Automatic Generation of Bulk-Synchronous Parallel code

Thibaut Tachon

UNIVERSITÉ D'ORLÉANS

ÉCOLE DOCTORALE MATHÉMATIQUES, INFORMATIQUE, PHYSIQUE THÉORIQUE ET INGÉNIERIE DES SYSTÈMES

LABORATOIRE D'INFORMATIQUE FONDAMENTALE D'ORLÉANS
HUAWEI PARIS RESEARCH CENTER

THÈSE présentée par :

Thibaut TACHON

soutenue le : 28 Juin Décembre 2019

pour obtenir le grade de : **Docteur de l'université d'Orléans**

Discipline : **Informatique**

Génération automatique de code parallèle isochrone

THÈSE DIRIGÉE PAR :

Frédéric LOULERGUE	Professeur, Northern Arizona University et Université d'Orléans

RAPPORTEURS :

Franck POMMEREAU	Professeur, Université d'Évry
John MULLINS	Professeur, École Polytechnique de Montréal

JURY :

Gaétan HAINS	Ingénieur-Chercheur, Huawei Technologies
Wijnand SUIJLEN	Ingénieur-Chercheur, Huawei Technologies
Jean-Michel COUVREUR	Professeur, Université d'Orléans

Résumé étendu en Français

Introduction

Un monde parallèle

Les ordinateurs parallèles sont partout aujourd'hui depuis les montres connectées à deux coeurs jusqu'aux super-ordinateurs aux millions de coeurs. La fréquence d'horloge des processeurs a stagné pendant des années et l'accroissement du nombre de processeurs ou coeurs est devenu le seul moyen pour augmenter la puissance de calcul. Programmer un processeur (*i.e. programmation séquentielle*) est difficile. Programmer plusieurs processeurs (*i.e. programmation parallèle*) demande de programmer non seulement leurs tâches locales mais aussi leurs communications. Cela demande plus de travail et procure donc davantage d'opportunité aux erreurs. Les conséquences des erreurs de programmation varient d'une altération imperceptible de la couleur d'un pixel jusqu'à l'explosion d'une fusée. Il n'est donc pas exagéré de dire que l'aide aux programmeurs peut être une question de vie ou de mort. La programmation parallèle est en général plus dure que la programmation séquentielle et réclame donc encore davantage d'aide. Par exemple, une erreur bien connue de la programmation parallèle est l'interblocage, une erreur arrivant quand des processeurs s'attendent les uns les autres pour toujours. Plusieurs approches existent pour aider les programmeurs parallèles à éviter ces erreurs.

Rendre le monde parallèle meilleur

Les modèles tel que le modèle parallèle isochrone BSP [60] restreint le programmeur pour l'empêcher de commettre de nombreuses erreurs. Par exemple, si le modèle est respecté, il force tous les processeurs à synchroniser en même temps, prévenant ainsi les risques d'interblocage. Restreindre le programmeur peut néanmoins coûter cher en performance. Ce compromis entre sûreté

et performance est analogue à l'efficacité des langages de bas niveau (*e.g.* assembleur ou C) contre la sûreté apportée par les langages (bien fait) de haut niveau (*e.g.* OCaml).

La vérification cherche à analyser des programmes pour dire au programmeurs les erreurs qu'il a commise. La vérification peut analyser des programmes écrits statiquement ou des programmes en exécution dynamiquement. Cette méthode peut être ajoutée aux précédentes pour vérifier des propriétés qui ne sont pas assurées par le modèle. Il peut aussi vérifier le respect du modèle, par exemple, si tous les processeurs synchronisent au même moment [27], ce qui est une condition nécessaire à BSP.

La certification prouve qu'un programme satisfait une spécification [42]. La spécification d'un programme est la description mathématique des conditions nécessaires qu'un programme doit remplir pour être correct. Cette approche est la seule qui assure l'absence complète d'erreurs dans le programme écrit. Cette méthode réclame cependant un travail considérable, même avec des outils adaptés, comme Coq. Coq [53] est un assistant de preuve pouvant être utilisé pour la preuve de théories mathématiques mais est aussi un langage de programmation depuis lequel un programme certifié peut être extrait.

La génération de programmes (*i.e.* code) depuis un langage de haut niveau procure à la fois la sûreté du langage source et les performances du langage cible. Les compilateurs sont généralement responsables de cette traduction et apportent souvent quelque vérification statique avant la génération de code. Par exemple, BSML est une bibliothèque BSP de haut niveau, sans interblocage, pouvant être compilés vers une bibliothèque efficace de bas niveau telle que MPI. MPI ne prévient normalement pas l'apparition d'interblocage mais il le fait quand il est généré depuis BSML. Quelques exemples de compilateurs certifiés existent qui garantissent que le programme compilé préserve la sémantique du programme source.

Portes vers le monde parallèle

Nous avons développé cette dernière approche pour générer du code parallèle BSP depuis des programmes séquentiels. Avoir le modèle parallèle BSP en tant que cible, rend notre génération plus facile de la même façon que les modèles facilitent la programmation parallèle. BSP apporte aussi un modèle de coût pour

anticiper le temps d'exécution d'un programme qui présente une valeur ajoutée intéressante à notre travail. La certification est trop fastidieuse pour être traitée durant les phases initiales de la conception et la ralentirait considérablement. Cependant, nous avons toujours pris en compte la certification future de nos approches.

La génération automatique de code séquentiel à usage général est un problème profond et difficile mais en restreindre l'application nous permet d'être plus efficace. Ainsi, deux domaines spécifiques furent ciblés avec différentes techniques utilisées pour chacun d'entre eux.

Automates BSP pour la recherche par expression régulière

Les automates BSP (BSPA) et expressions régulières BSP (BSPRE) furent introduites par Hains [19] en 2016. Nous avons développé cette théorie à travers trois transformations principales qui manquaient à la théorie jusqu'alors.

La génération d'automates BSP à partir d'expressions régulières BSP. Cette transformation est analogue à la transformation fondamentale depuis des expressions régulières vers des automates finis non-déterministes reconnaissant le même langage [59, 7, 6]. Il peut être compris comme la compilation de BSPRE, représentant un langage BSP, vers un BSPA, une machine reconnaissant les mots BSP appartenant à ce langage.

La déterminisation des automates BSP pour les rendre plus efficaces. Les automates et les automates BSP sont en général non déterministes. Cela veut dire que plusieurs transitions avec la même étiquette depuis le même état existent, ce qui donne naissance à une ambiguïté. Pendant la recherche, cette ambiguïté force un choix qui, si faux, force à revenir en arrière, perdant un temps précieux pour des applications telle que la recherche par expression régulière. Dans notre cas, les BSPA non-déterministes peuvent être vus comme des machines à état finis abstraites et leur déterminisation en BSPA déterministe est une compilation en code parallèle exécutable.

La parallélisation d'expression régulière en expression régulière BSP. Cette transformation est dépendante de la distribution des données vers les processeurs. Elle calcule une BSPRE prenant en compte toute les divisions possibles des données d'entrée durant la distribution. Étant donné une représentation

séquentielle, cette transformation retourne une représentation BSP qui constitue la première étape nécessaire à une compilation vers du code BSP.

La composition des trois transformations précédentes qui commence par la parallélisation d'expression régulière en BSPRE, suivie par la génération de BSPA non déterministe terminée par déterminisation en BSPA déterministe permet la recherche à l'aide d'expression régulière en parallèle.

Langage spécifique pour les calculs de tenseur

Nous nous sommes intéressés aux réseaux de neurones, un domaine recevant en ce moment beaucoup de concentration et de contribution. Les réseaux de neurones sont généralement basés sur les graphes de calculs implémentés avec des tenseurs (*i.e.* tableaux multidimensionnels). Des travaux récents tels que Relay [54] remplacent ces graphes de calcul par un langage de programmation. Cependant, les travaux existants présentent selon nous trop de fonctionnalités, rendant les méthodes formelles (*e.g.* vérification et certification) plus difficiles à mettre en oeuvre. Nous avons conçu un petit langage spécifique avec peu de primitives dont la composition fonctionnelle est suffisante pour ce domaine. Ce langage est compilé vers du code BSP et constitue ce faisant un autre exemple de code parallèle BSP généré depuis un programme séquentiel.

Publications

Cette section présente nos articles acceptés:

1. Thibaut Tachon, Chong Li, Gaétan Hains, and Frédéric Loulergue. Automated generation of BSP automata. *Parallel Processing Letters*, 27(01):1740002, 2017. DOI 10.1142/S0129626417400023.

2. Gaétan Hains, Thibaut Tachon and Youry Khmelevsky. From natural language to graph queries. In *The annual IEEE Canadian conference on Electrical and computer engineering (CCECE)*, 2019. En cour de publication.

3. Thibaut Tachon. Parallel Matching of Regular Expressions with BSP Automata. In *The International Symposium on Formal Approaches to Parallel and Distributed Systems (4PAD)*, 2019. En cour de publication.

Bien que l'article 2 peut aussi être considéré comme une compilation d'un langage séquentiel (langage naturel) vers un programme BSP (parce qu'il existe une implantation BSP pour le traitement de requête de graphe), ce papier présente principalement des travaux d'étudiants et sa contribution est davantage d'ordre pédagogique que technique. Ainsi, ce papier n'a pas été étendu par le document présent.

Organisation

Le résumé est organisé comme suit.

Nous commençons par présenter la création d'automates BSP. Une preuve mathématique informelle est donné en annexe A. Bien que nous n'ayons pas eu le temps de compléter cette preuve pour tous les cas, nous en avons complété suffisamment pour couvrir tous les cas nécessité par l'application de BSPRE pour la recherche parallèle par RE. Nous poursuivons avec la description détaillée de la Déterminisation d'automates BSP, succédé par l'application de BSPRE pour la recherche parallèle par RE. Cette application inclue la parallélisation d'expression régulière et l'implantation de la composition des trois transformations qui est évaluée par rapport à une approche standard de la recherche parallèle par expression régulière.

Nous étudions ensuite la Programmation de tenseurs avec BSP. Nous concluons ce document et présentons les travaux futur.

Des expressions régulières BSP aux automates BSP

Cette section introduit l'algorithme de génération de BSPA acceptant le même langage qu'une BSPRE donnée. Cette transformation est fondamentale pour la théorie des automates BSP puisqu'elle permet de construire un automate BSP dont le langage est connu. Elle est à la théorie des automates BSP ce qu'est l'algorithme de Thompson [59] ou celui de Glushkov [8] à la théorie des automates finis. Cette transformation est aussi le premier pas de notre système de reconnaissance parallèle par expression régulière.

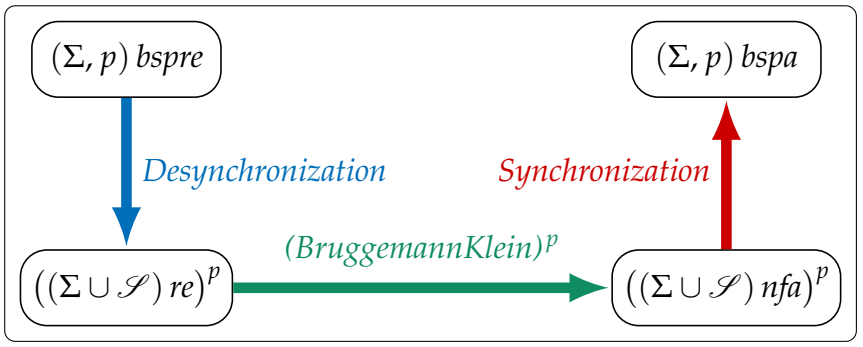


Figure 1 – *Schéma de la transformation de BSPRE à BSPA*

Cet algorithme est divisé en trois parties représentées dans la fig. 1, chacune décrite respectivement dans les sections Désynchronisation, D'expression régulière à automate fini et Synchronisation.

Désynchronisation

Cette transformation transforme une BSPRE dont l'alphabet est Σ et la longueur de ses vecteurs est p en un seul vecteur de longueur p dont les éléments sont des expressions régulières comportant la structure globale des BSPRE répliquée ainsi que la structure locale respective des RE. La différence entre structure locale et globale est marquée par un point-virgule concaténé à la partie locale. Ce point-virgule est annoté par l'identifiant du vecteur dont il est issu. L'ensemble de ces point-virgules annotés est noté $\mathscr{S}$. Ainsi, le type de sortie est un vecteur de taille p d'expressions régulières dont l'alphabet est celui d'origine élargit par les point-virgules.

Exemple 1 : Nous choisissons cette BSPRE simple en entrée de notre algorithme pour l'illustrer.

$$R = (\langle a\ ,\ b\rangle + \langle c\ ,\ d\rangle)\langle e\ ,\ f\rangle$$

Le langage BSP de cette expression est $\{\langle a,b\rangle\langle e,f\rangle\ ,\ \langle c,d\rangle\langle e,f\rangle\}$ et l'application de la désynchronisation à R résulte en

$$Dsn(R) = \langle (a\ ;_0 + c\ ;_1)\ e\ ;_2\ \ ,\ \ (b\ ;_0 + d\ ;_1)\ f\ ;_2 \rangle.$$

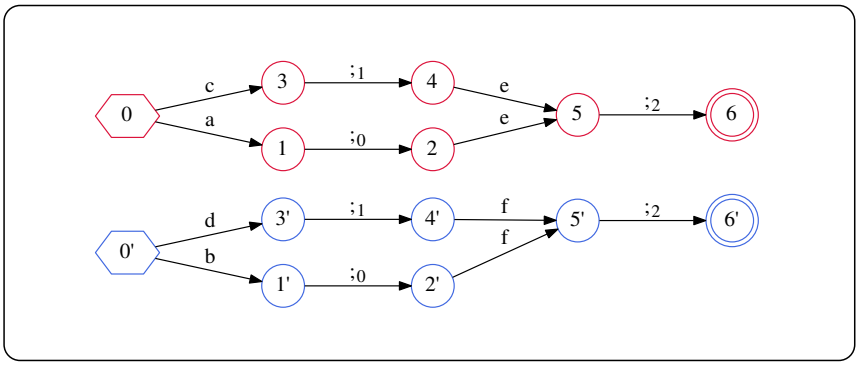
D'expression régulière à automate fini

L'étape suivante consiste à appliquer une transformation classique d'expression régulière vers un automate à chacun des membres du vecteur. La transformation que nous avons choisie est celle de Brüggemann-Klein [6] (une amélioration de

celle de Glushkov [8]) qui a la propriété d'avoir un état par symbole et nous assure l'existence d'états correspondant aux point-virgules.

Exemple 1 *(continuant de la page viii)* **:** l'application de l'algorithme de Brüggemann-Klein à tous les membres de la désynchronisation de R retourne ainsi

$BK^p(Dsn(R)) =$

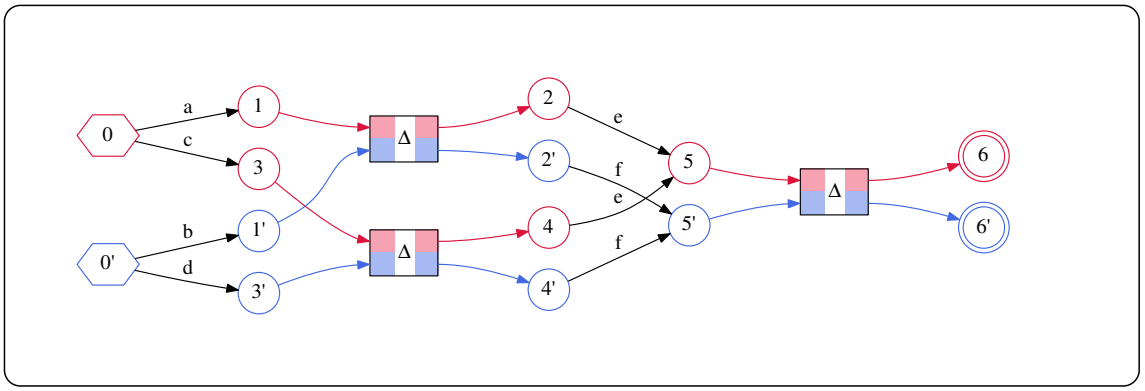


Synchronisation

La synchronisation consiste à joindre les automates avec des transitions globales dont les sources et destinations sont celles des transitions point-virgules.

Exemple 1 *(continuant de la page ix)* **:** la synchronisation du vecteur d'automate retourne ainsi

$Sync(BK^p(Dsn(R))) =$



Déterminisation d'automates BSP

Les BSPA produits des BSPRE par la transformation introduite dans la section précédente sont (généralement) non-déterministes. Pour faire en sorte que les automates BSP soient plus que des outils théoriques, la première étape est la déterminisation. Le déterminisme représente l'annihilation des ambiguïtés en prévenant les retours en arrière durant la recherche et ainsi évite toute perte

d'efficacité. Dans cette section, les automates non-déterministes seront dénotés par NBSPA et les automates déterministes par DBSPA.

Identification du problème

Regardons la figure 2. L'identifiant du composant de vecteur (*i.e.* numéro du processeur) est écrit en indice de l'identifiant d'état.

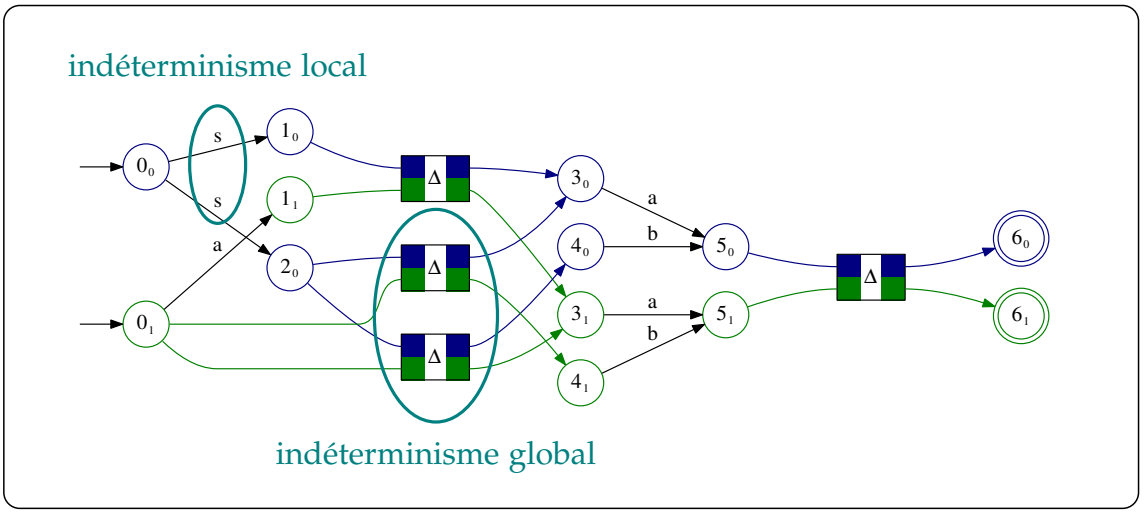


Figure 2 – *NBSPA comportant le langage* $L = \{\langle s,a\rangle\langle a,a\rangle, \langle s,\epsilon\rangle\langle a,b\rangle, \langle s,\epsilon\rangle\langle b,a\rangle\}$

Dans cette figure, deux types d'indéterminisme sont représentés. L'indéterminisme des transitions locales est dénoté par une source et étiquette commune pour une destination différente. Lorsque des transitions globales ont des sources identiques mais des destinations différentes, il s'agit d'indéterminisme global.

Solution par indexage

Nous traitons l'indéterminisme local en utilisant l'algorithme classique de construction par sous-ensemble entre les transitions globales. L'indéterminisme global est plus complexe comme en témoigne la figure 2. Si nous fusionnions naïvement les états 3_0 et 4_0 ainsi que 3_1 et 4_1 pour résoudre l'indéterminisme global, alors le langage BSP deviendrait $L = \{\langle s,a\rangle\langle a,a\rangle, \langle s,\epsilon\rangle\langle a,b\rangle, \langle s,\epsilon\rangle\langle b,a\rangle, \langle s,\epsilon\rangle\langle a,a\rangle, \langle s,\epsilon\rangle\langle b,b\rangle\}$ qui est différent de celui d'origine. Pour pallier à ce problème, nous dupliquons les états atteignables par les différents vecteurs de sortie des transitions globales en les indexant séparément. Ainsi, la déterminisation de la figure 2 donne (avec les index en exposants):

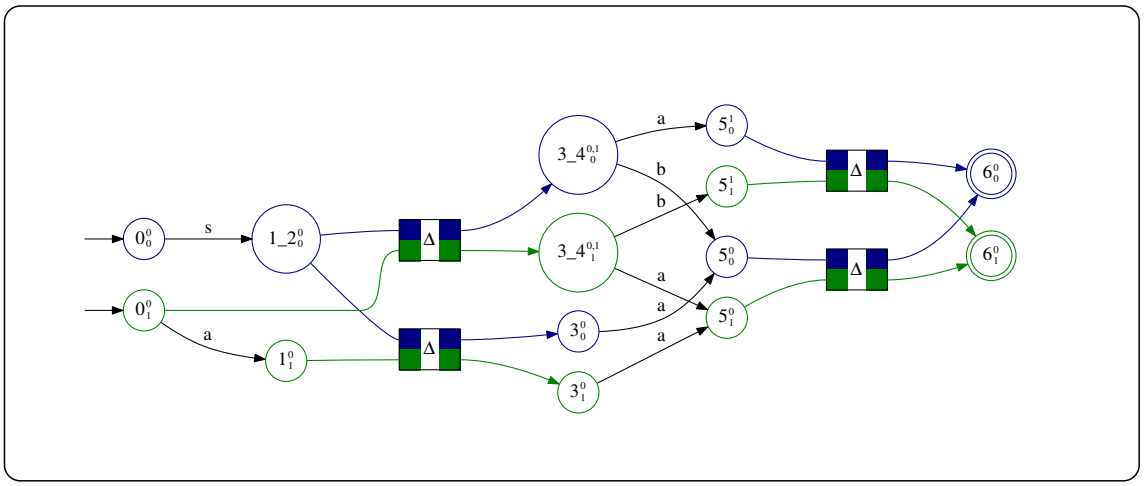


Figure 3 – *Déterminisation du NBSPA de fig. 2*

BSPRE pour la recherche parallèle par RE

Les applications de recherche par expression régulière varient des applications courantes telle que *grep*[17] aux applications spécialisées incluant par exemple l'inspection de paquets employée notamment par les pare-feux [50, 35, 61, 16]. La plupart des techniques existantes utilisées pour la recherche en parallèle par expression régulière simulent une exécution depuis un ensemble des états de l'automate [24, 48, 16]. D'autres techniques transforment l'automate en une forme propice à l'exécution parallèle [56], ou reposent sur du matériel dédié tel que les FPGAs [37, 40] ou TCAMs [47].

La nouvelle approche que nous introduisons se concentre sur les expressions régulières (RE) en les transformant en une forme parallèle (BSPRE) et en dérivant l'automate parallèle (BSPA) (voir figure 4). Notre approche commence par transformer l'expression régulière en une forme intermédiaire. Afin de reconnaître des données distribuées parmi p processeurs, l'expression régulière doit aussi être distribuée et la forme intermédiaire facilite cette étape. Les parties distribuées sont insérées dans des vecteurs de taille p, chacun représentant une potentielle distribution des données. La BSPRE finalement obtenue est la disjonction de ces vecteurs.

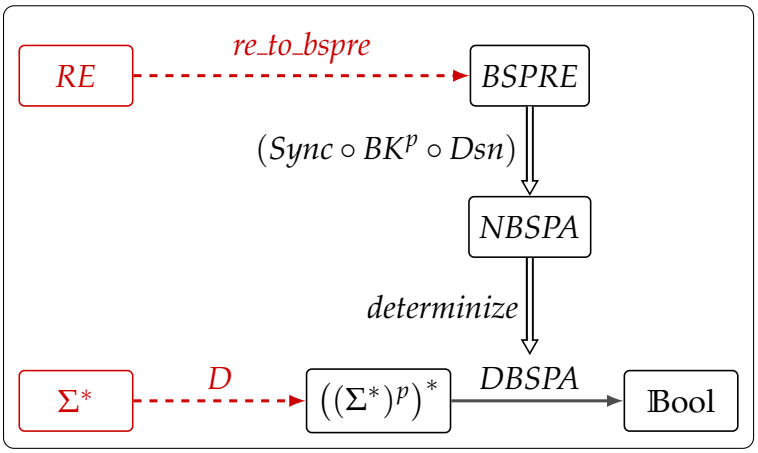


Figure 4 – *Schéma de reconnaissance parallèle*

Notre approche a été testée face à une approche standard de la reconnaissance parallèle par expression régulière. Les résultats montrent que notre approche est très efficace pour un petit nombre de processeurs mais avec un nombre plus important de processeurs, la taille du BSPA explose, et son temps de construction finit par devancer le temps de recherche.

Programmation de tenseurs avec BSP

Beaucoup d'applications de reconnaissance de motif et apprentissage machine reposent désormais sur les réseaux de neurones, par exemple la classification d'image et la reconnaissance d'objets dans des images ou des vidéos. Les réseaux de neurones sont des graphes de flux de données d'arithmétique linéaire pour des opérations de seuil sur des pixels, structurés en tableaux *couches* connectés par des dépendances pouvant être connectées de façon complète ou sporadique. Les éléments de ces couches sont appelés *neurones*. L'application d'un réseau de neurone à ses entrées est appelée *inférence*.

La structure des couches est définie à la main par des experts de réseaux de neurones pour chaque tâche spécifique et les coefficients qui constituent les opérations des couches (appelés *paramètres*) sont appris par une phase d'entraînement aux calculs intensifs dont l'algorithme est une optimisation à la plus forte décroissance avec la qualité de l'inférence comme fonction objectif. La qualité de l'inférence est définie de façon ad-hoc en notant les résultats sur un large ensemble de données.

Programmer un réseau de neurones se réduit à définir ses couches et leurs interconnections pour l'inférence puis l'entraînement pour obtenir des paramètres de haute qualité qui sont utilisés pour cible de l'application.

Le travail présenté ici est la tentative d'analyser plus loin la programmation des réseaux de neurones comme une programmation purement fonctionnelle et non-récursive avec un petit ensemble de primitives de tenseurs. Cela nous amène à une meilleure compréhension du (petit) fragment d'algèbre linéaire utilisé dans la programmation de réseaux de neurones. Il construit aussi une base presque universelle pour la construction de couches et réseaux de neurones, le seul exemple manquant actuel étant les réseaux dit récurrents *i.e.* ceux ayant des dépendances cycliques.

Nous décrivons nos primitives et un langage spécifique petit et simple appelé `HTL` pour leur applications à la programmation de réseaux de neurones (acyclique): type, sémantique, analyse statique, syntaxe, exemples de programmation et conception pour la génération de code parallèle et entraînement automatique.

Conclusion et Travaux futurs

Toutes les routes mènent à BSP

Nous avons montré deux différents ensembles de techniques pour la génération automatique de programme BSP.

Automates BSP

Automates BSP à partir d'expressions régulières BSP Les expressions régulières BSP peuvent être vues comme un langage déclaratif représentant un ensemble de mots. Les automates BSP sont quant à eux des machines pour résoudre le problème de décision d'appartenance d'un mot à un ensemble représenté par une expression régulière BSP. Cette transformation est donc la compilation d'un langage déclaratif à une machine testant l'appartenance d'un mot au langage représenté.

Déterminisation d'automate BSP Quand le temps de recherche est ce qui compte le plus (et qu'il n'y a pas de matériel spécial impliqué), les automates non-déterministes doivent faire place aux automates déterministes. Pour cette raison, nous traitons les automates non-déterministes comme une machine abstraite tandis que les automates BSP sont le code BSP concret responsable de la recherche de mots BSP. La déterminisation peut donc aussi être vue comme la compilation d'une machine abstraite vers du code BSP concret.

Recherche par expression régulière BSP La chaîne d'outils de transformations à partir d'expressions régulières vers des expressions régulières BSP puis des automates BSP non-déterministes puis enfin déterministes pour la recherche en parallèle par expression régulière montre de meilleurs temps de recherche comparé aux approches existantes qui gardent l'automate fini déterministe et spéculent sur l'état local de départ de la recherche. Néanmoins, la construction du BSPA prend en compte toutes les distributions possibles des données d'entrée, ce qui en effet enlève le besoin de spéculer mais aussi augmente grandement la taille de l'automate qui à son tour augmente le temps et l'espace requis par la déterminisation menant à un BSPA gargantuesque pour des grands nombres de processeurs. Notre approche reste toutefois très efficace pour de petits nombres de processeurs qui, rappelons-le, constituent la majeure partie des ordinateurs parallèles, en particulier ceux disponibles pour le grand public.

Cette approche est une compilation d'un programme séquentiel, l'expression régulière vers un programme BSP, l'automate BSP.

Calculs de Tenseurs en BSP

Nous avons introduit `HTL` , un langage spécifique minimal pour les calculs de tenseurs avec seulement 6 primitives de tenseurs. Ces primitives incluent le constructeur `init`, les transformateurs `map` et `content` (qui retourne un tenseur plus petit) et les destructeurs `to_scalar`, `shape` et `reduce`. Ces primitives ont été définies pour une implémentation séquentielle et BSP pour lequel un modèle de coût fut conçu. Sa pertinence a été montrée au travers d'un exemple réel. Son système de type a été introduit et inclue l'analyse de formes qui devrait être statiquement déduite de façon à prendre les meilleures décisions en terme de distribution de données, similairement à [52].

Les routes continuent en BSP

Le futur des automates BSP

Les prochaines étapes de recherche en parallèle par expression régulière incluent d'abord l'optimisation de l'implantation des BSPA pour en réduire la taille et ainsi diminuer davantage son temps de recherche. Par la suite, les expressions régulières étendues seront considérées. La certification de toute la chaîne d'outil serait un atout majeur pour la théorie des automates BSP. La preuve commencée en annexe A est le premier pas vers cet objectif. Celui-ci demanderait d'abord de transformer la conjecture 1 en théorème par la complétion de tous

les cas de la preuve. Par la suite viendrait la conservation du langage durant la déterminisation et le déterminisme de l'automate retourné incluant l'absence des ϵ-transitions et l'unicité des états résultant par transitions locales et globales. Concernant la parallélisation des RE en BSPRE, le langage des BSPRE devra être égal à la distribution du langage de la RE, pour une distribution par bloc. Cette dernière bénéficierait aussi d'une preuve que les BSPRE produites sont minimales ainsi qu'une quantification précise ou une sur-approximation de la taille de la BSPRE retournée.

Nous avons la conviction de ne pas avoir encore mesuré la portée des applications des BSPA. L'application de la reconnaissance en parallèle par expression régulière a le mérite d'utiliser toutes nos transformations et d'être bien connue. Pourtant, la première transformation des expressions régulières vers expressions régulières BSP crée des BSPA trop larges pour passer à l'échelle et la BSPRE est simplement une disjonction de vecteur d'expression régulière. Elle n'utilise ni l'étoile de Kleene, ni la concaténation (*i.e.* synchronisation), même s'il est vrai que nous nous reposons sur ce fait pour justifier de la suffisance des cas couverts par la conjecture 1 dans l'annexe A. Une application requérant l'utilisateur de concevoir la BSPRE lui-même devrait prévenir l'explosion due à la première transformation et faire bon usage de l'expressivité des BSPRE. Nous espérons encore l'apparition d'une application mettant à profit toute la puissance des automates BSP.

HTL pour l'apprentissage machine

Nous avons démontré que `HTL` est un concept de langage productif et effectif pour la programmation de réseaux de neurones *inférence i.e.* le calcul temps-réel. Mais les programmes de réseaux de neurones sont inutiles sans entraînement pour obtenir une inférence de haute qualité à partir d'une optimisation de ses paramètres. A cet effet, nous définirons un ensemble d'équations de différenciations pour nos primitives de tenseurs. Une telle définition permettrait le calcul automatique de la dérivé d'un programme par rapport à ses paramètres de poids. Le petit nombre de primitives avec des définitions mathématiques simples dans `HTL` devrait rendre ce processus à la fois sûr et efficace.

Il est aussi prévu d'augmenter les cibles matérielles d'`HTL` au GPUs pour lequel de nombreuses approches sont disponibles. Nous pourrions générer directement le code CUDA, l'API C pour programmer des GPUs Nvidia ou le

moins spécifique OpenCL, un standard ouvert pour la programmation de GPU en C++. Ceci est l'approche bas niveau. L'approche haut-niveau serait soit de reposer sur SPOC [4], une bibliothèque OCaml visant à la fois CUDA et OpenCL, soit d'utiliser les approches à base de modèle telles que BSP avec BSPGP [25], une bibliothèque de C consistant en un petit nombre de primitives visant CUDA ou alors multi-BSP avec multi-ML [1], une extension de BSML pour multi-BSP ou encore SGL [39], un langage basé sur deux primitives majeures pour les architectures hétérogènes. Enfin, viser les récents TPUs [29] devrait constituer la suite logique pour laquelle peu de langages sont encore disponibles.

Acknowledgements

I would like to thank first my supervisors Frédéric Loulergue and Gaétan Hains for guiding and helping me so much throughout this journey. I will never forget how tough it was to write the thesis in time, but I could hang on because I knew Frédéric was working just as hard as me to improve it. My thesis topic was born from Gaétan's idea, it's evolution through other projects and the following of this thesis was also thanks to Gaétan. If this journey was on boat, then I paddled a lot, Frédéric blew in the sails to move forward and Gaétan showed me lighthouses to follow.

I thank the reviewers Franck Pommereau and John Mullins for the great work they achieved in so little time. Wijnand Suijlen, thank you for being in my jury and for everything you taught me on the cluster. I thank Jean-Michel Couvreur, for accepting to be in my jury and helping me while Frédéric was away. I would also like to thank Chong Li here, even though he was not in my jury, because he is the one who brought me to this thesis in Huawei.

I thank my colleagues and friends Arvid Jakobsson, who shared my pain (especially after our biweekly meetings) and passion for climbing, Anthony Palmieri who managed to put me both on a game and in a bar (the former was easier), Filip Pawlowski successfully brought me to move (out of my couch) to his country, Pierre Leca shared with me the great title of "caveman", Jan-Willem Goossens fought alongside me in the summoner's rift and Juan Carlos Bucheli Garcia who will appreciate the lack of drug joke and also totally deserves the long space taken by his name here, for their moral support and sharing the same burden.

I thank my dear friends from university, the too bright (to be modest) Mattias Roux for his humor (at my expense), the handsome Bruno Fruchard for our heartfelt discussions in your car or during the few times you managed to extract me from home, the great Albin Coquereau for supporting me and anyone when so few choose this role and the kind Marie Laveau. They all aimed to do research which straightened my winding path to the same destination. Hopefully I will like it for a while.

I'm grateful to my family for their support and enthusiasm despite the distance in my choice of doing this PhD. I already thanked friends from other countries who had it worse but I already had to change language. Yes, they do not understand chocolatine here. I thank my sister Marie for also pursuing research although there is not that many other choices in musicology. I also thank my brother Vincent for his so warm welcome "Adiou c****d" and always providing me the best anecdotes to tell. I thank my mother Fanny for being so funny and looking after me and my siblings, the family wouldn't have survived two days without you. I thank my dad Pierre, who pushed me to studies by showing me how I disliked working with my hands.

I especially thank Geraldine Tacuel for supporting, feeding and bearing with the moody me during the final intense period of this PhD.

I would have liked to thank many more but I hope they will agree that their involvement in this thesis was perhaps not paramount and, as usual, I'm late and need to finish writing this now.

I would like to end on a quote of the greatest movie but the few English sentences pronounced there would require too many asterisks to be readable, therefore, "faisons comme ça".

List of Figures

1 Introduction

1.1 A parallel world

Parallel computers are everywhere today from two cores smart watches to million cores supercomputers. Processor clock frequency has been stagnating for years and increasing the number of processors or cores has been the sole mean to increase computing power. Programming one processor (*i.e. sequential programming*) is hard. Programming several processors (*i.e. parallel programming*) requires to program not only their local tasks but also their communications. This requires more work and thus provides more opportunities for errors. Consequences of programming errors range from an unnoticeable different shade of a pixel color to a rocket explosion. It is thus not an overstatement to say that helping programmers can be a matter of life and death. Parallel programming is in general even harder than sequential programming and thus requires even more help. For example, a well-known error in parallel programming is the *deadlock*, an error happening when processors are waiting for each other forever. Several approaches exist to help parallel programmers to avoid errors.

1.2 Make the parallel world a better place

Models such as *bulk-synchronous parallel* model [60] (BSP) restrict the programmers to prevent them from committing numerous mistakes. For example, if respected, it enforces all processors to synchronize at the same time thereby preventing deadlocks. Restricting the programmers may nevertheless come at the price of performances. This trade-off between safety and performances is analogous to the efficiency of low-level languages (*e.g.* assembly or C) against safety of the (well-made) high-level ones (*e.g.* OCaml).

Verification aims at analyzing programs to tell the programmer about errors he made. Verification may analyze written programs statically or running programs dynamically. This method may be added to the previous ones to verify properties that were not ensured by the model. It may also verify the respect of the model, for example, if all processors synchronized at the same time [27] which is a necessary condition for BSP.

Certification proves that a program satisfies a specification [42]. Specification of a program is the mathematical description of conditions that need to be fulfilled by the program to be correct. This approach is the only one that may ensure the complete lack of errors in the written program. This method requires nonetheless a considerable amount of work even with the right tools, *e.g.* Coq. Coq [53] is a proof assistant that may be used for proofs of mathematical theories as well as a programming language from which certified programs may be extracted.

Generation of programs (*i.e.* code) from a higher level language provides the safety of the source language with performances of the target. Compilers are usually responsible for this translation and often comes with some static verification before code generation. For example, BSML is a high level BSP library, free of deadlocks, that may be compiled to an efficient low level library such as MPI. MPI normally does not prevent deadlocks, but it does when generated from BSML. A few examples of certified compilers exists [38] which guaranty that the compiled program preserves the source program semantic.

1.3 Gates to the parallel world

We developed the latter approach to generate parallel BSP code from sequential programs. Having the parallel model BSP as the target of our generation makes it easier in the same manner that models make parallel programming easier. BSP also yields a cost model to measure the time a program would take which is an interesting added value to our work. Certification is too cumbersome to be treated in early stages of the design as it would slow it tremendously. However, we always took into account future certification in our approaches.

However automatic generation of general-purpose code is a deep and hard problem, but narrowing the application allows us to be more effective in doing

so. Thus, two specific domains were targeted with different techniques used for each of them.

1.3.1 BSP automata for regular expression matching

BSP Automata (BSPA) and BSP regular expressions (BSPRE) were introduced by Hains [19] in 2016. We developed this theory through three main transformations that the theory was lacking until then.

Generation of BSP Automata from BSP regular expressions. This transformation is akin to the fundamental transformation from regular expression to a non-deterministic finite automaton recognizing the language represented by the former [59, 7, 6]. It may be understood as the compilation of BSPRE, a representation of BSP language, to BSPA, a machine recognizing BSP words belonging to this language.

Determinization of BSP Automata to make them more efficient. Automata and BSP automata are in general not deterministic which means that several transitions with the same label from the same state may exist, giving birth to an ambiguity. During matching, this ambiguity forces a choice which, if wrong, forces in turn to a backtrack, loosing time that is precious in applications like regular expression matching. In our case, non-deterministic BSPA are an abstract finite state machine while their determinization into deterministic BSPA is a compilation to executable parallel code.

Parallelization of regular expressions into BSP regular expressions. This transformation is dependent on the input distribution to the processors. It computes the BSPRE to take into account all possible splits of the input occurring during distribution. Given a sequential representation, this transformation returns a BSP representation, which is the first step needed to eventually compile to BSP code.

Composition of these three previous transformations which starts from parallelization of regular expression into BSPRE, followed by the generation of non-deterministic BSPA which are determinized into deterministic BSPA, enables a parallel regular expression matching scheme.

1.3.2 Domain specific language for tensor computation

We were also interested in neural nets, a field currently receiving a lot of focus. Neural nets are usually based on computational graphs implemented with tensors (*i.e.* multidimensional arrays). Recent works such as Relay [54] replaces computational graphs by a programming language. However, existing works provide in our opinion too many features, making formal methods (*e.g.* verification or certification) harder to apply. We designed a small (domain) specific language (DSL) with a few primitives of which composition is sufficient for this domain. This DSL is compiled to BSP code thereby being another example of generated parallel BSP code from sequential program.

1.4 Publications

This section presents our accepted articles.

1. Thibaut Tachon, Chong Li, Gaétan Hains, and Frédéric Loulergue. Automated generation of BSP automata. *Parallel Processing Letters*, 27(01):1740002, 2017. DOI 10.1142/S0129626417400023.

2. Gaétan Hains, Thibaut Tachon and Youry Khmelevsky. From natural language to graph queries. In *The annual IEEE Canadian conference on Electrical and computer engineering (CCECE)*, 2019. *to appear*.

3. Thibaut Tachon. Parallel Matching of Regular Expressions with BSP Automata. In *The International Symposium on Formal Approaches to Parallel and Distributed Systems (4PAD)*, 2019. *to appear*.

Although article 2 may also be considered as a compilation from sequential language (natural language) to BSP program (because there exist a BSP back-end for processing graph queries), this paper mostly present work of students and its contribution is more pedagogical than technical. Thus, this paper was not extended in the present document.

1.5 Organization

This document is organized as follows.

After this brief Introduction in Chapter 1, Preliminaries will introduce the concepts we rely on as well as notations used throughout the document in Chapter 2. State of the Art follows to introduce the related work, to which we are compared, build on or will try to include in Chapter 3. Note that because there is no common notation or related work, Preliminaries and State of the Art do not include our specific work on Chapter Tensor Programming with BSP which is self-contained.

We present creation of BSP Automata in Chapter 4. An informal mathematical proof is given that generation preserves language in Appendix A. Although we did not have the time to complete this proof for all cases, we completed enough cases to cover the application of BSPRE for Parallel Matching of RE. We describe in detail Determinization of BSP Automata in Chapter 5. BSPRE for Parallel Matching of RE is shown in Chapter 6 It includes parallelization of regular expression and the implementation of the three transformations composition that is evaluated against a standard approach of parallel regular expression matching.

Tensor Programming with BSP is introduced in Chapter 7, including its own related work, notations and the presentation of the DSL. We conclude this document and present future work in Chapter 8.

Preliminaries 2

Contents

This chapter aim at defining the concepts used throughout this thesis in order to be as self-contained as possible. Those concept definitions rely on basic notations introduced in section 2.1. Our contributions in Bulk-Synchronous Parallel (BSP) Automata theory (section 2.4) builds on finite automata theory and BSP model, respectively over-viewed in section 2.2 and section 2.3. The Chapter 7 is self contained and do not need preliminaries.

2.1 Notations

This section presents general notations. We describe notations for type (in subsection 2.1.1) as well as some functions frequently used (in subsection 2.1.2)

2.1.1 Type notations

- In "$v : t$", the right hand side of the colon is the type of the left hand side (for example $7 : \mathbb{N}$).
- Type "$\alpha\ t$" is read t of α. For example $\mathbb{N}$ *vector set* refers to set of vectors of natural numbers.
- A function type has form "$f : \underbrace{\alpha \to \beta \to ...}_{\text{inputs type}} \to \underbrace{\zeta}_{\text{output type}}$."
- A tuple type has form "$(t_1 * \ldots * t_n)$". In particular, a pair type is written "$(t_1 * t_2)$".

2.1.2 Functions

- $\mathcal{P}(X)$ is the powerset, set of all subsets of set X.
- $fst : \alpha \times \beta \to \alpha$ is a function to get the first member of a pair.
- $snd : \alpha \times \beta \to \beta$ is a function to get the second member of a pair.
- X^*
  - ▷ when X is a (BSP) regular expression, it is the Kleene closure.
  - ▷ when X is a set, is the set of all sequences of elements of set X.
  - ▷ when X is a function, it is the application of this function to a sequence of elements (*e.g.* $snd^*(a_1, 1)(a_2, 2)\ldots(a_n, n) = 1\ldots n$)

Lists will be also used and are defined as in the OCaml language.

Definition 1 *α list* **:** *Σ re list*

$$\begin{array}{rll} l ::= a :: l' & \text{Left extend operator} : \alpha \to \alpha\ list \to \alpha\ list \\ \mid\ [\,] & \text{Empty list} : \alpha\ list \end{array}$$

Lists are enumerated between square brackets. We also define concatenation of lists.

Definition 2 @: $\alpha\ list \to \alpha\ list \to \alpha\ list$ **:** List concatenation.

$$\begin{aligned}(a :: l_1) \mathbin{@} l_2 &= a :: (l_1 \mathbin{@} l_2)\\ [\,] \quad \mathbin{@} l_2 &= l_2\end{aligned}$$

We also rely on sets in their mathematical definition and notation.

Vectors are also used. We refer to an element of vector v at position i with notation v^i. The content of a vector v is set to a at position i with $v^i \leftarrow a$. Note that setting the content of a vector returns that vector.

2.2 Finite automata theory

2.2.1 Word, alphabet and language

Let Σ be a non-empty finite set of symbols, called the alphabet. A sequence of symbols is named a word, a word is an element of Σ^*. A language L is a subset of Σ^*, it is a set of words, or in other words an element of $\mathcal{P}(\Sigma^*)$.

2.2.2 Regular expressions

$L(E)$ is the language represented by the regular expression E. The empty word is denoted by ϵ. The empty language is denoted by $L(\emptyset)$.

Definition 3 RE **:** $\Sigma\, re$

$\Sigma\, re$ is the type of regular expression parametrized with Σ, the set of symbols. A regular expression E is defined as following.

$$E := F + G \mid F \cdot G \mid F^* \mid a \in \Sigma \mid \epsilon \mid \emptyset$$

Definition 4 L_{RE} **:** $\Sigma\, re \to \mathcal{P}(\Sigma^*)$

The language of a regular expression on Σ is recursively defined as usual:

- $\emptyset$ is a regular expression representing the empty language : $\{\}$.
- ϵ represents the language which only contains the empty word : $\{\epsilon\}$.

- $\forall a \in \Sigma, a$ represents the language $\{a\}$.
- F+G represents the language $L(F) \cup L(G)$.
- $F \cdot G$ (or just FG) represents the language $L(F) \cdot L(G)$ (*def.* $L_1 \cdot L_2$).
- F^* represents the language $\bigcup_{i=0}^{\infty} L(F^i)$ (*def.* F^i).

Definition 5 $L_1 \cdot L_2$ **:** $\mathcal{P}(\Sigma^*) \to \mathcal{P}(\Sigma^*) \to \mathcal{P}(\Sigma^*)$

Operator $L_1 \cdot L_2$ on languages is defined as usual :

$$L_1 \cdot L_2 = \left\{ w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2 \right\}$$

Remark 5.1 ($L_1 \cdot L_2$). $\{\epsilon\} \cdot L = L \cdot \{\epsilon\} = L \qquad \emptyset \cdot L = L \cdot \emptyset = \emptyset$

$$\epsilon \in (L_1 \cdot L_2) \iff (\epsilon \in L_1) \wedge (\epsilon \in L_2)$$

Definition 6 F^i **:** $\Sigma\, re \to \mathbb{N} \to \Sigma\, re$

$$F^i = \begin{cases} F^0 &= \epsilon \\ F^{i+1} &= F^i \cdot F \end{cases}$$

2.2.3 Finite automata

a) Non-deterministic Finite Automaton

A non-deterministic finite automaton (NFA) A is is formally defined as

$$A = (Q, \Sigma, \delta, I, F).$$

Where Q is the set of all states, I the set of initial states and F the set of final states. Σ is the set of symbols also called *alphabet*. $\delta : Q \to (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$ is the transition function, *i.e.* a function that given a state source and a label (symbol or ϵ) returns the set of states destination. In some formalisms, I is not a set of states, but a single one. However, having a set of states or a single one with ϵ-transitions to other states makes no difference for the language represented by the automaton.

An NFA A recognizes (*i.e.* match) a word $u_0 \dots u_{n-1}$ if, from an initial state, successively taking transition labeled by the successive symbols of the word leads to a final state. More formally,

$$\begin{aligned} u_0 \dots u_{n-1} \in L(A) &\iff \exists q \in I.\ \delta(\dots \delta(\delta(q, u_0), u_1), \dots u_{n-1}) \in F \\ &\iff \exists q \in I.\ \delta^*(q, u_0 \dots u_{n-1}) \in F \end{aligned}$$

b) Deterministic Finite Automaton

A deterministic finite automaton (DFA) A is is formally defined as

$$A = (Q, \Sigma, \delta, q_0, F).$$

Where, like NFA, Q is the set of all states, Σ is the alphabet and F is the set of final states. Unlike NFA, q_0 is the only initial state as there must not be any ambiguity where the matching starts, because it's deterministic. Also, $\delta : Q \to \sigma \to Q$ is deterministic as it has no ϵ transition and there is only one transition from a given state with a given label (leading to only one destination state).

A word $u_0 \dots u_n$ belongs to the language represented by the DFA A when

$$\begin{aligned} u_0 \dots u_{n-1} \in L(A) &\iff \delta(\dots \delta(\delta(q_0, u_0), u_1), \dots u_{n-1}) \in F \\ &\iff \delta(\dots \delta(\delta(q_0, u_0), u_1), \dots u_{n-1}) \in F \\ &\iff \delta^*(q_0, u_0 \dots u_{n-1}) \in F \end{aligned}$$

2.3 Bulk-synchronous parallel model

BSP stands for Bulk Synchronous Parallel. It is a programming model which structure the parallel program into a sequence of steps (described in subsection 2.3.1). Programmers are used to write sequential programs and having a program divided in a sequence makes it much easier to understand. BSP also comes with a cost model (subsection 2.3.2) thereby enabling comparison and assessment of BSP algorithms.

2.3.1 Supersteps

A BSP machine is a set of homogeneous processor-memory pairs interconnected by a network and having a global synchronization mechanism. A program obeying the BSP model is structured as a sequence of *supersteps* (represented in Figure 2.1). Each of them starts with asynchronous computations, followed by exchange of data, and ends with a synchronization barrier after which the exchanged data is available in each local memory.

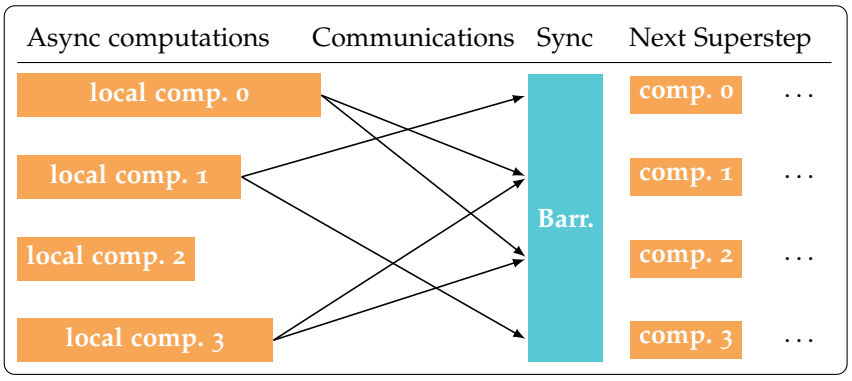


Figure 2.1 – *BSP computation example for* $p = 4$ *processors*

2.3.2 Cost model

Performance predictability is a major feature of the BSP model, in other words the execution time taken by a BSP program or its cost. In order to compute this cost, parameters of the BSP machine must be taken into account, noted p the number of processors, g the throughput of communications (amount of data communicated per time unit) and l the latency, being the time needed to perform a synchronization.

Note S the number of supersteps in a BSP program, the execution time of a BSP program is the sum of the time taken by all S supersteps. The time of a superstep is the sum of the time taken by the local computation, the communication and the synchronization. Note w_s^i the local computation time of a processor i $(0 \leq i \leq p-1)$ in a superstep s $(0 \leq s \leq S-1)$. The local computation time of a superstep s is thus: $\max_{i=0}^{p-1}(w_s^i)$. Note h_s^i and r_s^i the data sent (resp. received) by a processor i in a superstep s. Therefore, communication time of a superstep s is $g \cdot \max_{i=0}^{p-1}(\max(h_s^i, r_s^i))$.

To sum up, time taken by a superstep s is

$$\max_{i=0}^{p-1}(w_s^i) + g \cdot \max_{i=0}^{p-1}(\max(h_s^i, r_s^i)) + l.$$

Hence, the total execution time of a BSP program is

$$\sum_{s=0}^{S-1}\left(\max_{i=0}^{p-1}(w_s^i) + g \cdot \max_{i=0}^{p-1}(\max(h_s^i, r_s^i)) + l\right),$$

or

$$\sum_{s=0}^{S-1}\max_{i=0}^{p-1}(w_s^i) + g \cdot \sum_{s=0}^{S-1}\max_{i=0}^{p-1}(\max(h_s^i, r_s^i)) + Sl.$$

2.4 BSP AUTOMATA THEORY

A major part of the thesis builds on the BSPA theory [19], which must be presented. As main elements, BSP words, BSP regular expressions and BSP automata are defined in this section.

2.4.1 BSP words and languages

To model asynchronous computation of one superstep, we use a vector of size p (p is the number of processors) whose components are words. Words may be understood as sequential traces of one processor in one superstep where symbols are elementary operations.

Definition 7 : Elements of $(\Sigma^*)^p$ are called word vectors. A BSP word over Σ is a sequence of word vectors, i.e., a sequence $((\Sigma^*)^p)^*$. A BSP language over Σ is a set of BSP words over Σ.

A word vector models computation within one superstep and a sequence of supersteps is modeled by a BSP word. A BSP language models a set of different traces of a BSP program.

2.4.2 BSP regular expressions

In this section we present how to adapt regular expressions to BSP languages. The set of positions of a p-sized vector is denote by $[p] = \{0 \dots p-1\}$. Two different notations are presented: enumerated BSP regular expressions and intensional BSPRE

a) Enumerated BSPRE

An enumerated BSPRE (in this subsection, all BSPRE will be enumerated) is an expression R from the following grammar:

$$R ::= \emptyset \mid \epsilon \mid \langle r^0, \dots, r^{p-1} \rangle \mid R \cdot R' \mid R + R' \mid R^*$$

where $r^{i \in [p]}$ is a scalar regular expression, as defined in subsection 2.2.2.

For scalar regular expressions and BSP regular expressions, the product is often written $r\ r'$ (resp. $R\ R'$) instead of $r \cdot r'$ (resp. $R \cdot R'$). R is parametrized by

R	$L(R)$
$\emptyset$	$\{\}$
ϵ	$\{\epsilon\}$
$\langle r_0, \ldots, r_{p-1} \rangle$	$L(r_0) \times \cdots \times L(r_{p-1})$
$R \cdot R'$	$L(R) \cdot L(R')$
$R + R'$	$L(R) \cup L(R')$
R^*	$\bigcup_{i=0}^{\infty} L^i(R)$

Table 2.1 – *Enumerated BSPRE language*

the set of local symbols and the number of processors p. When needed, we indicate this information by a type: (Σ, p) *bspre*. Table 2.1 defines the BSP language generated by a BSP regular expression. L is a function with signature $L : (\Sigma, p)$ *bspre*$\rightarrow \mathcal{P}(((\Sigma^*)^p)^*)$.

