ORSAY
N^o d'ordre: 9749

UNIVERSITÉ DE PARIS-SUD 11
CENTRE D'ORSAY

THÈSE

présentée
pour obtenir

le grade de docteur en sciences DE L'UNIVERSITÉ PARIS XI

PAR

Florence PLATEAU

—×—

SUJET:

Modèle n-synchrone pour la programmation de réseaux de Kahn à mémoire bornée

soutenue le 06 janvier 2010 devant la commission d'examen composée de

Président :	Gérard Berry
Rapporteurs :	Nicolas Halbwachs
	Robert de Simone
Examinateurs :	Jean-Paul Allouche
	Marc Duranton
Directeur de thèse :	Marc Pouzet
Co-encadrant :	Louis Mandel

Remerciements

Le modèle n-synchrone est né quelques mois avant mon arrivée dans le monde de la recherche, de la collaboration entre Marc Duranton de chez Philips/NXP, Marc Pouzet et Albert Cohen. Je remercie Marc Pouzet de m'avoir offert ce sujet en stage de Master et d'avoir renouvelé sa confiance en me permettant de travailler dessus en thèse. Le modèle n-synchrone n'est qu'un exemple de la créativité scientifique pertinente de Marc. On pourrait en citer bien d'autres, à commencer par le langage Lucid Synchrone dont l'élégance m'a séduite lors de mon stage de Licence.

Je tiens à remercier Nicolas Halbwachs et Robert de Simone de m'avoir fait l'honneur d'être rapporteurs de ma thèse. Merci à Nicolas pour ses réactions encourageantes lors de mes exposés successifs au workshop Synchron. Merci à Robert de m'avoir accueillie à Sophia Antipolis pour que nous échangions sur les liens entre nos travaux respectifs.

Gérard Berry est un acteur clé pour la diffusion de la science informatique dans la société française, je le remercie chaleureusement d'avoir pris le temps de présider mon jury. Je tiens aussi à remercier les autres membres de mon jury de s'être intéressés à mon travail. Jean-Paul Allouche est un grand spécialiste des mots, je le remercie de l'œil bienveillant avec lequel il a considéré mes travaux sur le sujet. Marc Duranton a posé la question qui m'a occupée pendant mes années de thèse : comment mieux programmer les applications vidéo ? Je le remercie d'avoir étudié avec attention les éléments de réponse que j'ai essayé d'apporter dans ce mémoire.

Louis Mandel m'a accompagnée pendant les deux dernières années de ma thèse. Un grand merci à lui pour tout ce qu'il m'a généreusement appris, tant sur le plan scientifique, que méthodologique et relationnel. Merci Louis pour toutes nos séances de travail partagées, les meilleurs moments de ma thèse. Je pense en particulier à nos deux étés studieux "à quatre mains", le premier ayant donné naissance aux preuves Coq ✿, le second au typeur Lucy-n.

Merci à tous ceux qui m'ont fait le plaisir de venir assister à ma soutenance. Celle-ci n'aurait pu avoir lieu sans l'aide de mon médecin Jacques Pieri qui m'a reçue la veille pour traiter une laryngite "surprise". Je le remercie d'avoir saisi l'importance que ce problème avait pour moi. Je remercie aussi Simon Pouzet pour son dessin d'usine à sucettes qui était affiché au dessus de mon bureau, et m'a inspiré l'exemple utilisé au début de mon exposé.

Merci à Albert Cohen pour sa faculté d'écoute et de compréhension des problèmes scientifiques des autres. C'est ainsi qu'au détour d'une pause café, il m'a débloquée sur la dernière grosse difficulté rencontrée avant l'obtention des résultats que j'ai eu la chance de présenter à la conférence APLAS 2008 en Inde.

J'ai eu la chance de travailler ma thèse dans une équipe toujours très animée. Je voudrais remercier tous les membres de Proval pour leur contribution à cette ambiance agréable. En particulier : merci à Christine Paulin qui a su me conseiller quand j'ai traversé des périodes difficiles, merci à Yannick Moy pour sa bonne humeur inconditionnelle et son soutien quotidien, merci à Alexandre Bertails, Stéphane Lescuyer et Johannes Kanig d'avoir été mes camarades de bureau, et merci à Régine Bricquet d'avoir géré mes missions avec professionnalisme et gaieté. Merci à Sylvain Conchon, Aurélien Oudot, Andrei Paskevich et Philippe Dumont de m'avoir souvent donné envie de les rejoindre à la pause café, ainsi qu'à Jean-Christophe Filliâtre d'avoir toujours été prêt à me rendre service. Merci à Johannes, Stéphane et aux autres développeurs de Mlpost (l'outil que j'ai utilisé pour dessiner la quasi-intégrité de mes figures) d'avoir inventé cet outil ... et de l'avoir adapté à mes besoins. Merci à Stéphane pour le soutien en Coq et la relecture professionnelle et bienveillante qu'il a faite de ce mémoire. Merci à Gwenaël Delaval de m'avoir fourni l'exemple de l'encodeur/décodeur GSM. Enfin, merci à Gwenaël et Élodie d'avoir été mon grand frère et ma grande sœur de thèse.

J'ai eu le plaisir d'enseigner à l'IUT d'Orsay pendant ma thèse. Merci à mes étudiants et à mes collègues de l'IUT pour ce qu'ils m'ont appris sur l'enseignement. Tout particulièrement, merci à Cécile Balkanski de m'avoir accueillie dans son équipe pédagogique dont le fonctionnement est pour moi exemplaire.

Merci à mes enseignants de s'être impliqués dans leur métier, complexe, important, et pourtant trop peu valorisé. En particulier, merci à Mathieu Jaume pour la qualité de ses cours de Caml et d'interprétation abstraite que j'ai eu la chance de suivre.

Merci à mes amis "apprentis chercheurs" Benoît, Vincent et Zaynah pour ces années de peines et de réjouissances communes. Merci aussi à mes proches "non chercheurs" de m'avoir soutenue dans ce projet qui m'éloignait d'eux, et d'avoir été là quand je relevais la tête du guidon : Clémence, Morgane, Paul, Benoît et tous les compas, mon nid Morier-Bérelle, Aurélien, Maud, je suis de retour dans la vie normale !

Merci infiniment à mes parents adorés de croire en moi depuis toujours, et d'avoir tout fait pendant ma thèse pour me rendre la vie douce et agréable. Pour finir, mille mercis à mon bien-aimé de m'avoir épaulée tout au long de cette entreprise. Merci Léon d'avoir vécu cette thèse avec moi.

Table des matières

Table des figures

Chapitre 1

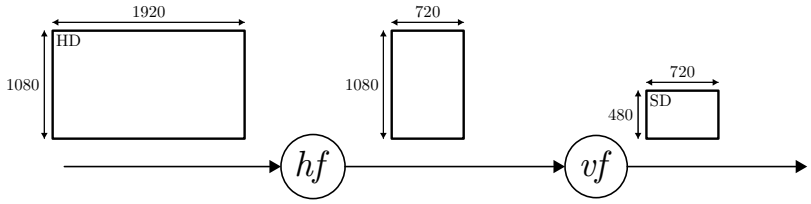
Introduction

Dans cette thèse, nous nous intéressons aux modèles et aux langages pour la programmation d'applications de traitement de flux ayant des contraintes de temps réel. Les applications multimédias sont de bons exemples de tels systèmes. Elles manipulent des flots de données infinis, auxquels sont appliquées des opérations, ou *filtres*, successifs. La figure 1.1 décrit un exemple d'application vidéo, appelée *Picture in Picture*, dont l'objet est d'incruster une vidéo dans une autre vidéo plus grande.

Notre objectif est de définir un langage adapté à la programmation de ce type d'applications. Par "langage adapté", on entend un langage qui permette de spécifier formellement les applications, et qui puisse être compilé vers du code impératif raisonnablement efficace. Par cette approche, on évite de concevoir l'application dans un langage sur lequel on sait faire des analyses de sûreté, puis de devoir la reprogrammer dans un langage de plus bas niveau pour l'exécuter. L'originalité du langage que nous proposons est d'autoriser la communication par *mémoires tampon* (ou *buffers*), tout en garantissant statiquement que les programmes peuvent être exécutés en mémoire bornée. Nous traitons aussi la question du calcul d'une taille suffisante pour chacun des *buffers*.

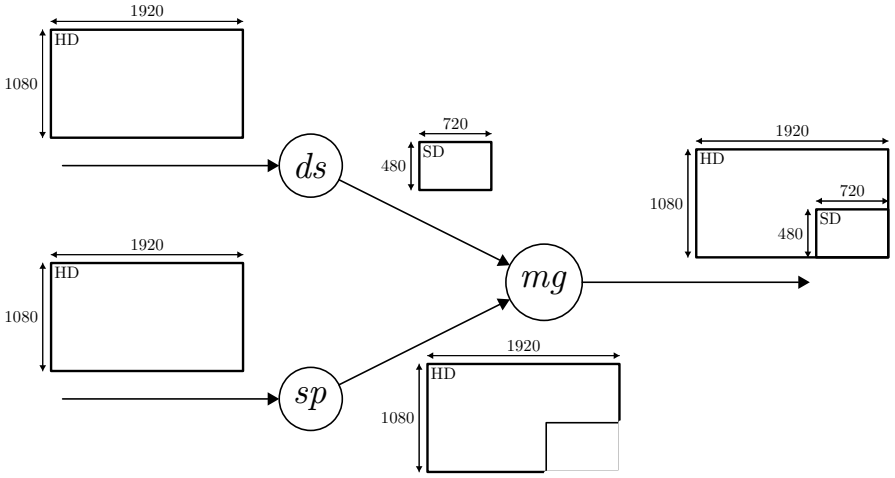
Les principales contraintes des applications considérées sont les suivantes :

- *Contrainte de temps-réel :* dans le pire cas, le rythme de production doit être maintenu. Par exemple, dans le cas d'une application vidéo, la fréquence d'affichage des pixels ne peut être diminuée, et on veut afficher tous les pixels (par opposition à une politique de type *best effort*).
- *Contrainte de performance :* des milliards d'opérations doivent être effectuées par seconde. Par exemple, pour une application vidéo hautes performances, si l'on veut effectuer en temps-réel 1000 opérations élémentaires par pixel d'une image en Haute Définition (HD), 150 milliards d'opérations par seconde doivent être effectuées.
  Pour respecter la contrainte de temps réel, l'exécution parallèle est indispensable.
- *Contrainte de sûreté :* on veut garantir que le comportement de l'application est déterministe, sans blocage à l'exécution et que la mémoire nécessaire est bornée.



Downscaler.

Le *Downscaler* reçoit un flot de pixels représentant des images en haute définition, et produit un flot contenant moins de pixels, représentant ces images en basse définition. Pour cela, le flot d'entrée est traité par un premier filtre (nommé *hf*) qui diminue le nombre de pixels par ligne, puis il traverse un second filtre (nommé *vf*) qui diminue le nombre de pixels par colonne. Le *Downscaler* peut lui même être considéré comme un filtre, qui peut être utilisé au sein d'une application plus complexe, comme l'application *Picture in Picture*.



Picture in Picture.

Le *Picture in Picture* reçoit les flots des pixels de deux images en haute définition. Il diminue la taille de la première en utilisant le *Downscaler*, crée un espace libre dans la seconde en supprimant des pixels et y intègre les pixels de la première image, pour produire une incrustation de la petite image dans la grande.

FIG. 1.1 – Exemples d'applications de traitement vidéo.

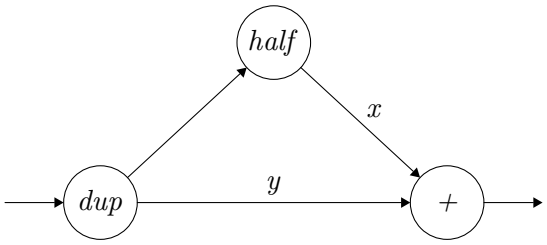


FIG. 1.2 – Réseau de Kahn nécessitant un buffer infini. Le nœud *dup* duplique son entrée, *half* recopie sur sa sortie une entrée sur deux, et + additionne ses deux entrées.

1.1 Réseaux de Kahn

Un modèle adapté au traitement de flux

Les ingénieurs du domaine sont habitués à modéliser leurs systèmes par des réseaux de processus de Kahn [Kah74]. La figure 1.1 en est un exemple. Dans ce modèle, des nœuds de calcul s'exécutent de manière concurrente et communiquent par des canaux munis de *mémoires tampon* infinies de type *FIFO (First In First Out)*. La lecture est bloquante si la file (le *buffer*) est vide, et comme les files sont de taille infinie, l'écriture est non bloquante. On ne peut pas tester l'absence de valeur dans une file.

Ce modèle de concurrence garantit que si les nœuds de calcul sont déterministes, alors leur composition l'est aussi. Cela signifie que la suite des valeurs en sortie du réseau ne dépend que de la suite des valeurs en entrée. En particulier, elle ne dépend pas de l'ordonnancement de l'exécution des nœuds, ainsi que du temps de calcul et de communication. Par conséquent, l'ordre d'exécution des nœuds n'est contraint que par les dépendances de données : un nœud peut être exécuté dès qu'il y a une valeur dans les buffers de chacun de ses canaux d'entrée.

On peut donc voir ce modèle comme un modèle mathématique où chaque nœud correspond à une fonction sur les suites et le réseau correspond à la composition de ces fonctions, qui est aussi une fonction de suites. Ainsi, les nœuds peuvent être eux-mêmes définis comme des réseaux de Kahn, comme le *Downscaler* de la figure 1.1 qui est à la fois un des nœuds composant le *Picture in Picture* et un réseau de Kahn composé des nœuds *hf* et *vf*.

En bref, le modèle des réseaux de Kahn présente l'avantage d'être un modèle hiérarchique, qui permet d'exprimer le parallélisme intrinsèque à l'application, et sur lequel on sait raisonner de manière formelle. L'inconvénient de ce modèle est qu'il ne donne pas de garanties sur l'exécution en temps-réel, sur l'absence de blocage et sur le caractère borné de la mémoire nécessaire à la communication. Par exemple, dans le réseau de la figure 1.2, comme le nœud + consomme une valeur sur chaque entrée à chaque activation, et comme deux fois plus de valeurs arrivent sur le canal y que sur le canal x, le nombre de valeurs dans le buffer du canal y croit indéfiniment.

On souhaite donc programmer les applications modélisées par ces réseaux de Kahn en ayant la garantie statique que l'implémentation a la même sémantique que le réseau de Kahn considéré, et que le réseau n'atteint pas d'état de blocage.

Programmer des réseaux de Kahn

En pratique, les buffers de taille infinie n'existent pas. Il est possible d'implémenter des réseaux de Kahn sans prendre le risque d'écritures dans des buffers pleins en mettant en place un mécanisme de rétroaction (ou *back pressure*). Ce mécanisme permet de rendre l'écriture bloquante lorsque le buffer est plein. Il peut être décrit dans le modèle de Kahn en ajoutant pour chaque buffer un canal allant du nœud qui lit dans le buffer vers le nœud qui écrit dedans, afin d'informer ce dernier de la présence de places libres [Par95]. Pour cela, après chaque lecture dans le buffer, le nœud consommateur doit produire une valeur sur le canal de rétroaction. De façon symétrique, avant chaque écriture dans le buffer, le nœud qui produit doit consommer une valeur sur le canal de rétroaction. Ainsi, une augmentation du nombre de valeurs dans le buffer se traduit par une diminution du nombre de valeurs sur le canal de rétroaction. La lecture dans un canal vide étant bloquante, les écritures dans le buffer sont contrôlées par le nombre de valeurs présentes sur le canal de rétroaction. Pour modéliser un buffer de taille n, il suffit de placer n valeurs initiales sur ce canal de rétroaction : quand celui-ci est vide, le buffer contenant les données est plein.

Ce mécanisme permet d'éviter les débordements de capacité des buffers, mais il crée des blocages dans le cas où les tailles de buffer sont insuffisantes. C'est le cas quand les tailles de buffers nécessaires sont infinies, et quand elle sont finies mais ont été sous-estimées. Ce problème peut-être traité en augmentant les tailles de buffers durant l'exécution en cas de blocage dû à un manque de places [Par95]. Cette technique garantit que s'il existe une exécution en mémoire bornée, et que la mémoire disponible est suffisante, alors l'exécution sera conforme au réseau de Kahn avec buffers infinis. Cependant, on aimerait avoir des garanties statiques sur l'exécution. Pour cela, il faut savoir calculer des tailles suffisantes pour les buffers si elles existent, et rejeter les réseaux qui nécessitent des buffers de taille infinie.

Le problème de savoir si un réseau de Kahn quelconque peut être exécuté en mémoire bornée est indécidable [Buc93]. Pour avoir des garanties sur le caractère borné de la mémoire nécessaire, il faut donc se placer dans un sous-ensemble des réseaux de Kahn sur lesquels des informations supplémentaires sont disponibles. C'est l'approche choisie par le modèle des *Synchronous Dataflow* [LM87b] qui se restreignent à des réseaux de Kahn dans lesquels le nombre de valeurs produites et consommées par chaque nœud à chaque activation est connu statiquement. Cela permet de calculer un ordonnancement statique périodique et une taille suffisante pour les buffers pour cet ordonnancement. De nombreuses variantes et améliorations de ce modèle ont été mises au point dans le projet Ptolemy [Pto], comme par exemple les *Cyclo-Static Dataflow* [PPL95].

L'approche consistant à se placer dans un sous-ensemble des réseaux de Kahn est aussi l'approche suivie par le modèle de programmation synchrone. Dans ce modèle, on ne considère que les réseaux qui peuvent être exécutés sans buffers. Pour cela, chaque canal est associé à une horloge qui définit les instants où une valeur est présente. Nous avons choisi de baser notre langage sur le modèle synchrone, et d'étendre ce modèle pour autoriser l'utilisation de buffers s'il existe pour ces derniers une taille finie ne provoquant pas de blocage à l'exécution.

1.2 Langages synchrones flot de données

Les langages synchrones flot de données Lustre [CPHP87] et Signal [BGJ91] ont été créés dans les années 1980 pour concevoir et implémenter des systèmes temps-réel critiques. Dans le modèle synchrone flot de données, on manipule des flots de valeurs, de manière concurrente et déterministe. Ces langages permettent de programmer des réseaux de Kahn synchrones, c'est-à-dire des réseaux de Kahn exécutables sans buffers sur les canaux de communication. Cette propriété est vérifiée à la compilation par une analyse appelée *calcul d'horloge*. Une autre analyse, *l'analyse de causalité* permet de vérifier statiquement que les programmes modélisent des réseaux de Kahn sans interblocage. Les programmes synchrones peuvent être compilés vers du matériel et du logiciel.

L'abstraction synchrone [BB91, BCE+03] est basée sur un temps logique discret qui est partagé par tous les nœuds. L'exécution d'un programme est une succession d'instants logiques, et l'hypothèse synchrone établit que les calculs et les communications se font de manière instantanée, c'est-à-dire au sein du même instant logique. Cela permet de raisonner facilement et formellement sur le programme. Cette hypothèse synchrone doit ensuite être validée par un calcul du pire temps de réaction (ou *Worst Case Reaction Time*), c'est-à-dire le plus long temps qui peut être nécessaire pour effectuer l'ensemble des calculs et les communications d'un même instant logique. Celui-ci doit être inférieur au temps physique associé à un instant logique. Enfin, la communication entre les nœuds de calcul doit être établie sans buffers, c'est-à-dire qu'une donnée produite doit être consommée au sein du même instant.

Le modèle synchrone flot de données [CPHP87, BGJ91, CP96] manipule des flots de valeurs, qui peuvent être combinés par l'application point-à-point d'opérateurs scalaires. Des opérateurs "temporels" permettent d'augmenter ou de diminuer le nombre de valeurs sur les flots. Par exemple un opérateur de sous-échantillonnage permet d'oublier des valeurs du flot, et un opérateur de sur-échantillonnage permet d'ajouter des valeurs sur un flot (en dupliquant les valeurs, ou en fusionnant deux flots en un seul). Chaque flot est associé à une horloge, c'est-à-dire un flot booléen indiquant les instants où une valeur est présente sur le flot. Pour que le réseau d'opérateurs puisse être exécuté sans buffers, il faut que les flots d'entrée d'un opérateur point-à-point soient présents aux mêmes instants, c'est-à-dire qu'ils aient la même horloge. Le *calcul d'horloge* est une analyse statique permettant de vérifier que les compositions vérifient cette règle.

Les langages synchrones flot de données fournissent un cadre simple et bien compris pour programmer les réseaux de Kahn sans buffers. La programmation de réseaux avec buffers n'est possible qu'en rajoutant un nœud ayant le comportement d'un buffer. Par exemple, ce nœud peut remplir un tableau avec ses entrées, et produire en sorties les valeurs du tableau, dans l'ordre où elles y ont été stockées et au rythme où elles sont consommées par le nœud suivant du réseau. Ce travail est cependant fastidieux et source d'erreurs. Le programmeur doit décider de la taille du tableau, passer en paramètre au buffer les horloges de l'entrée et de la sortie, et rien ne garantit que le nœud a effectivement le comportement d'un buffer de taille suffisante pour que la sémantique du programme coïncide avec la sémantique du réseau de Kahn associé. Cette solution n'est donc pas satisfaisante, et va même à l'encontre de la philosophie du modèle synchrone d'écrire des programmes proches de leur spécification.

1.3 Du synchrone au n-synchrone

En somme, l'intérêt des langages synchrones est de permettre de spécifier formellement une application temps-réel tout en étant compilables vers du code exécutable. Ils expriment la concurrence des différents traitements, et sont donc bien adaptés à une compilation permettant l'exécution parallèle des filtres. Ils offrent des garanties fortes, comme le déterminisme, l'absence de blocage et un besoin en mémoire borné. La contrepartie des garanties fournies malgré une grande expressivité des rythmes est un manque de souplesse dans la composition des flots. La composition sans buffer n'est acceptée que si le calcul d'horloge sait vérifier l'égalité des horloges des flots que l'on compose.

La vérification des horloges en Lustre et Lucid Synchrone est un problème de typage et peut être décrit par un système de types [CP96, CP03]. Chaque expression est associée à un type d'horloges et le compilateur vérifie que le programme respecte des contraintes sur ces types. Considérons par exemple la règle suivante :

$$\frac{H \vdash e_1 : ck \quad H \vdash e_2 : ck}{H \vdash e_1 + e_2 : ck}$$

Elle indique qu'une addition de deux flots e_1 et e_2 est correcte à condition que e_1 et e_2 aient le même type d'horloges ck. Dans ce cas, le type d'horloges du résultat $e_1 + e_2$ est aussi ck. Ainsi, pour typer un programme, les questions qui se posent concernent toutes l'égalité de types. L'idée du modèle n-synchrone est d'assouplir cette contrainte d'égalité afin de permettre la composition de flots non nécessairement synchrones dès lors que leurs horloges sont proches l'une de l'autre. Si un flot x dont l'horloge est de type ck peut être consommé un peu plus tard sur une horloge du type ck' à condition d'utiliser un buffer de taille bornée, on dira que ck est un sous-type de ck' et on écrira $ck <: ck'$. Le typage est enrichi avec une règle de la forme :

$$\frac{H \vdash e : ck \quad ck <: ck'}{H \vdash e : ck'}$$

À chaque utilisation de cette règle, un buffer doit être inséré. Une fois le typage effectué, le programme peut être traduit automatiquement vers un nouveau programme, synchrone au sens classique, où les buffers sont placés explicitement et de tailles connues. On peut décider d'appliquer cette règle de typage à chaque application de fonction, de sorte que des buffers soient potentiellement placés sur chacune des entrées de tous les nœuds. Une version plus simple de cette approche consiste à marquer explicitement dans le programme les points où des buffers doivent être utilisés, leur taille restant calculée automatiquement :

$$\frac{H \vdash e : ck \quad ck <: ck'}{H \vdash \texttt{buffer}(e) : ck'}$$

C'est l'approche que nous avons suivie, afin de limiter le nombre de contraintes de sous-typage à résoudre durant l'inférence des types, et de permettre au programmeur de décider du mode de communication (avec ou sans buffer). Ce modèle de communication au synchronisme relâché est qualifié de n-synchrone, n représentant la taille des buffers à mettre en place.

1.4 Contribution et plan

Dans la partie I, on présente le langage proposé et les bases algébriques permettant de construire des langages d'horloges pour ce langage de programmation.

Le **chapitre 2** présente tout d'abord les constructions du langage, qui est basé sur un noyau à la Lustre enrichi d'un opérateur de bufferisation. Après avoir établi la sémantique de ce langage, on décrit son calcul d'horloge. La nouveauté de ce calcul d'horloge est l'introduction d'une règle de sous-typage, permettant d'accepter des compositions qui ne sont pas strictement synchrones. C'est cette idée publiée dans [CDE$^+$05, CDE$^+$06] qui est à la base de mon travail de thèse. La contrainte de sous-typage se traduit en une contrainte sur les horloges impliquées dans les types.

Pour savoir comparer les horloges, on met en place dans le **chapitre 3** une algèbre manipulant les horloges comme des mots binaires infinis, et définissant les opérations et relations qui permettent de vérifier que la contrainte de composition à buffers bornés est respectée. Les jalons principaux de cette algèbre étaient déjà posés à mon arrivée, ma contribution relève d'une étude plus poussée des propriétés des opérateurs et relations. En particulier, j'ai formulé ces propriétés sous l'angle de la fonction de cumul des mots binaires, qui était absente dans les travaux précédents. Cette formulation s'est imposée lors du travail de preuve en Coq. Elle permet une meilleure compréhension des horloges et facilite souvent les preuves. Pour cette raison, elle a permis des évolutions dans les travaux présentés dans la partie III.

La règle de sous-typage introduit une contrainte entre des types. L'ensemble de contraintes collectées lors du typage d'un programme doit être satisfait, afin de pouvoir assurer que les communications par buffers sont sûres. Pour cela, il faut être capable de vérifier en un temps fini qu'une contrainte entre deux types et donc entre deux horloges est satisfaite. L'algèbre présentée dans le chapitre 3 ne fournit que des opérations et des tests qui nécessitent un parcours des mots infinis représentant les horloges, donc qui ne sont pas calculables. Pour mettre en œuvre cette algèbre sur les horloges d'un langage de programmation, il faut donc définir un langage d'horloges sur lequel on sait effectuer ces calculs en un temps fini. D'autre part, comme nous inférons les types, il faut être capable de résoudre des contraintes sur les horloges, c'est-à-dire de trouver des valeurs pour les horloges inconnues, telles que les contraintes soient satisfaites. L'idée première pour atteindre cet objectif de vérifier et de résoudre les contraintes en un temps fini est de se restreindre au cas des horloges périodiques. C'est ce qui est présenté dans la partie II. Une deuxième idée est de se restreindre au cas d'horloges qu'on est capable d'abstraire en les encadrant par des rythmes réguliers sur lesquels on sait raisonner. C'est ce qui est présenté dans la partie III. Ces deux parties sont indépendantes et peuvent être lues séparément. Elles suivent le même plan : un premier chapitre décrit l'algèbre des horloges considérées, un deuxième chapitre la mise en œuvre de ces horloges dans l'algorithme d'inférence d'horloges du langage de programmation, et un troisième chapitre montre des applications et discute des résultats obtenus. Les applications choisies sont les mêmes pour les deux parties, ce qui permet de les utiliser comme point de comparaison des deux méthodes.

