 2011.

[5] Artiom Alhazov, Carlos Martín-Vide, and Yurii Rogozhin. On the number of nodes in universal networks of evolutionary processors. *Acta Informatica*, 43(5):331–339, 2006.

[6] Artiom Alhazov and Sergey Verlan. Minimization strategies for maximally parallel multiset rewriting systems. *Theoretical Computer Science*, 412(17):1581 – 1591, 2011.

[7] Toshiro Araki and Tadao Kasami. Some decision problems related to the reachability problem for Petri nets. *Theoretical Computer Science*, 3(1):85 – 104, 1976.

[8] Gheorghe Păun and Nguyen Xuan My. On the inner contextual grammars. *Revue Roumaine de Mathématiques Pures et Appliquées*, 25:641–651, 1980.

[9] Brenda S. Baker and Ronald V. Book. Reversal-bounded multipushdown machines. *Journal of Computer and System Sciences*, 8(3):315 – 332, 1974.

[10] Falko Bause. On the analysis of Petri nets with static priorities. *Acta Informatica*, 33(5):669–685, 1996.

[11] Robert Benne. RNA editing in trypanosomes. *Molecular Biology Reports*, 16(4):217–227, 1992.

[12] Robert Benne. *RNA-Editing: The Alteration of Protein Coding Sequences of RNA.* Ellis Horwood, Chichester, West Sussex, 1993.

[13] Franziska Biegler, Michael J. Burrell, and Mark Daley. Regulated RNA rewriting: Modelling RNA editing with guided insertion. *Theoretical Computer Science*, 387(2):103–112, November 2007.

[14] Rémi Bonnet. The reachability problem for vector addition system with one zero-test. In Filip Murlak and Piotr Sankowski, editors, *Mathematical Foundations of Computer Science 2011*, volume 6907 of *Lecture Notes in Computer Science*, pages 145–157. Springer Berlin Heidelberg, 2011.

[15] Paolo Bottoni, Carlos Martín-Vide, Gheorghe Păun, and Grzegorz Rozenberg. Membrane systems with promoters/inhibitors. *Acta Informatica*, 38(10):695–720, 2002.

[16] Hans Kleine Büning, Theodor Lettmann, and Ernst W. Mayr. Projections of vector addition system reachability sets are semilinear. *Theoretical Computer Science*, 64(3):343–350, May 1989.

[17] Michael J. Burrell. Computational modelling of uridine insertion and deletion in kinetoplastid RNA, Master's Thesis. London, Canada, 2005.

[18] Juan Castellanos, Carlos Martín-Vide, Victor Mitrana, and Jose M. Sempere. Solving NP-complete problems with networks of evolutionary processors. In José Mira and Alberto Prieto, editors, *Connectionist Models of Neurons, Learning Processes, and Artificial Intelligence*, volume 2084 of *Lecture Notes in Computer Science*, pages 621–628. Springer Berlin Heidelberg, 2001.

[19] Juan Castellanos, Carlos Martín-Vide, Victor Mitrana, and José M. Sempere. Networks of evolutionary processors. *Acta Informatica*, 39(6-7):517–529, 2003.

[20] Matteo Cavaliere and Dragoş Sburlan. Time-independent P systems. In Giancarlo Mauri, Gheorghe Păun, Mario J. Pérez-Jiménez, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing, 5th International Workshop, WMC 2004, Milan, Italy, June 14-16, 2004, Revised Selected and Invited Papers*, volume 3365 of *Lecture Notes in Computer Science*, pages 239–258. Springer, 2004.

[21] Pierre Chambart and Philippe Schnoebelen. Toward a compositional theory of leftist grammars and transformations. In Luke Ong, editor, *Foundations of Software Science and Computational Structures*, volume 6014 of *Lecture Notes in Computer Science*, pages 237–251. Springer Berlin Heidelberg, 2010.

[22] Ashish Choudhary and Kamala Krithivasan. Network of evolutionary processors with splicing rules and permitting context. *Biosystems*, 87(2–3):111 – 116, 2007.

[23] Gabriel Ciobanu, Linqiang Pan, Gheorghe Păun, and Mario J. Pérez-Jiménez. P systems with minimal parallelism. *Theoretical Computer Science*, 378(1):117 – 130, 2007.

[24] Erzsébet Csuhaj-Varjú, Maurice Margenstern, György Vaszil, and Sergey Verlan. On small universal antiport P systems. *Theoretical Computer Science*, 372(2-3):152–164, 2007.

[25] Erzsébet Csuhaj-Varjú and Arto Salomaa. Networks of parallel language processors. In Gheorghe Păun and Arto Salomaa, editors, *New Trends in Formal Languages*, volume 1218 of *Lecture Notes in Computer Science*, pages 299–318. Springer, 1997.

[26] Silvano Dal Zilio and Enrico Formenti. On the dynamics of PB systems: A Petri net view. In Carlos Martín-Vide, Giancarlo Mauri, Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing*, volume 2933 of *Lecture Notes in Computer Science*, pages 153–167. Springer Berlin Heidelberg, 2004.

[27] Jrgen Dassow and Gheorghe Paun. *Regulated Rewriting in Formal Language Theory.* Springer Publishing Company, Incorporated, 1st edition, 2012.

[28] Jürgen Dassow, Florin Manea, and Bianca Truthe. Networks of evolutionary processors: the power of subregular filters. *Acta Inf.*, 50(1):41–75, 2013.

[29] Jürgen Dassow and Bianca Truthe. On networks of evolutionary processors with state limited filters. In Henning Bordihn, Rudolf Freund, Markus Holzer, Thomas Hinze, Martin Kutrib, and Friedrich Otto, editors, *Second Workshop on Non-Classical Models for Automata and Applications - NCMA 2010, Jena, Germany, August 23 - August 24, 2010. Proceedings*, volume 263 of *books@ocg.at*, pages 57–70. Austrian Computer Society, 2010.

[30] Erik P. de Vink, Hans Zantema, and Dragan Bošnački. RNA-editing with combined insertion and deletion preserves regularity. *Scientific Annals of Computer Science*, 23(1):39–73, 2013.

[31] Michael Domaratzki and Alexander Okhotin. Representing recursively enumerable languages by iterated deletion. *Theoretical Computer Science*, 314(3):451–457, 2004.

[32] Catherine Dufourd, Petr Jancar, and Philippe Schnoebelen. Boundedness of reset P/T nets. In Jiří Wiedermann, Peter van Emde Boas, and Mogens Nielsen, editors, *Automata, Languages and Programming, 26th International Colloquium, ICALP'99, Prague, Czech Republic, July 11-15, 1999, Proceedings*, volume 1644 of *Lecture Notes in Computer Science*, pages 301–310. Springer, 1999.

[33] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming.* Duxbury Press, Brooks/Cole Publishing Company, 2nd edition, 2002.

[34] Rudolf Freund. Asynchronous P systems and P systems working in the sequential mode. In Giancarlo Mauri, Gheorghe Păun, MarioJ. Pérez-Jiménez, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing*, volume 3365 of *Lecture Notes in Computer Science*, pages 36–62. Springer Berlin Heidelberg, 2005.

[35] Rudolf Freund. Transition and halting modes in (tissue) P systems. In Gheorghe Păun, Mario J. Pérez-Jiménez, Agustín Riscos-Núñez, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing*, volume 5957 of *Lec-*

ture Notes in Computer Science, pages 18–29. Springer Berlin Heidelberg, 2010.

[36] Rudolf Freund, Oscar Ibarra, Gheorghe Păun, and Hsu-Chun Yen. Matrix languages, register machines, vector addition systems. *Third Brainstorming Week on Membrane Computing*, pages 155–167, 1/31//05-2/4/05 2005.