These BSP expressions yield two levels: the BSP level and the scalar level. The former describes the structure of the main parallel program while the latter describes the local program structure. For example, $\langle a, b \rangle^*$ models a BSP program for 2 processors where the first processor does a, the second processor does b and both loop after synchronization.

b) Intensional BSPRE

The former notation is not a scalable point of view on parallel programming. To address this issue, intensional notations are introduced where local regular expressions are defined for a set of locations independent from p.

This set of locations will be the language represented by a regular expression. Assume that locations $i \in [p]$ are written in binary notation (0,1,10,11, ...). We define a binary regular expression (BRE) by the following grammar :

$$b ::= \emptyset \mid 0 \mid 1 \mid b \cdot b \mid b + b \mid b^*$$

Note that the empty word has no signification in terms of location, so it does not appear in BRE. As an example, $b_1 = (0+1)^*1$ represents the set of odd-rank locations.

We redefine the BSPRE with intensional notation :

$$\begin{aligned} R &::= \langle\ V\ \rangle \mid \emptyset \mid \epsilon \mid R\,;R \mid R + R \mid R^* \\ V &::= r\,@\,b \qquad \textit{local RE at location encoded by } b \\ &\quad \mid\ V \parallel V \qquad \textit{superposition by pointwise-disjunction of RE} \end{aligned}$$

where $r \in \text{RE}$ and $b \in \text{BRE}$.

For example, the intensional BSPRE $\langle a \,@\, 0 \parallel b \,@\, (0+1)^*0 \parallel c \,@\, (0+1)^*1 \rangle$
is equivalent, when $p = 4$, to $\langle a \,@\, 0 \parallel b \,@\, 0 + 10 \quad \parallel c \,@\, 1 + 11 \rangle$
which corresponds to enumerated BSPRE $\langle \underset{0}{a+b}, \underset{1}{c}, \underset{10}{b}, \underset{11}{c} \rangle$

We overload the type *bspre* for intensional BSPRE with $\Sigma\, bspre$ (not p-dependent) and the function L with type : $(\Sigma\, bspre, p) \to \mathcal{P}(((\Sigma^*)^p)^*)$. Table 2.2 defines the BSP language generated by an intensional BSP regular expression.

R	$L(R,p)$
$\emptyset$	$\{\ \}$
ϵ	$\{\epsilon\}$
$\langle r \,@\, b \rangle$	$\prod_{i=0}^{i=p-1} \begin{cases} L(r) & \text{if } i \in L(b) \\ \{\emptyset\} & \text{else} \end{cases}$
$\langle r_1 \,@\, b_1 \parallel \ldots \parallel r_k \,@\, b_k \rangle$	$\prod_{i=0}^{i=p-1} \bigcup \{L(r_j) \mid i \in L(b_j), 1 \le j \le k\}$
$R_1\,;R_2$	$L(R_1) \cdot L(R_2)$
$R_1 + R_2$	$L(R_1) \cup L(R_2)$
R^*	$\bigcup_{i=0}^{\infty} L^i(R)$

Table 2.2 – *Intensional BSPRE language*

2.4.3 BSP automata

Definition 8 : A BSPA is a tuple

$$\vec{A} = (\{Q^i\}_{i\in[p]}, \Sigma, \{\delta^i\}_{i\in[p]}, \{q_0^i\}_{i\in[p]}, \{F^i\}_{i\in[p]}, \Delta)$$

such that for every i, $(Q^i, \Sigma, \delta^i, q_0^i, F^i)$ is a finite deterministic automaton[1], and $\Delta : \vec{Q} \to \vec{Q}$ is called the synchronization function where $\vec{Q} = (Q^0 \times ... \times Q^{(p-1)})$ is the set of global states. In other words a BSPA is a vector of sequential automata A^i over the same alphabet Σ, together with a synchronization function (Δ) that maps state-vectors to state-vectors.

[1] Q^i is the finite set of states, δ the transition function, $q_0^i \in Q^i$ the initial state, and $F^i \subseteq Q^i$ the non-empty set of accepting states.

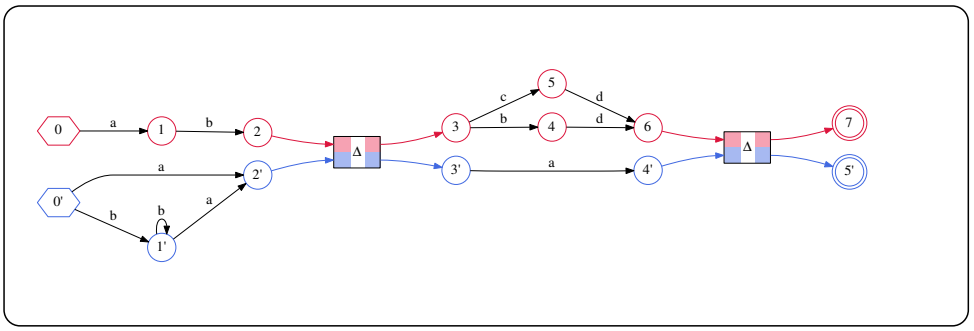


Figure 2.2 – *A BSP automaton*

To complete the explanation of BSP automata, we give an example automaton by its graphical representation in Figure 2.2. In this example, $p = 2$. The local automaton $(Q^0, \Sigma, \delta^0, q_0^0, F^0)$ is represented in red, the local automaton $(Q^1, \Sigma, \delta^1, q_0^1, F^1)$ in blue. Initial states are hexagons, final states are double circles. The function $\Delta = \{\langle 2, 2' \rangle \to \langle 3, 3' \rangle, \langle 6, 4' \rangle \to \langle 7, 5' \rangle\}$ models synchronization happening at the end of a word vector in a BSP word. As an example, the above BSPA recognize the BSP word $\langle ab, ba \rangle \langle cd, a \rangle$. This may be checked with the BSPRE matching algorithm described below.

BSPRE Matching of a BSP word. The algorithm described here recognize a sequence of word vectors among those which belong to the language described by a BSPRE.

Note $\delta^* : Q \to \Sigma^* \to Q$ the extended transition function such that $\delta^*(q, \{u_0 \dots u_{n-1}\}) = \delta(\dots(\delta(q, u_0), \dots), u_{n-1})$

1. If the sequence of word vectors is empty, the vector state remains the vector of local initial states until step 5; otherwise continue.
2. $\langle w^0, \dots, w^{p-1} \rangle$ is the first word vector of the sequence. Local automaton $i (\in [p])$ applies $\delta^*(q_0^i, w^i)$ to reach some state q^i.
3. If $\langle q^0, \dots, q^{p-1} \rangle$ is not argument of the synchronization function Δ, the BSP word is rejected.
4. The synchronization function Δ maps: $\langle q^0, \dots q^{p-1} \rangle \to \langle q'^0, \dots q'^{p-1} \rangle$
5. If there are no more word vectors, and $\forall i.\ q'^i \in F^i$, the BSP word is accepted.
6. If there are no more word vectors, and $\exists i.\ q'^i \notin F^i$, the BSP word is rejected.
7. If there are more word vectors, control returns to step 2 but with local automaton i in state q'^i, for every location i.

State of the Art

3

Contents

We focus on BSP programs generation thus we study languages for writing BSP programs directly in section 3.1.1 or deriving them from specification in section 3.1.2. As BSP Automata may also be considered as a BSP computation semantic, we also study BSP semantics in section 3.1.3.

The main application we developed being parallel regular expression matching we also study the related work. We start by describing automata optimizations in a sequential context (section 3.2.1) to have a better grasp of the domain and understand better optimizations in parallel context studied in section 3.2.2. We also study dedicated hardware for parallel regular expression matching as it may be of interest for future work.

3.1 Bulk synchronous parallel model

3.1.1 BSP programming

We present in this section different libraries for writing BSP programs in different languages, BSPlib for C, BSML for OCaml and JBSP for Java.

Hill, McColl et al. [23] presented in 1998 BSPlib, a C-library for programming BSP programs consisting of only[1] 20 basic operations. Programming of all processors is coded in a single program, *i.e.* following the *Single Program Multiple Data* (SPMD) programming model. It offers two communications mode. The direct remote memory access allow a process to manipulate certain registered areas of a remote memory which have been made available by the corresponding process. The bulk-synchronous message passing is more convenient when the volume of data is not static but data dependent as it relies on buffers.

Gu, Lee and Cai [18] present JBSP, a BSP programming library for Java. Like BSPlib, it relies on SPMD programming and also propose message passing and direct remote memory access as communication modes. Parallelism is implemented with Java threads and each JBSP host uses two daemons, one for tasks, one for communications. The Java virtual machine (JVM) makes the program executable on any platform that supports JVM without recompiling.

Loulergue, Gava and Billiet [43] present BSML, an OCaml library for writing functional BSP programs consisting of only 4 primitives (*i.e.* basic operations coded with lower level libraries). It prevents dead-locks and indeterminism while providing an estimation of its execution time. BSML does not rely on SPMD programming but on parallel vectors, an explicit distributed data structure with a component per processor. The primitives are `mkpar` to create a parallel vector, `apply` to apply a vector of function to another, `put` to communicate vector values and `at` (name today is `proj`) to make a vector component available for all. It also provides a function to get the static number of processor, `bsp_p`.

3.1.2 BSP programs from specifications

Chen and Sanders [10] proposed *Logic of global synchrony* (Logs), a specification formalism for the specification of BSP programs. They also describe a *top-down*

[1] compared to MPI

method for designing BSP programs from Logs. Their top-down method starts from an abstract formal specification which is refined into a more concrete form until an actual BSP program.

Niculescu [52] presents a method to refine formally BSP programs. This method takes different data distribution into account which lead to different algorithms. It also allows to compute BSP cost model in early stages of the refinement in order to choose the best option according to characteristics of each concrete systems.

Loulergue, Bousdira and Tesson [42] rely on SyDPaCC to generate correct parallel programs from a formal specification. SyDPaCC is a set of libraries of the proof assistant Coq. The specification is written in Coq, from which the sequential program is derived and then automatically refined into a parallel program thanks to algorithmic skeletons. Afterwards, the parallel program is extracted to OCaml code with BSML and C code with MPI.

3.1.3 BSP computation semantics

Loulergue, Hains and Foisy [44] introduce BSλ, an extension of λ-calculus for functional languages expressing BSP algorithms. BSλ was presented as basis for programming BSP algorithms as pure functional programs (which was successful because BSML is largely inspired from it). BSλ is also an equational theory and thus offers a possible basis for BSP programs equivalence proofs.

Merlin and Hains [49] formalize BSP semantics with CCS. The Calculus of Communicating Systems (CCS) is a process algebra used for the analysis of safety, liveness and security in protocols or distributed programs. This study lead to a complex theory named bisimulation or trace equivalence of BSP-like computations.

Fortin and Gava [15] present BSP-Why, a tool for verifying BSP programs. It relies on the verification condition generator Why which takes an annotated program as input and produce verification conditions as output to ensure correctness of the properties given in the annotations. BSP-Why is based on a sequential simulation of the BSP programs which allows to generate pure sequential code for Why.

3.2 Efficient regular expression matching

Our research for efficient parallelization of regular expression matching (REM) brought us to study optimization of automata processing. We studied different category of optimization of sequential automata and provide the specific study of a few iconic paper for each of them. Parallel REM is subsequently studied from the most naive to state of the art DFA run as well as new parallel automata model. REM is such an important challenge that compatibility with costly dedicated hardware is extensively studied of which major principles are reported here.

3.2.1 Sequential optimization

Optimization of sequential automata often rely on compressing them. Smaller automata means more efficient memory use with less cache misses which lead to a decreasing matching time. Major compressing techniques are reported in this section.

a) State reduction

Becchi and Crowly [2] introduce Hybrid Finite Automata (HFA). DFA are known for being efficient but space-consuming whereas NFA are small but time-consuming. HFA aim at retaining the best of both. In the context of extended regular expressions, *dot-star* (*i.e.* Kleene closure of the disjunction of the whole alphabet) and fixed-length repetitions of expressions are the main cause of state explosion during DFA computation. DFA computation from NFA is classically made with subset construction. Becchi and Crowly are able identify during subset construction when state explosion will happen and stop it by keeping a non-deterministic piece of the produced HFA. Their experiments shows an average memory storage requirement comparable to that of a NFA and an average memory bandwidth requirement similar to that of a DFA.

Kumar et al. [34] propose three different optimization of state number orthogonal to each other hence simultaneously implementable.

1. Usually, only a few states of the automata are very often explored. Kumar et al. computes the few often explored states into a fast DFA and the others into a NFA.

2. When a DFA construction lead to a state explosion, some parts are often repeated. This may be avoided by using a history, hence invention of History-based Finite Automata (H-FA).

3. k-repetition of an expression r in extended RE lead to a k-multiplication of the number of states representing r. Such a multiplication could be easily avoided with a counter thereby creating History based counting Finite Automata (H-cFA).

Smith, Estan and Jha [57] introduce eXtanded Finite Automata (XFA). Extended regular expressions provides features such as wildcard or fixed-length repetitions that makes generated automata size no longer a function of the regular expression size. XFA extends FA with variables, modified through transitions and checked in final states to confirm matching validity. For example, a XFA matching a hundred repetition of character 'a' would have one state (being both inital and final), a transition increasing a counter and the state checking if the value of the counter is equal to a hundred before validating the matching. For the same purpose, a FA would have a hundred and one state.

b) Transition compressing

Kumar et al. [35] present Delayed Deterministic Finite Automata (D^2FA), an efficient transition compression in the context of deep packet inspection. D^2FA are defined as DFA extended with a new kind of transition, unlabeled, referred as *default transition*. During matching, if no transitions are labeled with the current input character, the default transition is followed but the input character is not consumed, it is retained or put in another way, *delayed*. After the default transition is followed, output transitions of the new current state are matched with the delayed input character. Their experiments shows a data saving of 95% although transformation from DFA to D^2FA is NP-hard.

c) Alphabet compression

Kong, Smith and Estan [30] Introduce the use of several Alphabet Transition Tables (ACTs) to reduce the size of transitions table attached to individual states. An ACT partition alphabet in equivalence class with each class having transitions with identical behavior for a group of states $S \subset Q$. Previously, only one ACT was computed ($S = Q$). Kong et al. had the idea of partitioning Q with each partition having an ACT associated thereby improving the compression. Having

a lot of ACT saves memory by a factor between 4 and 70 but slow the matching by 35% to 85%. This method was also experimented together with (D^2FA) [35] since their approach is orthogonal.

d) Regular expressions rewrites

Yu et al. [61] describe two classes of DFA suffering from state explosion during their generation from regular expression. These two classes are the quadratic and the exponential state explosion. Regular expressions considered in this paper are extended regular expression, including wildcard and quantified repetitions which are in particular responsible for state explosion. A rewrite rule is presented for each class and according to their experiment, both yield a 99% state number reduction.

e) Matching time reduction

Kumar et al. [33] improve Delayed DFA (D^2FA) previously discussed in section b) with Content addressing (CD^2FA). CD^2FA replace state identifiers with content labels that include information about the output state of the default transition. Contents labels allow to skip the travel time used by a default transition exploration thereby achieving the same throughput as the original DFA for much less space used thanks D^2FA implementation. Besides, the more compact a data structure is, the less cache miss happen. Their experiments shows that with a 1KB cache, CD^2FA achieve twice the throughput of an uncompressed DFA while only requiring a tenth of its space.

3.2.2 Parallel optimizations

In most research papers, parallel regular expression matching keep the DFA in its sequential form but operate in parallel over it. A few transform the automata in a parallel form suitable for parallel matching.

a) Automata parallel run

Ladner and Fischer [36] used transducers to compute parallel prefix. Transducers are automata which produce output along the input consumption. This article is known as the first description of parallel run of transducers.

Holub and Stekr [24] detailed implementations of parallel DFA run for distributed memory in cluster of workstation and (symmetric) shared memory (multiprocessors). In this method, the automata is not modified and the input is distributed in blocs for each process. The difficulty for each process is thus to know at which state the previous process automata will end it's matching to know where to start. Only the first process knows for sure which state is its initial one. If each process wait for the previous one to end it's computation then it will be as slow as (or slower than) a sequential matching. Holub and Stekr method involves a matching from all states (as if all states were initial states) for each processes. Assume states are identified with contiguous natural integers starting by 0. At the end of the matching, each process will have a vector of size the number of states with, at position i, the state at which the computation would end if started at state i. A parallel reduction of that vector is all that is needed to join the results to know whether the last state, after matching, is final.

This method will be later referred as *enumeration* method because it enumerates results for all possible initial states or *speculative simulation* as computations for each states as possibly initial may be called simulation. This method was reused with different techniques.

Holub and Stekr [24] also proposed their own improvement for a category of automata they named *synchronizing automata*. An automaton is synchronizing if the matching of any word of at least a certain length k from any state q will always result in the same state q'. With this property, reading k characters more before each processes actual input part allow them to *synchronize* the DFA into correct initial state before starting their own matching. This prevent from having to try all states as initial, only q' is known as the good one.

Memeti and Pllana [48] created a tool named PaREM (standing for *Parallel Regular Expression Matching*). This tool is based on the parallel DFA run method (as [24]) but presents a novel improvement to greatly reduce the number of possible local initial states. The input is also distributed in block and each processor looks at 1 more character before its own input chunk, noted w_{k-1}. Note S the states having an outgoing edge labeled with w_{k-1}. Note L the states having an incoming edge labeled with w_k (the following character of w_{k-1}). Possible local initial states are thus $I = S \cap L$, which is a great improvement compared to Q.

Fu, Liu and Li [16] implemented their own tool named ParaRegex for deep inspection in network security systems. Their approach is also an improvement

of the enumeration method. They start from postulate that during enumeration, after a couple of character read and regardless of the input state, the distinct *active* states (current state during matching) are greatly fewer than the initial states. They claim that the average number of active state after one character read is less than one percent of the total state number. Thus, one computation per active states and not for all possible initial states (which is Q for enumeration) would be much faster, provided that each active states remembers all initial states leading to itself. Bit vectors are used to code the original initial state of each active states. Two optimizations are also presented, *smart split* and *quick start*. Smart split allow each processors to look into a reasonably few number of characters before their own input block to select the one which will decrease the number of active states the most. Quick start uses a different data structure for the first characters read to relieve the overhead of having Q bit vector of size Q, of which most will merge very fast.

Luchaup et al. [45] present speculative parallel regular expression matching. Rather than considering all states as potential initial states as the enumeration method, they bet on the most probable local initial state. This method seems to have two major drawbacks : (1) speculating has only one in the number of state chance to succeed and (2) if the bet is not successful, the matching has to backtrack all over again and at best the speed would be as fast as a sequential matching. However, they have a postulate similar to [16] : only a few states are *hot* (*i.e.* highly often used during matching) so the bet has a much higher chance to succeed rather than the rate said in (1). Moreover, in case speculation is not successful, Luchaup et al. maintain a history of states explorer so that after backtrack, if the sate visited is the same as the state in the history (at the same step), then the matching can stop and the state reached will be the final state reached in the history. History helps to greatly mitigate (2).

Najamn Younis and Rasool [51] developed speculative parallel regular expression matching with other techniques. Multibyte matching is used to increase matching throughput as it was done in another work of Luchaud et al. [46]. Najamn et al. use k-stride DFA for multibyte matching in which transition are not labeled with symbols anymore but with words of length k. Although this technique increases throughput, the memory used explodes because the number of transitions is raised at power k. To mitigate this explosion, transition size is reduced with alphabet compression table (ACT). Transitions of k-stride DFA often

follows a trend: transitions with the same label w often leads to the same state q while others labeled w' also lead to q. These trends allow to group transitions according to their label thereby leading to an efficient compression.

b) Parallel automata

Sin'ya, Matsuzaki and Sassa [56] introduce *Simultaneous finite automata* (SFA). SFA are computed from DFA and retain all simulations of enumeration method in its states which are mapping of the DFA states. Compared to enumeration method, parallel matching with SFA only needs to compute one transition per input character read and after input reading, reduction is just as efficient. The drawbacks lie in the much bigger size of the SFA compared to the original DFA.

3.2.3 Dedicated hardware

Regular expression matching (REM) is such an important challenge that compatibility with costly dedicated hardware is extensively studied. Major principles behind the use of Field-Programmable Gate Array for parallel REM are reported here. We also describe how the recent Ternary Content Addressable Memory hardware is relevant for REM.

a) FPGA

Sidhu and Prasanna [55] demonstrate how Field-Programmable Gate Array (FPGA) can significantly improve performance of regular expression matching. In particular, it is shown that FPGA requires an equal time for DFA and NFA matching. Indeed, NFA are implemented in logical gates, thus transitions with the same label (l) are set in parallel thereby achieving a simultaneously exploring of all the l-labeled transitions. Sidhu and Prasanna also present their own algorithm of NFA generation from regular expression run in hardware. Their algorithm input is a regular expression in postfix form (operators written after their operands) and a stack based data structure allow them to construct the NFA in time linear with the regular expression size.

Moscola et al. [50] takes the opposite direction of Sidhu and Prasanna [55] by matching with DFA instead of NFA whereas their implementation also rely on FPGA. Also, contrary to [55] who targeted an efficient matching of one regular expression, the context of deep packet inspection of [50] requires the matching of several regular expression. Moscola et al. do not care about the time or space

required for constructing the automata, only the space occupied by the final automata is meaningful. Their experiments show that minimized DFA are most of the time smaller than NFA quickly computed from the same regular expression. Thus, DFAs computed from several regular expression are run in parallel thanks to FPGA properties.

Brodie, Taylo and Cytron [5] present a scalable architecture for high-throughput regular expression matching suitable for both FPGA and ASIC. Their approach, apparently similar to [50], targets a set of regular expression and rely on minimized DFA. However, their real improvement for high throughput rely on multibyte matching *i.e.* automata transitions are not labeled with a symbol but a word, theoretically multiplying throughput by the word length. Multibyte matching is know for exploding the transition number of the automata, hence the need to compress them. Compression is achieved here by grouping symbols with similar behavior in equivalence class in a way similar to [51].

b) TCAM

Meiners et al. [47] demonstrate how Ternary Content Addressable Memory (TCAM) is a valuable asset in regular expression matching for deep packet inspection. TCAM are ternary, a TCAM bit value may be '0','1' or '*', '*' standing for (matching) either 0 or 1. This third value helps in encoding several transitions within one entry (for example, entry '0*' matches '00' and '01') thereby compressing greatly the automata by decreasing the number of transition entry. As seen previously in [51] and [5], compression of transition is very useful for multibyte matching which is exactly what is done by Meiners et al.

4 From BSP Regular Expression to BSP Automata

Contents

In this chapter we introduce our algorithm to generate the BSPA that accepts the same language as a given enumerated BSPRE. This transformation is fundamental for the BSP Automata theory as it allows to construct a BSPA whose language is known. It is to BSPA theory what is Thompson's [59] or Glushkov's [8] algorithm to finite automata theory. This transformation is also the first step in our parallel regular expression matching scheme detailed in Chapter 6.

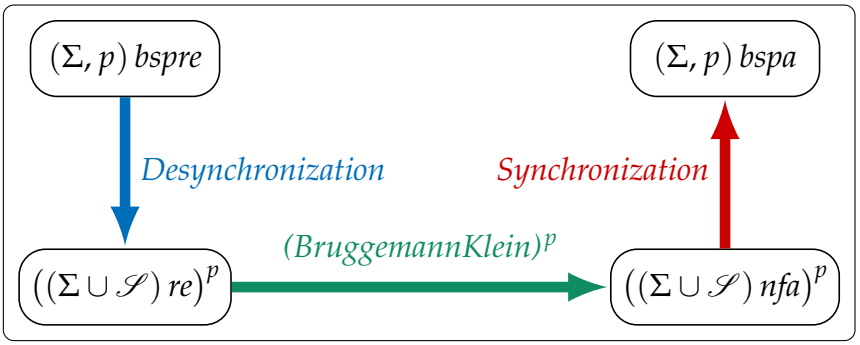


Figure 4.1 – *Schema of BSPRE to BSPA algorithm*

BSPA are defined for a fixed p which allows intentional BSPRE to be trivially transformed into their equivalent enumerated form. So if the algorithm works for the enumerated form, it will work for the intentional one. Therefore, for this algorithm and the whole chapter, BSPRE will be enumerated. This algorithm is

divided in three parts represented in Figure 4.1, each of them explained respectively in section 4.1, 4.2 and 4.3. This chapter is closed with an example of BSPA generation from a simple BSPRE in section 4.4.

For reference, we give a detailed structure of the whole algorithm with a zoom of the abstract schema Figure 4.1 in Figure 4.2.

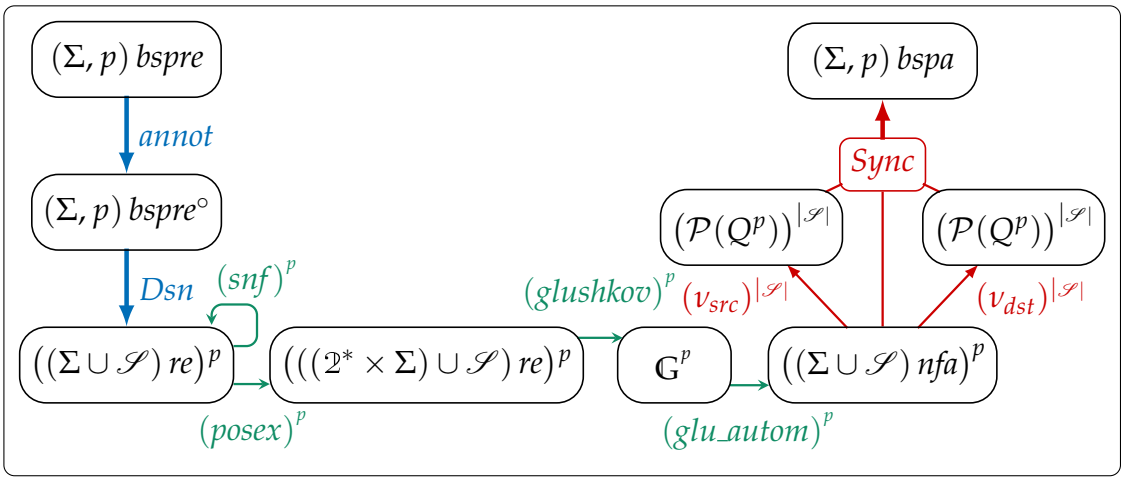


Figure 4.2 – *Detailed schema of BSPRE to BSPA algorithm*

Functions of Figure 4.2 are over-viewed below.

annot : Annotates each occurrence of vector constructor $\langle \ldots \rangle$ with a unique identifier 31

Dsn : Transforms a BSPRE into a RE vector which preserves BSPRE structure and vector annotation 31

snf : Normal form of re that makes *glushkov* faster 37

posex : Localizes each symbol of a re so that they are distinct from each other 33

glushkov : Computes position of symbols relatively to each other in words of the language 34

glu_autom : Computes an nfa with one state per symbol and transitions deduced from *glushkov* result 36

$\nu_{src/dst}$: Computes vector source (resp. destination) of Δ, the global synchronization function of the bspa encoded by the given vector of nfa. Used for each vector annotation 39

Sync : Computes the bspa from the nfa vector and Δ from the $\nu_{src/dst}$ result. 39

Types of functions appearing in Figure 4.2 are briefly explained in Table 4.1

Σ *re*	Regular expression with alphabet Σ
Σ *nfa*	Non-deterministic finite automaton with transitions labeled by symbols in $\Sigma \cup \{\epsilon\}$.
Σ *dfa*	Deterministic finite automaton with transitions labeled by symbols in Σ.
(Σ, p) *bspre*	BSPRE with alphabet Σ and p-sized vectors of regular expression.
(Σ, p) *bspa*	BSPA (not deterministic in general) with alphabet Σ and with p-sized vectors of automata.
(Σ, p) *bspre*$^\circ$	BSPRE whose occurrences of vector constructors $\langle \ldots \rangle$ are annotated by unique identifiers.
$\mathscr{S}$	Set of annotated semicolons $\{;_t \mid t \in \mathbb{N}\}$ such that $;_t$ represents a barrier separator corresponding to (the end of) annotated vector constructor $\langle \ldots \rangle_t$. t is a unique identifier.
$\mathbb{2}$	The set of bit values $\{0, 1\}$.
$\mathbb{P} = (\mathbb{2}^* \times \Sigma)$	Type of localized symbols where the sequence of $\mathbb{2}$ represents the syntactic position (a (left,right)*-path from the root) of the given symbol in a certain regular expression. An empty sequence denotes the syntactic root, a 0 denotes a step to the left (or unique) subterm and a 1 a step to the right subterm.
$\mathbb{G}$	Output type of *glushkov* function. This is a quadruple whose members have the name and type : $\big($*first* : $\mathcal{P}(\mathbb{2}^* \times \Sigma)$, *last* : $\mathcal{P}(\mathbb{2}^* \times \Sigma)$, *null* : $\big\{\{\epsilon\}, \emptyset\big\}$, *follow* : $(\mathbb{2}^* \times \Sigma) \to \mathcal{P}(\mathbb{2}^* \times \Sigma)\big)$
Q	Set of states in the automata. Here, a state will correspond to a localized symbol ($Q = \mathbb{P}$).

Table 4.1 – *Function types used in the transformation from BSPRE to BSPA*

4.1 Desynchronization

The first step of this algorithm is named *Desynchronization* because, given a BSPRE where the end of each vector models a synchronization, this transformation will combine all vectors to output only one at the end. To keep the synchronization information, a semicolon is written at the end of each vector.

From here to the end of this section, we will differentiate between operators (with arity 0, 1, or 2) of BSPRE and scalar regular expressions by writing "BSP" in subscript next to the former.

One important point during the desynchronization, is that we need to know to which vector the synchronization was associated (represented by a semicolon). Therefore we assume that each vector is annotated by a unique identifier: a natural number indicating the position of a vector, given to vectors by reading the BSPRE from left to right. For example, $(\langle a, b\rangle + \langle c, d\rangle)\langle e, f\rangle$ is annotated as $(\langle a, b\rangle_0 + \langle c, d\rangle_1)\langle e, f\rangle_2$.

In the following, t denotes such a position, and S the set of annotations. (Σ, p) *bspre* is the set of annotated BSP regular expressions for alphabet Σ and p processors. We denote the alphabet $\left\{;_t \mid t \in S\right\}$ by $\mathscr{S}$.

The recursively defined function *Dsn* (abbreviate desynchronization) transforms a BSPRE into a regular expression for a given position.

Definition 9 *number* **:** (Σ, p) *bspre* $\to \mathbb{N} \to \left((\Sigma, p)\ bspre^{\circ}\ , \mathbb{N}\right)$

Gives a unique identifier to each vector and attaches the next least available *i.e.* "fresh" integer.

$$number(F \cdot G, t) = \begin{cases} \textbf{let} & (F_1, t_1) = number(F, t) \\ \textbf{and} & (G_1, t_2) = number(G, t_1) \\ \textbf{in} & (F_1 \cdot G_1\ , t_2) \end{cases}$$

$$number(F + G, t) = \begin{cases} \textbf{let} & (F_1, t_1) = number(F, t) \\ \textbf{and} & (G_1, t_2) = number(G, t_1) \\ \textbf{in} & (F_1 + G_1\ , t_2) \end{cases}$$

$$number(F^*, t) = \textbf{let}\ (F_1, t_1) = number(F, t)\ \textbf{in}\ (F_1^*\ , t_1)$$

$$number(\langle r_0, \ldots, r_{p-1}\rangle, t) = (\langle r_0, \ldots, r_{p-1}\rangle_t\ , t + 1)$$

$$number(\epsilon, t) = (\epsilon, t)$$

$$number(\emptyset, t) = (\emptyset, t)$$

Definition 10 *annot* **:** $(\Sigma, p)\ bspre \to (\Sigma, p)\ bspre^{\circ}$

Call *number* with the first identifier and remove the counter of the result

$$annot(R) = fst(number(R, 0))$$

Definition 11 *dsn* **:** $[p] \to (\Sigma, p)\ bspre^{\circ} \to (\Sigma \cup \mathscr{S})\ re$

Transforms a BSPRE to a RE. When a vector is encountered, a projection is made. In this function, a distinction is made between RE operators and BSPRE ones (BSP in subscript for the latter).

$$\begin{aligned}
dsn^i(F \cdot_{BSP} G) &= dsn^i(F) \cdot dsn^i(G) \\
dsn^i(F +_{BSP} G) &= dsn^i(F) + dsn^i(G) \\
dsn^i(F^{*_{BSP}}) &= dsn^i(F)^* \\
dsn^i(\langle r_0, \ldots, r_{p-1} \rangle_t) &= r_i \cdot_; t \\
dsn^i(\epsilon_{BSP}) &= \epsilon \\
dsn^i(\varnothing_{BSP}) &= \varnothing
\end{aligned}$$

The semicolon keeps synchronization existence while t differentiates between them. The next function is the main one that calls the previous one for each position of a p size vector. *Dsn* and *dsn* abbreviate *desynchronization*.

Definition 12 *Dsn* **:** $(\Sigma, p)\ bspre^{\circ} \to \big((\Sigma \cup \mathscr{S})\ re\big)^p$

Creates a vector whose content results from calls to the previous function for each position with the position as parameter.

$$Dsn(R) = \langle dsn^0(R), \ldots, dsn^{p-1}(R) \rangle$$

Definition 13 *Dsync* **:** $(\Sigma, p)\ bspre \to \big((\Sigma \cup \mathscr{S})\ re\big)^p$

$$Dsync = Dsn \circ annot$$

Eventually, we get a vector of regular expressions whose alphabet is enlarged with annotated semicolon, $\Sigma_{dsn} = \Sigma \cup \mathscr{S}$.

4.2 From regular expression to finite automata

To apply the second step, an algorithm was needed that, given a regular expression, gives an automaton that recognizes (*i.e.* match) a word of its language. There exist many algorithms to do this transformation and the fastest computes NFAs. We chose Brüggemann-Klein's algorithm's[6] with the following characteristics (with n the size of input regular expression):

- Optimal time complexity: $O(n^2)$.
- Automata size: $O(n)$ (one state for each symbol occurrence).
- No ϵ-transitions.
- One initial state.

This algorithm is an improvement of the Glushkov's algorithm [8] that basically computes for each symbol which ones may follow. Symbols become states, and those that may follow become destination of a transition from the former to themselves. This is the reason why we relied on this algorithm, the reason why we could introduce a new symbol (the semicolon) and know which is the one and only state it refers to.

There are a lot of set unions during computation of Glushkov automaton and in order to make them disjoint, Brüggemann-Kleinintroduced the so-called *Star-Normal Form*.

Before applying the Glushkov algorithm, the regular expression is transformed into star-normal form which preserves the language and changes from cubic to quadratic complexity the next algorithm through making unions disjoint.

There exists also an optimal parallel algorithm presented by Ziadi [62] to do this transformation. However, the chosen algorithm is processed in parallel on each vector component, thus, using a parallel algorithm will not benefit our transformation.

After application of the Brüggemann-Klein's algorithm to all regular expressions, we get:

$$\forall i \in [p],\ A^i = (Q^i, \Sigma_{dsn}, \delta^i_{dsn}, I^i, F^i)$$

where A^i is an NFA.

Whether we want to determinize (remove indeterminism) or minimize our automata, if a transformation is desired, it is now possible to map this transformation on the obtained automaton vector. Evidently, determinizing a desynchronized automaton is not guaranteed to determinize the BSPA obtained after complete transformation.

Definition 14 $\bullet$ **:** $\mathbb{2} \to (\mathbb{2}^* \times \Sigma)\,re \to (\mathbb{2}^* \times \Sigma)\,re$

Left extend of syntactic position.

$$\begin{aligned} b \bullet (F \cdot G) &= (b \bullet F) \cdot (b \bullet G) \\ b \bullet (F + G) &= (b \bullet F) + (b \bullet G) \\ b \bullet F^* &= (b \bullet F)^* \\ b \bullet (b', a) &= (b \cdot b', a) \\ b \bullet \epsilon &= \epsilon \\ b \bullet \emptyset &= \emptyset \end{aligned}$$

Definition 15 *posex* **:** $\Sigma\,re \to (\mathbb{2}^* \times \Sigma)\,re$

Localizes symbols of a regular expression.

$$\begin{aligned} posex(F \cdot G) &= \big(0 \bullet posex(F)\big) \cdot \big(1 \bullet posex(G)\big) \\ posex(F + G) &= \big(0 \bullet posex(F)\big) + \big(1 \bullet posex(G)\big) \\ posex(F^*) &= \big(0 \bullet posex(F)\big)^* \\ posex(a) &= (\epsilon, a) \\ posex(\epsilon) &= \epsilon \\ posex(\emptyset) &= \emptyset \end{aligned}$$

Definition 16 *pos* **:** $(\mathbb{2}^* \times \Sigma)\,re \to \mathcal{P}(\mathbb{2}^* \times \Sigma)$

Compute the set of all (localized) symbols of a regular expression.

$$\begin{aligned} pos(F \cdot G) &= pos(F) \cup pos(G) \\ pos(F + G) &= pos(F) \cup pos(G) \\ pos(F^*) &= pos(F) \\ pos((b, a)) &= \{(b, a)\} \\ pos(\epsilon) &= \emptyset \\ pos(\emptyset) &= \emptyset \end{aligned}$$

Definition 17 *glushkov* : $(\mathbb{2}^* \times \Sigma)\, re \to \mathbb{G}$

Computes *first*, *last*, *null* and *follow* which are necessary to compute the automaton. Intuitively (the formal Glushkov's properties are on p.35),

first : all symbols that can be first of a word in the language
last : all symbols that can be last of a word in the language
null : has the value $\{\epsilon\}$ if ϵ is in the language, $\emptyset$ otherwise.
follow : all symbols that can follow a given symbol of a word in the language

$$glushkov(\overline{E}) = \begin{pmatrix} first = First'(\overline{E}), \\ last = Last'(\overline{E}), \\ null = Null'(\overline{E}), \\ follow = \lambda x.Follow'(\overline{E}, x) \end{pmatrix}$$

where

def 17.1 $Null'$: $(\mathbb{2}^* \times \Sigma)\, re \to \{\{\epsilon\}, \emptyset\}$

$$\begin{aligned}
Null'(\emptyset) &= \emptyset && (1)\\
Null'(\epsilon) &= \{\epsilon\} && (2)\\
Null'(x) &= \emptyset && (3)\\
Null'(\overline{F}+\overline{G}) &= Null'(\overline{F}) \cup Null'(\overline{G}) && (4)\\
Null'(\overline{F} \cdot \overline{G}) &= Null'(\overline{F}) \cap Null'(\overline{G}) && (5)\\
Null'(\overline{F}^*) &= \{\epsilon\} && (6)
\end{aligned}$$

def 17.2 $First'$: $(\mathbb{2}^* \times \Sigma)\, re \to \mathcal{P}(\mathbb{2}^* \times \Sigma)$

$$\begin{aligned}
First'(\emptyset) &= \emptyset && (1)\\
First'(\epsilon) &= \emptyset && (2)\\
First'(x) &= \{x\} && (3)\\
First'(\overline{F}+\overline{G}) &= First'(\overline{F}) \cup First'(\overline{G}) && (4)\\
First'(\overline{F} \cdot \overline{G}) &= First'(\overline{F}) \cup && (5)\\
&\quad Null(\overline{F}) \cdot First'(\overline{G}) &&\\
First'(\overline{F}^*) &= First'(\overline{F}) && (6)
\end{aligned}$$

def 17.3 $Last'$: $(\mathbb{2}^* \times \Sigma)\, re \to \mathcal{P}(\mathbb{2}^* \times \Sigma)$

$$
\begin{array}{lcll}
Last'(\emptyset) & = & \emptyset & (1)\\
Last'(\epsilon) & = & \emptyset & (2)\\
Last'(x) & = & \{x\} & (3)\\
Last'(\overline{F}+\overline{G}) & = & Last'(\overline{F}) \cup Last'(\overline{G}) & (4)\\
Last'(\overline{F} \cdot \overline{G}) & = & Last'(\overline{G}) \cup & (5)\\
 & & Null(\overline{G}) \cdot Last'(\overline{F}) & \\
Last'(\overline{F}^*) & = & Last'(\overline{F}) & (6)
\end{array}
$$

def 17.4 $Follow'$ **:** $(\mathbb{2}^* \times \Sigma)\, re \to (\mathbb{2}^* \times \Sigma) \to \mathcal{P}(\mathbb{2}^* \times \Sigma)$

$$
\begin{array}{lcll}
Follow'(\emptyset, x) & = & \emptyset & (1)\\
Follow'(\epsilon, x) & = & \emptyset & (2)\\
Follow'(a, x) & = & \emptyset & (3)\\
Follow'(\overline{F}+\overline{G}, x) & = & \begin{cases} Follow'(\overline{F}, x) & \textbf{if } x \in pos(\overline{F}) \\ Follow'(\overline{G}, x) & \textbf{if } x \in pos(\overline{G}) \end{cases} & (4)\\
Follow'(\overline{F} \cdot \overline{G}, x) & = & \begin{cases} Follow'(\overline{F}, x) \cup First(\overline{G}) & \textbf{if } x \in Last(\overline{F}) \\ Follow'(\overline{F}, x) & \textbf{if } x \in pos(\overline{F}) \setminus Last(\overline{F}) \\ Follow'(\overline{G}, x) & \textbf{if } x \in pos(\overline{G}) \end{cases} & (5)\\
Follow'(\overline{F}^*, x) & = & \begin{cases} Follow'(\overline{F}, x) \cup First(\overline{F}) & \textbf{if } x \in Last(\overline{F}) \\ Follow'(\overline{F}, x) & \textbf{if } x \in pos(\overline{F}) \setminus Last(\overline{F}) \end{cases} & (6)
\end{array}
$$

Remark 17.1. The recursive clauses defining $Follow(\_,\_)$ do not intersect each other because x is a *localized* symbol, and therefore occurs in at most one of the sub-expressions. For example in $Follow(\overline{F}+\overline{G}, x)$, it is impossible to have *both* $x \in \overline{F}$ and $x \in \overline{G}$ because x, being a localized symbol, refers to a unique occurrence in the global regular expression.

Glushkov's properties

prop 17.1 ($Null$) **:**

$$Null(\overline{E}) = \begin{cases} \{\epsilon\} & \text{if } \epsilon \in L(\overline{E}) \\ \emptyset & \text{otherwise} \end{cases}$$

prop 17.2 (*First*) **:**

$$First(\overline{E}) = \left\{ x \in (\mathbb{2}^* \times \Sigma) \mid \exists w \in (\mathbb{2}^* \times \Sigma)^* : xw \in L(\overline{E}) \right\}$$

prop 17.3 (*Last*) **:**

$$Last(\overline{E}) = \left\{ x \in (\mathbb{2}^* \times \Sigma) \mid \exists w \in (\mathbb{2}^* \times \Sigma)^* : wx \in L(\overline{E}) \right\}$$

prop 17.4 (*Follow*) **:**

$$Follow(\overline{E}, x) = \left\{ y \in (\mathbb{2}^* \times \Sigma) \mid \exists v \in (\mathbb{2}^* \times \Sigma)^*, \exists w \in (\mathbb{2}^* \times \Sigma)^* : vxyw \in L(\overline{E}) \right\}$$

Proof It is proven in section A.1 that the previous definitions satisfies the properties (*i.e.* $Null = Null', First = First', Last = Last'$ and $Follow = Follow'$). From this point onwards, they will be referred to with the same name (without the prime, *e.g.* *prop. Follow* and *def. Follow*).

□

Definition 18 *glu_autom* **:** $\Sigma\, re \to \mathbb{G} \to (Q, \Sigma, \delta, I, F)$

Uses the former regular expression with *first*, *last*, *null*, *follow* to compute the automaton.

let q_I a fresh new state.

$$\begin{cases} Q = pos(posex(E)) \cup \{q_I\} \\ \Sigma = \Sigma \\ \delta = \begin{cases} \forall a \in \Sigma, \delta(q_I, a) = \left\{ x \mid x \in first, snd(x) = a \right\} \\ \forall x \in (\mathbb{2}^* \times \Sigma), \forall a \in \Sigma, \delta(x, a) = \left\{ y \mid y \in follow(x), snd(y) = a \right\} \end{cases} \\ I = \{q_I\} \\ F = \begin{cases} last \cup \{q_I\} & \textbf{if } null = \{\epsilon\} \\ last & \textbf{otherwise} \end{cases} \end{cases}$$

Star Normal Form (SNF) is introduced by Brüggemann-Klein[6]. Normal form of a regular expression used as preprocessing to make *def. glushkov* faster. Indeed, during processing of *def. Follow*, the union in case (5) is necessarily disjoint (no elements are found in both sets). However, the union in case (6) may not be disjoint and forces to ensure that no duplicates exists which is costly. Yet,

a regular expression in SNF makes this union disjoint. See ex. 1 for an example. $L(E) = L(snf(E))$ is proved in [6].

Definition 19 $snf : \Sigma re \to \Sigma re$

$$snf(E) = E^{\bullet}$$

where

$Null^{\mathbb{B}}(E) =$ **if** $Null(E) = \{\epsilon\}$ **then** $\top$ **else** $\bot$

$E^{\bullet} =$ **Match** E **with**

$$\begin{array}{rcl} \emptyset & \to & \emptyset \\ \epsilon & \to & \epsilon \\ x & \to & x \\ F{+}G & \to & F^{\bullet}{+}G^{\bullet} \\ F \cdot G & \to & F^{\bullet} \cdot G^{\bullet} \\ F^{*} & \to & (F^{\circ\bullet})^{*} \end{array}$$

$E^{\circ\bullet} =$ **Match** E **with**

$$\begin{array}{rcl} \emptyset & \to & \emptyset \\ \epsilon & \to & \emptyset \\ x & \to & x \\ F{+}G & \to & F^{\circ\bullet}{+}G^{\circ\bullet} \\ F \cdot G & \to & \begin{cases} F^{\bullet} \cdot G^{\bullet} & \textbf{if } \neg Null^{\mathbb{B}}(F) \wedge \neg Null^{\mathbb{B}}(G) \\ F^{\circ\bullet} \cdot G^{\bullet} & \textbf{if } \neg Null^{\mathbb{B}}(F) \wedge Null^{\mathbb{B}}(G) \\ F^{\bullet} \cdot G^{\circ\bullet} & \textbf{if } Null^{\mathbb{B}}(F) \wedge \neg Null^{\mathbb{B}}(G) \\ F^{\circ\bullet}{+}G^{\circ\bullet} & \textbf{if } Null^{\mathbb{B}}(F) \wedge Null^{\mathbb{B}}(G) \end{cases} \\ F^{*} & \to & F^{\circ\bullet} \end{array}$$

Example 1 : let $r = (a^*b^*)^*$ a regular expression. There is no need to localize r as a and b only have one occurrence of each. Let us go through the computation of which symbols may follow a a in r.

$$\begin{aligned} Follow(r,a) &= Follow((a^*b^*)^*, a) \\ &= \underbrace{Follow(a^*b^*, a)}_{(1)} \cup \underbrace{First(a^*b^*)}_{(2)} \qquad [\, a \in Last((a^*b^*)) \,] \end{aligned}$$

The two operands are computed separately to observe whether the union is disjoint or not.

$$\begin{aligned} (1)\ Follow(a^*b^*, a) &= Follow(a^*, a) \cup First(b^*) && [\, a \in Last(a^*) \,] \\ &= Follow(a, a) \cup First(a) \cup First(b) && [\, a \in Last(a^*) \,] \\ &= \emptyset \cup \{a\} \cup \{b\} \\ &= \{a, b\} \end{aligned}$$

$$
\begin{aligned}
(2)\ First(a^*b^*) &= First(a^*) \cup First(b^*) && [\ Null(a^*) = \{\epsilon\}\ ] \\
&= First(a) \cup First(b) \\
&= \{a\} \cup \{b\} \\
&= \{a, b\}
\end{aligned}
$$

After computation of both operands, it appears the union is not disjoint because elements in one are found in the other (actually even all of them). We go through the same process after computation of SNF.

$$
\begin{aligned}
snf(r) &= (a^*b^*)^{*\bullet} \\
&= (a^*b^*)^{\circ\bullet *} \\
&= (a^{*\circ\bullet} + b^{*\circ\bullet})^* && [\ Null^{\mathbb{B}}(a^*) \wedge Null^{\mathbb{B}}(b^*)\ ] \\
&= (a^{\circ\bullet} + b^{\circ\bullet})^* \\
&= (a + b)^*
\end{aligned}
$$

We compute followers of a in $snf(r)$.

$$
\begin{aligned}
Follow(snf(r), a) &= Follow((a + b)^*, a) \\
&= Follow(a + b, a) \cup First(a + b) && [\ a \in Last((a^*b^*))\ ] \\
&= Follow(a, a) \cup First(a) \cup First(b) \\
&= \emptyset \cup \{a\} \cup \{b\} \\
&= \{a, b\}
\end{aligned}
$$

We remark that all unions were disjoint in this computation. In fact, It was proved in [6] that all unions are always disjoint in the computation of followers in a regular expression in star normal form. This allow the use of a simple data structure with unions in constant time without worrying about duplicates to remove which would makes unions in $O(n \cdot log(n))$ at best.

Definition 20 $BK : \Sigma\, re \to \Sigma\, nfa$

Brüggemann-Klein's algorithm computes a NFA from a regular expression

$$BK(E) = glu\_autom(E) \circ glushkov \circ posex \circ snf(E)$$

4.3 Synchronization

We define two functions *src* and *dst* that select all transitions labeled with semicolons and respectively give the set of source state and destination state of these transitions.

Definition 21 $\delta^* : Q \to \Sigma^* \to Q$

$$\delta^*(q, u_0 u_1 \dots u_m) = \delta(\dots(\delta(\delta(q, u_0), u_1) \dots, u_m)$$

Definition 22 $src : \mathbb{N} \to \big((Q \times (\Sigma \cup \mathscr{S})) \to Q\big) \to \mathcal{P}(Q)$

Get the source state of each semicolon transition annotated with a given t

$$src^t(\delta_{dsn}) = \Big\{ q \mid \exists q' \in Q, \delta_{dsn}(q, ;_t) = q' \Big\}$$

Definition 23 $dst : \mathbb{N} \to \big((Q \times (\Sigma \cup \mathscr{S})) \to Q\big) \to \mathcal{P}(Q)$

Get the destination state of semicolon transitions annotated with a given t

$$dst^t(\delta_{dsn}) = \Big\{ q' \mid \exists q \in Q, \delta_{dsn}(q, ;_t) = q' \Big\}$$

The next two functions give the set of all possible combination of sources (or destinations) state of semicolon transition t for all transitions in every automata of an automata vector. The set of source (resp. destination) vector models the set of possible configuration before (resp. after) the synchronization barrier t.

Definition 24 $v_{src}, v_{dst} : \mathbb{N} \to \big((Q \times (\Sigma \cup \mathscr{S})) \to Q\big)^p \to \mathcal{P}(Q^p)$

Map *src*/*dst* on each of the p automata transition function and combine results with a Cartesian product.

$$\begin{aligned} v^t_{src}(\langle \delta^i_{dsn} \rangle_{i \in [p]}) &= src^t(\delta^0_{dsn}) \times \dots \times src^t(\delta^{p-1}_{dsn}) \\ v^t_{dst}(\langle \delta^i_{dsn} \rangle_{i \in [p]}) &= dst^t(\delta^0_{dsn}) \times \dots \times dst^t(\delta^{p-1}_{dsn}) \end{aligned}$$

The synchronization function gathers all automata of the input vector by computing Δ from semicolon transition and thereby removes semicolon from the alphabet.

Definition 25 $Sync : \big((\Sigma \cup \mathscr{S})\ nfa\big)^p \to (\Sigma, p)\ bspa$

Computes a BSPA from a vector of NFA whose labels includes semicolons representing the end of vectors from the original BSPRE.

$$Sync(\ (\langle Q^i\rangle_{i\in[p]}, \Sigma_{dsn}, \langle\delta^i_{dsn}\rangle_{i\in[p]}, \langle I^i\rangle_{i\in[p]}, \langle F^i\rangle_{i\in[p]}))$$

$$= (\langle Q^i\rangle_{i\in[p]}, \Sigma, \langle\delta^i\rangle_{i\in[p]}, \langle I^i\rangle_{i\in[p]}, \langle F^i\rangle_{i\in[p]}, \Delta)$$

where

$$\Delta = \bigcup_{t\in|\mathscr{S}|}\left(\bigcup_{\vec{q}\in arg^t}\bigcup_{\vec{q'}\in res^t}\left(\vec{q}\to\vec{q'}\right)\right)$$

$$\delta^i = \delta^i_{dsn} \setminus \bigcup_{;_t\in\mathscr{S}}\left((q, ;_t) \to q'\right) \in \delta^i_{dsn}$$

$$\Sigma = \Sigma_{dsn} \setminus \mathscr{S}$$

$$arg^t = \nu^t_{src}(\langle\delta^i_{dsn}\rangle_{i\in[p]})$$

$$res^t = \nu^t_{dst}(\langle\delta^i_{dsn}\rangle_{i\in[p]})$$

Remark 25.1. During desynchronization ($Dsync : (\Sigma, p)\ bspre \to \big((\Sigma\cup\mathscr{S})\ re\big)^p$), the alphabet is enlarged with $\mathscr{S}$ and becomes the same for each RE of the vector. Then, the alphabet is preserved during BK which means, the p alphabets are identical and their value is named Σ_{dsn}. For this reason, Σ_{dsn} is written instead of $\langle\Sigma^i_{dsn}\rangle_{i\in[p]}$ as argument of *Sync*.

4.4 Algorithm example

For a better understanding, we illustrate our algorithm with the following example. We choose this simple BSPRE as input of our algorithm:

$$R = (\langle a\ ,\ b\rangle + \langle c\ ,\ d\rangle)\langle e\ ,\ f\rangle$$

This expression represents the BSP language $\{\langle a,b\rangle\langle e,f\rangle\ ,\ \langle c,d\rangle\langle e,f\rangle\}$

Desynchronisation applied to R results in the following regular expression vector where the semicolon keeps synchronization information ($;_t$ marks the end of vector number t which means the synchronization barrier t).

$$Dsn(R) = \langle (a\ ;_0 + c\ ;_1)\ e\ ;_2\ \ ,\ \ (b\ ;_0 + d\ ;_1)\ f\ ;_2\ \rangle$$

Brüggemann-Klein's algorithm is mapped to each vector components and results in the automata vector of Figure 4.3. Legend of the NFA (and BSPA after) is as described section 2.4.3, initial states are hexagon and final states are double circles. Computation and drawings were automatically made by our software.