Cette thèse est associée à des implémentations réalisées avec Louis Mandel :

- La majorité des travaux présentés a été programmée en OCaml. Le langage et le calcul d'horloge ont été codés de manière générique, afin de pouvoir utiliser au choix le langage d'horloges de la partie II ou le langage d'horloges de la partie III. Tous les types des exemples donnés dans ce document correspondent à la sortie fournie par notre typeur.
- Une partie importante des propriétés énoncées dans les chapitres 3 et 7 a été prouvée à l'aide de l'assistant de preuves Coq [BC04]. Chaque énoncé défini ou prouvé en Coq est suivi d'un lien ✿ pointant vers le code correspondant.

Tous les développements sont disponibles à l'adresse http://www.lri.fr/~plateau/these/.

Dans la partie II, nous mettons en place un langage d'horloges périodiques pour notre langage de programmation. On entend par horloges périodiques des mots binaires infinis composés d'un préfixe quelconque, suivi de la répétition infinie d'un même motif.

Le **chapitre 4** définit de manière algorithmique l'algèbre des mots binaires présentée dans le chapitre précédent, spécialisée au cas des mots ultimement périodiques. Ce travail consiste principalement à trouver pour chaque opérateur le nombre d'étapes de calcul à effectuer avant d'atteindre le comportement périodique du résultat. L'existence de telles bornes est intuitive et avait été annoncée dans [CDE+05, CDE+06]. Ma contribution porte sur la définition précise de ces bornes, et sur les preuves de leur correction et leur minimalité.

Le **chapitre 5** présente plusieurs algorithmes de résolution des contraintes d'égalité sur les types et un algorithme de résolution des contraintes de sous-typage. Dans le cadre des horloges périodiques, les contraintes d'égalité sur les types peuvent être résolues de manière plus fine que dans le cas général des horloges de Lustre, en interprétant la valeur des horloges plutôt qu'en comparant leur syntaxe. Une méthode de résolution des contraintes une à une, au fur et à mesure qu'elles sont collectées a été présentée dans [CDE+06]. J'ai montré que celle-ci n'est pas complète, et pas toujours plus précise que la résolution basée sur la syntaxe. Il n'est en fait pas possible de trouver une solution aux contraintes d'égalité sans les considérer dans leur globalité, car il existe différentes valeurs non comparables pour les horloges inconnues permettant de satisfaire une contrainte d'égalité. Ce résultat est une de mes contributions. Pour les mêmes raisons, la résolution des contraintes de sous-typage doit elle aussi être réalisée de manière globale. L'article [CDE+06] présentait un algorithme de résolution qui transforme le système de contraintes avec des règles le simplifiant par des traitements sur une contrainte à la fois, plus une règle nécessitant d'observer l'ensemble des contraintes avant d'être appliquée (ce qui donne un caractère global à l'algorithme), ceci jusqu'à l'obtention d'un système vide. J'ai découvert plus tard que cet algorithme n'aboutit pas toujours à un système vide lorsqu'il existe une solution. Le chapitre présente donc un nouvel algorithme de résolution globale des contraintes. Cet algorithme s'applique à tous les systèmes ne contenant que des horloges strictement périodiques (sans phase d'initialisation). Par contre il n'est pas défini sur un certain nombre de systèmes contenant des horloges qui ne sont périodiques qu'à partir d'un certain rang. Il est prouvé correct, et nous pensons qu'il est complet sur les systèmes qu'il sait traiter, mais la preuve reste à faire. En plus de ne pas être applicable à toutes les horloges ultimement périodiques, cet algorithme est de complexité coûteuse et calcule des types corrects mais pas toujours au plus près de nos attentes. Nous considérons donc ce travail plus comme un travail en cours que comme un travail abouti. Nous pensons néanmoins que la partie II donne un certain nombre de clés permettant de comprendre la complexité du problème de la résolution correcte et complète des contraintes d'égalité et de sous-typage sur des types d'horloges ultimement périodiques.

Des résultats encourageants sont décris dans le **chapitre 6**, qui applique le calcul d'horloges périodiques à deux cas d'étude issus du domaine du traitement multimédia. Le premier est tiré du protocole de communication GSM pour la téléphonie mobile, le second est l'exemple de l'incrustation d'images que nous avons décrit page 12. Ces exemples illustrent tous les points abordés dans le chapitre précédent, et sont suivis d'une discussion sur les forces et les faiblesses des algorithmes présentés.

Dans la partie III, nous mettons en place un langage d'horloges abstraites pour notre langage de programmation. Par horloges abstraites, on entend des mots binaires infinis dont on connaît le rythme moyen et l'avance ou le retard extrêmes qu'ils peuvent prendre par rapport à ce rythme.

Le **chapitre 7** est à mon sens la contribution principale de cette thèse. Il présente tout d'abord l'intuition de l'abstraction des horloges. Pour cela, la méthode d'abstraction que nous avons présentée dans [CMPP08] est utilisée comme support. Nous avons fait évoluer la technique d'abstraction en reconsidérant notre théorie sous l'angle des fonctions de cumul des mots. Une version intermédiaire et les preuves Coq associées ont été publiées dans [MP09] et la version finalement retenue a été présentée dans [CMPP09]. Le chapitre détaille très précisément cette dernière approche. Elle consiste à encadrer la fonction de cumul d'un mot binaire par deux droites. Ces droites définissent un ensemble d'horloges, qu'on appelle *enveloppe*. Les enveloppes donnent donc des bornes sur le nombre de valeurs prises par un flot depuis le début de l'exécution. Il s'agit de savoir vérifier à partir de ces bornes les contraintes assurant que la communication peut-être établie de manière sûre à travers un buffer. Notons que dans ce cadre, l'égalité de deux horloges ne peut être assurée que par l'égalité syntaxique de leur noms. La communication sans buffer ne peut donc être établie que lorsque les horloges sont syntaxiquement égales, comme dans Lustre. Pour être capables de vérifier les contraintes de sous-typage sur des types d'horloges abstraites, les opérations et relations décrites dans l'algèbre du chapitre 3 doivent être définies sur les enveloppes. Le chapitre reprend donc ces opérations et relations, en fournissant des formules permettant de les calculer. La force de ces formules est leur simplicité et leur faible coût algorithmique : ce sont de simples opérations arithmétiques sur les nombres rationnels. Dans ce travail, nous avons été principalement confrontés à deux difficultés techniques. La première a été de trouver la formule calculant les enveloppes des horloges de flots obtenus par échantillonnages successifs. La seconde difficulté a été de trouver des formules produisant des résultats les plus précis possibles. Nous avons réussi à trouver de telles formules pour tous les tests et pour la plupart des opérations. Pour les autres opérations, nous avons cependant borné formellement l'imprécision de la formule.

Le **chapitre 8** présente un algorithme de résolution des contraintes de sous-typage basé sur la méthode d'abstraction des horloges présentée dans le chapitre précédent. Nous montrons que la résolution du système de contraintes de sous-typage se ramène à la résolution d'un système d'inéquations linéaires. La taille de ce système étant proportionnelle au nombre de contraintes de sous-typage, les performances de l'algorithme sont satisfaisantes. Le principe d'abstraction implique forcément une incomplétude de l'algorithme. Il n'est de plus pas complet dans le domaine abstrait car toutes les opérations ne sont pas les plus précises, mais il présente des résultats pratiques concluants.

Ces résultats sont décrits dans le **chapitre 9** qui applique l'algorithme de résolution avec abstraction à nos cas d'étude. Le code de l'application *Picture in Picture*, ainsi que les types d'horloges obtenus sont expliqués en détail. L'exemple extrait du protocole GSM (dont le code est décrit dans le chapitre 6) est utilisé pour montrer le déroulement pas à pas de l'algorithme de résolution des contraintes. Nous terminons le chapitre par une discussion sur la précision et l'efficacité de la méthode présentée, et les utilisations possibles des horloges abstraites pour modéliser des aspects qui ne sont habituellement pas représentés dans le modèle synchrone, comme la gigue ou le temps de calcul.

Dans la partie IV, nous concluons cette thèse, en positionnant nos travaux par rapport aux travaux reliés et en présentant nos perspectives de recherche.

Première partie

Le modèle n-synchrone

Chapitre 2

Présentation du langage

Ce chapitre présente un langage flot de données synchrone disposant d'un opérateur de bufferisation. Cet opérateur permet de relâcher la condition de synchronisme, c'est-à-dire de consommer certains flots à un instant ultérieur à celui où ils ont été produits. Tout comme les langages flot de données synchrones traditionnels, ce langage est doté d'un *calcul d'horloge* assurant que les programmes peuvent être exécutés en mémoire bornée. Le calcul d'horloge des langages synchrones comme Lustre garantit que la communication ne nécessite pas de mémoire, car chaque nœud du réseau consomme ses flots d'entrée à l'instant même où ceux-ci sont produits par le nœud qui les calcule. Le calcul d'horloge du langage que nous présentons garantit que la mémoire nécessaire à la communication est de taille bornée mais pas nécessairement nulle, et calcule automatiquement cette taille. Pour cette raison, le langage est qualifié de n-synchrone, n représentant la taille de la mémoire nécessaire. Nous l'appellerons Lucy-n.

La section 2.1 présente le noyau de Lucy-n, ainsi que des exemples de programmes avec buffers. Nous donnons ensuite la sémantique du langage dans la section 2.2 et son calcul d'horloge dans la section 2.3.

2.1 Un noyau synchrone avec opérateur de bufferisation

Dans cette section, nous présentons le langage au travers de la définition de son noyau et de la programmation d'exemples synchrones classiques puis d'exemples montrant l'utilisation de l'opérateur de bufferisation.

2.1.1 Noyau du langage

Le noyau de Lucy-n est défini dans la figure 2.1. Les expressions (e) s'évaluent vers des flots de valeurs. Les opérateurs binaires importés ($op(e, e)$) sont appliqués point-à-point aux valeurs des flots passés en argument. Il est possible d'importer de la même manière des opérateurs d'autres arités, en particulier un opérateur `if/then/else` d'arité trois, implémentant une conditionnelle. La construction `where rec` permet de définir localement un ensemble de variables à l'aide d'équations mutuellement récursives. L'opérateur `fby` retourne la valeur courante de son premier argument au premier instant, puis la valeur précédente de son second argument pour toujours. Les opérateurs `when`, `whenot` et `merge` agissent sur la longueur des flots. Il prennent en argument une expression d'horloges ce. Cette expression d'horloges doit

e	::=	i	flot constant
		$\mid x$	variable de flot
		$\mid$ `(`e`,`e`)`	paire
		$\mid$ `fst` e	premier élément d'une paire
		$\mid$ `snd` e	second élément d'une paire
		$\mid op(e, e)$	opérateur binaire importé
		$\mid f\ e$	application
		$\mid e$ `where rec` eqs	définitions locales
		$\mid e$ `fby` e	délai initialisé
		$\mid e$ `when` ce	échantillonnage
		$\mid e$ `whenot` ce	échantillonnage
		$\mid$ `merge` $ce\ e\ e$	fusion
		$\mid$ `buffer`(e)	bufferisation
eqs	::=	x `=` e	équation
		$\mid eqs$ `and` eqs	ensemble d'équations
d	::=	`let node` $f\ x$ `=` e	définition d'un nœud
		$\mid$ `let clock` c `=` ce	définition d'une horloge
		$\mid d\ d$	séquence de définitions

FIG. 2.1 – Noyau du langage.

être écrite dans un langage s'évaluant en une suite infinie de valeurs booléennes. e `when` ce échantillonne le flot e sur la condition exprimée par l'expression d'horloges ce, pour produire un flot plus court : les valeurs prises par e ne sont conservées que quand ce prend la valeur *vrai*. e `whenot` ce fait de même avec la négation de ce. `merge` combine des flots complémentaires pour construire un flot plus long. Enfin, l'opérateur `buffer` permet de consommer les valeurs prises par une expression après l'instant où elles sont disponibles en les conservant dans un buffer. Un programme (d) est une séquence de définitions de fonctions sur les flots, appelées nœuds, et d'horloges.

Dans le langage Lustre, les horloges peuvent être définies par n'importe quelle expression booléenne du langage. Pour simplifier la définition du calcul d'horloge, nous nous limitons à des langages d'horloges ne contenant pas de variables de flots du programme. Le cas général n'a pas été considéré dans cette thèse. Le noyau synchrone avec opérateur de bufferisation peut être utilisé avec tout langage d'horloges s'évaluant vers des séquences booléennes infinies et muni des opérations suivantes :

- un test d'égalité des horloges, pour vérifier que les communications réalisées sans buffer sont synchrones ;
- un test d'*adaptabilité* des horloges, pour vérifier que les communications réalisées à travers un buffer sont n-synchrones ;
- une opération permettant de calculer une taille suffisante pour chaque buffer.

Dans Lustre, le test d'égalité est l'égalité syntaxique. Comme les expressions d'horloges sont des expressions arbitraires du langage, il serait complexe de mettre en place un test plus puissant. Dans ce cadre, on peut difficilement imaginer un test d'adaptabilité et un calcul des tailles de buffers. On doit donc utiliser un langage d'horloges différent.

Le langage d'horloges considéré dans ce chapitre

Les exemples de ce chapitre sont écrits à l'aide d'expressions d'horloges définies par des mots binaires ultimement périodiques, décrits par la grammaire suivante :

$$\begin{array}{lcl} ce & ::= & c \mid p \\ p & ::= & u(v) \\ u & ::= & \varepsilon \mid b.u \\ v & ::= & b \mid b.v \\ b & ::= & \texttt{0} \mid \texttt{1} \end{array}$$

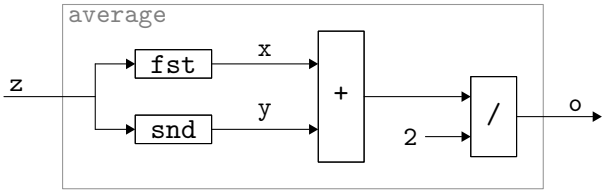
Une expression d'horloges est ici soit un identificateur c, soit un mot binaire composé d'un préfixe u (éventuellement vide ε) suivi de la répétition infinie d'un motif v. Par exemple, l'expression `(10)` désigne le flot booléen alternant `101010`..., où `1` désigne la valeur *vrai* et `0` désigne la valeur *faux*. Le test d'égalité est l'égalité de deux mots ultimement périodiques qui est décidable. Le test d'adaptabilité est défini dans le chapitre 3 et est lui aussi décidable sur les mots ultimement périodiques. Le choix d'un langage d'horloges minimal vise à alléger ce chapitre, afin de se focaliser sur la compréhension du style de programmation n-synchrone (sections 2.1.2 et 2.1.3), de la sémantique (section 2.2) et du calcul d'horloge (sections 2.3). Ceux-ci sont indépendants du langage d'horloges utilisé. Les parties II et III détaillent deux langages d'horloges plus expressifs.

2.1.2 Premiers programmes

Le langage considéré est un langage premier ordre, semblable à Lustre, dans lequel on a remplacé l'opérateur de sur-échantillonnage `current` par l'opérateur de fusion de flots `merge` [Pou06].

Un nœud qui calcule la moyenne de deux flots réunis dans une paire s'écrit :

```
let node average z = o where
  rec x = fst z
  and y = snd z
  and o = (x + y) / 2
```



La sortie de ce nœud est la valeur définie par `o`. Le mot clé `where` introduit un ensemble d'équations définissant des variables locales `x` et `y` pour destructurer la paire reçue en argument, et la variable `o` calculée par l'application point à point des opérateurs `+` et `/`.
Si l'on applique le nœud `average` sur l'entrée `(2,6) (8,4) (4,2) (3,3)` ..., on obtient les suites de valeurs suivantes pour les variables du nœud :

z	(2,6)	(8,4)	(4,2)	(3,3)	...
x	2	8	4	3	...
y	6	4	2	3	...
o	4	6	3	3	...

Extension du noyau

Le noyau du langage ne contient que des fonctions unaires, mais on peut encoder facilement les fonctions n-aires par des fonctions prenant des n-uplets en paramètre. Par exemple, le nœud `average` constitue un encodage d'une fonction binaire effectuant la moyenne de ses

deux arguments. Afin de ne pas avoir à déconstruire l'entrée d'un nœud avec les opérateurs `fst` et `snd`, on autorise la notation utilisée dans le nœud `average2` suivant :

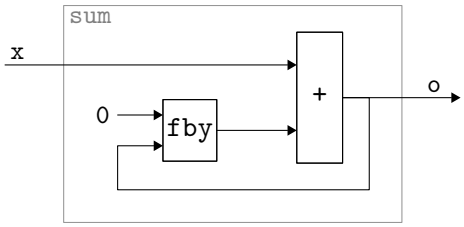
```
let node average2 (x, y) = (x + y) / 2
```

Les n-uplets autres que les paires ne sont pas disponibles dans le noyau mais peuvent être facilement encodés à l'aide des paires. Pour cette raison, nous nous permettrons d'utiliser cette construction dans la suite.

Les équations x `=` e ne permettent de définir qu'une variable par équation, mais on peut encoder la définition des n-uplets. Par exemple, l'équation (x_1, x_2) `=` e s'encode par les trois équations z `=` e `and` x_1 `=` `fst` z `and` x_2 `=` `snd` z.

Délai initialisé

L'opérateur `fby` permet d'utiliser des valeurs précédentes d'un flot, et en particulier de définir un flot en fonction de ses valeurs précédentes. Illustrons cela au travers de l'exemple d'un intégrateur, qui calcule la somme des valeurs prises par son flot d'entrée :




```
let node sum x = o where
  rec o = x + (0 fby o)
```

L'expression `0 fby o` vaut 0 au premier instant, puis la valeur précédente de `o` aux instants suivants. La sortie `o` est donc égale à la somme de sa valeur précédente et de l'entrée courante. Si l'on applique le nœud `sum` sur l'entrée 5 7 3 6 2 8 . . ., on obtient les suites de valeurs suivantes pour les expressions du nœud :

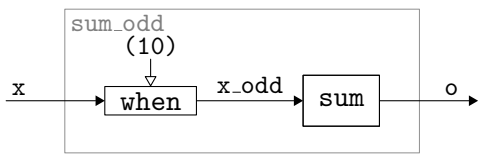
`x`	5	7	3	6	2	8	...
`0 fby o`	0	5	12	15	21	23	...
`o`	5	12	15	21	23	31	...

L'opérateur `fby` permet donc d'écrire des flots calculables séquentiellement : la valeur de `o` à l'instant n ne dépend que de la valeur de l'entrée `x` à l'instant n et de la valeur de `o` à l'instant $n-1$. On dit qu'un système d'équations est *causal* lorsqu'il est calculable de cette manière. L'expression `o = x + o` n'est donc pas causale, car elle définit la valeur courante de `o` en fonction d'elle-même. Nous ne traiterons pas du sujet de la causalité, qui a été étudié dans [CP01, TN98, SBT96].

Échantillonnage et fusion de flots

L'opérateur `when` permet de supprimer certaines valeurs d'un flot. Dans l'expression `x when c`, `x` est le flot qu'on échantillonne et `c` est la condition d'échantillonnage : les valeurs de `x` sont conservées lorsque `c` vaut `1`, et supprimées sinon. Par exemple, si l'on ne veut conserver que les valeurs de rang impair d'un flot `x`, on peut l'échantillonner avec l'horloge `(10)` = `101010`. . .. Ainsi, on peut réutiliser le nœud `sum` précédent pour écrire un nœud qui somme les éléments de rang impair d'un flot :

```
let node sum_odd x = o where
  rec x_odd = x when (10)
  and o = sum x_odd
```



Si l'on applique ce nœud au flot d'entrée de l'exemple précédent, on obtient les suites de valeurs suivantes :

x	5	7	3	6	2	8	...
(10)	1	0	1	0	1	0	...
x_odd	5		3		2		...
o	5		8		10		...

On observe que les flots `x_odd` et `o` ne sont définis qu'un instant sur deux. Pour distinguer les instants où un flot est défini des instants où il n'est pas défini, on lui associe une horloge. Celle-ci prend la valeur `1` aux instants où le flot est présent, et la valeur `0` aux instants où le flot est absent. Ainsi, le flot `x` et la condition d'échantillonnage qui sont définis à chaque instant sont associés à l'horloge `(1)` = `1111`.... Le flot `x_odd` n'est quant à lui défini qu'un instant sur deux à partir du premier instant et son horloge est donc `(10)`. On verra dans le chapitre 3 que cette horloge est le résultat de l'opération `(1)` *on* `(10)`. En effet, l'opérateur *on* calcule l'horloge de `x when (10)` en fonction de l'horloge de `x` et de l'horloge `(10)`.

Remarque 2.1. L'appellation *horloge* est utilisée pour deux notions différentes : les conditions d'échantillonnage et les suites représentant les instants de présence des flots. Une condition d'échantillonnage n'est présente qu'aux instants où le flot à échantillonner est présent. Il s'agit donc d'un flot booléen qui peut être absent à certains instants. Les suites représentant les instants de présence des flots sont quant à elles des suites booléennes toujours définies. L'utilisation d'une même terminologie pour ces deux notions est donc un abus de langage. Nous nous permettons cet abus car ces deux types de suites sont définies à partir du même langage, qu'on appelle langage d'horloges. ◇

Il est possible de déclarer une horloge de la manière suivante : `let clock half = (10)`. Cela permet d'utiliser dans la suite du programme le nom `half` pour désigner l'horloge périodique `(10)`. Le nœud `sum_odd` peut alors être réécrit ainsi :

```
let node sum_odd_bis x = o where
  rec x_odd = x when half
  and o = sum x_odd
```

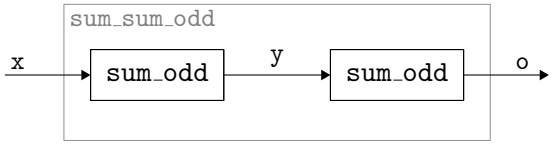
On peut échantillonner à nouveau un flot qui a déjà été échantillonné. Par exemple :

flot	valeurs							horloge
x	5	7	3	6	2	8	...	(1)
(10)	1	0	1	0	1	0	...	(1)
x when (10)	5		3		2		...	(1) *on* (10) = (10)
(110)	1		1		0		...	(10)
(x when (10)) when (110)	5		3				...	((1) *on* (10)) *on* (110) = (101000)

L'horloge du flot échantillonné par `(10)` puis par `(110)` est ((`1`) *on* (`10`)) *on* (`110`). On peut observer dans le tableau ci-dessus que cette horloge vaut `(101000)` : le flot est présent au premier instant, puis absent, puis présent, puis absent durant trois instants. Les conditions d'échantillonnage étant périodiques, ce comportement se reproduit infiniment.

On obtient de telles horloges lors de la composition d'opérateurs qui échantillonnent. Par exemple, si l'on applique deux fois de suite le nœud `sum_odd` précédemment défini, qui rend deux fois moins de valeurs en sortie qu'il n'en consomme en entrée, on obtient un flot quatre fois plus lent.

```
let node sum_sum_odd x = o where
  rec y = sum_odd x
  and o = sum_odd y
```

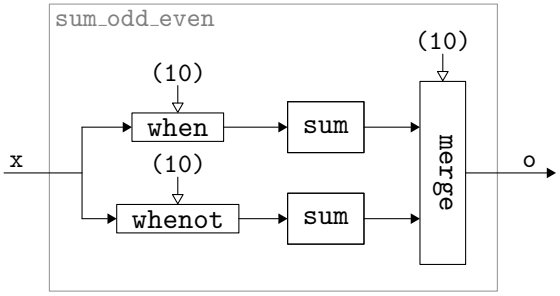


flot	valeurs	horloge
`x`	5 7 3 6 2 8 1 4 6 ...	`(1)`
`y = sum_odd x`	5 8 10 11 17 ...	`(1)` *on* `(10)`
`o = sum_odd y`	5 15 32 ...	`((1)` *on* `(10))` *on* `(10)` = `(1000)`

Le flot `o` est d'horloge `((1)` *on* `(10))` *on* `(10)`. On peut observer dans le tableau que cette horloge vaut `(1000)`.

L'opérateur `merge` permet de fusionner deux flots échantillonnés. Dans l'expression `merge c x1 x2`, les flots que l'on veut fusionner sont `x1` et `x2`, et `c` est la condition de fusion : si elle vaut `1`, c'est la valeur de `x1` qui doit être produite en sortie, et `x2` doit être absent, si elle vaut `0` c'est `x2` qui est consommé et retourné, et `x1` qui doit être absent.

Le nœud `sum_odd_even` sépare son flot d'entrée en deux flots plus lents complémentaires, leur applique l'opérateur d'intégration, puis les fusionne pour rendre une sortie présente aussi souvent que l'entrée.




```
let node sum_odd_even x = o where
  rec x_odd = x when (10)
  and x_even = x whenot (10)
  and o = merge (10) (sum x_odd) (sum x_even)
```

flot	valeurs	horloge
`x`	5 7 3 6 2 8 ...	`(1)`
`(10)`	1 0 1 0 1 0 ...	`(1)`
`x_odd`	5 3 2 ...	`(1)` *on* `(10)` = `(10)`
`x_even`	7 6 8 ...	`(1)` *on not* `(10)` = `(01)`
`sum x_odd`	5 8 10 ...	`(10)`
`sum x_even`	7 13 21 ...	`(01)`
`o`	5 7 8 13 10 21 ...	`(1)`

Communication synchrone et n-synchrone

De même qu'en Lustre, la composition de flots doit être synchrone, c'est-à-dire que les valeurs sur les flots doivent être présentes exactement aux instants où elles sont nécessaires aux calculs.