[37] Rudolf Freund, Marian Kogler, and Marion Oswald. A general framework for regulated rewriting based on the applicability of rules. In Jozef Kelemen and Alica Kelemenová, editors, *Computation, Cooperation, and Life*, volume 6610 of *Lecture Notes in Computer Science*, pages 35–53. Springer Berlin Heidelberg, 2011.

[38] Rudolf Freund, Marian Kogler, Yurii Rogozhin, and Sergey Verlan. Graph-controlled insertion-deletion systems. In Ian McQuillan and Giovanni Pighizzini, editors, *Proceedings Twelfth Annual Workshop on Descriptional Complexity of Formal Systems, DCFS 2010, Saskatoon, Canada, 8-10th August 2010.*, volume 31 of *EPTCS*, pages 88–98, 2010.

[39] Rudolf Freund, Alberto Leporati, Giancarlo Mauri, Antonio E. Porreca, Sergey Verlan, and Claudio Zandron. Flattening in (tissue) P systems. In Artiom Alhazov, Svetlana Cojocaru, Marian Gheorghe, Yurii Rogozhin, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing*, volume 8340 of *Lecture Notes in Computer Science*, pages 173–188. Springer Berlin Heidelberg, 2014.

[40] Rudolf Freund and Marion Oswald. A small universal antiport P system with forbidden context. In Hing Leung and Giovanni Pighizzini, editors, *8th International Workshop on Descriptional Complexity of Formal Systems - DCFS 2006, Las Cruces, New Mexico, USA, June 21 - 23, 2006. Proceedings*, pages 259–266. New Mexico State University, Las Cruces, New Mexico, USA, 2006.

[41] Rudolf Freund and Sergey Verlan. A formal framework for static (tissue) P systems. In George Eleftherakis, Petros Kefalas, Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing*, volume 4860 of *Lecture Notes in Computer Science*, pages 271–284. Springer Berlin Heidelberg, 2007.

[42] Pierluigi Frisco. P systems, Petri nets, and program machines. In Rudolf Freund, Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing*, volume 3850 of *Lecture Notes in Computer Science*, pages 209–223. Springer Berlin Heidelberg, 2006.

[43] Kaoru Fujioka. Morphic characterizations of languages in Chomsky hierarchy with insertion and locality. *Information and Computation*, 209(3):397 – 408, 2011. 3rd International Conference on Language and Automata Theory and Applications (LATA 2009).

[44] Kaoru Fujioka. Morphic characterizations with insertion systems controlled by a context of length one. *Theoretical Computer Science*, 469(0):69–76, 2013.

[45] Boris S. Galiukschov. Semicontextual grammars. *Matematicheskaya Logica i Matematicheskaya Lingvistika*, pages 38–50, 1981. Tallin University, (in russian).

[46] Viliam Geffert. Normal forms for phrase-structure grammars. *Informatique Théorique et Applications*, 25:473–498, 1991.

[47] Sheila A. Greibach. Remarks on blind and partially blind one-way multicounter machines. *Theoretical Computer Science*, 7(3):311 – 324, 1978.

[48] Michel Hack. The recursive equivalence of the reachability problem and the liveness problem for Petri nets and vector addition systems. In *Proceedings of the 15th Annual Symposium on Switching and Automata Theory (Swat 1974)*, SWAT '74, pages 156–164, Washington, DC, USA, 1974. IEEE Computer Society.

[49] Michel Hack. *Decidability Questions for Petri Nets*. PhD thesis, Cambridge, MA, USA, 1976.

[50] David Haussler. *Insertion and Iterated Insertion as Operations on Formal Languages*. PhD thesis, University of Colorado at Boulder, 1982.

[51] David Haussler. Insertion languages. *Information Sciences*, 31(1):77–89, 1983.

[52] John Hopcroft and Jean-Jacques Pansiot. On the reachability problem for 5-dimensional vector addition systems. *Theoretical Computer Science*, 8(2):135 – 159, 1979.

[53] Sergiu Ivanov. Basic concurrency resolution in clock-free P systems. In Marian Gheorghe, Gheorghe Pŭun, Grzegorz Rozenberg, Arto Salomaa, and Sergey Verlan, editors, *Membrane Computing*, volume 7184 of *Lecture Notes in Computer Science*, pages 226–242. Springer Berlin Heidelberg, 2012.

[54] Sergiu Ivanov. A formal framework for clock-free networks of cells. *International Journal of Computer Mathematics*, 90(4):776–788, 2013.

[55] Sergiu Ivanov, Elisabeth Pelz, and Sergey Verlan. Small universal nondeterministic Petri nets with inhibitor arcs. In Helmut Jürgensen, Juhani Karhumäki, and Alexander Okhotin, editors, *Descriptional Complexity of Formal Systems - 16th International Workshop, DCFS 2014, Turku, Finland, August 5-8, 2014. Proceedings*, volume 8614 of *Lecture Notes in Computer Science*, pages 186–197. Springer, 2014.

[56] Sergiu Ivanov, Elisabeth Pelz, and Sergey Verlan. Small universal Petri nets with inhibitor arcs. In *Informal Electronic Proceedings of Computability in Europe*, 2014.

[57] Sergiu Ivanov, Yurii Rogozhin, and Sergey Verlan. Small universal networks of evolutionary processors. *Journal of Automata, Languages and Combinatorics*, 19(1-4):133–144, 2014.

[58] Sergiu Ivanov and Sergey Verlan. Random context and semi-conditional insertion-deletion systems. *CoRR*, abs/1112.5947, 2011.

[59] Sergiu Ivanov and Sergey Verlan. About one-sided one-symbol insertion-deletion P systems. In Artiom Alhazov, Svetlana Cojocaru, Marian Gheorghe, Yurii Rogozhin, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing*, volume 8340 of *Lecture Notes in Computer Science*, pages 225–237. Springer Berlin Heidelberg, 2014.

[60] Kurt Jensen. Coloured Petri nets and the invariant-method. *Theoretical Computer Science*, 14(3):317 – 336, 1981.

[61] Kurt Jensen. Coloured Petri nets: A high level language for system design and analysis. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 342–416. Springer Berlin Heidelberg, 1991.

[62] Tomasz Jurdziński. On complexity of grammars related to the safety problem. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *Automata, Languages and Programming*, volume 4052 of *Lecture Notes in Computer Science*, pages 432–443. Springer Berlin Heidelberg, 2006.

[63] Tomasz Jurdziński. Leftist grammars are non-primitive recursive. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming*, volume 5126 of *Lecture Notes in Computer Science*, pages 51–62. Springer Berlin Heidelberg, 2008.

[64] Tomasz Jurdziński and Krzysztof Loryś. Leftist grammars and the Chomsky hierarchy. In Maciej Liśkiewicz and Rüdiger Reischuk, editors, *Fundamentals of Computation Theory*, volume 3623 of *Lecture Notes in Computer Science*, pages 293–304. Springer Berlin Heidelberg, 2005.

[65] Lila Kari. *On Insertion and Deletion in Formal Languages.* PhD thesis, University of Turku, 1991.

[66] Lila Kari, Gheorghe Păun, Gabriel Thierrin, and Sheng Yu. At the crossroads of dna computing and formal languages: Characterizing RE using insertion-deletion systems. In *Proc. of 3rd DIMACS Workshop on DNA Based Computing*, pages 318–333. Philadelphia, 1997.

[67] Lila Kari and Petr Sosík. On the weight of universal insertion grammars. *Theoretical Computer Science*, 396(1–3):264 – 270, 2008.

[68] Richard M. Karp and Raymond E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147 – 195, 1969.

[69] Stephen Cole Kleene. Representation of events in nerve nets and finite automata. In Claude Shannon and John McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, Princeton, NJ, 1956.

[70] Pascal Koiran and Cristopher Moore. Closed-form analytic maps in one and two dimensions can simulate universal Turing machines. *Theoretical Computer Science*, 210(1):217–223, January 1999.