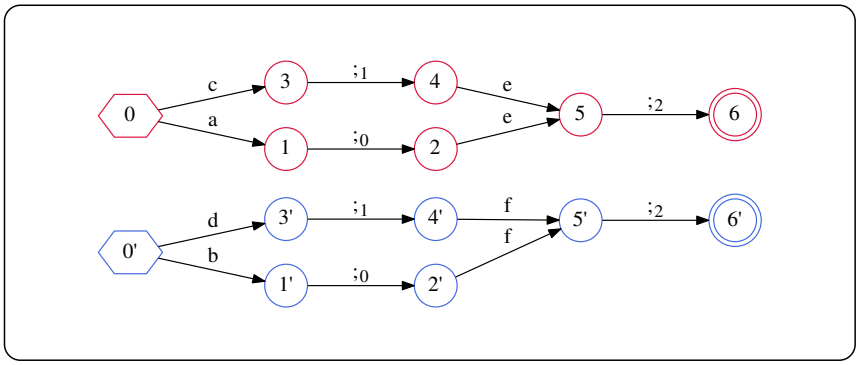


Figure 4.3 – *BSP automaton desynchronized*

Synchronization of above automata leads to the following BSPA (Figure 4.4) where all $;_t$ are glued together to synchronize automata with Δ.

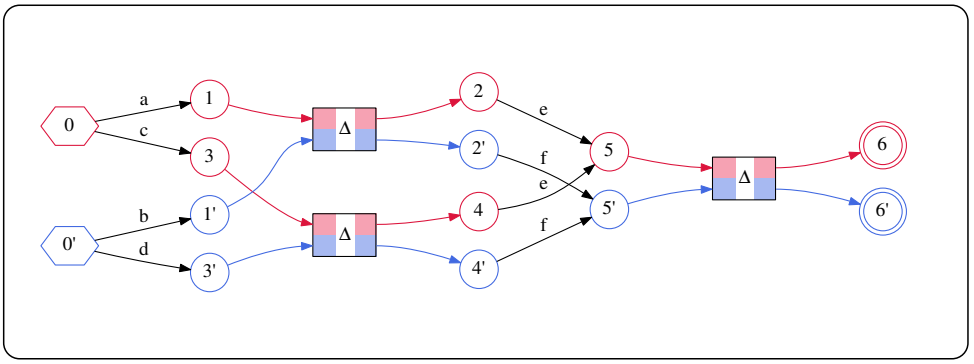


Figure 4.4 – *BSP automaton synchronized*

Determinization of BSP Automata 5

Contents

BSPA produced from BSPRE by the transformation introduced in Chapter 4 are (generally) not deterministic. In order to make BSPA more than a theoretical tool, the first step is the determinization. Determinism represents annihilation of ambiguity and prevents backtracking from occurring during matching thereby avoiding any loss of efficiency.

In this chapter, non-deterministic BSPA will be referred as NBSPA and deterministic BSPA as DBSPA. This chapter starts by enunciating the classical determinization of non-deterministic finite automata to solve local indeterminism of NBSPA in section 5.1. Follows the crux of the problem: the global indeterminism of which solution is provided in section 5.2. Articulation of these is detailed in section 5.3.

5.1 Local indeterminism

Before introducing the determinization of BSPA, we clarify here determination of non-deterministic finite automata (NFA). It serves as both a reminder because BSPA determinization shares a similar style and a subroutine definition for the latter. From here onwards, finite automata types (*i.e. (n+d)fa* and *(n+d+ϵ)bspa*) will no longer be parametrized by their alphabet, which will remain the same during determinization, but by their states.

An important function of determinization is the closure of ϵ-transitions. It return states given in argument with those connected by ϵ-transitions. It is also the least fixed point of the following operator on sets of states Q.

Definition 26 ϵ-closure : $2^Q \to (Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q) \to 2^Q$

$$\epsilon\text{-closure}(Q, \delta) = Q \cup \left(\bigcup_{q \in Q} \epsilon\text{-closure}(\delta(q, \epsilon), \delta) \right)$$

Unlike functions presented previously which were written in functional fashion, determinization algorithm is widely known in its imperative form thus the remaining of this chapter is written with algorithm written in imperative fashion.

We note determinized states (q), functions (δ) and set of states (Q) with the notation q_d, δ_d and Q_d respectively. The following algorithm is known as the Scott-Rabin algorithm. It was not put in preliminaries because our algorithm, following right after this one, lean greatly on it and have a very similar structure.

Definition 27 *determinize* : Q *nfa* $\to 2^Q$ *dfa*

```
input  : (Q, Σ, δ, I, F)
output : (Q_d, Σ, δ_d, s_d, F_d)
begin
    s_d ← ε-closure(I, δ)
    W ← {s_d}
    while W ≠ ∅ do
        q_d ← select_from(W)
        W ← W \ {q_d}
        for each a ∈ Σ do
            p_d ← ε-closure(⋃_{q∈q_d} δ(q, a), δ)
            δ_d(q_d, a) ← p_d
            if p_d ∉ (W ∪ Q_d) then
                W ← W ∪ {p_d}
        done
        Q_d ← Q_d ∪ {q_d}
    done
    F_d ← { q_d ∈ Q_d | ∃q ∈ q_d, q ∈ F }
end
```

5.2 Global indeterminism

Determinization of NBSPA presents a new difficulty that does not exist for NFA. *Local indeterminism* is an ambiguity for δ-transitions and was treated in section 5.1 with determinization for NFA. However, *global indeterminism*, which arise when a delta transition has different outputs for the same input cannot be resolved by a mere merging of outputs.

5.2.1 Problem statement

Take a look at Figure 5.1. In subscript is written the identifier of the automata vector component.

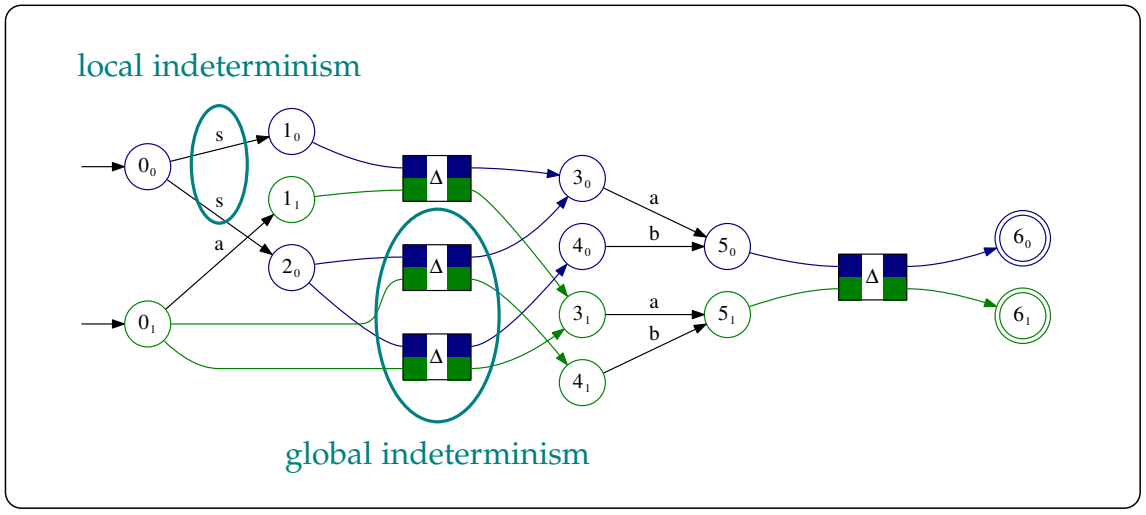


Figure 5.1 – *NBSPA with language* $L = \{\langle s,a\rangle\langle a,a\rangle, \langle s,\epsilon\rangle\langle a,b\rangle, \langle s,\epsilon\rangle\langle b,a\rangle\}$

If output of $\Delta(\langle 2_0, 0_1\rangle)$ was the merge of states 3_0 and 4_0 as well as 3_1 and 4_1 then language would become:

$$L = \{\langle s,a\rangle\langle a,a\rangle, \langle s,\epsilon\rangle\langle a,b\rangle, \langle s,\epsilon\rangle\langle b,a\rangle, \langle s,\epsilon\rangle\langle a,a\rangle, \langle s,\epsilon\rangle\langle b,b\rangle\}$$

Instead of $L = \{\langle s,a\rangle\langle a,a\rangle, \langle s,\epsilon\rangle\langle a,b\rangle, \langle s,\epsilon\rangle\langle b,a\rangle\}$. Language is locally unchanged but globally, the new language is the Cartesian product of the union component-wise of the previous language recognized by the part of the automaton following the merged states. In our case, the union component wise of $\{\langle a,b\rangle, \langle b,a\rangle\}$ is $\langle\{a,b\},\{b,a\}\rangle$ of which Cartesian product of its components is $\{\langle a,b\rangle, \langle b,a\rangle, \langle a,a\rangle, \langle b,b\rangle\}$. This Cartesian product occurs because the distinction brought by Δ is removed. Informally, in the NBSPA, the computation of a non-deterministic Δ-transition lead to one of the state vectors outputs whereas

merging those outputs is the same as allowing the non-deterministic Δ-transition to output any combination of those vectors thereby enlarging the language with those combinations.

5.2.2 Indexing

However, those states must be merged, otherwise global indeterminism will not be resolved. Our solution evolved from the key idea of duplicating the automata for each output of Δ-transitions before determinization. Merging Δ output states after duplication do not change the language because the distinction previously brought by the several outputs of the non-deterministic Δ-transition is translated to the several duplications.

We applied our solution for determinization of Figure 5.1. The result of our determinization is displayed in Figure 5.2. In superscript is written the identifier of the duplication. State names are the concatenation of identifiers of state merged (separated by underscores).

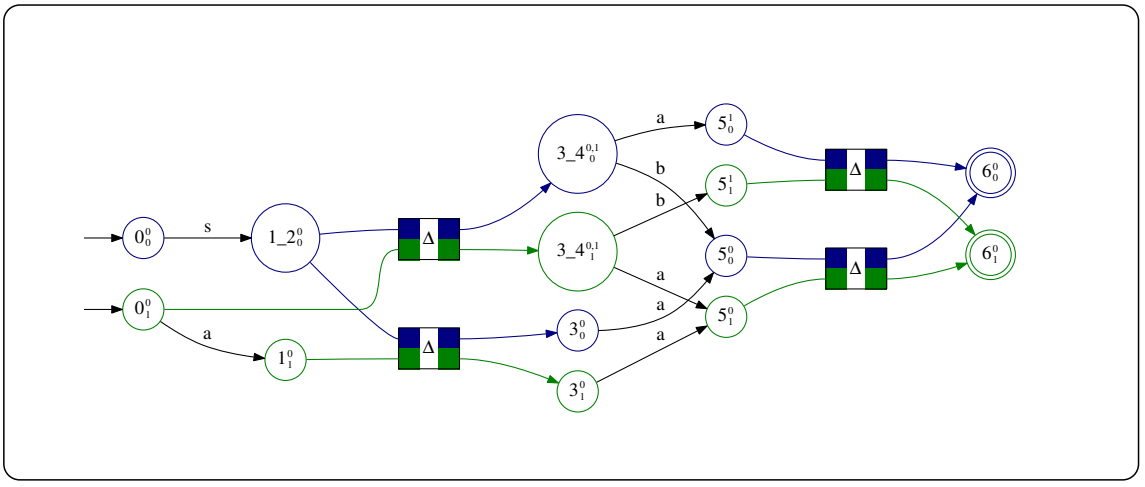


Figure 5.2 – *Determinization of NBSPA in Figure 5.1*

Finite automata theory specify that automata have a *set* of states, not a multi-set (bag) which implies that states cannot have doubloons. In order to duplicate states, they are thus given a different index for each duplications. Duplicated (or indexed) automata have the type $(Q \times \mathbb{N})$ *nfa*. Note that index is reset for different each Δ input thereby limiting duplication to non-deterministic Δ-transitions.

Our algorithm for duplication is named *index_reachable* because it extract and index a sub-local part of a BSPA. The part extracted is reachable by δ-transition from the set of initial states given until states within an input vector of Δ and the

latter are returned as final. To extract which state vector are inputs of Δ, we rely the domain of Δ.

Definition 28 $\mathcal{D}(f) : (\alpha \to \beta) \to 2^{\alpha}$

$\mathcal{D}(f)$ is the domain of the function f. It is the set of all possible inputs of f.

The algorithm *index_reachable* was written with a structure very similar to *determinize*. The first argument $t : \mathbb{N}$ is the index given to all states of the extracted part. Arguments $i : \mathbb{N}$ and $\Delta : (Q^p \to 2^{Q^p})$ are only used line 17 to build final states. Other arguments are components of a classical NFA of which final states are not used.

We note indexed states (q), functions (δ) and set of states (Q) with the notation q_x, δ_x and Q_x respectively.

Definition 29 *index_reachable* : $\mathbb{N} \to \mathbb{N} \to (Q^p \to 2^{Q^p}) \to Q\,nfa \to (Q \times \mathbb{N})\,nfa$

```
input : t, i, Δ, (Q, Σ, δ, I, _)
output: (Q_x, Σ, δ_x, I_x, F_x)
begin
    I_x ← ⋃_{q∈I} (q, t)
5   W ← I_x
    while W ≠ ∅ do
        (q, t) ← select_from(W)
        W ← W \{(q, t)}
        for each p ∈ δ(q, _) do
10          δ_x(q, t) ← (p, t)
            if (p, t) ∉ (W ∪ Q_x) then
                W ← W ∪ {(p, t)}
            endif
            Q_x ← Q_x ∪ {(q, t)}
15      done
    done
    F_x ← { (q, t) ∈ Q_x | ⟨..., q^i, ...⟩ ∈ D(Δ) }
end
```

5.3 BSPA determinization algorithm

Determinization of BSPA proceed superstep per superstep. The main routine starts from a delta input $\vec{q}$ and locally duplicate the reachable part of each of its

output before determinizing the union of these parts. The routine restarts from each Δ-input reached. $\mathcal{W}$ is the set of Δ input reached.

Definition 30 *determinize_bspa* : Q *nbspa* $\rightarrow 2^{Q\times\mathbb{N}}$ *dbspa*

input : $(\langle Q^i\rangle_{i\in[p]}, \Sigma, \langle\delta^i\rangle_{i\in[p]}, \langle I^i\rangle_{i\in[p]}, \langle F^i\rangle_{i\in[p]}, \Delta)$
output: $(\langle Q^i_d\rangle_{i\in[p]}, \Sigma, \langle\delta^i_d\rangle_{i\in[p]}, \langle s^i_d\rangle_{i\in[p]}, \langle F^i_d\rangle_{i\in[p]}, \Delta_d)$
begin
  par $i = 0$ **to** $p-1$ **do**
    $A_x \leftarrow$ *index_reachable*$(0, i, \Delta, Q^i, \Sigma, \delta^i, I^i, F^i)$
    $(Q^i_d, \Sigma, \delta^i_d, s^i_d, \mathcal{F}^i_d) \leftarrow$ *determinize*(A_x)
  done
  $\mathcal{W} \leftarrow \left\{\vec{q}_d \in \prod_{i\in[p]}\mathcal{F}^i_d \mid \forall i \in [p], \exists(q,0)^i \in q^i_d, \langle q^0,\ldots,q^{p-1}\rangle \in \mathcal{D}(\Delta)\right\}$
  while $\mathcal{W} \neq \emptyset$ **do**
    $\vec{q}_d \leftarrow$ *select_from*$(\mathcal{W})$
    $\mathcal{W} \leftarrow \mathcal{W} \setminus \{\vec{q}_d\}$
    $\mathcal{I} \leftarrow \forall\vec{q} \in \prod_{i\in[p]} q^i_d, \bigcup_{\vec{q}}\Delta(\vec{q})$
    $t \leftarrow 0$
    $A_x \leftarrow \emptyset$
    for each $\vec{p} \in \mathcal{I}$ **do**
      par $i = 0$ **to** $p-1$ **do**
        $A_x \leftarrow A_x \cup$ *index_reachable*$(t, i, \Delta, Q^i, \Sigma, \delta^i, p^i, F^i)$
        $t \leftarrow t+1$
      done
    done
    par $i = 0$ **to** $p-1$ **do**
      $(P^i_d, \Sigma, d^i_d, r^i_d, \mathcal{F}^i_d) \leftarrow$ *determinize*(A_x)
      $(Q^i_d, \Sigma, \delta^i_d, s^i_d, \mathcal{F}^i_d) \leftarrow (Q^i_d \cup P^i_d, \Sigma, \delta^i_d \cup d^i_d, s^i_d, \mathcal{F}^i_d)$
    done
    $\Delta_d(\vec{q}_d) \leftarrow \vec{r}_d$
    $$\mathcal{W} \leftarrow \left\{\vec{q}_d \in \prod_{i\in[p]}\mathcal{F}^i_d \;\middle|\; \begin{array}{l}\exists t, \forall i \in [p], \exists(q,t)^i \in q^i_d, \\ \quad \langle q^0,\ldots,q^{p-1}\rangle \in \mathcal{D}(\Delta) \\ \quad \wedge\, q^i_d \notin \mathcal{D}(\Delta_d)\end{array}\right\} \cup \mathcal{W}$$
  done
  par $i = 0$ **to** $p-1$ **do**
    $F^i_d \leftarrow \left\{q^i_d \in Q^i_d \mid \exists q^i \in q^i_d, q^i \in F^i\right\}$
  done
end

6 BSP Regular Expression for Parallel Matching of Regular Expression

Contents

6.1 Parallel regular expression matching

Applications of regular expression matching ranges from mainstream applications such as *grep* [17] to specialized applications including for example deep packet inspection [50, 35, 61, 16]. Most existing techniques used for parallel

matching of regular expression simulate an execution from a subset of the DFA states [24, 48, 16]. Other techniques makes the automata suitable for parallel execution [56], or rely on dedicated software such as FPGAs [37, 40] or TCAMs [47].

The novel approach introduced in this chapter focuses on regular expressions (RE) by transforming them into a parallel form called BSP regular expressions (BSPRE) and derive parallel BSP automata (BSPA). The transformation from BSPRE to BSPA was proposed in [58]. Such technique was not possible before [19] as no model for parallel regular expression such as BSPRE existed to the best of our knowledge.

The remaining of this section 6.1 details the process of parallel regular expression matching as an extension of sequential matching, precise our input distribution and define preconditions for the next section. The transformation from RE to BSPRE is defined and explained in section 6.2. Our approach begins by transforming the regular expression into an *intermediate form*. In order to match an input split among p processors, the regular expression must also be split and the intermediate form make this computation easier. The splits are then allocated into p-sized vectors, each vector representing a potential split of the input. The BSPRE eventually obtained is the disjunction of those vectors. Experiment results are provided and discussed in section 6.3. Section 6.4 conclude this chapter.

6.1.1 Sequential matching

The sequential matching of regular expression is represented in the following fig. 6.1 and involves two transformations.

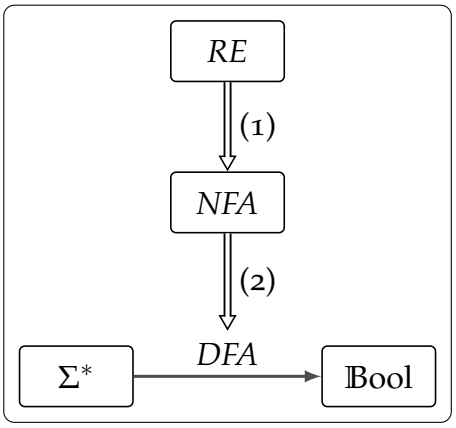


Figure 6.1 – *Sequential matching*

The first transformation (1) is the computation of an acceptance machine

for the language represented by a given regular expression. Algorithms known for this transformation include algorithms of Thompson [59], Glushkov [8] and Brzozowski [7].

Transformation (2), called determinization and explained previously in section 5.1, makes the acceptance machine deterministic in order to remove ambiguities. An ambiguity imply the exploration of several paths in the acceptance machine, thereby decreasing the efficiency. Thus, although this transformation is not mandatory, most regular expression matching approaches favors deterministic finite state automata (DFA) over non-deterministic finite state automata (NFA) for their efficiency. The matching is processed by the computed DFA. The output is a boolean value representing whether a given input word $w \in \Sigma^*$ belong to its language.

6.1.2 Parallel matching

Our parallel matching scheme of a regular expression is represented in fig. 6.2 and requires four transformations.

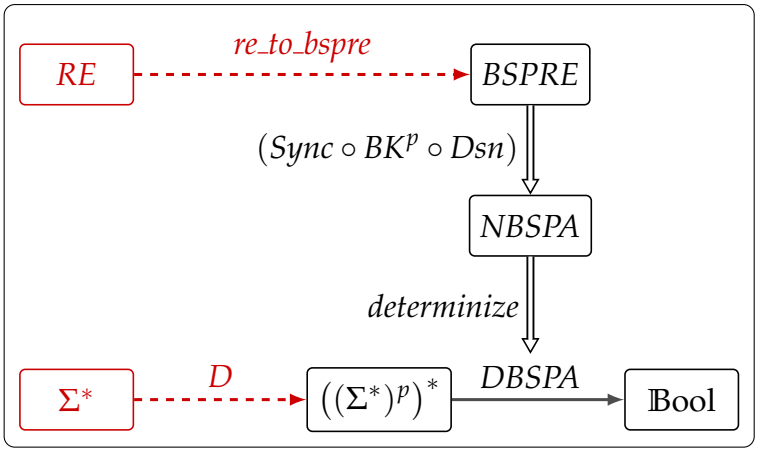


Figure 6.2 – *Parallel matching*

1. $D : \mathbb{N} \to \Sigma^* \to ((\Sigma^*)^p)^*$

   $D(p, w)$ Distributes word w to a p-sized vector of processors.

2. $re\_to\_bspre : \mathbb{N} \to \Sigma\, re \to \Sigma\, bspre$

   Computes $R_{BSP} = re\_to\_bspre(p, R)$ where $L(R_{BSP}) = \left\{ D(p, w) \mid w \in L(R) \right\}$

3. $(Sync \circ BK^p \circ Dsn) : \Sigma\, bspre \to \Sigma\, nbspa$

   Generates the acceptance machine of a BSPRE.

4. *determinize* : $\Sigma\, nbspa \to \Sigma\, dbspa$

   Makes the acceptance machine deterministic.

The first transformation is needed in any parallel program where data is not shared by processors. If the memory is not shared then data needs to be distributed. Section 6.1.3 details two different distributions where the BSP word generated is a sequence of a single BSP vector. This way, only one synchronization is performed. The second transformation will be detailed in section 6.2. The third and fourth transformations were explained in chapter 4.

6.1.3 Input distribution

The main algorithm designed in section 6.2 will automatically compute a BSPRE whose language is $\{D(p, w) \mid w \in L(r)\}$ for p a given natural integer and r a given regular expression. Thus, before designing the algorithm, an input distribution D must be fixed. Note that only distributions computing a single vector, and not a sequence, were considered for performance reasons. In order to give a better illustration of them, the following example will be tackled according to both distributions considered: cyclic and block distribution.

Example 2 *(string search)* **:** The chosen example is the following simple but widely use-case of regular expression matching. We want to know if there exists a string, say "Valiant", somewhere in a file. In POSIX, this expression may be written:

$$r = .^* \text{ Valiant } .^*$$

Where '.' means any characters. Note $\sigma = \forall \sigma_i \in \Sigma, (\sigma_0 + \sigma_1 + \dots)$. The above expression is easily transformed into the following regular expression:

$$r = \sigma^* \text{ Valiant } \sigma^*$$

a) Block distribution

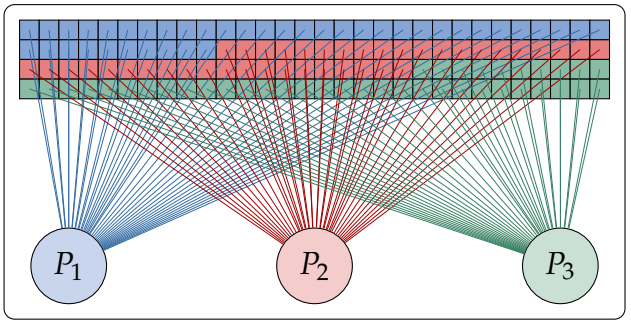


Figure 6.3 – *Block distribution into 3 processors*

Note n the input size. The block distribution D_b distributes each letter u_i (with $0 \leq i < n$) in a word vector at processor i/p.

$$\begin{aligned} D_b(p, w_{input}) &= D_b(p, u_0 \dots u_{n-1}) \\ &= \langle u_0 u_1 u_2 \dots \, , \, u_{n/p} \dots \, , \, \dots \, \dots \, , \, \dots u_{n-3} u_{n-2} u_{n-1} \rangle \end{aligned}$$

Example 2 *(continuing from p. 52)* **:** We choose processor number $p = 3$ (a little value is taken for the sake of clarity). Knowing the distribution used and p, we design the BSPRE R. R will be a disjunction with one case for each acceptable regular expression vector. Let m be the length of the string sought, we assume that $n > p \cdot m$ so that the string cannot be split between more than two processors. This condition will be refined and generalized in section 6.1.4. There is one possible case according to which location "Valiant" might be and one case for each possible split.

$$R = \begin{cases} \langle \quad \sigma^* \quad , \quad \sigma^* \quad , \sigma^* \texttt{Valiant} \, \sigma^* \rangle \\ + \langle \quad \sigma^* \quad , \quad \sigma^* \quad \texttt{V} \quad , \texttt{aliant} \quad \sigma^* \rangle \\ + \langle \quad \sigma^* \quad , \quad \sigma^* \quad \texttt{Va} \, , \texttt{liant} \quad \sigma^* \rangle \\ \qquad\qquad\qquad \vdots \\ + \langle \quad \sigma^* \quad , \sigma^* \texttt{Valiant} \, \sigma^* , \quad \sigma^* \quad \rangle \\ \qquad\qquad\qquad \vdots \\ + \langle \sigma^* \texttt{Valiant} \, \sigma^* , \quad \sigma^* \quad , \quad \sigma^* \quad \rangle \end{cases}$$

In this example, the resulting BSPRE is a disjunction of $m \cdot (p-1) + 1$ cases.

b) Cyclic distribution

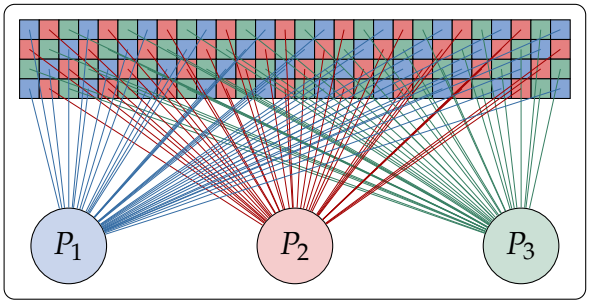


Figure 6.4 – *Cyclic distribution into 3 processors*

The cyclic distribution D_c distribute each letter u_i in a word vector at location i `modulo` p.

$$\begin{aligned} D_c(p, w_{input}) &= D_c(p, u_0 \dots u_{n-1}) \\ &= \langle u_0 u_p u_{2p} \dots \, , \, u_1 u_{p+1} \dots \, , \, u_2 u_{p+2} \dots \, , \, \dots \, \dots \rangle \end{aligned}$$

Example 2 *(continuing from p. 53)* **:** Under the same conditions as in the previous distribution, we design the BSPRE. R will be also a disjunction with one case for each acceptable word. In the cyclic distribution, knowing the location of one letter, means knowing the location of all others. Say u_i is located at j then u_{i+1} is located at $(j+1)$ `modulo` p. One letter has p possible location, so R here yields p cases.

$$R = \left\{ \begin{array}{lll} & \langle \sigma^* \texttt{ Vit } \sigma^* \, , \, \sigma^* \texttt{ aa } \sigma^* \, , \, \sigma^* \texttt{ ln } \sigma^* \rangle \\ + & \langle \sigma^* \texttt{ ln } \sigma^* \, , \, \sigma^* \texttt{ Vit } \sigma^* \, , \, \sigma^* \texttt{ aa } \sigma^* \rangle \\ + & \langle \sigma^* \texttt{ aa } \sigma^* \, , \, \sigma^* \texttt{ ln } \sigma^* \, , \, \sigma^* \texttt{ Vit } \sigma^* \rangle \end{array} \right.$$

However, it appeared that regular expressions parallelized according to this input distribution are not always representable with BSPRE.

Example 3 *(Conterexample)* **:** Take for example the regular expression $(aa)^*$ and find a BSPRE representing the BSP language $L = \left\{ D_c(2, w) \mid w \in L((aa)^*) \right\}$. The best we can find is $\langle a^n, a^n \rangle$ (to not confuse with $\langle a, a \rangle^*$ which represents a different language). However its language is not regular and it is not a BSPRE.

From here onwards, only the block distribution will be considered and the function *re_to_bspre* will be designed accordingly.

6.1.4 Precondition

The algorithm presented in section 6.2 requires a precondition on the input size. It was previously claimed in section a) that ex. 2 requires the input to be longer than $p \cdot m$ with m the length of the string searched. This precondition prevents for example the string "`Valiant`" from being split in two locations by the distribution. This section generalizes this condition to any regular expression.

Definition 31 $re\_size : \Sigma\, re \to \mathbb{N}$

$$
\begin{aligned}
re\_size(F + G) &= max\big(re\_size(F), re\_size(G)\big) && (1)\\
re\_size(F \cdot G) &= re\_size(F) + re\_size(G) && (2)\\
re\_size(F^*) &= 0 && (3)\\
re\_size(a) &= 1 && (4)\\
re\_size(\epsilon) &= 0 && (5)\\
re\_size(\emptyset) &= 0 && (6)
\end{aligned}
$$

with *max* trivially defined as:

Definition 32 $max : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ **:**
$$max(a, b) = \begin{cases} a & \textbf{if } a > b \\ b & \textbf{otherwise} \end{cases}$$

Having defined *re_size*, the precondition is written:

Proposition 32.1 (large input)**.**

$$|w_{input}| > p \times re\_size(R)$$

6.2 From regular expressions to BSP regular expressions

6.2.1 From tree-form regular expression to sequence set

Since the input is distributed in a sequence of blocks, it was easier for the algorithm design to also consider the regular expressions as a sequence rather than a tree. There is obviously more than just products in a regular expression and it cannot be completely represented with sequences. Below is our representation of regular expression :

Definition 33 *intermediate form* **:** $re^{\circ} * re^{\square} * re^{\bullet}$

$$
\begin{array}{ll}
re^{\circ} = re^{\square}\, set & \text{unions} \\
re^{\square} = re^{\bullet}\, list & \text{concatenations} \\
re^{\bullet} = (re^{\circ})^* \mid a \mid \epsilon \mid \emptyset & \text{closure, symbols, epsilon, nil}
\end{array}
$$

As one may remark, unions, concatenations and closures are not intertwined with each other as it is the case for regular expression. Here, the type impose the following hierarchy: closure of sets of sequence (*list*) of perhaps, recursively, closure of ... *etc*. Transforming the regular expression into such a hierarchy is similar to a polynomial expansion in algebra or obtaining a disjunctive normal form in propositional logic. This transformation is described by the function provided below:

Definition 34 $re\_to\_re^{\circ} : \Sigma\, re \to \Sigma\, re^{\circ}$

$$
\begin{aligned}
re\_to\_re^{\circ}(F \cdot G) &= \left\{ F' \mathbin{@} G' \mid F' \in re\_to\_re^{\circ}(F), G' \in re\_to\_re^{\circ}(G) \right\} \\
re\_to\_re^{\circ}(F + G) &= re\_to\_re^{\circ}(F) \cup re\_to\_re^{\circ}(G) \\
re\_to\_re^{\circ}(R^*) &= \{[re\_to\_re^{\circ}(R)^*]\} \\
re\_to\_re^{\circ}(a) &= \{[a]\} \\
re\_to\_re^{\circ}(\epsilon) &= \{[\epsilon]\} \\
re\_to\_re^{\circ}(\emptyset) &= \{[\emptyset]\}
\end{aligned}
$$

Converting back to a regular expression presents no subtlety and is described here with the following three mutually recursive functions.

Definition 35 $re^{\circ}\_to\_re, re^{\square}\_to\_re, re^{\bullet}\_to\_re$ **:**
$$
\begin{cases}
re^{\circ}\_to\_re : \Sigma\, re^{\circ} \to \Sigma\, re, \\
re^{\square}\_to\_re : \Sigma\, re^{\square} \to \Sigma\, re, \\
re^{\bullet}\_to\_re : \Sigma\, re^{\bullet} \to \Sigma\, re
\end{cases}
$$

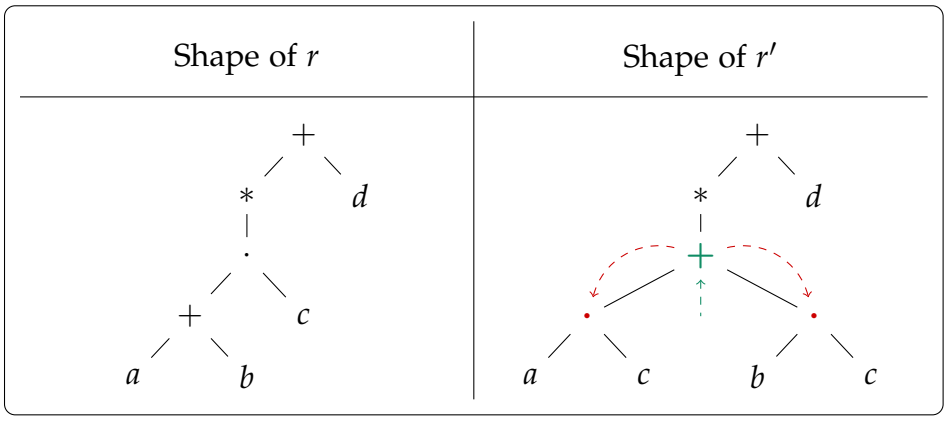


Figure 6.5 – *Shapes of r and r'*

$$
\begin{aligned}
re^{\circ}\_to\_re(s_1 \cup s_2) &= re^{\circ}\_to\_re(s_1) + re^{\circ}\_to\_re(s_2) \\
re^{\circ}\_to\_re(\{l\}) &= re^{\square}\_to\_re(l) \\
re^{\circ}\_to\_re(\{\}) &= \emptyset \\
re^{\square}\_to\_re(l_1 \mathbin{@} l_2) &= re^{\square}\_to\_re(l_1) \cdot re^{\square}\_to\_re(l_2) \\
re^{\square}\_to\_re([r]) &= re^{\bullet}\_to\_re(r) \\
re^{\square}\_to\_re([\,]) &= \epsilon
\end{aligned}
\qquad
\begin{aligned}
re^{\bullet}\_to\_re(s^*) &= re^{\circ}\_to\_re(s)^* \\
re^{\bullet}\_to\_re(a) &= a \\
re^{\bullet}\_to\_re(\epsilon) &= \epsilon \\
re^{\bullet}\_to\_re(\emptyset) &= \emptyset
\end{aligned}
$$

Converting back and forth may not be obvious merely with the converting functions definitions and is illustrated through example 4.

Example 4 : Let $r \in \Sigma\, re$ a regular expression, defined below in equation (1). The converting functions defined above are applied on r.

$$
\begin{aligned}
r &= ((a+b)\cdot c)^* + d && (1)\\
re\_to\_re^{\circ}(r) &= \{[\{[a;c],[b;c]\}^*],[d]\} && (2)\\
re^{\circ}\_to\_re(re\_to\_re^{\circ}(r)) &= (ac+bc)^* + d \quad = r' && (3)
\end{aligned}
$$

Although $r \neq r'$, we have $L(r) = L(r')$

Figure 6.5 illustrate the expansion from product of disjunction into disjunction of product happening in ex. 4. Moreover, although the regular expression shape is not preserved, the language is.

Lemma 1 (Conversion preserves language). $\forall r \in \Sigma\, re$,

$$L(re^{\circ}\_to\_re(re\_to\_re^{\circ}(r))) = L(r)$$

Proof of lem. 1 is done by structural recursion on regular expressions.

6.2.2 Algorithm overview

We present an overview of the algorithm with its main functions.

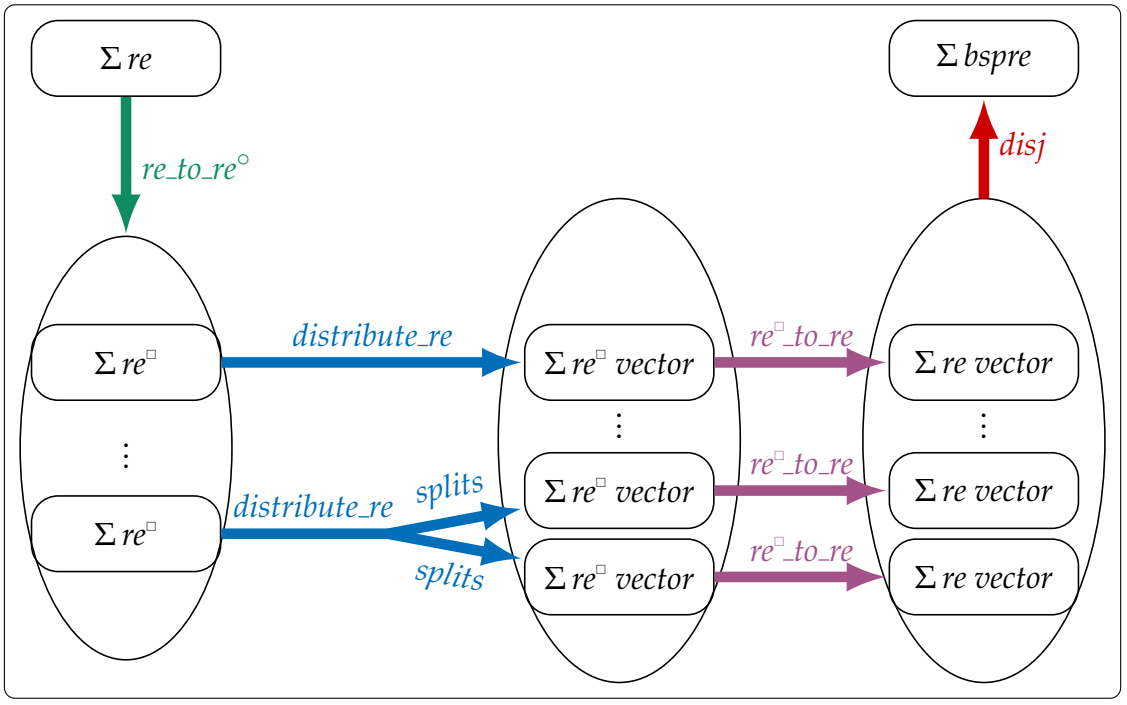


Figure 6.6 – *Overview of re_to_bspre algorithm*

re_to_re° transforms the regular expression into our intermediate form, re°, that is a set of $re^{\square}$.

distribute_re relies on *splits* to reflect input distribution in the regular expression, creating different possible splits. Those splits are then distributed into vectors.

re_to_re° is mapped on the vectors to output regular expression vectors.

disj computes the disjunction of regular expression vectors into a BSPRE.

6.2.3 Splitting the regular expression

Let us proceed with the following motivational example.

Example 5 : In this example, the RE a^*bba^* is distributed into a BSPRE with vectors of size 2. Vectors in red color are not computed by the algorithm.

$$re\_to\_bspre(2, a^*bba^*) = \langle a^*bba^*, a^*\rangle + \langle a^*b, ba^*\rangle + \langle a^*, a^*bba^*\rangle + \langle a^*, bba^*\rangle + \langle a^*bb, a^*\rangle$$

In fact, the BSPRE output in ex. 5 would have the same language with or without the red vectors. And purposely not computing them add complexity to the algorithm. So why bother avoiding those red vectors? This is merely a question of size. After computing the BSPRE, a BSPA will be derived and the time it takes to build and its size will be dependent on the size of the BSPRE [58]. After deriving the BSPA, we may want to determinize it which is exponential with the BSPA size. Therefore the BSPRE size is kept to a minimum as much as possible. So, $\langle a^*, bba^*\rangle$ and $\langle a^*bb, a^*\rangle$ are not computed because $L(\langle a^*, bba^*\rangle) \subset L(\langle a^*, a^*bba^*\rangle)$ and $L(\langle a^*bb, a^*\rangle) \subset L(\langle a^*bba^*, a^*\rangle)$. Those insights will be useful to understand the choices made in the next function, *splits*.

The critical point of the whole algorithm is to compute where the regular expression can be split in order to represent all possible input blocks separation by the input distribution. The function *splits* computes the location of such splits. A split is represented by a couple where the first member is the sequence before the split and the second member is the sequence after the split.

Also, the following property holds.

Proposition 35.1. $\forall l \in \Sigma\, re^{\square}, \forall (ll, rl) \in splits(l), ll \mathbin{@} rl = l$

Definition 36 *splits* **:** $\Sigma\, re^{\square} \to (\Sigma\, re^{\square} * \Sigma\, re^{\square})\ set$

$$splits(r :: l) = spliter([r], l) \quad (1)$$

$$splits([\,]) = \{\} \quad (2)$$

where

$$spliter(ll, R_1^* :: R_2^* :: rl) = \left(\left\{ \begin{pmatrix} ll \ @ ll', \\ rl' \ @ (R_2^* :: rl) \end{pmatrix} \middle| (ll', rl') \in split\_star(R_1) \right\} \right. \\ \left. \cup \quad spliter(ll \ @ [R_1^*], R_2^* :: rl) \right) \tag{3}$$

$$spliter(ll, R_1^* :: R_2 :: rl) = \left(\left\{ \begin{pmatrix} ll \ @ ll', \\ rl' \ @ (R_2 :: rl) \end{pmatrix} \middle| (ll', rl') \in split\_star(R_1) \right\} \right. \\ \left. \cup \quad spliter(ll \ @ [R_1^*] \ @ [R_2]), rl) \right) \tag{4}$$

$$spliter(ll, [R^*]) = \left\{ (ll \ @ ll', rl') \mid (ll', rl') \in split\_star(R) \right\} \tag{5}$$

$$spliter(ll, R :: rl) = \{(ll, R :: rl)\} \cup spliter(ll \ @ [R], rl) \tag{6}$$

$$spliter(ll, [\,]) = \{\} \tag{7}$$

An important algorithm invariant is that *splits* is always called on sequences that will eventually be surrounded by closures. Therefore, none of the split couple member can be $[\,]$, otherwise a superfluous vector like the ones of ex. 5 would be created. Hence, equation (1) of *splits* calls *spliter* with a non empty left member. Equation (6) is the recursive case on unstarred expression which add the current split and recursively calls with the first symbol of right member becoming the last of the left one. Thereby reducing the size of the right member until terminal case (7) occurs. Equations (4) and (3) requires *split_star* which computes the splits of a closure. The only difference between those two cases right hand side is the recursive call. If a starred expression is followed by a symbol (4), then the symbol is also put in the other side, again to avoid splits leading eventually to superfluous vectors (*i.e.* red cases of ex. 5).

Let us see some examples of how closures are handled. One may wonder why closures are not kept as such and merely duplicated across adjacent vector components. Indeed, such cases exist as shown in ex. 6.

Example 6 :

$$re\_to\_bspre(2, (a+b)^*, 2) = \langle (a+b)^*, (a+b)^* \rangle$$

Remark 36.1. Given a starred expression $R^* \in \Sigma\, re$, if $\forall w \in L(R), |w| \leq 1$ then R^* won't be modified by the algorithm.

However such cases are a minority and keeping the starred expression as such is not enough anymore when starred expression are bigger.

Example 7 : $(ab)^*$ is distributed and a could be the last symbol caught by a processor and b the first caught by the next one because $|ab| > 1$.

$$re\_to\_bspre(2, (ab)^*) = \langle (ab)^*, (ab)^* \rangle + \langle (ab)^* a, b(ab)^* \rangle$$

The function *split_star* is mutually recursive with *splits*. While *splits* purpose is to computes the splits of a sequence, *split_star* manages the starred expression within the sequence. Conversely, *split_star* needs *splits* to compute splits within a starred expression.

Definition 37 $split\_star : \Sigma\, re^\circ \to (\Sigma\, re^\square * \Sigma\, re^\square)\ set$

$$split\_star(R) = ([R^*], [R^*]) \cup \left\{ ([R^*] \ @\ ll, rl \ @\ [R^*]) \mid l \in R, (ll, rl) \in splits(l) \right\}$$

First, this function always return the split with the given starred expression kept as such for both split members (as needed in ex. 6). This split is added to the possible splits within the stared expression (ex. 7). For each sequence within $R \in \Sigma\, re^\circ$, the splits computed are surrounded by the original starred expression.

Incidentally, if all words in the starred expression language have a length lesser than 1, it means that (1) there is no star inside and (2), $\forall l \in R, |l| \leq 1$. if $|l| = 0$ then $splits(l)$ will apply equation (2) of *splits* and no splits is returned. if $|l| = 1$ then equation (1) then (7) apply and no splits is returned again. Therefore, if all words in the starred expression language have a length lesser than 1, then $split\_star(R^*) = ([R^*], [R^*])$. This proves rmk. 36.1.

It was shown that closures must be split when the expression matches words longer than a symbol. But whether a symbol sequence must be split is the responsibility of function *until_next_star*.

Example 8 :

$$re\_to\_bspre(3, abc^*de) = \langle abc^*, c^*, c^*de \rangle$$

In ex. 8, the regular expression $R = abc^*de$ is distributed over 3 components. If there is only one star in the regular expression, the symbol sequence is not split because as long as the input size is greater than $p \times re\_size(R)$, the part before and after the star must be matched by the first and last vector component respectively. If there are two stars (ex. 5) or more, the sequence in between must be split

because depending on the proportion of input size matched by the closures, the sequence symbol members may match in a vector component or another.

Definition 38 *until_next_star* **:** $\Sigma\, re^{\square} \to (\Sigma\, re^{\square} * \Sigma\, re^{\square})$

$$until\_next\_star(l) = helper([\,], l) \quad (1)$$

where

$$helper(ll, [\,]) = ([\,], ll) \quad (2)$$

$$helper(ll, R^* :: rl) = (ll, R^* :: rl) \quad (3)$$

$$helper(ll, R :: rl) = helper(ll@[R], rl) \quad (4)$$

For a given sequence, the function *until_next_star* calls *helper* (equation (1)) with an empty sequence to be filled with elements and the given sequence as second argument on which recursion will occur. Elements will be moved from the second argument to the first (equation (4)) along the recursion. The function *until_next_star* will be called right after a star and the following given sequence is cut before the first star encountered (equation (3)). The first part being between two stars, must be split afterwards for the reasons shown above. Conversely, if no star is encountered then there is no need to split. So, if no stars are found (equation (2)), the sequence returned is not given in the first couple member (that would be split) but the second one.

6.2.4 Splits distribution into vectors

Previously was shown how to split the regular expression sequence. Next is presented how the splits are distributed into vectors.

Definition 39 *distribute_re* **:** $\mathbb{N} \to \Sigma\, re^{\square} \to \Sigma\, re^{\square}$ *vector set*

$$distribute\_re(p, l) = helper(\langle [\,], \ldots, [\,] \rangle, 0, l) \quad (1)$$

where

$$helper(v, p-1, l) = \{v^{p-1} \leftarrow v^{p-1} \mathbin{@} l\} \tag{2}$$

$$helper(v, i, R^* :: l') = \Big(\bigcup_{(ll,rl)\in split\_star(R^*)} helper(v^i \leftarrow v^i \mathbin{@} ll, i+1, rl \mathbin{@} l')\Big) \tag{3}$$

$$\cup\ treat\_star(v^i \leftarrow v^i \mathbin{@} [R^*], i, l') \tag{4}$$

$$helper(v, i, R :: l) = helper(v^i \leftarrow v^i \mathbin{@} [R], i, l') \tag{5}$$

$$helper(v, i, [\,]) = \emptyset \tag{6}$$

and

$$treat\_star(v, i, [\,]) = \emptyset \tag{7}$$

$$treat\_star(v, i, R^* :: l'') = helper(v, i, R^* :: l'') \tag{8}$$

$$treat\_star(v, i, R :: l'') = helper(v^i \leftarrow v^i \mathbin{@} [R], i, l'')\ \cup \tag{9}$$

$$\textbf{let}\ (bf\_star, af\_star) = until\_next\_star(R :: l'')\ \textbf{in} \tag{10}$$

$$\bigcup_{(ll,rl)\in splits(bf\_star)} helper(v^i \leftarrow v^i \mathbin{@} ll, i+1, rl \mathbin{@} af\_star)\Big) \tag{11}$$

Given a vector size p and a sequence l, *distribute_re* only calls the first auxiliary recursive function *helper* (line (1)) with arguments (which are the same for the second recursive function *treat_star*):

1. a p-sized vector v of empty sequence that will be filled and duplicated for each different possible splits.
2. an index i initialized at 0. It will grow until $p-1$ and marks the vector location to be filled.
3. the given unmodified sequence l that will be consumed along the recursion.

During recursion, l size decreases until $[\,]$ and i increases until $p-1$.
Let us detail some key points in the function.

Line (2) is the first terminal case and relates to the vector index i. When *helper* is called with the last index, the vector's last position is filled with the rest of the sequence.

If the index is not the last then the behaviour depends on whether the first element is a starred expression. If so, the input that would be matched by the starred expression may be cut by the input distribution (this case is treated in line (3)). Else, if the input matched by the starred expression is not cut by the input distribution, the starred expression is added in the vector (line (4)) before calling *treat_star* where a distinction is made according to what follows the starred expression.

If nothing follows (line (7)) then nothing is returned because the last vector index is not reached yet (otherwise case (2) would have captured it and the only call to *treat_star* do not change the index i). If another starred expression is found right after then only a recursive call is needed (line (8)) as the same procedure needs to be redone. If there is no two closures in a row (line (9)), then there is a sequence, possibly between two starred expression, whose corresponding matched input may be cut by the distribution and a splits must be computed (as shown in ex. 5). If the corresponding input stays within the same position of the vector input, then the following input may as well. This case is handled line (9) with a recursive call on the same index. Otherwise, the splits are computed until the next starred expression (lines (10) - (11)). If no other closures are found by *until_next_star* then *bf_star* is empty and nothing is returned because a sequence without closure could only be put at the last vector position which is impossible as in line (7).

The function *distribute_re* distributes a sequence into a set of vectors. However, the complete algorithm input is not a sequence nor its output is a set of vectors. Its input should a regular expression and its output, a BSP regular expression. The function *re_to_bspre* will call *distribute_re* and do the necessary conversions.

Definition 40 *re_to_bspre* **:** $\mathbb{N} \to \Sigma\, re \to \Sigma\, bspre$

$$re\_to\_bspre(p,R) = disj\left(\left\{ v \;\middle|\; \begin{array}{ll} r \in re\_to\_re^{\circ}(R), & (1)\\ v_{\square} \in distribute\_re(p,r), & (2)\\ \forall k \in \{0 \dots p-1\}, v^k \leftarrow re^{\square}\_to\_re(v_{\square}^k) & (3) \end{array} \right\}\right)$$

where

$$disj(s_1 \cup s_2) = disj(s_1) + disj(s_2) \quad (4)$$

$$disj(\{v\}) = v \quad (5)$$

$$disj(\emptyset) = \emptyset \quad (6)$$

Function *re_to_bspre* starts by transforming the given regular expression into our representation (line (1)), the sequence set. For each sequences in the set, *distribute_re* is called (line (2)). Then function $re^{\circ}\_to\_re$ is mapped on each vectors returned by *distribute_re* (line (3)). After this, a set of regular expressions vectors is computed. The disjunction of those vectors is the outputted BSP regular expression. The disjunction is computed by function *disj*. The BSPRE returned by this algorithm being a disjunction, without any concatenation or Kleene closure (at the global BSP level), makes the proof in Appendix A, which does not prove cases concatenation and closure, sufficient for this application.

Example 9 : As a summary of this section, we detail in this example the processing of ex. 7 by function *re_to_bspre*. Changes between equations are written in a different color.

$$re\_to\_bspre(2,(ab)^*)$$

$$= disj\left(\left\{ v \;\middle|\; \begin{array}{l} r \in re\_to\_re^{\circ}((ab)^*), \\ v_{\square} \in distribute\_re(2,r), \\ \forall k \in \{0,1\}, v^k \leftarrow re^{\square}\_to\_re(v_{\square}^k) \end{array} \right\}\right) \qquad \left[\, def.\ re\_to\_bspre \,\right]$$

$$= disj\left(\left\{ v \;\middle|\; \begin{array}{l} r \in \{[\{[a;b]\}^*]\}, \\ v_{\square} \in distribute\_re(2,r), \\ \forall k \in \{0,1\}, v^k \leftarrow re^{\square}\_to\_re(v_{\square}^k) \end{array} \right\}\right) \qquad \left[\, def.\ re\_to\_re^{\circ} \,\right]$$

$$= disj\left(\left\{ v \;\middle|\; \begin{array}{l} v_{\square} \in distribute\_re(2,[\{[a;b]\}^*]), \\ \forall k \in \{0,1\}, v^k \leftarrow re^{\square}\_to\_re(v_{\square}^k) \end{array} \right\}\right)$$

$$= disj\left(\left\{ v \;\middle|\; \begin{array}{l} v_{\square} \in \{\langle[\{[a;b]\}^*],[\{[a;b]\}^*]\rangle, \langle[\{[a;b]\}^*,\{a\}],[\{b\},\{[a;b]\}^*]\rangle\}, \\ \forall k \in \{0,1\}, v^k \leftarrow re^{\square}\_to\_re(v_{\square}^k) \end{array} \right\}\right)$$

$$\left[\begin{array}{l} def.\ distribute\_re \\ \textbf{with}\ split\_star(\{[a;b]\}^*) = \left\{\begin{array}{c} ([\{[a;b]\}^*],[\{[a;b]\}^*]), \\ ([\{[a;b]\}^*,\{a\}],[\{b\},\{[a;b]\}^*]) \end{array}\right\} \end{array}\right]$$

$$= disj\left(\left\{ v \;\middle|\; \begin{array}{c} v \in \{\langle(ab)^*,(ab)^*\rangle, \\ \langle(ab)^*a, b(ab)^*\rangle\} \end{array} \right\}\right)$$

$$= \langle(ab)^*,(ab)^*\rangle + \langle(ab)^*a, b(ab)^*\rangle \qquad \left[\, disj \text{ in } def.\ 40 \,\right]$$

6.3 Experimental evaluation

6.3.1 Context

Experiments were made to compare our approach (Figure 6.7) to the standard method of parallel regular expression matching (Figure 6.8), referred here as *enumeration* [24] introduced by Holub and Stekr and previously discussed in subsection 3.2.2.a). This work being a proof of concept rather than an efficient

tool aiming to be distributed, the matching was implemented with a high level language (OCaml) and no particular optimization was brought to the automaton. In consequence, absolute times are quite high. However, our approach and the enumeration method were coded and evaluated under the same conditions to present a relevant comparison.

System configuration All experiments were run with the operating system CentOS Linux version 7 with Intel Xeon Processor E5-2690 of 40 cores and 251.87 GiB memory. Parallel code was written with BSML 0.5 [43], a BSP library for OCaml.

Input word generation is summarized herein. Our automata are not complete and only transitions potentially leading to a final state are represented. Consequently, as soon as no transitions are found for an encountered input symbol, the word is rejected and the result is false (*i.e.* the input word is not in the language of the given regular expression). Therefore, being only interested in the worst case matching time, we needed input words (files in practice) belonging to the language of the given regular expression.