Par exemple, dans le nœud `foo` suivant, la constante `4` est un flot qui n'est présent qu'un instant sur deux, en même temps que `x when (10)` :

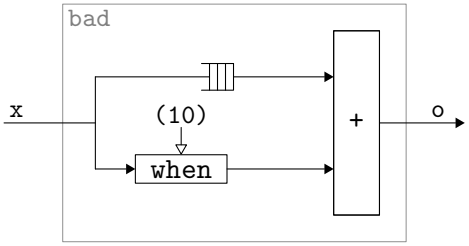
```
let node foo x = (x when (10)) + 4
```

flot	valeurs	horloge
`x`	5 7 3 6 2 8 ...	`(1)`
`x when (10)`	5 3 2 ...	`(1) on (10) = (10)`
`4`	4 4 4 ...	`(10)`
`x when (10) + 4`	9 7 6 ...	`(10)`

Cette condition de synchronisme permet de garantir une exécution sans buffers, et en particulier de rejeter le programme suivant :

```
let node bad x = x + (x when (10))
```

flot	valeurs	horloge
`x`	5 7 3 6 2 8 ...	`(1)`
`x when (10)`	5 3 2 ...	`(1) on (10) = (10)`



L'opérateur `+` doit être appliqué point à point : par conséquent, comme deux fois plus de valeurs arrivent sur le flot `x` que sur le flot `x when (10)`, l'opération ne peut pas être réalisée de manière synchrone. Le stockage des valeurs du flot `x` dans un buffer dans l'attente qu'elles soient traitées nécessiterait une mémoire infinie. L'absence de tels problèmes est garantie par les restrictions imposées par les langages synchrones [CPHP87]. Cependant, dans certains cas, on aimerait pouvoir composer des flots qui ne sont pas strictement synchrones. Par exemple, le programme suivant peut être exécuté en mémoire bornée, mais il est rejeté dans le cadre strictement synchrone car `x1` et `x2` n'ont pas la même horloge :

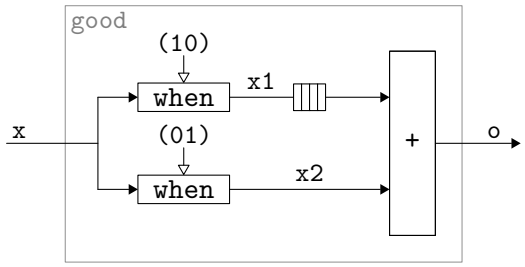
```
let node not_so_bad x = o where
  rec x1 = x when (10)
  and x2 = x when (01)
  and o = x1 + x2
```

flot	valeurs	horloge
`x1`	5 3 2 ...	`(10)`
`x2`	7 6 8 ...	`(01)`

Pour cela, on ajoute un opérateur `buffer` qui permet de spécifier les endroits où l'on autorise une composition nécessitant de stocker des flots dans des buffers, car leurs valeurs arrivent avant les instants où elles sont nécessaires. La communication est alors qualifiée de n-synchrone, n représentant la taille du buffer à mettre en place, par opposition à 0-synchrone qui ne nécessite pas de buffer.

Le nœud `not_so_bad` peut ainsi être accepté s'il est réécrit de la manière suivante :

```
let node good x = o where
  rec x1 = x when (10)
  and x2 = x when (01)
  and o = buffer(x1) + x2
```



flot	valeurs	horloge
`x1`	5 3 2 ...	`(10)`
`buffer(x1)`	5 3 2 ...	`(01)`
`x2`	7 6 8 ...	`(01)`
`buffer(x1) + x2`	12 9 10 ...	`(01)`

Un programme utilisant l'opérateur `buffer` est accepté seulement si l'horloge du flot stocké dans le buffer est *adaptable* à l'horloge où l'on veut consommer ce flot. Cette relation sera définie formellement dans le chapitre 3, en voici l'intuition :

L'horloge c_1 est *adaptable* à l'horloge c_2 si et seulement si tout flot d'horloge c_1 peut être consommé sur l'horloge c_2 par insertion d'un buffer de taille bornée.

Elle garantit non seulement que la mémoire à mettre en place pour bufferiser le flot en attendant son utilisation est de taille bornée, mais aussi que l'on ne risque aucune lecture dans une mémoire vide.

Ainsi, le nœud `bad` réécrit de la manière suivante en utilisant un buffer est tout de même rejeté :

```
let node also_bad x = buffer(x) + (x when (10))
```

flot	valeurs										horloge
`x`	5	7	3	6	2	8	1	4	6	...	`(1)`
`buffer(x)`	5		7		3		6		2	...	`(1)` *on* `(10)` = `(10)`
`x when (10)`	5		3		2		1		6	...	`(1)` *on* `(10)` = `(10)`

En effet, le nombre de valeurs du flot `x` à stocker dans le buffer croit infiniment. L'horloge `(1)` n'est pas adaptable à l'horloge `(10)`.

De même, si l'on réécrit le nœud `not_so_bad` en bufferisant `x2` plutôt que `x1`, le programme est rejeté car on devrait effectuer l'addition point à point aux instants d'arrivée de `x1`, instants auxquels la valeur correspondante de `x2` n'est pas encore dans le buffer. L'horloge `(01)` n'est pas adaptable à l'horloge `(10)`.

Pour plus d'exemples sur les programmes ne contenant pas de buffers, le lecteur pourra se référer aux manuels et tutoriaux des langages Lustre [Hal93] et Lucid Synchrone [Pou06]. La section suivante présente un ensemble de programmes simples profitant de l'opérateur de bufferisation.

2.1.3 Programmer avec des buffers

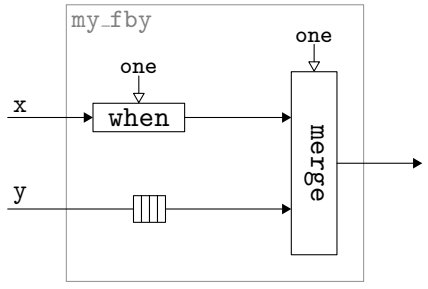
Nous présentons dans cette section des exemples tirant profit de l'expressivité de l'opérateur `buffer` qui fait l'originalité de Lucy-n.

Buffers initialisés

L'opérateur `fby` n'est autre qu'un buffer initialisé à une place qui produit ses sorties de manière synchrone à l'arrivée de ses entrées.

Cet opérateur peut donc être encodé à l'aide de l'opérateur `buffer` de la manière suivante :

```
let clock one = 1(0)

let node my_fby (x, y) =
  merge one (x when one) (buffer(y))
```



On constate ci-dessous que `my_fby x y` et `x fby y` sont équivalents :

flot	valeurs							horloge
`x`	5	7	3	6	2	8	...	`(1)`
`one`	1	0	0	0	0	0	...	`(1)`
`x when one`	5						...	`1(0)`
`y`	12	32	24	48	74	23	...	`(1)`
`buffer(y)`		12	32	24	48	74	...	*not* `1(0)`
`merge one (x when one) (buffer(y))`	5	12	32	24	48	74	...	`(1)`

En utilisant cette méthode, on peut programmer des buffers initialisés de taille quelconque. Par exemple, un buffer à trois places initialisé avec les trois premières valeurs du flot `x` peut être programmé ainsi :

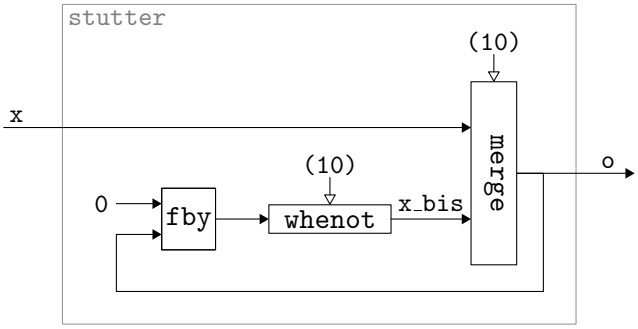
```
let clock three = 111(0)

let node my_fby3 (x, y) = merge three (x when three) (buffer(y))
```

flot	valeurs							horloge
`x`	5	7	3	6	2	8	...	`(1)`
`three`	1	1	1	0	0	0	...	`(1)`
`x when three`	5	7	3				...	`111(0)`
`y`	12	32	24	48	74	23	...	`(1)`
`buffer(y)`				12	32	24	...	*not* `111(0)`
`merge three (x when three) (buffer(y))`	5	12	32	24	48	74	...	`(1)`

Bégaiement

Un exemple classique de programme synchrone est le nœud `stutter` qui fait bégayer son flot d'entrée. Il est traditionnellement codé de la manière suivante :

```
let node stutter x = o where
  rec o = merge (10) x x_bis
  and x_bis = ((0 fby o) whenot (10))
```



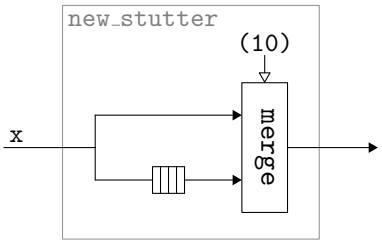
flot	valeurs													horloge
`x`	5		7		3		6		2		8		...	`(10)`
`(10)`	1	0	1	0	1	0	1	0	1	0	1	0	...	`(1)`
`0 fby o`	0	5	5	7	7	3	3	6	6	2	2	8	...	`(1)`
`x_bis = (0 fby o) whenot (10)`		5		7		3		6		2		8	...	*not* `(10)`
`o = merge (10) x x_bis`	5	5	7	7	3	3	6	6	2	2	8	8	...	`(1)`

L'entrée `x` du nœud `stutter` est consommée par `merge` lorsque l'horloge `(10)` vaut `1`. Par conséquent, l'hypothèse synchrone impose que ce flot ne soit présent qu'à ces instants-là. L'expression définissant `o` vaut l'entrée un instant sur deux, et la valeur précédente de la sortie aux autres instants. Comme la valeur précédente de la sortie n'est utilisée par le `merge` qu'un

instant sur deux, la condition synchrone impose que ce flot soit échantillonné avant d'être passé en argument à cet opérateur.

Grâce au nouvel opérateur `buffer`, ce nœud peut être codé de manière beaucoup plus simple :

```
let node new_stutter x = merge (10) x (buffer(x))
```



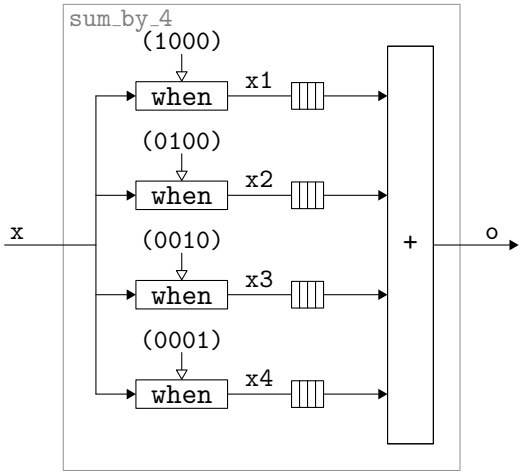
flot	valeurs													horloge
`x`	5		7		3		6		2		8		...	`(10)`
`(10)`	1	0	1	0	1	0	1	0	1	0	1	0	...	`(1)`
`buffer(x)`		5		7		3		6		2		8	...	*not* `(10)`
`merge (10) x (buffer(x))`	5	5	7	7	3	3	6	6	2	2	8	8	...	`(1)`

Le `merge` impose la même contrainte sur l'horloge de `x` qui doit n'être présent qu'un instant sur deux. Aux autres instants, il suffit de produire à nouveau la valeur précédente de `x`, que l'on stocke dans un buffer.

Séparation et réunion de flots

L'opérateur `buffer` permet de composer facilement des flots obtenus par séparation d'un flot d'entrée en plusieurs branches :

```
let node sum_by_4 x = o where
  rec x1 = x when (1000)
  and x2 = x when (0100)
  and x3 = x when (0010)
  and x4 = x when (0001)
  and o = buffer(x1)
          + buffer(x2)
          + buffer(x3)
          + buffer(x4)
```

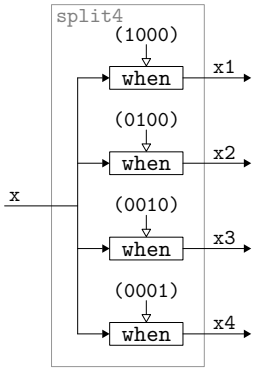


flot	valeurs										horloge
`x`	5	7	3	6	2	8	1	4	6	...	`(1)`
`x1 = x when (1000)`	5				2				6	...	`(1000)`
`x2 = x when (0100)`		7				8				...	`(0100)`
`x3 = x when (0010)`			3				1			...	`(0010)`
`x4 = x when (0001)`				6				4		...	`(0001)`
`buffer(x1)`				5				2		...	`(0001)`
`buffer(x2)`				7				8		...	`(0001)`
`buffer(x3)`				3				1		...	`(0001)`
`buffer(x4)`				6				4		...	`(0001)`
`o`				21				15		...	`(0001)`

Dans un cadre strictement synchrone, les flots `x1` à `x4` n'auraient pas pu être composés car ils ne sont pas présents aux mêmes instants. Ceux-ci auraient donc du être transformés par le programmeur (un peu comme dans le nœud `stutter`) afin d'être ramenés à la même horloge, travail fastidieux et source d'erreurs.

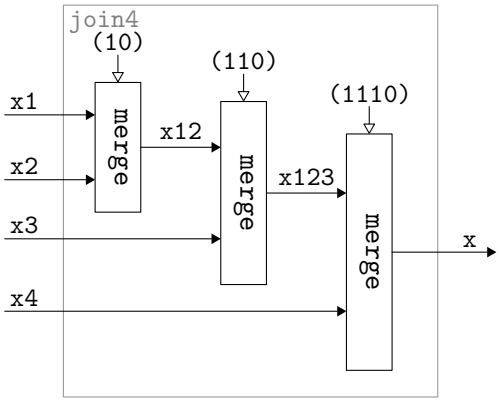
On peut isoler le premier traitement de `sum_by_4` dans un nœud qui sépare un flot d'entrée en quatre branches :

```
let node split4 x = (x1, x2, x3, x4) where
  rec x1 = x when (1000)
  and x2 = x when (0100)
  and x3 = x when (0010)
  and x4 = x when (0001)
```



L'opérateur `join4` dual à `split4` s'écrit :

```
let node join4 (x1, x2, x3, x4) = x where
  rec x12 = merge (10) x1 x2
  and x123 = merge (110) x12 x3
  and x = merge (1110) x123 x4
```



flot	valeurs										horloge
`x1`	5				2				6	...	`(1000)`
`x2`		7				8				...	`(0100)`
`(10)`	1	0			1	0			1	...	`(1100)`
`x12 = merge (10) x1 x2`	5	7			2	8			6	...	`(1100)`
`x3`			3				1			...	`(0010)`
`(110)`	1	1	0		1	1	0		1	...	`(1110)`
`x123 = merge (110) x12 x3`	5	7	3		2	8	1		6	...	`(1110)`
`x4`				6				4		...	`(0001)`
`(1110)`	1	1	1	0	1	1	1	0	1	...	`(1)`
`x = merge (1110) x123 x4`	5	7	3	6	2	8	1	4	6	...	`(1)`

L'opérateur `join4` fusionne les quatre flots en prenant à tour de rôle une valeur sur chacun d'eux. Il est programmé de manière systématique en intégrant successivement chacun des flots grâce à l'opérateur `merge`.

Les opérateurs `split4` et `join4` produisent ou consomment des flots qui ne sont pas synchrones. Si l'on veut appliquer des opérateurs synchrones sur les quadruplets de flots produits, il est donc très utile de disposer de l'opérateur `buffer` pour les synchroniser, comme dans le nœud `sum_by_4`. Si l'on veut appliquer l'opérateur `join4` à un quadruplet de flots synchrones, il faut cette fois utiliser l'opérateur `buffer` pour désynchroniser les quatre flots.

Le langage Lucy-n est suffisamment expressif pour programmer l'application *Picture in Picture* présentée dans le chapitre d'introduction. Le code source de cette application, ainsi que celui d'un encodeur de canal GSM sont présentés dans les chapitres 6 et 9.

2.2 Sémantique

Dans cette section, nous donnons au langage Lucy-n une sémantique dénotationnelle sur les séquences finies et infinies suivant la formulation de Kahn [Kah74], puis une sémantique dénotationnelle n-synchrone, qui caractérise les rythmes d'évolution des flots ainsi que le contenu des buffers. Cette dernière s'appuie la formulation donnée dans [Pou02, Ham02]. Nous rappelons d'abord des propriétés classiques sur les séquences finies et infinies, et introduisons les notations utiles à la définition des deux sémantiques.

Séquences finies et infinies. Si T est un ensemble, T^∞ désigne l'ensemble des séquences finies ou infinies d'éléments de l'ensemble T ($T^\infty = T^* + T^\omega$). La séquence vide est notée ε et $x.s$ est la séquence dont la tête est l'élément x et la queue est la séquence s. On note $\leq$ l'ordre préfixe sur les séquences, c'est-à-dire $s \leq s'$ si s est un préfixe de s' (pour tout s, $\varepsilon \leq s$). On appelle *chaîne* dans l'ensemble ordonné $(T^\infty, \leq)$ tout sous-ensemble non vide de T^∞ totalement ordonné par $\leq$.

L'ensemble ordonné $D = (T^\infty, \leq)$ est un ordre partiel complet (CPO), car $\leq$ est un ordre partiel, T^∞ a un plus petit élément, et toute chaîne de D a une borne supérieure.

Une fonction $f : D_1 \to D_2$ est *monotone* si et seulement si pour tout $s \in D_1$, pour tout $s' \in D_1$, $s \leq s' \Rightarrow f(s) \leq f(s')$. Une fonction $f : D_1 \to D_2$ est *continue* si et seulement si pour toute chaîne C dans D_1, on a $f(sup(C)) = sup(\{f(d),\ d \in C\})$. Toute fonction continue $f : D \to D$ sur un CPO D admet un plus petit point fixe $fix(f) = lim_{n\to\infty} f^n(\varepsilon)$, avec $f^0(x) = x$ et $f^{n+1}(x) = f(f^n(x))$ (théorème de Kleene).

Si D_1 et D_2 sont des CPO, alors $D_1 \times D_2$ vu comme l'ensemble des paires (x_1, x_2) formées d'un élément x_1 de D_1 et un élément x_2 de D_2 est un CPO en prenant l'ordre point-à-point tel que $(x_1, x_2) \leq (y_1, y_2) \Leftrightarrow x_1 \leq y_1 \wedge x_2 \leq y_2$. Si f est une fonction continue de $D_1 \times \ldots \times D_k$ vers $D_1 \times \ldots \times D_k$, nous écrirons $fix(f) = lim_{n\to\infty} f^n(\varepsilon, \ldots, \varepsilon)$ le plus petit point fixe de f.

Notations pour la sémantique. On notera ρ les environnements, et $\rho(x)$ la valeur associée à la variable x dans l'environnement ρ. L'environnement $\rho + [x \leftarrow v]$ est l'environnement ρ auquel on a ajouté l'association de la valeur v à la variable x. L'environnement $\rho + \rho'$ est l'environnement contenant les associations de l'environnement ρ et les associations de l'environnement ρ' (aucune variable ne doit apparaître à la fois dans ρ et ρ').

L'interprétation d'une expression e dans un environnement ρ sera notée $[\![e]\!]_\rho$. Cette notation sera aussi utilisée pour la dénotation des équations et des déclarations.

Enfin, pour donner l'interprétation des primitives comme transformateurs de flots, nous utilisons le sigle $\sharp$ en exposant du nom de celles-ci afin de distinguer la construction syntaxique de son interprétation.

2.2.1 Sémantique de Kahn

La sémantique décrite dans cette section suit le modèle de la sémantique de Kahn [Kah74]. Chaque nœud de calcul est interprété comme une fonction monotone et continue (pour l'ordre

$$
\begin{array}{lcl}
op^\sharp(v_1.s_1, v_2.s_2) & = & v.op^\sharp(s_1, s_2) \text{ où } v = op(v_1, v_2) \\
\texttt{fby}^\sharp(v_1.s_1, s_2) & = & v_1.s_2 \\
\texttt{when}^\sharp(v_1.s_1, \texttt{1}.w) & = & v_1.\texttt{when}^\sharp(s_1, w) \\
\texttt{when}^\sharp(v_1.s_1, \texttt{0}.w) & = & \texttt{when}^\sharp(s_1, w) \\
\texttt{whenot}^\sharp(v_1.s_1, \texttt{0}.w) & = & v_1.\texttt{whenot}^\sharp(s_1, w) \\
\texttt{whenot}^\sharp(v_1.s_1, \texttt{1}.w) & = & \texttt{whenot}^\sharp(s_1, w) \\
\texttt{merge}^\sharp(\texttt{1}.w, v_1.s_1, s_2) & = & v_1.\texttt{merge}^\sharp(w, s_1, s_2) \\
\texttt{merge}^\sharp(\texttt{0}.w, s_1, v_2.s_2) & = & v_2.\texttt{merge}^\sharp(w, s_1, s_2)
\end{array}
$$

FIG. 2.2 – Sémantique de Kahn des primitives du langage.

préfixe sur les séquences) qui transforme des séquences finies ou infinies d'éléments. Intuitivement, un nœud est une fonction monotone si l'on peut calculer progressivement les éléments des séquences de sortie en parcourant les éléments des séquences d'entrées. Elle est continue s'il n'a pas besoin d'attendre d'avoir consommé une infinité d'entrées avant de commencer à produire des sorties. Comme ces propriétés sont conservées par la composition de fonctions, si toutes les primitives du langage sont monotones et continues, alors les nœuds qui sont définis par composition de ces primitives le sont aussi, et le programme lui-même est une fonction monotone et continue. La sémantique de ce programme est alors le plus petit point fixe de la fonction qui lui est associée, qui existe car l'ensemble des séquences finies ou infinies muni de l'ordre préfixe est un CPO.

La sémantique de Kahn des primitives du langage est décrite sur la figure 2.2.

- *op* applique point à point un opérateur (sur des valeurs scalaires) importé ;
- `fby` est le délai unitaire initialisé : il concatène la tête de son premier argument avec l'intégralité de son second argument ;
- `when` est l'opérateur d'échantillonnage : il produit sur sa sortie la valeur de son entrée seulement si l'horloge d'échantillonnage vaut `1`, et ne produit rien dans le cas contraire ;
- `whenot` est l'opérateur d'échantillonnage symétrique qui produit la valeur de son entrée sur sa sortie seulement si l'horloge d'échantillonnage vaut `0` ;
- `merge` est l'opérateur de fusion : il reporte en sortie la valeur de sa première entrée si l'horloge de fusion vaut `1`, et celle de sa deuxième entrée dans le cas contraire.

Il faut ajouter à ces règles le traitement des flots vides : ε. Les opérateurs se réduisent en ε si un de leurs arguments est un flot vide (ε est absorbant), mis à part pour l'opérateur `fby` qui supporte un deuxième argument vide : $\texttt{fby}^\sharp(v_1.s_1, \varepsilon) = v_1.\varepsilon$.

Toutes les fonctions présentées ci-dessus sont continues.

La sémantique des expressions du langage est définie dans la figure 2.3. Cette définition structurelle utilise l'interprétation des primitives donnée précédemment. $[\![e]\!]_\rho$ retourne le flot des valeurs correspondant à l'évaluation de l'expression e dans l'environnement ρ. Cet environnement est en fait un triplet d'environnements : ρ_s qui contient des flots de valeurs correspondant aux variables libres de e, ρ_n qui contient les fermetures associées aux nœuds et ρ_c qui contient les mots binaires associés aux horloges. Si les éléments des flots appartiennent à un ensemble T, et les identificateurs des flots, des nœuds et des horloges sont

respectivement rassemblés dans des ensembles Var_s, Var_n et Var_c, on a donc :

$$\begin{array}{lcl} Stream(T) & = & T^\infty \\ \rho_s & : & Var_s \to Stream(T) \\ \rho_n & : & Var_n \to (Stream(T) \to Stream(T)) \\ \rho_c & : & Var_c \to Stream(Bool) \end{array}$$

Si $\rho = (\rho_s, \rho_n, \rho_c)$ et $\rho' = (\rho'_s,\ \rho'_n,\ \rho'_c)$ alors $\rho + \rho' = (\rho_s + \rho'_s,\ \rho_n + \rho'_n,\ \rho_c + \rho'_c)$. Nous noterons $\rho + [z \leftarrow v]$ pour ajouter l'association $z \leftarrow v$ dans la partie adéquate du triplet ρ, selon que z est un flot, un nœud ou une horloge.

L'interprétation de e `where rec` *eqs* utilise comme environnement supplémentaire l'interprétation de l'ensemble d'équations *eqs*. Si cet ensemble est x_1 `=` e_1 `and ... and` x_k `=` e_k, son interprétation est un environnement associant à chaque variable x_i l'interprétation de l'expression e_i :

$$[\![x_1 \texttt{ = } e_1 \texttt{ and ... and } x_k \texttt{ = } e_k]\!]_\rho \quad = \quad [x_1 \leftarrow {x_1}^\sharp, \ldots, x_k \leftarrow {x_k}^\sharp]$$

où ${x_1}^\sharp, \ldots, {x_k}^\sharp = \mathit{fix}(\lambda s_1, \ldots s_k.[\![e_1]\!]_{\rho+[x_1 \leftarrow s_1, \ldots, x_k \leftarrow s_k]}, \ldots, [\![e_k]\!]_{\rho+[x_1 \leftarrow s_1, \ldots, x_k \leftarrow s_k]})$.

L'interprétation des opérations `when`, `whenot` et `merge` utilise l'interprétation de l'horloge *ce* qui leur est passée en argument : $[\![ce]\!]^{ce}_\rho$ est l'évaluation d'une expression d'horloges en un mot binaire infini. Celle-ci doit être fournie avec le langage des horloges. Par exemple, pour le langage d'horloges considéré dans ce chapitre, la sémantique des expressions d'horloges est :

$$\begin{array}{rcl} [\![u(v)]\!]^{ce}_\rho & = & u.(lim_{n \to +\infty} v^n) \text{ avec } v^0 = \varepsilon \text{ et } v^{n+1} = v.v^n \\ [\![c]\!]^{ce}_\rho & = & \rho_c(c) \text{ si } \rho = (\rho_s, \rho_n, \rho_c) \end{array}$$

L'opération `buffer` ne fait que conserver son flot d'entrée un certain nombre d'instants avant de le rendre à l'identique en sortie. La sémantique de Kahn ne permettant pas de parler de la notion de temps, l'interprétation de cette opération est donc l'identité.