[71] Ivan Korec. Small universal register machines. *Theoretical Computer Science*, 168(2):267–301, 1996.

[72] S. Rao Kosaraju. Decidability of reachability in vector addition systems (preliminary version). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, pages 267–281, New York, NY, USA, 1982. ACM.

[73] Alexander Krassovitskiy. *Complexity and Modeling Power of Insertion-Deletion Systems*. PhD thesis, Departament de Filologies Romániques, Universitat Rovira and Virgili, 2011.

[74] Alexander Krassovitskiy, Yurii Rogozhin, and Sergey Verlan. Further results on insertion-deletion systems with one-sided contexts. In Carlos Martín-Vide, Friedrich Otto, and Henning Fernau, editors, *Language and Automata Theory and Applications, Second International Conference, LATA 2008, Tarragona, Spain, March 13-19, 2008. Revised Papers*, volume 5196 of *Lecture Notes in Computer Science*, pages 333–344. Springer, 2008.

[75] Alexander Krassovitskiy, Yurii Rogozhin, and Sergey Verlan. Computational power of insertion-deletion (P) systems with rules of size two. *Natural Computing*, 10(2):835–852, 2011.

[76] Alexander Krassovitskiy, Yurii Rogozhin, and Serghey Verlan. Further results on insertion-deletion systems with one-sided contexts. In Carlos Martín-Vide, Friedrich Otto, and Henning Fernau, editors, *Language and Automata Theory and Applications*, volume 5196 of *Lecture Notes in Computer Science*, pages 333–344. Springer Berlin Heidelberg, 2008.

[77] Lars M. Kristensen, Soren Christensen, and Kurt Jensen. The practitioner's guide to coloured Petri nets. *International Journal on Software Tools for Technology Transfer*, 2(2):98–132, 1998.

[78] Joachim Lambek. How to program an infinite abacus. *Canadian Mathematical Bulletin*, 4:295–302, 1961.

[79] Jérôme Leroux. Vector addition system reachability problem: A short self-contained proof. In Adrian-Horia Dediu, Shunsuke Inenaga, and Carlos Martín-Vide, editors, *Language and Automata Theory and Applications*, volume 6638 of *Lecture Notes in Computer Science*, pages 41–64. Springer Berlin Heidelberg, 2011.

[80] Jérôme Leroux and Philippe Schnoebelen. On functions weakly computable by petri nets and vector addition systems. In Joël Ouaknine, Igor Potapov, and James Worrell, editors, *Reachability Problems - 8th International Workshop, RP 2014, Oxford, UK, September 22-24, 2014. Proceedings*, volume 8762 of *Lecture Notes in Computer Science*, pages 190–202. Springer, 2014.

[81] Monica Marcus and Gheorghe Păun. Regulated Galiukshov semicontextual grammars. *Kybernetika*, 26(4):316–326, 1990.

[82] Solomon Marcus. Contextual grammars. *Revue Roumaine de Mathématiques Pures et Appliquées*, 14:1525–1534, 1969.

[83] Solomon Marcus, Gheorghe Păun, and Carlos Martín-Vide. Contextual grammars as generative models of natural languages. *Computational Linguistics*, 24(2):245–274, June 1998.

[84] Maurice Margenstern, Gheorghe Păun, Yurii Rogozhin, and Sergey Verlan. Context-free insertion-deletion systems. *Theoretical Computer Science*, 330(2):339–348, 2005.

[85] Carlos Martín-Vide, Alexandru Mateescu, Joan Miquel-Vergés, and Gheorghe Păun. Internal contextual grammars: Minimal, maximal, and scattered use of selectors. In *Proceedings of the Fourth Bar-Ilan Symposium on Foundations of Artificial Intelligence*, 1995.

[86] Carlos Martín-Vide, Gheorghe Păun, and Arto Salomaa. Characterizations of recursively enumerable languages by means of insertion grammars. *Theoretical Computer Science*, 205(1-2):195–205, 1998.

[87] Artiom Matveevici, Yurii Rogozhin, and Sergey Verlan. Insertion-deletion systems with one-sided contexts. In Jérôme Olivier Durand-Lose and Maurice Margenstern, editors, *Machines, Computations, and Universality, 5th International Conference, MCU 2007, Orléans, France, September 10-13, 2007, Proceedings*, volume 4664 of *Lecture Notes in Computer Science*, pages 205–217. Springer, 2007.

[88] Ernst W. Mayr. An algorithm for the general Petri net reachability problem. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, pages 238–246, New York, NY, USA, 1981. ACM.

[89] Zdzislaw Melzak. An informal arithmetical approach to computability and computation. *Canadian Mathematical Bulletin*, 4:279–293, 1961.

[90] Marvin Minsky. Recursive unsolvability of Post's problem of "tag" and other topics in the theory of Turing machines. *Annals of Mathematics, second series*, 74:437–455, 1961.

[91] Marvin Minsky. Size and structure of universal Turing machines using tag systems. In *Recursive Function Theory: Proceedings, Symposium in Pure Mathematics, Provelence*, volume 5, pages 229–238, 1962.

[92] Marvin Minsky. *Computations: Finite and Infinite Machines.* Prentice Hall, Englewood Cliffts, NJ, 1967.

[93] Rajeev Motwani, Rina Panigrahy, Vijay Saraswat, and Suresh Ventkatasubramanian. On the decidability of accessibility problems (extended abstract). In *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*, STOC '00, pages 306–315, New York, NY, USA, 2000. ACM.

[94] Gurobi Optimization, Inc. Gurobi optimizer reference manual, 2014.

[95] Suhas Shrikrishna Patil. *Coordination of Asynchronous Events.* PhD thesis, Cambridge, MA, USA, 1970.

[96] Elisabeth Pelz. Closure properties of deterministic Petri nets. In *Symposium on Theoretical Aspects of Computer Science, STACS '87*, volume 247 of *Lecture Notes in Computer Science*, pages 371–382. Springer, 1986.

[97] Ion Petre and Sergey Verlan. Matrix insertion-deletion systems. *Theoretical Computer Science*, 456(0):80 – 88, 2012.

[98] Carl Adam Petri. *Communication with automata.* PhD thesis, Universität Hamburg, 1966.

[99] Gheorghe Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61:108–143, 1998.

[100] Gheorghe Păun. Computing with membranes: Attacking NP-complete problems. In I. Antoniou, C.S. Calude, and M.J. Dinneen, editors, *Unconventional Models of Computation, UMC'2K*, Discrete Mathematics and Theoretical Computer Science, pages 94–115. Springer London, 2001.

[101] Gheorghe Păun. *Membrane Computing. An Introduction.* Springer–Verlag, 2002.

[102] Gheorghe Păun, Mario J. Pérez-Jiménez, and Takashi Yokomori. Representations and characterizations of languages in Chomsky hierarchy by means of insertion-deletion systems. *International Journal of Foundations of Computer Science*, 19:859–871, 2008.

[103] Gheorghe Păun, Grzegorz Rozenberg, and Aarto Salomaa. *DNA Computing: New Computing Paradigms.* Springer, 1998.

[104] Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa. *The Oxford Handbook Of Membrane Computing.* Oxford University Press, 2009.

[105] Chander Ramchandani. *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets.* PhD thesis, Cambridge, MA, USA, 1974.

[106] Klaus Reinhardt. Reachability in Petri nets with inhibitor arcs. *Electronic Notes in Theoretical Computer Science*, 223:239–264, 2008.

[107] Vladimir Rogojin. *Gene Assembly in Stichotrichous Ciliates: Elementary Operations, Parallelism and Computation.* PhD thesis, Turku Centre for Computer Science, 2009.

[108] Yurii Rogozhin. Small universal Turing machines. *Theoretical Computer Science*, 168(2):215–240, 1996.