The computation relies on Brzozowski's derivative [7] which, for a given symbol, transform a given regular expression so that its language is reduced to words starting by the given symbol and truncated from this symbol.

More formally, with D being the derivative, a a symbol, E a regular expression and L the function computing the language of a regular expression,

$$L\big(D(a,E)\big) = \Big\{w \mid aw \in L(E)\Big\}$$

Additionally, the function *First* (involved in Glushkov's automata construction [8]) is used to compute the first symbols of the words represented by a given regular expression.

$$First(E) = \Big\{a \mid aw \in L(E)\Big\}$$

If a word is a list of symbol and *random_pick* is a function randomly selecting an element in a set then the function (*make_input*) producing the input word is

$$\begin{aligned} make\_input(E) = \ & \textbf{let } a = random\_pick(First(E)) \\ & \textbf{in } \ a :: make\_input(D(a,E)) \end{aligned}$$

In practice, *random_pick* is not really random as a certain word length is required. Brzozowki's derivative also outputs a larger regular expression than the

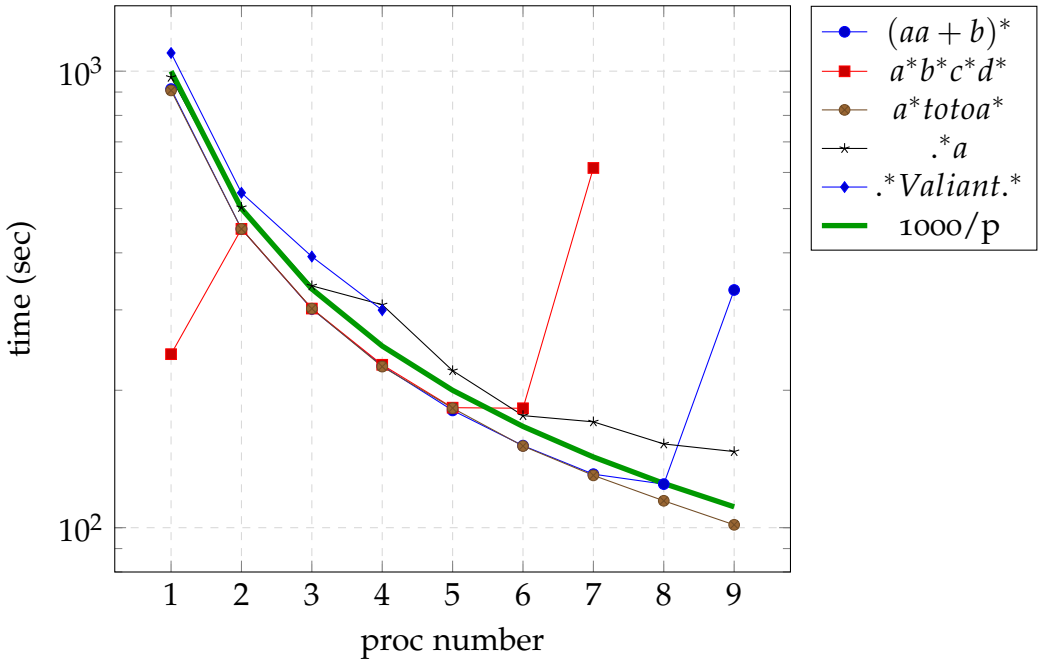


Figure 6.7 – *Building+Matching time of 10Go file with* ***BSPA***

one inputted. Thus, it is necessary to reduce the regular expression(with properties such as $\emptyset + a = a$ or $\epsilon \cdot a = a$) along recursion.

6.3.2 Results

The matching time of 10Go files is observed in Figure 6.7 and 6.8 with a few different regular expressions and up to 9 processors for two method: our method relying on BSPA and Enumeration, the parallel run of a DFA. Times include both automata construction and input matching. Due to a time out of 1 hour (3600 seconds), some curves for our method (Figure 6.7) are not drawn until 9 processors. It comes from the fact the BSPRE size produced is worst-case exponential with p and determinization is also known to be worst-case exponential with the size of the automata ($2^{|Q|}$). Thus, for increasing number of processors, the time required for automata construction eventually out-scales the time required for input matching (see Figure 6.9). However, before construction time explodes, our method is faster than enumeration because enumeration requires a transition computation from each states for each symbol read. To ease comparison, a reference curve appears in both graphs (—), its equation is $time = 1000/p$

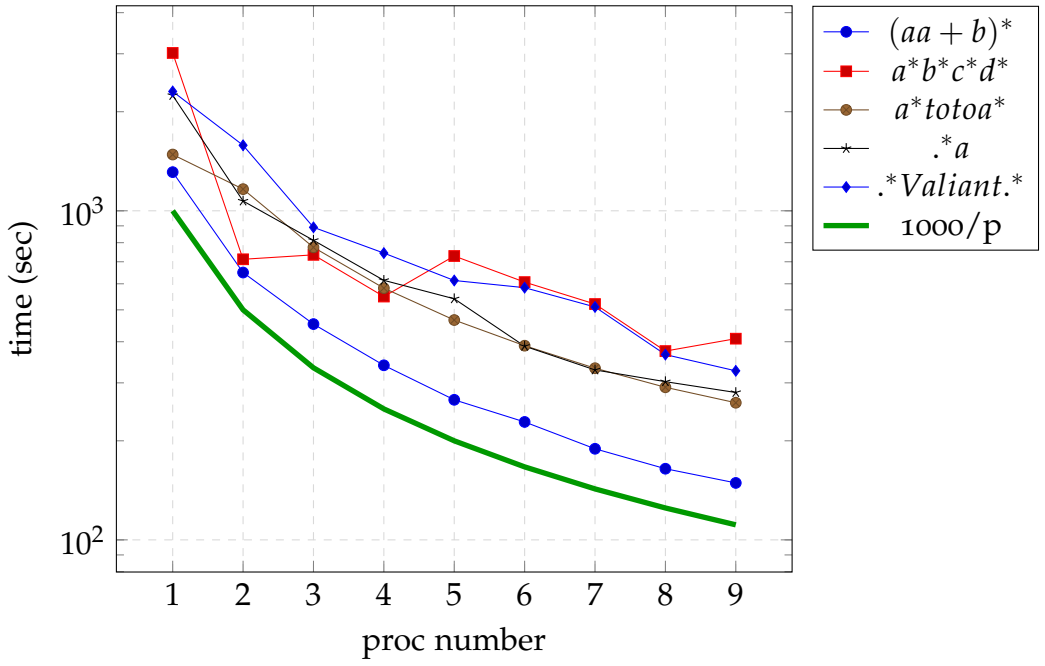


Figure 6.8 – *Building+Matching time of 10Go file with* ***Enumeration***

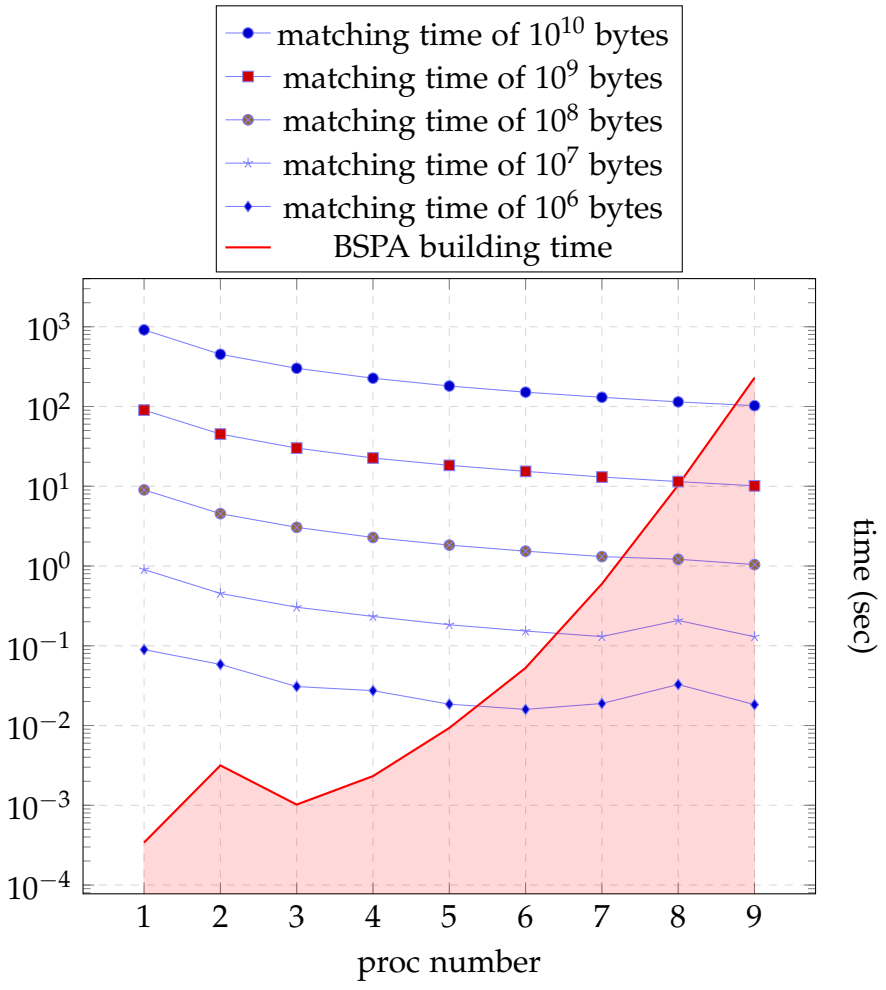


Figure 6.9 – *Matching vs building time of BSPA for RE* $(aa+b)^*$

6.4 Conclusion

The transformation from RE to BSPRE was presented in this chapter. This transformation, added to the transformation from BSPRE to BSPA presented in previous chapters, enables a novel approach for parallel matching of regular expression. The scaling of this approach was also evaluated and compared with a standard in parallel regular expression matching. The transformation from RE to BSPA is also the first infinite non-trivial family of BSP programs automatically generated . Our method was faster for small number of processors because our method only requires the computation of one transition per symbol read. It was nevertheless slower for higher number of processor because time required to construct the automata is exponential with p. Our method would also be particularly suited for applications where the automaton computation time has little importance or when it is computed beforehand.

We compare now to state of the art in parallel regular expression matching introduced in section 3.2.

Approaches such as parallel run of automata [36, 24, 48, 16] and speculative parallel matching [45, 51] we introduced in section 3.2.2.a) involve several simulations while in our approach, there is no ambiguity: only the computation of one transition per input symbol read is required.

Method relying on parallel automata [56] we introduced in section 3.2.2.b) require a reduction to join the results. Instead of joining the local results after reading the input, we computed the possible splits beforehand.

We over-viewed approaches relying on dedicated hardware such as FPGAs [55, 50, 5] in section 3.2.3.a) and TCAMs [47] in section 3.2.3.b) which present great efficiency for regular expression matching. Such hardware is nevertheless rare compared to general purpose CPU and not affordable by everyone.

Tensor Programming with BSP 7

Contents

7.1 Introduction

Many pattern-recognition and machine-learning applications now use neural networks, for example image classification and object recognition in images or videos. The neural nets are dataflow graphs of linear arithmetic and threshold operations on the input pixels, structured as array-shaped *layers* connected by element-regular dependencies that can be fully-connected or sparse as in convolution operations. Layer elements are called *neurons*. An illustration of neural nets is given in Figure 7.1, page 85. The application of a neural net to its intended input is called *inference*.

The layer structure is hand-defined by neural net experts for each specific task and the coefficients that constitute the layer operations (called *parameters*) are learned by an extremely compute-intensive training phase whose algorithm is a

steepest descent optimization with the quality of inference as objective function. Inference quality is defined in an ad-hoc manner by scoring the results on a large dataset.

Programming a neural net amounts to the definition of its layers and their interconnect for inference, then training to obtain high-quality parameters that are used for the target application.

The work we present here is an attempt to further analyze neural net programming as purely-functional non-recursive programming with a small set of tensor primitives. This leads to a better understanding of the (small) fragment of linear algebra used in neural net programming. It also builds an almost-universal basis for building layers and neural nets, the only missing examples for now are so-called recurrent nets i.e. those with cyclic dependencies.

The rest of this chapter describes our language primitives and a small and simple DSL called `HTL` for their application to (acyclic) neural net programming: types, semantics, static analysis, syntax, programming examples and design for parallel code generation and automatic training.

7.2 Related work

The general inference safety problem is an object of growing interest [20] but remains an open problem for lack of a notion of dataset-independent specification. Yet the implementation of neural nets is evolving from an art to normal software engineering with platforms [28, 9] that provide libraries and tools for a high degree of automation in code production and training. A more recent research direction is the design and implement of DSLs for neural-net programming like Diesel [13] and the more advanced Relay [54]. They allow the development of layers as source code rather than black-box libraries, thus improving flexibility and productivity, and can provide *automatic differentiation* a key operation for training. We intend to go one step further and define a "functional MPI of neural net layers": a set of operations that serve as bridge between the great variety of layer structures and the complexity of target architectures.

7.3 Theory

Image or video elements and neural net layers to operate on them are naturally represented as multi-dimensional arrays (called *tensors* in this context, by

analogy with the broader theory of tensor algebra). For example a 2-dimensional image with three color channels leads to a 3-dimensional array of input processed by layers of 1- to 3 dimensions.

Our language design therefore begins with multi-dimensional arrays for which declarative programming languages date back at least to APL [14] and MOA [21]. Systems like MATLAB [22] are completely based around array data structures and are heavily used in all areas of signal processing and scientific computing. Like BSML [41] builds parallel skeletons [12] from a small number of functional primitives, our tensor primitives can generate a large variety of layers.

7.3.1 Data types

Quantization [3] is the mapping of values from a large set to a smaller set. For example using one byte to code some values represented with 8 bytes. Because quantization is an important technique for neural network implementation [11], the scalar data types could be concrete and varied (many numerical types depending on precision like `single`, `half`, `INT8`, etc) or at least parametrized on their precision. But since this choice is independent of the rest of the language, we only retain `int` and `float` in our initial design.

The tensor data type constructor takes a single non-functional type as argument e.g. `float tensor`. In mathematical notation we write $T^\diamond$ for the tensor type whose basic elements are from type T. The type $T^\diamond$ includes every rectangular shape and (non-zero) number of dimensions. Every tensor has a *shape* which is a vector of positive integers. If the shape contains only 1s, e.g. [1,1] then the tensor is equivalent to a scalar. For $\rho = [\rho_0, \ldots, \rho_{D-1}]$ a tensor shape, D is called its number of dimensions[1].

For n a positive integer, the associated ordinal is $[n] = \{0, 1, \ldots, n-1\} = [0 : n-1]$, a non-empty totally-ordered set $\{0 < 1 < \ldots < n-1\}$. The *index set* associated with a shape ρ is $I(\rho) = [\rho_0] \times [\rho_1] \times \ldots [\rho_{D-1}]$.

For example if $\rho = [2, 1, 2]$ then $D = 3$ and $I(\rho) = ([2] \times [1] \times [2]) = \{(0,0,0), (0,0,1), (1,0,0), (1,0,1)\}$.

The content $\kappa(t)$ of a tensor $t \in T^\diamond$ is a map from its index set to elements of type T. So if t is a tensor of shape ρ on type T then its *content* is a map (total function) $\kappa(t) : I(\rho) \to T$.

In addition to scalars and tensors, `HTL` programs must manipulate index

[1] It's best to avoid the word *dimension* (singular) which is ambiguous.

vectors and shapes, that are certain integer vectors. It is theoretically possible to merge vectors into the tensor type, but this would blur the distinction between indexing and data which is not coherent with strong typing and good software engineering practice. So just as scalars are distinct from singleton tensors, we define vectors as distinct from one-dimensional tensors.

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ and $\mathbb{N}^+ = \{1, 2, \ldots\}$.
For a tensor $t \in T^\diamond$, its number of dimensions is a positive integer

$$D(t) \in \mathbb{N}^+$$

its shape is a vector of positive integers of length $D(t)$

$$\rho(t) \in (\mathbb{N}^+)^{D(t)}$$

and its index set is a set of vectors of non-negative integers of the same length

$$I(t) = I(\rho(t)) \subset \mathbb{N}^{D(t)}.$$

To facilitate static analysis and optimization, the language for programming operations on shapes and indices should **not** be a complete arithmetic language. It should be restricted to a fixed number of predefined operations, without general recursion. The `HTL` concrete language design of section 7.6 is coherent with this choice.

7.3.2 Tensor primitives

`HTL` programs are built from primitive operations on tensors. Each one has a precise functional semantics and many possible parallel implementations. The small number of primitives reduces all compilation and implementation systems to just those constructions. We present here their specification. Types are refined in section 7.5 to include shapes. Here, tensors are defined as a pair of shape and content function.

The tensor constructor `init` builds a tensor from a given shape and content function.

Specification 1 (`init`)

Type: $\mathbb{N}^{+D} \to (\mathbb{N}^D \to T) \to T^\diamond$.

Input variables: $\vec{n} : \mathbb{N}^{+D}, \kappa : (\mathbb{N}^D \to T)$

Pre-conditions: $D > 0$

Output: $\texttt{init}(\vec{n})(\kappa) = (\vec{n}, \kappa)$

Notice that the `init` constructor can be used to build "flat" tensors of scalars, for example if T is a numerical type, but also to build "nested" tensors of tensors. Indeed if $\kappa : (\mathbb{N}^D \to T^\diamond)$ then $\texttt{init}(\vec{n})(\kappa) \in T^{\diamond\diamond}$. We use this feature in our `HTL` programming of neural-net layers like convolution.

The `shape` operator returns the shape of a given tensor.

Specification 2 (`shape`)

Type: $T^\diamond \to \mathbb{N}^{+D}$ *where* $D > 0$ *is the number of dimensions.*

Input variables: $t : T^\diamond$

Pre-conditions: *Always defined*

Output: $\texttt{shape}(t) = \rho(t)$

The `content` operator returns a generalization of the content function κ of a given tensor[2]. Given (as second argument, after the tensor) an empty index vector, it returns the input tensor itself. Given an index vector as long as the number of dimensions of the input vector, it returns a singleton tensor. For intermediate lengths of index vector, it returns a sub-tensor with an intermediate number of dimensions. In what follows $\mathbb{N}^*$ denotes vectors of natural numbers (not to be confused with strictly positive naturals).

Specification 3 (`content`)

Type: $T^\diamond \to \mathbb{N}^* \to T^\diamond$.

Input variables: $t : T^\diamond, \vec{r} = [r_0, \ldots, r_{K-1}] : \mathbb{N}^*$

Pre-conditions: $\rho(t) = [\rho_0, \ldots, \rho_{D-1}],\ K \leq D,\ \forall d < K.\, r_d < \rho_d$

Output: $\texttt{content}(t)(\vec{r}) = \texttt{init}[\rho_K, \ldots, \rho_{D-1}](\lambda \vec{i}.\, \kappa(t)(\vec{r} \cdot \vec{i}))$

Here $(\vec{r} \cdot \vec{i})$ denotes index-vector concatenation.

Example 10 : Function `content` is not as usual as the others so we provide a small example here.

[2]It is intended to be implemented lazily so as to avoid constant copying and serialization of the tensor content values.

let a tensor $t = ([2,3], \kappa(i,j) \to i+j) = \begin{bmatrix} 0\ 1\ 2 \\ 1\ 2\ 3 \end{bmatrix}$

$$\texttt{content}(t)([1]) = ([3], \kappa(j) \to 1+j) = \begin{bmatrix} 1\ 2\ 3 \end{bmatrix}$$

The tensor destructor `to_scalar` extracts the scalar content of a singleton tensor.

Specification 4 (`to_scalar`)

Type: $T^\diamond \to T$.

Input variables: $t : T^\diamond$

Pre-conditions: $\rho(t) = [1, 1, \ldots, 1]$

Output: $\texttt{to\_scalar}(t) = \kappa(t)[0, 0, \ldots, 0]$

The `map` operator applies a scalar function to every element of a given tensor.

Specification 5 (`map`)

Type: $(T_1 \to T_2) \to T_1^\diamond \to T_2^\diamond$

Input variables: $f : (T_1 \to T_2), t : T_1^\diamond$

Pre-conditions: *Always defined*

Output: $\texttt{map}(f)(t) = (\rho(t), (\lambda \vec{i}.\ f(\kappa(t)(\vec{i}))))$

The `reduce` operator takes a binary operator on scalars, assumed to be associative and commutative (like addition, multiplication, maximum, minimum) and applies it to reduce a given tensor to a scalar. To ensure those properties, binary operators used in reduce are provided by the language, no defined by the user.

Specification 6 (`reduce`)

Type: $(T \to T \to T) \to T^\diamond \to T$.

Input variables: $\oplus : (T \to T \to T), t : T^\diamond$

Pre-conditions: $\begin{cases} t \text{ is not empty i.e. } \texttt{shape}(t) \text{ does not contain } 0. \\ \oplus \text{ is associative and commutative.} \end{cases}$

Output: $\texttt{reduce}(\oplus)(t) = \sum^{\oplus}_{\vec{i} \in I(t)} \kappa(t)(\vec{i})$

7.4 Abstract data types and expressiveness

To experiment with the core language elements we have first built OCaml abstract types for the primitives. The signature for vectors is as follows and is coherent with the denotational semantics given in subsection 7.3.2.

```
module Vec :
sig
  type α t
  val init : Natplus.t -> (Nat.t -> α) -> α t
  val length : α t -> Natplus.t
  val get : α t -> Nat.t -> α
  val map : (α -> β) -> α t -> β t
  val reduce : (α -> α -> α) -> α t -> α
end
```

Constructor `Vec.init` builds a vector from its size and content function. Destructors `Vec.length`, `Vec.get` and `Vec.reduce` return length, element at a given position and reduction with a binary-associative operation respectively. Transformer `Vec.map` applies a unary function to every element.

The signature for tensors in this proof-of-concept model is as follows.

```
module Tensor:
sig
  type α t
  val init : int vector -> (int vector -> α) -> α t
  val shape : α t -> int vector
  val content : α t -> int vector -> α t
  val to_scalar : α t -> α
  val map : (α -> β) -> α t -> β t
  val reduce : (α -> α -> α) -> α t -> α
end
```

Constructor `Tensor.init` builds a vector from its shape and content function. Destructors `Tensor.shape`, `Tensor.to_scalar` and `Tensor.reduce` return shape vector, scalar value of a singleton vector and reduction with a binary associative-commutative operation respectively.
Transformers `Tensor.content` and `Tensor.map` return the content function ($\kappa()$ of 7.3.2) and apply a unary function to every element respectively.

To demonstrate the core of HTL programming and the expressive power of its primitives, we then wrote very short OCaml programs using only integer/float operations, trivial integer-vector operations for shapes and indices, non-recursive function definitions and the vector/tensor primitives. In this manner the following mini-library has been built.

```
val tensor_scalar_content :
  α Tensor.t -> int vector -> α
val tensor_map2 : (α -> β -> γ) ->
  α Tensor.t -> β Tensor.t -> γ Tensor.t
val float_tensor_addition :
  float Tensor.t -> float Tensor.t -> float Tensor.t
val float_tensor_dot_product :
  float Tensor.t -> float Tensor.t -> float
val constant_tensor : int vector -> α -> α Tensor.t
val zero_float_tensor : int vector -> float Tensor.t
val one_float_tensor : int vector -> float Tensor.t
val indexing1d:(int -> α) -> int vector -> α
val indexing2d:(int*int -> α) -> int vector -> α
val indexing3d:(int*int*int -> α) -> int vector -> α
val tensor_mapi :
  (int vector -> α -> β) -> α Tensor.t -> β Tensor.t
val nil : float Tensor.t
val nats_less_than : int -> int Tensor.t
val nats_up_to : int -> int Tensor.t
```

We have also written short and simple programs that implement the four types of CIFAR10 [31, 32] neural-net layers, namely: convolution, relu-activation, pooling and fully-connected output layer. The extreme simplicity of those programs is a great advantage of HTL . For example 2-dimensional convolution is written as follows:

```
let sub_image_at2d =
 fun image (i_image, j_image) filter_shape2d ->
  Tensor.map Tensor.to_scalar
  (Tensor.init filter_shape2d
    (fun index -> match index with
      | [i_filter; j_filter] ->
        Tensor.content image
```

```
            [i_image + i_filter; j_image + j_filter]
  ) ;;

let convolution2d =
 fun image filter_result_shape2d filter2d ->
  Tensor.init
  filter_result_shape2d
  (indexing2d
    (fun (i,j) -> (* RESULT(i,j) *)
      float_tensor_dot_product
      filter2d (sub_image_at2d image (i,j))
    )
  ) ;;
```

In this operation, the input (image) 2D-tensor is turned into a tensor of small tensors that represent a pixel's neighborhood. One then maps a tensor-reduction operation on the outer tensor so as to sum the neighborhood values. In BSML, such a nested parallel operation is prohibited to avoid unwanted implicit communications. But HTL having static types and shapes, it can be analyzed statically to compile such an operation into a more efficient implementation e.g. reduced communications or even reduced arithmetic operations as in Winograd-convolution. We will return to those neural net layer examples with the actual concrete HTL language. The above constructions are not intended as final source language, precisely to avoid the excessive expressive power of a full host language like OCaml.

7.5 Type-shape system

Our HTL language has static strong (monomorphic) typing and we can take decorate its types with vector lengths and tensor shapes to produce an inference system for static analysis.

Type $\texttt{vector}(n)$ is an n-sized vector of integer. Type α $\texttt{tensor}(\rho)$ is a ρ-shaped tensor of α. A shape ρ has type $\texttt{vector}(n)$.

$$\frac{\rho : \texttt{vector}(n) \qquad f : \texttt{vector}(n) \to \alpha}{init(\rho)(f) : \alpha\ \texttt{tensor}(\rho)}$$

$$\frac{t : \alpha\ \texttt{tensor}(\rho) \quad \rho : \texttt{vector}(n)}{\mathit{shape}(t) = \rho : \texttt{vector}(n)}$$

Remember that the content function has been generalized to produce a sub-tensor of fewer dimensions. For example it can extract lines form a two-dimensional tensor, singletons, lines or planes from a three-dimensional one etc.

$$\frac{\begin{array}{c} t : \alpha\ \texttt{tensor}(\rho) \quad \rho : \texttt{vector}(n) \quad \iota : \texttt{vector}(d) \\ \rho' : \texttt{vector}(n-d) \quad \forall i < n-d.\ \rho'[i] = \rho[i+d] \end{array}}{\mathit{content}(t)(\iota) : \alpha\ \texttt{tensor}(\rho')}$$

$$\frac{f : \alpha \to \alpha' \quad t : \alpha\ \texttt{tensor}(\rho)}{\mathit{map}(f)(t) : \alpha'\ \texttt{tensor}(\rho)}$$

$$\frac{(+) : \alpha \to \alpha \to \alpha \quad t : \alpha\ \texttt{tensor}(n)}{\mathit{reduce}((+))(t) : \alpha}$$

$$\frac{t : \alpha\ \texttt{tensor}(\rho) \quad \rho : \texttt{vector}(n) \quad \forall i < n.\ \rho[i] = 1}{\mathit{to\_scalar}(t) : \alpha}$$

With sufficient restrictions on the source language, this inference system can be applied for static analysis so that every tensor shape is known before execution time. Consequently, language of vectors operations is intended to be weak.

This is an enormous advantage for code generation on complex systems like GPUs or tensor accelerators.

7.6 HTL

This section is an informal but complete presentation of the `HTL` concrete syntax and semantics. Its types and primitives are exactly those presented in the above theory, with a minor change for singleton-to-scalar conversion that appears as a special case of *casting* i.e. type conversion.

We show source code and some output evaluations, assuming the verified/inferred types will be obvious to the reader.

Functions are non-recursive and there are no explicit loops or iterators, all variables being non-mutable (single-assignment). Vectors and tensors are built by a comprehension syntax, whose semantics is the corresponding `init` constructor. For example

```
function zeroVector(int n) return
  vector i < n -> 0;
def zeroVector12 = zeroVector(12);
```

evaluates to

```
zeroVector -> (function)
zeroVector12 -> vector[0,0,0,0,0,0,0,0,0,0,0,0]
```

Note that type constructor keywords "**vector**" and "**tensor**" bind index variables (in the same way that "**fun** `i` `->` `...`" binds `i` in OCaml).

Here is a function to build a two-dimensional tensor.

```
function identityMatrix(int n) return
  tensor (i,j) < (n,n) -> if i=j then 1.0 else 0.0;
```

Conditionals are expressions, not statements.

```
def mySign =
  if x < 0.0 then (-1) else if x=0.0 then 0 else 1;
```

Here is an example of a singleton vector and its transformation into a scalar.

```
def vSingleton5 = vector i < 1 -> 5 ;

/*  Casting vectors to scalars,
    corresponds to Vector.to_scalar */
def fiveIntScalar = (int)vSingleton5;
def fiveFloatScalar = (float)(int)vSingleton5;
```

with evaluation to

```
vSingleton5 -> vector[5]
fiveIntScalar -> 5
fiveFloatScalar -> 5.0
```

and here an example of max-reduction, then a sum-of-squares function:

```
function vectorMax(int[] v) return
  reduce v with max;

def vInts = vector i<5 -> i;
def vecMaxvInts = vectorMax(vInts);
```

```
function intToFloat (int i) return (float) i;

def vFloats = map intToFloat on vInts;

function squareFloat(float x) return x * x;

function sumOfSquares(float[] v) return
  reduce (map squareFloat on v) with + ;

def sumOfSquaresvInts = sumOfSquares(vFloats) ;
```

with evaluation:

```
vectorMax    -> (function)
vInts        -> vector[0,1,2,3,4]
vecMaxvInts -> 4
intToFloat   -> (function)
vFloats      -> vector[0.0,1.0,2.0,3.0,4.0]
squareFloat -> (function)
sumOfSquares -> (function)
sumOfSquaresvInts -> 30.
```

The following is a tensor example inspired by the CIFAR convolution neural net

```
def cifarTensor =
  tensor (i,j,k) < (32,32,3) -> (float)(i+j+k) ;

function tensorSum(float[[]] t) return
  (reduce t with +) ;

def sumOfFloats = tensorSum(cifarTensor);
```

its evaluation:

```
cifarTensor -> tensor[[[0.0,1.0,2.0],
  [1.0,2.0,3.0],
  [2.0,3.0,4.0],
  ...
  [61.,62.,63.],
```

```
  [62.,63.,64.]]]
tensorSum -> (function)
sumOfFloats -> 98304.
```

then an example of a tensor of tensors:

```
def t123 = tensor i<3 -> (float) i;
def tOft = tensor i<32 -> t123;
def flattenedt = map tensorSum on tOft ;

function flattenTensor(float[[]][[]] t) return
  map tensorSum on t;

def flattenedt2 = flattenTensor(tOft);

def lengthFlattenedT2 = shape flattenedt2 ;
```

with evaluation:

```
t123 -> tensor[0.0,1.0,2.0]
tOft -> tensor[
  tensor[0.0,1.0,2.0], ... ,tensor[0.0,1.0,2.0]]
flattenedt    -> tensor[3.0,3.0,3.0,3.0,3.0, ... ,3.0]
flattenTensor -> (function)
flattenedt2   -> tensor[3.0,3.0,3.0,3.0,3.0, ... ,3.0]
lengthFlattenedT2 -> vector[32]
```

and finally a transpose function (which is therefore not a primitive) that illustrates functions with local program blocs, in this case a one-statement block used to avoid multiple computation of a value:

```
/* The inside block prevents computing
(shape t) twice. The cast operation in
(float)t[j][i] is a to_scalar operation. */

function transpose2D(float [[]] t)
  { def shapeT = shape t;}
  return tensor (i,j) < (shapeT[1], shapeT[0]) ->
      (float)(t[j][i]);

def floatAsymmetric =
```

```
  tensor (i,j) < (8,7) -> (float)i;

def transposeTest = transpose2D(floatAsymmetric);
```

Its evaluation is the following:

```
transpose2D -> (function)

floatAsymmetric -> tensor[
  [0.0,0.0,0.0,0.0,0.0,0.0,0.0],
  [1.0,1.0,1.0,1.0,1.0,1.0,1.0],
  [2.0,2.0,2.0,2.0,2.0,2.0,2.0],
  [3.0,3.0,3.0,3.0,3.0,3.0,3.0],
  [4.0,4.0,4.0,4.0,4.0,4.0,4.0],
  [5.0,5.0,5.0,5.0,5.0,5.0,5.0],
  [6.0,6.0,6.0,6.0,6.0,6.0,6.0],
  [7.0,7.0,7.0,7.0,7.0,7.0,7.0]]

transposeTest -> tensor[
  [0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0],
  [0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0],
  [0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0],
  [0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0],
  [0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0],
  [0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0],
  [0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0]]
```

7.7 Programming neural nets

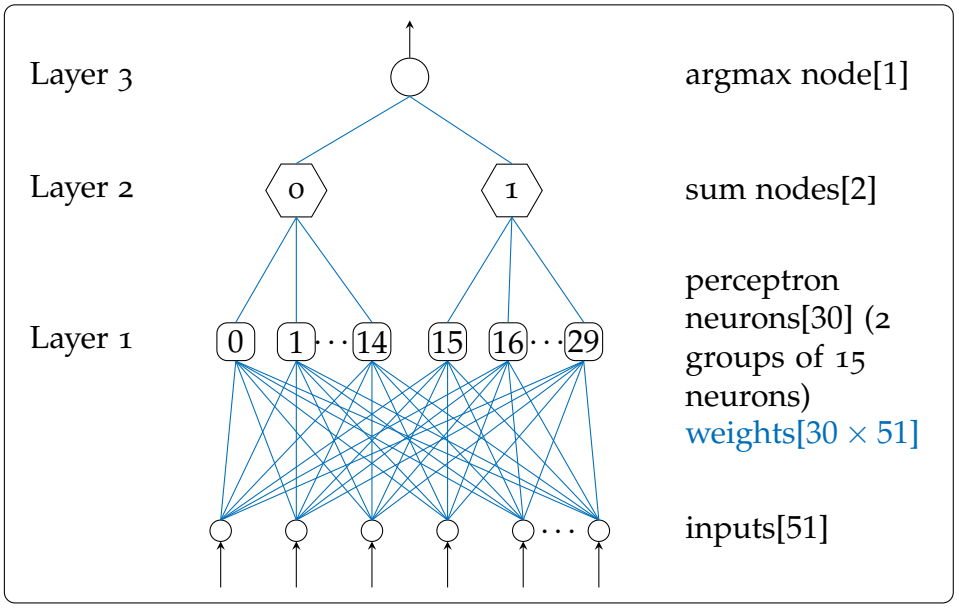


Figure 7.1 – *3-Layers Binary Neural Network Architecture*

This section shows a concrete use-case of HTL issued in [26]. The problem is a two-class binary neural network for handwritten digits. The neural network goal is to distinguish hand-written 0 from other digits. Input images of handwritten digits are given in a compressed form of 51 bits. This problem is tackled with a binary neural network architecture of three layers (in addition to the input) represented in Figure 7.1.

Layer 1: 30 *perceptron nodes*, each connected to the 51 inputs.

Layer 2: 2 *sum nodes*, each aggregating results of 15 perceptron nodes.

Layer 3: 1 *argmax node*, selects the sum node with the largest output.

```
include "myLib.ht";

/** The Binary Neural Network (Forward-Pass only) **/
/*- Hyper-Parameters of the Network -*/
def sizePerceptronLayer = 30;
def sizeSumLayer = 2;

/*- Perceptron Layer -*/
function perceptronLayer(int[[]] perceptronWeights,
```

```
                          int[[]] input) {
  function signOf(int a) return
    if a > 0 then 1 else -1; }
  return map signOf on
    (tProduct2D(input, perceptronWeights));

/*- Sum Layer -*/
function sumLayer(int[[]] input) {
  def nNeuronsPerSum =
    (sizePerceptronLayer div sizeSumLayer);
  function tensorSum(int[[]] t) return
    (int)(reduce t with +);
  }
  return tensor i < sizeSumLayer -> (
    tensorSum(
      tensor j < nNeuronsPerSum ->
        (int)(input[0][nNeuronsPerSum*i + j]))
  );

/*- Argmax Layer -*/
function argmax(int[[]] t) {
  def max = reduce t with max;
  def maxIds = tensor i < (shape t)[0] ->
    if(((int)t[i]) == max) then i else sizeSumLayer;
  }
  return reduce maxIds with min;

/*- Auxiliary Random Generator -*/
function randomSign() {
  function randomBit() return
    (int)random 2.0; }
  return 1 - (2 * randomBit());

/*- Get input -*/
def sizeInput = 51;
def input =
  tensor (i,j) < (1,sizeInput) -> randomSign();
```

```
def output =
  argmax(sumLayer(perceptronLayer(
    tensor (i,j) < (sizeInput,sizePerceptronLayer) ->
      randomSign(),
    input)));
```

7.8 Parallel code generation and costs

The current implementation for HTL is an interpreter-compiler written in about 5000 lines of OCaml. It includes parser, type analyzer (not including shape analysis), and sequential evaluation. Under development is a static shape analysis, parallel code generator for BSML, a CUDA code generator. For the large family of neural-net programs whose tensor shapes are known statically, we show below the design of a BSP cost model (introduced in subsection 2.3.2) that estimates parallel execution time and memory usage for multicore and multinode. It can adapt to GPU targets by using a multi-level variant like [39] or [1].

We present here the parallel implementation of the primitives with their cost assuming a BSP implementation and tensors split along their first dimension. For example, a tensor t with shape $\rho(t) = [12, 36, 3]$ split in $p = 3$ processors would result in $\langle t_0, t_1, t_2 \rangle$ where $\rho(t_0) = \rho(t_1) = \rho(t_2) = [3, 36, 3]$. To ease understanding, we also assume that first dimension size is a multiple of p and thus tensors are evenly distributed (except when the tensor is a singleton). The shape $[n_0, \ldots, n_{p-1}]$ distributed (as previously shown) is

$$split([n_0, \ldots, n_{p-1}]) = \langle [n_0/p, n_1, \ldots, n_{p-1}], \ldots, [n_0/p, n_1, \ldots, n_{p-1}] \rangle$$

and sum of the first dimension size of a vector of tensors $t^{[x]}$ (notation $[x] = \{0, \ldots, x-1\}$) is

$$\sigma(t^{[x]}) = \sum_{i=0}^{x-1} \rho(t^i)_0.$$

The number of elements in a tensors of shape $\vec{n}$ is

$$\pi(\vec{n}) = \pi([n_0, \ldots, n_{D-1}]) = \prod_{i=0}^{D-1} n_i$$

and the cost of evaluating program function f is an estimate of its BSP evaluation time written $cost(f)$ and defined below on the language primitives. Characteristics of the BSP machine (reminder of subsection 2.3.2) of p processors are the global synchronization time l and the time g for collectively delivering at most one word.

Specification 7 ($\texttt{init}^{\texttt{par}}$)

Type: $\mathbb{N}^{+D} \to (\mathbb{N}^D \to T) \to T^{\diamond^{\texttt{par}}}$.

Input variables: $\vec{n} : \mathbb{N}^{+D}, \kappa : (\mathbb{N}^D \to T)$

Pre-conditions: $D > 0$

Output:
$$\begin{cases} \texttt{init}^{\texttt{par}}(\vec{n})(\kappa) = \langle t_0, \ldots, t_{p-1} \rangle \\ where \quad \forall \vec{n}_i \in split(\vec{n}).\ t_i = \texttt{init}(\vec{n}_i)(\kappa) \end{cases}$$

$\texttt{init}^{\texttt{par}}(\vec{n})(\kappa)$ splits the shape as shown previously and runs its sequential implementation for each processor with the local shape. Function κ is applied once per element. Having $\pi(\vec{n})$ elements evenly distributed in p processors, the cost of $\texttt{init}$ is thus $\pi(\vec{n})/p$ applications of κ.

$$cost(\texttt{init}^{\texttt{par}}(\vec{n})(\kappa)) = \frac{cost(\kappa) \cdot \pi(\vec{n})}{p}$$

Specification 8 ($\texttt{shape}^{\texttt{par}}$)

Type: $T^{\diamond^{\texttt{par}}} \to \left(\mathbb{N}^{+D}\right)^{\texttt{par}}$

Input variables: $t^{[p]} : T^{\diamond^{\texttt{par}}}$

Pre-conditions: *True*

Output:
$$\begin{cases} \texttt{shape}^{\texttt{par}}(t^{[p]}) = \langle \rho_0, \ldots, \rho_{p-1} \rangle \\ \ where\ (\rho_1 = \cdots = \rho_{p-1}), \\ \forall i \in [p].\ (\rho_i)_0 = \sigma(t^{[p]}), \\ \forall i \in [p], d \in \{1 \ldots D-1\}.\ (\rho_i)_d = (\texttt{shape}(t_i))_d \end{cases}$$

$\texttt{shape}^{\texttt{par}}(t^{[p]})$ first runs its sequential implementation locally then sends the first dimension size of local tensor to others, to be added in order to reconstruct the shape of the global tensor. Assuming that each p tensor computes its shape in constant time, each processor sends its first dimension to all others with cost gp. After a synchronization costing l, each processor adds p values to compute the first dimension.

$$cost(\texttt{shape}^{\texttt{par}}(t)) = g(p+1) + l$$

Specification 9 ($\texttt{content}^{\texttt{par}}$)

Type: $T^{\diamond^{\texttt{par}}} \to \mathbb{N}^* \to T^{\diamond^{\texttt{par}}}$.

Input variables: $t^{[p]} : T^{\diamond^{\texttt{par}}}, \vec{r} = [r_0, \ldots, r_{K-1}] : \mathbb{N}^*$

Pre-conditions:
$$\begin{cases} \texttt{shape}^{\texttt{par}}(t^{[p]}) = \rho^{[p]}, \forall i \in [p].\ \rho^i = [\rho^i_0, \ldots, \rho^i_{D-1}] \\ K \leq D,\ \forall d < K.\ r_d < \rho^i_d \end{cases}$$

Output:
$$\begin{cases} \texttt{content}^{\texttt{par}}(t^{[p]})(\vec{r}) = \langle t'_0, \ldots, t'_{p-1} \rangle \ \textit{where} \\ \textit{let } k = \min\{k \mid \sigma(t^{[k]}) > r^0\} - 1 \\ \textit{and } \theta = \texttt{content}([r^0 - \sigma(t^{[k]}) - 1, r^1, \ldots r^k])(t^k) \\ \textit{in } \langle t'_0, \ldots, t'_{p-1} \rangle = \texttt{init}^{\texttt{par}}(\rho(\theta))(\kappa(\theta)) \end{cases}$$

$\texttt{content}^{\texttt{par}}(t)(\vec{r})$ starts by running sequential `content` on the processor owning the tensor slice designated by the (second) index argument and distributes the among the processors. Note $\rho(t) = [n_0, \ldots, n_{D-1}]$ and k is the length of $\vec{r}$. The number of elements of the resulting tensor is $\pi([n_k, \ldots, n_{D-1}])$. Those elements are distributed over p processors. The cost to send to all others is gp while each communication carry m/p elements. After communication, a barrier costing l is executed for the former to be effective. In total, the cost is $gp(\pi([n_k, \ldots, n_{D-1}]))/p + l$.

$$cost(\texttt{content}^{\texttt{par}}(t)(\vec{r})) = g \cdot \pi([n_k, \ldots, n_{D-1}]) + l$$

Specification 10 ($\texttt{to\_scalar}^{\texttt{par}}$)

Type: $T^{\diamond^{\texttt{par}}} \to T^{\texttt{par}}$.

Input variables: $t^{[p]} : T^{\diamond^{\texttt{par}}}$

Pre-conditions: $\texttt{shape}^{\texttt{par}}(t^{[p]}) = \rho^{[p]}$ *where* $\rho^0 = [1, 1, \ldots, 1]$

Output:
$$\begin{cases} \texttt{to\_scalar}^{\texttt{par}}(t^{[p]}) = \langle a, \ldots, a \rangle \\ \textit{where } a = \texttt{to\_scalar}(t^0) \end{cases}$$

$\texttt{to\_scalar}^{\texttt{par}}(t)$ of a distributed tensor extract the value in t found at processor 0 which is sent to all other processors. Communication cost is gp. Barrier cost is l for its result to be available.

$$cost(\texttt{to\_scalar}^{\texttt{par}}(t)) = gp + l$$

Specification 11 ($\texttt{map}^{\texttt{par}}$)

Type: $(T_1 \to T_2) \to T_1^{\diamond\,\texttt{par}} \to T_2^{\diamond\,\texttt{par}}$

Input variables: $f : (T_1 \to T_2), t^{[p]} : T_1^{\diamond\,\texttt{par}}$

Pre-conditions: *Always defined*

Output:
$$\begin{cases} \texttt{map}^{\texttt{par}}(f)(\langle t_0, \ldots, t_{p-1}\rangle) = \langle t'_0, \ldots, t'_{p-1}\rangle \\ \textit{where } \forall i \in [p].\ t'_i = \texttt{map}(f)(t_i) \end{cases}$$

$\texttt{map}^{\texttt{par}}(f)(t)$ applies function f to all local elements. Total number of elements is $\pi(\rho(t))$ distributed in p processors. No barriers are needed because no communications are performed.

$$cost(\texttt{map}^{\texttt{par}}(f)(t)) = \frac{cost(f) \cdot \pi(\rho(t))}{p}$$

Specification 12 ($\texttt{reduce}^{\texttt{par}}$)

Type: $(T \to T \to T) \to T^{\diamond\,\texttt{par}} \to T^{\texttt{par}}$.

Input variables: $\oplus : (T \to T \to T), t^{[p]} : T^{\diamond\,\texttt{par}}$

Pre-conditions:
$$\begin{cases} t \textit{ is not empty} \\ \quad \textit{i.e. } \forall \rho_i \in \texttt{shape}^{\texttt{par}}(t^{[p]}).\ \rho_i \textit{ does not contain } o. \\ \oplus \textit{ is associative and commutative.} \end{cases}$$

Output:
$$\begin{cases} \texttt{reduce}^{\texttt{par}}(\oplus)(t) = \langle a, \ldots, a\rangle \\ \textit{where } a = \sum_{j\in[p]}^{\oplus} \texttt{reduce}(\oplus)(t_j) \end{cases}$$

$\texttt{reduce}^{\texttt{par}}(\oplus)(t)$ reduces the local tensors, sends local reduced values to all processors which then reduce values received with their own. Operator $\oplus$ is applied to local elements to reduce locally and is assumed to have a constant processing time. Total number of elements is $\pi(\rho(t))$ distributed in p processors. Local reduction results are sent from all to all, costing gp. A barrier costing l is performed for communications to be effective. The p values received are then reduced. In total, the cost is $\pi(\rho(t))/p + gp + l + p$.

$$cost(\texttt{reduce}^{\texttt{par}}(\oplus)(t)) = \frac{\pi(\rho(t))}{p} + g(p+1) + l$$

Cost will be integrated to static analysis in order to perform transformations of tensors aiming at minimizing the HTL program cost. An example of

such transformation would be the following. If the first tensor dimension size is smaller than p but a dimension exist for which its size is greatly higher than p, then the tensor could be statically transposed before being distributed. Operations performed on this tensor will also need to be transformed accordingly.

Correctness of the parallel implementation can be based on the following formal property of our primitives. Let $cat : T^{\diamond^{\texttt{par}}} \to T^{\diamond}$ be inverse of tensor distribution and let $proj : T^{\texttt{par}} \to T$ be the function extracting a replicated value to a scalar.

$$\begin{aligned}
cat \circ \texttt{content}^{\texttt{par}}(\vec{r}) &= \texttt{content}(\vec{r}) \circ cat \\
cat \circ \texttt{map}^{\texttt{par}}(f) &= \texttt{map}(f) \circ cat \\
proj \circ \texttt{reduce}^{\texttt{par}}(\oplus) &= \texttt{reduce}(\oplus) \circ cat \\
proj \circ \texttt{to_scalar}^{\texttt{par}} &= \texttt{to_scalar} \circ cat \\
split^{-1} \circ \texttt{shape}^{\texttt{par}} &= \texttt{shape} \circ cat \\
cat \circ \texttt{init}^{\texttt{par}} &= \texttt{init}
\end{aligned}$$

Moreover, in the absence of barrier-elimination (an optimization called "layer-fusion" in the context of neural nets) we have $\forall F_1, F_2$: primitives.

$$cost(F_1; F_2) = cost(F_1) + cost(F_2)$$

7.9 Conclusions

We have described a minimal but sufficient DSL for programming the neural nets as declarative parallel programs over tensors. Unlike MATLAB it has a very small base to allow for multiple code-generation and optimization techniques to be applied. Unlike DIESEL [13] it has a clear and declarative semantics. Unlike RELAY [54] it has a simple design (no polymorphism or dependent types) and very small set of parallel primitives to clarify implementation and parallel cost modeling.

Conclusion and Future Works

8

Contents

8.1 All roads lead to BSP

We have shown two different set of techniques for automatic generation of BSP programs.

8.1.1 BSP Automata

a) BSP automata from BSP regular expressions

BSP regular expressions may be seen as a declarative language to represent a set of words. BSP Automata are a machine solving the decision problem of the belonging to the word set represented by a BSP regular expression. This transformation was thus a compilation from a declarative language to a machine checking the belonging of a word to the language represented.

b) Determinization of BSP automata

When matching time is what matters the most (and no special hardware is involved[1]) non-deterministic automata have to give way to deterministic automata.

[1] see section 3.2.3.a)

For this reason, we treat non-deterministic BSP automata as an abstract machine whereas deterministic BSP Automata are the concrete BSP code responsible for matching BSP words. Determinization may thus also be seen as a compilation, from an abstract machine to concrete BSP code.

c) BSP regular expression matching

The tool-chain transformation from regular expression to BSP regular expression to non-deterministic BSP automata to deterministic BSP automata for parallel regular expression matching shows better matching time compared to existing approaches which keep the sequential DFA and speculate on the starting local state. However, the BSPA construction takes every possible distribution of the input into account which indeed removes the need to speculate (over the first local state as in usual approaches) but also increase greatly the automata size which in turn increase time and space required by determinization, leading to a gargantuan BSPA for high number of processors. Our approach still remains very efficient for small numbers of processors which, as should be recalled, is the majority of parallel computers, in particular those available for the general public.

This approach was a compilation from a sequential program, the regular expression to a BSP program, the BSP Automata.

8.1.2 BSP tensors computation

We introduced `HTL`, a minimal domain specific language (DSL) for tensor computation with only 6 tensor primitives. Those primitives include the constructor `init`, the transformers `map` and `content` (which returns a smaller tensor) and the destructors `to_scalar`, `shape` and `reduce`. These primitives were defined for a sequential and BSP implementation for which the BSP cost was designed. Its relevance was shown through a real world example. Its type system was introduced including tensor shape analysis which ought to be statically deduced in order to take the best decisions in terms of data distribution, similarly to [52].

8.2 Roads continue in BSP

8.2.1 BSP automata future

Next steps of parallel regular expression matching include first the optimization of BSPA implementation to reduce its size thereby decreasing further its matching time. Afterwards, extended regular expressions will be considered. Certification of the whole tool chain would be a great asset for the BSP automata theory. The proof started in Appendix A is the first step toward this goal. It would first require making conjecture 1 into a theorem by completing the proof for all cases. Thereafter would come the language preservation of determinization as well as the determinism of the determinization output with the lack of ϵ-transition and the uniqueness of output per local and global transitions. Concerning the parallelization of RE to BSPRE, the BSPRE language would have to be equal to the RE distributed language, for a block distribution. The latter transformation would also benefit from a proof that the BSPRE produced is minimal and a precise quantification or an upper approximation of the size of the BSPRE output.

It is the author's belief that BSPA applications have yet to be unraveled. The application of parallel regular expression matching had the merit to use all our transformations and especially, to be widely known. However, the first transformations from regular expression to BSP regular expression (BSPRE) creates BSPA too large to be scalable and the BSPRE is merely a disjunction of regular expression vector. It does not make use of neither Kleene closure nor concatenation (*i.e.* synchronization). Although we lean on this fact to justify that the cases covered by our conjecture 1 proof in Appendix A are sufficient. An application requiring the user to design the BSPRE himself would prevent the explosion due to the former transformation and make full use of BSPRE expressiveness. We still hope for the advent of an application putting BSP automata full power to use.

8.2.2 HTL for deep learning

We have only demonstrated that `HTL` is a productive and effective language design for programming neural-net *inference* i.e. the so-called forward, real-time computation. But neural net programs are useless without training to obtain high-quality inference from an optimization of their parameters. To this end we

will define a set of differentiation equations for tensor programs. Such a definition allows the automatic computation of a program's (inference quality *loss* function) derivative w.r.t. its weight parameters. The small number of primitives with simple mathematical definitions in `HTL` should make this process both safe and efficient.

It is also planned to increase hardware targets of HTL compilation to GPUs (Graphics processing units) for which number of approaches are available. We could generate directly CUDA code, the C API for programming Nvidia GPUs or the less specific OpenCL, an open standard for programming GPUs in C++. This is the low level approach. The high level approach would be to rely on SPOC [4], an OCaml framework targeting both CUDA and OpenCL. Model-based approaches would also be relevant such as BSP with BSPGP [25], a C framework consisting of a low amount of primitives targeting CUDA or multi-BSP with multi-ML [1], an extension of BSML for multi-BSP or SGL [39], a scatter-gather based language for heterogeneous architectures. Eventually, targeting the recent TPUs [29] (Tensor processing units) ought to be the logical sequel for which few programming languages are available yet.

Appendix

Proofs

A

Contents

We prove here that the transformation From BSP Regular Expression to BSP Automata defined in Chapter 4 preserves the language in section A.3. This transformation relies on Glushkov's algorithm which thus ought to be proven beforehand in section A.2. The first step of Glushkov's algorithm is the computation of four sets on which 4 properties are written. This appendix starts by the proof of these properties in section A.1. We also give our writing conventions in Table A.1 to help the reader navigate through the proof.