La sémantique d'un programme est définie de la manière suivante :

$$\begin{array}{rcl} [\![\texttt{let node } f\ x \texttt{ = } e]\!]_\rho & = & \rho + [f \leftarrow \lambda s.[\![e]\!]_{\rho+[x \leftarrow s]}] \\ [\![\texttt{let clock } c \texttt{ = } ce]\!]_\rho & = & \rho + [c \leftarrow [\![ce]\!]^{ce}_\rho] \\ [\![d_1; d_2]\!]_\rho & = & [\![d_2]\!]_{\rho+\rho_1} \text{ où } \rho_1 = [\![d_1]\!]_\rho \end{array}$$

Si l'on considère un programme d dont f est le nœud principal, l'évaluation du programme dans un environnement où le flot d'entrée est I est définie par :

$$\rho_n(f)\ I \text{ où } (\rho_s, \rho_n, \rho_c) = [\![d]\!]_{(\varnothing,\varnothing,\varnothing)}$$

La sémantique que nous venons de définir permet de connaître les suites de valeurs attendues en sortie d'un programme pouvant être exécuté sans buffers sur les canaux de communication. La notion de temps n'étant pas présente dans cette sémantique, elle donne aussi un sens aux programmes ne vérifiant pas cette condition (comme les nœuds `bad` et `not_so_bad` de la section 2.1). La section suivante présente une sémantique qui ne donne un sens qu'aux programmes pouvant être exécutés avec des buffers bornés aux points d'application de l'opérateur de bufferisation et sans buffers partout ailleurs.

$$
\begin{array}{rcl}
[\![i]\!]_\rho & = & i.[\![i]\!]_\rho \\
[\![x]\!]_\rho & = & \rho_s(x) \text{ si } \rho = (\rho_s, \rho_n, \rho_c) \\
\\
[\![ce]\!]_\rho & = & [\![ce]\!]_\rho^{ce} \\
\\
[\![(e_1\,,e_2)]\!]_\rho & = & ([\![e_1]\!]_\rho, [\![e_2]\!]_\rho) \\
[\![\texttt{fst}\ e]\!]_\rho & = & s_1 \text{ si } [\![e]\!]_\rho = (s_1\,,s_2) \\
[\![\texttt{snd}\ e]\!]_\rho & = & s_2 \text{ si } [\![e]\!]_\rho = (s_1\,,s_2) \\
\\
[\![op(e_1, e_2)]\!]_\rho & = & op^\sharp([\![e_1]\!]_\rho, [\![e_2]\!]_\rho) \\
\\
[\![f\ e]\!]_\rho & = & \rho_n(f)\ [\![e]\!]_\rho \text{ si } \rho = (\rho_s, \rho_n, \rho_c) \\
\\
[\![e\ \texttt{where rec}\ eqs]\!]_\rho & = & [\![e]\!]_{\rho+\rho_s} \text{ où } \rho_s = [\![eqs]\!]_\rho \\
\\
[\![e_1\ \texttt{fby}\ e_2]\!]_\rho & = & \texttt{fby}^\sharp([\![e_1]\!]_\rho, [\![e_2]\!]_\rho) \\
\\
[\![e\ \texttt{when}\ ce]\!]_\rho & = & \texttt{when}^\sharp([\![e]\!]_\rho, [\![ce]\!]_\rho) \\
[\![e\ \texttt{whenot}\ ce]\!]_\rho & = & \texttt{whenot}^\sharp([\![e]\!]_\rho, [\![ce]\!]_\rho) \\
[\![\texttt{merge}\ ce\ e_1\ e_2]\!]_\rho & = & \texttt{merge}^\sharp([\![ce]\!]_\rho, [\![e_1]\!]_\rho, [\![e_2]\!]_\rho) \\
\\
[\![\texttt{buffer}(e)]\!]_\rho & = & [\![e]\!]_\rho
\end{array}
$$

FIG. 2.3 – Sémantique de Kahn des expressions du langage.

2.2.2 Sémantique flot de données n-synchrone

Nous donnons maintenant à Lucy-n une sémantique n-synchrone, basée sur la théorie des langages synchrones [Pou02, Ham02]. L'absence de valeur sur les flots est rendue explicite. Pour cela, l'ensemble des valeurs qui peuvent être prises par un flot est complété par une valeur spéciale $\perp$ représentant l'absence. Le nombre de places disponibles dans les buffers et leur contenu sont eux aussi rendus explicites.

Nous définissons l'ensemble *ClockedStream*(T) des *séquences avec horloge* comme l'ensemble des séquences finies et infinies d'éléments de l'ensemble $T_\perp = T \cup \{\perp\}$.

$$
\begin{array}{rcl}
T_\perp & = & T \cup \{\perp\} \\
\mathit{ClockedStream}(T) & = & (T_\perp)^\infty
\end{array}
$$

Une séquence avec horloge est faite de valeurs présentes ou absentes. L'horloge d'une telle séquence est un mot binaire indiquant si une valeur est présente. Pour cela, on définit la fonction *clock* des séquences avec horloge vers les mots binaires :

$$
\begin{array}{rcl}
\mathit{clock}(\varepsilon) & = & \varepsilon \\
\mathit{clock}(v.s) & = & \texttt{1}.\mathit{clock}(s) \\
\mathit{clock}(\perp.s) & = & \texttt{0}.\mathit{clock}(s)
\end{array}
$$

Nous utiliserons la lettre v pour les valeurs présentes. Par conséquent, $v.s$ désigne un flot dont le premier élément est présent et dont la suite est s alors que $\perp.s$ désigne un flot dont le premier élément est absent.

$$
\begin{array}{lcl}
\texttt{const}^\sharp(i, \texttt{1}.w) & = & i.\texttt{const}^\sharp(i, w) \\
\texttt{const}^\sharp(i, \texttt{0}.w) & = & \bot.\texttt{const}^\sharp(i, w) \\
\texttt{cond}^\sharp(\mathit{cond}, \texttt{0}.w) & = & \bot.\texttt{cond}^\sharp(\mathit{cond}, w) \\
\texttt{cond}^\sharp(b.\mathit{cond}, \texttt{1}.w) & = & b.\texttt{cond}^\sharp(\mathit{cond}, w) \\
\mathit{op}^\sharp(\bot.s_1, \bot.s_2) & = & \bot.\mathit{op}^\sharp(s_1, s_2) \\
\mathit{op}^\sharp(v_1.s_1, v_2.s_2) & = & v.\mathit{op}^\sharp(s_1, s_2) \text{ où } v = \mathit{op}(v_1, v_2) \\
\texttt{fby}^\sharp(\bot.s_1, \bot.s_2) & = & \bot.\texttt{fby}^\sharp(s_1, s_2) \\
\texttt{fby}^\sharp(v_1.s_1, v_2.s_2) & = & v_1.\texttt{fby1}^\sharp(v_2, s_1, s_2) \\
\texttt{fby1}^\sharp(v, \bot.s_1, \bot.s_2) & = & \bot.\texttt{fby1}^\sharp(v, s_1, s_2) \\
\texttt{fby1}^\sharp(v, v_1.s_1, v_2.s_2) & = & v.\texttt{fby1}^\sharp(v_2, s_1, s_2) \\
\texttt{when}^\sharp(\bot.s_1, \bot.s_2) & = & \bot.\texttt{when}^\sharp(s_1, s_2) \\
\texttt{when}^\sharp(v_1.s_1, \texttt{1}.s_2) & = & v_1.\texttt{when}^\sharp(s_1, s_2) \\
\texttt{when}^\sharp(v_1.s_1, \texttt{0}.s_2) & = & \bot.\texttt{when}^\sharp(s_1, s_2) \\
\texttt{whenot}^\sharp(\bot.s_1, \bot.s_2) & = & \bot.\texttt{whenot}^\sharp(s_1, s_2) \\
\texttt{whenot}^\sharp(v_1.s_1, \texttt{1}.s_2) & = & \bot.\texttt{whenot}^\sharp(s_1, s_2) \\
\texttt{whenot}^\sharp(v_1.s_1, \texttt{0}.s_2) & = & v_1.\texttt{whenot}^\sharp(s_1, s_2) \\
\texttt{merge}^\sharp(\bot.s_0, \bot.s_1, \bot.s_2) & = & \bot.\texttt{merge}^\sharp(s_0, s_1, s_2) \\
\texttt{merge}^\sharp(\texttt{1}.s_0, v.s_1, \bot.s_2) & = & v.\texttt{merge}^\sharp(s_0, s_1, s_2) \\
\texttt{merge}^\sharp(\texttt{0}.s_0, \bot.s_1, v.s_2) & = & v.\texttt{merge}^\sharp(s_0, s_1, s_2) \\
\texttt{buffer}^\sharp(v.s, n, \bot.s_1, \texttt{1}.w) & = & v.\texttt{buffer}^\sharp(s, n+1, s_1, w) \\
\texttt{buffer}^\sharp(v.s, n, v_1.s_1, \texttt{1}.w) & = & v.\texttt{buffer}^\sharp(s.v_1, n, s_1, w) \\
\texttt{buffer}^\sharp(\varepsilon, n, v_1.s_1, \texttt{1}.w) & = & v_1.\texttt{buffer}^\sharp(\varepsilon, n, s_1, w) \\
\texttt{buffer}^\sharp(s, n, \bot.s_1, \texttt{0}.w) & = & \bot.\texttt{buffer}^\sharp(s, n, s_1, w) \\
\texttt{buffer}^\sharp(s, n, v_1.s_1, \texttt{0}.w) & = & \bot.\texttt{buffer}^\sharp(s.v_1, n-1, s_1, w) \text{ si } n > 0
\end{array}
$$

FIG. 2.4 – Sémantique n-synchrone des primitives du langage.

La sémantique n-synchrone est définie par $[\![e]\!]^\perp_\rho$ qui retourne la séquence avec horloge correspondant à l'évaluation de l'expression e dans l'environnement $\rho = (\rho_s, \rho_n, \rho_c)$, avec :

$$
\begin{array}{lcl}
\rho_s & : & \mathit{Var}_s \rightarrow \mathit{ClockedStream}(T) \\
\rho_n & : & \mathit{Var}_n \rightarrow (\mathit{ClockedStream}(T) \rightarrow \mathit{ClockedStream}(T)) \\
\rho_c & : & \mathit{Var}_c \rightarrow \mathit{Stream}(\mathit{Bool})
\end{array}
$$

L'interprétation sur les séquences avec horloge des primitives de notre langage est décrite sur la figure 2.4. L'aspect synchrone de cette sémantique est garanti par l'absence de certaines règles. Par exemple, il n'existe pas de règle pour évaluer $\mathit{op}^\sharp(v_1.s_1, \bot.s_2)$ car dans le paradigme synchrone un opérateur binaire doit recevoir ses deux paramètres de manière simultanée. En ce qui concerne le caractère n-synchrone, il est garanti par l'absence de règle pour ajouter un élément dans un buffer plein, ou consommer une valeur dans un buffer vide. Nous détaillons ci-dessous l'interprétation de chacune des primitives.

- La primitive `const` produit un flot constant à partir d'une valeur scalaire. Cette primitive produit des valeurs selon les besoins de l'environnement. Pour cette raison, elle a besoin d'un argument supplémentaire indiquant son horloge de production. Par conséquent, $\texttt{const}^\sharp(i, w)$ désigne un flot constant d'horloge w ($\mathit{clock}(\texttt{const}^\sharp(i, w)) = w$).
- La fonction `cond` produit une séquence avec horloge dont les valeurs sont celles d'une suite booléenne sans horloge *cond*, sur le rythme défini par une horloge w.
- Un opérateur binaire importé impose que ses deux opérandes soient synchrones (tous les deux présents ou tous les deux absents).
- Les arguments et le résultat de `fby` doivent avoir la même horloge. `fby` correspond à une machine à deux états : tant que ses deux arguments sont absents, il n'émet rien et reste dans son état initial ($\texttt{fby}^\sharp$). Quand ils sont présents tous les deux, il émet son premier argument et entre dans un nouvel état ($\texttt{fby1}^\sharp$) stockant la valeur précédente de son second argument. Dans cet état, il émet la valeur en stock à chaque fois que ses deux arguments sont présents. Ceci permet de vérifier le caractère synchrone de ces derniers.
- Le résultat de l'opérateur d'échantillonnage `when` n'est présent que si son entrée est présente et la condition d'échantillonnage est présente et vaut `1`.
- La définition de `merge` établit qu'une branche doit être absente lorsque l'autre branche est présente.
- Tout comme pour l'opérateur `const`, la production ou non d'une valeur par l'opérateur `buffer` dépend de l'environnement. L'horloge de sa sortie (w) doit donc lui être passée en paramètre. La sémantique n-synchrone ne donne un sens qu'à des programmes utilisant des buffers bornés. Notre opérateur prend donc aussi en premier paramètre le contenu courant du buffer, et en second paramètre le nombre de places restantes (n). Il produit une valeur lorsque son horloge de sortie vaut `1`, à condition d'avoir au moins une valeur en stock ou une valeur en entrée, et stocke les valeurs d'entrée à leur arrivée, à condition que le nombre de places restantes soit non nul.

Il faut ajouter à ces règles le traitement des flots vides. Tout comme dans la sémantique de Kahn, les opérateurs $op^\sharp$, $\texttt{when}^\sharp$, $\texttt{whenot}^\sharp$ et $\texttt{merge}^\sharp$ se réduisent en ε si un de leurs arguments est un flot vide. ε est donc absorbant pour ces opérateurs. Les règles pour les opérateurs `fby` et `buffer` appliqués à au moins un argument ε sont détaillées ci-dessous :

$$\begin{array}{lcllcl}
\texttt{fby}^\sharp(\varepsilon, s_2) & = & \varepsilon & \texttt{fby1}^\sharp(\varepsilon, s_1, s_2) & = & \varepsilon \\
\texttt{fby}^\sharp(v_1.s_1, \varepsilon) & = & v_1.\varepsilon & \texttt{fby1}^\sharp(v, \varepsilon, s_2) & = & v.\varepsilon \\
\texttt{fby}^\sharp(\bot.s_1, \varepsilon) & = & \bot.\varepsilon & \texttt{fby1}^\sharp(v, s_1, \varepsilon) & = & v.\varepsilon
\end{array}$$

$$\texttt{buffer}^\sharp(s, n, \varepsilon, w) = \varepsilon$$

Toutes ces fonctions sur les séquences avec horloge sont continues. En particulier, la fonction $\texttt{buffer}^\sharp$ est monotone : étant donné une mémoire s et un nombre de places restantes n (ces paramètres ne sont pas des entrées du programme), pour tout couple d'entrées constitué d'un flot s_1 et d'une horloge w, et tout couple d'entrées s_1' et w' tels que $s_1 \leq s_1'$ et $w \leq w'$, on a $\texttt{buffer}^\sharp(s, n, s_1, w) \leq \texttt{buffer}^\sharp(s, n, s_1', w')$. Comme $\texttt{buffer}^\sharp$ conserve la longueur des flots, on peut en déduire qu'il est continu.

La sémantique ne porte pas exactement sur les expressions du noyau décrit en section 2.1, mais sur un langage où les constantes, les conditions d'échantillonnage et de fusion, ainsi que les buffers prennent en paramètre leur horloge de production. Les buffers prennent également

en paramètre le nombre de places dont ils disposent. On peut se ramener à ce langage par traduction des programmes du noyau :

$$\begin{array}{lcl} i & \rightsquigarrow & \texttt{const}(i, w) \\ ce & \rightsquigarrow & \texttt{cond}(ce, w) \\ \texttt{buffer}(e) & \rightsquigarrow & \texttt{buffer}(n, e', w) \text{ avec } e \rightsquigarrow e' \end{array}$$

Les arguments supplémentaires w et n sont inférés par le calcul d'horloge décrit en section 2.3. Nous reviendrons sur cette traduction dans la section 2.3.4.

La sémantique n-synchrone des expressions et des programmes du langage est définie de manière similaire à la sémantique de Kahn (voir la figure 2.3), mis à part pour les constantes, les conditions d'échantillonnage et de fusion, ainsi que les flots conservés dans un buffer, qui sont interprétés ainsi :

$$\begin{array}{lcl} [\![\texttt{const}(i, w)]\!]^{\perp}_{\rho} & = & \texttt{const}^{\sharp}(i, w) \\ [\![\texttt{cond}(ce, w)]\!]^{\perp}_{\rho} & = & \texttt{cond}^{\sharp}([\![ce]\!]^{ce}_{\rho}, w) \\ [\![\texttt{buffer}(n, e, w)]\!]^{\perp}_{\rho} & = & \texttt{buffer}^{\sharp}(\varepsilon, n, [\![e]\!]^{\perp}_{\rho}, w) \end{array}$$

Ces opérateurs peuvent produire ou non en fonction des besoins du nœud consommant leur sortie. C'est pour cette raison que nous leur ajoutons un argument supplémentaire d'horloge (w), indiquant les instants où une valeur doit être produite. D'autre part, l'opérateur `buffer` transporte une mémoire qui doit être initialisée à la séquence vide ε. La taille de cette mémoire (n) est elle aussi synthétisée par le calcul d'horloge, et passée en argument supplémentaire à l'opérateur.

La sémantique présentée dans cette section est partielle : elle ne permet de donner un sens qu'aux programmes n-synchrones, c'est-à-dire pouvant être exécutés sans buffers, sauf aux points d'application de l'opérateur `buffer` où l'on accepte une mémoire de taille bornée. Le calcul d'horloge présenté dans la section suivante permet de garantir statiquement qu'un programme vérifie cette condition. Il calcule aussi les horloges de sortie des constantes, des conditions d'échantillonnage, et des buffers, ainsi que la taille de ces derniers.

2.3 Calcul d'horloge

La cohérence des données manipulées dans les programmes synchrones peut être vérifiée statiquement par des règles de typage que nous ne détaillerons pas ici (voir par exemple [CGHP04]). Les problèmes de causalité peuvent être détectés statiquement par un autre calcul de types, décrit dans [CP01, Cuo02]. La cohérence des horloges peut elle aussi être vérifiée par un calcul de types, appelé *calcul d'horloge* [Pou02].

Le calcul d'horloge traditionnel n'autorise qu'une communication strictement synchrone, il ne contient donc qu'une notion d'égalité d'horloges. Afin de pouvoir typer l'opérateur `buffer`, le calcul d'horloge doit être enrichi de la notion *d'adaptabilité* d'une horloge à une autre, qui garantit qu'une communication est n-synchrone. Ceci est mis en place par l'introduction d'une règle de sous-typage appliquée aux points d'utilisation de l'opérateur `buffer`. Intuitivement, un type d'horloges ck_1 est un sous-type d'un type d'horloges ck_2 si toute horloge correspondant au type ck_1 est adaptable à toute horloge correspondant au type ck_2.

2.3.1 Types d'horloges

Tout comme les types de données abstraient les données, les types d'horloges abstraient les horloges. Par exemple, le nœud `sum_odd` présenté dans la section 2.1 a le type d'horloges $\forall\alpha.\,\alpha \to \alpha$ `on (10)`. Cela signifie que quelque soit l'horloge $c = clock(x)$ du flot d'entrée x, l'horloge du flot de sortie sera c *on* `(10)`.

Dans la suite, nous ne reparlerons plus de types de données et nous nous permettrons donc de nommer simplement "types" les types d'horloges.

Les types sont séparés en trois catégories : les schémas de types (σ) qui décrivent les horloges des sorties des nœuds en fonction des horloges de leurs entrées, les types des expressions (ct) et les types des flots (ck). Les types des flots peuvent décrire les horloges de flots échantillonnés (ck `on` ce, ck `on not` ce) et les horloges de flots quelconques (α).

$$\begin{array}{lcl} \sigma & ::= & \forall\beta_1,\ldots,\beta_m.\forall\alpha_1,\ldots,\alpha_n.\, ct \to ct \\ ct & ::= & \beta \mid ct \times ct \mid ck \\ ck & ::= & \alpha \mid ck \texttt{ on } ce \mid ck \texttt{ on not } ce \end{array}$$

La distinction entre ces trois catégories de types n'est pas étonnante. En effet, bien que les réseaux de Kahn n'aient pas de types d'horloges, il existe une claire distinction entre une fonction sur les flots (qui reçoit un type de la forme σ), un canal (qui reçoit un type de la forme ck) et un ensemble de canaux (qui reçoit un type de la forme ct).

L'environnement de typage H est un triplet contenant les types d'un ensemble de variables de flots x_1 à x_p, les types d'un ensemble de nœuds f_1 à f_m, et les valeurs d'un ensemble d'horloges c_1 à c_n.

$$\begin{array}{lcl} H & ::= & ([\,x_1 : ct_1,\ldots,x_p : ct_p\,], \\ & & [\,f_1 : \sigma_1,\ldots,f_m : \sigma_m\,], \\ & & [\,c_1 : ce_1,\ldots,c_n : ce_n\,]) \end{array}$$

Nous utiliserons la notation $H + [z : t]$ pour ajouter l'association $z : t$ dans la partie adéquate du triplet, selon que z est un flot, un nœud ou une horloge, et de même la notation $H(z)$ pour accéder à l'information associée à z dans l'environnement adéquat.

Un ensemble de contraintes C contient des contraintes de sous-typage portant sur des types de flots et notées $ck <: ck'$. L'union de deux ensembles de contraintes C_1 et C_2 est notée C_1, C_2.

$$C ::= \{ck_1 <: ck'_1; \ldots; ck_t <: ck'_m\}$$

Les types peuvent être instanciés et généralisés en suivant les règles ci-dessous. Ceci est un point clé pour obtenir une analyse modulaire.

$$\begin{array}{l} inst(\forall\beta_1,\ldots,\beta_m.\forall\alpha_1,\ldots,\alpha_n.\, ct) = \\ \quad \{\, ct' \mid ct' = ct[\beta_1 \leftarrow ct_1,\ldots,\beta_m \leftarrow ct_m, \alpha_1 \leftarrow ck_1,\ldots,\alpha_n \leftarrow ck_n]\,\} \\ \\ gen(ct, C) = \forall\beta_1,\ldots,\beta_m.\forall\alpha_1,\ldots,\alpha_n.\, ct' \\ \quad \text{où } ct' = \theta(ct) \text{ tel que } \theta(C) \text{ est satisfait} \\ \quad \text{et } \{\beta_1,\ldots,\beta_m,\alpha_1,\ldots,\alpha_n\} = FV(ct') \end{array}$$

Un type peut être instancié en remplaçant les variables quantifiées universellement par des types. Un type ct contraint par C peut être généralisé :

- en instanciant les variables contraintes dans C par des types de flots satisfaisant ces contraintes (c'est le rôle de la substitution θ) ;
- puis en quantifiant universellement les variables du type ainsi obtenu ($FV(ct')$ désigne l'ensemble des variables libres dans ct').

Nous avons choisi de résoudre les contraintes au moment de la généralisation, plutôt que de les transporter dans le type. C'est une restriction par rapport aux techniques traditionnelles de sous-typage [Pie05] qui utilisent une règle proche de celle-ci :

$$gen(ct, C) = \forall \beta_1, \ldots, \beta_m.\ \forall \alpha_1, \ldots, \alpha_n[C].\ ct$$
$$\text{si } C \text{ est satisfaisable}$$
$$\text{et } \{\beta_1, \ldots, \beta_m, \alpha_1, \ldots, \alpha_n\} = FV(ct)$$

Cette règle se lit "pour tous types d'expressions β_1 à β_m, pour tous types de flots α_1 à α_n tels que C est vérifié, le nœud a le type ct". Notre choix a pour but de pouvoir effectuer une compilation séparée des nœuds. En effet, si on ne résout pas les contraintes lors de la définition d'un nœud, on ne connaît pas la taille des buffers nécessaires à l'exécution de celui-ci indépendamment de son contexte d'appel. À chaque contexte d'appel, une taille différente peut-être nécessaire et on ne peut donc pas compiler ce nœud modulairement. Le choix que nous avons fait implique qu'un buffer placé sur un flot d'entrée qui n'est utilisé qu'une fois dans le nœud (ou sur un flot de sortie qui n'est pas réutilisé dans le nœud) aura une taille dépendant uniquement de l'algorithme de résolution des contraintes. [1] En effet ce buffer n'est soumis à aucune vraie contrainte puisque les types d'horloges des entrées et des sorties sont inconnus. Pour être utiles, de tels buffers doivent donc être placés dans le programme appelant.

2.3.2 Règles de typage

Le but du calcul d'horloge est de produire des jugements de la forme $H \vdash e : ct \,|\, C$, signifiant que "l'expression e a le type d'horloges ct dans l'environnement H, sous les contraintes de l'ensemble C". Les règles définissant ce prédicat sont expliquées ci-dessous et récapitulées dans la figure 2.5, page 45.

– Un flot constant peut avoir n'importe quel type ck :

$$\text{(IM)} \quad H \vdash i : ck \,|\, \varnothing$$

– Une expression d'horloges peut elle aussi avoir n'importe quel type ck. Le jugement $H \vdash ce$ signifie que tous les noms libres apparaissant dans ce sont définis dans H.

$$\text{(CE)} \ \frac{H \vdash ce}{H \vdash ce : ck \,|\, \varnothing}$$

– Le type d'une variable se trouve dans l'environnement :

$$\text{(VAR)} \quad H \vdash x : H(x) \,|\, \varnothing$$

– Les deux éléments d'une paire peuvent avoir des types différents :

$$\text{(PAIR)} \ \frac{H \vdash e_1 : ct_1 \,|\, C_1 \quad H \vdash e_2 : ct_2 \,|\, C_2}{H \vdash (e_1, e_2) : ct_1 \times ct_2 \,|\, C_1, C_2}$$

$$\text{(FST)} \ \frac{H \vdash e : ct_1 \times ct_2 \,|\, C}{H \vdash \texttt{fst}\ e : ct_1 \,|\, C} \qquad \text{(SND)} \ \frac{H \vdash e : ct_1 \times ct_2 \,|\, C}{H \vdash \texttt{snd}\ e : ct_2 \,|\, C}$$

[1] Les algorithmes de résolution que nous allons présenter choisissent des horloges au plus tôt, donc la taille des buffers ne sera pas forcément nulle en entrée, et toujours nulle en sortie.