[109] Grzegorz Rozenberg and Arto Salomaa, editors. *Handbook of Formal Languages.* Springer, Berlin, 1997.

[110] Lila Sântean. Six arithmetic-like operations on languages. *Revue Roumaine de Linguistique, Tome XXXIII*, (1):65–73, 1988. Lila Sântean is Lila Kari's maiden name.

[111] Claude E. Shannon. A universal Turing machine with two internal states. *Automata Studies, Annals of Mathematics Studies*, 34:157–165, 1956.

[112] Warren D. Smith. DNA computers in vitro and in vivo. In R.J. Lipton and E.B. Baum, editors, *Proceedings of DIMACS Workshop on DNA Based Computers*, DIMACS Series in Discrete Mathematicts and Theoretical Computer Science, pages 121–185. American Mathematical Society, 1996.

[113] Akihiro Takahara and Takashi Yokomori. On the computational power of insertion-deletion systems. *Natural Computing*, 2(4):321–336, 2003.

[114] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.

[115] Sergey Verlan. *Head Systems and Applications to Bioinformatics*. PhD thesis, University of Metz, 2004.

[116] Sergey Verlan. On minimal context-free insertion-deletion systems. *Journal of Automata, Languages and Combinatorics*, 12(1-2):317–328, 2007.

[117] Sergey Verlan. Study of language-theoretic computational paradigms inspired by biology. Paris, 2010. Habilation thesis.

[118] Hao Wang. A variant to Turing's theory of computing machines. *Journal of the ACM*, 4(1):63–92, January 1957.

[119] Shinichi Watanabe. 5-symbol 8-state and 5-symbol 6-state universal Turing machines. *Journal of the ACM*, 8(4):476–483, 1961.

[120] Damien Woods and Turlough Neary. The complexity of small universal Turing machines: A survey. *Theoretical Computer Science*, 410(4–5):443 – 450, 2009. Computational Paradigms from Nature.

[121] Hsu-Chun Yen. Introduction to Petri net theory. In Zoltán Ésik, Carlos Martín-Vide, and Victor Mitrana, editors, *Recent Advances in Formal Languages and Applications*, volume 25 of *Studies in Computational Intelligence*, pages 343–373. Springer, 2006.

[122] Dmitry A. Zaitsev. Universal Petri net. *Cybernetics and Systems Analysis*, 48(4):498–511, 2012.

[123] Dmitry A. Zaitsev. A small universal Petri net. *EPTCS*, 128:190–202, 2013. In Proceedings of Machines, Computations and Universality (MCU 2013), arXiv:1309.1043.

[124] Hans Zantema. Complexity of guided insertion-deletion in RNA-editing. In Adrian-Horia Dediu, Henning Fernau, and Carlos Martín-Vide, editors, *Language and Automata Theory and Applications*, volume 6031 of *Lecture Notes in Computer Science*, pages 608–619. Springer Berlin Heidelberg, 2010.

Appendix

The following table gives the program of the generalised register machine U_7' obtained by state compression from the weakly universal register machine U_{20} [71].

q_i	q_j	Conditions	Operations
q_1	q_1	NZ(1)	S(1), A(7)
q_1	q_4	Z(1)	A(6)
q_4	q_4	Z(5), Z(6)	
q_4	q_4	NZ(5)	S(5), A(6)
q_4	q_{10}	Z(5), NZ(6)	A(5), S(6)
q_{10}	q_1	Z(6), Z(7)	
q_{10}	q_1	NZ(4), NZ(6), Z(7)	S(4)
q_{10}	q_4	Z(6), NZ(7)	A(1), S(7)
q_{10}	q_{10}	NZ(6), NZ(7)	A(1), A(5), S(6), S(7)
q_{10}	q_{16}	Z(4), NZ(6), Z(7)	
q_{16}	q_1	Z(0), Z(5)	
q_{16}	q_1	NZ(0), NZ(4), Z(5)	S(0), S(4)
q_{16}	q_{18}	NZ(5)	S(5)
q_{16}	*Stop*	NZ(0), Z(4), Z(5)	S(0)
q_{18}	q_1	Z(2), Z(5)	
q_{18}	q_1	NZ(2), NZ(4), Z(5)	S(2), S(4)
q_{18}	q_{20}	NZ(5)	S(5)
q_{18}	*Stop*	NZ(2), Z(4), Z(5)	S(2)
q_{20}	q_1	NZ(4), Z(5)	A(0), A(2), S(4)
q_{20}	q_{16}	NZ(5)	A(4), S(5)
q_{20}	*Stop*	Z(4), Z(5)	A(0), A(2)

The following is the program of the weakly universal 2-register machine U_2 simulating the weakly universal U_{20} from [71].