A.1 Glushkov's properties

A.1.1 Null

def 17.1 $Null' : (2^* \times \Sigma)\, re \to \{\{\epsilon\}, \emptyset\}$

$$
\begin{aligned}
Null'(\emptyset) &= \emptyset && (1)\\
Null'(\epsilon) &= \{\epsilon\} && (2)\\
Null'(x) &= \emptyset && (3)\\
Null'(\overline{F}+\overline{G}) &= Null'(\overline{F}) \cup Null'(\overline{G}) && (4)\\
Null'(\overline{F}\cdot\overline{G}) &= Null'(\overline{F}) \cap Null'(\overline{G}) && (5)\\
Null'(\overline{F}^*) &= \{\epsilon\} && (6)
\end{aligned}
$$

Symbols	**Explanation**	**Type**
a, u, v	symbol of a regular expression	Σ
w	sequence of symbols	Σ^*
	or sequence of localised symbols	$(\mathbb{2}^* \times \Sigma)^*$
x, y	localised symbols	$(\mathbb{2}^* \times \Sigma)$
E, F, G, R	regular expressions	$\Sigma\, re$
	or BSP regular expressions	$(\Sigma, p)\, bspre$
$\overline{E}, \overline{F}, \overline{G}, \overline{R}$	localization of a regular expression	$(\mathbb{2}^* \times \Sigma)\, re)$
$F^\circ, G^\circ, R^\circ$	BSPRE with vectors annotated	$(\Sigma, p)\, bspre^\circ$
A	Automaton	$\Sigma\, nfa$
	or BSP autoamton	$(\Sigma, p)\, bspa$
b	bit sequence	$\mathbb{2}^*$
	or just one bit	$\mathbb{2}$
t	unique identifier of vectors	$\mathbb{N}$
L	Regular language	$\mathcal{P}(\Sigma^*)$
	BSP language	$\mathcal{P}\big(\big((\Sigma^*)^p\big)^*\big)$
	Language of a R	$\Sigma\, re \to \mathcal{P}(\Sigma^*)$
	Language of a BSPRE	$(\Sigma, p)\, bspre \to \mathcal{P}\big(\big((\Sigma^*)^p\big)^*\big)$
Σ_{dsn}	alphabet including annotated semicolons	$\Sigma \cup \mathscr{S}$
δ_{dsn}	transitions labeled with alphabet Σ_{dsn}	$Q \times (\Sigma \cup \mathscr{S}) \to Q$

Table A.1 – *Writing conventions*

prop 17.1 (*Null*) **:**

$$Null(\overline{E}) = \begin{cases} \{\epsilon\} \text{ if } \epsilon \in L(\overline{E}) \\ \emptyset \quad \text{otherwise} \end{cases}$$

Proof We prove $Null = Null'$ by induction on $\overline{E}$. We write *IH* for induction hypothesis.

Case (1)	**Case (2)**	**Case (3)**	
$L(\emptyset) = \{\}$	$L(\epsilon) = \{\epsilon\}$	$L(x) = \{x\}$	*def*. L_{RE}
$\implies \epsilon \notin L(\emptyset)$	$\implies \epsilon \in L(\epsilon)$	$\implies \epsilon \notin L(x)$	
$\implies Null(\emptyset) = \emptyset$	$\implies Null(\epsilon) = \epsilon$	$\implies Null(x) = \emptyset$	
and $Null'(\emptyset) = \emptyset$	**and** $Null'(\epsilon) = \epsilon$	**and** $Null'(x) = \emptyset$	*def*. $Null'$

Case (4)

$$Null(\overline{F}+\overline{G}) = \begin{cases} \{\epsilon\} \text{ if } \epsilon \in L(\overline{F}+\overline{G}) \\ \emptyset \quad \text{otherwise} \end{cases} \quad [\ \textit{prop. Null}\ ]$$

$$= \begin{cases} \{\epsilon\} \text{ if } \epsilon \in (L(\overline{F}) \cup L(\overline{G})) \\ \emptyset \quad \text{otherwise} \end{cases} \quad [\ \textit{def}.\ L_{RE}\ ]$$

$$= \begin{cases} \{\epsilon\} \text{ if } (\epsilon \in L(\overline{F})) \vee (\epsilon \in L(\overline{G})) \\ \emptyset \quad \text{otherwise} \end{cases}$$

$$= \begin{cases} \{\epsilon\} \text{ if } (Null(\overline{F}) = \{\epsilon\}) \vee (Null(\overline{G}) = \{\epsilon\}) \\ \emptyset \quad \text{otherwise} \end{cases} \quad [\ \textit{prop. Null}\ ]$$

All possible cases are displayed in the following table.

$Null(\overline{F})$	$Null(\overline{G})$	$Null(\overline{F}+\overline{G})$	$Null(\overline{F}) \cup Null(\overline{G})$
$\{\epsilon\}$	$\{\epsilon\}$	$\{\epsilon\}$	$\{\epsilon\}$
$\{\epsilon\}$	$\emptyset$	$\{\epsilon\}$	$\{\epsilon\}$
$\emptyset$	$\{\epsilon\}$	$\{\epsilon\}$	$\{\epsilon\}$
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Which is equivalent to

$$\begin{aligned} Null(\overline{F}+\overline{G}) &= Null(\overline{F}) \cup Null(\overline{G}) \\ &= Null'(\overline{F}) \cup Null'(\overline{G}) && [\ IH\ ] \\ &= Null'(\overline{F}+\overline{G}) && [\ def.\ Null'\ ] \end{aligned}$$

Case (5)

$$\begin{aligned} Null(\overline{F} \cdot \overline{G}) &= \begin{cases} \{\epsilon\} \text{ if } \epsilon \in L(\overline{F} \cdot \overline{G}) \\ \emptyset \quad \text{otherwise} \end{cases} && [\ prop.\ Null\ ] \\ &= \begin{cases} \{\epsilon\} \text{ if } \epsilon \in (L(\overline{F}) \cdot L(\overline{G})) \\ \emptyset \quad \text{otherwise} \end{cases} && [\ def.\ L_{RE}\ ] \\ &= \begin{cases} \{\epsilon\} \text{ if } (\epsilon \in L(\overline{F})) \wedge (\epsilon \in L(\overline{G})) \\ \emptyset \quad \text{otherwise} \end{cases} \\ &= \begin{cases} \{\epsilon\} \text{ if } (Null(\overline{F}) = \{\epsilon\}) \wedge (Null(\overline{G}) = \{\epsilon\}) \\ \emptyset \quad \text{otherwise} \end{cases} && [\ prop.\ Null\ ] \end{aligned}$$

Likewise

$Null(\overline{F})$	$Null(\overline{G})$	$Null(\overline{F} \cdot \overline{G})$	$Null(\overline{F}) \cap Null(\overline{G})$
$\{\epsilon\}$	$\{\epsilon\}$	$\{\epsilon\}$	$\{\epsilon\}$
$\{\epsilon\}$	$\emptyset$	$\emptyset$	$\emptyset$
$\emptyset$	$\{\epsilon\}$	$\emptyset$	$\emptyset$
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

$$
\begin{aligned}
\iff Null(\overline{F} \cdot \overline{G}) &= Null(\overline{F}) \cap Null(\overline{G}) \\
&= Null'(\overline{F}) \cap Null'(\overline{G}) && [\ IH\ ] \\
&= Null'(\overline{F} \cdot \overline{G}) && [\ def.\ Null'\ ]
\end{aligned}
$$

Case (6)

$$
\begin{aligned}
Null(\overline{F}^*) &= \textbf{if } \epsilon \in L(\overline{F}^*) \textbf{ then } \{\epsilon\} \textbf{ else } \emptyset && [\ prop.\ Null\ ] \\
&= \textbf{if } \epsilon \in (\bigcup_{i=0}^{\infty} L(F^i)) \textbf{ then } \{\epsilon\} \textbf{ else } \emptyset && [\ def.\ L_{RE}\ ] \\
Null(\overline{F}^*) &= \{\epsilon\} = Null'(\overline{F}^*) && [\ def.\ F^i\ ]
\end{aligned}
$$

□

Let $\mathbb{P} = (\mathbb{2}^* \times \Sigma)$

A.1.2 First

def 17.2 *First′* **:** $(\mathbb{2}^* \times \Sigma)\, re \to \mathcal{P}(\mathbb{2}^* \times \Sigma)$

$$
\begin{aligned}
First'(\emptyset) &= \emptyset && (1)\\
First'(\epsilon) &= \emptyset && (2)\\
First'(x) &= \{x\} && (3)\\
First'(\overline{F}+\overline{G}) &= First'(\overline{F}) \cup First'(\overline{G}) && (4)\\
First'(\overline{F} \cdot \overline{G}) &= First'(\overline{F}) \cup && (5)\\
&\quad\ Null(\overline{F}) \cdot First'(\overline{G}) &&\\
First'(\overline{F}^*) &= First'(\overline{F}) && (6)
\end{aligned}
$$

prop 17.2 (*First*) **:**

$$First(\overline{E}) = \left\{ x \in (\mathbb{2}^* \times \Sigma) \mid \exists w \in (\mathbb{2}^* \times \Sigma)^* : xw \in L(\overline{E}) \right\}$$

Proof We prove $First = First'$ by induction on $\overline{E}$.

Case (1)	**Case (2)**	**Case (3)**	
$First(\emptyset)$	$First(\epsilon)$	$First(x)$	
$= \{y \mid \exists w, yw \in L(\emptyset)\}$	$= \{y \mid \exists w, yw \in L(\epsilon)\}$	$= \{y \mid \exists w, yw \in L(x)\}$	*prop. First*
$= \{y \mid \exists w, yw \in \emptyset\}$	$= \{y \mid \exists w, yw \in \{\epsilon\}\}$	$= \{y \mid \exists w, yw \in \{x\}\}$	*def.* L_{RE}
$= \emptyset$	$= \emptyset \quad [\, \epsilon \notin \mathbb{P} \Rightarrow y \neq \epsilon \,]$	$= \{x\} \qquad [\, w = \epsilon \,]$	
$= First'(\emptyset)$	$= First'(\epsilon)$	$= First'(x)$	*def. First′*

Case (4)

$$
\begin{aligned}
First(\overline{F}+\overline{G}) &= \left\{x \mid \exists w, xw \in L(\overline{F}+\overline{G})\right\} && [\ \textit{prop. First}\ ]\\
&= \left\{x \mid \exists w, xw \in (L(\overline{F}) \cup L(\overline{G}))\right\} && [\ \textit{def. } L_{RE}\ ]\\
&= \left\{x \mid \exists w, xw \in L(\overline{F})\right\} \cup \left\{x \mid \exists w, xw \in L(\overline{G})\right\}\\
&= First(\overline{F}) \cup First(\overline{G}) && [\ \textit{prop. First}\ ]\\
&= First'(\overline{F}) \cup First'(\overline{G}) && [\ \textit{IH}\ ]\\
First(\overline{F}+\overline{G}) &= First'(\overline{F}+\overline{G}) && [\ \textit{def. First}'\ ]
\end{aligned}
$$

Case (5)

$$
\begin{aligned}
&First(\overline{F} \cdot \overline{G})\\
&= \left\{x \mid \exists w, xw \in L(\overline{F} \cdot \overline{G})\right\} && [\ \textit{prop. First}\ ]\\
&= \left\{x \mid \exists w, xw \in (L(\overline{F}) \cdot L(\overline{G}))\right\} && [\ \textit{def. } L_{RE}\ ]\\
&= \left\{x \mid \exists w, xw \in \left\{w_1 w_2 \mid w_1 \in L(\overline{F}), w_2 \in L(\overline{G})\right\}\right\} && [\ \textit{def. } L_1 \cdot L_2\ ]
\end{aligned}
$$

Two cases are distinguished according to whether ϵ belong to $L(\overline{F})$.

- $\epsilon \notin L(\overline{F})$

$$
\begin{aligned}
&First(\overline{F} \cdot \overline{G})\\
&= \left\{x \mid \exists w', xw' \in \left\{w_1 \mid w_1 \in L(\overline{F})\right\}\right\} && \left[\begin{array}{rcl}\epsilon \notin L(\overline{F}) &\implies& |w_1| > 0\\ &\implies& x \notin w_2\end{array}\right]\\
&= \left\{x \mid \exists w_1', xw_1' \in L(\overline{F})\right\} && [\ w_1 = xw_1'\ ]\\
&= First(\overline{F}) && [\ \textit{prop. First}\ ]
\end{aligned}
$$

- $\epsilon \in L(\overline{F})$

First, we consider the following subcase

- ▷ $L(\overline{F}) = \{\epsilon\}$

$$
\begin{aligned}
& First(\overline{F} \cdot \overline{G}) \\
&= \left\{ x \mid \exists w, xw \in \left\{ \epsilon w_2 \mid \epsilon \in L(\overline{F}), w_2 \in L(\overline{G}) \right\} \right\} && [\ w_1 = \epsilon\ ] \\
&= \left\{ x \mid \exists w, xw \in \left\{ w_2 \mid w_2 \in L(\overline{G}) \right\} \right\} \\
&= \left\{ x \mid \exists w_2', xw_2' \in L(\overline{G}) \right\} && [\ w_2 = xw_2'\ ] \\
&= First(\overline{G}) && [\ \textit{prop. First}\ ]
\end{aligned}
$$

▷ $\epsilon \in L(\overline{F})$

$$
\begin{aligned}
& First(\overline{F} \cdot \overline{G}) \\
&= \left\{ x \mid \exists w, xw \in \begin{pmatrix} \left\{ \epsilon\, w_2 \mid \epsilon \in L(\overline{F}), w_2 \in L(\overline{G}) \right\} \\ \cup \left\{ w_1 w_2 \mid w_1 \in L(\overline{F}), w_2 \in L(\overline{G}) \right\} \end{pmatrix} \right\} \begin{matrix} [\ L(\overline{F}) = \{\epsilon\}\ ] \\ [\ \epsilon \notin L(\overline{F})\ ] \end{matrix} \\
&= \begin{pmatrix} \left\{ x \mid \exists w, xw \in \left\{ w_2 \mid w_2 \in L(\overline{G}) \right\} \right\} \\ \cup \left\{ x \mid \exists w', xw' \in \left\{ w_1 \mid w_1 \in L(\overline{F}) \right\} \right\} \end{pmatrix} \\
&= First(\overline{G}) \cup First(\overline{F}) && [\ \text{from cases } L(\overline{F}) = \{\epsilon\} \text{ and } \epsilon \notin L(\overline{F})\ ]
\end{aligned}
$$

Therefore,

$$
\begin{aligned}
First(\overline{F} \cdot \overline{G}) &= \begin{cases} Null(\overline{F}) = \emptyset & \Longrightarrow First(\overline{F}) \\ Null(\overline{F}) = \{\epsilon\} & \Longrightarrow First(\overline{F}) \cup First(\overline{G}) \end{cases} \\
&= \begin{cases} Null(\overline{F}) = \emptyset & \Longrightarrow First(\overline{F}) \cup First(\overline{G}) \cdot \emptyset && [\ First(\overline{G}) \cdot \emptyset = \emptyset\ ] \\ Null(\overline{F}) = \{\epsilon\} & \Longrightarrow First(\overline{F}) \cup First(\overline{G}) \cdot \{\epsilon\} && [\ First(\overline{G}) \cdot \{\epsilon\} = First(\overline{G})\ ] \end{cases} \\
&= \begin{cases} Null(\overline{F}) = \emptyset & \Longrightarrow First(\overline{F}) \cup First(\overline{G}) \cdot Null(\overline{F}) && [\ Null(\overline{F}) = \emptyset\ ] \\ Null(\overline{F}) = \{\epsilon\} & \Longrightarrow First(\overline{F}) \cup First(\overline{G}) \cdot Null(\overline{F}) && [\ Null(\overline{F}) = \{\epsilon\}\ ] \end{cases} \\
&= First(\overline{F}) \cup First(\overline{G}) \cdot Null(\overline{F}) \\
&= First'(\overline{F}) \cup First'(\overline{G}) \cdot Null(\overline{F}) && [\ \textit{IH}\ ] \\
&= First'(\overline{F} \cdot \overline{G}) && [\ \textit{def. First'}\ ]
\end{aligned}
$$

Case (6)

prop 40.1 (First of iteration) **:** Let $P(i)$ the property $First(\overline{F}^i) = First(\overline{F})$

Proof We prove that $\forall i > 0, P(i)$ holds.

$\circ$ $First(\overline{F}^1) = First(\overline{F})$ [Base case]

$$
\begin{aligned}
\circ\ First(\overline{F}^{i>1}) &= \left\{x \mid \exists w, xw \in L(\overline{F}^i)\right\} && [\ \textit{prop. First}\ ]\\
&= \left\{x \mid \exists w, xw \in L(\overline{F}^{i-1}.\overline{F})\right\} && [\ \textit{def. } F^i\ ]\\
&= \left\{x \mid \exists w, xw \in L(\overline{F}^{i-1}) \cdot L(\overline{F})\right\} && [\ \textit{def. } L_{RE}\ ]\\
&= First(\overline{F}^{i-1}) \cup First(\overline{F}) \cdot Null(\overline{F}^{i-1}) && [\ \text{prf. (5)}\ ]\\
&= First(\overline{F}) \cup First(\overline{F}) \cdot Null(\overline{F}^{i-1}) && [\ \textit{IH}\ ]
\end{aligned}
$$

$$
First(F^{i>1}) = \begin{cases} Null(\overline{F}^{i-1}) = \{\epsilon\} \implies First(\overline{F}) \cup First(\overline{F}) = First(\overline{F}) \\ Null(\overline{F}^{i-1}) = \emptyset \implies First(\overline{F}) \cup \emptyset = First(\overline{F}) \end{cases}
$$

$$
First(\overline{F}^{i>1}) = First(\overline{F})
$$

□

Hence,

$$
\begin{aligned}
First(\overline{F}^*) &= \left\{x \mid \exists w, xw \in \bigcup_{i=0}^{\infty} L(F^i)\right\} && [\ \textit{prop. First}\ ]\\
&= \left\{x \mid \exists w, xw \in L(F^0)\right\} \cup \left\{x \mid \exists w, xw \in \bigcup_{i=1}^{\infty} L(F^i)\right\}\\
&= \left\{x \mid \exists w, xw \in \{\epsilon\}\right\} \cup \bigcup_{i=1}^{\infty}\left\{x \mid \exists w, xw \in L(F^i)\right\}\\
&= \emptyset \cup \bigcup_{i=1}^{\infty} First(\overline{F}^i) && [\ xw \notin \{\epsilon\}\ \text{ and }\ \textit{prop. First}\ ]\\
&= First(\overline{F}) && [\ \forall i > 0, First(\overline{F}^i) = First(\overline{F})\ ]\\
First(\overline{F}^*) &= First'(\overline{F}) && [\ \textit{IH}\ ]
\end{aligned}
$$

□

A.1.3 Last

def 17.3 $Last' : (\mathbb{2}^* \times \Sigma)\, re \to \mathcal{P}(\mathbb{2}^* \times \Sigma)$

$$
\begin{aligned}
Last'(\emptyset) &= \emptyset && (1)\\
Last'(\epsilon) &= \emptyset && (2)\\
Last'(x) &= \{x\} && (3)\\
Last'(\overline{F}+\overline{G}) &= Last'(\overline{F}) \cup Last'(\overline{G}) && (4)\\
Last'(\overline{F} \cdot \overline{G}) &= Last'(\overline{G}) \cup && (5)\\
&\quad\ Null(\overline{G}) \cdot Last'(\overline{F}) &&\\
Last'(\overline{F}^*) &= Last'(\overline{F}) && (6)
\end{aligned}
$$

prop 17.3 (*Last*) **:**

$$Last(\overline{E}) = \left\{ x \in (\mathbb{2}^* \times \Sigma) \mid \exists w \in (\mathbb{2}^* \times \Sigma)^* : wx \in L(\overline{E}) \right\}$$

Proof We prove $Last = Last'$ by induction on $\overline{E}$.

Case (1)	**Case (2)**	**Case (3)**	
$Last(\emptyset)$	$Last(\epsilon)$	$Last(x)$	
$= \{y \mid wy \in L(\emptyset)\}$	$= \{y \mid wy \in L(\epsilon)\}$	$= \{y \mid wy \in L(x)\}$	*prop. Last*
$= \{y \mid wy \in \{\}\}$	$= \{y \mid wy \in \{\epsilon\}\}$	$= \{y \mid wy \in \{x\}\}$	*def.* L_{RE}
$= \emptyset$	$= \emptyset \quad [\ \epsilon \notin \mathbb{P} \Rightarrow y \neq \epsilon\ ]$	$= \{x\} \quad [\ w = \epsilon\ ]$	
$= Last(\emptyset)$	$= Last(\epsilon)$	$= Last(x)$	*def.* $Last'$

Case (4)

$$
\begin{aligned}
Last(\overline{F}+\overline{G}) &= \left\{ x \mid \exists w, wx \in L(\overline{F}+\overline{G}) \right\} && [\ \textit{prop. First}\ ]\\
&= \left\{ x \mid \exists w, wx \in (L(\overline{F}) \cup L(\overline{G})) \right\} && [\ \textit{def.}\ L_{RE}\ ]\\
&= \left\{ x \mid \exists w, wx \in L(\overline{F}) \right\} \cup \left\{ x \mid \exists w, wx \in L(\overline{G}) \right\}\\
Last(\overline{F}+\overline{G}) &= Last(\overline{F}) \cup Last(\overline{G}) && [\ \textit{IH}\ ]
\end{aligned}
$$

Case (5)

$$
\begin{aligned}
& Last(\overline{F} \cdot \overline{G}) \\
&= \left\{ x \mid \exists w, wx \in L(\overline{F} \cdot \overline{G}) \right\} && [\ \textit{prop. Last}\ ] \\
&= \left\{ x \mid \exists w, wx \in (L(\overline{F}) \cdot L(\overline{G})) \right\} && [\ \textit{def.}\ L_{RE}\ ] \\
&= \left\{ x \mid \exists w, wx \in \left\{ w_1 w_2 \mid w_1 \in L(\overline{F}), w_2 \in L(\overline{G}) \right\} \right\} && [\ \textit{def.}\ L_1 \cdot L_2\ ]
\end{aligned}
$$

Two cases are distinguished according to whether ϵ belong to $L(\overline{G})$.

- $\epsilon \notin L(\overline{G})$

$$
\begin{aligned}
& Last(\overline{F} \cdot \overline{G}) \\
&= \left\{ x \in \mathbb{P} \mid \exists w' \in \mathbb{P}^*, w'x \in \left\{ w_2 \mid w_2 \in L(\overline{G}) \right\} \right\} && \left[\begin{array}{rl} \epsilon \notin L(\overline{G}) & \implies |w_2| > 0 \\ & \implies x \notin w_1 \end{array} \right] \\
&= \left\{ x \in \mathbb{P} \mid \exists w_2' \in \mathbb{P}^*, w_2' x \in L(\overline{G}) \right\} && [w_2 = w_2' x] \\
&= Last(\overline{G}) && [\textit{prop. Last}]
\end{aligned}
$$

- $\epsilon \in L(\overline{G})$

First, we consider the following subcase

- ▷ $L(\overline{G}) = \{\epsilon\}$

$$
\begin{aligned}
& Last(\overline{F} \cdot \overline{G}) \\
&= \left\{ x \mid \exists w, wx \in \left\{ w_1 \epsilon \mid w_1 \in L(\overline{F}), \epsilon \in L(\overline{G}) \right\} \right\} && [\ w_2 = \epsilon\ ] \\
&= \left\{ x \mid \exists w, wx \in \left\{ w_1 \mid w_1 \in L(\overline{F}) \right\} \right\} \\
&= \left\{ x \mid \exists w_1', w_1' x \in L(\overline{F}) \right\} && [\ w_1 = x w_1'\ ] \\
&= Last(\overline{F}) && [\ \textit{prop. Last}\ ]
\end{aligned}
$$

The case $\epsilon \in L(\overline{G})$ is the union of the two previous studied cases

$$
\begin{aligned}
& Last(\overline{F} \cdot \overline{G}) \\
&= \left\{ x \mid \exists w, wx \in \begin{pmatrix} \left\{ w_1\epsilon \mid w_1 \in L(\overline{F}), \epsilon \in L(\overline{G}) \right\} \\ \cup \left\{ w_1 w_2 \mid w_1 \in L(\overline{F}), w_2 \in L(\overline{G}) \right\} \end{pmatrix} \right\} \begin{matrix} [\, L(\overline{G}) = \{\epsilon\} \,] \\ [\, \epsilon \notin L(\overline{G}) \,] \end{matrix} \\
&= \begin{pmatrix} \left\{ x \mid \exists w, wx \in \left\{ w_1 \mid w_1 \in L(\overline{F}) \right\} \right\} \\ \cup \left\{ x \mid \exists w', w'x \in \left\{ w_2 \mid w_2 \in L(\overline{G}) \right\} \right\} \end{pmatrix} \\
&= Last(\overline{F}) \cup Last(\overline{G}) \qquad [\, \text{from } L(\overline{G}) = \{\epsilon\} \text{ and } \epsilon \notin L(\overline{G}) \,]
\end{aligned}
$$

Therefore,

$$
\begin{aligned}
& Last(\overline{F} \cdot \overline{G}) \\
&= \begin{cases} Null(\overline{G}) = \emptyset & \Longrightarrow Last(\overline{G}) \\ Null(\overline{G}) = \{\epsilon\} & \Longrightarrow Last(\overline{G}) \cup Last(\overline{F}) \end{cases} \\
&= \begin{cases} Null(\overline{G}) = \emptyset & \Longrightarrow Last(\overline{G}) \cup Last(\overline{F}) \cdot \emptyset & [\, Last(\overline{G}) \cdot \emptyset = \emptyset \,] \\ Null(\overline{G}) = \{\epsilon\} & \Longrightarrow Last(\overline{G}) \cup Last(\overline{F}) \cdot \{\epsilon\} & [\, Last(\overline{G}) \cdot \{\epsilon\} = Last(\overline{G}) \,] \end{cases} \\
&= \begin{cases} Null(\overline{G}) = \emptyset & \Longrightarrow Last(\overline{G}) \cup Last(\overline{F}) \cdot Null(\overline{G}) & [\, Null(\overline{G}) = \emptyset \,] \\ Null(\overline{G}) = \{\epsilon\} & \Longrightarrow Last(\overline{G}) \cup Last(\overline{F}) \cdot Null(\overline{G}) & [\, Null(\overline{G}) = \{\epsilon\} \,] \end{cases} \\
&= Last(\overline{G}) \cup Last(\overline{F}) \cdot Null(\overline{G}) \\
&= Last'(\overline{G}) \cup Last'(\overline{F}) \cdot Null(\overline{G}) \qquad [\, IH \,] \\
&= Last'(\overline{F} \cdot \overline{G}) \qquad [\, def.\ Last' \,]
\end{aligned}
$$

Case (6) In the same manner as First,

prop 40.2 (Last of iteration) **:** Let $P(i)$ the property : $Last(\overline{F}^i) = Last(\overline{F})$

Proof We prove that $\forall i > 0, P(i)$ holds.

$\circ$ $Last(\overline{F}^1) = Last(\overline{F})$ [Base case]

$\circ$
$$
\begin{aligned}
Last(\overline{F}^{i>1}) &= \left\{ x \mid \exists w, wx \in L(\overline{F}^i) \right\} && [\, prop.\ Last \,] \\
&= \left\{ x \mid \exists w, wx \in L(\overline{F}.\overline{F}^{i-1}) \right\} && [\, def.\ F^i \,] \\
&= \left\{ x \mid \exists w, wx \in L(\overline{F}) \cdot L(\overline{F}^{i-1}) \right\} && [\, prop.\ Last \,] \\
&= Last(\overline{F}^{i-1}) \cup Last(\overline{F}) \cdot Null(\overline{F}^{i-1}) && [\, \text{prf. (5)} \,] \\
&= Last(\overline{F}) \cup Last(\overline{F}) \cdot Null(\overline{F}^{i-1}) && [\, IH \,]
\end{aligned}
$$

$$Last(F^{i>1}) = \begin{cases} Null(\overline{F}^{i-1}) = \{\epsilon\} \implies Last(\overline{F}) \cup Last(\overline{F}) = Last(\overline{F}) \\ Null(\overline{F}^{i-1}) = \emptyset \implies Last(\overline{F}) \cup \emptyset = Last(\overline{F}) \end{cases}$$
$$Last(\overline{F}^{i>1}) = Last(\overline{F})$$

□

Hence,

$$\begin{aligned} Last(\overline{F}^*) &= \left\{ x \mid \exists w, wx \in \bigcup_{i=0}^{\infty} L(F^i) \right\} && [\ \textit{prop. Last}\ ] \\ &= \left\{ x \mid \exists w, wx \in L(F^0) \right\} \cup \left\{ x \mid \exists w, wx \in \bigcup_{i=1}^{\infty} L(F^i) \right\} \\ &= \left\{ x \mid \exists w, wx \in \{\epsilon\} \right\} \cup \bigcup_{i=1}^{\infty} \left\{ x \mid \exists w, wx \in L(F^i) \right\} \\ &= \emptyset \cup \bigcup_{i=1}^{\infty} Last(\overline{F}^i) && [\ wx \notin \{\epsilon\} \ \text{ and } \ \textit{prop. Last}\ ] \\ &= Last(\overline{F}) && [\ \forall i > 0, Last(\overline{F}^i) = Last(\overline{F})\ ] \\ Last(\overline{F}^*) &= Last'(\overline{F}) && [\ \textit{IH}\ ] \end{aligned}$$

□

A.1.4 Follow

def 17.4 $Follow' : (2^* \times \Sigma)\, re \to (2^* \times \Sigma) \to \mathcal{P}(2^* \times \Sigma)$

$$
\begin{aligned}
Follow'(\emptyset, x) &= \emptyset && (1)\\
Follow'(\epsilon, x) &= \emptyset && (2)\\
Follow'(a, x) &= \emptyset && (3)\\
Follow'(\overline{F}+\overline{G}, x) &= \begin{cases} Follow'(\overline{F}, x) & \textbf{if } x \in pos(\overline{F}) \\ Follow'(\overline{G}, x) & \textbf{if } x \in pos(\overline{G}) \end{cases} && (4)\\
Follow'(\overline{F} \cdot \overline{G}, x) &= \begin{cases} Follow'(\overline{F}, x) \cup First(\overline{G}) & \textbf{if } x \in Last(\overline{F}) \\ Follow'(\overline{F}, x) & \textbf{if } x \in pos(\overline{F}) \setminus Last(\overline{F}) \\ Follow'(\overline{G}, x) & \textbf{if } x \in pos(\overline{G}) \end{cases} && (5)\\
Follow'(\overline{F}^*, x) &= \begin{cases} Follow'(\overline{F}, x) \cup First(\overline{F}) & \textbf{if } x \in Last(\overline{F}) \\ Follow'(\overline{F}, x) & \textbf{if } x \in pos(\overline{F}) \setminus Last(\overline{F}) \end{cases} && (6)
\end{aligned}
$$

prop 17.4 (*Follow*) **:**

$$Follow(\overline{E}, x) = \left\{ y \in (2^* \times \Sigma) \mid \exists v \in (2^* \times \Sigma)^*, \exists w \in (2^* \times \Sigma)^* : vxyw \in L(\overline{E}) \right\}$$

Proof We prove $Follow = Follow'$ by induction on $\overline{E}$.

Case (1)

$$
\begin{aligned}
Follow(\emptyset, x) &= \left\{ y \mid \exists v, \exists w, vxyw \in L(\emptyset) \right\} && [\ prop.\ Follow\ ]\\
Follow(\emptyset, x) &= \left\{ y \mid \exists v, \exists w, vxyw \in \{\} \right\} && [\ def.\ L_{RE}\ ]\\
Follow(\emptyset, x) &= \emptyset = Follow'(\emptyset, x)
\end{aligned}
$$

Case (2)

$$
\begin{aligned}
Follow(\epsilon, x) &= \left\{ y \mid \exists v, \exists w, vxyw \in L(\epsilon) \right\} && [\ prop.\ Follow\ ]\\
Follow(\epsilon, x) &= \left\{ y \mid \exists v, \exists w, vxyw \in \{\epsilon\} \right\} && [\ def.\ L_{RE}\ ]\\
Follow(\epsilon, x) &= \emptyset = Follow'(\epsilon, x)
\end{aligned}
$$

Case (3)

$$\begin{aligned} Follow(a,x) &= \Big\{ y \mid \exists v, \exists w, vxyw \in L(a) \Big\} && [\ prop.\ Follow\ ] \\ Follow(a,x) &= \Big\{ y \mid \exists v, \exists w, vxyw \in \{a\} \Big\} && [\ def.\ L_{RE}\ ] \\ Follow(a,x) &= \emptyset = Follow'(a,x) \end{aligned}$$

Case (4)

$$\begin{aligned} Follow(\overline{F}+\overline{G},x) &= \Big\{ y \mid \exists v, \exists w, vxyw \in L(\overline{F}+\overline{G}) \Big\} && [\ prop.\ Follow\ ] \\ Follow(\overline{F}+\overline{G},x) &= \Big\{ y \mid \exists v, \exists w, vxyw \in \big(L(\overline{F}) \cup L(\overline{G})\big) \Big\} && [\ def.\ L_{RE}\ ] \\ Follow(\overline{F}+\overline{G},x) &= \Big\{ y \mid \exists v, \exists w, vxyw \in L(\overline{F}) \Big\} \cup \Big\{ y \mid \exists v, \exists w, vxyw \in L(\overline{G}) \Big\} \end{aligned}$$

Because of *posex* (localization of symbols (done before *glushkov*)), we know that x is unique. So,

$$Follow(\overline{F}+\overline{G},x) = \begin{cases} \Big\{ y \in \mathbb{P} \mid \exists v \in \mathbb{P}^*, \exists w \in \mathbb{P}^*, vxyw \in L(\overline{F}) \Big\} & \textbf{if } x \in pos(\overline{F}) \\ \Big\{ y \in \mathbb{P} \mid \exists v \in \mathbb{P}^*, \exists w \in \mathbb{P}^*, vxyw \in L(\overline{G}) \Big\} & \textbf{if } x \in pos(\overline{G}) \end{cases}$$

$$Follow(\overline{F}+\overline{G},x) = \left\{ \begin{array}{ll} Follow(\overline{F},x) & \textbf{if } x \in pos(\overline{F}) \\ Follow(\overline{G},x) & \textbf{if } x \in pos(\overline{G}) \end{array} \right\} = Follow'(\overline{F}+\overline{G},x) \qquad [\ IH\ ]$$

Case (5)

$$\begin{aligned} Follow(\overline{F}\cdot\overline{G},x) &= \Big\{ y \mid \exists v, \exists w, vxyw \in L(\overline{F}\cdot\overline{G}) \Big\} && [\ prop.\ Follow\ ] \\ Follow(\overline{F}\cdot\overline{G},x) &= \Big\{ y \mid \exists v, \exists w, vxyw \in L(\overline{F})\cdot L(\overline{G}) \Big\} && [\ def.\ L_{RE}\ ] \end{aligned}$$

There are several possible cases regarding x membership.

With $w = w_1 w_2$ and $v = v_1 v_2$:

$$
\begin{aligned}
x \in pos(\overline{F}) &\implies vxyw_1 \in L(\overline{F}) \text{ and } && w_2 \in L(\overline{G}) \\
x \in Last(\overline{F}) &\implies vx \quad\ \in L(\overline{F}) \text{ and } && yw \in L(\overline{G}) \\
x \in pos(\overline{G}) &\implies v_1 \quad\ \in L(\overline{F}) \text{ and } && v_2xyw \in L(\overline{G})
\end{aligned}
$$

Therefore,

$$
Follow(\overline{F}\cdot\overline{G},x) = \begin{cases}
\left\{y \mid \exists v, \exists w_1, vxyw_1 \in L(\overline{F})\right\} & \textbf{if } x \in pos(\overline{F}) \\
\left\{y \mid \exists w, yw \in L(\overline{G})\right\} & \textbf{if } x \in Last(\overline{F}) \\
\left\{y \mid \exists v_2, \exists w_1, v_2xyw \in L(\overline{G})\right\} & \textbf{if } x \in pos(\overline{G})
\end{cases}
$$

$$
Follow(\overline{F}\cdot\overline{G},x) = \begin{cases}
Follow(\overline{F},x) & [\ \textit{prop. Follow}\ ] \ \textbf{if } x \in pos(\overline{F}) & (a) \\
First(\overline{G}) & [\ \textit{prop. First}\ ] \ \textbf{if } x \in Last(\overline{F}) & (b) \\
Follow(\overline{G},x) & [\ \textit{prop. Follow}\ ] \ \textbf{if } x \in pos(\overline{G}) & (c)
\end{cases}
$$

Case (b) is included in case (a) because $Last(\overline{F}) \subset pos(\overline{F})$.

So, if $x \in Last(\overline{F})$ then cases (a) and (b) are taken, thus $Follow(\overline{F}\cdot\overline{G},x) = First(\overline{G}) \cup Follow(\overline{F},x)$.

With the distinct cases

$$
\begin{cases}
1)\ pos(\overline{F}) \setminus Last(\overline{F}) \implies x \in pos(\overline{F}) & \to (a) \\
2)\ Last(\overline{F}) \implies (\ x \in pos(\overline{F})) \wedge (x \in Last(\overline{F})) & \to (a) \cup (b) \\
3)\ pos(\overline{G}) \implies x \in pos(\overline{G}) & \to (c)
\end{cases}
$$

We have,

$$Follow(\overline{F}\cdot\overline{G},x) = \begin{cases} Follow(\overline{F},x) & \textbf{if } x \in pos(\overline{F}) \setminus Last(\overline{F}) \\ Follow(\overline{F},x) \cup First(\overline{G}) & \textbf{if } x \in Last(\overline{F}) \\ Follow(\overline{G},x) & \textbf{if } x \in pos(\overline{G}) \end{cases}$$

$$Follow(\overline{F}\cdot\overline{G},x) = Follow'(\overline{F}\cdot\overline{G},x) \qquad [\ IH\ ]$$

Case (6)

$$\begin{aligned} Follow(\overline{F}^*,x) &= \left\{ y \mid \exists v, \exists w, vxyw \in L(F^*) \right\} && [\ prop.\ Follow\ ] \\ Follow(\overline{F}^*,x) &= \left\{ y \mid \exists v, \exists w, vxyw \in \bigcup_{i=0}^{\infty} L(F^i) \right\} && [\ def.\ L_{RE}\ ] \\ Follow(\overline{F}^*,x) &= \bigcup_{i=0}^{\infty} \left\{ y \mid \exists v, \exists w, vxyw \in L(F^i) \right\} \\ Follow(\overline{F}^*,x) &= \bigcup_{i=0}^{\infty} Follow(\overline{F}^i,x) && [\ prop.\ Follow\ ] \end{aligned}$$

prop 40.3 : Let $P(i)$ the property :

$$Follow(\overline{F}^i,x) = \begin{cases} Follow(\overline{F},x) & \textbf{if } x \in pos(\overline{F}) \setminus Last(\overline{F}) \\ Follow(\overline{F},x) \cup First(\overline{F}) & \textbf{if } x \in Last(\overline{F}) \\ Follow(\overline{F},x) & \textbf{if } x \in pos(\overline{F}) \end{cases}$$

Proof We prove that $\forall i \geq 2, P(i)$ holds.

◦ $i = 2$

$$\begin{aligned} & Follow(\overline{F}^2,x) \\ = {} & Follow(\overline{F}\cdot\overline{F},x) \\ = {} & \left\{ \begin{array}{ll} Follow(\overline{F},x) & \textbf{if } x \in pos(\overline{F}) \setminus Last(\overline{F}) \\ Follow(\overline{F},x) \cup First(\overline{F}) & \textbf{if } x \in Last(\overline{F}) \\ Follow(\overline{F},x) & \textbf{if } x \in pos(\overline{F}) \end{array} \right\} [\ \text{prf. (5)}\ ] \end{aligned}$$

- $\forall i \geq 3$

$$
\begin{aligned}
& Follow(\overline{F}^{i}, x) \\
= \; & Follow(\overline{F}^{i-1} \cdot \overline{F}, x) \\
= \; & \left\{ \begin{array}{ll} Follow(\overline{F}^{i-1}, x) & \textbf{if } x \in pos(\overline{F}^{i-1}) \setminus Last(\overline{F}^{i-1}) \\ Follow(\overline{F}^{i-1}, x) \cup First(\overline{F}) & \textbf{if } x \in Last(\overline{F}^{i-1}) \\ Follow(\overline{F}, x) & \textbf{if } x \in pos(\overline{F}) \end{array} \right\} [\text{ prf. (5) }]
\end{aligned}
$$

It is known that

$$
\begin{aligned}
Last(\overline{F}^{i-1}) &= Last(\overline{F}) && [\text{ } \textit{prop.} \text{ Last of iteration }] \\
pos(\overline{F}^{i-1}) &= pos(\overline{F}) && [\text{ Iteration does not change symbols }]
\end{aligned}
$$

Thus,

$$
\begin{aligned}
& Follow(\overline{F}^{i}, x) \\
= \; & \left\{ \begin{array}{ll} Follow(\overline{F}^{i-1}, x) & \textbf{if } x \in pos(\overline{F}) \setminus Last(\overline{F}) \\ Follow(\overline{F}^{i-1}, x) \cup First(\overline{F}) & \textbf{if } x \in Last(\overline{F}) \\ Follow(\overline{F}, x) & \textbf{if } x \in pos(\overline{F}) \end{array} \right. \\
= \; & \left\{ \begin{array}{ll} Follow(\overline{F}^{i}, x) & \textbf{if } x \in pos(\overline{F}) \setminus Last(\overline{F}) \\ Follow(\overline{F}^{i}, x) \cup First(\overline{F}) & \textbf{if } x \in Last(\overline{F}) \\ Follow(\overline{F}, x) & \textbf{if } x \in pos(\overline{F}) \end{array} \right\} \quad [\text{ } \textit{IH} \text{ }]
\end{aligned}
$$

□

Consequently, $\forall i \geq 2, \forall j \geq 2, Follow(\overline{F}^{i}) = Follow(\overline{F}^{j})$

$$
\begin{aligned}
\text{So, } \bigcup_{i=2}^{\infty} Follow(\overline{F}^{i}, x) &= Follow(\overline{F}^{2}, x) \\
&= \left\{ \begin{array}{ll} Follow(\overline{F}, x) & \textbf{if } x \in pos(\overline{F}) \setminus Last(\overline{F}) \\ Follow(\overline{F}, x) \cup First(\overline{F}) & \textbf{if } x \in Last(\overline{F}) \\ Follow(\overline{F}, x) & \textbf{if } x \in pos(\overline{F}) \end{array} \right.
\end{aligned}
$$

Also, it was shown at the beginning of proof (6), that

$$Follow(\overline{F}^*, x) = \bigcup_{i=0}^{\infty} Follow(\overline{F}^i, x)$$

Is deduced,

$$
\begin{aligned}
& Follow(\overline{F}^*, x) \\
= & \bigcup_{i=0}^{\infty} Follow(\overline{F}^i, x) \\
= & \emptyset \cup Follow(\overline{F}, x) \cup \bigcup_{i=2}^{\infty} Follow(\overline{F}^i, x) \\
= & \emptyset \cup Follow(\overline{F}, x) \cup \begin{cases} Follow(\overline{F}, x) & \textbf{if } x \in pos(\overline{F}) \setminus Last(\overline{F}) \\ Follow(\overline{F}, x) \cup First(\overline{F}) & \textbf{if } x \in Last(\overline{F}) \\ Follow(\overline{F}, x) & \textbf{if } x \in pos(\overline{F}) \end{cases} \\
= & \begin{cases} \emptyset \cup Follow(\overline{F}, x) \cup Follow(\overline{F}, x) & \textbf{if } x \in pos(\overline{F}) \setminus Last(\overline{F}) \\ \emptyset \cup Follow(\overline{F}, x) \cup Follow(\overline{F}, x) \cup First(\overline{F}) & \textbf{if } x \in Last(\overline{F}) \\ \emptyset \cup Follow(\overline{F}, x) \cup Follow(\overline{F}, x) & \textbf{if } x \in pos(\overline{F}) \end{cases} \\
= & \begin{cases} Follow(\overline{F}, x) & \textbf{if } x \in pos(\overline{F}) \setminus Last(\overline{F}) \quad (a) \\ Follow(\overline{F}, x) \cup First(\overline{F}) & \textbf{if } x \in Last(\overline{F}) \quad (b) \\ Follow(\overline{F}, x) & \textbf{if } x \in pos(\overline{F}) \quad (c) \end{cases}
\end{aligned}
$$

[As in proof of (5), last case (c) covers the other, so it is added to them]

$$
\begin{aligned}
= & \begin{cases} Follow(\overline{F}, x) \cup Follow(\overline{F}, x) \textbf{ if } x \in pos(\overline{F}) \setminus Last(\overline{F}) & [\ (a) \subset (c)\ ] \\ Follow(\overline{F}, x) \cup Follow(\overline{F}, x) \cup First(\overline{F}) \textbf{ if } x \in Last(\overline{F}) & [\ (b) \subset (c)\ ] \end{cases} \\
= & \begin{cases} Follow(\overline{F}, x) & \textbf{if } x \in pos(\overline{F}) \setminus Last(\overline{F}) \\ Follow(\overline{F}, x) \cup First(\overline{F}) & \textbf{if } x \in Last(\overline{F}) \end{cases} \\
= & Follow'(\overline{F}^*, x) \qquad [\ IH\ ]
\end{aligned}
$$

□

A.2 Language preservation of Glushkov

Before the start of the language preservation of Glushkov's algorithm, some lemmas are necessary.

Lemma 2 (Disjoint pos). $\forall F \in \Sigma re, \forall G \in \Sigma re$

$$pos\big(0 \bullet posex(F)\big) \cap pos\big(1 \bullet posex(G)\big) = \emptyset$$

This lemma will be useful during computation of automaton in different induction cases (in order to separate set of transitions)

Proof

def 2.1 $\diamond : \{0,1\} \to \mathcal{P}((\mathbb{2}^* \times \Sigma)) \to \mathcal{P}((\mathbb{2}^* \times \Sigma))$

$$b \diamond P = \Big\{(b \cdot b', a) \mid (b', a) \in P\Big\}$$

We define this function to take out o (1) from "$pos\big(0 \bullet posex(F)\big)$" (respectively "$pos\big(1 \bullet posex(G)\big)$").

- prove $pos(b \bullet \overline{E}) = b \diamond pos(\overline{E})$

  by induction on $\overline{E}$

  case $\overline{E} = \overline{F} \cdot \overline{G}$

$$\begin{aligned}
& pos(b \bullet (\overline{F} \cdot \overline{G})) \\
&= pos\big((b \bullet \overline{F}) \cdot (b \bullet \overline{G})\big) && [\ def.\ \bullet\ ] \\
&= pos(b \bullet \overline{F}) \cup pos(b \bullet \overline{G}) && [\ def.\ pos\ ] \\
&= (b \diamond pos(\overline{F})) \cup (b \diamond pos(\overline{G})) && [\ IH\ ] \\
&= \Big\{(b \cdot b', a) \mid (b', a) \in pos(\overline{F})\Big\} \cup \Big\{(b \cdot b', a) \mid (b', a) \in pos(\overline{G})\Big\} && [\ def.\ \diamond\ ] \\
&= \Big\{(b \cdot b', a) \mid (b', a) \in (pos(\overline{F}) \cup pos(\overline{G}))\Big\} \\
&= b \diamond \big(pos(\overline{F}) \cup pos(\overline{G})\big) && [\ def.\ \diamond\ ] \\
&= b \diamond \big(pos(\overline{F} \cdot \overline{G})\big) && [\ def.\ pos\ ]
\end{aligned}$$

case $\overline{E} = \overline{F} + \overline{G}$

Same as case $(\overline{E} = \overline{F} \cdot \overline{G})$

case $\overline{E} = \overline{F}^*$

$$
\begin{aligned}
& pos(b \bullet (\overline{F}^*)) \\
&= pos\big((b \bullet \overline{F})^*\big) && [\ def.\ \bullet\ ] \\
&= pos\big(b \bullet \overline{F}\big) && [\ def.\ pos\ ] \\
&= b \diamond pos(\overline{F}) && [\ IH\ ] \\
&= b \diamond pos(\overline{F}^*) && [\ def.\ pos\ ]
\end{aligned}
$$

case $\overline{E} = (b', a)$

$$
\begin{aligned}
& pos(b \bullet (b', a)) \\
&= pos(b \cdot b', a) && [\ def.\ \bullet\ ] \\
&= \{(b \cdot b', a)\} && [\ def.\ pos\ ]
\end{aligned}
\qquad\Big|\qquad
\begin{aligned}
& b \diamond pos\big((b', a)\big) \\
&= b \diamond \{(b', a)\} && [\ def.\ pos\ ] \\
&= \{(b \cdot b', a)\} && [\ def.\ \diamond\ ]
\end{aligned}
$$

$$\implies pos(b \bullet (b', a)) = b \diamond pos\big((b', a)\big)$$

case $\overline{E} = \epsilon$

$$
\begin{aligned}
& pos(b \bullet \epsilon) \\
&= pos(\epsilon) && [\ def.\ \bullet\ ] \\
&= \{\} && [\ def.\ pos\ ]
\end{aligned}
\qquad\Big|\qquad
\begin{aligned}
& b \diamond pos\big(\epsilon\big) \\
&= b \diamond \{\} && [\ def.\ pos\ ] \\
&= \{\} && [\ def.\ \diamond\ ]
\end{aligned}
$$

$$\implies pos(b \bullet \epsilon) = b \diamond pos\big(\epsilon\big)$$

case $\overline{E} = \emptyset$

Same as case $\overline{E} = \epsilon$.