– Les entrées des primitives importées doivent toutes avoir le même type, qui est aussi le type du résultat :

$$\text{(OP)}\ \frac{H \vdash e_1 : ck \,|\, C_1 \quad H \vdash e_2 : ck \,|\, C_2}{H \vdash op(e_1, e_2) : ck \,|\, C_1, C_2}$$

– Le type d'une application de fonction suit la règle de typage classique des systèmes de type de ML :

$$\text{(APP)}\ \frac{ct_1 \to ct_2 \in inst(H(f)) \quad H \vdash e : ct_1 \,|\, C}{H \vdash f\,e : ct_2 \,|\, C}$$

– Le typage d'un ensemble de définitions locales de variables *eqs* produit un environnement de typage utilisé pour typer l'expression principale e :

$$\text{(WHERE)}\ \frac{H \vdash eqs : H' \,|\, C_1 \quad H + H' \vdash e : ct \,|\, C_2}{H \vdash e\ \texttt{where rec}\ eqs : ct \,|\, C_1, C_2}$$

– Les types des deux arguments d'un délai initialisé doivent être les mêmes, et ce type est aussi le type du résultat :

$$\text{(FBY)}\ \frac{H \vdash e_1 : ck \,|\, C_1 \quad H \vdash e_2 : ck \,|\, C_2}{H \vdash e_1\ \texttt{fby}\ e_2 : ck \,|\, C_1, C_2}$$

– Le type de l'échantillonnage d'un flot e de type ck par une horloge ce est noté ck `on` ce. De même, celui de l'échantillonnage d'un flot par la négation de ce est ck `on not` ce.

$$\text{(WHEN)}\ \frac{H \vdash e : ck \,|\, C \quad H \vdash ce : ck \,|\, \varnothing}{H \vdash e\ \texttt{when}\ ce : ck\ \texttt{on}\ ce \,|\, C} \qquad \text{(WHENOT)}\ \frac{H \vdash e : ck \,|\, C \quad H \vdash ce : ck \,|\, \varnothing}{H \vdash e\ \texttt{whenot}\ ce : ck\ \texttt{on not}\ ce \,|\, C}$$

– La fusion de deux flots e_1 et e_2 sur l'horloge ce est bien typée si e_1 est de type ck `on` ce et si e_2 est du type complémentaire ck `on not` ce. Le flot résultat est alors de type ck :

$$\text{(MERGE)}\ \frac{H \vdash ce : ck \,|\, \varnothing \quad H \vdash e_1 : ck\ \texttt{on}\ ce \,|\, C_1 \quad H \vdash e_2 : ck\ \texttt{on not}\ ce \,|\, C_2}{H \vdash \texttt{merge}\ ce\ e_1\ e_2 : ck \,|\, C_1, C_2}$$

– Le type d'une expression bufferisée de type ck_1 peut-être n'importe quel type ck_2 tel que $ck_1 <: ck_2$. Pour cela, on donne le type ck_2 à la sortie du buffer, et on ajoute la contrainte $ck_1 <: ck_2$ à l'ensemble de contraintes C :

$$\text{(BUF)}\ \frac{H \vdash e : ck_1 \,|\, C}{H \vdash \texttt{buffer}(e) : ck_2 \,|\, C, \{ck_1 <: ck_2\}}$$

– Le typage d'une équation produit l'association d'un type à une variable. Le typage d'un ensemble d'équations mutuellement récursives se fait de manière structurelle et produit un environnement contenant les types des variables définies par les équations :

$$\text{(EQ)}\ \frac{H + [x : ct] \vdash e : ct \,|\, C}{H \vdash x\ \texttt{=}\ e : [x : ct] \,|\, C} \qquad \text{(EQS)}\ \frac{H + H_2 \vdash eqs_1 : H_1 \,|\, C_1 \quad H + H_1 \vdash eqs_2 : H_2 \,|\, C_2}{H \vdash eqs_1\ \texttt{and}\ eqs_2 : H_1 + H_2 \,|\, C_1, C_2}$$

– Les déclarations de nœuds sont typées comme des définitions classiques de fonctions. Étant donné que les définitions de nœud n'apparaissent qu'au niveau le plus extérieur (et ne peuvent pas être imbriquées), on peut généraliser toutes les variables apparaissant dans le type (aucune variable n'est libre dans l'environnement). La particularité de cette règle est que la généralisation du type $ct_1 \to ct_2$ calcule un type résultat satisfaisant les contraintes C accumulées sur les variables de $ct_1 \to ct_2$:

$$\text{(NODE)}\ \frac{H + [x : ct_1] \vdash e : ct_2 \,|\, C}{H \vdash \texttt{let node}\ f\ x\ \texttt{=}\ e : [f : gen(ct_1 \to ct_2, C)]}$$

– Les déclarations d'horloges produisent l'association d'une expression d'horloges à une variable d'horloge :

$$\text{(CLOCK)}\ \frac{H \vdash ce}{H \vdash \texttt{let clock}\ c\ \texttt{=}\ ce : [c : ce]}$$

– Le typage d'un ensemble de définitions collecte les associations produites :

$$\text{(DEF)}\ \frac{H \vdash d_1 : H_1 \quad H + H_1 \vdash d_2 : H_2}{H \vdash d_1; d_2 : H_1 + H_2}$$

2.3.3 Inférence des types d'horloges

Dans la section précédente, nous avons présenté un ensemble de règles qui définissent la relation de typage. Tout comme dans le compilateur Lucid Synchrone [CP03], nous avons choisi de réaliser une inférence de types d'horloges. Inférer les types signifie calculer les types associés aux expressions et aux nœuds, sans demander au programmeur d'annoter son programme.

L'inférence de types nécessite de savoir satisfaire deux sortes de contraintes sur les types d'horloges : d'une part, des contraintes d'égalité, résolues par *unification*, d'autre part, des contraintes de sous-typage.

Unification

Les communications synchrones imposent des contraintes d'égalité entre les types d'horloges. Par exemple, supposons que l'on veuille inférer le type de l'expression $x + y$ dans un environnement où les types α_1 `on` c_1 et α_2 `on` c_2 ont déjà été attribués respectivement à x et y. La règle (OP) indique que les types des paramètres de + doivent être égaux. Elle impose donc l'égalité entre α_1 `on` c_1 et α_2 `on` c_2. Pour satisfaire cette contrainte, il faut trouver une instanciation pour les variables de type α_1 et α_2 telle qu'une fois la substitution appliquée, α_1 `on` c_1 soit égal à α_2 `on` c_2.

Deux types ck_1 et ck_2 sont égaux si et seulement les horloges de tout flot x de type ck_1 et de tout flot y de type ck_2 sont égales :

$$ck_1 = ck_2 \Leftrightarrow \forall x : ck_1,\ \forall y : ck_2,\ clock(x) = clock(y)$$
$$ct_1 \times ct_2 = ct_3 \times ct_4 \Leftrightarrow ct_1 = ct_3 \wedge ct_2 = ct_4$$

$$\text{(IM)}\quad H \vdash i : ck \,|\, \varnothing \qquad\qquad \text{(CE)}\ \frac{H \vdash ce}{H \vdash ce : ck \,|\, \varnothing}$$

$$\text{(VAR)}\quad H \vdash x : H(x) \,|\, \varnothing$$

$$\text{(PAIR)}\ \frac{H \vdash e_1 : ct_1 \,|\, C_1 \quad H \vdash e_2 : ct_2 \,|\, C_2}{H \vdash (e_1, e_2) : ct_1 \times ct_2 \,|\, C_1, C_2}$$

$$\text{(FST)}\ \frac{H \vdash e : ct_1 \times ct_2 \,|\, C}{H \vdash \texttt{fst}\ e : ct_1 \,|\, C} \qquad \text{(SND)}\ \frac{H \vdash e : ct_1 \times ct_2 \,|\, C}{H \vdash \texttt{snd}\ e : ct_2 \,|\, C}$$

$$\text{(OP)}\ \frac{H \vdash e_1 : ck \,|\, C_1 \quad H \vdash e_2 : ck \,|\, C_2}{H \vdash op(e_1, e_2) : ck \,|\, C_1, C_2}$$

$$\text{(APP)}\ \frac{ct_1 \to ct_2 \in inst(H(f)) \quad H \vdash e : ct_1 \,|\, C}{H \vdash f e : ct_2 \,|\, C}$$

$$\text{(WHERE)}\ \frac{H \vdash eqs : H' \,|\, C_1 \quad H + H' \vdash e : ct \,|\, C_2}{H \vdash e\ \texttt{where rec}\ eqs : ct \,|\, C_1, C_2}$$

$$\text{(FBY)}\ \frac{H \vdash e_1 : ck \,|\, C_1 \quad H \vdash e_2 : ck \,|\, C_2}{H \vdash e_1\ \texttt{fby}\ e_2 : ck \,|\, C_1, C_2}$$

$$\text{(WHEN)}\ \frac{H \vdash e : ck \,|\, C \quad H \vdash ce : ck \,|\, \varnothing}{H \vdash e\ \texttt{when}\ ce : ck\ \texttt{on}\ ce \,|\, C} \qquad \text{(WHENOT)}\ \frac{H \vdash e : ck \,|\, C \quad H \vdash ce : ck \,|\, \varnothing}{H \vdash e\ \texttt{whenot}\ ce : ck\ \texttt{on not}\ ce \,|\, C}$$

$$\text{(MERGE)}\ \frac{H \vdash ce : ck \,|\, \varnothing \quad H \vdash e_1 : ck\ \texttt{on}\ ce \,|\, C_1 \quad H \vdash e_2 : ck\ \texttt{on not}\ ce \,|\, C_2}{H \vdash \texttt{merge}\ ce\ e_1\ e_2 : ck \,|\, C_1, C_2}$$

$$\text{(BUF)}\ \frac{H \vdash e : ck_1 \,|\, C}{H \vdash \texttt{buffer}(e) : ck_2 \,|\, C, \{ck_1 <: ck_2\}}$$

$$\text{(EQ)}\ \frac{H + [x : ct] \vdash e : ct \,|\, C}{H \vdash x = e : [x : ct] \,|\, C} \qquad \text{(EQS)}\ \frac{H + H_2 \vdash eqs_1 : H_1 \,|\, C_1 \quad H + H_1 \vdash eqs_2 : H_2 \,|\, C_2}{H \vdash eqs_1\ \texttt{and}\ eqs_2 : H_1 + H_2 \,|\, C_1, C_2}$$

$$\text{(NODE)}\ \frac{H + [x : ct_1] \vdash e : ct_2 \,|\, C}{H \vdash \texttt{let node}\ f\ x = e : [f : gen(ct_1 \to ct_2, C)]} \qquad \text{(CLOCK)}\ \frac{H \vdash ce}{H \vdash \texttt{let clock}\ c = ce : [c : ce]}$$

$$\text{(DEF)}\ \frac{H \vdash d_1 : H_1 \quad H + H_1 \vdash d_2 : H_2}{H \vdash d_1; d_2 : H_1 + H_2}$$

FIG. 2.5 – Le calcul d'horloge.

Si l'on n'interprète pas les expressions d'horloges, on ne sait affirmer que deux types sont égaux que lorsqu'ils sont syntaxiquement égaux. Un test d'égalité plus puissant peut être fourni par le langage d'horloges.

Une substitution est une fonction θ qui associe des types de flots (ck) et des types d'ensembles de flots (ct) à des variables de type (resp. α et β). L'application d'une substitution à un type est définie par :

$$\begin{aligned} \theta(ck \texttt{ on } ce) &= \theta(ck) \texttt{ on } ce \\ \theta(ct_1 \times ct_2) &= \theta(ct_1) \times \theta(ct_2) \end{aligned}$$

L'unification de deux types d'horloges ct_1 et ct_2 consiste à trouver une substitution θ telle que $\theta(ct_1) = \theta(ct_2)$. La substitution θ est alors nommée *unificateur* de ct_1 et ct_2. Un unificateur θ est dit *unificateur principal* (ou *le plus général*) si tout autre unificateur θ' est tel qu'il existe une substitution θ'' telle que $\theta'' \circ \theta' = \theta$. Dans le cas où l'on sait calculer l'unificateur principal, les contraintes peuvent être résolues au fur et à mesure qu'elles sont rencontrées car l'unification n'implique aucune perte de généralité dans les types inférés. C'est le cas dans Lucid Synchrone qui, comme Lustre, unifie de manière structurelle syntaxique. Cette unification est définie ainsi :

$$\begin{aligned} \beta \equiv ct &\overset{def}{\Leftrightarrow} \beta \notin FV(ct) \\ ct_1 \times ct_2 \equiv ct'_1 \times ct'_2 &\overset{def}{\Leftrightarrow} ct_1 \equiv ct'_1 \wedge ct_2 \equiv ct'_2 \\ \alpha \equiv ck &\overset{def}{\Leftrightarrow} \alpha \notin FV(ck) \\ ck \texttt{ on } ce \equiv ck' \texttt{ on } ce' &\overset{def}{\Leftrightarrow} ck \equiv ck' \wedge ce = ce' \\ ck \texttt{ on not } ce \equiv ck' \texttt{ on not } ce' &\overset{def}{\Leftrightarrow} ck \equiv ck' \wedge ce = ce' \end{aligned}$$

On peut unifier une variable de type avec n'importe quel autre type moins général ne contenant pas cette variable. L'unificateur associe alors le type moins général à la variable. Pour unifier deux types produits, on unifie leur composantes correspondantes. Pour unifier deux types échantillonnés, on impose l'égalité des horloges d'échantillonnage, et on unifie les parties gauches du `on`. L'égalité des horloges d'échantillonnage est, dans Lucid Synchrone comme dans Lustre, l'égalité syntaxique. En effet, comme les flots booléens d'horloges peuvent être arbitrairement complexes, il serait difficile et coûteux d'utiliser une autre égalité.

Nous utiliserons pour l'instant une unification structurelle avec l'égalité des mots binaires ultimement périodiques. Nous verrons dans le chapitre 5 qu'il existe des fonctions d'unification plus puissantes sur ces horloges.

Finissons par l'exemple de l'inférence du type du nœud `sum_sum_odd`, dans un environnement H tel que $H(\texttt{sum\_odd}) = \forall \alpha.\, \alpha \to \alpha \texttt{ on (10)}$. L'opérateur `buffer` n'étant pas utilisé dans ce nœud, l'ensemble de contraintes C restera vide et par soucis de légèreté nous omettrons de l'écrire dans l'arbre. Toujours dans un soucis de légèreté, nous typons une version de `sum_sum_odd` n'utilisant pas de variables intermédiaires. Sont indiquées à droite des applications de règles les contraintes d'unifications nécessaires à leur application.

$$\dfrac{\dfrac{\alpha_1 \to \alpha_1 \texttt{ on (10)} \in inst(H(\texttt{sum\_odd})) \qquad \dfrac{\alpha_2 \to \alpha_2 \texttt{ on (10)} \in inst(H(\texttt{sum\_odd})) \qquad H, \texttt{x} : \beta_x \vdash \texttt{x} : \beta_x}{H, \texttt{x} : \beta_x \vdash \texttt{sum\_odd x} : \alpha_2 \texttt{ on (10)}}\ \alpha_2 \equiv \beta_x}{H, \texttt{x} : \beta_x \vdash \texttt{sum\_odd(sum\_odd x)} : (\alpha_2 \texttt{ on (10)}) \texttt{ on (10)}}\ \alpha_1 \equiv \alpha_2 \texttt{ on (10)}}{H \vdash \texttt{let node sum\_sum\_odd x = sum\_odd(sum\_odd x)} : [\texttt{sum\_sum\_odd} : \forall \alpha_2.\, \alpha_2 \to (\alpha_2 \texttt{ on (10)}) \texttt{ on (10)}]}$$

Contraintes de sous-typage

Les communications n-synchrones (présentes aux points d'application de l'opérateur `buffer`) imposent des contraintes de sous-typage entre les types d'horloges. Par exemple, supposons que l'on veuille inférer le type de l'expression `buffer`(x) dans un environnement où le type α_1 `on` c_1 a déjà été attribué à x. La règle (BUF) permet d'attribuer à `buffer`(x) n'importe quel type α_2 tel que α_1 `on` c_1 soit un sous-type de α_2. Pour satisfaire cette contrainte, il faut choisir une instanciation pour les variables de type α_1 et α_2 telle qu'une fois la substitution appliquée, α_1 `on` c_1 soit un sous-type de α_2.

La relation de sous-typage est définie de la manière suivante, en fonction de la relation d'adaptabilité sur les horloges (elle aussi notée $<:$), qui doit être fournie par le langage d'horloges :

$$ck_1 <: ck_2 \stackrel{def}{\Leftrightarrow} \forall x : ck_1, y : ck_2,\ clock(x) <: clock(y)$$

La résolution d'une contrainte de sous-typage entre un type ck_1 et un type ck_2 consiste à trouver une substitution θ telle que $\theta(ck_1) <: \theta(ck_2)$.

Contrairement au cas de l'unification, il n'y a pas de notion de substitution la plus générale pour résoudre une contrainte de sous-typage. Les contraintes doivent donc être collectées puis résolues de manière globale.

Considérons par exemple l'inférence du type du nœud `good` (les variables intermédiaires ont été supprimées pour limiter la taille de l'arbre). La résolution des contraintes d'unification étant faite au fur et à mesure, les substitutions calculées dans les sous-arbres gauches sont appliquées à l'environnement des sous arbres droits.

$$\dfrac{\dfrac{\dfrac{\dfrac{H, \mathtt{x} : \beta_x \vdash x : \beta_x \,|\quad \dfrac{H, \mathtt{x} : \beta_x \vdash \mathtt{(10)}}{H, \mathtt{x} : \beta_x \vdash \mathtt{(10)} : \alpha_x \,|}}{H, \mathtt{x} : \beta_x \vdash \mathtt{x\ when\ (10)} : \alpha_x\ \mathtt{on\ (10)} \,|}\ \beta_x \equiv \alpha_x}{H, \mathtt{x} : \beta_x \vdash \mathtt{buffer(x\ when\ (10))} : \alpha \,|\, \{\alpha_x\ \mathtt{on\ (10)} <: \alpha\}} \quad \dfrac{\dfrac{H, \mathtt{x} : \alpha_x \vdash \mathtt{x} : \alpha_x \,|\quad \dfrac{H, \mathtt{x} : \alpha_x \vdash \mathtt{(01)}}{H, \mathtt{x} : \alpha_x \vdash \mathtt{(01)} : \alpha'_x \,|}}{H, \mathtt{x} : \alpha_x \vdash \mathtt{x\ when\ (01)} : \alpha_x\ \mathtt{on\ (01)} \,|}\ \alpha_x \equiv \alpha'_x}{H, \mathtt{x} : \beta_x \vdash \mathtt{buffer(x\ when\ (10))\ +\ x\ when\ (01)} : \alpha_x\ \mathtt{on\ (01)} \,|\, \alpha_x\ \mathtt{on\ (10)} <: \alpha_x\ \mathtt{on\ (01)}}\ \alpha \equiv \alpha_x\ \mathtt{on\ (01)}$$

$$H \vdash \mathtt{let\ node\ good\ x\ =\ buffer(x\ when\ (10))\ +\ x\ when\ (01)} :$$
$$[\ \mathtt{good} : gen(\alpha_x \to \alpha_x\ \mathtt{on\ (01)}, \{\alpha_x\ \mathtt{on\ (10)} <: \alpha_x\ \mathtt{on\ (01)}\})\ ]$$

La généralisation du type inféré pour l'expression définissant le nœud nécessite la résolution de l'ensemble de contraintes $\{\alpha_x$ `on (10)` $<: \alpha_x$ `on (01)`$\}$. La seule manipulation syntaxique possible sur les types d'horloges est le test d'égalité structurelle qui est plus contraignant que la relation de sous-typage. Par conséquent, il faut interpréter la valeur des horloges pour résoudre ces contraintes. La fonction de résolution des contraintes de sous-typage doit donc être fournie avec le langage d'horloges.

2.3.4 Sémantique des programmes typés

Après la phase de typage, on annote chaque expression e par son type d'horloges ck (noté e^{ck}). Si une expression e est de type ck, alors l'horloge de son exécution à rythme w_{base} est $h(ck, w_{base})$ avec :

$$\begin{array}{lcl} h(\alpha, w_{base}) & = & w_{base} \\ h(ck \texttt{ on } w', w_{base}) & = & h(ck, w_{base}) \texttt{ on } w' \end{array}$$

On peut donc traduire les expressions typées du noyau vers le langage auquel on a donné une sémantique n-synchrone dans la section 2.2.2, de la manière suivante :

$$\begin{array}{rcl} i^{ck} & \rightsquigarrow & \texttt{const}(i, h(ck, w_{base})) \\ ce^{ck} & \rightsquigarrow & \texttt{cond}(ce, h(ck, w_{base})) \\ \texttt{buffer}(e^{ck_1})^{ck_2} & \rightsquigarrow & \texttt{buffer}(size(w_1, w_2), e', w_2) \\ & & \text{avec } w_1 = h(ck_1, w_{base}),\ w_2 = h(ck_2, w_{base}) \text{ et } e \rightsquigarrow e' \end{array}$$

La traduction des autres constructions du noyau est l'identité.

La fonction *size* donne la taille de buffer nécessaire pour consommer sur l'horloge w_2 un flot produit sur l'horloge w_1. Cette fonction doit être fournie avec le langage d'horloges.

2.4 Conclusion

Nous avons présenté dans ce chapitre un langage flot de données synchrone disposant d'un opérateur de bufferisation. Nous avons vu que le langage d'horloges utilisé doit fournir un test d'égalité des horloges, pour vérifier que les communications réalisées sans buffers sont synchrones. Il doit aussi disposer d'un test d'adaptabilité pour vérifier que les communications faites à travers un buffer sont n-synchrones, et d'un calcul d'une taille suffisante pour ce buffer. Enfin, comme nous avons choisi de réaliser une inférence de types plutôt qu'une vérification, le langage d'horloges doit fournir des algorithmes de résolution des contraintes d'égalité des types et des contraintes de sous-typage.

Pour atteindre ces objectifs, nous proposons deux méthodes. La première consiste à considérer un langage d'horloges périodiques, sur lesquelles on sait calculer les opérations *not* et *on*, vérifier les relations d'égalité et d'adaptabilité, ainsi que calculer la taille des buffers nécessaires. Ces travaux sont décrits dans la partie II. La seconde consiste à considérer un langage d'horloges où les horloges sont abstraites par des spécifications. On sait calculer une spécification des horloges composées d'opérations *not* et *on* et vérifier la relation d'adaptabilité à partir des spécifications, ainsi que calculer les tailles de buffers nécessaires. Ces travaux sont décrits dans la partie III.

Avant d'aborder ces parties, le chapitre 3 définit formellement les horloges en tant que mots binaires, ainsi que les opérations permettant de les composer et les comparer.

Chapitre 3

Expression des horloges par des mots binaires infinis

Nous avons vu dans le chapitre précédent que les instants de présence d'un flot peuvent être exprimés par un mot binaire infini, appelé horloge. L'occurrence d'un `1` dans ce mot dénote la présence d'une valeur sur le flot, et l'occurrence d'un `0` dénote l'absence de valeur. Nous présentons dans ce chapitre un ensemble d'opérations algébriques permettant de décrire les mots binaires infinis, de les transformer et de les comparer.

La section 3.1 définit des fonctions caractérisant les mots binaires. La section 3.2 présente l'opérateur *on*, permettant de calculer l'horloge d'un flot échantillonné, et l'opérateur *not*, permettant de calculer les horloges de flots destinés à être fusionnés. La section 3.3 montre comment ralentir deux horloges afin de les rendre égales, lorsque l'on veut établir une communication sans buffers. Enfin, la section 3.4 définit la relation d'adaptabilité, qui assure qu'une communication peut être faite de manière sûre à travers un buffer de taille bornée, et définit aussi le calcul de cette taille.

Chaque énoncé défini ou prouvé en Coq est suivi d'un lien ✿ pointant vers le code correspondant.

3.1 Mots binaires infinis, définitions et notations

Les mots binaires sont des séquences de `0` et de `1`. On notera u les mots binaires finis éventuellement vides, v les mots binaires finis non vides et w les mots binaires infinis :

$$\begin{array}{lcl} u & ::= & \varepsilon \mid \mathtt{0}.u \mid \mathtt{1}.u \\ v & ::= & \mathtt{0} \mid \mathtt{1} \mid \mathtt{0}.v \mid \mathtt{1}.v \\ w & ::= & \mathtt{0}.w \mid \mathtt{1}.w \end{array}$$

La concaténation d'un mot binaire fini u et d'un mot binaire fini ou infini t est notée $u.t$. On notera $\mathtt{0}^a$ la concaténation de a valeurs `0`, $\mathtt{1}^a$ la concaténation de a valeurs `1`, et plus généralement u^a la concaténation de a fois le mot fini u. Le mot infini $\mathtt{1}^{+\infty} = \mathtt{1111}\ldots$ est noté `(1)`, et le mot $\mathtt{0}^{+\infty} = \mathtt{0000}\ldots$ est noté `(0)`.

Les notations introduites dans la suite de cette section sont résumées dans la figure 3.1.

$\|u\|$	nombre d'éléments du mot fini u
$\|u\|_{\mathtt{1}}$	nombre de $\mathtt{1}$ dans le mot fini u (resp. $\|u\|_{\mathtt{0}}$, nombre de $\mathtt{0}$ dans u)
$w[i]$	élément à l'indice i de w
$w[i_1..i_2]$	sous-chaîne des éléments de w, de l'indice i_1 à l'indice i_2 compris
$\mathcal{I}_w(j)$	indice du $j^{ième}$ $\mathtt{1}$ de w
$\mathcal{O}_w(i)$	nombre de $\mathtt{1}$ dans w jusqu'à l'indice i (fonction de cumul de w)
$\mathcal{Z}_w(i)$	nombre de $\mathtt{0}$ dans w jusqu'à l'indice i

FIG. 3.1 – Tableau récapitulatif des notations sur les mots. La lettre $\mathcal{I}$ dans $\mathcal{I}_w(j)$ a été choisie comme une abréviation de *"Index of the* j^{th} *occurrence of* $\mathtt{1}$ *in* w*"*. La lettre $\mathcal{O}$ dans $\mathcal{O}_w(i)$ est une abréviation de *"number of Ones that occur in* w *until index* i*"*. Les indices commencent à 1.

La taille d'un mot binaire fini u est notée $|u|$, le nombre de $\mathtt{1}$ qu'il contient $|u|_{\mathtt{1}}$ et son nombre de $\mathtt{0}$ est noté $|u|_{\mathtt{0}}$.

$$\begin{array}{lllllllll} |\mathtt{1}.u| & \stackrel{def}{=} & 1+|u| & \quad |\mathtt{1}.u|_{\mathtt{1}} & \stackrel{def}{=} & 1+|u|_{\mathtt{1}} & \quad |\mathtt{1}.u|_{\mathtt{0}} & \stackrel{def}{=} & |u|_{\mathtt{0}} \\ |\mathtt{0}.u| & \stackrel{def}{=} & 1+|u| & \quad |\mathtt{0}.u|_{\mathtt{1}} & \stackrel{def}{=} & |u|_{\mathtt{1}} & \quad |\mathtt{0}.u|_{\mathtt{0}} & \stackrel{def}{=} & 1+|u|_{\mathtt{0}} \end{array}$$

Nous présentons ci-dessous trois fonctions permettant de décrire un mot binaire w :
- la fonction qui à i associe l'élément à l'indice i de w : $w[i]$;
- la fonction qui à j associe l'indice du $j^{ième}$ $\mathtt{1}$ de w : $\mathcal{I}_w(j)$;
- la fonction qui à i associe le nombre de $\mathtt{1}$ dans w jusqu'à l'indice i : $\mathcal{O}_w(i)$.

Ces trois fonctions définissent complètement un mot binaire :

$$\begin{array}{rcl} w_1 = w_2 & \Leftrightarrow & \forall i \geq 1,\ w_1[i] = w_2[i] \\ & \Leftrightarrow & \forall j \geq 1,\ \mathcal{I}_{w_1}(j) = \mathcal{I}_{w_2}(j) \\ & \Leftrightarrow & \forall i \geq 1,\ \mathcal{O}_{w_1}(i) = \mathcal{O}_{w_2}(i) \ \text{✿} \end{array}$$

Elles ont été introduites dans [Hal84]. La fonction d'accès au $i^{ième}$ élément de w est descriptive. La fonction $\mathcal{I}_w$ présente l'avantage d'étudier les mots par le biais de l'occurrence des $\mathtt{1}$, qui représentent les instants de production et consommation de valeurs sur les flots. La fonction $\mathcal{O}_w$ a été introduite car sa valeur en un point donne une information sur l'historique de w. C'est cette dernière formulation qui a été utilisée pour réaliser les preuves en Coq [BC04]. Nous utiliserons dans la suite du document celle des trois qui est la plus adaptée à la propriété que l'on veut prouver.