$(q_1, S(0), q_2, q_{25})$	$(q_2, S(0), q_3, q_{23})$	$(q_3, S(0), q_4, q_{22})$
$(q_4, S(0), q_5, q_{21})$	$(q_5, S(0), q_6, q_{20})$	$(q_6, S(0), q_7, q_{19})$
$(q_7, S(0), q_8, q_{18})$	$(q_8, S(0), q_9, q_{17})$	$(q_9, S(0), q_{10}, q_{16})$
$(q_{10}, S(0), q_{11}, q_{15})$	$(q_{11}, S(0), q_{12}, q_{14})$	$(q_{12}, A(1), q_1)$
$(q_{13}, A(0), q_{14})$	$(q_{14}, A(0), q_{15})$	$(q_{15}, A(0), q_{16})$
$(q_{16}, A(0), q_{17})$	$(q_{17}, A(0), q_{18})$	$(q_{18}, A(0), q_{19})$
$(q_{19}, A(0), q_{20})$	$(q_{20}, A(0), q_{21})$	$(q_{21}, A(0), q_{22})$
$(q_{22}, A(0), q_{23})$	$(q_{23}, A(0), q_{24})$	$(q_{24}, S(1), q_{13}, q_{55})$
$(q_{25}, S(1), q_{26}, q_{27})$	$(q_{26}, A(0), q_{25})$	$(q_{27}, S(0), q_{28}, q_{45})$
$(q_{28}, A(1), q_{29})$	$(q_{29}, A(1), q_{30})$	$(q_{30}, A(1), q_{31})$
$(q_{31}, A(1), q_{32})$	$(q_{32}, A(1), q_{33})$	$(q_{33}, A(1), q_{34})$
$(q_{34}, A(1), q_{35})$	$(q_{35}, A(1), q_{36})$	$(q_{36}, A(1), q_{37})$
$(q_{37}, A(1), q_{38})$	$(q_{38}, A(1), q_{39})$	$(q_{39}, A(1), q_{40})$
$(q_{40}, A(1), q_{41})$	$(q_{41}, A(1), q_{42})$	$(q_{42}, A(1), q_{43})$
$(q_{43}, A(1), q_{44})$	$(q_{44}, A(1), q_{27})$	$(q_{45}, S(1), q_{46}, q_1)$
$(q_{46}, A(0), q_{45})$	$(q_{47}, S(0), q_{48}, q_{53})$	$(q_{48}, S(0), q_{49}, q_{51})$
$(q_{49}, A(1), q_{47})$	$(q_{50}, A(0), q_{51})$	$(q_{51}, A(0), q_{52})$
$(q_{52}, S(1), q_{50}, q_{61})$	$(q_{53}, S(1), q_{54}, q_{55})$	$(q_{54}, A(0), q_{53})$
$(q_{55}, S(0), q_{56}, q_{59})$	$(q_{56}, A(1), q_{57})$	$(q_{57}, A(1), q_{58})$
$(q_{58}, A(1), q_{55})$	$(q_{59}, S(1), q_{60}, q_{47})$	$(q_{60}, A(0), q_{59})$
$(q_{61}, S(0), q_{62}, q_{69})$	$(q_{62}, S(0), q_{63}, q_{67})$	$(q_{63}, S(0), q_{64}, q_{66})$
$(q_{64}, A(1), q_{61})$	$(q_{65}, A(0), q_{66})$	$(q_{66}, A(0), q_{67})$
$(q_{67}, A(0), q_{68})$	$(q_{68}, S(1), q_{65}, q_{47})$	$(q_{69}, S(1), q_{70}, q_{71})$
$(q_{70}, A(0), q_{69})$	$(q_{71}, S(0), q_{72}, q_{74})$	$(q_{72}, A(1), q_{73})$
$(q_{73}, A(1), q_{71})$	$(q_{74}, S(1), q_{75}, q_{76})$	$(q_{75}, A(0), q_{74})$
$(q_{76}, S(0), q_{77}, q_{112})$	$(q_{77}, S(0), q_{78}, q_{110})$	$(q_{78}, S(0), q_{79}, q_{109})$
$(q_{79}, S(0), q_{80}, q_{108})$	$(q_{80}, S(0), q_{81}, q_{107})$	$(q_{81}, S(0), q_{82}, q_{106})$
$(q_{82}, S(0), q_{83}, q_{105})$	$(q_{83}, S(0), q_{84}, q_{104})$	$(q_{84}, S(0), q_{85}, q_{103})$
$(q_{85}, S(0), q_{86}, q_{102})$	$(q_{86}, S(0), q_{87}, q_{101})$	$(q_{87}, S(0), q_{88}, q_{100})$
$(q_{88}, S(0), q_{89}, q_{99})$	$(q_{89}, S(0), q_{90}, q_{98})$	$(q_{90}, S(0), q_{91}, q_{97})$
$(q_{91}, S(0), q_{92}, q_{96})$	$(q_{92}, S(0), q_{93}, q_{95})$	$(q_{93}, A(1), q_{76})$
$(q_{94}, A(0), q_{95})$	$(q_{95}, A(0), q_{96})$	$(q_{96}, A(0), q_{97})$
$(q_{97}, A(0), q_{98})$	$(q_{98}, A(0), q_{99})$	$(q_{99}, A(0), q_{100})$
$(q_{100}, A(0), q_{101})$	$(q_{101}, A(0), q_{102})$	$(q_{102}, A(0), q_{103})$
$(q_{103}, A(0), q_{104})$	$(q_{104}, A(0), q_{105})$	$(q_{105}, A(0), q_{106})$
$(q_{106}, A(0), q_{107})$	$(q_{107}, A(0), q_{108})$	$(q_{108}, A(0), q_{109})$
$(q_{109}, A(0), q_{110})$	$(q_{110}, A(0), q_{111})$	$(q_{111}, S(1), q_{94}, q_{128})$
$(q_{112}, S(1), q_{113}, q_{114})$	$(q_{113}, A(0), q_{112})$	$(q_{114}, S(0), q_{115}, q_{126})$
$(q_{115}, A(1), q_{116})$	$(q_{116}, A(1), q_{117})$	$(q_{117}, A(1), q_{118})$
$(q_{118}, A(1), q_{119})$	$(q_{119}, A(1), q_{120})$	$(q_{120}, A(1), q_{121})$
$(q_{121}, A(1), q_{122})$	$(q_{122}, A(1), q_{123})$	$(q_{123}, A(1), q_{124})$
$(q_{124}, A(1), q_{125})$	$(q_{125}, A(1), q_{114})$	$(q_{126}, S(1), q_{127}, q_{61})$
$(q_{127}, A(0), q_{126})$	$(q_{128}, S(0), q_{129}, q_{136})$	$(q_{129}, S(0), q_{130}, q_{134})$
$(q_{130}, S(0), q_{131}, q_{133})$	$(q_{131}, A(1), q_{128})$	$(q_{132}, A(0), q_{133})$
$(q_{133}, A(0), q_{134})$	$(q_{134}, A(0), q_{135})$	$(q_{135}, S(1), q_{132}, q_1)$
$(q_{136}, S(1), q_{137}, q_{272})$	$(q_{137}, A(0), q_{136})$	$(q_{138}, S(0), q_{139}, q_{150})$