- prove $pos\big(0 \bullet posex(F)\big) \cap pos\big(1 \bullet posex(G)\big) = \emptyset$

$$
\begin{aligned}
& pos\big(0 \bullet posex(F)\big) \\
&= 0 \diamond pos\big(posex(F)\big) && [\ pos(b \bullet \overline{E}) = b \diamond pos(\overline{E})\ ] \\
&= \Big\{(0 \cdot b', a) \mid (b', a) \in pos\big(posex(F)\big)\Big\} && [\ def.\ \diamond\ ]
\end{aligned}
$$

$$
\begin{aligned}
& pos\big(1 \bullet posex(G)\big) \\
&= 1 \diamond pos\big(posex(G)\big) && \big[\ pos(b \bullet \overline{E}) = b \diamond pos(\overline{E})\ \big] \\
&= \Big\{(1 \cdot b', a) \mid (b', a) \in pos\big(posex(G)\big)\Big\} && \big[\ def.\ \diamond\ \big]
\end{aligned}
$$

$\forall (b', a) \in (\mathbb{2}^* \times \Sigma)$,
$(0 \cdot b', a) \neq (1 \cdot b', a) \implies pos\big(0 \bullet posex(F)\big) \cap pos\big(1 \bullet posex(G)\big) = \emptyset$

□

Lemma 3 ($posex^{-1}$). $\forall E \in \Sigma\, re,\ \forall n \in \mathbb{N},\ \forall u_0, \ldots, u_{n-1} \in \Sigma^n$,

$$
u_0 \ldots u_{n-1} \in L(E) \iff \exists x_0, \ldots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \begin{cases} snd^*(x_0 \ldots x_{n-1}) = u_0 \ldots u_{n-1} \\ \wedge \quad x_0 \ldots x_{n-1} \in L(posex(E)) \end{cases}
$$

Remark 3.1 ($\epsilon \in L(posex^{-1})$).

$$
\epsilon \in L(posex(E)) \iff \epsilon \in L(E)
$$

Proof By induction on E

- $E = \emptyset$

1. Implication : $u_0 \ldots u_{n-1} \in L(E) \implies \begin{cases} \exists x_0, \ldots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ \begin{pmatrix} snd^*(x_0 \ldots x_{n-1}) = u_0 \ldots u_{n-1} \\ \wedge \quad x_0 \ldots x_{n-1} \in L(posex(E)) \end{pmatrix} \end{cases}$

   $\nexists u_0 \ldots u_{n-1} \in L(\emptyset)$

2. Implication : $u_0 \ldots u_{n-1} \in L(E) \impliedby \begin{cases} \exists x_0, \ldots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ \begin{pmatrix} snd^*(x_0 \ldots x_{n-1}) = u_0 \ldots u_{n-1} \\ \wedge \quad x_0 \ldots x_{n-1} \in L(posex(E)) \end{pmatrix} \end{cases}$

   $\nexists x_0 \ldots x_{n-1} \in L(posex(\emptyset))$

- $E = \epsilon$

1. Implication : $u_0 \ldots u_{n-1} \in L(E) \implies \begin{cases} \exists x_0, \ldots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ \begin{pmatrix} snd^*(x_0 \ldots x_{n-1}) = u_0 \ldots u_{n-1} \\ \wedge \quad x_0 \ldots x_{n-1} \in L(posex(E)) \end{pmatrix} \end{cases}$

   $L(\epsilon) = \{\epsilon\} \implies \exists!\ u_0 \ldots u_{n-1} \in L(\epsilon), u_0 \ldots u_{n-1} = \epsilon$
   $\implies \begin{cases} snd^*(x_0 \ldots x_{n-1}) = \epsilon \implies x_0 \ldots x_{n-1} = \epsilon \\ \wedge\ L(posex(\epsilon)) = \{\epsilon\} \implies x_0 \ldots x_{n-1} = \epsilon \end{cases}$

2. Implication : $u_0 \dots u_{n-1} \in L(E) \impliedby \begin{cases} \exists x_0, \dots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ \begin{pmatrix} snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \\ \wedge \quad x_0 \dots x_{n-1} \in L(posex(E)) \end{pmatrix} \end{cases}$

$$\begin{cases} L(posex(\epsilon)) = L(\epsilon) = \{\epsilon\} \implies \exists!\ x_0 \dots x_{n-1} \in L(posex(\epsilon)), x_0 \dots x_{n-1} = \epsilon \\ \wedge \quad snd^*(\epsilon) = u_0 \dots u_{n-1} \implies u_0 \dots u_{n-1} = \epsilon \end{cases}$$
$$\implies u_0 \dots u_{n-1} = \epsilon \in L(\epsilon)$$

- $E = a$

1. Implication : $u_0 \dots u_{n-1} \in L(E) \implies \begin{cases} \exists x_0, \dots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ \begin{pmatrix} snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \\ \wedge \quad x_0 \dots x_{n-1} \in L(posex(E)) \end{pmatrix} \end{cases}$

$$L(a) = \{a\} \implies \exists!\ u_0 \dots u_{n-1} \in L(a), u_0 \dots u_{n-1} = a$$
$$\implies \begin{cases} \exists x_0, \dots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ \begin{pmatrix} snd^*(x_0 \dots x_{n-1}) = a \implies \exists b \in \mathbb{2}^*, x_0 \dots x_{n-1} = (b, a) \\ \wedge\ L(posex(a)) = \{(\epsilon, a)\} \implies x_0 \dots x_{n-1} = (\epsilon, a) \end{pmatrix} \end{cases}$$
$$\implies b = \epsilon, \quad x_0 \dots x_{n-1} = (\epsilon, a)$$

2. Implication : $u_0 \dots u_{n-1} \in L(E) \impliedby \begin{cases} \exists x_0, \dots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ \begin{pmatrix} snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \\ \wedge \quad x_0 \dots x_{n-1} \in L(posex(E)) \end{pmatrix} \end{cases}$

$$\begin{cases} L(posex(a)) = L(\epsilon, a) = \{(\epsilon, a)\} \implies \exists!\ x_0 \dots x_{n-1} \in L(posex(a)), \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad x_0 \dots x_{n-1} = (\epsilon, a) \\ \wedge \quad snd^*(\epsilon, a) = u_0 \dots u_{n-1} \qquad \implies u_0 \dots u_{n-1} = a \end{cases}$$
$$\implies u_0 \dots u_{n-1} = a \in L(a)$$

- $E = F + G$

1. Implication : $u_0 \dots u_{n-1} \in L(E) \implies \begin{cases} \exists x_0, \dots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ \begin{pmatrix} snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \\ \wedge \quad x_0 \dots x_{n-1} \in L(posex(F)) \end{pmatrix} \end{cases}$

$$L(F + G) = L(F) \cup L(G)$$

▷ **if** $u_0 \dots u_{n-1} \in L(F) \implies \begin{cases} \exists x_0, \dots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ \begin{pmatrix} snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \\ \wedge \quad x_0 \dots x_{n-1} \in L(posex(F)) \end{pmatrix} \end{cases}$ [*IH*]

Let $x_i = (b_i, a_i)$ **and** $x'_i = (0 \cdot b_i, a_i)$

in $u_0 \dots u_{n-1} \in L(F) \begin{cases} \exists x_0, \dots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ \begin{pmatrix} snd^*(x'_0 \dots x'_{n-1}) = u_0 \dots u_{n-1} \\ \wedge \quad x'_0 \dots x'_{n-1} \in L(posex(F)) \end{pmatrix} \end{cases}$

▷ **if** $u_0 \dots u_{n-1} \in L(G) \implies \left\{ \begin{pmatrix} \exists x_0, \dots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \\ \wedge \quad x_0 \dots x_{n-1} \in L(posex(G)) \end{pmatrix} \right.$ [*IH*]

Let $x_i = (b_i, a_i)$ **and** $x'_i = (1 \cdot b_i, a_i)$

in $u_0 \dots u_{n-1} \in L(G) \implies \left\{ \begin{pmatrix} \exists x'_0, \dots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ snd^*(x'_0 \dots x'_{n-1}) = u_0 \dots u_{n-1} \\ \wedge \quad x'_0 \dots x'_{n-1} \in L(posex(G)) \end{pmatrix} \right.$

Hence,

$$u_0 \dots u_{n-1} \in \big(L(F) \cup L(G)\big)$$

$$\implies \left\{ \begin{pmatrix} \exists x'_0, \dots, x'_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ snd^*(x'_0 \dots x'_{n-1}) = u_0 \dots u_{n-1} \\ \wedge \quad x'_0 \dots x'_{n-1} \in \big(L(0 \bullet posex(F)) \cup L(1 \bullet posex(G))\big) \end{pmatrix} \right.$$

$$\iff \left(\begin{array}{l} u_0 \dots u_{n-1} \in L(F+G) \\ \quad \implies \left\{ \begin{pmatrix} \exists x'_0, \dots, x'_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ snd^*(x'_0 \dots x'_{n-1}) = u_0 \dots u_{n-1} \\ \wedge \quad x'_0 \dots x'_{n-1} \in L\big(0 \bullet posex(F) + 1 \bullet posex(G)\big) \end{pmatrix} \right. \end{array} \right)$$

$$\iff \left(\begin{array}{l} u_0 \dots u_{n-1} \in L(F+G) \\ \quad \implies \left\{ \begin{pmatrix} \exists x'_0, \dots, x'_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ snd^*(x'_0 \dots x'_{n-1}) = u_0 \dots u_{n-1} \\ \wedge \quad x'_0 \dots x'_{n-1} \in L\big(posex(F+G)\big) \qquad [\, def.\ posex \,] \end{pmatrix} \right. \end{array} \right)$$

2. Implication : $u_0 \dots u_{n-1} \in L(E) \impliedby \left\{ \begin{pmatrix} \exists x_0, \dots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \\ \wedge \quad x_0 \dots x_{n-1} \in L(posex(F)) \end{pmatrix} \right.$

$$\begin{aligned} L(posex(F+G)) &= L\big(\big(0 \bullet posex(F)\big) + \big(1 \bullet posex(G)\big)\big) \\ &= L\big(0 \bullet posex(F)\big) \cup L\big(1 \bullet posex(G)\big) \end{aligned}$$

▷ **if** $x_0 \dots x_{n-1} \in L\big(0 \bullet posex(F)\big) \wedge snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1}$

$$\Longrightarrow \begin{cases} (0\cdot b_0,u_0)\dots(0\cdot b_{n-1},u_{n-1}) \in L\big(0\bullet posex(F)\big) \\ \wedge \quad snd^*\big((0\cdot b_0,u_0)\dots(0\cdot b_{n-1},u_{n-1})\big) = u_0\dots u_{n-1} \end{cases}$$

$$\Longrightarrow \begin{cases} (b_0,u_0)\dots(b_{n-1},u_{n-1}) \in L\big(posex(F)\big) \\ \wedge \quad snd^*\big((b_0,u_0)\dots(b_{n-1},u_{n-1})\big) = u_0\dots u_{n-1} \end{cases}$$

$$\Longrightarrow u_0\dots u_{n-1} \in L(F) \qquad [\ IH\ ]$$

▷ **if** $x_0\dots x_{n-1} \in L\big(1\bullet posex(G)\big) \wedge snd^*(x_0\dots x_{n-1}) = u_0\dots u_{n-1}$

$$\Longrightarrow \begin{cases} (1\cdot b_0,u_0)\dots(1\cdot b_{n-1},u_{n-1}) \in L\big(1\bullet posex(G)\big) \\ \wedge \quad snd^*\big((1\cdot b_0,u_0)\dots(1\cdot b_{n-1},u_{n-1})\big) = u_0\dots u_{n-1} \end{cases}$$

$$\Longrightarrow \begin{cases} (b_0,u_0)\dots(b_{n-1},u_{n-1}) \in L\big(posex(G)\big) \\ \wedge \quad snd^*\big((b_0,u_0)\dots(b_{n-1},u_{n-1})\big) = u_0\dots u_{n-1} \end{cases}$$

$$\Longrightarrow u_0\dots u_{n-1} \in L(G) \qquad [\ IH\ ]$$

Hence,

$$\begin{cases} \exists x_0,\dots,x_{n-1} \in (2^*\times\Sigma)^n, \\ \left(\begin{array}{l} snd^*(x_0\dots x_{n-1}) = u_0\dots u_{n-1} \\ \wedge \quad x_0\dots x_{n-1} \in \big(L(0\bullet posex(F)) \cup L(1\bullet posex(G))\big) \end{array}\right) \end{cases}$$
$$\Longrightarrow u_0\dots u_{n-1} \in \big(L(F)\cup L(G)\big)$$

$$\Longleftrightarrow \left(\begin{array}{l} \left\{\begin{array}{l} \exists x_0,\dots,x_{n-1} \in (2^*\times\Sigma)^n, \\ \left(\begin{array}{l} snd^*(x_0\dots x_{n-1}) = u_0\dots u_{n-1} \\ \wedge \quad x_0\dots x_{n-1} \in L\big(posex(F+G)\big) \qquad [\ def.\ posex\ ] \end{array}\right) \end{array}\right. \\ \\ \Longrightarrow u_0\dots u_{n-1} \in L(F+G) \end{array}\right)$$

◦ $E = F\cdot G$

1. Implication : $u_0\dots u_{n-1} \in L(E) \implies \begin{cases} \exists x_0,\dots,x_{n-1} \in (2^*\times\Sigma)^n, \\ \left(\begin{array}{l} snd^*(x_0\dots x_{n-1}) = u_0\dots u_{n-1} \\ \wedge \quad x_0\dots x_{n-1} \in L(posex(E)) \end{array}\right) \end{cases}$

$L(F\cdot G) = L(F)\cdot L(G)$

$u_0\dots u_{n-1} \in L(F)\cdot L(G) \implies \exists m, 0\le m<n, \begin{cases} u_0\dots u_{m-1} \in L(F) \\ u_m\dots u_{n-1} \in L(G) \end{cases}$

$$\triangleright\ u_0 \dots u_{m-1} \in L(F) \implies \left\{ \begin{array}{l} \exists x_0, \dots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ \left(\begin{array}{ll} snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \\ \wedge \quad x_0 \dots x_{n-1} \in L(posex(F)) \end{array} \right) \end{array} \right. \qquad [\ IH\ ]$$

Let $x_i = (b_i, a_i)$ **and** $x'_i = (0 \cdot b_i, a_i)$

$$\textbf{in } u_0 \dots u_{m-1} \in L(F) \implies \left\{ \begin{array}{l} \exists x'_0, \dots, x'_{m-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ \left(\begin{array}{ll} snd^*(x'_0 \dots x'_{m-1}) = u_0 \dots u_{n-1} \\ \wedge \quad x'_0 \dots x'_{m-1} \in L(posex(F)) \end{array} \right) \end{array} \right.$$

$\triangleright$ **if** $u_m \dots u_{n-1} \in L(G) \implies x_m \dots x_{n-1} \in L(posex(G))$ $\quad [\ IH\ ]$

Let $x_i = (b_i, a_i)$ **and** $x'_i = (1 \cdot b_i, a_i)$

$$\textbf{in } u_m \dots u_{n-1} \in L(G) \implies \left\{ \begin{array}{l} \exists x'_m, \dots, x'_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ \left(\begin{array}{ll} snd^*(x'_m \dots x'_{n-1}) = u_m \dots u_{n-1} \\ \wedge \quad x'_m \dots x'_{n-1} \in L(posex(G)) \end{array} \right) \end{array} \right.$$

Hence,

$$u_0 \dots u_{m-1} u_m \dots u_{n-1} \in \big(L(F) \cdot L(G)\big)$$

$$\implies \left\{ \begin{array}{l} \exists x'_0, \dots, x'_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ \left(\begin{array}{ll} snd^*(x'_0 \dots x'_{n-1}) = u_0 \dots u_{n-1} \\ \wedge \quad x'_0 \dots x'_{n-1} \in \big(L(0 \bullet posex(F)) \cdot L(1 \bullet posex(G))\big) \end{array} \right) \end{array} \right.$$

$$\iff \left(\begin{array}{l} u_0 \dots u_{m-1} u_m \dots u_{n-1} \in L(F \cdot G) \\ \implies \left\{ \begin{array}{l} \exists x'_0, \dots, x'_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ \left(\begin{array}{ll} snd^*(x'_0 \dots x'_{n-1}) = u_0 \dots u_{n-1} \\ \wedge \quad x'_0 \dots x'_{n-1} \in L\big((0 \bullet posex(F)) \cdot (1 \bullet posex(G))\big) \end{array} \right) \end{array} \right. \end{array} \right)$$

$$\iff \left(\begin{array}{l} u_0 \dots u_{n-1} \in L(F \cdot G) \\ \implies \left\{ \begin{array}{l} \exists x'_0, \dots, x'_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ \left(\begin{array}{ll} snd^*(x'_0 \dots x'_{n-1}) = u_0 \dots u_{n-1} \\ \wedge \quad x'_0 \dots x'_{n-1} \in L\big(posex(F \cdot G)\big) \end{array} \right) \end{array} \right. \end{array} \right)$$

2. Implication : $u_0 \dots u_{n-1} \in L(E) \impliedby \left\{ \begin{array}{l} \exists x_0, \dots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ \left(\begin{array}{ll} snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \\ \wedge \quad x_0 \dots x_{n-1} \in L(posex(E)) \end{array} \right) \end{array} \right.$

We have

$L(posex(F \cdot G)) = L\big((0 \bullet posex(F)) \cdot (1 \bullet posex(G))\big)$

$L(posex(F \cdot G)) = L\big((0 \bullet posex(F))\big) \cdot L\big((1 \bullet posex(G))\big)$

and

$x_0 \dots x_{n-1} \in L\big((0 \bullet posex(F))\big) \cdot L\big((1 \bullet posex(G))\big)$

$$\implies \exists m, 0 \leq m < n, \begin{cases} x_0 \dots x_{m-1} \in L\big((0 \bullet posex(F))\big) \\ x_m \dots x_{n-1} \in L\big((1 \bullet posex(G))\big) \end{cases}$$

$\triangleright\ x_0 \dots x_{m-1} \in L\big(0 \bullet posex(F)\big) \wedge snd^*(x_0 \dots x_{m-1}) = u_0 \dots u_{m-1}$

$$\implies \begin{cases} (0 \cdot b_0, u_0) \dots (0 \cdot b_{m-1}, u_{m-1}) \in L\big(0 \bullet posex(F)\big) \\ \wedge \quad snd^*\big((0 \cdot b_0, u_0) \dots (0 \cdot b_{m-1}, u_{m-1})\big) = u_0 \dots u_{m-1} \end{cases}$$

$$\implies \begin{cases} (b_0, u_0) \dots (b_{m-1}, u_{m-1}) \in L\big(posex(F)\big) \\ \wedge \quad snd^*\big((b_0, u_0) \dots (b_{m-1}, u_{m-1})\big) = u_0 \dots u_{m-1} \end{cases}$$

$$\implies u_0 \dots u_{m-1} \in L(F) \qquad [\ IH\ ]$$

$\triangleright\ x_0 \dots x_{n-1} \in L\big(1 \bullet posex(G)\big) \wedge snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1}$

$$\implies \begin{cases} (1 \cdot b_m, u_m) \dots (1 \cdot b_{n-1}, u_{n-1}) \in L\big(1 \bullet posex(G)\big) \\ \wedge \quad snd^*\big((1 \cdot b_m, u_m) \dots (1 \cdot b_{n-1}, u_{n-1})\big) = u_m \dots u_{n-1} \end{cases}$$

$$\implies \begin{cases} (b_m, u_m) \dots (b_{n-1}, u_{n-1}) \in L\big(posex(G)\big) \\ \wedge \quad snd^*\big((b_m, u_m) \dots (b_{n-1}, u_{n-1})\big) = u_m \dots u_{n-1} \end{cases}$$

$$\implies u_m \dots u_{n-1} \in L(G) \qquad [\ IH\ ]$$

Hence,

$$\exists x_0 \dots x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \begin{cases} snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \\ \wedge \quad x_0 \dots x_{n-1} \in \big(L(0 \bullet posex(F)) \cdot L(1 \bullet posex(G))\big) \end{cases}$$

$$\implies u_0 \dots u_m u_{m-1} \dots u_{n-1} \in \big(L(F) \cdot L(G)\big)$$

$$\iff \left(\begin{array}{l} \exists x_0, \dots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ \left(\begin{array}{ll} snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \\ \wedge \quad x_0 \dots x_{n-1} \in L(posex(F \cdot G)) \end{array} \right) \end{array} \right\} \implies u_0 \dots u_{n-1} \in L(F \cdot G) \right)$$

$\circ\ E = F^*$

1. Implication : $u_0 \dots u_{n-1} \in L(E) \impliedby \begin{cases} \exists x_0, \dots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ \left(\begin{array}{ll} snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \\ \wedge \quad x_0 \dots x_{n-1} \in L(posex(E)) \end{array} \right) \end{cases}$

   $L(F^*) = \bigcup_{i=0}^{\infty=k} L(F^i)$

   By induction on k.

$\triangleright\ k = 0 \implies L(F^0) = \{\epsilon\} \qquad [\ \text{see } E = \epsilon \text{ case}\ ]$

$\triangleright\ k > 0 \implies \bigcup_{i=0}^{k} L(F^i) = \Big(\bigcup_{i=0}^{k-1} L(F^i)\Big) \cup L(F^k)$

- **if** $u_0 \dots u_{n-1} \in \bigcup_{i=0}^{k-1} L(F^i)$ **then** *IH*

$$u_0 \dots u_{n-1} \in \bigcup_{i=0}^{k-1} L(F^i) \implies \begin{cases} \exists x_0, \dots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ \left(\begin{array}{ll} snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \\ \wedge \quad x_0 \dots x_{n-1} \in \bigcup_{i=0}^{k-1} L\big((0 \bullet posex(F))^i\big) \end{array} \right) \end{cases}$$

- **if** $u_0 \dots u_{n-1} \in L(F^k)$

$\implies \forall j \in \{1 \dots k\}, \exists m^j, m^1 = 0 \leq \dots \leq m^{j-1} \leq m^j \leq m^{j+1} \leq \dots \leq n-1,$

(m^j is the first symbol index of the j^{th} word belonging to $L(F)$)

$$\begin{cases} u_{m^0} \dots u_{m^1-1} \in L(F) \implies \exists x_{m^0}, \dots, x_{m^1-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ \qquad\qquad \begin{cases} snd^*(x_{m^0} \dots x_{m^1-1}) = u_{m^0} \dots u_{m^1-1} \\ \wedge \quad x_{m^0} \dots x_{m^1-1} \in L(posex(F)) \end{cases} \quad [\, IH \,] \\ \qquad \vdots \\ u_{m^{j-1}} \dots u_{m^j-1} \in L(F) \implies \exists x_{m^{j-1}}, \dots, x_{m^j-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ \qquad\qquad \begin{cases} snd^*(x_{m^{j-1}} \dots x_{m^j-1}) = u_{m^{j-1}} \dots u_{m^j-1} \\ \wedge \quad x_{m^{j-1}} \dots x_{m^j-1} \in L(posex(F)) \end{cases} \quad [\, IH \,] \\ \qquad \vdots \\ u_{m^k} \dots u_{n-1} \in L(F) \implies \exists x_{m^k}, \dots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ \qquad\qquad \begin{cases} snd^*(x_{m^k} \dots x_{n-1} = u_{m^k} \dots u_{n-1} \\ \wedge \quad x_{m^k} \dots x_{n-1} \in L(posex(F)) \end{cases} \quad [\, IH \,] \end{cases}$$

Let $x_i = (b_i, a_i)$ **and** $x'_i = (0 \cdot b_i, a_i)$ **in**

$$\begin{cases} u_{m^0} \dots u_{m^1-1} \in L(F) \implies \exists x'_{m^0}, \dots, x'_{m^1-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ \qquad\qquad \begin{cases} snd^*(x'_{m^0} \dots x'_{m^1-1}) = u_{m^0} \dots u_{m^1-1} \\ \wedge \quad x'_{m^0} \dots x'_{m^1-1} \in L(0 \bullet posex(F)) \end{cases} \quad [\, IH \,] \\ \qquad \vdots \\ u_{m^{j-1}} \dots u_{m^j-1} \in L(F) \implies \exists x'_{m^{j-1}}, \dots, x'_{m^j-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ \qquad\qquad \begin{cases} snd^*(x'_{m^{j-1}} \dots x'_{m^j-1}) = u_{m^{j-1}} \dots u_{m^j-1} \\ \wedge \quad x'_{m^{j-1}} \dots x'_{m^j-1} \in L(0 \bullet posex(F)) \end{cases} \quad [\, IH \,] \\ \qquad \vdots \\ u_{m^k} \dots u_{n-1} \in L(F) \implies \exists x'_0, \dots, x'_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ \qquad\qquad \begin{cases} snd^*(x'_{m^k} \dots x'_{n-1} = u_0 \dots u_{n-1} \\ \wedge \quad x'_{m^k} \dots x'_{n-1} \in L(0 \bullet posex(F)) \end{cases} \quad [\, IH \,] \end{cases}$$

$$\iff \left(u_0 \dots u_{n-1} \in L(F^k) \implies \exists x_0, \dots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \begin{cases} snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \\ \wedge\, x_0 \dots x_{n-1} \in L\big((0 \bullet posex(F))^k\big) \end{cases} \right)$$

Hence,

$$u_0 \dots u_{n-1} \in \left(\bigcup_{i=0}^{k-1} L(F^i)\right) \cup L(F^k)$$

$$\implies \exists x_0, \dots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \begin{cases} snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \\ \wedge\ x_0 \dots x_{n-1} \in \left(\bigcup_{i=0}^{k-1} L\big((0 \bullet posex(F))^i\big)\right) \cup L\big((0 \bullet posex(F))^k\big) \end{cases}$$

$$\iff \left(\begin{array}{l} u_0 \dots u_{n-1} \in \bigcup_{i=0}^{k} L(F^i) \\ \quad \implies \exists x_0, \dots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \begin{cases} snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \\ \wedge\ x_0 \dots x_{n-1} \in \bigcup_{i=0}^{k} L\big((0 \bullet posex(F))^i\big) \end{cases} \end{array} \right)$$

has been proved $\forall k \in \mathbb{N}$, so

$$\iff \left(\begin{array}{l} u_0 \dots u_{n-1} \in \bigcup_{i=0}^{\infty} L(F^i) \\ \quad \implies \exists x_0, \dots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \begin{cases} snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \\ \wedge\ x_0 \dots x_{n-1} \in \bigcup_{i=0}^{\infty} L\big((0 \bullet posex(F))^i\big) \end{cases} \end{array} \right)$$

$$\iff \left(\begin{array}{l} u_0 \dots u_{n-1} \in L(F^*) \\ \quad \implies \exists x_0, \dots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \begin{cases} snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \\ \wedge\ x_0 \dots x_{n-1} \in L\big((0 \bullet posex(F))^*\big) \end{cases} \end{array} \right) \qquad [\, def.\ L_{RE} \,]$$

$$\iff \left(\begin{array}{l} u_0 \dots u_{n-1} \in L(F^*) \\ \quad \implies \exists x_0, \dots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \begin{cases} snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \\ \wedge\ x_0 \dots x_{n-1} \in L(posex(F^*)) \end{cases} \end{array} \right) \qquad [\, def.\ posex \,]$$

2. Implication $u_0 \dots u_{n-1} \in L(E) \impliedby \begin{cases} \exists x_0, \dots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \\ \left(\begin{array}{ll} snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \\ \wedge & x_0 \dots x_{n-1} \in L(posex(E)) \end{array} \right) \end{cases}$

$$L(posex(F^*)) = L\big((0 \bullet posex(F))^*\big) = \bigcup_{i=0}^{\infty = k} L\big((0 \bullet posex(F))^i\big)$$

By induction on k.

▷ $k = 0 \implies \bigcup_{i=0}^{0} L\big((0 \bullet posex(F))^i\big) = L\big((0 \bullet posex(F))^0\big) = \{\epsilon\}$ [see $E = \epsilon$ case]

▷ $k > 0 \implies \bigcup_{i=0}^{k} L\big((0 \bullet posex(F))^i\big) = \left(\bigcup_{i=0}^{k-1} L\big((0 \bullet posex(F))^i\big)\right) \cup L\big((0 \bullet posex(F))^k\big)\Big)$

► **if** $\exists x_0, \dots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \begin{cases} snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \\ \wedge\ x_0 \dots x_{n-1} \in \bigcup_{i=0}^{k-1} L\big((0 \bullet posex(F))^i\big) \end{cases}$ **then** *IH*

$$\exists x_0, \dots, x_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \left\{ \begin{array}{l} snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \\ \wedge\ x_0 \dots x_{n-1} \in \bigcup_{i=0}^{k-1} L\big((0 \bullet posex(F))^i\big) \end{array} \right\} \implies u_0 \dots u_{n-1} \in \bigcup_{i=0}^{k-1} L(F^i)$$

▶ **if** $\exists x_0,\dots,x_{n-1}\in(2^*\times\Sigma)^n, \begin{cases} snd^*(x_0\dots x_{n-1}) = u_0\dots u_{n-1} \\ \wedge\ x_0\dots x_{n-1}\in L\big((0\bullet posex(F))^k\big)\end{cases}$

$\implies \forall j\in\{1\dots k\}, \exists m^j, m^1 = 0 \le \dots \le m^{j-1} \le m^j \le m^{j+1} \le \dots \le n-1,$

$$\begin{cases} \exists x_{m^0},\dots,x_{m^1-1}\in(2^*\times\Sigma)^n, & \begin{cases} snd^*(x_{m^0}\dots x_{m^1-1}) = u_{m^0}\dots u_{m^1-1} \\ \wedge\quad x_{m^0}\dots x_{m^1-1}\in L(0\bullet posex(F))\end{cases} \\ \vdots & \\ \exists x_{m^{j-1}},\dots,x_{m^j-1}\in(2^*\times\Sigma)^n, & \begin{cases} snd^*(x_{m^{j-1}}\dots x_{m^1-1}) = u_{m^{j-1}}\dots u_{m^j-1} \\ \wedge\quad x_{m^{j-1}}\dots x_{m^j-1}\in L(0\bullet posex(F))\end{cases} \\ \vdots & \\ \exists x_{m^k},\dots,x_{n-1}\in(2^*\times\Sigma)^n, & \begin{cases} snd^*(x_{m^k}\dots x_{n-1}) = u_{m^k}\dots u_{n-1} \\ \wedge\quad x_{m^k}\dots x_{n-1}\in L(0\bullet posex(F))\end{cases}\end{cases}$$

Let $x_i = (0\cdot b_i, a_i)$ $\quad$ [because $x_0\dots x_{n-1}\in L\big((0\bullet posex(F))^k\big)$]

and $x'_i = (b_i, a_i)$ **in**

$$\begin{cases} \left(\begin{array}{l}\exists x'_{m^0},\dots,x'_{m^1-1}\in(2^*\times\Sigma)^n, \\ \quad\begin{cases} snd^*(x'_{m^0}\dots x'_{m^1-1}) = u_{m^0}\dots u_{m^1-1} \\ \wedge\quad x'_{m^0}\dots x'_{m^1-1}\in L(posex(F))\end{cases}\end{array}\right) & \implies u_{m^0}\dots u_{m^1-1}\in L(F) \quad [\,IH\,] \\ \vdots & \\ \left(\begin{array}{l}\exists x'_{m^{j-1}},\dots,x'_{m^j-1}\in(2^*\times\Sigma)^n, \\ \quad\begin{cases} snd^*(x'_{m^{j-1}}\dots x'_{m^1-1}) = u_{m^{j-1}}\dots u_{m^j-1} \\ \wedge\quad x'_{m^{j-1}}\dots x'_{m^j-1}\in L(posex(F))\end{cases}\end{array}\right) & \implies u_{m^{j-1}}\dots u_{m^j-1}\in L(F) \quad [\,IH\,] \\ \vdots & \\ \left(\begin{array}{l}\exists x'_{m^k},\dots,x'_{n-1}\in(2^*\times\Sigma)^n, \\ \quad\begin{cases} snd^*(x'_{m^k}\dots x'_{n-1}) = u_{m^k}\dots u_{n-1} \\ \wedge\quad x'_{m^k}\dots x'_{n-1}\in L(posex(F))\end{cases}\end{array}\right) & \implies u_{m^k}\dots u_{n-1}\in L(F) \quad [\,IH\,]\end{cases}$$

$$\iff \left(u_0\dots u_{n-1}\in L(F^k) \implies \left\{\begin{array}{l}\exists x'_0,\dots,x'_{n-1}\in(2^*\times\Sigma)^n, \\ \quad\left(\begin{array}{l} snd^*(x'_0\dots x'_{n-1}) = u_0\dots u_{n-1} \\ \wedge\ x'_0\dots x'_{n-1}\in L\big((0\bullet posex(F))^k\big)\end{array}\right)\end{array}\right.\right)$$

Hence,

$$\left(\begin{array}{l}\exists x'_0,\dots,x'_{n-1}\in(2^*\times\Sigma)^n, \begin{cases} snd^*(x'_0\dots x'_{n-1}) = u_0\dots u_{n-1} \\ \wedge\ x'_0\dots x'_{n-1}\in\left(\bigcup_{i=0}^{k-1}L\big((0\bullet posex(F))^i\big)\right)\cup L\big((0\bullet posex(F))^k\big)\end{cases} \\ \implies u_0\dots u_{n-1}\in\left(\bigcup_{i=0}^{k-1}L(F^i)\right)\cup L(F^k)\end{array}\right)$$

$$\iff \left(\begin{array}{l} \exists x'_0, \ldots, x'_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \left\{ \begin{array}{l} snd^*(x'_0 \ldots x'_{n-1}) = u_0 \ldots u_{n-1} \\ \wedge\ x'_0 \ldots x'_{n-1} \in \bigcup_{i=0}^{k} L\big((0 \bullet posex(F))^i\big) \end{array} \right. \\ \quad \Longrightarrow\ u_0 \ldots u_{n-1} \in \bigcup_{i=0}^{k} L(F^i) \end{array} \right)$$

It has been proved $\forall k \in \mathbb{N}$, so

$$\iff \left(\begin{array}{l} \exists x'_0, \ldots, x'_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \left\{ \begin{array}{l} snd^*(x'_0 \ldots x'_{n-1}) = u_0 \ldots u_{n-1} \\ \wedge\ x'_0 \ldots x'_{n-1} \in \bigcup_{i=0}^{\infty} L\big((0 \bullet posex(F))^i\big) \end{array} \right. \\ \quad \Longrightarrow\ u_0 \ldots u_{n-1} \in \bigcup_{i=0}^{\infty} L(F^i) \end{array} \right)$$

$$\iff \left(\begin{array}{l} \exists x'_0, \ldots, x'_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \left\{ \begin{array}{l} snd^*(x'_0 \ldots x'_{n-1}) = u_0 \ldots u_{n-1} \\ \wedge\ x'_0 \ldots x'_{n-1} \in L\big((0 \bullet posex(F))^*\big) \end{array} \right. \\ \quad \Longrightarrow\ u_0 \ldots u_{n-1} \in L(F^*) \end{array} \right) \qquad [\ def.\ L_{RE}\ ]$$

$$\iff \left(\begin{array}{l} \exists x'_0, \ldots, x'_{n-1} \in (\mathbb{2}^* \times \Sigma)^n, \left\{ \begin{array}{l} snd^*(x'_0 \ldots x'_{n-1}) = u_0 \ldots u_{n-1} \\ \wedge\ x'_0 \ldots x'_{n-1} \in L\big(posex(F^*)\big) \end{array} \right. \\ \quad \Longrightarrow\ u_0 \ldots u_{n-1} \in L(F^*) \end{array} \right) \qquad [\ def.\ posex\ ]$$

□

Theorem 1 (Glushkov preserves language). $\forall E \in \Sigma\, re$

$$L\big((glu\_autom(E) \circ glushkov \circ posex \circ snf)(E)\big) = L(E)$$

Proof

We know from [6] that $L(snf(E)) = L(E)$. It remains to prove

$$L\big((glu\_autom(E) \circ glushkov \circ posex)(E)\big) = L(E)$$

By induction on E.

In the following, *follow* will often be considered as the graph of the function, (*i.e.* a set of transitions) thereby giving us the possibility to use set operators.

Case $E = \emptyset$: $L(E) = L(\emptyset) = \emptyset$

$$\begin{aligned} A_\emptyset &= (glu\_autom(E) \circ glushkov \circ posex)(\emptyset) \\ &= (glu\_autom(E) \circ glushkov)(\emptyset) \\ &= glu\_autom\ (E)\ (first = \emptyset, last = \emptyset, null = \emptyset, follow = \emptyset) \end{aligned}$$

$$A_\emptyset = \begin{cases} Q = \{q_I\} \\ \Sigma = \emptyset \\ \delta = \emptyset \\ I = \{q_I\} \\ F = \emptyset \end{cases}$$

$A_\emptyset$ $F = \emptyset \implies L(A_\emptyset) = \emptyset$

Case $E = \epsilon$: $L(E) = L(\epsilon) = \{\epsilon\}$

$$\begin{aligned} A_\epsilon &= (glu\_autom(E) \circ glushkov \circ posex)(\epsilon) \\ &= (glu\_autom(E) \circ glushkov)(\epsilon) \\ &= glu\_autom\ (E)\ (first = \emptyset, last = \emptyset, null = \{\epsilon\}, follow = \emptyset) \end{aligned}$$

$$A_\epsilon = \begin{cases} Q = \{q_I\} \\ \Sigma = \emptyset \\ \delta = \emptyset \\ I = \{q_I\} \\ F = \{q_I\} \end{cases}$$

A_ϵ $(F = I) \wedge (\delta = \emptyset) \implies L(A_\epsilon) = \{\epsilon\}$

Case $E = a$: $L(E) = L(a) = \{a\}$

$$\begin{aligned} A_a &= (glu\_autom(E) \circ glushkov \circ posex)(a) \\ &= (glu\_autom(E) \circ glushkov)((\epsilon, a)) \\ &= glu\_autom\ (E)\ (first = \{(\epsilon, a)\}, last = \{(\epsilon, a)\}, null = \emptyset, follow = \{Follow((\epsilon, a)) = \emptyset\}) \end{aligned}$$

$$A_a = \begin{cases} Q = \{q_I\} \cup \{(\epsilon, a)\} \\ \Sigma = \{a\} \\ \delta = \{(q_I, a) \to (\epsilon, a)\} \\ I = \{q_I\} \\ F = \{(\epsilon, a)\} \end{cases}$$

A_a There exists only one transition in δ, this transition goes from the initial state to the final state and its label is 'a'. So, $L(A_a) = \{a\}$

Case $E = E_1 + E_2$:

$$
\begin{aligned}
A_+ &= (glu\_autom(E) \circ glushkov \circ posex)(E) \\
&= (glu\_autom(E) \circ glushkov \circ posex)(E_1 + E_2) \qquad [\, E = E_1 + E_2 \,] \\
&= (glu\_autom(E) \circ glushkov)\big((0 \bullet posex(E_1)) + (1 \bullet posex(E_2))\big) \qquad [\, \textit{def. posex} \,]
\end{aligned}
$$

$$
= glu\_autom\ (E) \left(\begin{array}{rl}
first = & First(0 \bullet posex(E_1)) \cup First(1 \bullet posex(E_2)) \\
last = & Last(0 \bullet posex(E_1)) \cup Last(1 \bullet posex(E_2)) \\
null = & Null(0 \bullet posex(E_1)) \cup Null(1 \bullet posex(E_2)) \\
follow = & \left(\begin{array}{l}
\big\{ Follow\big(0 \bullet posex(E_1), x\big) \mid x \in pos\big(0 \bullet posex(E_1)\big) \big\} \\
\cup \\
\big\{ Follow\big(1 \bullet posex(E_2), x\big) \mid x \in pos\big(1 \bullet posex(E_2)\big) \big\}
\end{array} \right)
\end{array} \right)
$$

$[\, \textit{def. glushkov} \,]$

$$
\left\{ \begin{array}{rl}
Q &= pos(posex(E)) \cup \{q_I\} \\
\Sigma &= \Sigma \\
\delta &= \left\{ \begin{array}{r}
\forall a \in \Sigma, \delta(q_I, a) = \big\{ x \mid x \in first, snd(x) = a \big\} \\
\forall x \in (\mathbb{2}^* \times \Sigma), \forall a \in \Sigma, \delta(x, a) = \big\{ y \mid y \in follow(x), snd(y) = a \big\}
\end{array} \right. \\
I &= \{q_I\} \\
F &= \left\{ \begin{array}{ll}
last \cup \{q_I\} & \textbf{if } null = \{\epsilon\} \\
last & \textbf{otherwise}
\end{array} \right.
\end{array} \right.
$$

[reminder of *def. glu_autom*]

$$
= \left\{
\begin{array}{l}
Q = \{q_I\} \cup pos(0 \bullet posex(E_1)) \cup pos(1 \bullet posex(E_2)) \\
\\
\Sigma = \Big\{ snd(x) \mid x \in pos\big(0 \bullet posex(E_1)\big) \Big\} \cup \Big\{ snd(x) \mid x \in pos\big(1 \bullet posex(E_2)\big) \Big\} \\
\\
\delta = \left\{
\begin{array}{l}
\forall a \in \Big\{ snd(x) \mid x \in pos\big(0 \bullet posex(E_1)\big) \Big\} \cup \Big\{ snd(x) \mid x \in pos\big(1 \bullet posex(E_2)\big) \Big\}, \\
\quad \delta(q_I, a) = \Big\{ x \mid x \in First(0 \bullet posex(E_1)) \cup First(1 \bullet posex(E_2)), snd(x) = a \Big\} \\
\\
\forall x \in pos(0 \bullet posex(E_1)) \cup pos(1 \bullet posex(E_2)), \\
\forall a \in \Big\{ snd(x') \mid x' \in pos\big(0 \bullet posex(E_1)\big) \Big\} \cup \Big\{ snd(x') \mid x' \in pos\big(1 \bullet posex(E_2)\big) \Big\}, \\
\quad \delta(x, a) = \left\{ y \;\middle|\; \begin{array}{l} y \in \Big(Follow\big(0 \bullet posex(E_1), x\big) \cup Follow\big(1 \bullet posex(E_2), x\big) \Big) \\ \wedge\ snd(y) = a \end{array} \right\}
\end{array}
\right. \\
\\
I = \{q_I\} \\
\\
F = \left\{
\begin{array}{l}
\textbf{if } Null(0 \bullet posex(E_1)) \cup Null(1 \bullet posex(E_2)) = \{\epsilon\} \\
\qquad \{q_I\} \cup Last(0 \bullet posex(E_1)) \cup Last(1 \bullet posex(E_2)) \\
\textbf{otherwise } Last(0 \bullet posex(E_1)) \cup Last(1 \bullet posex(E_2))
\end{array}
\right.
\end{array}
\right.
$$

[application of *def. glu_autom*]

$$
= \begin{cases}
Q = \{q_I\} \cup pos(0 \bullet posex(E_1)) \cup pos(1 \bullet posex(E_2)) \\
\Sigma = \Big\{snd(x) \mid x \in pos\big(0 \bullet posex(E_1)\big)\Big\} \cup \Big\{snd(x) \mid x \in pos\big(1 \bullet posex(E_2)\big)\Big\} \\
\delta = \begin{cases}
\forall a \in \Big\{snd(x) \mid x \in pos\big(0 \bullet posex(E_1)\big)\Big\}, \\
\quad \delta(q_I, a) = \Big\{x \mid x \in First(0 \bullet posex(E_1)), snd(x) = a\Big\} & [\ \delta_1\ ] \\
\forall a \in \Big\{snd(x) \mid x \in pos\big(1 \bullet posex(E_2)\big)\Big\}, \\
\quad \delta(q_I, a) = \Big\{x \mid x \in First(1 \bullet posex(E_2)), snd(x) = a\Big\} & [\ \delta_2\ ] \\
\\
\forall x \in pos(0 \bullet posex(E_1)), \\
\ \forall a \in \Big\{snd(x) \mid x \in pos\big(0 \bullet posex(E_1)\big)\Big\}, \\
\quad \delta(x, a) = \Big\{y \mid y \in Follow\big(0 \bullet posex(E_1), x\big), snd(y) = a\Big\} & [\ \delta_3\ ] \\
\\
\forall x \in pos(1 \bullet posex(E_2)), \\
\ \forall a \in \Big\{snd(x) \mid x \in pos\big(1 \bullet posex(E_2)\big)\Big\}, \\
\quad \delta(x, a) = \Big\{y \mid y \in Follow\big(1 \bullet posex(E_2), x\big), snd(y) = a\Big\} & [\ \delta_4\ ]
\end{cases} \\
I = \{q_I\} \\
F = \begin{cases}
\textbf{if } Null(0 \bullet posex(E_1)) \cup Null(1 \bullet posex(E_2)) = \{\epsilon\} \\
\quad \{q_I\} \cup Last(0 \bullet posex(E_1)) \cup Last(1 \bullet posex(E_2)) \\
\textbf{otherwise } Last(0 \bullet posex(E_1)) \cup Last(1 \bullet posex(E_2))
\end{cases}
\end{cases}
$$

[*lem.* Disjoint pos, $(\delta_1, \ldots, \delta_4)$ are names given to the subset of delta described in its line]

$L(A) = L(E) \iff \big(\forall w \in \Sigma^*, w \in L(A) \iff w \in L(E)\big)$

Two distinct cases on $|w|$ are considered

- $|w| = 0 \to w = \epsilon$

  $\epsilon \in L(A) \iff I \subset F \iff q_I \in F$

  $\iff Null(0 \bullet posex(E_1)) \cup Null(1 \bullet posex(E_2)) = \{\epsilon\}$ [*def* $F, q_I \notin last$]

  $\iff \epsilon \in \big(L(0 \bullet posex(E_1)) \cup L(1 \bullet posex(E_2))\big)$ [*prop. Null*]

  $\iff \epsilon \in L\big((0 \bullet posex(E_1)) + (1 \bullet posex(E_2))\big)$ [*def.* L_{RE}]

  $\iff \epsilon \in L(posex(E_1 + E_2))$ [*def. posex*]

  $\iff \epsilon \in L(posex(E))$ [$E = E_1 + E_2$]

  $\iff \epsilon \in L(E)$ [*rmk.* $\epsilon \in L(posex^{-1})$]

- $|w| = n, (n \geq 1) \to w = u_0 \ldots u_{n-1} \in L(A)$

$\iff \exists x, \left(x \in (\delta^*(\delta(\{q_I\}, u_0), u_1 \dots u_{n-1})\right) \wedge (x \in F)$

From A_+, two cases according to membership of the word's first symbol must be distinguished

▷ **if** $u_0 \in \left\{snd(x) \mid x \in pos\left(0 \bullet posex(E_1)\right)\right\}, w \in L(A)$

$$\iff \begin{cases} \forall i \in \{1 \dots n-1\} & \\ \quad \exists x_0 \in First(0 \bullet posex(E_1)), & [\ A_+[\delta_1], u_0 \text{ membership}\ ] \\ \quad \exists x_i \in Follow(0 \bullet posex(E_1), x_{i-1}), & [\ A_+[\delta_2], x_{i-1} \text{ membership}\ ] \\ x_{n-1} \in Last(0 \bullet posex(E_1)), & [\ A_+[F], x_{n-1} \text{ membership}, x_{n-1} \neq q_I\ ] \\ snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} & \end{cases}$$

For the following , each introduced variable $x \in pos(0 \bullet posex(E_1))$ and $w \in pos(0 \bullet posex(E_1))^*$

$$\iff \begin{cases} \forall i \in \{1 \dots n-1\} & \\ \quad \exists x_0, \exists w'_0, & x_0 w'_0 \in L(0 \bullet posex(E_1)), & [\ \textit{prop. First}\ ] \\ \exists x_i, \exists w_{i-1}, \exists w'_i, & w_{i-1} x_{i-1} x_i w'_i \in L(0 \bullet posex(E_1)), & [\ \textit{prop. Follow}\ ] \\ \quad \exists w_{n-1}, & w_{n-1} x_{n-1} \in L(0 \bullet posex(E_1)), & [\ \textit{prop. Last}\ ] \\ snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} & & \end{cases}$$

For $x_0 \dots x_{n-1}$ to be the word constrained by all definitions above, we have

$$\forall i \in \{0 \dots n-1\}, \begin{bmatrix} w_i = x_0 \dots x_{i-1} \\ w'_i = x_{i+1} \dots x_{n-1} \end{bmatrix}$$

So, we also have $\begin{bmatrix} x_0 \dots x_{n-1} \in L(0 \bullet posex(E_1)) \\ snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \end{bmatrix}$

▷ **if** $u_0 \in \left\{snd(x) \mid x \in pos\left(1 \bullet posex(E_2)\right)\right\}$

For the same symmetric reasons, we have $\begin{bmatrix} x_0 \dots x_{n-1} \in L(1 \bullet posex(E_2)) \\ snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \end{bmatrix}$

▷ **hence**, $u_0 \in \left\{snd(x) \mid x \in pos\left(0 \bullet posex(E_1)\right)\right\} \cup \left\{snd(x) \mid x \in pos\left(1 \bullet posex(E_2)\right)\right\}$
[no other possibility]

$$\iff \exists x_0 \dots x_{n-1}, \begin{cases} snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \wedge \\ \Big((x_0 \dots x_{n-1} \in L(0 \bullet posex(E_1))) \vee (x_0 \dots x_{n-1} \in L(1 \bullet posex(E_2))) \Big) \end{cases}$$

$$\iff \exists x_0 \dots x_{n-1}, \begin{cases} snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \wedge \\ x_0 \dots x_{n-1} \in \big(L(0 \bullet posex(E_1)) \cup L(1 \bullet posex(E_2)) \big) \end{cases}$$

$$\iff \exists x_0 \dots x_{n-1}, \begin{cases} snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \wedge \\ x_0 \dots x_{n-1} \in L\big((0 \bullet posex(E_1)) + (1 \bullet posex(E_2)) \big) \end{cases} \quad [\ def.\ L_{RE}\ ]$$

$$\iff \exists x_0 \dots x_{n-1}, \begin{cases} snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \wedge \\ x_0 \dots x_{n-1} \in L(posex(E_1 + E_2)) \end{cases} \quad [\ def.\ posex\ ]$$

$$\iff \exists x_0 \dots x_{n-1}, \begin{cases} snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \wedge \\ x_0 \dots x_{n-1} \in L((posex(E)) \end{cases} \quad [\ E = E_1 + E_2\ ]$$

$$\iff u_0 \dots u_{n-1} \in L\big(E\big) \quad [\ lem.\ posex^{-1}\ ]$$

$$L(A_+) = L(E)$$

Case $E = E_1 \cdot E_2$:

$$
\begin{aligned}
A. &= (glu\_autom(E) \circ glushkov \circ posex)(E) \\
&= (glu\_autom(E) \circ glushkov \circ posex)(E_1 \cdot E_2) \qquad [\, E = E_1 \cdot E_2 \,] \\
&= (glu\_autom(E) \circ glushkov)\big((0 \bullet posex(E_1)) \cdot (1 \bullet posex(E_2))\big) \qquad [\, \textit{def. posex} \,]
\end{aligned}
$$

$$
= glu\_autom(E) \left(\begin{array}{rl}
first &= First(0 \bullet posex(E_1)) \cup First(1 \bullet posex(E_2)) \cdot Null(0 \bullet posex(E_1)) \\
last &= Last(1 \bullet posex(E_2)) \cup Last(0 \bullet posex(E_1)) \cdot Null(1 \bullet posex(E_2)) \\
null &= Null(0 \bullet posex(E_1)) \cap Null(1 \bullet posex(E_2)) \\
follow &= \left(\begin{array}{l}
\left\{ Follow\big(0 \bullet posex(E_1), x\big) \mid x \in pos\big(0 \bullet posex(E_1)\big) \right\} \\
\cup \left\{ \lambda x. First\big(1 \bullet posex(E_2)\big) \mid x \in Last\big(0 \bullet posex(E_1)\big) \right\} \\
\cup \left\{ Follow\big(1 \bullet posex(E_2), x\big) \mid x \in pos\big(1 \bullet posex(E_2)\big) \right\}
\end{array} \right)
\end{array} \right)
$$

$$
\left\{ \begin{array}{rl}
Q &= pos(posex(E)) \cup \{q_I\} \\
\Sigma &= \Sigma \\
\delta &= \left\{ \begin{array}{r}
\forall a \in \Sigma, \delta(q_I, a) = \left\{ x \mid x \in first, snd(x) = a \right\} \\
\forall x \in (2^* \times \Sigma), \forall a \in \Sigma, \delta(x, a) = \left\{ y \mid y \in follow(x), snd(y) = a \right\}
\end{array} \right. \\
I &= \{q_I\} \\
F &= \left\{ \begin{array}{ll}
last \cup \{q_I\} & \textbf{if } null = \{\epsilon\} \\
last & \textbf{otherwise}
\end{array} \right.
\end{array} \right.
$$

[reminder of *def. glu_autom*]

$$
= \left\{
\begin{array}{l}
Q = \{q_I\} \cup pos(0 \bullet posex(E_1)) \cup pos(1 \bullet posex(E_2)) \\
\\
\Sigma = \Big\{ snd(x) \mid x \in pos\big(0 \bullet posex(E_1)\big) \Big\} \cup \Big\{ snd(x) \mid x \in pos\big(1 \bullet posex(E_2)\big) \Big\} \\
\\
\delta = \left\{
\begin{array}{l}
\forall a \in \Sigma, \delta(q_I, a) = \left\{ x \;\middle|\; x \in \left(\begin{array}{l} First(0 \bullet posex(E_1)) \\ \cup\, First(1 \bullet posex(E_2)) \cdot Null(0 \bullet posex(E_1)) \end{array} \right) \wedge snd(x) = a \right\} \\
\forall x \in pos(0 \bullet posex(E_1)) \cup pos(1 \bullet posex(E_2)), \\
\quad \forall a \in \Sigma, \delta(x, a) = \left\{ y \;\middle|\; y \in Follow\big(0 \bullet posex(E_1), x\big) \cup Follow\big(1 \bullet posex(E_2), x\big) \wedge snd(y) = a \right\} \\
\qquad\qquad \cup \Big\{ y \mid y \in First\big(1 \bullet posex(E_2)\big), x \in Last\big(0 \bullet posex(E_1)\big), snd(y) = a \Big\}
\end{array}
\right. \\
\\
I = \{q_I\} \\
\\
F = \left\{
\begin{array}{l}
\textbf{if } Null(0 \bullet posex(E_1)) \cap Null(1 \bullet posex(E_2)) = \{\epsilon\} \\
\qquad \{q_I\} \cup \big(Last(0 \bullet posex(E_1)) \cdot Null(1 \bullet posex(E_2))\big) \cup Last(1 \bullet posex(E_2)) \\
\textbf{otherwise } \big(Last(0 \bullet posex(E_1)) \cdot Null(1 \bullet posex(E_2))\big) \cup Last(1 \bullet posex(E_2))
\end{array}
\right.
\end{array}
\right.
$$

[application of *def. glu_autom*]

$$
A. = \begin{cases}
Q = \{q_I\} \cup pos(0 \bullet posex(E_1)) \cup pos(1 \bullet posex(E_2)) \\
\Sigma = \left\{ snd(x) \mid x \in pos\big(0 \bullet posex(E_1)\big) \right\} \cup \left\{ snd(x) \mid x \in pos\big(1 \bullet posex(E_2)\big) \right\} \\
\delta = \begin{cases}
\forall a \in \left\{ snd(x) \mid x \in pos\big(0 \bullet posex(E_1)\big) \right\}, \\
\quad \delta(q_I, a) = \left\{ x \mid x \in First(0 \bullet posex(E_1)), snd(x) = a \right\} & [\ \delta_1\ ] \\
\forall a \in \left\{ snd(x) \mid x \in pos\big(1 \bullet posex(E_2)\big) \right\}, \\
\quad \delta(q_I, a) = \left\{ x \;\middle|\; \begin{array}{l} x \in First(1 \bullet posex(E_2)) \cdot Null(0 \bullet posex(E_1)), \\ snd(x) = a \end{array} \right\} & [\ \delta_2\ ] \\
\forall x \in pos(0 \bullet posex(E_1)) \\
\quad \forall a \in \left\{ snd(x) \mid x \in pos\big(0 \bullet posex(E_1)\big) \right\}, \\
\quad \delta(x, a) = \left\{ y \mid y \in Follow\big(0 \bullet posex(E_1), x\big), snd(y) = a \right\} & [\ \delta_3\ ] \\
\forall x \in Last\big(0 \bullet posex(E_1)\big) \\
\quad \forall a \in \left\{ snd(x) \mid x \in pos\big(1 \bullet posex(E_2)\big) \right\}, \\
\quad \delta(x, a) = \left\{ y \mid y \in First\big(1 \bullet posex(E_2)\big), snd(y) = a \right\} & [\ \delta_4\ ] \\
\forall x \in pos(1 \bullet posex(E_2)), \\
\quad \forall a \in \left\{ snd(x) \mid x \in pos\big(1 \bullet posex(E_2)\big) \right\}, \\
\quad \delta(x, a) = \left\{ y \mid y \in Follow\big(1 \bullet posex(E_2), x\big), snd(y) = a \right\} & [\ \delta_5\ ]
\end{cases} \\
I = \{q_I\} \\
F = \begin{cases}
\textbf{if } Null(0 \bullet posex(E_1)) \cap Null(1 \bullet posex(E_2)) = \{\epsilon\} \\
\quad \{q_I\} \cup \big(Last(0 \bullet posex(E_1)) \cdot Null(1 \bullet posex(E_2))\big) \cup Last(1 \bullet posex(E_2)) \\
\textbf{otherwise } \big(Last(0 \bullet posex(E_1)) \cdot Null(1 \bullet posex(E_2))\big) \cup Last(1 \bullet posex(E_2))
\end{cases}
\end{cases}
$$

[*lem.* Disjoint pos]

$$L(A) = L(E) \iff \big(\forall w \in \Sigma^*, w \in L(A) \iff w \in L(E)\big)$$

- $|w| = 0 \to w = \epsilon$

$$
\begin{aligned}
\epsilon \in L(A) &\iff q_I \in F \\
&\iff \quad Null(0 \bullet posex(E_1)) \cap Null(1 \bullet posex(E_2)) = \{\epsilon\} && [\ def\ F, q_I \notin last\ ] \\
&\iff \epsilon \in \big(L(0 \bullet posex(E_1)) \cap L(1 \bullet posex(E_2))\big) && [\ prop.\ Null\ ] \\
&\iff \big(\epsilon \in L(0 \bullet posex(E_1))\big) \wedge \big(\epsilon \in L(1 \bullet posex(E_2))\big) \\
&\iff \epsilon \in \big(L(0 \bullet posex(E_1)) \cdot L(1 \bullet posex(E_2))\big) && [\ rmk.\ L_1 \cdot L_2\ ] \\
&\iff \epsilon \in L\big((0 \bullet posex(E_1)) \cdot (1 \bullet posex(E_2))\big) && [\ def.\ L_{RE}\ ] \\
&\iff \epsilon \in L(posex(E_1 \cdot E_2)) && [\ def.\ posex\ ] \\
&\iff \epsilon \in L(posex(E)) && [\ E = E_1 \cdot E_2\ ] \\
&\iff \epsilon \in L(E) && [\ rmk.\ \epsilon \in L(posex^{-1})\ ]
\end{aligned}
$$

- $|w| = n, (n \geq 1) \to w = u_0 \dots u_{n-1} \in L(A) \iff \exists x, \big(x \in \delta^*(\delta(\{q_I\}, u_0), u_1 \dots u_{n-1})\big) \wedge (x \in F)$

▷ **if** $u_0 \in \Big\{ snd(x) \mid x \in pos\big(0 \bullet posex(E_1)\big) \Big\}$

▶ **if** $\exists m, 0 \leqslant m \leqslant n-2, \forall i \in \{1 \dots m\}, \forall j \in \{m+2 \dots n-1\}$, $x_{i-1} \in pos\big(0 \bullet posex(E_1)\big), w \in L(A)$

$$
\iff \begin{cases}
\exists x_0 \in First(0 \bullet posex(E_1)), & [\ A_\bullet[\delta_1], u_0 \text{ membership}\ ] \\
\exists x_i \in Follow(0 \bullet posex(E_1), x_{i-1}), & [\ A_\bullet[\delta_3], x_{i-1} \text{ membership}\ ] \\
\exists x_m \in Last\big(0 \bullet posex(E_1)\big), & [\ \text{Definition of } m\ ] \\
\exists x_{m+1} \in First\big(1 \bullet posex(E_2)\big), & [\ A_\bullet[\delta_4], x_m \text{ membership}\ ] \\
\exists x_j \in Follow(1 \bullet posex(E_2), x_{j-1}), & [\ A_\bullet[\delta_5], x_{j-1} \text{ membership}\ ] \\
x_{n-1} \in F, & [\ \text{Definition of } w \in L(A)\ ] \\
snd^*(x_0 \dots x_m x_{m+1} \dots x_{n-1}) = u_0 \dots u_m u_{m+1} \dots u_{n-1}
\end{cases}
$$