Définition 3.1 (éléments de w). Soit $w = b.w'$ avec $b \in \{\mathtt{0}, \mathtt{1}\}$.

$$\begin{array}{rcl} w[1] & \stackrel{def}{=} & b \\ \forall i > 1,\ w[i] & \stackrel{def}{=} & w'[i-1] \end{array}$$

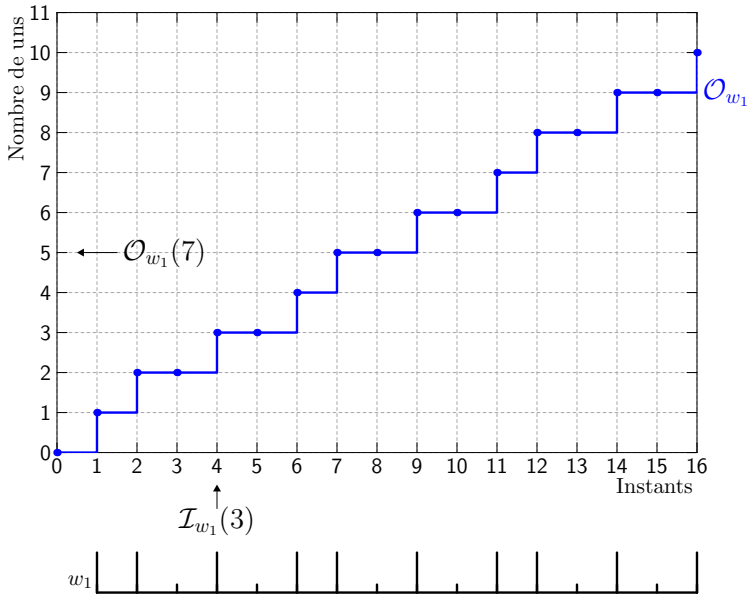


FIG. 3.2 – Graphique et chronogramme représentant le mot $w_1 = \texttt{11010 11010 110}\ldots$. Un front montant à l'instant i signifie que l'élément à l'indice i de w_1 est un `1`. Si la fonction ne change pas de valeur à l'instant i, alors l'élément à l'indice i de w_1 est un `0`. Le chronogramme situé sous le graphique représente le mot w_1, c'est une projection des fronts montants de $\mathcal{O}_{w_1}$.

Cette fonction peut être appliquée à un mot fini u si $i \leq |u|$. La sous-chaîne de w de l'indice i_1 à l'indice i_2 compris est notée $w[i_1..i_2]$. En particulier, comme les indices commencent à 1, $w[1..i]$ est le préfixe de w de longueur i.

Définition 3.2 (indice du $j^{ième}$ `1` de w : $\mathcal{I}_w(j)$).

$$\begin{array}{rcll} \mathcal{I}_w(1) & \stackrel{def}{=} & 1 & \text{si } w = \texttt{1}.w' \\ \forall j > 1,\ \mathcal{I}_w(j) & \stackrel{def}{=} & 1 + \mathcal{I}_{w'}(j-1) & \text{si } w = \texttt{1}.w' \\ \forall j > 0,\ \mathcal{I}_w(j) & \stackrel{def}{=} & d + \mathcal{I}_{w'}(j) & \text{si } w = \texttt{0}^d.w' \end{array}$$

Par exemple l'indice du $3^{ième}$ `1` de $w_1 = \texttt{11010 11010}\ldots$ est 4. Cette fonction peut être appliquée à un mot fini u si $j \leq |u|_1$.

Remarque 3.1. La fonction $\mathcal{I}_w$ est une fonction des mots binaires infinis vers les entiers complétés par $+\infty$. En effet, si w contient l'élément `1` moins de j fois, alors $\mathcal{I}_w(j) = +\infty$. La proposition $\forall j,\ \mathcal{I}_w(j) \neq +\infty$ signifie donc que w contient une infinité de `1`. La fonction $\mathcal{I}_w$ est croissante (strictement si le mot contient une infinité de `1`). ◇

Définition 3.3 (fonction de cumul de w : $\mathcal{O}_w$). ✿

$$\begin{array}{rcl} \mathcal{O}_w(0) & \stackrel{def}{=} & 0 \\ \forall i \geq 1,\ \mathcal{O}_w(i) & \stackrel{def}{=} & \left\{ \begin{array}{ll} \mathcal{O}_w(i-1) & \text{si } w[i] = \texttt{0} \\ \mathcal{O}_w(i-1) + 1 & \text{si } w[i] = \texttt{1} \end{array} \right. \end{array}$$

La fonction de cumul $\mathcal{O}_w$ associe à un indice i le nombre d'occurrences de l'élément `1` jusqu'à l'indice i de w. Elle est égale à $|w[1..i]|_1$. Cette fonction peut être appliquée à un mot fini u si $i \leq |u|$. La fonction $\mathcal{Z}_w(i) = |w[1..i]|_0$ donne le nombre de `0` apparaissant dans w jusqu'à l'indice i et peut être définie de manière similaire à $\mathcal{O}_w$. Les fonctions $\mathcal{O}_w$ et $\mathcal{Z}_w$ sont reliées ainsi : $\forall i \geq 1,\ \mathcal{Z}_w(i) = i - \mathcal{O}_w(i)$. ✿

La fonction de cumul de w_1 prend les valeurs suivantes à partir de l'instant 1 :

w_1	1	1	0	1	0	1	1	0	1	0	...
$\mathcal{O}_{w_1}$	1	2	2	3	3	4	5	5	6	6	...

Elle est représentée dans la figure 3.2.

Remarque 3.2. ✿ ✿ La fonction $\mathcal{O}_w$ est une fonction discrète qui ne varie que par fronts montants de hauteur 1 ($\forall i \geq 0,\ 0 \leq \mathcal{O}_w(i+1) - \mathcal{O}_w(i) \leq 1$). Les valeurs possibles pour cette fonction à l'instant i sont donc les valeurs comprises entre 0 et i ($\forall i \geq 0,\ 0 \leq \mathcal{O}_w(i) \leq i$). ◇

Remarque 3.3 (liens entre $\mathcal{O}_w(i)$ et $\mathcal{I}_w(j)$)**.** La fonction de cumul d'un mot et la fonction donnant les indices de ses `1` sont reliées de la manière suivante. L'indice du $j^{ième}$ `1` de w est le plus petit i tel que $\mathcal{O}_w(i) = j$, ou autrement dit l'indice du $j^{ième}$ front montant de $\mathcal{O}_w(i)$. Pour tout i, $\mathcal{O}_w(i) = 0$ si et seulement si i est strictement inférieur à l'indice du premier `1`, et $\mathcal{O}_w(i) = j$ si et seulement si i est compris entre l'indice du $j^{ième}$ `1` (inclus) et l'indice du $(j+1)^{ième}$ `1` (exclu).

$$\begin{array}{lll} \forall j \geq 1, & \mathcal{I}_w(j) = \min\{i \mid \mathcal{O}_w(i) = j\} & \\ \forall i \geq 0, & \mathcal{O}_w(i) = 0 & \Leftrightarrow \quad i < \mathcal{I}_w(1) \\ & \mathcal{O}_w(i) = j \text{ avec } j \geq 1 & \Leftrightarrow \quad \mathcal{I}_w(j) \leq i < \mathcal{I}_w(j+1) \end{array}$$

◇

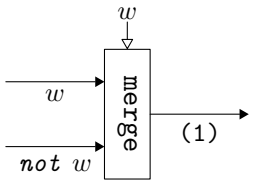
Définition 3.4 (taux de `1` dans w : $rate(w)$).
On appellera taux de w la proportion de `1` dans un mot binaire w :

$$rate(w) = \lim_{i \to +\infty} \frac{\mathcal{O}_w(i)}{i}$$

3.2 Opérateurs *not* et *on*

Nous allons maintenant définir deux opérateurs sur les mots binaires. Ceux-ci sont utiles pour exprimer les rythmes de production et consommation de nœuds qui ne préservent pas la longueur des flots.

L'opérateur *not* donne le rythme d'un flot à partir du rythme de son flot complémentaire. Il permet d'exprimer les horloges d'entrée de l'opérateur `merge` qui consomme deux flots de rythmes complémentaires et les fusionne pour produire un flot toujours présent :



Définition 3.5 (opérateur `not`).

$$\begin{aligned} \texttt{not}\ \texttt{1}.w &\stackrel{def}{=} \texttt{0}.(\texttt{not}\ w) \\ \texttt{not}\ \texttt{0}.w &\stackrel{def}{=} \texttt{1}.(\texttt{not}\ w) \end{aligned}$$

Par exemple :

w	1	1	0	1	0	0	1	1	0	...
`not` w	0	0	1	0	1	1	0	0	1	...

Remarque 3.4 (caractérisations de `not`)**.** Les éléments de `not` w sont la négation booléenne des éléments de w, et sa fonction de cumul peut être calculée à partir de celle de w.

- éléments : $\forall i \geq 1,\ (\texttt{not}\ w)[i] = \begin{cases} \texttt{0} & \text{si } w[i] = \texttt{1} \\ \texttt{1} & \text{si } w[i] = \texttt{0} \end{cases}$
- fonction de cumul : $\mathcal{O}_{\texttt{not}\ w}(0) = 0$ et $\forall i \geq 1,\ \mathcal{O}_{\texttt{not}\ w}(i) = \mathcal{Z}_w(i) = i - \mathcal{O}_w(i)$

◇

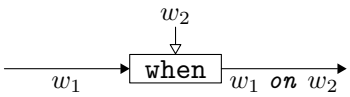
Remarque 3.5 (`not` et égalité)**.** L'opérateur `not` est une bijection.

$$\texttt{not}\ w_1 = \texttt{not}\ w_2 \Leftrightarrow w_1 = w_2$$

◇

Le test d'égalité `not` $w_1 =$ `not` w_2 peut donc être réalisé de manière correcte et complète sans calculer l'application du `not`, en testant simplement l'égalité $w_1 = w_2$.

L'opérateur `on` donne le rythme d'un flot échantillonné. Il permet d'exprimer l'horloge de sortie de l'opérateur `when` qui laisse passer certaines valeurs selon un rythme w_2 d'un flot consommé sur un rythme w_1 :



Définition 3.6 (opérateur `on`).

$$\begin{aligned} \texttt{0}.w_1\ \texttt{on}\ w_2 &\stackrel{def}{=} \texttt{0}.(w_1\ \texttt{on}\ w_2) \\ \texttt{1}.w_1\ \texttt{on}\ \texttt{1}.w_2 &\stackrel{def}{=} \texttt{1}.(w_1\ \texttt{on}\ w_2) \\ \texttt{1}.w_1\ \texttt{on}\ \texttt{0}.w_2 &\stackrel{def}{=} \texttt{0}.(w_1\ \texttt{on}\ w_2) \end{aligned}$$

Par exemple, si $w_1 = \texttt{01011101}\ldots$ et $w_2 = \texttt{10011}\ldots$, alors w_1 `on` $w_2 = \texttt{01000101}\ldots$.

On peut constater dans la définition de l'opérateur `on` que le mot `11111`... est neutre à gauche et à droite et que le mot `00000`... est absorbant à gauche et à droite : pour tout mot w, `(1)` `on` $w = w = w$ `on` `(1)` et `(0)` `on` $w =$ `(0)` $= w$ `on` `(0)`. En effet, échantillonner un flot sur la condition `(1)` ne supprime aucune valeur de celui-ci et laisse donc son l'horloge inchangée. À l'inverse, l'échantillonnage selon la condition `(0)` supprime toutes les valeurs du flot et produit donc un flot sans valeurs, d'horloge `(0)`.

On peut aussi remarquer que les `0` de w_1 sont reportés dans w_1 `on` w_2 et qu'un élément de w_2 est utilisé à chaque `1` dans w_1. Par conséquent, l'élément à l'indice i de w_1 `on` w_2 peut être obtenu sans effectuer le calcul du mot w_1 `on` w_2.

Proposition 3.1 (éléments de w_1 on w_2). ✿

$$\forall i \geq 1,\ (w_1 \texttt{ on } w_2)[i] = \begin{cases} \texttt{0} & \text{si } w_1[i] = \texttt{0} \\ w_2[\mathcal{O}_{w_1}(i)] & \text{si } w_1[i] = \texttt{1} \end{cases}$$

Démonstration :

Notons que si $w_1[i] = \texttt{1}$, $\mathcal{O}_{w_1}(i) \geq 1$ donc $w_2[\mathcal{O}_{w_1}(i)]$ est bien défini.
Preuve par induction sur i.

Cas de base : $i = 1$

Si $w_1 = \texttt{0}.w_1'$
$(w_1 \texttt{ on } w_2)[1] = (\texttt{0}.w_1' \texttt{ on } w_2)[1] = (\texttt{0}.(w_1' \texttt{ on } w_2))[1] = \texttt{0}$

Si $w_1 = \texttt{1}.w_1'$ **et** $w_2 = b.w_2'$
D'une part, $(w_1 \texttt{ on } w_2)[1] = (\texttt{1}.w_1' \texttt{ on } b.w_2')[1] = (b.(w_1' \texttt{ on } w_2'))[1] = b$.
D'autre part, $\mathcal{O}_{w_1}(1) = 1$ donc $w_2[\mathcal{O}_{w_1}(1)] = b$.

Induction :

Si $w_1 = \texttt{0}.w_1'$
On a $(w_1 \texttt{ on } w_2)[i+1] = (\texttt{0}.w_1' \texttt{ on } w_2)[i+1] = (\texttt{0}.(w_1' \texttt{ on } w_2))[i+1]$
$= (w_1' \texttt{ on } w_2)[i]$
Par hypothèse d'induction, $(w_1' \texttt{ on } w_2)[i] = \texttt{0}$ si $w_1'[i] = \texttt{0}$, i.e. si $w_1[i+1] = \texttt{0}$.
Toujours par hypothèse d'induction, $(w_1' \texttt{ on } w_2)[i] = w_2[\mathcal{O}_{w_1'}(i)]$ si $w_1'[i] = \texttt{1}$, c'est-à-dire si $w_1[i+1] = \texttt{1}$.

Si $w_1 = \texttt{1}.w_1'$ **et** $w_2 = b.w_2'$
D'une part, $(w_1 \texttt{ on } w_2)[i+1] = (\texttt{1}.w_1' \texttt{ on } b.w_2')[i+1] = (b.(w_1' \texttt{ on } w_2'))[i+1]$
$= (w_1' \texttt{ on } w_2')[i]$
Par hypothèse d'induction, $(w_1' \texttt{ on } w_2')[i] = \texttt{0}$ si $w_1'[i] = \texttt{0}$, i.e. si $w_1[i+1] = \texttt{0}$.
Toujours par hypothèse d'induction, $(w_1' \texttt{ on } w_2')[i] = w_2'[\mathcal{O}_{w_1'}(i)]$ si $w_1'[i] = \texttt{1}$, donc $(w_1' \texttt{ on } w_2')[i] = w_2'[\mathcal{O}_{w_1}(i+1) - 1] = w_2[\mathcal{O}_{w_1}(i+1)]$ si $w_1[i+1] = \texttt{1}$.

□

Reprenons l'exemple précédent :

w_1	0	1	0	1	1	1	0	1	...
w_2		1		0	0	1		1	...
w_1 on w_2	0	1	0	0	0	1	0	1	...

(3.1)

On peut vérifier qu'à chaque instant i on lit un élément de w_1.
- Si celui-ci vaut 0, alors l'élément à l'indice i dans w_1 on w_2 vaut 0 et aucun élément de w_2 n'est consommé.
- Si celui-ci vaut 1, alors on lit un élément de w_2 qui détermine l'élément à l'indice i dans w_1 on w_2. Un élément de w_2 est lu à chaque 1 dans w_1. À l'instant i, il y a eu $\mathcal{O}_{w_1}(i)$ occurrences de 1 dans w_1, donc on lit l'élément à l'indice $\mathcal{O}_{w_1}(i)$ de w_2.

Les éléments de w_1 on w_2 sont donc les éléments de w_1, dont les 1 ont été "filtrés" par les éléments de w_2.

Les éléments de w_1 `on` w_2 peuvent aussi être vus comme les éléments de w_2, quand w_2 est parcouru au rythme des `1` dans w_1. Il n'est donc pas nécessaire de calculer le mot w_1 `on` w_2 pour avoir sa fonction de cumul : celle-ci peut être obtenue par composition des fonctions de cumul de w_1 et w_2.

Proposition 3.2 (fonction de cumul de w_1 `on` w_2). ✿

$$\forall i \geq 0,\ \mathcal{O}_{w_1 \texttt{ on } w_2}(i) = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i))$$

Démonstration :

Par induction sur i.
Par définition de $\mathcal{O}$, $\mathcal{O}_{w_1 \texttt{ on } w_2}(0) = 0 = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(0))$.
Supposons que $\mathcal{O}_{w_1 \texttt{ on } w_2}(i) = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i))$, et montrons que $\mathcal{O}_{w_1 \texttt{ on } w_2}(i+1) = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i+1))$.

Par définition de $\mathcal{O}$, $\mathcal{O}_{w_1 \texttt{ on } w_2}(i+1) = \begin{cases} \mathcal{O}_{w_1 \texttt{ on } w_2}(i) & \text{si } (w_1 \texttt{ on } w_2)[i+1] = \texttt{0} \\ \mathcal{O}_{w_1 \texttt{ on } w_2}(i) + 1 & \text{si } (w_1 \texttt{ on } w_2)[i+1] = \texttt{1} \end{cases}$

Cas où $(w_1 \texttt{ on } w_2)[i+1] = \texttt{0}$ **:** $\mathcal{O}_{w_1 \texttt{ on } w_2}(i+1) = \mathcal{O}_{w_1 \texttt{ on } w_2}(i)$.
Par hypothèse d'induction, $\mathcal{O}_{w_1 \texttt{ on } w_2}(i) = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i))$.
Par la proposition 3.1, on a :

$$(w_1 \texttt{ on } w_2)[i+1] = \texttt{0} \ \Leftrightarrow\ w_1[i+1] = \texttt{0} \ \vee\ (w_1[i+1] = \texttt{1} \wedge w_2[\mathcal{O}_{w_1}(i+1)] = \texttt{0})$$

Si $w_1[i+1] = \texttt{0}$, on a $\mathcal{O}_{w_1}(i+1) = \mathcal{O}_{w_1}(i)$. Donc $\mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i+1)) = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i))$ et la propriété $\mathcal{O}_{w_1 \texttt{ on } w_2}(i+1) = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i+1))$ est prouvée.

Si $w_1[i+1] = \texttt{1} \wedge w_2[\mathcal{O}_{w_1}(i+1)] = \texttt{0}$, on a $\mathcal{O}_{w_1}(i+1) = \mathcal{O}_{w_1}(i) + 1$ et $\mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i+1)) = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i+1) - 1)$. Donc $\mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i+1)) = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i))$ et la propriété $\mathcal{O}_{w_1 \texttt{ on } w_2}(i+1) = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i+1))$ est prouvée.

Cas où $(w_1 \texttt{ on } w_2)[i+1] = \texttt{1}$ **:** $\mathcal{O}_{w_1 \texttt{ on } w_2}(i+1) = \mathcal{O}_{w_1 \texttt{ on } w_2}(i) + 1$.
Par hypothèse d'induction, $\mathcal{O}_{w_1 \texttt{ on } w_2}(i) = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i))$.
Par la proposition 3.1, on a :
$(w_1 \texttt{ on } w_2)[i+1] = \texttt{1} \ \Leftrightarrow\ (w_1[i+1] = \texttt{1} \ \wedge\ w_2[\mathcal{O}_{w_1}(i+1)] = \texttt{1})$.
Donc $\mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i+1)) = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i+1) - 1) + 1 = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i)) + 1$ et la propriété $\mathcal{O}_{w_1 \texttt{ on } w_2}(i+1) = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i+1))$ est prouvée.

□

Enfin, il n'est pas non plus nécessaire de calculer le mot w_1 `on` w_2 pour avoir l'indice de son $j^{ième}$ `1`. En effet, si le $j^{ième}$ `1` de w_2 est le $i^{ième}$ élément de w_2 ($\mathcal{I}_{w_2}(j) = i$) , alors le $j^{ième}$ `1` de w_1 `on` w_2 est à l'indice du $i^{ième}$ `1` de w_1 ($\mathcal{I}_{w_1 \texttt{ on } w_2}(j) = \mathcal{I}_{w_1}(i)$).

Proposition 3.3 (indice du $j^{ième}$ `1` de w_1 `on` w_2).

$$\forall j \geq 1,\ \mathcal{I}_{w_1 \texttt{ on } w_2}(j) = \mathcal{I}_{w_1}(\mathcal{I}_{w_2}(j))$$

Démonstration :

D'après la remarque 3.3, $\mathcal{I}_{w_1 \texttt{ on } w_2}(j)$ est le plus petit i tel que $\mathcal{O}_{w_1 \texttt{ on } w_2}(i) = j$. Par conséquent, d'après la proposition 3.2, $\mathcal{I}_{w_1 \texttt{ on } w_2}(j) = \min_{i\in\mathbb{N}^*}\{i \mid (\mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i)) = j)\}$. Comme $\mathcal{O}$ est croissante, on peut en déduire que $\mathcal{I}_{w_1 \texttt{ on } w_2}(j) = \min_{i\in\mathbb{N}^*}\{i \mid (\mathcal{O}_{w_1}(i) = \min_{i'\in\mathbb{N}^*}\{i' \mid (\mathcal{O}_{w_2}(i') = j)\})\}$. Par la remarque 3.3, on peut conclure que $\mathcal{I}_{w_1 \texttt{ on } w_2}(j) = \min\{i \mid (\mathcal{O}_{w_1}(i) = \mathcal{I}_{w_2}(j)) = \mathcal{I}_{w_1}(\mathcal{I}_{w_2}(j))\}$. Remarquons que la formule est valide pour le cas particulier où j est supérieur au nombre de `1` de w_1 `on` w_2. D'une part, si $\mathcal{I}_{w_2}(j) = +\infty$, alors $\mathcal{I}_{w_1 \texttt{ on } w_2}(j) = \mathcal{I}_{w_1}(+\infty) = +\infty$. En effet, si w_2 n'a pas de $j^{ième}$ `1`, alors w_1 `on` w_2 non plus. D'autre part, si $\mathcal{I}_{w_2}(j) = j'$ avec $\mathcal{I}_{w_1}(j') = +\infty$, alors $\mathcal{I}_{w_1 \texttt{ on } w_2}(j) = \mathcal{I}_{w_1}(j') = +\infty$. En effet, si le nombre total de `1` dans w_1 est inférieur à l'indice du $j^{ième}$ `1` dans w_2, alors w_1 `on` w_2 contient moins de j `1`. □

Dans l'exemple (3.1) page 54, le deuxième `1` de w_2 est le quatrième élément de w_2 donc le deuxième `1` de w_1 `on` w_2 est à l'indice du quatrième `1` de w_1, c'est-à-dire à l'indice 6.

On peut déduire à la fois de la proposition 3.2 et de la proposition 3.3 que l'opérateur `on` est associatif. Ce corollaire est important pour les calculs impliquant des inconnues (voir les chapitres 5 et 8).

Corollaire 3.1 (associativité de `on`). ✿ $(w_1 \texttt{ on } w_2) \texttt{ on } w_3 = w_1 \texttt{ on } (w_2 \texttt{ on } w_3)$

Démonstration :

Par la proposition 3.3,

$$\begin{aligned}\forall j \geq 1,\ \mathcal{I}_{(w_1 \texttt{ on } w_2) \texttt{ on } w_3}(j) &= \mathcal{I}_{w_1 \texttt{ on } w_2}(\mathcal{I}_{w_3}(j)) = \mathcal{I}_{w_1}(\mathcal{I}_{w_2}(\mathcal{I}_{w_3}(j)))\\ &= \mathcal{I}_{w_1}(\mathcal{I}_{w_2 \texttt{ on } w_3}(j)) = \mathcal{I}_{w_1 \texttt{ on } (w_2 \texttt{ on } w_3)}(j)\end{aligned}$$

□

En guise d'entraînement au calcul de l'opération `on`, vérifions cette propriété sur un exemple. Échantillonnons le mot $(w_1$ `on` $w_2)$ de l'exemple (3.1) par $w_3 =$ `101`$\ldots$:

w_1	0	1	0	1	1	1	0	1	...
w_2		1		0	0	1		1	...
w_1 `on` w_2	0	1	0	0	0	1	0	1	...
w_3		1				0		1	...
$(w_1$ `on` $w_2)$ `on` w_3	0	1	0	0	0	0	0	1	...

On peut constater qu'on obtient le même résultat en échantillonnant d'abord w_2 par w_3, puis en échantillonnant w_1 par $(w_2$ `on` $w_3)$:

w_1	0	1	0	1	1	1	0	1	...
w_2		1		0	0	1		1	...
w_3		1				0		1	...
w_2 `on` w_3		1		0	0	0		1	...
w_1 `on` $(w_2$ `on` $w_3)$	0	1	0	0	0	0	0	1	...

Remarque 3.6. L'opérateur `on` n'est pas commutatif.
Dans l'exemple (3.1), on a w_1 `on` $w_2 =$ `01000101`$\ldots$ alors que w_2 `on` $w_1 =$ `00010`$\ldots$. ◇

Trivialement, si $w_1 = w_2$, alors w_3 `on` $w_1 = w_3$ `on` w_2 et w_1 `on` $w_3 = w_2$ `on` w_3. Les réciproques de cette propriété ne sont pas immédiates. La première est vérifiée à condition que w_3 contienne une infinité de `1`, et la seconde n'est pas vérifiée :

Lemme 3.1 (`on` et égalité).
Si $\forall j,\ \mathcal{I}_{w_3}(j) \neq +\infty$, alors w_3 `on` $w_1 = w_3$ `on` $w_2 \Rightarrow w_1 = w_2$.
Par contre, w_1 `on` $w_3 = w_2$ `on` $w_3 \not\Rightarrow w_1 = w_2$

Démonstration :

Montrons que w_3 `on` $w_1 = w_3$ `on` $w_2 \Rightarrow w_1 = w_2$

$$\begin{aligned} w_3 \texttt{ on } w_1 = w_3 \texttt{ on } w_2 &\Leftrightarrow \forall j \geq 1,\ \mathcal{I}_{w_3 \texttt{ on } w_1}(j) = \mathcal{I}_{w_3 \texttt{ on } w_2}(j) \\ &\Leftrightarrow \forall j \geq 1,\ \mathcal{I}_{w_3}(\mathcal{I}_{w_1}(j)) = \mathcal{I}_{w_3}(\mathcal{I}_{w_2}(j)) \end{aligned}$$

Comme $\mathcal{I}_{w_3}$ est strictement croissante et $\forall i,\ \mathcal{I}_{w_3}(i) < +\infty$,

$$\begin{aligned} &\Leftrightarrow \forall j \geq 1,\ \mathcal{I}_{w_1}(j) = \mathcal{I}_{w_2}(j) \\ &\Leftrightarrow w_1 = w_2 \end{aligned}$$

Montrons que w_1 `on` $w_3 = w_2$ `on` $w_3 \not\Rightarrow w_1 = w_2$

Voici un contre exemple : $w_1 = \texttt{101 101 101 ...}$, $w_2 = \texttt{110 110 110 ...}$, $w_3 = \texttt{10 10 10 ...}$. L'échantillonnage par w_3 supprime les `1` de rang pair : w_1 `on` $w_3 = \texttt{100 100 100 ...} = w_2$ `on` w_3. Donc w_1 `on` $w_3 = w_2$ `on` w_3 mais $w_1 \neq w_2$ car leurs `1` de rang pair ne sont pas situés aux mêmes indices.