$(q_{139}, S(0), q_{140}, q_{148})$	$(q_{140}, S(0), q_{141}, q_{147})$	$(q_{141}, S(0), q_{142}, q_{146})$
$(q_{142}, S(0), q_{143}, q_{145})$	$(q_{143}, A(1), q_{138})$	$(q_{144}, A(0), q_{145})$
$(q_{145}, A(0), q_{146})$	$(q_{146}, A(0), q_{147})$	$(q_{147}, A(0), q_{148})$
$(q_{148}, A(0), q_{149})$	$(q_{149}, S(1), q_{144}, q_{152})$	$(q_{150}, S(1), q_{151}, q_{1})$
$(q_{151}, A(0), q_{150})$	$(q_{152}, S(0), q_{153}, q_{158})$	$(q_{153}, S(0), q_{154}, q_{156})$
$(q_{154}, A(1), q_{152})$	$(q_{155}, A(0), q_{156})$	$(q_{156}, A(0), q_{157})$
$(q_{157}, S(1), q_{155}, q_{184})$	$(q_{158}, S(1), q_{159}, q_{160})$	$(q_{159}, A(0), q_{158})$
$(q_{160}, S(0), q_{161}, q_{166})$	$(q_{161}, S(0), q_{162}, q_{164})$	$(q_{162}, A(1), q_{160})$
$(q_{163}, A(0), q_{164})$	$(q_{164}, A(0), q_{165})$	$(q_{165}, S(1), q_{163}, q_{202})$
$(q_{166}, S(1), q_{167}, q_{168})$	$(q_{167}, A(0), q_{166})$	$(q_{168}, S(0), q_{169}, q_{174})$
$(q_{169}, S(0), q_{170}, q_{172})$	$(q_{170}, A(1), q_{168})$	$(q_{171}, A(0), q_{172})$
$(q_{172}, A(0), q_{173})$	$(q_{173}, S(1), q_{171}, q_{232})$	$(q_{174}, S(1), q_{175}, q_{176})$
$(q_{175}, A(0), q_{174})$	$(q_{176}, S(0), q_{177}, q_{182})$	$(q_{177}, A(1), q_{178})$
$(q_{178}, A(1), q_{179})$	$(q_{179}, A(1), q_{180})$	$(q_{180}, A(1), q_{181})$
$(q_{181}, A(1), q_{176})$	$(q_{182}, S(1), q_{183}, q_{152})$	$(q_{183}, A(0), q_{182})$
$(q_{184}, S(0), q_{185}, q_{200})$	$(q_{185}, S(0), q_{186}, q_{198})$	$(q_{186}, S(0), q_{187}, q_{197})$
$(q_{187}, S(0), q_{188}, q_{196})$	$(q_{188}, S(0), q_{189}, q_{195})$	$(q_{189}, S(0), q_{190}, q_{194})$
$(q_{190}, S(0), q_{191}, q_{193})$	$(q_{191}, A(1), q_{184})$	$(q_{192}, A(0), q_{193})$
$(q_{193}, A(0), q_{194})$	$(q_{194}, A(0), q_{195})$	$(q_{195}, A(0), q_{196})$
$(q_{196}, A(0), q_{197})$	$(q_{197}, A(0), q_{198})$	$(q_{198}, A(0), q_{199})$
$(q_{199}, S(1), q_{192}, q_{1})$	$(q_{200}, S(1), q_{201}, q_{258})$	$(q_{201}, A(0), q_{200})$
$(q_{202}, S(0), q_{203}, q_{230})$	$(q_{203}, S(0), q_{204}, q_{228})$	$(q_{204}, S(0), q_{205}, q_{227})$
$(q_{205}, S(0), q_{206}, q_{226})$	$(q_{206}, S(0), q_{207}, q_{225})$	$(q_{207}, S(0), q_{208}, q_{224})$
$(q_{208}, S(0), q_{209}, q_{223})$	$(q_{209}, S(0), q_{210}, q_{222})$	$(q_{210}, S(0), q_{211}, q_{221})$
$(q_{211}, S(0), q_{212}, q_{220})$	$(q_{212}, S(0), q_{213}, q_{219})$	$(q_{213}, S(0), q_{214}, q_{218})$
$(q_{214}, S(0), q_{215}, q_{217})$	$(q_{215}, A(1), q_{202})$	$(q_{216}, A(0), q_{217})$
$(q_{217}, A(0), q_{218})$	$(q_{218}, A(0), q_{219})$	$(q_{219}, A(0), q_{220})$
$(q_{220}, A(0), q_{221})$	$(q_{221}, A(0), q_{222})$	$(q_{222}, A(0), q_{223})$
$(q_{223}, A(0), q_{224})$	$(q_{224}, A(0), q_{225})$	$(q_{225}, A(0), q_{226})$
$(q_{226}, A(0), q_{227})$	$(q_{227}, A(0), q_{228})$	$(q_{228}, A(0), q_{229})$
$(q_{229}, S(1), q_{216}, q_{1})$	$(q_{230}, S(1), q_{231}, q_{258})$	$(q_{231}, A(0), q_{230})$
$(q_{232}, S(0), q_{233}, q_{240})$	$(q_{233}, A(1), q_{234})$	$(q_{234}, A(1), q_{235})$
$(q_{235}, A(1), q_{236})$	$(q_{236}, A(1), q_{237})$	$(q_{237}, A(1), q_{238})$
$(q_{238}, A(1), q_{239})$	$(q_{239}, A(1), q_{232})$	$(q_{240}, S(1), q_{241}, q_{242})$
$(q_{241}, A(0), q_{240})$	$(q_{242}, S(0), q_{243}, q_{256})$	$(q_{243}, A(1), q_{244})$
$(q_{244}, A(1), q_{245})$	$(q_{245}, A(1), q_{246})$	$(q_{246}, A(1), q_{247})$
$(q_{247}, A(1), q_{248})$	$(q_{248}, A(1), q_{249})$	$(q_{249}, A(1), q_{250})$
$(q_{250}, A(1), q_{251})$	$(q_{251}, A(1), q_{252})$	$(q_{252}, A(1), q_{253})$
$(q_{253}, A(1), q_{254})$	$(q_{254}, A(1), q_{255})$	$(q_{255}, A(1), q_{242})$
$(q_{256}, S(1), q_{257}, q_{258})$	$(q_{257}, A(0), q_{256})$	$(q_{258}, S(0), q_{259}, q_{270})$
$(q_{259}, S(0), q_{260}, q_{268})$	$(q_{260}, S(0), q_{261}, q_{267})$	$(q_{261}, S(0), q_{262}, q_{266})$
$(q_{262}, S(0), q_{263}, q_{265})$	$(q_{263}, A(1), q_{258})$	$(q_{264}, A(0), q_{265})$
$(q_{265}, A(0), q_{266})$	$(q_{266}, A(0), q_{267})$	$(q_{267}, A(0), q_{268})$
$(q_{268}, A(0), q_{269})$	$(q_{269}, S(1), q_{264}, q_{278})$	$(q_{270}, S(1), q_{271}, q_{1})$
$(q_{271}, A(0), q_{270})$	$(q_{272}, S(0), q_{273}, q_{276})$	$(q_{273}, A(1), q_{274})$
$(q_{274}, A(1), q_{275})$	$(q_{275}, A(1), q_{272})$	$(q_{276}, S(1), q_{277}, q_{138})$
$(q_{277}, A(0), q_{276})$	$(q_{278}, Stop)$	

The following is the program of the strongly universal 3-register machine U_3 simulating the strongly universal U_{22} from [71].

$(q_1, S(2), q_2, q_{16})$	$(q_2, S(0), q_3, q_{14})$	$(q_3, A(1), q_4)$
$(q_4, A(1), q_5)$	$(q_5, A(1), q_6)$	$(q_6, A(1), q_7)$
$(q_7, A(1), q_8)$	$(q_8, A(1), q_9)$	$(q_9, A(1), q_{10})$
$(q_{10}, A(1), q_{11})$	$(q_{11}, A(1), q_{12})$	$(q_{12}, A(1), q_{13})$
$(q_{13}, A(1), q_2)$	$(q_{14}, S(1), q_{14}, q_1)$	$(q_{15}, A(0), q_{14})$
$(q_{16}, S(0), q_{17}, q_{44})$	$(q_{17}, S(0), q_{18}, q_{42})$	$(q_{18}, S(0), q_{19}, q_{41})$
$(q_{19}, S(0), q_{20}, q_{40})$	$(q_{20}, S(0), q_{21}, q_{39})$	$(q_{21}, S(0), q_{22}, q_{38})$
$(q_{22}, S(0), q_{23}, q_{37})$	$(q_{23}, S(0), q_{24}, q_{36})$	$(q_{24}, S(0), q_{25}, q_{35})$
$(q_{25}, S(0), q_{26}, q_{34})$	$(q_{26}, S(0), q_{27}, q_{33})$	$(q_{27}, S(0), q_{28}, q_{32})$
$(q_{28}, S(0), q_{29}, q_{31})$	$(q_{29}, A(1), q_{16})$	$(q_{30}, A(0), q_{31})$
$(q_{31}, A(0), q_{32})$	$(q_{32}, A(0), q_{33})$	$(q_{33}, A(0), q_{34})$
$(q_{34}, A(0), q_{35})$	$(q_{35}, A(0), q_{36})$	$(q_{36}, A(0), q_{37})$
$(q_{37}, A(0), q_{38})$	$(q_{38}, A(0), q_{39})$	$(q_{39}, A(0), q_{40})$
$(q_{40}, A(0), q_{41})$	$(q_{41}, A(0), q_{42})$	$(q_{42}, A(0), q_{43})$
$(q_{43}, S(1), q_{30}, q_{76})$	$(q_{44}, S(1), q_{45}, q_{46})$	$(q_{45}, A(0), q_{44})$
$(q_{46}, S(0), q_{47}, q_{66})$	$(q_{47}, A(1), q_{48})$	$(q_{48}, A(1), q_{49})$
$(q_{49}, A(1), q_{50})$	$(q_{50}, A(1), q_{51})$	$(q_{51}, A(1), q_{52})$
$(q_{52}, A(1), q_{53})$	$(q_{53}, A(1), q_{54})$	$(q_{54}, A(1), q_{55})$
$(q_{55}, A(1), q_{56})$	$(q_{56}, A(1), q_{57})$	$(q_{57}, A(1), q_{58})$
$(q_{58}, A(1), q_{59})$	$(q_{59}, A(1), q_{60})$	$(q_{60}, A(1), q_{61})$
$(q_{61}, A(1), q_{62})$	$(q_{62}, A(1), q_{63})$	$(q_{63}, A(1), q_{64})$
$(q_{64}, A(1), q_{65})$	$(q_{65}, A(1), q_{46})$	$(q_{66}, S(1), q_{67}, q_{16})$
$(q_{67}, A(0), q_{66})$	$(q_{68}, S(0), q_{69}, q_{74})$	$(q_{69}, S(0), q_{70}, q_{72})$
$(q_{70}, A(1), q_{68})$	$(q_{71}, A(0), q_{72})$	$(q_{72}, A(0), q_{73})$
$(q_{73}, S(1), q_{71}, q_{82})$	$(q_{74}, S(1), q_{75}, q_{76})$	$(q_{75}, A(0), q_{74})$
$(q_{76}, S(0), q_{77}, q_{80})$	$(q_{77}, A(1), q_{78})$	$(q_{78}, A(1), q_{79})$
$(q_{79}, A(1), q_{76})$	$(q_{80}, S(1), q_{81}, q_{68})$	$(q_{81}, A(0), q_{80})$
$(q_{82}, S(0), q_{83}, q_{90})$	$(q_{83}, S(0), q_{84}, q_{88})$	$(q_{84}, S(0), q_{85}, q_{87})$
$(q_{85}, A(1), q_{82})$	$(q_{86}, A(0), q_{87})$	$(q_{87}, A(0), q_{88})$
$(q_{88}, A(0), q_{89})$	$(q_{89}, S(1), q_{86}, q_{68})$	$(q_{90}, S(1), q_{91}, q_{92})$
$(q_{91}, A(0), q_{90})$	$(q_{92}, S(0), q_{93}, q_{95})$	$(q_{93}, A(1), q_{94})$
$(q_{94}, A(1), q_{92})$	$(q_{95}, S(1), q_{96}, q_{97})$	$(q_{96}, A(0), q_{95})$
$(q_{97}, S(0), q_{98}, q_{137})$	$(q_{98}, S(0), q_{99}, q_{135})$	$(q_{99}, S(0), q_{100}, q_{134})$
$(q_{100}, S(0), q_{101}, q_{133})$	$(q_{101}, S(0), q_{102}, q_{132})$	$(q_{102}, S(0), q_{103}, q_{131})$
$(q_{103}, S(0), q_{104}, q_{130})$	$(q_{104}, S(0), q_{105}, q_{129})$	$(q_{105}, S(0), q_{106}, q_{128})$
$(q_{106}, S(0), q_{107}, q_{127})$	$(q_{107}, S(0), q_{108}, q_{126})$	$(q_{108}, S(0), q_{109}, q_{125})$
$(q_{109}, S(0), q_{110}, q_{124})$	$(q_{110}, S(0), q_{111}, q_{123})$	$(q_{111}, S(0), q_{112}, q_{122})$
$(q_{112}, S(0), q_{113}, q_{121})$	$(q_{113}, S(0), q_{114}, q_{120})$	$(q_{114}, S(0), q_{115}, q_{119})$
$(q_{115}, S(0), q_{116}, q_{118})$	$(q_{116}, A(1), q_{97})$	$(q_{117}, A(0), q_{118})$
$(q_{118}, A(0), q_{119})$	$(q_{119}, A(0), q_{120})$	$(q_{120}, A(0), q_{121})$
$(q_{121}, A(0), q_{122})$	$(q_{122}, A(0), q_{123})$	$(q_{123}, A(0), q_{124})$
$(q_{124}, A(0), q_{125})$	$(q_{125}, A(0), q_{126})$	$(q_{126}, A(0), q_{127})$
$(q_{127}, A(0), q_{128})$	$(q_{128}, A(0), q_{129})$	$(q_{129}, A(0), q_{130})$
$(q_{130}, A(0), q_{131})$	$(q_{131}, A(0), q_{132})$	$(q_{132}, A(0), q_{133})$
$(q_{133}, A(0), q_{134})$	$(q_{134}, A(0), q_{135})$	$(q_{135}, A(0), q_{136})$
$(q_{136}, S(1), q_{117}, q_{155})$	$(q_{137}, S(1), q_{138}, q_{139})$	$(q_{138}, A(0), q_{137})$