Whether $m = n-2$ or $m < n-2$ we have

$$
x_{n-1} \in pos\big(1 \bullet posex(E_2)\big) \qquad \begin{bmatrix} \big(x_{n-1} \in First(1 \bullet posex(E_2))\big) \\ \vee\ \big(x_{n-1} \in Follow\big(1 \bullet posex(E_2), x_{n-2}\big)\big) \end{bmatrix}
$$

$$
x_{n-1} \in F \iff x_{n-1} \in Last(1 \bullet posex(E_2))
$$

$$
[\ (x_{n-1} \neq q_I) \wedge \big(x_{n-1} \notin Last\big(0 \bullet posex(E_1)\big)\big)\ ]
$$

for the following , introduced variables belong to $\begin{bmatrix} x \in pos(0 \bullet posex(E_1)) \\ w \in pos(0 \bullet posex(E_1))^* \\ x \in pos(1 \bullet posex(E_2)) \\ w \in pos(1 \bullet posex(E_2))^* \end{bmatrix}$

$$\iff \begin{cases} \exists x_0, \exists w'_0, & x_0 w'_0 \in L(0 \bullet posex(E_1)), & [\ prop.\ First\ ] \\ \exists x_i, \exists w_{i-1}, \exists w'_i, & w_{i-1} x_{i-1} x_i w'_i \in L(0 \bullet posex(E_1)), & [\ prop.\ Follow\ ] \\ \exists x_m, \exists w_m, & w_m x_m \in L(0 \bullet posex(E_1)) & [\ prop.\ Last\ ] \\ \exists x_{m+1}, \exists w'_{m+1}, & x_{m+1} w'_{m+1} \in L(1 \bullet posex(E_2)), & [\ prop.\ First\ ] \\ \exists x_j, \exists w_{j-1}, \exists w'_j, & w_{j-1} x_{j-1} x_j w'_j \in L(1 \bullet posex(E_2)), & [\ prop.\ Follow\ ] \\ \exists x_{n-1}, \exists w_{n-1}, & w_{n-1} x_{n-1} \in L(1 \bullet posex(E_2)) & [\ prop.\ Last\ ] \\ snd^*(x_0 \dots x_m x_{m+1} \dots x_{n-1}) = u_0 \dots u_m u_{m+1} \dots u_{n-1} \end{cases}$$

For $x_0 \dots x_{n-1}$ to be the word constrained by all definitions above, we have

$$\begin{bmatrix} w_0 = \epsilon \\ w'_0 = x_1 \dots x_m \\ w_i = x_0 \dots x_{i-1} \\ w'_i = x_{i+1} \dots x_m \\ w_{m+1} = \epsilon \\ w'_{m+1} = x_{m+2} \dots x_{n-1} \\ w_j = x_{m+1} \dots x_{j-1} \\ w'_j = x_{j+1} \dots x_{n-1} \end{bmatrix}$$

Hence,

$$\iff \begin{cases} x_0 \dots x_m \in L(0 \bullet posex(E_1)) \\ x_{m+1} \dots x_{n-1} \in L(1 \bullet posex(E_2)) \\ snd^*(x_0 \dots x_m x_{m+1} \dots x_{n-1}) = u_0 \dots u_m u_{m+1} \dots u_{n-1} \end{cases}$$

▶ **if** $\forall i \in \{1 \dots n-1\}, x_{i-1} \in pos\big(0 \bullet posex(E_1)\big), w \in L(A)$ $\quad [\ \iff m = n-1\ ]$

$$\iff \begin{cases} \exists x_0 \in First(0 \bullet posex(E_1)), & [\ A_\bullet[\delta_1], u_0 \text{ membership}\ ] \\ \exists x_i \in Follow(0 \bullet posex(E_1), x_{i-1}), & [\ A_\bullet[\delta_3], u_0 \text{ membership}\ ] \\ x_{n-1} \in F & [\ \text{Definition of } w \in L(A)\ ] \\ snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \end{cases}$$

$$x_{n-1} \in F \iff x_{n-1} \in \big(Last(0 \bullet posex(E_1)) \cdot Null(1 \bullet posex(E_2))\big)$$
$$[\ x_{n-1} \in pos(0 \bullet posex(E_1))\ ]$$

$$x_{n-1} \in F \iff \big(x_{n-1} \in Last(0 \bullet posex(E_1))\big) \wedge \big(Null(1 \bullet posex(E_2)) = \{\epsilon\}\big)$$

for the following , each introduced variable $\begin{cases} x \in pos(0 \bullet posex(E_1)) \\ \text{and} \\ w \in pos(0 \bullet posex(E_1))^* \end{cases}$

$$\iff \begin{cases} \exists x_0, \exists w'_0, & x_0 w'_0 \in L(0 \bullet posex(E_1)), & [\ \textit{prop. First}\ ] \\ \exists x_i, \exists w_{i-1}, \exists w'_i, & w_{i-1} x_{i-1} x_i w'_i \in L(0 \bullet posex(E_1)), & [\ \textit{prop. Follow}\ ] \\ \exists x_m, \exists w_m, & w_m x_m \in L(0 \bullet posex(E_1)), & [\ \textit{prop. Last}\ ] \\ snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} & & \end{cases}$$

For $x_0 \dots x_{n-1}$ to be the word constrained by all definitions above, we have

$$\begin{bmatrix} w_0 = \epsilon \\ w'_0 = x_1 \dots x_{n-1} \\ w_i = x_0 \dots x_{i-1} \\ w'_i = x_{i+1} \dots x_{n-1} \end{bmatrix}$$

$$\iff \begin{cases} x_0 \dots x_{n-1} \in L(0 \bullet posex(E_1)) \\ snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \end{cases}$$

▷ **if** $u_0 \in \Big\{ snd(x) \mid x \in pos\big(1 \bullet posex(E_2)\big) \Big\}$

▶ $\forall i \in \{1 \dots n-1\} \in pos(1 \bullet posex(E_2)), w \in L(A) \quad [\iff m = -1]$

$$\iff \begin{cases} \exists x_0 \in \big(First(1 \bullet posex(E_2)) \cdot Null(0 \bullet posex(E_1))\big), & [\ A_\bullet[\delta_2], u_0 \text{ membership}\ ] \\ \exists x_i \in Follow(1 \bullet posex(E_2), x_{i-1}), & [\ A_\bullet[\delta_5], u_{i-1} \text{ membership}\ ] \\ x_{n-1} \in F & [\ \text{Definition of } w \in L(A)\ ] \\ snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} & \end{cases}$$

$$x_{n-1} \in F \iff x_{n-1} \in Last\big(1 \bullet posex(E_2)\big)$$
$$x_0 \in \big(First(1 \bullet posex(E_2)) \cdot Null(0 \bullet posex(E_1))\big)$$

$$\iff \big(x_0 \in First(1 \bullet posex(E_2))\big) \wedge \big(Null(0 \bullet posex(E_1))\big)$$

for the following , each introduced variable $x \in pos(1 \bullet posex(E_2))$
and $w \in pos(1 \bullet posex(E_2))^*$

$$\iff \begin{cases} \exists x_0, \exists w'_0, & x_0 w'_0 \in L(1 \bullet posex(E_2)), & [\ prop.\ First\ ] \\ \exists x_i, \exists w_{i-1}, \exists w'_i, & w_{i-1} x_{i-1} x_i w'_i \in L(1 \bullet posex(E_2)), & [\ prop.\ Follow\ ] \\ \exists x_m, \exists w_m, & w_m x_m \in L(1 \bullet posex(E_2)), & [\ prop.\ Last\ ] \\ snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} & & \end{cases}$$

For $x_0 \dots x_{n-1}$ to be the word constrained by all definitions above, we have

$$\begin{bmatrix} w_0 = \epsilon \\ w'_0 = x_1 \dots x_{n-1} \\ w_i = x_0 \dots x_{i-1} \\ w'_i = x_{i+1} \dots x_{n-1} \end{bmatrix}$$

$$\iff \begin{cases} x_0 \dots x_{n-1} \in L(1 \bullet posex(E_2)) \\ snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \end{cases}$$

▷ **Hence**,

$$\iff \begin{cases} \exists m, -1 \leqslant m \leqslant n-1, \exists x_0 \dots x_m \dots x_{n-1}, \\ \quad snd^*(x_0 \dots x_m \dots x_{n-1}) = u_0 \dots u_m \dots u_{n-1} \\ \quad \wedge \Big(\big(x_0 \dots x_m \in L(0 \bullet posex(E_1))\big) \wedge \big(x_{m+1} \dots x_{n-1} \in L(1 \bullet posex(E_2))\big) \Big) \end{cases}$$

$$\iff \begin{cases} \exists m, -1 \leqslant m \leqslant n-1, \exists x_0 \dots x_m \dots x_{n-1}, \\ \quad snd^*(x_0 \dots x_m \dots x_{n-1}) = u_0 \dots u_m \dots u_{n-1} & [\ def.\ L_1 \cdot L_2\ ] \\ \quad \wedge \Big(x_0 \dots x_m \dots x_{n-1} \in \big(L(0 \bullet posex(E_1)) \cdot L(1 \bullet posex(E_2))\big) \Big) \end{cases}$$

$$\iff \begin{cases} \exists m, -1 \leqslant m \leqslant n-1, \exists x_0 \dots x_m \dots x_{n-1}, \\ \quad snd^*(x_0 \dots x_m \dots x_{n-1}) = u_0 \dots u_m \dots u_{n-1} \\ \quad \wedge \Big(x_0 \dots x_m \dots x_{n-1} \in L\big(posex((E_1) \cdot L(E_2))\big)\Big) \end{cases} \qquad [\ def.\ posex\ ]$$

$$\iff \begin{cases} \exists x_0 \dots x_{n-1}, snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \\ \quad \wedge \big(x_0 \dots x_{n-1} \in L(posex(E))\big) \end{cases} \qquad [\ E = E_1 \cdot E_2,\ ]$$

$$\iff u_0 \dots u_{n-1} \in L\big(E\big) \qquad [\ lem.\ posex^{-1}\ ]$$

$$L(A_\bullet) = L(E)$$

Case $E = F^* :$

$$A_* = (glu\_autom(E) \circ glushkov \circ posex)(E)$$

$$= (glu\_autom(E) \circ glushkov \circ posex)(F^*) \qquad [\ E = F^*\ ]$$

$$= (glu\_autom(E) \circ glushkov)\big((0 \bullet posex(F))^*\big) \qquad [\ def.\ posex\ ]$$

$$= glu\_autom(E) \begin{pmatrix} first = First\big(0 \bullet posex(F)\big) \\ last = Last\big(0 \bullet posex(F)\big) \\ null = \{\epsilon\} \\ follow = \begin{pmatrix} \big\{Follow\big(0 \bullet posex(F), x\big) \mid x \in pos\big(0 \bullet posex(F)\big)\big\} \cup \\ \big\{\lambda x.First\big(1 \bullet posex(E_2)\big) \mid x \in Last\big(0 \bullet posex(F)\big)\big\} \end{pmatrix} \end{pmatrix}$$

$[\ def.\ glushkov\ ]$

$$\begin{cases} Q = pos(posex(E)) \cup \{q_I\} \\ \Sigma = \Sigma \\ \delta = \begin{cases} \forall a \in \Sigma, \delta(q_I, a) = \big\{x \mid x \in first, snd(x) = a\big\} \\ \forall x \in (2^* \times \Sigma), \forall a \in \Sigma, \delta(x, a) = \big\{y \mid y \in follow(x), snd(y) = a\big\} \end{cases} \\ I = \{q_I\} \\ F = \begin{cases} last \cup \{q_I\} & \textbf{if } null = \{\epsilon\} \\ last & \textbf{otherwise} \end{cases} \end{cases}$$

[reminder of *def. glu_autom*]

$$A_* = \begin{cases} Q = \{q_I\} \cup pos(0 \bullet posex(F)) \\ \Sigma = \left\{ snd(x) \mid x \in pos\big(0 \bullet posex(F)\big) \right\} \\ \delta = \begin{cases} \forall a \in \Sigma, \delta(q_I, a) = \left\{ x \mid x \in First\big(0 \bullet posex(F)\big), snd(x) = a \right\} & [\ \delta_1\ ] \\ \forall x \in pos(0 \bullet posex(F)), \\ \quad \forall a \in \Sigma, \delta(x, a) = \left\{ y \mid y \in Follow\big(0 \bullet posex(F), x\big), snd(y) = a \right\} & [\ \delta_2\ ] \\ \forall x \in Last\big(0 \bullet posex(F)\big) \\ \quad \forall a \in \Sigma, \delta(x, a) = \left\{ y \mid y \in First\big(0 \bullet posex(F)\big), snd(y) = a \right\} & [\ \delta_3\ ] \end{cases} \\ I = \{q_I\} \\ F = \{q_I\} \cup Last\big(0 \bullet posex(F)\big) \end{cases}$$

A_* [application of *def. glu_autom*]

$$L(A) = L(E) \iff \big(\forall w \in \Sigma^*, w \in L(A) \iff w \in L(E)\big)$$

$$w = u_0 \dots u_{n-1} \in L(A) \iff \exists x, \big(x \in \delta^*(\delta(\{q_I\}, u_0), u_1 \dots u_{n-1})\big) \wedge (x \in F)$$

$$\implies \begin{cases} \exists x_0, \in First(0 \bullet posex(F)), & [\ A_*[\delta_1]\ ] \\ \exists k, \forall i \in \{0 \dots k\}, & \\ \quad \exists x_i, \in Follow(0 \bullet posex(F), x_{i-1}) & [\ A_*[\delta_2]\ ] \\ \quad x_k \in Last(0 \bullet posex(F)) & [\ \text{Def } k\ ] \\ \exists x_{k+1} \in First(0 \bullet posex(F)) & [\ A_*[\delta_3]\ ] \\ \exists k', \forall i \in \{k+2 \dots k'\}, & \\ \quad \exists x_i, \in Follow(0 \bullet posex(F), x_{i-1}) & [\ A_*[\delta_2]\ ] \\ \qquad \vdots & \\ \exists x_{n-1}, \in Last(0 \bullet posex(F)) & [\ \text{Def of } w \in L(A)\ ] \end{cases}$$

An induction on the number of δ_3 applied (noted m) is chosen

- $m = 0$

  - $|w| = 0 \to w = \epsilon$

$$\begin{cases} \epsilon \in L(A) & [\ I = F\ ] \\ \epsilon \in L(F^*) & [\ def.\ L_{RE}\ ] \end{cases}$$

$\triangleright\ |w| = n \to w = u_0 \dots u_{n-1}, w \in L(A)$

$$\iff \begin{cases} \exists x_0, \in First(0 \bullet posex(F)), & [\ A_*[\delta_1]\ ] \\ \quad \forall i \in \{1 \dots n-1\}, & \\ \exists x_i, \in Follow(0 \bullet posex(F), x_{i-1}) & [\ A_*[\delta_2]\ ] \\ x_{n-1}, \in Last(0 \bullet posex(F)) & [\ \text{Def of } w \in L(A)\ ] \\ snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} & \end{cases}$$

for the following , each introduced variable $x \in pos(0 \bullet posex(F))$ and $w \in pos(0 \bullet posex(F))^*$

$$\iff \begin{cases} \exists x_0, \exists w'_0, \quad x_0 w'_0 \in L(0 \bullet posex(F)), & [\ prop.\ First\ ] \\ \forall i \in \{1 \dots n-1\}, & \\ \exists x_i, \exists w_{i-1}, \exists w'_i, \ w_{i-1} x_{i-1} x_i w'_i \in L(0 \bullet posex(F)), & [\ prop.\ Follow\ ] \\ \quad x_{n-1}, \exists w_{n-1}, \quad w_{n-1} x_{n-1} \in L(0 \bullet posex(F)), & [\ prop.\ Last\ ] \\ snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} & \end{cases}$$

For $x_0 \dots x_{n-1}$ to be the word constrained by all definitions above, we have

$$\begin{bmatrix} w_0 = \epsilon \\ w'_0 = x_1 \dots x_{n-1} \\ w_i = x_0 \dots x_{i-1} \\ w'_i = x_{i+1} \dots x_{n-1} \end{bmatrix}$$

$$\iff \begin{cases} x_0 \dots x_{n-1} \in L(0 \bullet posex(F)) \\ snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \end{cases}$$

$$\Longrightarrow \begin{cases} x_0 \dots x_{n-1} \in L\big((0 \bullet posex(F))^*\big) & \big[\, L(0 \bullet posex(F)) \subset L\big((0 \bullet posex(F))^*\big) \,\big] \\ snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \end{cases}$$

$$\Longrightarrow \begin{cases} x_0 \dots x_{n-1} \in L(posex(F^*)) & [\, \textit{def. posex} \,] \\ snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \end{cases}$$

$$\Longrightarrow \begin{cases} x_0 \dots x_{n-1} \in L(posex(E)) & [\, E = F^* \,] \\ snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \end{cases}$$

$$\Longrightarrow u_0 \dots u_{n-1} \in L(E) \qquad [\, \textit{lem. posex}^{-1} \,]$$

- $m \geq 1, w \in L(A)$

$$\Longleftrightarrow \begin{cases} \vdots & \\ \exists x_k \in First(0 \bullet posex(F)), & [\, A_*[\delta_3] \,] \\ \forall i \in \{k+1 \dots n-1\}, & \\ \quad \exists x_i \in Follow((0 \bullet posex(F)), x_{i-1}) & [\, A_*[\delta_2] \,] \\ x_{n-1} \in Last(0 \bullet posex(F)), & [\, w \in L(A) \,] \\ snd^*(x_k \dots x_{n-1}) = u_k \dots u_{n-1} & \end{cases}$$

$$\Longleftrightarrow \begin{cases} \vdots & \\ \exists x_k, \exists w'_k, \quad x_k w'_k \in L(0 \bullet posex(F)), & [\, \textit{prop. First} \,] \\ \forall i \in \{k+1 \dots n-1\}, & \\ \exists x_i, \exists w_{i-1}, \exists w'_i, \; w_{i-1} x_{i-1} x_i w'_i \in L(0 \bullet posex(F)), & [\, \textit{prop. Follow} \,] \\ \exists x_{n-1}, \exists w_{n-1}, \quad w_{n-1} x_{n-1} \in L(0 \bullet posex(F)), & [\, \textit{prop. Last} \,] \\ snd^*(x_k \dots x_{n-1}) = u_k \dots u_{n-1} & \end{cases}$$

For $x_0 \dots x_{n-1}$ to be the word constrained by all definitions above, we have

$$\left[\begin{array}{l} w_k = \epsilon \\ w'_k = x_{k+1} \dots x_{n-1} \\ w_i = x_k \dots x_{i-1} \\ w'_i = x_{i+1} \dots x_{n-1} \end{array}\right]$$

$$\text{So, } \begin{cases} x_k \dots x_{n-1} \in L(0 \bullet posex(F)) \\ snd^*(x_k \dots x_{n-1}) = u_k \dots u_{n-1} \end{cases}$$

From induction hypothesis,

$$\text{we also have } \begin{cases} x_0 \dots x_{k-1} \in L\big((0 \bullet posex(F))^m\big) \subset L\big((0 \bullet posex(F))^*\big) \\ snd^*(x_0 \dots x_{k-1}) = u_0 \dots u_{k-1} \end{cases}$$

$$\text{To sum up, } w \in L(A) \iff \begin{cases} x_0 \dots x_{k-1} \in L\big((0 \bullet posex(F))^m\big) \\ x_k \dots x_{n-1} \in L(0 \bullet posex(F)) \\ snd^*(x_0 \dots x_{k-1} x_k \dots x_{n-1}) = u_0 \dots u_{k-1} u_k \dots u_{n-1} \end{cases}$$

$$\iff \begin{cases} x_0 \dots x_{k-1} x_k \dots x_{n-1} \in L\big((0 \bullet posex(F))^m\big) \cdot L(0 \bullet posex(F)) \\ snd^*(x_0 \dots x_{k-1} x_k \dots x_{n-1}) = u_0 \dots u_{k-1} u_k \dots u_{n-1} \end{cases}$$

$$\iff \begin{cases} x_0 \dots x_{n-1} \in L\big((0 \bullet posex(F))^{m+1}\big) \subset L\big((0 \bullet posex(F))^*\big) \\ snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \end{cases}$$

$$\implies \begin{cases} x_0 \dots x_{n-1} \in L\big((0 \bullet posex(F))^*\big) \\ snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \end{cases}$$

$$\implies \begin{cases} x_0 \dots x_{n-1} \in L(posex(F^*)) \\ snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \end{cases} \quad \big[\ def.\ posex\ \big]$$

$$\implies \begin{cases} x_0 \dots x_{n-1} \in L(posex(E)) \\ snd^*(x_0 \dots x_{n-1}) = u_0 \dots u_{n-1} \end{cases} \quad \big[\ E = F^*\ \big]$$

$$\implies u_0 \dots u_{n-1} \in L(E) \qquad \big[\ lem.\ posex^{-1}\ \big]$$

It was just proved that :

$$\forall w \in \Sigma^*, w \in L(A) \implies w \in L(F^*) \qquad (\iff L(A) \subset L(F^*))$$

Is left to prove :

$$\forall w \in \Sigma^*, w \in L(F^*) \implies w \in L(A) \qquad (\iff L(F^*) \subset L(A))$$

$$L(F^*) = \bigcup_{i=0}^{\infty} L(F^i) \qquad [\ \textit{def. } L_{RE}\ ]$$

By induction on i

- $L^0(F) = \{\epsilon\}$

$$\epsilon \in L(A) \qquad [\ I = F\ ]$$

- $L(F^{i+1}) = L(F^i \cdot F) = L(F^i) \cdot L(F)$

$$w = u_0 \dots u_{n-1}, w \in L(F^i) \cdot L(F)$$

$$\implies \begin{cases} u_0 \dots u_{m-1} \in L(F^i), \\ u_m \dots u_{n-1} \in L(F), \end{cases}$$

$$\implies \begin{cases} (u_0 \dots u_{m-2})u_{m-1} \in L(F^*) & [\ w \in L(F^i) \implies w \in L(F^*)\ ] \\ u_m(u_{m+1} \dots u_{n-1}) \in L(F^*), & [\ w \in L(F) \implies w \in L(F^*)\ ] \\ \forall i \in \{m+1 \dots n-1\}, & \\ (u_m \dots)u_{i-1}u_i(\dots u_{n-1}) \in L(F^*), & [\ w \in L(F) \implies w \in L(F^*)\ ] \\ (u_m \dots u_{n-2})u_{n-1} \in L(F^*) & [\ w \in L(F) \implies w \in L(F^*)\ ] \end{cases}$$

$$\implies \begin{cases} \exists x_{m-1} \in \mathit{Last}(0 \bullet \mathit{posex}(F)) & \mathit{snd}(x_{m-1}) = u_{m-1} & [\ \textit{prop. Last}\ ] \\ \exists x_m \in \mathit{First}(0 \bullet \mathit{posex}(F)), & \mathit{snd}(x_m) = u_m & [\ \textit{prop. Follow}\ ] \\ \forall i \in \{m+1 \dots n-1\}, & & \\ \exists x_i \in \mathit{Follow}(0 \bullet \mathit{posex}(F), x_{i-1}), & \mathit{snd}(x_i) = u_i & [\ \textit{prop. Follow}\ ] \\ x_{n-1} \in \mathit{Last}(0 \bullet \mathit{posex}(F)) & & [\ \textit{prop. Last}\ ] \end{cases}$$

$$\Longrightarrow \begin{cases} \exists x_{m-1}, \big(x_{m-1} \in \delta^*(\delta(q_I, u_0), u_1 \dots u_{m-1})\big) & [\ IH\ ] \\ \qquad\qquad x_{m-1} \in F & [\ IH \wedge A_*[F]\ ] \\ \qquad \delta(x_{m-1}, snd(x_m)) = x_m & [\ A_*[\delta_3]\ ] \\ \forall i \in \{m+1 \dots n-1\}, & \\ \exists x_i \quad \delta(x_{i-1}, snd(x_i)) = x_i & [\ A_*[\delta_2]\ ] \\ \qquad\qquad x_{n-1} \in F & [\ A_*[F]\ ] \end{cases}$$

$\Longrightarrow w \in L(A)$

We have proved $L(A_*) \subset L(F^*)$ and $L(F^*) \subset L(A_*)$.
So, $L(A_*) = L(F^*)$

Now that all cases were covered

$$L(A) = L(E)$$

□

A.3 BSPA generation proof

Lemma 4 (dsn^{-1}). $\forall R^\circ \in (\Sigma, p)\, bspre^\circ,\ \forall m \in \mathbb{N}, \forall j \in \{0 \dots m-1\}, \forall w_i^t \in \Sigma^{*^m}, \exists\, ;_{t^j},$

$$\forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in L\big(dsn^i(R^\circ)\big)$$
$$\Longleftrightarrow$$
$$\langle w_0^0, \dots, w_{p-1}^0 \rangle_{t^0} \dots \langle w_0^{m-1}, \dots, w_{p-1}^{m-1} \rangle_{t^{m-1}} \in L\big(R^\circ\big)$$

Proof by structural induction on R°.

Case $R^\circ = \epsilon$

$\forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in L\big(dsn^i(\epsilon)\big)$

$\iff \forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in L(\epsilon)$ [*def. dsn*]

$\iff \forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in \{\epsilon\}$ [*def.* L_{RE}]

$\iff \forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} = \epsilon$

$\iff \langle w_0^0, \dots, w_{p-1}^0 \rangle_{t^0} \dots \langle w_0^{m-1}, \dots, w_{p-1}^{m-1} \rangle_{t^{m-1}} = \epsilon$

$\iff \langle w_0^0, \dots, w_{p-1}^0 \rangle_{t^0} \dots \langle w_0^{m-1}, \dots, w_{p-1}^{m-1} \rangle_{t^{m-1}} \in \{\epsilon\}$

$\iff \langle w_0^0, \dots, w_{p-1}^0 \rangle_{t^0} \dots \langle w_0^{m-1}, \dots, w_{p-1}^{m-1} \rangle_{t^{m-1}} \in L(\epsilon)$

$\iff \langle w_0^0, \dots, w_{p-1}^0 \rangle_{t^0} \dots \langle w_0^{m-1}, \dots, w_{p-1}^{m-1} \rangle_{t^{m-1}} \in L(R^\circ)$

OK

Case $R^\circ = \emptyset$

$\forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in L\big(dsn^i(\emptyset)\big)$

$\iff \forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in L(\emptyset)$ [*def. dsn*]

$\iff \forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in \{\}$ [*def.* L_{RE}]

$\iff \bot$

and

$\langle w_0^0, \dots, w_{p-1}^0 \rangle_{t^0} \dots \langle w_0^{m-1}, \dots, w_{p-1}^{m-1} \rangle_{t^{m-1}} \in L(\emptyset) \iff \bot$

which is equivalent.

OK

Case $R^\circ = \langle r_0, \dots, r_{p-1} \rangle_0$

$\forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in L\big(dsn^i(\langle r_0, \dots, r_{p-1} \rangle_0)\big)$

$\iff \forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in L\big(r_i \cdot ;_0\big)$ [*def. dsn*]

$\iff \forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in L(r_i) \cdot L(;_0)$ [*def.* L_{RE}]

$\iff \forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in L(r_i) \cdot \{;_0\}$ [*def.* L_{RE}]

$\iff \forall i \in [p], w_i^0 ;_0 \in L(r_i) \cdot \{;_0\}$ [$m = 1$]

$\iff \forall i \in [p], w_i^0 \in L(r_i)$

$\iff \langle w_0^0, \dots, w_{p-1}^0 \rangle_0 \in L(\langle r_0, \dots, r_{p-1} \rangle_0)$

OK

Case $R^\circ = R_1^\circ + R_2^\circ$

$$\forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in L\big(dsn^i(R_1^\circ + R_2^\circ)\big)$$
$$\iff \forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in L\big(dsn^i(R_1^\circ) + dsn^i(R_2^\circ)\big) \qquad [\ def.\ dsn\ ]$$
$$\iff \forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in L\big(dsn^i(R_1^\circ)\big) \cup L\big(dsn^i(R_2^\circ)\big) \qquad [\ def.\ L_{RE}\ ]$$

We have

- $\forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in L\big(dsn^i(R_1^\circ)\big)$
  $\iff \langle w_0^0, \dots, w_{p-1}^0 \rangle_{t^0} \dots \langle w_0^{m-1}, \dots, w_{p-1}^{m-1} \rangle_{t^{m-1}} \in L\big(R_1^\circ\big) \qquad [\ IH\ ]$
- $\forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in L\big(dsn^i(R_2^\circ)\big)$
  $\iff \langle w_0^0, \dots, w_{p-1}^0 \rangle_{t^0} \dots \langle w_0^{m-1}, \dots, w_{p-1}^{m-1} \rangle_{t^{m-1}} \in L\big(R_2^\circ\big) \qquad [\ IH\ ]$

Hence $\forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in L\big(dsn^i(R_1^\circ)\big) \cup L\big(dsn^i(R_2^\circ)\big)$
$\iff \langle w_0^0, \dots, w_{p-1}^0 \rangle_{t^0} \dots \langle w_0^{m-1}, \dots, w_{p-1}^{m-1} \rangle_{t^{m-1}} \in L\big(R_1^\circ\big) \cup L\big(R_2^\circ\big)$
$\iff \langle w_0^0, \dots, w_{p-1}^0 \rangle_{t^0} \dots \langle w_0^{m-1}, \dots, w_{p-1}^{m-1} \rangle_{t^{m-1}} \in L\big(R_1^\circ + R_2^\circ\big) \qquad [\ def.\ L_{RE}\ ]$

OK

Case $R^\circ = R_1^\circ \cdot R_1^\circ$

$$\forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in L\big(dsn^i(R_1^\circ \cdot R_2^\circ)\big)$$
$$\iff \forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in L\big(dsn^i(R_1^\circ) \cdot dsn^i(R_2^\circ)\big) \qquad [\ def.\ dsn\ ]$$
$$\iff \forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in L\big(dsn^i(R_1^\circ)\big) \cdot L\big(dsn^i(R_2^\circ)\big) \qquad [\ def.\ L_{RE}\ ]$$

$\exists n, 0 < n < m$

- $\forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{n-1} ;_{t^{n-1}} \in L\big(dsn^i(R_1^\circ)\big)$
  $\iff \langle w_0^0, \dots, w_{p-1}^0 \rangle_{t^0} \dots \langle w_0^{m-1}, \dots, w_{p-1}^{m-1} \rangle_{t^{m-1}} \in L\big(R_1^\circ\big) \qquad [\ IH\ ]$
- $\forall i \in [p], w_i^n ;_t^n \dots w_i^{m-1} ;_{t^{m-1}} \in L\big(dsn^i(R_2^\circ)\big)$
  $\iff \langle w_0^0, \dots, w_{p-1}^0 \rangle_{t^0} \dots \langle w_0^{m-1}, \dots, w_{p-1}^{m-1} \rangle_{t^{m-1}} \in L\big(R_2^\circ\big) \qquad [\ IH\ ]$

Hence

$$\left(\begin{array}{l} \forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{n-1} ;_{t^{n-1}} w_i^n ;_t^n \dots w_i^{m-1} ;_{t^{m-1}} \in L\big(dsn^i(R_1^\circ)\big) \cdot L\big(dsn^i(R_2^\circ)\big) \\ \iff \\ \langle w_0^0, \dots, w_{p-1}^0 \rangle \dots \langle w_0^{n-1}, \dots, w_{p-1}^{n-1} \rangle \langle w_0^n, \dots, w_{p-1}^n \rangle \dots \langle w_0^{m-1}, \dots, w_{p-1}^{m-1} \rangle \in L(R_1^\circ) \cdot L(R_2^\circ) \end{array}\right)$$
$$\iff \langle w_0^0, \dots, w_{p-1}^0 \rangle \dots \langle w_0^{m-1}, \dots, w_{p-1}^{m-1} \rangle \in L(R_1^\circ \cdot R_2^\circ) \qquad [\ def.\ L_{RE}\ ]$$

OK

Case $R^\circ = (R_1^\circ)^*$

$\forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in L\big(dsn^i((R_1^\circ)^*)\big)$

$\iff \forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in L\big(dsn^i((R_1^\circ))^*\big)$ [*def. dsn*]

$\iff \forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in \bigcup_{j=0}^{\infty} L^j\big(dsn^i(R_1^\circ)\big)$ [*def.* L_{RE}]

We prove first $\forall j, \forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in L^j\big(dsn^i(R_1^\circ)\big)$

$\iff \langle w_0^0, \dots, w_{p-1}^0 \rangle_{t^0} \dots \langle w_0^{m-1}, \dots, w_{p-1}^{m-1} \rangle_{t^{m-1}} \in L^j(R_1^\circ)$

by induction on j.

- $j = 0$

  $L^0\big(dsn^i(R_1^\circ)\big) = \{\epsilon\}$ OK. [See case $R^\circ = \epsilon$]

- $\forall j > 0$

$\forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in L^j\big(dsn^i(R_1^\circ)\big)$

$\iff \forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in L\big(dsn^i(R_1^\circ)\big) \cdot L^{j-1}\big(dsn^i(R_1^\circ)\big)$

$\iff \exists n, 0 < n < m, \forall i \in [p], \begin{cases} w_i^0 ;_{t^0} \dots w_i^{n-1} ;_{t^{n-1}} \in L\big(dsn^i(R_1^\circ)\big) \\ w_i^n ;_t^n \dots w_i^{m-1} ;_{t^{m-1}} \in L^{j-1}\big(dsn^i(R_1^\circ)\big) \end{cases}$

$\iff \exists n, 0 < n < m, \begin{cases} \langle w_0^0, \dots, w_{p-1}^0 \rangle_{t^0} \dots \langle w_0^{n-1}, \dots, w_{p-1}^{n-1} \rangle_{t^{n-1}} \in L(R_1^\circ) & [\text{ outer proof } IH\ ] \\ \langle w_0^n, \dots, w_{p-1}^n \rangle_{t^n} \dots \langle w_0^{m-1}, \dots, w_{p-1}^{m-1} \rangle_{t^{m-1}} \in L^{j-1}(R_1^\circ) & [\text{ inner proof } IH\ ] \end{cases}$

$\iff \exists n, \langle w_0^0, \dots, w_{p-1}^0 \rangle_{t^0} \dots \langle w_0^n, \dots, w_{p-1}^n \rangle_{t^n} \dots \langle w_0^{m-1}, \dots, w_{p-1}^{m-1} \rangle_{t^{m-1}} \in L^j\big(R_1^\circ\big)$

$\iff \langle w_0^0, \dots, w_{p-1}^0 \rangle_{t^0} \dots \langle w_0^{m-1}, \dots, w_{p-1}^{m-1} \rangle_{t^{m-1}} \in L^j\big(R_1^\circ\big)$

Thus, we have

$\forall j, \forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in L^j\big(dsn^i(R_1^\circ)\big)$

$\iff \langle w_0^0, \dots, w_{p-1}^0 \rangle_{t^0} \dots \langle w_0^{m-1}, \dots, w_{p-1}^{m-1} \rangle_{t^{m-1}} \in L^j(R_1^\circ)$

$\implies \left(\begin{array}{l} \forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in \bigcup_{j=0}^{\infty} L^j\big(dsn^i(R_1^\circ)\big) \\ \quad \iff \langle w_0^0, \dots, w_{p-1}^0 \rangle_{t^0} \dots \langle w_0^{m-1}, \dots, w_{p-1}^{m-1} \rangle_{t^{m-1}} \in \bigcup_{j=0}^{\infty} L^j(R_1^\circ) \end{array} \right)$

$\implies \left(\begin{array}{l} \forall i \in [p], w_i^0 ;_{t^0} \dots w_i^{m-1} ;_{t^{m-1}} \in L\big((R_1^\circ)^*\big) \\ \quad \iff \langle w_0^0, \dots, w_{p-1}^0 \rangle_{t^0} \dots \langle w_0^{m-1}, \dots, w_{p-1}^{m-1} \rangle_{t^{m-1}} \in L\big((R_1^\circ)^*\big) \end{array} \right)$

□

Lemma 5 ($annot^{-1}$). *$\forall R \in (\Sigma, p)$ bspre, $\forall m \in \mathbb{N}$,*

$$\begin{aligned} \forall j \in [m], \exists t^j, \langle w_0^0, \dots, w_{p-1}^0 \rangle_{t^0} \dots \langle w_0^{m-1}, \dots, w_{p-1}^{m-1} \rangle_{t^{m-1}} &\in L\big(annot(R)\big) \\ \iff \langle w_0^0, \dots, w_{p-1}^0 \rangle \dots \langle w_0^{m-1}, \dots, w_{p-1}^{m-1} \rangle &\in L(R) \end{aligned}$$

Proof

We prove this equivalence by proving the two implications

- $$\left[\begin{array}{r} \forall j \in [m], \exists t^j, \langle w_0^0, \ldots, w_{p-1}^0 \rangle_{t^0} \ \ldots \ \langle w_0^{m-1}, \ldots, w_{p-1}^{m-1} \rangle_{t^{m-1}} \in L\big(annot(R)\big) \\ \Longrightarrow \langle w_0^0, \ldots, w_{p-1}^0 \rangle \quad \ldots \ \langle w_0^{m-1}, \ldots, w_{p-1}^{m-1} \rangle \in L\big(R\big) \end{array}\right]$$

Let function *unannot* the function removing annotations of the vectors in a BSPRE.

def 5.1 *unannot* **:** $(\Sigma, p)\ bspre^{\circ} \rightarrow (\Sigma, p)\ bspre$

$$\begin{aligned} unannot(R_1 + R_2) &= unannot(R_1) + unannot(R_2) \\ unannot(R_1 \cdot R_2) &= unannot(R_1) \cdot unannot(R_2) \\ unannot(R_1^*) &= (unannot(R_1))^* \\ unannot(\langle r_0, \ldots, r_{p-1} \rangle_t) &= \langle r_0, \ldots, r_{p-1} \rangle \\ unannot(\epsilon) &= \epsilon \\ unannot(\emptyset) &= \emptyset \end{aligned}$$

This function is defined so that the following property holds.

prop 5.2 $R = unannot(annot(R))$

Indeed function *annot* only annotate vectors in the BSPRE with a unique identifier. It does not modify vectors content nor BSPRE structure (*def. annot*). This is enough to prove prop. 5.2.

We also introduce the same function operating on words (symbol list).

def 5.3 *unannot_word* **:** $\big((\Sigma\ re\ vec) \times \mathbb{N}\big)\ list \rightarrow \big(\Sigma\ re\ vec\big)\ list$

$$\begin{aligned} unannot\_word(\langle w_0, \ldots, w_{p-1} \rangle_t :: W) &= \langle w_0, \ldots, w_{p-1} \rangle :: unannot\_word(W) \\ unannot\_word([\,]) &= [\,] \end{aligned}$$

The two functions defined above allow us to write the following implication which is true for any annotation t because annotations are removed from both the language with *unannot* and from the word with *unannot_word*.

$$\begin{aligned} & \langle w_0^0, \ldots, w_{p-1}^0 \rangle_{t^0} \ \ldots \ \langle w_0^{m-1}, \ldots, w_{p-1}^{m-1} \rangle_{t^{m-1}} \in L\big(annot(R)\big) \\ \Longrightarrow\ & unannot\_word\big(\langle w_0^0, \ldots, w_{p-1}^0 \rangle_{t^0} \ \ldots \ \langle w_0^{m-1}, \ldots, w_{p-1}^{m-1} \rangle_{t^{m-1}}\big) \in L\big(unannot(annot(R))\big) \end{aligned}$$

Whose right-hand member may be reduced

$$
\begin{aligned}
&\mathit{unannot\_word}\big(\langle w_0^0,\dots,w_{p-1}^0\rangle_{t^0}\dots\langle w_0^{m-1},\dots,w_{p-1}^{m-1}\rangle_{t^{m-1}}\big) \in L\big(\mathit{unannot}(\mathit{annot}(R))\big)\\
&\iff \langle w_0^0,\dots,w_{p-1}^0\rangle\dots\langle w_0^{m-1},\dots,w_{p-1}^{m-1}\rangle \in L\big(\mathit{unannot}(\mathit{annot}(R))\big) \qquad [\ \textit{def. unannot\_word}\ ]\\
&\iff \langle w_0^0,\dots,w_{p-1}^0\rangle\dots\langle w_0^{m-1},\dots,w_{p-1}^{m-1}\rangle \in L\big(R\big) \qquad [\ \textit{prop. 5.2}\ ]
\end{aligned}
$$

By replacing the reduced right-hand member in the original implication, we have $\forall t^j$

$$
\begin{aligned}
&\langle w_0^0,\dots,w_{p-1}^0\rangle_{t^0}\ \dots\ \langle w_0^{m-1},\dots,w_{p-1}^{m-1}\rangle_{t^{m-1}} \in L\big(\mathit{annot}(R)\big)\\
\implies\ &\langle w_0^0,\dots,w_{p-1}^0\rangle\ \dots\ \langle w_0^{m-1},\dots,w_{p-1}^{m-1}\rangle \in L\big(R\big)
\end{aligned}
$$

OK

- $$\left[\begin{array}{r} \forall j\in[m], \exists t^j,\ \langle w_0^0,\dots,w_{p-1}^0\rangle_{t^0}\ \dots\ \langle w_0^{m-1},\dots,w_{p-1}^{m-1}\rangle_{t^{m-1}} \in L\big(\mathit{annot}(R)\big)\\ \impliedby \langle w_0^0,\dots,w_{p-1}^0\rangle\quad \dots\ \langle w_0^{m-1},\dots,w_{p-1}^{m-1}\rangle \in L\big(R\big)\end{array}\right]$$

  Quantifier "$\exists t^j$" allow us to pick the right annotation for all m vectors, thereby reflecting any annotation of $\mathit{annot}(R)$ in the BSP word. This implication is thus trivially true.

  OK

The two implications were proven thereby proving the equivalence of lem. 5.

□

Conjecture 1 (Language preservation). $\forall R \in (\Sigma, p)\ bspre,$

$$L\big((Sync \circ BK^p \circ Dsync)(R)\big) = L(R)$$

Proof

By induction on R.

Cases $R = R_1 \cdot R_2$ and $R = R_1^*$ are yet to be proved.

Nonetheless, the application of parallel matching for regular expression only uses a disjunction of vectors, (*i.e.* $R = R_1 + R_2 \mid \langle r_0, \ldots, r_{p-1} \rangle \mid \emptyset \mid \epsilon$) which makes the current proof sufficient for this application.

Case $R = \emptyset$: $L(R) = L(\emptyset) = \emptyset$

$$A = Sync(BK^p(Dsync(\emptyset)))$$

$$A = Sync(BK^p(Dsn(annot(\emptyset)))) \quad [\ def.\ Dsync\ ]$$

$$A = Sync(BK^p(Dsn(\emptyset))) \quad [\ def.\ annot\ ]$$

$$A = Syn(BK^p(\langle \emptyset, \ldots, \emptyset \rangle)) \quad [\ def.\ dsn\ ]$$

$$A = Syn(\langle \begin{pmatrix} Q = q_I \\ \Sigma = \emptyset \\ \delta = \emptyset \\ I = \{q_I\} \\ F = \{\} \end{pmatrix}, \ldots, \begin{pmatrix} Q = q_I \\ \Sigma = \emptyset \\ \delta = \emptyset \\ I = \{q_I\} \\ F = \{\} \end{pmatrix} \rangle) \quad [\ \text{From } A_\emptyset\ ]$$

$$A = \begin{pmatrix} \langle Q^i \rangle_{i\in[p]} = \prod_{i=0}^{p-1} \{q_I^i\} \\ \Sigma = \emptyset \\ \langle \delta^i \rangle_{i\in[p]} = \emptyset \\ \langle I^i \rangle_{i\in[p]} = \prod_{i=0}^{p-1} \{q_I^i\} \\ \langle F^i \rangle_{i\in[p]} = \emptyset \\ \Delta = \emptyset \end{pmatrix} \quad [\ def.\ Sync\ ]$$

$$F = \emptyset \implies L(A) = \emptyset = L(R)$$

Remark 1.1. The BSPRE $\emptyset$ is different from $\langle \emptyset, \ldots, \emptyset \rangle$. In the latter case, the BSPA outputted would have $2p$ states with p states in I, no final states and a delta transition from I to the other states. This case will be covered in $R =< r_0, \ldots, r_{p-1} >$.

Case $R = \epsilon$: $L(R) = L(\epsilon) = \epsilon$

$$A = Sync(BK^p(Dsync(\epsilon)))$$

$$A = Sync(BK^p(Dsn(annot(\epsilon)))) \quad [\ def.\ Dsync\ ]$$

$$A = Sync(BK^p(Dsn(\epsilon))) \quad [\ def.\ annot\ ]$$

$$A = Syn(BK^p(\langle \epsilon, \ldots, \epsilon \rangle)) \quad [\ def.\ dsn\ ]$$

$$A = Syn(\langle \begin{pmatrix} Q = q_I \\ \Sigma = \emptyset \\ \delta = \emptyset \\ I = \{q_I\} \\ F = \{q_I\} \end{pmatrix}, \ldots, \begin{pmatrix} Q = q_I \\ \Sigma = \emptyset \\ \delta = \emptyset \\ I = \{q_I\} \\ F = \{q_I\} \end{pmatrix} \rangle) \quad [\ \text{From } A_\epsilon\ ]$$

$$A = \begin{pmatrix} \langle Q^i \rangle_{i\in[p]} = \prod_{i=0}^{p-1}(\{q_I^i\}) \\ \Sigma = \emptyset \\ \langle \delta^i \rangle_{i\in[p]} = \emptyset \\ \langle I^i \rangle_{i\in[p]} = \prod_{i=0}^{p-1}(\{q_I^i\}) \\ \langle F^i \rangle_{i\in[p]} = \prod_{i=0}^{p-1}(\{q_I^i\}) \\ \Delta = \emptyset \end{pmatrix} \quad [\ def.\ Sync\ ]$$

$$\langle I^i \rangle_{i\in[p]} = \langle F^i \rangle_{i\in[p]} \implies L(A) = \{\epsilon\} = L(R)$$

Remark 1.2. The BSPRE ϵ is different from $\langle \epsilon, \ldots, \epsilon \rangle$. In the latter case, the BSPA outputted would have $2p$ states with p states in I, p states in F and a delta transition from I to F. This case will be covered in $R =< r_0, \ldots, r_{p-1} >$.

Case $R = \langle r_0, \ldots, r_{p-1} \rangle$: $L(R) = L(r_0) \times \cdots \times L(r_{p-1})$

$$
\begin{aligned}
A =& Sync(BK^p(Dsync(\langle r_0, \ldots, r_{p-1}\rangle))) \\
A =& Sync(BK^p(Dsn(annot(\langle r_0, \ldots, r_{p-1}\rangle)))) \qquad [\ def.\ Dsync\ ] \\
A =& Sync(BK^p(Dsn(\langle r_0, \ldots, r_{p-1}\rangle_0))) \qquad [\ def.\ annot\ ] \\
A =& Sync(BK^p(\langle r_0{\cdot};_0, \ldots, r_{p-1}{\cdot};_0 \rangle)) \qquad [\ def.\ dsn\ ]
\end{aligned}
$$

$$
A = Sync(\prod_{i=0}^{p-1}
\begin{pmatrix}
Q = \{q_I\} \cup pos(0 \bullet posex(r_i)) \cup \{(1, ;_0)\} \\
\Sigma = \left\{ snd(x) \mid x \in pos\left(0 \bullet posex(r_i)\right) \right\} \cup \{;_0\} \\
\delta = \begin{cases}
\forall a \in \left\{ snd(x) \mid x \in pos\left(0 \bullet posex(r_i)\right) \right\}, \\
\quad \delta(q_I, a) = \left\{ x \;\middle|\; \begin{array}{l} x \in First(0 \bullet posex(r_i)) \\ \wedge\ snd(x) = a \end{array} \right\} \\
\delta(q_I, ;_0) = \{(1, ;_0)\} \cdot Null(0 \bullet posex(r_i)) \\
\\
\forall x \in pos(0 \bullet posex(r_i)), \\
\quad \forall a \in \left\{ snd(x) \mid x \in pos\left(0 \bullet posex(r_i)\right) \right\}, \\
\quad\quad \delta(x, a) = \left\{ y \;\middle|\; \begin{array}{l} y \in Follow(0 \bullet posex(r_i), x) \\ \wedge ; snd(y) = a \end{array} \right\} \\
\\
\forall x \in Last\left(0 \bullet posex(r_i)\right), \\
\quad \delta(x, ;_0) = \{(1, ;_0)\}
\end{cases} \\
I = \{q_I\} \\
F = \{(1, ;_0)\}
\end{pmatrix}
\qquad [\ \text{From } A_\bullet\ ]
$$

$$A = \left(\begin{array}{rl} \langle Q^i \rangle_{i\in[p]} & = \bigcup_{i=0}^{p-1} \left(\{q_I^i\} \cup pos(0 \bullet posex(r_i)) \right) \\ \Sigma & = \bigcup_{i=0}^{p-1} \left\{ snd(x) \mid x \in pos\left(0 \bullet posex(r_i)\right) \right\} \\ \langle \delta^i \rangle_{i\in[p]} & = \bigcup_{i=0}^{p-1} \left\{ \begin{array}{l} \forall a \in \left\{ snd(x) \mid x \in pos\left(0 \bullet posex(r_i)\right) \right\}, \\ \delta(q_I^i, a) = \left\{ x \left| \begin{array}{l} x \in First(0 \bullet posex(r_i)) \\ \wedge\ snd(x) = a \end{array} \right. \right\} \\ \forall x \in pos(0 \bullet posex(r_i)), \\ \quad \forall a \in \left\{ snd(x) \mid x \in pos\left(0 \bullet posex(r_i)\right) \right\}, \\ \quad \delta(x, a) = \left\{ y \left| \begin{array}{l} y \in Follow\left(0 \bullet posex(r_i), x\right) \\ \wedge\ snd(y) = a \end{array} \right. \right\} \end{array} \right. \\ \langle I^i \rangle_{i\in[p]} & = \bigcup_{i=0}^{p-1} \{q_I^i\} \\ \langle F^i \rangle_{i\in[p]} & = \bigcup_{i=0}^{p-1} \{(1, ;_0^i)\} \\ \Delta & = \left\{ (\vec{q} \to \langle (1, ;_0), \ldots, (1, ;_0) \rangle) \mid \vec{q} \in \prod_{i=0}^{p-1} Last(0 \bullet posex(r_i)) \right\} \end{array} \right) \quad [\ def.\ Sync\ ]$$

$$L(R) = L(A) \iff \left(\forall w \in ((\Sigma^*)^p)^*, w \in L(R) \iff w \in L(A)\right)$$

- $w = \epsilon$

  $\epsilon \in L(A) \iff \langle I^i \rangle_{i\in[p]} \subseteq \langle F^i \rangle_{i\in[p]} \iff \bot$

  $\epsilon \in L(R) \iff \epsilon \in \left(L(r_0) \times \cdots \times L(r_{p-1})\right) \iff \bot$

- $w = \langle w_0, \ldots, w_{p-1} \rangle \in L(A)$

$$\iff \Delta(\langle \delta^*(q_I^0, w^0), \ldots, \delta^*(q_I^{p-1}, w^{p-1}) \rangle) \in \langle F^i \rangle_{i\in[p]}$$

$$\iff \langle \delta^*(q_I^0, w^0), \ldots, \delta^*(q_I^{p-1}, w^{p-1}) \rangle \in \prod_{i=0}^{p-1} Last(0 \bullet posex(r_i)) \qquad [\ \text{From } \Delta\ ]$$

Let $u_0^i \ldots u_{n_i}^i = w^i$

$$\iff \prod_{i=0}^{p-1} \left\{ \begin{array}{rl} x_0^i = \delta(q_I^i, u_0^i) & \in First(0 \bullet posex(r_i)) \qquad [\ \text{From } \langle \delta^i \rangle_{i\in[p]}\ ] \\ \forall j \in \{1 \ldots n_i\}, x_j^i = \delta(x_{j-1}^i, u_j^i) & \in Follow\left(0 \bullet posex(r_i), x_{j-1}^i\right) \qquad [\ \text{From } \langle \delta^i \rangle_{i\in[p]}\ ] \\ x_{n_i} & \in Last(0 \bullet posex(r_i)) \end{array} \right.$$

$$\iff \prod_{i=0}^{p-1} \begin{cases} \forall j \in \{1 \dots n_i\} & \\ \qquad \exists w_0^{i\prime}, & x_0^i w_0^{i\prime} \in L(0 \bullet posex(r_i)), \quad [\ \textit{prop. First}\ ] \\ \exists w_{j-1}^i, \exists w_j^{i\prime}, & w_{j-1}^i x_{j-1}^i x_j^i w_j^{i\prime} \in L(0 \bullet posex(r_i)), \quad [\ \textit{prop. Follow}\ ] \\ \qquad \exists w_{n-1}^i, & w_{n_i-1}^i x_{n_i-1}^i \in L(0 \bullet posex(r_i)), \quad [\ \textit{prop. Last}\ ] \\ snd^*(x_0^i \dots x_{n_i-1}^i) = u_0^i \dots u_{n_i}^i & \end{cases}$$

$$\iff \prod_{i=0}^{p-1} \begin{cases} \text{For } x_0^i \dots x_{n_i-1}^i \text{ to be the word constrained by all definitions above, we have} \\ \forall j \in \{0 \dots n_i\}, \quad \begin{bmatrix} w_j^i = x_0^i \dots x_{j-1}^i \\ w_j^{i\prime} = x_{j+1}^i \dots x_{n_i-1}^i \end{bmatrix} \\ \text{So, we also have, } \forall i \in \{0 \dots p-1\}, \quad \begin{bmatrix} x_0^i \dots x_{n_i-1}^i \in L(0 \bullet posex(r_i)) \\ snd^*(x_0^i \dots x_{n_i-1}^i) = u_0^i \dots u_{n_i}^i \end{bmatrix} \end{cases}$$

$$\iff \forall i \in \{0 \dots p-1\}, u_0^i \dots u_{n_i}^i \in L(r_i) \qquad [\ \textit{lem. posex}^{-1}\ ]$$
$$\iff \langle w_0, \dots, w_{p-1} \rangle \in \big(L(r_0) \times \dots \times L(r_{p-1})\big)$$
$$\iff \langle w_0, \dots, w_{p-1} \rangle \in L(R)$$

- $w = \langle w_0, \dots w_{p-1} \rangle \cdot \langle w'_0, \dots w'_{p-1} \rangle \cdot w' \in L(A)$

  To be accepted by A, at least two applications of delta are needed. The first application necessarily leads to the state vector $\langle (1, ;_0), \dots, (1, ;_0) \rangle$ (see Δ). Then, there is no δ-transition coming from a state $(1, ;_0)^i$ or Δ-transition from this state vector. Hence, $w \notin L(A)$.

  Concerning the belonging of w in $L(R) = L(r_0) \times \dots \times L(r_{p-1})$, since the structure is not the same, $w \notin L(R)$.

  Is deduced $\forall w \in ((\Sigma^*)^p)^*, w \in L(R) \iff w \in L(A)$, followed by $L(R) = L(A)$.