□

Le test d'égalité w_3 `on` $w_1 = w_3$ `on` w_2 peut donc être réalisé de manière correcte et complète sans calculer les opérations `on`, en testant l'égalité $w_1 = w_2$. Par contre, le test w_1 `on` $w_3 = w_2$ `on` w_3 ne peut être effectué de manière complète qu'en testant l'égalité des résultats des opérations `on`.

Montrons pour finir que l'opérateur `on` calcule en effet l'horloge des flots échantillonnés.

Proposition 3.4 (horloge de `when`).

$$\begin{aligned} \mathit{clock}(x \texttt{ when } w) &= \mathit{clock}(x) \texttt{ on } w \\ \mathit{clock}(x \texttt{ whenot } w) &= \mathit{clock}(x) \texttt{ on not } w \end{aligned}$$

Démonstration :

Pour montrer que $\mathit{clock}(x$ `when` $w) = \mathit{clock}(x)$ `on` w, on calcule l'horloge de l'interprétation de x `when` w, et on montre que celle-ci est égale à l'horloge de l'interprétation de x, à laquelle on applique l'opération `on` sur w, comme défini dans la définition 3.6.

Pour prouver cette égalité, on prouve que

$$\forall i \geq 1,\ (\mathit{clock}([\![x \texttt{ when } w]\!]^{\perp}))[i] = (\mathit{clock}(x) \texttt{ on } w)[i]$$

D'après la section 2.3.4, il faut traduire x `when` w en x `when cond`$(w, \mathit{clock}(x))$ pour se ramener au langage sur lequel on a défini la fonction d'interprétation.

La définition de *clock* se trouve page 37 et l'interprétation de x `when` w se trouve page 38.

L'interprétation de x `when` w est la suivante :

$$[\![x \texttt{ when cond}(w, \mathit{clock}(x))]\!]^{\perp} = \texttt{when}^\sharp(x, \texttt{cond}^\sharp(w, \mathit{clock}(x)))$$

Il faut donc montrer que

$$\forall i \geq 1,\ (\mathit{clock}(\texttt{when}^\sharp(x, \texttt{cond}^\sharp(w, \mathit{clock}(x)))))[i] = (\mathit{clock}(x)\ \texttt{on}\ w)[i]$$

On se permettra dans cette preuve d'utiliser la notation du $i^{ième}$ élément d'un mot pour désigner le $i^{ième}$ élément d'une séquence ($x[i]$ désignera le $i^{ième}$ élément de la séquence avec horloge x).

Par définition de $\texttt{when}^\sharp$:

$$\forall i \geq 1,\ \texttt{when}^\sharp(x, \texttt{cond}^\sharp(w, \mathit{clock}(x)))[i] = \begin{cases} \perp & \text{si } x[i] = \perp \text{ et } \texttt{cond}^\sharp(w, \mathit{clock}(x))[i] = \perp \\ v & \text{si } x[i] = v \text{ et } \texttt{cond}^\sharp(w, \mathit{clock}(x))[i] = \texttt{1} \\ \perp & \text{si } x[i] = v \text{ et } \texttt{cond}^\sharp(w, \mathit{clock}(x))[i] = \texttt{0} \end{cases}$$

On peut montrer par induction (voir la preuve en fin de démonstration) que :

$$\forall i \geq 1,\ \texttt{cond}^\sharp(w, \mathit{clock}(x))[i] = \begin{cases} \perp & \text{si } x[i] = \perp \\ w[\mathcal{O}_{\mathit{clock}(x)}(i)] & \text{si } x[i] = v \end{cases}$$

Donc on dispose du lemme intermédiaire suivant :

$$\forall i \geq 1,\ \texttt{when}^\sharp(x, \texttt{cond}^\sharp(w, \mathit{clock}(x)))[i] = \begin{cases} \perp & \text{si } x[i] = \perp \\ v & \text{si } x[i] = v \text{ et } w[\mathcal{O}_{\mathit{clock}(x)}(i)] = \texttt{1} \\ \perp & \text{si } x[i] = v \text{ et } w[\mathcal{O}_{\mathit{clock}(x)}(i)] = \texttt{0} \end{cases}$$

Cas où $x[i] = \perp$.
D'après le lemme intermédiaire, $\texttt{when}^\sharp(x, \texttt{cond}^\sharp(w, \mathit{clock}(x)))[i] = \perp$,
donc $\mathit{clock}(\texttt{when}^\sharp(x, \texttt{cond}^\sharp(w, \mathit{clock}(x))))[i] = \texttt{0}$.
D'autre part, $(\mathit{clock}(x))[i] = \texttt{0}$ donc on a $(\mathit{clock}(x)\ \texttt{on}\ w)[i] = \texttt{0}$.

Cas où $x[i] = v$ et $w[(\mathcal{O}_{\mathit{clock}(x)}(i))] = \texttt{1}$
D'après le lemme intermédiaire, $\texttt{when}^\sharp(x, \texttt{cond}^\sharp(w, \mathit{clock}(x)))[i] = v$,
donc $\mathit{clock}(\texttt{when}^\sharp(x, \texttt{cond}^\sharp(w, \mathit{clock}(x))))[i] = \texttt{1}$
D'autre part, $(\mathit{clock}(x))[i] = \texttt{1}$ et $w[(\mathcal{O}_{\mathit{clock}(x)}(i))] = \texttt{1}$ donc $(\mathit{clock}(x)\ \texttt{on}\ w)[i] = \texttt{1}$.

Cas où $x[i] = v$ et $\texttt{cond}^\sharp(w, \mathit{clock}(x))[i] = \texttt{0}$
D'après le lemme intermédiaire, $\texttt{when}^\sharp(x, \texttt{cond}^\sharp(w, \mathit{clock}(x)))[i] = \perp$,
donc $\mathit{clock}(\texttt{when}^\sharp(x, \texttt{cond}^\sharp(w, \mathit{clock}(x))))[i] = \texttt{0}$
D'autre part, $\mathit{clock}(x)[i] = \texttt{1}$ et $w[\mathcal{O}_{\mathit{clock}(x)}(i)] = \texttt{0}$ donc $(\mathit{clock}(x)\ \texttt{on}\ w)[i] = \texttt{0}$.

Dans tous les cas, l'égalité est vérifiée.
La preuve que $\mathit{clock}(x\ \texttt{whenot}\ w) = \mathit{clock}(x)\ \texttt{on not}\ w$ suit le même principe.

Il ne reste plus qu'à montrer que :

$$\forall i \geq 1,\ \texttt{cond}^\sharp(w, \mathit{clock}(x))[i] = \begin{cases} \perp & \text{si } x[i] = \perp \\ w[\mathcal{O}_{\mathit{clock}(x)}(i)] & \text{si } x[i] = v \end{cases}$$

Si $x[i] = \perp$ alors $\mathit{clock}(x)[i] = \texttt{0}$, et par définition de $\texttt{cond}^\sharp$, on a $\texttt{cond}^\sharp(w, \mathit{clock}(x))[i] = \perp$.
Montrons par induction que le cas $x[i] = v$ est correct.

Cas de base : i=1. Si $x[1] = v$ alors $clock(x)[1] = \mathtt{1}$.
D'une part, $\mathtt{cond}^\sharp(w, clock(x))[1] = \mathtt{cond}^\sharp(b.w', clock(\mathtt{1}.x'))[1]$
$= b.\mathtt{cond}^\sharp(w', clock(x'))[1] = b$
D'autre part, $w[\mathcal{O}_{clock(x)}(1)] = (b.w')[\mathcal{O}_{clock(v.x')}(1)]$
$= (b.w')[\mathcal{O}_{\mathtt{1}.clock(x')}(1)] = (b.w')[1] = b$
Donc l'égalité est vérifiée.
Induction. Si $x[i+1] = v$ alors $clock(x)[i+1] = \mathtt{1}$.
Si $x = \bot.x'$, alors
$\mathtt{cond}^\sharp(w, clock(x))[i+1] = \mathtt{cond}^\sharp(w, clock(\bot.x'))[i+1]$
$= \mathtt{cond}^\sharp(w, \mathtt{0}.clock(x'))[i+1]$
$= (\mathtt{0}.\mathtt{cond}^\sharp(w, clock(x')))[i+1]$
$= \mathtt{cond}^\sharp(w, clock(x'))[i]$
$= w[\mathcal{O}_{clock(x')}(i)]$ (par hypothèse d'induction)
Comme $\mathcal{O}_{clock(x)}(i+1) = \mathcal{O}_{clock(\bot.x')}(i+1) = \mathcal{O}_{\mathtt{0}.clock(x')}(i+1) = 0 + \mathcal{O}_{clock(x')}(i)$, l'égalité est vérifiée.
Si $x = v'.x'$ et $w = b.w'$, alors
$\mathtt{cond}^\sharp(w, clock(x))[i+1] = \mathtt{cond}^\sharp(b.w', clock(v'.x'))[i+1]$
$= \mathtt{cond}^\sharp(b.w', \mathtt{1}.clock(x'))[i+1]$
$= b.\mathtt{cond}^\sharp(w', clock(x'))[i+1]$
$= \mathtt{cond}^\sharp(w', clock(x'))[i]$
$= w'[\mathcal{O}_{clock(x')}(i)]$ (par hypothèse d'induction)
Comme $\mathcal{O}_{clock(x)}(i+1) = \mathcal{O}_{clock(v.x')}(i+1) = \mathcal{O}_{\mathtt{1}.clock(x')}(i+1) = 1 + \mathcal{O}_{clock(x')}(i)$, on a $w[\mathcal{O}_{clock(x)}(i+1)] = b.w'[1 + \mathcal{O}_{clock(x')}(i)] = w'[\mathcal{O}_{clock(x')}(i)]$.
Donc l'égalité est vérifiée.

□

Remarque 3.7. Comme w_1 *on* w_2 est l'horloge du flot échantillonné x `when` w_2 (avec w_1 l'horloge du flot x), on appelle souvent l'opération w_1 *on* w_2 *échantillonnage de* w_1 *par* w_2. Comme nous n'appliquerons jamais l'opérateur d'échantillonnage `when` à des mots binaires, il ne peut y avoir d'ambiguïté et nous nous permettrons donc d'utiliser ce vocabulaire, bien que l'opérateur *on* n'échantillonne pas les mots au sens de `when`. ◇

3.3 Communication sans buffers

Pour faire communiquer un nœud produisant un flot sur un rythme w_1 avec un nœud consommant un flot sur un rythme w_2 sans insérer de buffer, il faut ralentir le rythme d'activation de chacun des deux nœuds, afin que les valeurs soient produites à l'instant où elles sont consommées. Cette section définit la notion de ralentissement et le calcul du rythme commun.

3.3.1 Relation de ralentissement

Un mot w_1 est un *ralentissement* d'un mot w_2 s'il peut être obtenu en insérant des `0` dans w_2. Cette relation est un ordre partiel.

Définition 3.7 (ralentissement $\leqslant$).

$$w_1 \leqslant w_2 \overset{def}{\Leftrightarrow} \exists w,\ w_1 = w \ \textit{on}\ w_2$$

Comme le mot (1) est neutre pour l'opération *on*, on peut remarquer que tout mot w_1 est un ralentissement de (1) (en effet $w_1 = w_1$ *on* (1)). De même, comme le mot (0) est absorbant, il est un ralentissement de tout mot w_2 (en effet (0) = (0) *on* w_2).

Le mot $w_1 = 100\,100\,100\ldots$ est un ralentissement de $w_2 = 10101\,10101\ldots$ car on peut consulter les éléments de w_2 suffisamment lentement (en suivant un rythme w) pour que les instants d'occurrence de ses 1 coïncident avec les instants d'occurrence des 1 de w_1 :

w	1	1	0	1	1	0	1	0	0	1	...
w_2	1	0		1	0		1			1	...
w *on* $w_2 = w_1$	1	0	0	1	0	0	1	0	0	1	...

Proposition 3.5 (ralentisseurs).
L'ensemble des mots w tels que $w_1 = w$ *on* w_2 est noté $S(w_1, w_2) \stackrel{def}{=} \{w \mid w_1 = w \textit{ on } w_2\}$ et il est calculé ainsi :

$$\begin{array}{lcl} S(0^{d_1}.1.w_1, 0^{d_2}.1.w_2) & = & \{w \mid w = v.1.w' \text{ avec } |v| = d_1,\ |v|_1 = d_2 \text{ et } w' \in S(w_1, w_2)\} \\ S(0^{+\infty}, 0^{d_2}.1.w_2) & = & \{w \mid \forall i,\ \mathcal{O}_w(i) \leq d_2\} \\ S(0^{d_1}.1.w_1, 0^{+\infty}) & = & \varnothing \\ S(0^{+\infty}, 0^{+\infty}) & = & \{w\} \end{array}$$

avec $d_1 \in \mathbb{N}$, $d_2 \in \mathbb{N}$ et $0^{+\infty} = (0)$.

Démonstration :

Commençons par énoncer deux propriétés utiles à la preuve :

1. D'après la proposition 3.1 sur les éléments de w *on* w_2, on a l'équivalence suivante : $(w \textit{ on } w_2)[i] = 1 \Leftrightarrow w[i] = 1$ et $w_2[\mathcal{O}_w(i)] = 1$ ✿ ✿.
2. On peut en déduire le corollaire suivant : comme $\mathcal{O}_w(\mathcal{I}_w(i)) = i$ et $w[\mathcal{I}_w(i)] = 1$, si $w_2[i] = 1$, alors $(w \textit{ on } w_2)[\mathcal{I}_w(i)] = 1$.

Dans le cas où $w_2 = 0^{+\infty}$, comme $\forall w$, (0) = w *on* (0), tous les mots conviennent si $w_1 = 0^{+\infty}$, et aucun mot ne convient sinon.

Dans la suite de la preuve, on considère que w_2 contient au moins un 1, c'est-à-dire est de la forme $0^{d_2}.1.w_2'$. La preuve est faite par induction sur le nombre de 1 dans w_1.

Cas de base: $w_1 = 0^{+\infty}$ et $w_2 = 0^{d_2}.1.w_2'$
Dans ce cas, $S(w_1, w_2) = \{w \mid \forall i,\ \mathcal{O}_w(i) \leq d_2\}$.

Soit w tel que $0^{+\infty} = w$ *on* w_2. Montrons que $w \in S(0^{+\infty}, w_2)$.
Supposons que $\exists i,\ \mathcal{O}_w(i) > d_2$ et soit $i' = \mathcal{I}_w(d_2 + 1)$ (on a alors $i' < +\infty$). Alors comme $w[i'] = 1$, w *on* $w_2[i'] = w_2[\mathcal{O}_w(i')] = w_2[d_2 + 1] = 1$. Donc w *on* $w_2[i'] \neq 0^{+\infty}[i']$, ce qui est absurde. Donc $\forall i,\ \mathcal{O}_w(i) \leq d_2$, et on peut conclure que $w \in S(0^{+\infty}, w_2)$.

Soit $w \in S(0^{+\infty}, w_2)$. Montrons que $0^{+\infty} = w$ *on* w_2.
Montrons que $\forall i,\ w$ *on* $w_2[i] = 0^{\infty}[i] = 0$.
Si $w[i] = 0$ alors w *on* $w_2[i] = 0$.
Si $w[i] = 1$, alors w *on* $w_2[i] = w_2[\mathcal{O}_w(i)]$. Comme $w \in S(0^{+\infty}, w_2)$, $\mathcal{O}_w(i) \leq d_2$, et comme $w_2 = 0^{d_2}.1.w_2'$, on peut en conclure que $w_2[\mathcal{O}_w(i)] = 0$.
Par conséquent, w *on* $w_2 = 0^{+\infty}$.

Induction: $w_1 = \mathtt{0}^{d_1}.\mathtt{1}.w'_1$ **et** $w_2 = \mathtt{0}^{d_2}.\mathtt{1}.w'_2$

Soit w **tel que** $w_1 = w \ \mathtt{on}\ w_2$**. Montrons que** $w \in S(w_1, w_2)$**.**

On a $w_1[d_1 + 1] = \mathtt{1}$, donc $(w \ \mathtt{on}\ w_2)[d_1 + 1] = \mathtt{1}$.

Par la propriété 1 ci-dessus, $(w \ \mathtt{on}\ w_2)[d_1+1] = \mathtt{1}$ si et seulement si $w[d_1+1] = \mathtt{1}$ et $w_2[\mathcal{O}_w(d_1 + 1)] = \mathtt{1}$. Donc $w = v.\mathtt{1}.w'$ avec $|v| = d_1$ et $w_2[|v|_\mathtt{1} + 1] = \mathtt{1}$.

Montrons que $\forall i < |v|_\mathtt{1} + 1,\ w_2[i] = \mathtt{0}$.

Supposons que $\exists i < |v|_\mathtt{1} + 1$ tel que $w_2[i] = \mathtt{1}$. Alors par la propriété 2 ci-dessus, $w_1[\mathcal{I}_w(i)] = \mathtt{1}$. Or, comme $i \leq |v|_\mathtt{1}$, on sait que $\mathcal{I}_w(i) \leq |v| = d_1$. On peut en conclure que $\exists i' \leq d_1,\ w_1[i'] = \mathtt{1}$, ce qui est absurde car $w_1 = \mathtt{0}^{d_1}.\mathtt{1}.w'_1$.

Donc $\forall i < |v|_\mathtt{1} + 1, w_2[i] = \mathtt{0}$, et par conséquent $w_2[|v|_\mathtt{1} + 1] = \mathtt{1}$ implique que $|v|_\mathtt{1} + 1 = \mathcal{I}_{w_2}(1) = d_2 + 1$ car $w_2 = \mathtt{0}^{d_2}.\mathtt{1}.w'_2$.

On peut en conclure que $|v|_\mathtt{1} = d_2$.

On a montré que $w = v.\mathtt{1}.w'$ avec $|v| = d_1$ et $|v|_\mathtt{1} = d_2$.

On a donc $w \ \mathtt{on}\ w_2 = v.\mathtt{1}.w' \ \mathtt{on}\ \mathtt{0}^{d_2}.\mathtt{1}.w'_2 = \mathtt{0}^{d_1}.\mathtt{1}.(w' \ \mathtt{on}\ w'_2)$.

Par conséquent, $w_1 = w \ \mathtt{on}\ w_2 \Leftrightarrow \mathtt{0}^{d_1}.\mathtt{1}.w'_1 = \mathtt{0}^{d_1}.\mathtt{1}.(w' \ \mathtt{on}\ w'_2) \Leftrightarrow w'_1 = w' \ \mathtt{on}\ w'_2$, c'est-à-dire, par hypothèse d'induction, $w' \in S(w_1, w_2)$.

On peut maintenant conclure que $w = v.\mathtt{1}.w'$ avec $|v| = d_1$, $|v|_\mathtt{1} = d_2$ et $w' \in S(w_1, w_2)$, c'est-à-dire que $w \in S(w_1, w_2)$.

Soit $w \in S(w_1, w_2)$**. Montrons que** $w_1 = w \ \mathtt{on}\ w_2$**.**

On a $w = v.\mathtt{1}.w'$ avec $|v| = d_1$, $|v|_\mathtt{1} = d_2$ et $w' \in S(w'_1, w'_2)$.

Donc $w \ \mathtt{on}\ w_2 = v.\mathtt{1}.w' \ \mathtt{on}\ \mathtt{0}^{d_2}.\mathtt{1}.w'_2 = \mathtt{0}^{d_1}.\mathtt{1}.(w' \ \mathtt{on}\ w'_2)$.

Comme $w' \in S(w'_1, w'_2)$, par hypothèse d'induction on a $w' \ \mathtt{on}\ w'_2 = w'_1$. Par conséquent, $w \ \mathtt{on}\ w_2 = \mathtt{0}^{d_1}.\mathtt{1}.w'_1 = w_1$.

□

En présence d'un $\mathtt{1}$ en tête de w_1 et w_2, il faut reporter celui de w_2 dans $w \ \mathtt{on}\ w_2$ en plaçant un $\mathtt{1}$ en tête de w. En présence d'une suite $\mathtt{0}^{d_1}$ en tête de w_1 et $\mathtt{0}^{d_2}$ en tête de w_2 avec $d_1 \geq d_2$, il faut reporter ceux de w_2 en plaçant d_2 $\mathtt{1}$ dans w et rajouter les $\mathtt{0}$ manquants en plaçant $(d_1 - d_2)$ $\mathtt{0}$ dans w. N'importe quel entrelacement de ces éléments est valide, donc tout mot v de taille d_1 comportant d_2 $\mathtt{1}$ convient. Dans l'exemple précédent, il existe une infinité d'autres w possibles, comme $\mathtt{1011011}\ldots$ et $\mathtt{1101011}\ldots$.

Si l'on essaye d'appliquer la règle quand il y a moins de $\mathtt{0}$ en tête de w_1 qu'en tête de w_2 ($d_1 < d_2$), on obtient des contraintes contradictoires pour v : le nombre de $\mathtt{1}$ dans v doit alors être supérieur à sa taille. Dans ce cas, il n'existe pas de w tel que $w_1 = w \ \mathtt{on}\ w_2$ et par conséquent, w_1 n'est pas un ralentissement de w_2. Par exemple, les mots $w_1 = \mathtt{10\,10\,10}\ldots$ et $w_2 = \mathtt{1100\,1100}\ldots$ ne sont pas reliés par la relation de ralentissement.

D'après la définition 3.7 et la proposition 3.5, w_1 est un ralentissement de w_2 si et seulement si $S(w_1, w_2) \neq \varnothing$. On peut en déduire que w_1 est un ralentissement de w_2 si et seulement si ses séries de $\mathtt{0}$ sont toujours plus longues que les séries de $\mathtt{0}$ correspondantes dans w_2 :

Lemme 3.2 (ralentissement).

$$\mathtt{0}^{d_1}.\mathtt{1}.w_1 \leqslant \mathtt{0}^{d_2}.\mathtt{1}.w_2 \Leftrightarrow d_1 \geq d_2 \ \wedge \ w_1 \leqslant w_2$$

avec $d_1 \in \mathbb{N},\ d_2 \in \mathbb{N}$

Démonstration :

$$\begin{array}{lcl} \mathtt{0}^{d_1}.\mathtt{1}.w_1 \leqslant \mathtt{0}^{d_2}.\mathtt{1}.w_2 & \Leftrightarrow & S(\mathtt{0}^{d_1}.\mathtt{1}.w_1, \mathtt{0}^{d_2}.\mathtt{1}.w_2) \neq \varnothing \\ & \Leftrightarrow & d_1 \geq d_2 \ \wedge S(w_1, w_2) \neq \varnothing \\ & \Leftrightarrow & d_1 \geq d_2 \ \wedge w_1 \leqslant w_2 \end{array}$$

□

3.3.2 Calcul du rythme commun

Le mot le plus rapide[1] qui peut être obtenu à la fois par ralentissement de w_1 et par ralentissement de w_2 est l'infimum de ces deux mots pour la relation $\leqslant$. Il est calculé de la manière suivante.

Proposition 3.6 (infimum pour la relation $\leqslant$).

$$\begin{array}{lcl} \inf(\mathtt{0}^{d_1}.\mathtt{1}.w_1, \mathtt{0}^{d_2}.\mathtt{1}.w_2) & = & \mathtt{0}^{\max(d_1,d_2)}.\mathtt{1}.\inf(w_1, w_2) \\ \inf(\mathtt{0}^{+\infty}, w_2) & = & \mathtt{0}^{\infty} \\ \inf(w_1, \mathtt{0}^{+\infty}) & = & \mathtt{0}^{\infty} \end{array}$$

avec $d_1 \in \mathbb{N},\ d_2 \in \mathbb{N}$.

Démonstration :

La preuve se fait par induction sur le nombre de $\mathtt{1}$ dans w_1 et w_2.

Cas de base : $w_1 = \mathtt{0}^{+\infty}$ ou $w_2 = \mathtt{0}^{+\infty}$

Soit $m = \inf(\mathtt{0}^{+\infty}, w_2) = \mathtt{0}^{\infty}$.
Pour tout w, $(\mathtt{0}) \leqslant w$ donc $m \leqslant \mathtt{0}^{+\infty}$ et $m \leqslant w_2$.
Pour tout m' tel que $m' \leqslant \mathtt{0}^{+\infty}$ et $m \leqslant w_2$, $m' = \mathtt{0}^{+\infty}$ et donc $m' \leqslant m$.
m est donc le plus grand des minorants de w_1 et w_2 c'est-à-dire leur infimum.
Le cas où $w_2 = \mathtt{0}^{+\infty}$ est prouvé de la même manière.

Cas où $w_1 = \mathtt{0}^{d_1}.\mathtt{1}.w'_1$ et $w_2 = \mathtt{0}^{d_2}.\mathtt{1}.w'_2$

Soit $m = \inf(\mathtt{0}^{d_1}.\mathtt{1}.w'_1, \mathtt{0}^{d_2}.\mathtt{1}.w'_2) = \mathtt{0}^{\max(d_1,d_2)}.\mathtt{1}.\inf(w'_1, w'_2)$ comme défini dans la proposition.

Montrons que $m \leqslant w_1$ et $m \leqslant w_2$:

D'après le lemme 3.2, comme $\max(d_1, d_2) \geq d_1$ et $\inf(w'_1, w'_2) \leqslant w'_1$ (par hypothèse d'induction), $\mathtt{0}^{\max(d_1,d_2)}.\mathtt{1}.\inf(w'_1, w'_2) \leqslant \mathtt{0}^{d_1}.\mathtt{1}.w'_1$, c'est-à-dire $m \leqslant w_1$.
De la même manière, $m \leqslant w_2$.

Montrons que $\forall m',\ (m' \leqslant w_1 \wedge m' \leqslant w_2) \Rightarrow m' \leqslant m$:

Soit $m' = \mathtt{0}^{d}.\mathtt{1}.m''$. D'une part, $m' \leqslant \mathtt{0}^{d_1}.\mathtt{1}.w'_1 \Leftrightarrow d \geq d_1 \wedge m'' \leqslant w'_1$.
D'autre part, $m' \leqslant \mathtt{0}^{d_2}.\mathtt{1}.w'_2 \Rightarrow d \geq d_2 \wedge m'' \leqslant w'_2$.
Par conséquent, $d \geq \max(d_1, d_2) \wedge (m'' \leqslant w'_1 \wedge m'' \leqslant w'_2)$. Par hypothèse d'induction, $m'' \leqslant \inf(w'_1, w'_2)$. On peut donc en déduire par le lemme 3.2 que $m' \leqslant m$.

Par conséquent, m est bien le plus grand des minorants de $\mathtt{0}^{d_1}.\mathtt{1}.w_1$ et $\mathtt{0}^{d_2}.\mathtt{1}.w_2$, c'est-à-dire leur infimum.

□

[1] c'est-à-dire le plus grand par la relation $\leqslant$.

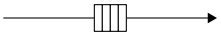
L'infimum doit pouvoir être obtenu à la fois par insertion de `0` dans w_1 et par insertion de `0` dans w_2. Par conséquent, chaque paquet de `0` de $\inf(w_1, w_2)$ doit être au moins aussi grand que les paquets correspondants dans w_1 et w_2. Comme on veut que l'infimum soit le plus rapide possible, on choisit la taille minimale satisfaisante, c'est-à-dire le maximum des tailles des paquets correspondants dans w_1 et w_2.