$(q_{139}, S(0), q_{140}, q_{153})$	$(q_{140}, A(1), q_{141})$	$(q_{141}, A(1), q_{142})$
$(q_{142}, A(1), q_{143})$	$(q_{143}, A(1), q_{144})$	$(q_{144}, A(1), q_{145})$
$(q_{145}, A(1), q_{146})$	$(q_{146}, A(1), q_{147})$	$(q_{147}, A(1), q_{148})$
$(q_{148}, A(1), q_{149})$	$(q_{149}, A(1), q_{150})$	$(q_{150}, A(1), q_{151})$
$(q_{151}, A(1), q_{152})$	$(q_{152}, A(1), q_{139})$	$(q_{153}, S(1), q_{154}, q_{82})$
$(q_{154}, A(0), q_{153})$	$(q_{155}, S(0), q_{156}, q_{163})$	$(q_{156}, S(0), q_{157}, q_{161})$
$(q_{157}, S(0), q_{158}, q_{160})$	$(q_{158}, A(1), q_{155})$	$(q_{159}, A(0), q_{160})$
$(q_{160}, A(0), q_{161})$	$(q_{161}, A(0), q_{162})$	$(q_{162}, S(1), q_{159}, q_{16})$
$(q_{163}, S(1), q_{164}, q_{351})$	$(q_{164}, A(0), q_{163})$	$(q_{165}, S(0), q_{166}, q_{177})$
$(q_{166}, S(0), q_{167}, q_{175})$	$(q_{167}, S(0), q_{168}, q_{174})$	$(q_{168}, S(0), q_{169}, q_{173})$
$(q_{169}, S(0), q_{170}, q_{172})$	$(q_{170}, A(1), q_{165})$	$(q_{171}, A(0), q_{172})$
$(q_{172}, A(0), q_{173})$	$(q_{173}, A(0), q_{174})$	$(q_{174}, A(0), q_{175})$
$(q_{175}, A(0), q_{176})$	$(q_{176}, S(1), q_{171}, q_{179})$	$(q_{177}, S(1), q_{178}, q_{16})$
$(q_{178}, A(0), q_{177})$	$(q_{179}, S(0), q_{180}, q_{185})$	$(q_{180}, S(0), q_{181}, q_{183})$
$(q_{181}, A(1), q_{179})$	$(q_{182}, A(0), q_{183})$	$(q_{183}, A(0), q_{184})$
$(q_{184}, S(1), q_{182}, q_{211})$	$(q_{185}, S(1), q_{186}, q_{187})$	$(q_{186}, A(0), q_{185})$
$(q_{187}, S(0), q_{188}, q_{193})$	$(q_{188}, S(0), q_{189}, q_{191})$	$(q_{189}, A(1), q_{187})$
$(q_{190}, A(0), q_{191})$	$(q_{191}, A(0), q_{192})$	$(q_{192}, S(1), q_{190}, q_{255})$
$(q_{193}, S(1), q_{194}, q_{195})$	$(q_{194}, A(0), q_{193})$	$(q_{195}, S(0), q_{196}, q_{201})$
$(q_{196}, S(0), q_{197}, q_{199})$	$(q_{197}, A(1), q_{195})$	$(q_{198}, A(0), q_{199})$
$(q_{199}, A(0), q_{200})$	$(q_{200}, S(1), q_{198}, q_{303})$	$(q_{201}, S(1), q_{202}, q_{203})$
$(q_{202}, A(0), q_{201})$	$(q_{203}, S(0), q_{204}, q_{209})$	$(q_{204}, A(1), q_{205})$
$(q_{205}, A(1), q_{206})$	$(q_{206}, A(1), q_{207})$	$(q_{207}, A(1), q_{208})$
$(q_{208}, A(1), q_{203})$	$(q_{209}, S(1), q_{210}, q_{179})$	$(q_{210}, A(0), q_{209})$
$(q_{211}, S(0), q_{212}, q_{235})$	$(q_{212}, S(0), q_{213}, q_{233})$	$(q_{213}, S(0), q_{214}, q_{232})$
$(q_{214}, S(0), q_{215}, q_{231})$	$(q_{215}, S(0), q_{216}, q_{230})$	$(q_{216}, S(0), q_{217}, q_{229})$
$(q_{217}, S(0), q_{218}, q_{228})$	$(q_{218}, S(0), q_{219}, q_{227})$	$(q_{219}, S(0), q_{220}, q_{226})$
$(q_{220}, S(0), q_{221}, q_{225})$	$(q_{221}, S(0), q_{222}, q_{224})$	$(q_{222}, A(1), q_{211})$
$(q_{223}, A(0), q_{224})$	$(q_{224}, A(0), q_{225})$	$(q_{225}, A(0), q_{226})$
$(q_{226}, A(0), q_{227})$	$(q_{227}, A(0), q_{228})$	$(q_{228}, A(0), q_{229})$
$(q_{229}, A(0), q_{230})$	$(q_{230}, A(0), q_{231})$	$(q_{231}, A(0), q_{232})$
$(q_{232}, A(0), q_{233})$	$(q_{233}, A(0), q_{234})$	$(q_{234}, S(1), q_{223}, q_{237})$
$(q_{235}, S(1), q_{236}, q_{337})$	$(q_{236}, A(0), q_{235})$	$(q_{237}, S(0), q_{238}, q_{253})$
$(q_{238}, S(0), q_{239}, q_{251})$	$(q_{239}, S(0), q_{240}, q_{250})$	$(q_{240}, S(0), q_{241}, q_{249})$
$(q_{241}, S(0), q_{242}, q_{248})$	$(q_{242}, S(0), q_{243}, q_{247})$	$(q_{243}, S(0), q_{244}, q_{246})$
$(q_{244}, A(1), q_{237})$	$(q_{245}, A(0), q_{246})$	$(q_{246}, A(0), q_{247})$
$(q_{247}, A(0), q_{248})$	$(q_{248}, A(0), q_{249})$	$(q_{249}, A(0), q_{250})$
$(q_{250}, A(0), q_{251})$	$(q_{251}, A(0), q_{252})$	$(q_{252}, S(1), q_{245}, q_{337})$
$(q_{253}, S(1), q_{254}, q_{16})$	$(q_{254}, A(0), q_{253})$	$(q_{255}, S(0), q_{256}, q_{291})$
$(q_{256}, S(0), q_{257}, q_{289})$	$(q_{257}, S(0), q_{258}, q_{288})$	$(q_{258}, S(0), q_{259}, q_{287})$
$(q_{259}, S(0), q_{260}, q_{286})$	$(q_{260}, S(0), q_{261}, q_{285})$	$(q_{261}, S(0), q_{262}, q_{284})$
$(q_{262}, S(0), q_{263}, q_{283})$	$(q_{263}, S(0), q_{264}, q_{282})$	$(q_{264}, S(0), q_{265}, q_{281})$
$(q_{265}, S(0), q_{266}, q_{280})$	$(q_{266}, S(0), q_{267}, q_{279})$	$(q_{267}, S(0), q_{268}, q_{278})$
$(q_{268}, S(0), q_{269}, q_{277})$	$(q_{269}, S(0), q_{270}, q_{276})$	$(q_{270}, S(0), q_{271}, q_{275})$
$(q_{271}, S(0), q_{272}, q_{274})$	$(q_{272}, A(1), q_{255})$	$(q_{273}, A(0), q_{274})$
$(q_{274}, A(0), q_{275})$	$(q_{275}, A(0), q_{276})$	$(q_{276}, A(0), q_{277})$
$(q_{277}, A(0), q_{278})$	$(q_{278}, A(0), q_{279})$	$(q_{279}, A(0), q_{280})$
$(q_{280}, A(0), q_{281})$	$(q_{281}, A(0), q_{282})$	$(q_{282}, A(0), q_{283})$