Case $R = R_1 + R_2$ $\qquad L(R) = L(R_1) \cup L(R_2)$

$$
\begin{aligned}
A &= Sync(BK^p(Dsync(R_1 + R_2))) \\
A &= Sync(BK^p(Dsn(annot(R_1 + R_2)))) && [\ def.\ Dsync\ ] \\
A &= Sync(BK^p(Dsn(R_1^\circ + R_2^\circ))) && [\ def.\ annot\ ] \\
A &= Sync(BK^p(\langle dsn^0(R_1^\circ + R_2^\circ), \ldots, dsn^{p-1}(R_1^\circ + R_2^\circ)\rangle)) && [\ def.\ Dsn\ ] \\
A &= Sync(BK^p(\langle dsn^0(R_1^\circ) + dsn^0(R_2^\circ), \ldots, dsn^{p-1}(R_1^\circ) + dsn^{p-1}(R_2^\circ)\rangle)) && [\ def.\ dsn\ ]
\end{aligned}
$$

$$
A = Sync(\prod_{i=0}^{p-1}
\left(
\begin{array}{l}
Q = \{q_I^i\} \cup pos(0 \bullet posex(dsn^i(R_1^\circ))) \cup pos(1 \bullet posex(dsn^i(R_2^\circ))) \\
\Sigma_{dsn} = \left\{
\begin{array}{l}
\left\{ snd(x^i) \mid x^i \in pos(0 \bullet posex(dsn^i(R_1^\circ))) \right\} \\
\cup \left\{ snd(x^i) \mid x^i \in pos(1 \bullet posex(dsn^i(R_2^\circ))) \right\}
\end{array}
\right. \\
\delta_{dsn} = \left\{
\begin{array}{l}
\forall a \in \left\{ snd(x^i) \mid x^i \in pos(0 \bullet posex(dsn^i(R_1^\circ))) \right\}, \\
\quad \delta(q_I^i, a) = \left\{ x^i \;\middle|\; \begin{array}{l} x^i \in First(0 \bullet posex(dsn^i(R_1^\circ))) \\ \wedge\ snd(x^i) = a \end{array} \right\} \\
\forall a \in \left\{ snd(x^i) \mid x^i \in pos(1 \bullet posex(dsn^i(R_2^\circ))) \right\}, \\
\quad \delta(q_I^i, a) = \left\{ x^i \;\middle|\; \begin{array}{l} x^i \in First(1 \bullet posex(dsn^i(R_2^\circ))) \\ \wedge\ snd(x^i) = a \end{array} \right\} \\
\\
\forall x^i \in pos(0 \bullet posex(dsn^i(R_1^\circ))), \\
\quad \forall a \in \left\{ snd(x^i) \mid x^i \in pos(0 \bullet posex(dsn^i(R_1^\circ))) \right\}, \\
\quad\quad \delta(x^i, a) = \left\{ y^i \;\middle|\; \begin{array}{l} y^i \in Follow(0 \bullet posex(dsn^i(R_1^\circ)), x^i) \\ \wedge\ snd(y^i) = a \end{array} \right\} \\
\\
\forall x^i \in pos(1 \bullet posex(dsn^i(R_2^\circ))), \\
\quad \forall a \in \left\{ snd(x^i) \mid x^i \in pos(1 \bullet posex(dsn^i(R_2^\circ))) \right\}, \\
\quad\quad \delta(x^i, a) = \left\{ y^i \;\middle|\; \begin{array}{l} y^i \in Follow(1 \bullet posex(dsn^i(R_2^\circ)), x^i) \\ \wedge\ snd(y^i) = a \end{array} \right\}
\end{array}
\right. \\
I = \{q_I^i\} \\
F = \left\{
\begin{array}{l}
\textbf{if } Null(0 \bullet posex(dsn^i(R_1^\circ))) \cup Null(1 \bullet posex(dsn^i(R_2^\circ))) = \{\epsilon\} \\
\quad \{q_I^i\} \cup Last(0 \bullet posex(dsn^i(R_1^\circ))) \cup Last(1 \bullet posex(dsn^i(R_2^\circ))) \\
\textbf{otherwise } Last(0 \bullet posex(dsn^i(R_1^\circ))) \cup Last(1 \bullet posex(dsn^i(R_2^\circ)))
\end{array}
\right.
\end{array}
\right)
$$

[From A_+]

$$
A = \left(\begin{array}{l}
\langle Q^i \rangle_{i\in[p]} = \bigcup_{i=0}^{p-1} \Big(\{q_I^i\} \cup pos(0 \bullet posex(dsn^i(R_1^\circ))) \cup pos(1 \bullet posex(dsn^i(R_2^\circ))) \Big) \\
\Sigma = \left(\begin{array}{l} \bigcup_{i=0}^{p-1} \Big\{ snd(x^i) \mid x^i \in pos(0 \bullet posex(dsn^i(R_1^\circ))) \Big\} \\ \cup \Big\{ snd(x^i) \mid x^i \in pos(1 \bullet posex(dsn^i(R_2^\circ))) \Big\} \end{array} \right) \setminus \mathscr{S} \\
\langle \delta^i \rangle_{i\in[p]} = \bigcup_{i=0}^{p-1} \left(\begin{array}{l}
\forall a \in \Big\{ snd(x^i) \mid x^i \in pos(0 \bullet posex(dsn^i(R_1^\circ))) \Big\}, \\
\quad \delta(q_I^i, a) = \left\{ x^i \;\middle|\; \begin{array}{l} x^i \in First(0 \bullet posex(dsn^i(R_1^\circ))) \\ \wedge\ snd(x^i) = a \end{array} \right\} \\
\forall a \in \Big\{ snd(x^i) \mid x^i \in pos(1 \bullet posex(dsn^i(R_2^\circ))) \Big\}, \\
\quad \delta(q_I^i, a) = \left\{ x^i \;\middle|\; \begin{array}{l} x^i \in First(1 \bullet posex(dsn^i(R_2^\circ))) \\ \wedge\ snd(x^i) = a \end{array} \right\} \\
\\
\forall x^i \in pos(0 \bullet posex(dsn^i(R_1^\circ))), \\
\quad \forall a \in \Big\{ snd(x^i) \mid x^i \in pos(0 \bullet posex(dsn^i(R_1^\circ))) \Big\}, \\
\quad\quad \delta(x^i, a) = \left\{ y^i \;\middle|\; \begin{array}{l} y^i \in Follow\big(0 \bullet posex(dsn^i(R_1^\circ)), x^i\big) \\ \wedge\ snd(y^i) = a \end{array} \right\} \\
\\
\forall x^i \in pos(1 \bullet posex(dsn^i(R_2^\circ))), \\
\quad \forall a \in \Big\{ snd(x^i) \mid x^i \in pos(1 \bullet posex(dsn^i(R_2^\circ))) \Big\}, \\
\quad\quad \delta(x^i, a) = \left\{ y^i \;\middle|\; \begin{array}{l} y^i \in Follow\big(1 \bullet posex(dsn^i(R_2^\circ)), x^i\big) \\ \wedge\ snd(y^i) = a \end{array} \right\}
\end{array} \right) \setminus \bigcup_{;_t \in \mathscr{S}} \big((q, ;_t) \to q' \big) \\
\langle I^i \rangle_{i\in[p]} = \bigcup_{i=0}^{p-1} \{q_I^i\} \\
\langle F^i \rangle_{i\in[p]} = \bigcup_{i=0}^{p-1} \left\{ \begin{array}{l} \textbf{if } Null(0 \bullet posex(dsn^i(R_1^\circ))) \cup Null(1 \bullet posex(dsn^i(R_2^\circ))) = \{\epsilon\} \\ \qquad \{q_I^i\} \cup Last(0 \bullet posex(dsn^i(R_1^\circ))) \cup Last(1 \bullet posex(dsn^i(R_2^\circ))) \\ \textbf{otherwise } Last(0 \bullet posex(dsn^i(R_1^\circ))) \cup Last(1 \bullet posex(dsn^i(R_2^\circ))) \end{array} \right. \\
\Delta = \forall ;_t \in \mathscr{S}, \left\{ \begin{array}{l} \forall \vec{q_1} \in \prod_{i=0}^{p-1} \Big\{ q_1^i \mid \delta^i(q_1^i, ;_t) = q_2^i \Big\}, \\ \forall \vec{q_2} \in \prod_{i=0}^{p-1} \Big\{ q_2^i \mid \delta^i(q_1^i, ;_t) = q_2^i \Big\}, \end{array} \right\} \Delta(\vec{q_1}) = \vec{q_2} \\
\quad \big[\textit{def. Sync} \big]
\end{array} \right)
$$

$$
A = \left(\begin{array}{rl}
\langle Q^i \rangle_{i\in[p]} = & \bigcup_{i=0}^{p-1} \Big(\{q_I^i\} \cup pos(0 \bullet posex(dsn^i(R_1^\circ))) \cup pos(1 \bullet posex(dsn^i(R_2^\circ))) \Big) \\
\Sigma = & \bigcup_{i=0}^{p-1} \left(\begin{array}{l} \Big\{ snd(x^i) \mid x^i \in pos(0 \bullet posex(dsn^i(R_1^\circ))) \Big\} \setminus \mathscr{S} \\ \cup \Big\{ snd(x^i) \mid x^i \in pos(1 \bullet posex(dsn^i(R_2^\circ))) \Big\} \setminus \mathscr{S} \end{array} \right) \\
\langle \delta^i \rangle_{i\in[p]} = & \bigcup_{i=0}^{p-1} \left\{ \begin{array}{ll}
\forall a \in \Big\{ snd(x^i) \mid x^i \in pos(0 \bullet posex(dsn^i(R_1^\circ))) \Big\} \setminus \mathscr{S}, & \\
\quad \delta(q_I^i, a) = \left\{ x^i \;\middle|\; \begin{array}{l} x^i \in First(0 \bullet posex(dsn^i(R_1^\circ))) \\ \wedge\ snd(x^i) = a \end{array} \right\} & [\ \delta_1\ ] \\
\forall a \in \Big\{ snd(x^i) \mid x^i \in pos(1 \bullet posex(dsn^i(R_2^\circ))) \Big\} \setminus \mathscr{S}, & \\
\quad \delta(q_I^i, a) = \left\{ x^i \;\middle|\; \begin{array}{l} x^i \in First(1 \bullet posex(dsn^i(R_2^\circ))) \\ \wedge\ snd(x^i) = a \end{array} \right\} & [\ \delta_2\ ] \\
\\
\forall x^i \in pos(0 \bullet posex(dsn^i(R_1^\circ))), & \\
\quad \forall a \in \Big\{ snd(x^i) \mid x^i \in pos(0 \bullet posex(dsn^i(R_1^\circ))) \Big\} \setminus \mathscr{S}, & \\
\quad\quad \delta(x^i, a) = \left\{ y^i \;\middle|\; \begin{array}{l} y^i \in Follow(0 \bullet posex(dsn^i(R_1^\circ)), x^i) \\ \wedge\ snd(y^i) = a \end{array} \right\} & [\ \delta_3\ ] \\
\\
\forall x^i \in pos(1 \bullet posex(dsn^i(R_2^\circ))), & \\
\quad \forall a \in \Big\{ snd(x^i) \mid x^i \in pos(1 \bullet posex(dsn^i(R_2^\circ))) \Big\} \setminus \mathscr{S}, & \\
\quad\quad \delta(x^i, a) = \left\{ y^i \;\middle|\; \begin{array}{l} y^i \in Follow(1 \bullet posex(dsn^i(R_2^\circ)), x^i) \\ \wedge\ snd(y^i) = a \end{array} \right\} & [\ \delta_4\ ]
\end{array} \right. \\
\langle I^i \rangle_{i\in[p]} = & \bigcup_{i=0}^{p-1} \{q_I^i\} \\
\langle F^i \rangle_{i\in[p]} = & \bigcup_{i=0}^{p-1} \left\{ \begin{array}{l}
\textbf{if } Null(0 \bullet posex(dsn^i(R_1^\circ))) \cup Null(1 \bullet posex(dsn^i(R_2^\circ))) = \{\epsilon\} \\
\qquad \{q_I^i\} \cup Last(0 \bullet posex(dsn^i(R_1^\circ))) \cup Last(1 \bullet posex(dsn^i(R_2^\circ))) \\
\textbf{otherwise } Last(0 \bullet posex(dsn^i(R_1^\circ))) \cup Last(1 \bullet posex(dsn^i(R_2^\circ)))
\end{array} \right. \\
\Delta = & \forall ;_t \in \mathscr{S}, \left\{ \begin{array}{l} \forall \vec{q_1} \in \prod_{i=0}^{p-1} \Big\{ q_1^i \mid \delta^i(q_1^i, ;_t) = q_2^i \Big\}, \\ \forall \vec{q_2} \in \prod_{i=0}^{p-1} \Big\{ q_2^i \mid \delta^i(q_1^i, ;_t) = q_2^i \Big\}, \end{array} \right\} \Delta(\vec{q_1}) = \vec{q_2}
\end{array} \right)
$$

$L(A) = L(R) \iff \big(\forall w \in \Sigma^*, w \in L(A) \iff w \in L(R)\big)$

Distinct cases on $|w|$ are considered

- $|w| = 0 \to w = \epsilon$

$\epsilon \in L(A)$
$\iff \forall i \in [p], q_I \in F^i$
$\iff \forall i \in [p], Null(0 \bullet posex(dsn^i(R_1^\circ))) \cup Null(1 \bullet posex(dsn^i(R_2^\circ))) = \{\epsilon\}$ [$def\ F, q_I \notin last$]
$\iff \forall i \in [p], \epsilon \in \big(L(0 \bullet posex(dsn^i(R_1^\circ))) \cup L(1 \bullet posex(dsn^i(R_2^\circ)))\big)$ [*prop. Null*]
$\iff \forall i \in [p], \epsilon \in L\big((0 \bullet posex(dsn^i(R_1^\circ))) + (1 \bullet posex(dsn^i(R_2^\circ)))\big)$ [*def.* L_{RE}]
$\iff \forall i \in [p], \epsilon \in L(posex(dsn^i(R_1^\circ)) + dsn^i(R_2^\circ)))$ [*def. posex*]
$\iff \forall i \in [p], \epsilon \in L(posex(dsn^i(R_1^\circ + R_2^\circ)))$ [*def. dsn*]
$\iff \forall i \in [p], \epsilon \in L(dsn^i(R_1^\circ + R_2^\circ))$ [*rmk.* $\epsilon \in L(posex^{-1})$]
$\iff \epsilon \in L(R_1^\circ + R_2^\circ)$ [*lem.* dsn^{-1}]
$\iff \epsilon \in L(R_1 + R_2)$ [*lem.* $annot^{-1}$]
$\iff \epsilon \in L(R)$ [$R = R_1 + R_2$]

- $|w| = 1, w = \langle u_0^0 \dots u_{n^0-1}^0, \dots\dots\dots, u_0^{p-1} \dots u_{n^{p-1}-1}^{p-1} \rangle \in L(A)$
  $\iff \forall i \in [p], \exists x^i_{n^i-1} \in \delta^{i*}(q_I^i, u_0^i \dots u_{n^i-1}^i), \begin{cases} \Delta\langle x^0_{n^0-1}, \dots, x^{p-1}_{n^{p-1}-1}\rangle = \langle x^0_{n^0}, \dots, x^{p-1}_{n^{p-1}}\rangle \\ \wedge\ x^i_{n^i} \in F^i \end{cases}$

  Different cases are considered according to membership of each word's first symbol of the word vector w.

  $\forall i \in [p]$,

  ▷ **if** $n^i = 0$ (this case is treated here because $x^i_{n^i}$ belong to *First* and not *Follow* as in the following cases)

  $\iff u_0^i \dots u_{n^i-1}^i = \epsilon$
  $\iff \exists\, x^i_{n^i}, \Delta(\langle \dots, q_I^i, \dots\rangle = \langle \dots, x^i_{n^i}, \dots\rangle \wedge x^i_{n^i} \in F^i$
  $\iff \exists ;_t \in \mathscr{S}, \exists\, x^i_{n^i},$

  $$\begin{cases} x^i_{n^i} \in \left\{ q_2^i \mid \delta^i_{dsn}(q_I^i, ;_t) = q_2^i \right\} & [\ [\Delta]\ ] \\ x^i_{n^i} \in Last(0 \bullet posex(dsn^i(R_1^\circ))) \cup Last(1 \bullet posex(dsn^i(R_2^\circ))), & [\ x^i_{n^i} \in F, x^i_{n^i} \neq q_I\ ] \end{cases}$$

  $\iff \exists ;_t \in \mathscr{S}, \exists\, x^i_{n^i},$

  $$\begin{cases} x^i_{n^i} \in First(0 \bullet posex(dsn^i(R_1^\circ))) \cup First(1 \bullet posex(dsn^i(R_2^\circ))) & [\ \delta^i \text{ from } q_I\ ] \\ snd(x^i_{n^i}) = ;_t & [\ \delta^i\ ] \\ x^i_{n^i} \in Last(0 \bullet posex(dsn^i(R_1^\circ))) \cup Last(1 \bullet posex(dsn^i(R_2^\circ))) & [\ x^i_{n^i} \in F, x^i_{n^i} \neq q_I\ ] \end{cases}$$

  $\iff \exists ;_t \in \mathscr{S}, \exists\, x^i_{n^i},$

$$\begin{cases} \exists v^i,\ x^i_{n^i} v^i \in L(0 \bullet posex(dsn^i(R_1^\circ))) \cup L(1 \bullet posex(dsn^i(R_2^\circ))) & [\ \textit{prop. First}\ ] \\ \exists w^i, w^i x^i_{n^i} \in L(0 \bullet posex(dsn^i(R_1^\circ))) \cup L(1 \bullet posex(dsn^i(R_2^\circ))) & [\ \textit{prop. Last}\ ] \\ snd^*(x^i_{n^i}) = ;_t \end{cases}$$

$$\iff \exists ;_t \in \mathscr{S}, \exists\, x^i_{n^i},$$

$$\begin{cases} x^i_{n^i} \in L(0 \bullet posex(dsn^i(R_1^\circ))) \cup L(1 \bullet posex(dsn^i(R_2^\circ))) & \begin{bmatrix} v^i = \epsilon \\ w^i = \epsilon \end{bmatrix} \\ snd^*(x^i_{n^i}) = ;_t \end{cases}$$

▷ **if** $n^i > 0 \wedge u^i_0 \in \left\{ snd(x^i) \mid x^i \in pos(0 \bullet posex(dsn^i(R_1^\circ))) \right\} \setminus \mathscr{S}$

$$\iff \begin{cases} \forall j \in \{1 \dots n_i - 1\}, & \\ \quad \exists x^i_0 \in First(0 \bullet posex(dsn^i(R_1^\circ))), & [\ [\delta_1], u^i_0 \text{ membership}\ ] \\ \quad \exists x^i_j \in Follow(0 \bullet posex(dsn^i(R_1^\circ)), x^i_{j-1}), & [\ [\delta_2], x^i_{j-1} \text{ membership}\ ] \\ snd^*(x^i_0 \dots x^i_{n_i-1}) = u^i_0 \dots u^i_{n_i-1}, & \\ \exists ;_t \in \mathscr{S}, & \\ x^i_{n^i-1} \in \left\{ q^i_1 \mid \delta^i_{dsn}(q^i_1, ;_t) = q^i_2 \right\} & [\ [\Delta]\ ] \\ \quad x^i_{n^i} \in \left\{ q^i_2 \mid \delta^i_{dsn}(q^i_1, ;_t) = q^i_2 \right\} & [\ [\Delta]\ ] \\ \quad x^i_{n^i} \in Last(0 \bullet posex(dsn^i(R_1^\circ))), & [\ x^i_{n^i} \in F, x^i_{n^i} \neq q_I\ ] \end{cases}$$

$$\iff \begin{cases} \forall j \in \{1 \dots n_i - 1\}, & \\ \exists x^i_0 \in First(0 \bullet posex(dsn^i(R_1^\circ))), & \\ \exists x^i_j \in Follow(0 \bullet posex(dsn^i(R_1^\circ)), x^i_{j-1}), & \\ \quad x^i_{n^i} \in Follow(0 \bullet posex(dsn^i(R_1^\circ)), x^i_{n^i-1}), & [\ [\Delta], \text{case } \textit{First} \text{ already treated}\ ] \\ \quad x^i_{n^i} \in Last(0 \bullet posex(dsn^i(R_1^\circ))), & \\ \exists ;_t \in \mathscr{S}, \quad snd(x^i_{n^i}) = ;_t & \\ snd^*(x^i_0 \dots x^i_{n_i-1}) = u^i_0 \dots u^i_{n_i-1} & \end{cases}$$

$$\iff \begin{cases} \forall j \in \{1 \dots n_i - 1\}, & & & \\ \qquad \exists x^i_0, \exists v^i_0, & x^i_0 v^i_0 & \in L(0 \bullet posex(dsn^i(R_1^\circ))), & [\ \textit{prop. First}\ ] \\ \exists x^i_j, \exists w^i_{j-1}, \exists v^i_j, & w^i_{j-1} x^i_{j-1} x^i_j v^i_j & \in L(0 \bullet posex(dsn^i(R_1^\circ))), & [\ \textit{prop. Follow}\ ] \\ \exists x^i_{n^i}, \exists w^i_{n^i-1}, \exists v^i_{n^i}, & w^i_{n^i-1} x^i_{n^i-1} x^i_{n^i} v^i_{n^i} & \in L(0 \bullet posex(dsn^i(R_1^\circ))), & [\ \textit{prop. Follow}\ ] \\ \qquad \exists w^i_{n^i}, & w^i_{n^i} x^i_{n^i} & \in L(0 \bullet posex(dsn^i(R_1^\circ))), & [\ \textit{prop. Last}\ ] \\ \exists ;_t \in \mathscr{S},\ snd(x^i_{n^i}) = ;_t & & & \\ snd^*(x^i_0 \dots x^i_{n-1}) = u^i_0 \dots u^i_{n_i-1} & & & \end{cases}$$

For $x_0 \dots x_{n^i}$ to be the word constrained by all definitions above, we have

$$\forall j \in \{0 \dots n_i\}, \begin{bmatrix} w^i_j = x^i_0 \dots x^i_{j-1} \\ v^i_j = x^i_{j+1} \dots x^i_{n_i} \end{bmatrix}$$

So, we also have, $\exists\, ;_t \in \mathscr{S}, \exists x^i_0 \dots x^i_{n_i}, \begin{bmatrix} x^i_0 \dots x^i_{n_i} \in L(0 \bullet posex(dsn^i(R^\circ_1))) \\ snd(x^i_0 \dots x^i_{n_i}) = u^i_0 \dots u^i_{n_i-1}\, ;_t \end{bmatrix}$

▷ **if** $n^i > 0 \wedge u^i_0 \in \left\{ snd(x^i) \mid x^i \in pos(1 \bullet posex(dsn^i(R^\circ_2))) \right\} \setminus \mathscr{S}$,
For the same symmetric reasons, we have

$$\exists\, ;_t \in \mathscr{S}, \exists x^i_0 \dots x^i_{n_i}, \begin{bmatrix} x^i_0 \dots x^i_{n_i} \in L(1 \bullet posex(dsn^i(R^\circ_2))) \\ snd(x^i_0 \dots x^i_{n_i}) = u^i_0 \dots u^i_{n_i-1}\, ;_t \end{bmatrix}$$

▷ **Hence**, considering the global language of the BSPA, taking a Δ-transition entails that every desynchronized local transitions take the same label '$;_t$'. So it is not $\forall i \in [p], \exists\, ;_t \in \mathscr{S}, \dots$ but $\exists\, ;_t \in \mathscr{S}, \forall i \in [p], \dots$.

$$\begin{aligned}
&\iff \exists\, ;_t \in \mathscr{S}, \forall i \in [p], \exists x^i_0 \dots x^i_{n_i-1}, \\
&\qquad \begin{cases} x^i_0 \dots x^i_{n_i} \in \left(L(0 \bullet posex(dsn^i(R^\circ_1))) \cup L(1 \bullet posex(dsn^i(R^\circ_2)))\right) \\ \wedge \quad snd^*(x^i_0 \dots x^i_{n_i}) = u^i_0 \dots u^i_{n_i-1}\, ;_t \end{cases} \\
&\iff \exists\, ;_t \in \mathscr{S}, \forall i \in [p], \exists x^i_0 \dots x^i_{n_i-1}, \\
&\qquad \begin{cases} x^i_0 \dots x^i_{n_i} \in L\big(0 \bullet posex(dsn^i(R^\circ_1)) + (1 \bullet posex(dsn^i(R^\circ_2)))\big) \qquad [\, def.\ L_{RE}\,] \\ \wedge \quad snd^*(x^i_0 \dots x^i_{n_i}) = u^i_0 \dots u^i_{n_i-1}\, ;_t \end{cases} \\
&\iff \exists\, ;_t \in \mathscr{S}, \forall i \in [p], \exists x^i_0 \dots x^i_{n_i-1}, \\
&\qquad \begin{cases} x^i_0 \dots x^i_{n_i-1} \in L\big(posex(dsn^i(R^\circ_1) + dsn^i(R^\circ_2))\big) \qquad [\, def.\ posex\,] \\ \wedge \quad snd^*(x^i_0 \dots x^i_{n_i}) = u^i_0 \dots u^i_{n_i-1}\, ;_t \end{cases} \\
&\iff \exists\, ;_t \in \mathscr{S}, \forall i \in [p], \exists x^i_0 \dots x^i_{n_i-1}, \\
&\qquad \begin{cases} x^i_0 \dots x^i_{n_i-1} \in L\big(posex(dsn^i(R^\circ_1 + R^\circ_2))\big) \qquad [\, def.\ dsn\,] \\ \wedge \quad snd^*(x^i_0 \dots x^i_{n_i}) = u^i_0 \dots u^i_{n_i-1}\, ;_t \end{cases} \\
&\iff \exists\, ;_t \in \mathscr{S}, \forall i \in [p], u^i_0 \dots u^i_{n_i-1}\, ;_t \in L\big(dsn^i(R^\circ_1 + R^\circ_2)\big) \qquad [\, lem.\ posex^{-1}\,] \\
&\iff \exists\, ;_t \in \mathscr{S}, \langle \dots, u^i_0 \dots u^i_{n_i-1}, \dots \rangle_t \in L\big(R^\circ_1 + R^\circ_2\big) \qquad [\, lem.\ dsn^{-1}\,]
\end{aligned}$$

$$\iff w \in L(R_1 + R_2) \qquad [\ \textit{lem. annot}^{-1}\ ]$$
$$\iff w \in L(R) \qquad [\ R = R_1 + R_2\ ]$$

- $|w| = m$, $m \geq 2$, $w = w_1 \dots w_m$.

  The first BSP vector is explicitly shown for taking initial state into account and differentiating the cases according to the membership of the first letter (as in *thm.* Glushkov preserves language proof) and the last for using final states.

$$\iff w = \langle a_0^0 \dots a_{k^0-1}^0, \dots, a_0^{p-1} \dots a_{k^{p-1}-1}^{p-1} \rangle \dots \langle u_0^0 \dots u_{n^0-1}^0, \dots, u_0^{p-1} \dots u_{n^{p-1}-1}^{p-1} \rangle \in L(A)$$

$$\iff \exists \vec{q_0} \in \left\{ \begin{array}{l} \forall i \in [p], \exists z_{k^i-1}^i \in \delta^{i*}(q_I^i, a_0^i \dots a_{k^i-1}^i),\ \Delta\langle z_{k^0-1}^0, \dots, z_{k^{p-1}-1}^{p-1} \rangle = \langle z_{k^0}^0, \dots, z_{k^{p-1}}^{p-1} \rangle, \\ \forall i \in [p], \exists y_{l^i-1}^i \in \delta^{i*}(z_{k^i}^i, b_0^i \dots b_{l^i-1}^i),\ \Delta\langle y_{l^0-1}^0, \dots, y_{l^{p-1}-1}^{p-1} \rangle = \langle y_{l^0}^0, \dots, y_{l^{p-1}}^{p-1} \rangle, \\ \qquad \vdots \end{array} \right. \Big\uparrow m-1 \Big\downarrow$$
$$, \forall i \in [p], \exists x_{n^i-1}^i \in \delta^{i*}(q_0^i, u_0^i \dots u_{n^i-1}^i),\ \Delta\langle x_{n^0-1}^0, \dots, x_{n^{p-1}-1}^{p-1} \rangle = \langle x_{n^0}^0, \dots, x_{n^{p-1}}^{p-1} \rangle,\ x_{n^i}^i \in F^i$$

▷ **if** $a_0^i \in \left\{ snd(x^i) \mid x^i \in pos(0 \bullet posex(dsn^i(R_1^\circ))) \right\} \setminus \mathscr{S}$

$$\iff \left\{ \begin{array}{ll} \forall j \in \{1 \dots k_i - 1\}, & \\ \quad \exists z_0^i \in First(0 \bullet posex(dsn^i(R_1^\circ))), & [\ [\delta_1], a_0^i \text{ membership}\ ] \\ \quad \exists z_j^i \in Follow(0 \bullet posex(dsn^i(R_1^\circ)), z_{j-1}^i), & [\ [\delta_2], z_{j-1}^i \text{ membership}\ ] \\ snd^*(z_0^i \dots z_{n_i-1}^i) = a_0^i \dots a_{n_i-1}^i, & \\ \exists ;_{t^1} \in \mathscr{S}, & \\ z_{k^i-1}^i \in \left\{ q_1^i \mid \delta_{dsn}^i(q_1^i, ;_{t^1}) = q_2^i \right\} & [\ [\Delta]\ ] \\ \quad z_{k^i}^i \in \left\{ q_2^i \mid \delta_{dsn}^i(q_1^i, ;_{t^1}) = q_2^i \right\} & [\ [\Delta]\ ] \\ \qquad \vdots & \\ \forall j \in \{1 \dots n_i - 1\}, & \\ \quad \exists x_0^i \in Follow(0 \bullet posex(dsn^i(R_1^\circ)), q_0^i), & [\ q_0 \neq q_I, [\delta_2], u_0^i \text{ membership}\ ] \\ \quad \exists x_j^i \in Follow(0 \bullet posex(dsn^i(R_1^\circ)), x_{j-1}^i), & [\ [\delta_2], x_{j-1}^i \text{ membership}\ ] \\ snd^*(x_0^i \dots x_{n_i-1}^i) = u_0^i \dots u_{n_i-1}^i, & \\ \exists ;_{t^m} \in \mathscr{S}, & \\ x_{n^i-1}^i \in \left\{ q_1^i \mid \delta_{dsn}^i(q_1^i, ;_{t^m}) = q_2^i \right\} & [\ [\Delta]\ ] \\ \quad x_{n^i}^i \in \left\{ q_2^i \mid \delta_{dsn}^i(q_1^i, ;_{t^m}) = q_2^i \right\} & [\ [\Delta]\ ] \\ \quad x_{n^i}^i \in Last(0 \bullet posex(dsn^i(R_1^\circ))), & [\ x_{n^i}^i \in F, x_{n^i}^i \neq q_I\ ] \end{array} \right.$$

$$
\iff \begin{cases}
\forall j \in \{1 \dots k_i - 1\}, \\
\exists z^i_0 \in \mathit{First}(0 \bullet \mathit{posex}(\mathit{dsn}^i(R^\circ_1))), \\
\exists z^i_j \in \mathit{Follow}(0 \bullet \mathit{posex}(\mathit{dsn}^i(R^\circ_1)), z^i_{j-1}), \\
z^i_{k^i} \in \mathit{Follow}(0 \bullet \mathit{posex}(\mathit{dsn}^i(R^\circ_1)), z^i_{k^i-1}), & [\,[\Delta], (\text{case } \mathit{First} \text{ already treated in } m = 1)\,] \\
\exists\, ;_{t^1} \in \mathscr{S}, \quad \mathit{snd}(z^i_{k^i}) = ;_{t^1} \\
\mathit{snd}^*(z^i_0 \dots z^i_{k_i-1}) = a^i_0 \dots a^i_{k_i-1} \\
\qquad \vdots \\
\forall j \in \{1 \dots n_i - 1\}, \\
\exists x^i_0 \in \mathit{Follow}(0 \bullet \mathit{posex}(\mathit{dsn}^i(R^\circ_1)), q^i_0), \\
\exists x^i_j \in \mathit{Follow}(0 \bullet \mathit{posex}(\mathit{dsn}^i(R^\circ_1)), x^i_{j-1}), \\
x^i_{n^i} \in \mathit{Follow}(0 \bullet \mathit{posex}(\mathit{dsn}^i(R^\circ_1)), x^i_{n^i-1}), & [\,[\Delta]\,] \\
x^i_{n^i} \in \mathit{Last}(0 \bullet \mathit{posex}(\mathit{dsn}^i(R^\circ_1))), \\
\exists\, ;_{t^m} \in \mathscr{S}, \quad \mathit{snd}(x^i_{n^i}) = ;_{t^m} \\
\mathit{snd}^*(x^i_0 \dots x^i_{n_i-1}) = u^i_0 \dots u^i_{n_i-1}
\end{cases}
$$

$$
\iff \begin{cases}
\forall j \in \{1 \dots k_i - 1\}, \\
\exists z^i_0, \exists v^i_0, \quad z^i_0 v^i_0 \in L(0 \bullet \mathit{posex}(\mathit{dsn}^i(R^\circ_1))), & [\,\textit{prop. First}\,] \\
\exists z^i_j, \exists w^i_{j-1}, \exists v^i_j, \quad w^i_{j-1} z^i_{j-1} z^i_j v^i_j \in L(0 \bullet \mathit{posex}(\mathit{dsn}^i(R^\circ_1))), & [\,\textit{prop. Follow}\,] \\
\exists z^i_{k^i}, \exists w^i_{k^i-1}, \exists v^i_{k^i}, \quad w^i_{k^i-1} z^i_{k^i-1} z^i_{k^i} v^i_{k^i} \in L(0 \bullet \mathit{posex}(\mathit{dsn}^i(R^\circ_1))), & [\,\textit{prop. Follow}\,] \\
\exists\, ;_{t^1} \in \mathscr{S}, \ \mathit{snd}(z^i_{k^i}) = ;_{t^1} \\
\mathit{snd}^*(z^i_0 \dots z^i_{k-1}) = a^i_0 \dots a^i_{k_i-1} \\
\qquad \vdots \\
\forall j \in \{1 \dots n_i - 1\}, \\
\exists x^i_0, \exists w^i_{-1}, \exists v^i_0, \quad w^i_{-1} q^i_0 x^i_0 v^i_0 \in L(0 \bullet \mathit{posex}(\mathit{dsn}^i(R^\circ_1))), & [\,\textit{prop. Follow}\,] \\
\exists x^i_j, \exists w^i_{j-1}, \exists v^i_j, \quad w^i_{j-1} x^i_{j-1} x^i_j v^i_j \in L(0 \bullet \mathit{posex}(\mathit{dsn}^i(R^\circ_1))), & [\,\textit{prop. Follow}\,] \\
\exists x^i_{n^i}, \exists w^i_{n^i-1}, \exists v^i_{n^i}, \quad w^i_{n^i-1} x^i_{n^i-1} x^i_{n^i} v^i_{n^i} \in L(0 \bullet \mathit{posex}(\mathit{dsn}^i(R^\circ_1))), & [\,\textit{prop. Follow}\,] \\
\exists w^i_{n^i}, \quad w^i_{n^i} x^i_{n^i} \in L(0 \bullet \mathit{posex}(\mathit{dsn}^i(R^\circ_1))), & [\,\textit{prop. Last}\,] \\
\exists\, ;_{t^m} \in \mathscr{S}, \quad \mathit{snd}(x^i_{n^i}) = ;_{t^m} \\
\mathit{snd}^*(x^i_0 \dots x^i_{n-1}) = u^i_0 \dots u^i_{n_i-1}
\end{cases}
$$

Thus, let l_i the length of the whole desynchronized local word i

$$\forall j \in \{0 \dots k_i \dots l_i - n_i \dots l_i\}, \begin{bmatrix} w^i_{j\leq k_i} = z^i_0 \quad \dots z^i_{j-1} \\ v^i_{j\leq k_i} = z^i_{j+1} \dots z^i_{k_i} \dots x^i_{l_i-n_i} \dots x^i_{n_i} \\ \vdots \\ w^i_{j\geq l_i-n_i} = z^i_0 \quad \dots z^i_{k_i} \dots x^i_{l_i-n_i} \dots x^i_{j-1} \\ v^i_{j\geq l_i-n_i} = x^i_{j+1} \dots x^i_{n_i} \end{bmatrix}$$

Thus, $\forall j \in \{1 \dots m\}, \exists ;_{t^j} \in \mathscr{S}, \exists x^i_0 \dots x^i_{n_i}, \begin{bmatrix} z^i_0 \dots z^i_{k_i} \dots x^i_0 \dots x^i_{n_i} \ \in L(0 \bullet posex(dsn^i(R^\circ_1))) \\ snd(z^i_0 \dots z^i_{k_i} \dots x^i_0 \dots x^i_{n_i}) = a^i_0 \dots ;_{t^1} \dots u^i_0 \dots ;_{t^m} \end{bmatrix}$

▷ **if** $a^i_0 \in \left\{ snd(x^i) \mid x^i \in pos(1 \bullet posex(dsn^i(R^\circ_2))) \right\} \setminus \mathscr{S}$,
For the same symmetric reasons, we have
$\forall j \in \{1 \dots m\}, \exists ;_{t^j} \in \mathscr{S}, \exists z^i_0 \dots z^i_{k_i} \dots x^i_0 \dots x^i_{n_i},$
$\begin{bmatrix} z^i_0 \dots z^i_{k_i} \dots x^i_0 \dots x^i_{n_i} \ \in L(1 \bullet posex(dsn^i(R^\circ_2))) \\ snd(z^i_0 \dots z^i_{k_i} \dots x^i_0 \dots x^i_{n_i}) = a^i_0 \dots ;_{t^1} \dots u^i_0 \dots ;_{t^m} \end{bmatrix}$

▷ **Hence**, considering the global language of the BSPA, taking a Δ-transition entails that every desynchronized local transitions take the same label '$;_t$'. So again, it is not $\forall i \in [p], \exists ;_t \in \mathscr{S}, \dots$ but $\exists ;_t \in \mathscr{S}, \forall i \in [p], \dots$.

$$\iff \forall j \in \{1 \dots m\}, \exists ;_{t^j} \in \mathscr{S}, \forall i \in [p], \exists z^i_0 \dots z^i_{k_i} \dots x^i_0 \dots x^i_{n_i},$$
$$\begin{cases} z^i_0 \dots z^i_{k_i} \dots x^i_0 \dots x^i_{n_i} \in \left(L(0 \bullet posex(dsn^i(R^\circ_1))) \cup L(1 \bullet posex(dsn^i(R^\circ_2)))\right) \\ \wedge \quad snd^*(z^i_0 \dots z^i_{k_i} \dots x^i_0 \dots x^i_{n_i}) = a^i_0 \dots ;_{t^1} \dots u^i_0 \dots ;_{t^m} \end{cases}$$

$$\iff \forall j \in \{1 \dots m\}, \exists ;_{t^j} \in \mathscr{S}, \forall i \in [p], \exists z^i_0 \dots z^i_{k_i} \dots x^i_0 \dots x^i_{n_i},$$
$$\begin{cases} z^i_0 \dots z^i_{k_i} \dots x^i_0 \dots x^i_{n_i} \in L\big(0 \bullet posex(dsn^i(R^\circ_1)) + (1 \bullet posex(dsn^i(R^\circ_2)))\big) \quad [\ def.\ L_{RE}\ ] \\ \wedge \quad snd^*(z^i_0 \dots z^i_{k_i} \dots x^i_0 \dots x^i_{n_i}) = a^i_0 \dots ;_{t^1} \dots u^i_0 \dots ;_{t^m} \end{cases}$$

$$\iff \forall j \in \{1 \dots m\}, \exists ;_{t^j} \in \mathscr{S}, \forall i \in [p], \exists z^i_0 \dots z^i_{k_i} \dots x^i_0 \dots x^i_{n_i},$$
$$\begin{cases} z^i_0 \dots z^i_{k_i} \dots x^i_0 \dots x^i_{n_i} \in L\big(posex(dsn^i(R^\circ_1) + dsn^i(R^\circ_2))\big) \quad [\ def.\ posex\ ] \\ \wedge \quad snd^*(z^i_0 \dots z^i_{k_i} \dots x^i_0 \dots x^i_{n_i}) = a^i_0 \dots ;_{t^1} \dots u^i_0 \dots ;_{t^m} \end{cases}$$

$$\iff \forall j \in \{1 \dots m\}, \exists ;_{t^j} \in \mathscr{S}, \forall i \in [p], \exists z_0^i \dots z_{k_i}^i \dots x_0^i \dots x_{n_i}^i,$$
$$\begin{cases} z_0^i \dots z_{k_i}^i \dots x_0^i \dots x_{n_i}^i \in L\big(posex(dsn^i(R_1^\circ + R_2^\circ))\big) & [\ def.\ dsn\ ] \\ \wedge \quad snd^*(z_0^i \dots z_{k_i}^i \dots x_0^i \dots x_{n_i}^i) = a_0^i \dots ;_{t^1} \dots u_0^i \dots ;_{t^m} \end{cases}$$

$$\iff \forall j \in \{1 \dots m\}, \exists ;_{t^j} \in \mathscr{S}, \forall i \in [p], a_0^i \dots ;_{t^1} \dots u_0^i \dots ;_{t^m} \in L\big(dsn^i(R_1^\circ + R_2^\circ)\big) \quad [\ lem.\ posex^{-1}\ ]$$
$$\iff \forall j \in \{1 \dots m\}, \exists t^j, w_{t^1}^1 \dots w_{t^m}^m \in L\big(R_1^\circ + R_2^\circ\big) \quad [\ lem.\ dsn^{-1}\ ]$$
$$\iff w^1 \dots w^m \in L(R_1 + R_2) \quad [\ lem.\ annot^{-1}\ ]$$
$$\iff w^1 \dots w^m \in L(R) \quad [\ R = R_1 + R_2\ ]$$

□

Bibliography

[1] V. Allombert, F. Gava, and J. Tesson. Multi-ML: Programming multi-BSP algorithms in ML. *International Journal of Parallel Programming*, 45(2):340–361, Apr 2017. xvi, 87, 96

[2] Michela Becchi and Patrick Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proceedings of the 2007 ACM CoNEXT Conference*, CoNEXT '07, pages 1:1–1:12, New York, NY, USA, 2007. ACM. 20

[3] Justine Bonnot, Erwan Nogues, and Daniel Menard. New non-uniform segmentation technique for software function evaluation. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 131–138. IEEE, 2016. 73

[4] Mathias Bourgouin, Emmanuel Chailloux, and Anastasios Doumoulakis. Profiliing High Level Heterogeneous Programs Using the SPOC GPGPU framework for OCaml, 2017. LaMHA Presentation, 34 slides. xvi, 96

[5] Benjamin C. Brodie, David E. Taylor, and Ron K. Cytron. A scalable architecture for high-throughput regular-expression pattern matching. *SIGARCH Comput. Archit. News*, 34(2):191–202, May 2006. 26, 70

[6] Anne Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120(2):197–213, 1993. v, viii, 3, 32, 36, 37, 38, 130

[7] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, October 1964. v, 3, 51, 67

[8] Pascal Caron and Djelloul Ziadi. Characterization of Glushkov automata. *Theoretical Computer Science*, 233(1–2):75–90, 2000. vii, ix, 27, 32, 51, 67

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799*, pages 1–15, 2018. 72

[10] Yifeng Chen and J. W. Sanders. Top-down design of bulk-synchronous parallel programs. *Parallel Processing Letters*, 13(03):389–400, 2003. 18

[11] Renato J Cintra, Stefan Duffner, Christophe Garcia, and André Leite. Low-complexity approximate convolutional neural networks. *IEEE transactions on neural networks and learning systems*, pages 1–12, 2018. 73

[12] Murray I Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989. 73

[13] V. Elango, N. Rubin, M. Ravishankar, H. Sandanagobalane, and V. Grover. Diesel: DSL for linear algebra and neural net computations on GPUs. In *Proceedings of MAPL'18*. ACM, 2018. 72, 91

[14] Adin D. Falkoff and Kenneth E. Iverson. The design of APL. *IBM Journal of Research and Development*, 17(4):324–334, 1973. 73

[15] Jean Fortin and Frédéric Gava. BSP-Why: an intermediate language for deductive verification of BSP programs. In *4th workshop on High-Level Parallel Programming and applications (HLPP)*, pages 35–44. ACM, 2010. 19

[16] Z. Fu, Z. Liu, and J. Li. Efficient parallelization of regular expression matching for deep inspection. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9, July 2017. xi, 23, 24, 49, 50, 70

[17] Eric Goebelbecker. Using grep: Moving from dos? discover the power of this linux utility. *Linux J.*, 1995(18es), October 1995. xi, 49

[18] Yan Gu, Bu-Sung Lee, and Wentong Cai. JBSP: A BSP programming library in java. *Journal of Parallel and Distributed Computing*, 61(8):1126 – 1142, 2001. 18

[19] Gaétan Hains. Enumerated BSP automata. In Editor Andrew Adamatzky, editor, *Emergent Computation. A Festschrift for Selim G. Akl*, number 24 in Emergence, Complexity and Computation, pages 233–268. Springer Verlag, 2016. v, 3, 13, 50

[20] Gaétan Hains, Arvid Jakobsson, and Youry Khmelevsky. Towards formal methods and software engineering for deep learning: Security, safety and productivity for DL systems development. In *2018 Annual IEEE International Systems Conference (SysCon)*, Vancouver, Canada, 2018. IEEE. 72

[21] Gaetan Hains and Lenore M. R. Mullin. Parallel functional programming with arrays. *The Computer Journal*, 36(3):238–245, 1993. 73

[22] Marc E Herniter. *Programming in MATLAB*. Brooks/Cole-Thomson Learning, 2001. 73

[23] Jonathan M.D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947 – 1980, 1998. 18

[24] Jan Holub and Stanislav Štekr. On parallel implementations of deterministic finite automata. In *International Conference on Implementation and Application of Automata*, pages 54–64. Springer, 2009. xi, 23, 50, 66, 70

[25] Qiming Hou, Kun Zhou, and Baining Guo. Bsgp: bulk-synchronous gpu programming. In *ACM Transactions on Graphics (TOG)*, volume 27, page 19. ACM, 2008. xvi, 96

[26] International Collegiate Programming Contest. World finals challenge, problem B: Two-class binary neural network for handwritten digits. `myicpc.live/cdn/icpc-challenge-2019.pdf`, 2019. 85

[27] Arvid Jakobsson, Frédéric Dabrowski, Wadoud Bousdira, Frédéric Loulergue, and Gaetan Hains. Replicated synchronization for imperative BSP programs. *Procedia Computer Science*, 108:535–544, 2017. iv, 2

[28] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014. 72

[29] N. Jouppi, C. Young, N. Patil, and D. Patterson. Motivation for and evaluation of the first tensor processing unit. *IEEE Micro*, 38(3):10–19, May 2018. xvi, 96

[30] Shijin Kong, Randy Smith, and Cristian Estan. Efficient signature matching with multiple alphabet compression tables. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks*, SecureComm '08, pages 1:1–1:10, New York, NY, USA, 2008. ACM. 21

[31] A Krizhevsky and G Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 01 2009. http://www.cs.toronto.edu/ kriz/learning-features-2009-TR.pdf. 78

[32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012. 78

[33] S. Kumar, J. Turner, and J. Williams. Advanced algorithms for fast and scalable deep packet inspection. In *2006 Symposium on Architecture For Networking And Communications Systems*, pages 81–92, Dec 2006. 22

[34] Sailesh Kumar, Balakrishnan Chandrasekaran, Jonathan Turner, and George Varghese. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS '07, pages 155–164, New York, NY, USA, 2007. ACM. 20

[35] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '06, pages 339–350, New York, NY, USA, 2006. ACM. xi, 21, 22, 49

[36] Richard E Ladner and Michael J Fischer. Parallel prefix computation. *Journal of the ACM (JACM)*, 27(4):831–838, 1980. 22, 70

[37] Janghaeng Lee, Sung Ho Hwang, Neungsoo Park, Seong-Won Lee, Sunglk Jun, and Young Soo Kim. A high performance NIDS using FPGA-based regular expression matching. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 1187–1191. ACM, 2007. xi, 50

[38] Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. *SIGPLAN Not.*, 41(1):42–54, January 2006. 2

[39] Chong Li and Gaétan Hains. Sgl: towards a bridging model for heterogeneous hierarchical platforms. *International Journal of High Performance Computing and Networking*, 7(2):139–151, 2012. xvi, 87, 96

[40] C. Lin, C. Huang, C. Jiang, and S. Chang. Optimization of Pattern Matching Circuits for Regular Expression on FPGA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(12):1303–1310, Dec 2007. xi, 50

[41] Frédéric Loulergue. Implementation of a functional bulk synchronous parallel programming library. In *IASTED PDCS*, pages 447–452, 2002. 73

[42] Frédéric Loulergue, Wadoud Bousdira, and Julien Tesson. Calculating parallel programs in coq using list homomorphisms. *International Journal of Parallel Programming*, 45(2):300–319, Apr 2017. iv, 2, 19

[43] Frédéric Loulergue, Frédéric Gava, and David Billiet. Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction. In *International Conference on Computational Science (ICCS)*, volume 3515 of *LNCS*, pages 1046–1054. Springer, 2005. 18, 67

[44] Frédéric Loulergue, Gaétan Hains, and Christian Foisy. A Calculus of Functional BSP Programs. *Sci Comput Program*, 37(1-3):253–277, 2000. 19

[45] D. Luchaup, R. Smith, C. Estan, and S. Jha. Speculative parallel pattern matching. *IEEE Transactions on Information Forensics and Security*, 6(2):438–451, June 2011. 24, 70

[46] Daniel Luchaup, Randy Smith, Cristian Estan, and Somesh Jha. Multi-byte regular expression matching with speculation. In *International Workshop on Recent Advances in Intrusion Detection*, pages 284–303. Springer, 2009. 24

[47] Chad R. Meiners, Jignesh Patel, Eric Norige, Alex X. Liu, and Eric Torng. Fast regular expression matching using small TCAM. *IEEE/ACM Trans. Netw.*, 22(1):94–109, February 2014. xi, 26, 50, 70

[48] Suejb Memeti and Sabri Pllana. Parem: A novel approach for parallel regular expression matching. In *2014 IEEE 17th International Conference on Computational Science and Engineering*, pages 690–697. IEEE, 2014. xi, 23, 50, 70

[49] Armelle Merlin and Gaétan Hains. A bulk-synchronous parallel process algebra. *Comput. Lang. Syst. Struct.*, 33(3-4):111–133, October 2007. 19

[50] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos. Implementation of a content-scanning module for an internet firewall. In *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2003. FCCM 2003.*, pages 31–38, April 2003. xi, 25, 26, 49, 70

[51] Maleeha Najam, Usman Younis, and Raihan ur Rasool. Speculative parallel pattern matching using stride-k dfa for deep packet inspection. *Journal of Network and Computer Applications*, 54:78 – 87, 2015. 24, 26, 70

[52] Virginia Niculescu. Formal refinement of BSP programs with early cost evaluation. In *10th International Symposium on Parallel and Distributed Computing, (ISPDC)*, pages 49–56. IEEE Computer Society, 2011. xiv, 19, 94

[53] Christine Paulin-Mohring. Introduction to the coq proof-assistant for practical software verification. In *LASER Summer School on Software Engineering*, pages 45–95. Springer, 2011. iv, 2

[54] J. Roesch, S. Lyubomirsky, L. Weber, J. Pollock, M. Kirisame, T. Chen, and Z. Tatlock. Relay: A new IR for machine learning frameworks. In *Proceedings of MAPL'18*. ACM, 2018. vi, 4, 72, 91

[55] R. Sidhu and V. K. Prasanna. Fast regular expression matching using FPGAs. In *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*, pages 227–238, March 2001. 25, 70

[56] R. Sinya, K. Matsuzaki, and M. Sassa. Simultaneous finite automata: An efficient data-parallel model for regular expression matching. In *2013 42nd International Conference on Parallel Processing*, pages 220–229, Oct 2013. xi, 25, 50, 70

[57] R. Smith, C. Estan, and S. Jha. XFA: Faster signature matching with extended automata. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 187–201, May 2008. 21

[58] Thibaut Tachon, Chong Li, Gaétan Hains, and Frédéric Loulergue. Automated generation of bsp automata. *Parallel Processing Letters*, 27(01):1740002, 2017. 50, 59

[59] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968. v, vii, 3, 27, 51

[60] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8), 1990. iii, 1

[61] Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the 2006 ACM/IEEE Symposium on Architecture*

for Networking and Communications Systems, ANCS '06, pages 93–102, New York, NY, USA, 2006. ACM. xi, 22, 49

[62] Djelloul Ziadi and Jean-Marc Champarnaud. An optimal parallel algorithm to convert a regular expression into its Glushkov automaton. *Theoretical Computer Science*, 215(1-2):69–87, 1999. 32

Thibaut TACHON

Génération automatique de code parallèle isochrone

Résumé :

Depuis la stagnation de la fréquence d'horloge des processeurs, l'accroissement de la puissance de calcul a dépendu entièrement de l'accroissement du nombre d'unités de calcul. Plus que la difficulté algorithmique impliquée par l'écriture de tout programme séquentiel, la programmation parallèle demande au programmeur de gérer de nombreuses unités de calcul, incluant leurs tâches et leurs interactions. Pour alléger le fardeau du programmeur, cette thèse propose deux approches différentes de génération automatique de code parallèle. Le modèle parallèle isochrone BSP possède des propriétés intéressantes telles que son modèle de coût qui en font la cible de notre génération de code parallèle. Les automates et expressions régulières sont souvent choisis pour modéliser les calculs séquentiels et leurs parallélisation devrait, à long terme, aboutir à de solides fondations pour la génération de code parallèle. Pour notre approche principale, nous développons la théorie des automates BSP avec leur génération et déterminisation. Cette théorie est utilisée dans une nouvelle méthode pour la recherche de motif à l'aide d'expressions régulières. Notre autre approche propose un langage spécifique au domaine des réseaux de neurones où la composition fonctionnelle d'un petit nombre de primitives facilite le développement, la maintenance et la définition formelle du langage par rapport aux approches existantes.

Mots clés : Programmation parallèle, BSP, génération de code, automate, expression régulière, réseaux de neurones

Automatic Generation of Bulk-Synchronous Parallel code

Abstract :

Since we are in an era of processor clock stagnation, computing power growth has been relying on parallel computing. More than the algorithmic difficulty involved in any program writing, parallel computing additionally requires the programmer to manage numerous processing units including their tasks and interactions. In order to alleviate the parallel programmer's burden, this thesis proposes two different approaches for automatic parallel code generation. The bulk-synchronous parallel (BSP) model provides good properties such as its cost model and is therefore chosen as the target of our parallel code generation. Automata and regular expressions are often chosen to model sequential computation and their parallelization will lead to a strong foundation for general parallel code generation. For our main approach, we develop the theory of BSP automata with their generation and determinization. This theory is used in a novel method for parallel regular expression matching. As another approach, we propose a domain specific language for programming neural nets where the functional composition of only a few primitives eases development, maintenance and formal definition of the language compared to existing approaches.

Keywords : Parallel programming, BSP, code generation, automata, regular expression matching, neural nets

Centre Val de Loire UNIVERSITÉ Spirit of the Loire Valley

LABORATOIRE D'INFORMATIQUE
FONDAMENTALE D'ORLÉANS
6 rue Léonard de Vinci
45067 Orléans

UNIVERSITE D'ORLEANS