$(q_{283}, A(0), q_{284})$	$(q_{284}, A(0), q_{285})$	$(q_{285}, A(0), q_{286})$
$(q_{286}, A(0), q_{287})$	$(q_{287}, A(0), q_{288})$	$(q_{288}, A(0), q_{289})$
$(q_{289}, A(0), q_{290})$	$(q_{290}, S(1), q_{273}, q_{293})$	$(q_{291}, S(1), q_{292}, q_{337})$
$(q_{292}, A(0), q_{291})$	$(q_{293}, S(0), q_{294}, q_{301})$	$(q_{294}, A(1), q_{295})$
$(q_{295}, A(1), q_{296})$	$(q_{296}, A(1), q_{297})$	$(q_{297}, A(1), q_{298})$
$(q_{298}, A(1), q_{299})$	$(q_{299}, A(1), q_{300})$	$(q_{300}, A(1), q_{293})$
$(q_{301}, S(1), q_{302}, q_{16})$	$(q_{302}, A(0), q_{301})$	$(q_{303}, S(0), q_{304}, q_{315})$
$(q_{304}, A(1), q_{305})$	$(q_{305}, A(1), q_{306})$	$(q_{306}, A(1), q_{307})$
$(q_{307}, A(1), q_{308})$	$(q_{308}, A(1), q_{309})$	$(q_{309}, A(1), q_{310})$
$(q_{310}, A(1), q_{311})$	$(q_{311}, A(1), q_{312})$	$(q_{312}, A(1), q_{313})$
$(q_{313}, A(1), q_{314})$	$(q_{314}, A(1), q_{303})$	$(q_{315}, S(1), q_{316}, q_{317})$
$(q_{316}, A(0), q_{315})$	$(q_{317}, S(0), q_{318}, q_{335})$	$(q_{318}, A(1), q_{319})$
$(q_{319}, A(1), q_{320})$	$(q_{320}, A(1), q_{321})$	$(q_{321}, A(1), q_{322})$
$(q_{322}, A(1), q_{323})$	$(q_{323}, A(1), q_{324})$	$(q_{324}, A(1), q_{325})$
$(q_{325}, A(1), q_{326})$	$(q_{326}, A(1), q_{327})$	$(q_{327}, A(1), q_{328})$
$(q_{328}, A(1), q_{329})$	$(q_{329}, A(1), q_{330})$	$(q_{330}, A(1), q_{331})$
$(q_{331}, A(1), q_{332})$	$(q_{332}, A(1), q_{333})$	$(q_{333}, A(1), q_{334})$
$(q_{334}, A(1), q_{317})$	$(q_{335}, S(1), q_{336}, q_{337})$	$(q_{336}, A(0), q_{335})$
$(q_{337}, S(0), q_{338}, q_{349})$	$(q_{338}, S(0), q_{339}, q_{347})$	$(q_{339}, S(0), q_{340}, q_{346})$
$(q_{340}, S(0), q_{341}, q_{345})$	$(q_{341}, S(0), q_{342}, q_{344})$	$(q_{342}, A(1), q_{337})$
$(q_{343}, A(0), q_{344})$	$(q_{344}, A(0), q_{345})$	$(q_{345}, A(0), q_{346})$
$(q_{346}, A(0), q_{347})$	$(q_{347}, A(0), q_{348})$	$(q_{348}, S(1), q_{343}, q_{357})$
$(q_{349}, S(1), q_{350}, q_{16})$	$(q_{350}, A(0), q_{349})$	$(q_{351}, S(0), q_{352}, q_{355})$
$(q_{352}, A(1), q_{353})$	$(q_{353}, A(1), q_{354})$	$(q_{354}, A(1), q_{351})$
$(q_{355}, S(1), q_{356}, q_{165})$	$(q_{356}, A(0), q_{355})$	$(q_{357}, S(0), q_{358}, q_{365})$
$(q_{358}, S(0), q_{359}, q_{368})$	$(q_{359}, S(0), q_{360}, q_{368})$	$(q_{360}, S(0), q_{361}, q_{368})$
$(q_{361}, S(0), q_{362}, q_{368})$	$(q_{362}, S(0), q_{363}, q_{368})$	$(q_{363}, S(0), q_{364}, q_{368})$
$(q_{364}, A(1), q_{357})$	$(q_{365}, S(1), q_{366}, q_{367})$	$(q_{366}, A(0), q_{365})$
$(q_{367}, A(2), q_{357})$	$(q_{368}, Stop)$	