Remarque 3.8. Le supremum peut être calculé de manière similaire en conservant le minimum du nombre de `0` en tête des mots. L'ensemble des mots binaires muni de la relation de ralentissement forme donc un treillis. ◇

On peut faire communiquer un nœud qui produit sur le rythme w_1 avec un nœud qui consomme sur le rythme w_2 sur leur rythme commun le plus rapide qui est $m = \inf(w_1, w_2)$. Pour cela, il faut ralentir le rythme d'activation des nœuds afin que la production et la consommation soient faites sur le rythme m. Le ralentissement s_1 appliqué au premier nœud doit appartenir à $S(m, w_1)$, et le ralentissement s_2 appliqué au second doit appartenir à $S(m, w_2)$. La communication sera ainsi bien faite sur le rythme s_1 **on** $w_1 = m = s_2$ **on** w_2.

3.4 Communication avec buffers

Dans cette section, nous définissons la relation d'*adaptabilité* entre deux mots w_1 et w_2, notée $w_1 <: w_2$. Cette relation assure qu'un flot produit sur le rythme w_1 peut être consommé sur le rythme w_2 par insertion d'un buffer borné.



Pour cela, il faut d'une part qu'il n'y ait pas de lecture dans un buffer vide, c'est-à-dire que la $j^{ième}$ écriture dans le buffer ait toujours lieu avant la $j^{ième}$ lecture. Cette propriété est exprimée par la relation de *précédence* entre les mots w_1 et w_2, notée $w_1 \preceq w_2$ et décrite dans la section 3.4.1. D'autre part, il faut que le nombre de valeurs présentes dans le buffer au cours de l'exécution soit borné. Cette propriété est exprimée par la relation de *synchronisabilité* entre les mots w_1 et w_2, notée $w_1 \bowtie w_2$ et décrite dans la section 3.4.2.

La relation d'*adaptabilité* est la conjonction des relations de précédence et de synchronisabilité. Elle est décrite dans la section 3.4.3.

3.4.1 Relation de précédence

Un mot w_1 précède un mot w_2 si l'occurrence de son $j^{ième}$ `1` a toujours lieu avant ou en même temps que celle du $j^{ième}$ `1` de w_2. Cette relation est un ordre partiel ✿.

Définition 3.8 (précédence $\preceq$).

$$w_1 \preceq w_2 \overset{def}{\Leftrightarrow} \forall j \geq 1,\ \mathcal{I}_{w_1}(j) \leq \mathcal{I}_{w_2}(j)$$

Dans la figure 3.3, le $j^{ième}$ front montant de la courbe représentant w_1 arrive toujours avant le $j^{ième}$ front montant de la courbe représentant w_2. Si cette propriété continue à être vérifiée après les derniers fronts représentés sur le graphique alors $w_1 \preceq w_2$. Par contre, dans la figure 3.4 les courbes de w_3 et w_4 sont entrelacées, donc $w_3 \not\preceq w_4$ et $w_4 \not\preceq w_3$.

Comme les fonctions de cumul sont croissantes, w_1 précède w_2 si et seulement si la fonction de cumul de w_1 est toujours au dessus de ou égale à la fonction de cumul de w_2.

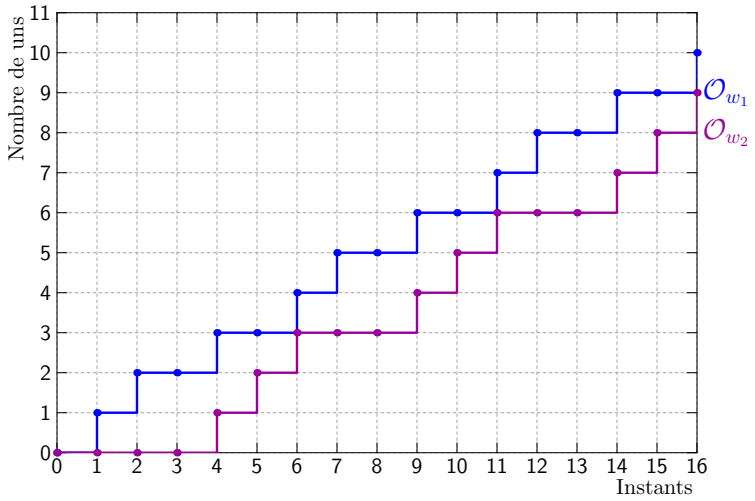


FIG. 3.3 – Fonctions de cumul des mots $w_1 =$11010 11010 110... et $w_2 =$0 00111 00111 001....

Proposition 3.7 (précédence et fonction de cumul). ✿

$$w_1 \preceq w_2 \Leftrightarrow \forall i > 0,\ \mathcal{O}_{w_1}(i) \geq \mathcal{O}_{w_2}(i)$$

Démonstration :

Montrons que $\forall j \geq 1,\ \mathcal{I}_{w_1}(j) \leq \mathcal{I}_{w_2}(j) \Leftrightarrow \forall i \geq 1,\ \mathcal{O}_{w_1}(i) \geq \mathcal{O}_{w_2}(i)$.
Par la remarque 3.3, il suffit de montrer que : $\forall j \geq 1$,
$(\min_{i\in\mathbb{N}^*}\{i \mid \mathcal{O}_{w_1}(i) = j\}) \leq (\min_{i\in\mathbb{N}^*}\{i \mid \mathcal{O}_{w_2}(i) = j\}) \Leftrightarrow \forall i \geq 1,\ \mathcal{O}_{w_1}(i) \geq \mathcal{O}_{w_2}(i)$.
$\Rightarrow$) vérifié car $\mathcal{O}$ est croissante.
$\Leftarrow$) vérifié car $\mathcal{O}_w(i) > j \Rightarrow \exists i' < i,\ \mathcal{O}_w(i') = j$. □

Ainsi, dans la figure 3.3, pour tout i, $\mathcal{O}_{w_1}(i) \geq \mathcal{O}_{w_2}(i)$, mais les courbes des mots de la figure 3.4 qui ne sont pas reliés par la relation de précédence, ne vérifient pas cette condition.

Le délai nécessaire pour qu'un flot produit sur le rythme w_1 puisse être consommé sur le rythme w_2 est le plus petit entier d tel que w_1 précède $0^d.w_2$:

$$\mathit{compute\_delay}(w_1, w_2) = \min\left\{d \in \mathbb{N} \mid w_1 \preceq 0^d.w_2\right\}$$

Ainsi, si l'on veut connecter à travers un buffer un nœud produisant sur le rythme w_1 avec un nœud consommant sur le rythme w_2, il faudra commencer l'activation du deuxième nœud d instants après l'activation du premier pour ne jamais lire dans un buffer vide. Le délai nécessaire est égal à la plus grande différence entre l'indice du $j^{ième}$ 1 de w_1 et l'indice du $j^{ième}$ 1 de w_2 :

$$\mathit{compute\_delay}(w_1, w_2) = \max_{j\in\mathbb{N}^*}\{\mathcal{I}_{w_1}(j) - \mathcal{I}_{w_2}(j)\}$$

Par exemple, dans la figure 3.4, $\mathit{compute\_delay}(w_3, w_4) = 1$ car il faut décaler la fonction de cumul de w_4 d'un instant vers la droite pour qu'elle se situe après celle de w_3, et de même,

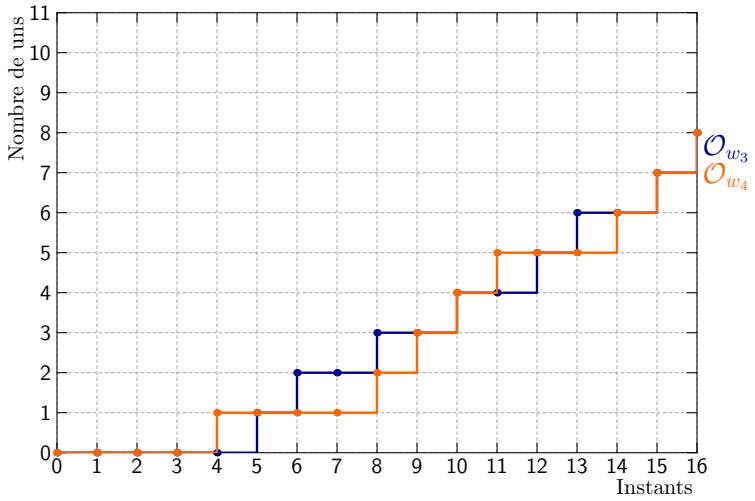
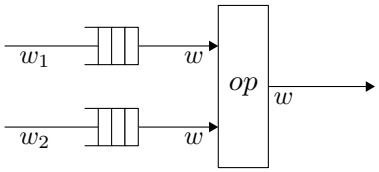


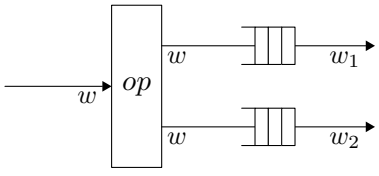
FIG. 3.4 – Fonctions de cumul des mots w_3 =00001101010110... et w_4 =00010001111001....

$compute\_delay(w_4, w_3) = 2$ car il faut décaler la courbe de w_3 de deux instants vers la droite pour que la relation de précédence soit vérifiée.

Le supremum de deux mots w_1 et w_2, noté $w_1 \sqcup w_2$ est le plus petit mot w au sens de la relation $\preceq$, tel que $w_1 \preceq w$ et $w_2 \preceq w$. Ainsi, si l'on veut lire simultanément dans deux buffers, dont l'un est rempli au rythme w_1 et l'autre au rythme w_2, le rythme le plus précoce possible pour ne jamais lire dans un buffer vide est le supremum de w_1 et w_2 :



L'infimum de deux mots w_1 et w_2, noté $w_1 \sqcap w_2$, est le plus grand mot w au sens de $\preceq$, tel que $w \preceq w_1$ et $w \preceq w_2$. Ainsi, si l'on veut remplir simultanément deux buffers, dont l'un est lu au rythme w_1 et l'autre au rythme w_2, le rythme le plus tardif possible pour qu'il n'y ait pas de lecture dans un buffer vide est l'infimum de w_1 et w_2 :



Lemme 3.3 (indice du $j^{ième}$ 1 du supremum $\sqcup$, de l'infimum $\sqcap$).
Soit un ensemble de mots binaires infinis $W = \{w_1, \ldots, w_n\}$. Le supremum et l'infimum de W pour la relation de précédence (notés $\sqcup W$ et $\sqcap W$) sont définis par :

$$\forall j \geq 1,\ \mathcal{I}_{\sqcup W}(j) = \max(\mathcal{I}_{w_1}(j), ..., \mathcal{I}_{w_n}(j))$$
$$\forall j \geq 1,\ \mathcal{I}_{\sqcap W}(j) = \min(\mathcal{I}_{w_1}(j), ..., \mathcal{I}_{w_n}(j))$$

Lorsque l'opérateur $\sqcup$ (resp. $\sqcap$) s'applique sur un ensemble à deux éléments $\{w_1, w_2\}$, nous adopterons la notation infixe $w_1 \sqcup w_2$.

Démonstration :

Montrons d'abord que $\sqcup W$ est un majorant de W et $\sqcap W$ un minorant de W pour la relation $\preceq$. Soit $w_x \in W$. $\forall j \geq 1,\ \mathcal{I}_{\sqcap W}(j) \leq \mathcal{I}_{w_x}(j) \leq \mathcal{I}_{\sqcup W}(j)$ donc par définition de la relation de précédence, $\sqcap W \preceq w_x \preceq \sqcup W$.
Réciproquement, soient w un minorant de W et w' un majorant de W. Montrons que $w \preceq \sqcap W$ et $\sqcup W \preceq w'$. $\forall w_x \in W,\ w \preceq w_x \preceq w'$ donc par définition de $\preceq$:
$\forall w_x \in W,\ \forall j \geq 1,\ \mathcal{I}_w(j) \leq \mathcal{I}_{w_x}(j) \leq \mathcal{I}_{w'}(j)$.
Donc $\forall j \geq 1,\ \mathcal{I}_w(j) \leq \min(\mathcal{I}_{w_1}(j), ..., \mathcal{I}_{w_n}(j)) \leq \max(\mathcal{I}_{w_1}(j), ..., \mathcal{I}_{w_n}(j)) \leq \mathcal{I}_{w'}(j)$,
c'est-à-dire $\forall j \geq 1,\ \mathcal{I}_w(j) \leq \mathcal{I}_{\sqcap W}(j) \leq \mathcal{I}_{\sqcup W}(j) \leq \mathcal{I}_{w'}(j)$.
Par définition de $\preceq$, on a donc $w \preceq \sqcap W \preceq \sqcup W \preceq w'$. □

Par exemple, le supremum des mots représentés sur la figure 3.4 est le suivant :

w_3	0	0	0	0	1	1	0	1	0	1	0	1	1	0	...
w_4	0	0	0	1	0	0	0	1	1	1	1	0	0	1	...
$w_3 \sqcup w_4$	0	0	0	0	1	0	0	1	1	1	0	1	0	1	...

Sur les graphiques, les fronts montants du supremum sont situés à l'indice maximal des fronts montants des mots considérés. Dans la figure 3.3, comme $w_1 \preceq w_2$, on a $w_1 \sqcup w_2 = w_2$.

Les fonctions de cumul du supremum et de l'infimum d'un ensemble W peuvent être calculées à partir des fonctions de cumul des mots de W. Sur les graphiques, la fonction de cumul du supremum suit le chemin le plus bas dans la zone délimitée par les fonctions de cumul des mots considérés. Celle de l'infimum suit le chemin le plus haut.

Lemme 3.4 (fonction de cumul du supremum, de l'infimum).
Soit $W = \{w_1, \ldots, w_n\}$.

$$\forall i \geq 0,\ \mathcal{O}_{\sqcup W}(i) = \min(\mathcal{O}_{w_1}(i), \ldots, \mathcal{O}_{w_n}(i)) \quad ✿✿✿$$
$$\forall i \geq 0,\ \mathcal{O}_{\sqcap W}(i) = \max(\mathcal{O}_{w_1}(i), \ldots, \mathcal{O}_{w_n}(i)) \quad ✿✿✿$$

Démonstration :

Soient deux mots w_{sup} et w_{inf} tels que
$\forall i \geq 0,\ \mathcal{O}_{w_{sup}}(i) = \min(\mathcal{O}_{w_1}(i), \ldots, \mathcal{O}_{w_n}(i))$ et $\mathcal{O}_{w_{inf}}(i) = \max(\mathcal{O}_{w_1}(i), \ldots, \mathcal{O}_{w_n}(i))$.
Montrons que $w_{sup} = \sqcup W$ et $w_{inf} = \sqcap W$.
Soit $w_x \in W$. $\forall i \geq 0,\ \mathcal{O}_{w_{inf}}(i) \geq \mathcal{O}_{w_x}(i) \geq \mathcal{O}_{w_{sup}}(i)$.
Par la proposition 3.7, $w_{inf} \preceq w_x \preceq w_{sup}$. Donc w_{inf} est un minorant et w_{sup} un majorant.
Soient w un minorant de W et w' un majorant de W. $\forall w_x \in W,\ w \preceq w_x \preceq w'$, donc par la proposition 3.7, $\forall w_x \in W,\ \forall i \geq 0,\ \mathcal{O}_w(i) \geq \mathcal{O}_{w_x}(i) \geq \mathcal{O}_{w'}(i)$.
Par conséquent,
$\forall i \geq 0,\ \mathcal{O}_w(i) \geq \max(\mathcal{O}_{w_1}(i), \ldots, \mathcal{O}_{w_n}(i)) \geq \min(\mathcal{O}_{w_1}(i), \ldots, \mathcal{O}_{w_n}(i)) \geq \mathcal{O}_{w'}(i)$,
et $\forall i \geq 0,\ \mathcal{O}_w(i) \geq \mathcal{O}_{w_{inf}}(i) \geq \mathcal{O}_{w_{sup}}(i) \geq \mathcal{O}_{w'}(i)$.
Par la proposition 3.7, on obtient $w \preceq w_{inf} \preceq w_{sup} \preceq w'$. Donc w_{inf} est le plus grand minorant et w_{sup} est le plus petit majorant. On peut en conclure que $w_{inf} = \sqcap W$ et $w_{sup} = \sqcup W$. □

Remarque 3.9 (calcul du supremum, de l'infimum)**.** On peut calculer les résultats des opérations $\sqcup$ et $\sqcap$ de manière paresseuse en transportant dans un argument supplémentaire la différence entre les valeurs des fonctions de cumul des deux mots parcourus :

$$\begin{array}{lcll} w_1 \sqcup w_2 & = & \sqcup'(0, w_1, w_2) & \\ \sqcup'(n, b.w_1, b.w_2) & = & b.\sqcup'(n, w_1, w_2) & \\ \sqcup'(n, \mathtt{1}.w_1, \mathtt{0}.w_2) & = & \mathtt{1}.\sqcup'(n+1, w_1, w_2) & \text{si } n < 0 \\ & & \mathtt{0}.\sqcup'(n+1, w_1, w_2) & \text{si } n \geq 0 \\ \sqcup'(n, \mathtt{0}.w_1, \mathtt{1}.w_2) & = & \mathtt{0}.\sqcup'(n-1, w_1, w_2) & \text{si } n < 0 \\ & & \mathtt{1}.\sqcup'(n-1, w_1, w_2) & \text{si } n \geq 0 \end{array}$$

Quand le premier argument n est négatif, la fonction de cumul du premier mot est en dessous de celle du second mot, et c'est donc l'élément du premier mot qui est reporté dans le résultat. Quand n est positif, c'est l'élément du second mot qui est reporté. L'infimum de deux mots peut être calculé de manière similaire. ◇

Effet des opérateurs sur la relation précédence

L'opérateur ***not*** est monotone par rapport à la relation $\preceq$. L'ordre de précédence entre deux mots est l'inverse de l'ordre existant entre leur négation.

Lemme 3.5 (*not* et précédence).

$$w_1 \preceq w_2 \Leftrightarrow \textbf{\textit{not}}\ w_2 \preceq \textbf{\textit{not}}\ w_1$$

Démonstration :

Par la proposition 3.7, $w_1 \preceq w_2 \Leftrightarrow \forall i > 0,\ \mathcal{O}_{w_1}(i) \geq \mathcal{O}_{w_2}(i)$.
Or $\forall i > 0,\ \mathcal{O}_{\textbf{\textit{not}}\ w_1}(i) = i - \mathcal{O}_{w_1}(i)$ et $\mathcal{O}_{\textbf{\textit{not}}\ w_2}(i) = i - \mathcal{O}_{w_2}(i)$.
Par conséquent, $w_1 \preceq w_2 \Leftrightarrow \forall i > 0,\ \mathcal{O}_{\textbf{\textit{not}}\ w_2}(i) \geq \mathcal{O}_{\textbf{\textit{not}}\ w_1}(i) \Leftrightarrow \textbf{\textit{not}}\ w_2 \preceq \textbf{\textit{not}}\ w_1$. □

L'opérateur ***on*** est lui aussi monotone par rapport à la relation $\preceq$. L'ordre de précédence entre les opérandes se retrouve dans l'ordre existant entre les résultats.

Lemme 3.6 (*on* et précédence).

$$w_1 \preceq w_3 \wedge w_2 \preceq w_4 \Rightarrow w_1\ \textbf{\textit{on}}\ w_2 \preceq w_3\ \textbf{\textit{on}}\ w_4$$

Démonstration :

Par la proposition 3.3, $\mathcal{I}_{w_1\ \textbf{\textit{on}}\ w_2}(j) = \mathcal{I}_{w_1}(\mathcal{I}_{w_2}(j))$ et $\mathcal{I}_{w_3\ \textbf{\textit{on}}\ w_4}(j) = \mathcal{I}_{w_3}(\mathcal{I}_{w_4}(j))$.
Comme $w_1 \preceq w_3$ par définition de $\preceq$, on a : $\forall j \geq 1,\ \mathcal{I}_{w_1}(j) \leq \mathcal{I}_{w_3}(j)$.
Donc $\mathcal{I}_{w_1}(\mathcal{I}_{w_2}(j)) \leq \mathcal{I}_{w_3}(\mathcal{I}_{w_2}(j))$.
Comme $w_2 \preceq w_4$ par définition de $\preceq$, on a : $\forall j \geq 1,\ \mathcal{I}_{w_2}(j) \leq \mathcal{I}_{w_4}(j)$.
Par la remarque 3.1, on a $\mathcal{I}_{w_3}(\mathcal{I}_{w_2}(j)) \leq \mathcal{I}_{w_3}(\mathcal{I}_{w_4}(j))$.
Donc $\forall j \geq 1,\ \mathcal{I}_{w_1}(\mathcal{I}_{w_2}(j)) \leq \mathcal{I}_{w_3}(\mathcal{I}_{w_4}(j))$, i.e. $\forall j \geq 1,\ \mathcal{I}_{w_1\ \textbf{\textit{on}}\ w_2}(j) \leq \mathcal{I}_{w_3\ \textbf{\textit{on}}\ w_4}(j)$.
Par conséquent, par définition de $\preceq$, on peut en conclure que $w_1\ \textbf{\textit{on}}\ w_2 \preceq w_3\ \textbf{\textit{on}}\ w_4$. □

En particulier, comme la relation de précédence est réflexive, on a :

$$\begin{array}{rcl} w_1 \preceq w_2 & \Rightarrow & w_3 \ \texttt{on}\ w_1 \preceq w_3 \ \texttt{on}\ w_2 \\ w_1 \preceq w_2 & \Rightarrow & w_1 \ \texttt{on}\ w_3 \preceq w_2 \ \texttt{on}\ w_3 \end{array}$$

Par conséquent, si deux rythmes vérifient la relation de précédence, alors les ralentir ou les échantillonner de la même manière conserve cette relation.

En ce qui concerne la première implication, il s'agit en fait d'une équivalence si w_3 contient une infinité de `1` :

Lemme 3.7 (*on*, simplification à gauche).
Si $\forall j,\ \mathcal{I}_{w_3}(j) \neq +\infty$, alors $w_3 \ \texttt{on}\ w_1 \preceq w_3 \ \texttt{on}\ w_2 \Leftrightarrow w_1 \preceq w_2$.

Démonstration :

Par définition de la relation de précédence,

$$\begin{array}{rcl} w_3 \ \texttt{on}\ w_1 \preceq w_3 \ \texttt{on}\ w_2 & \Leftrightarrow & \forall j \geq 1,\ \mathcal{I}_{w_3 \ \texttt{on}\ w_1}(j) \leq \mathcal{I}_{w_3 \ \texttt{on}\ w_1}(j) \\ & \Leftrightarrow & \forall j \geq 1,\ \mathcal{I}_{w_3}(\mathcal{I}_{w_1}(j)) \leq \mathcal{I}_{w_3}(\mathcal{I}_{w_2}(j)) \end{array}$$

Comme $\mathcal{I}_{w_3}$ est croissante et $\forall i,\ \mathcal{I}_{w_3}(i) < +\infty$,

$$\begin{array}{rcl} & \Leftrightarrow & \forall j \geq 1,\ \mathcal{I}_{w_1}(j) \leq \mathcal{I}_{w_2}(j) \\ & \Leftrightarrow & w_1 \preceq w_2 \end{array}$$

□

On peut donc faire des simplifications structurelles à gauche pour vérifier la relation de précédence. Par contre, on ne peut pas faire de simplifications structurelles à droite car la réciproque est fausse pour la seconde implication :

Lemme 3.8 (*on*, simplification à droite). $w_1 \ \texttt{on}\ w_3 \preceq w_2 \ \texttt{on}\ w_3 \overset{\not\Rightarrow}{\Leftarrow} w_1 \preceq w_2$

Démonstration :

$\not\Rightarrow$) Voici un contre exemple :
$w_1 = \texttt{101 101 101}\ldots$, $w_2 = \texttt{110 110 110}\ldots$, $w_3 = \texttt{10 10 10}\ldots$.
L'échantillonnage par w_3 supprime les `1` de rang pair :
$w_1 \ \texttt{on}\ w_3 = \texttt{100 100 100}\ldots = w_2 \ \texttt{on}\ w_3$.
Par conséquent, $w_1 \ \texttt{on}\ w_3 \preceq w_2 \ \texttt{on}\ w_3$. Cependant, le deuxième `1` de w_1 arrive après le deuxième `1` de w_2, donc $w_1 \not\preceq w_2$.

$\Leftarrow$) Par le lemme 3.6 et par reflexivité de $\preceq$.

□

Ces propriétés auront de l'influence sur les règles de simplification utilisées dans les chapitres 5 et 8.

Les opérateurs $\sqcup$ et $\sqcap$ sont monotones par rapport à la relation de précédence.

Lemme 3.9 ($\sqcup$, $\sqcap$ et précédence).

$$\begin{array}{rcl} w_1 \preceq w_3 \wedge w_2 \preceq w_4 & \Rightarrow & w_1 \sqcup w_2 \preceq w_3 \sqcup w_4 \\ w_1 \preceq w_3 \wedge w_2 \preceq w_4 & \Rightarrow & w_1 \sqcap w_2 \preceq w_3 \sqcap w_4 \end{array}$$

Démonstration :

$$\begin{array}{rcl} w_1 \preceq w_3 \wedge w_2 \preceq w_4 & \Leftrightarrow & \forall i,\ \mathcal{O}_{w_1}(i) \geq \mathcal{O}_{w_3}(i) \wedge \mathcal{O}_{w_2}(i) \geq \mathcal{O}_{w_4}(i) \\ & \Rightarrow & \forall i,\ \min(\mathcal{O}_{w_1}(i), \mathcal{O}_{w_2}(i)) \geq \min(\mathcal{O}_{w_3}(i), \mathcal{O}_{w_4}(i)) \\ & & \text{et } \max(\mathcal{O}_{w_1}(i), \mathcal{O}_{w_2}(i)) \geq \max(\mathcal{O}_{w_3}(i), \mathcal{O}_{w_4}(i)) \\ & \Leftrightarrow & \mathcal{O}_{w_1 \sqcup w_2}(i) \geq \mathcal{O}_{w_3 \sqcup w_4}(i) \text{ et } \mathcal{O}_{w_1 \sqcap w_2}(i) \geq \mathcal{O}_{w_3 \sqcap w_4}(i) \\ & \Leftrightarrow & w_1 \sqcup w_2 \preceq w_3 \sqcup w_4 \text{ et } w_1 \sqcap w_2 \preceq w_3 \sqcap w_4 \end{array}$$

□

Pour finir, on peut remarquer que on est distributif par rapport aux opérateurs $\sqcup$ et $\sqcap$.

Lemme 3.10 (on et $\sqcup$, $\sqcap$).

$$\begin{array}{rcl} w_1 \ \mathit{on}\ (w_2 \sqcup w_3) & = & (w_1 \ \mathit{on}\ w_2) \sqcup (w_1 \ \mathit{on}\ w_3) \\ (w_1 \sqcup w_2) \ \mathit{on}\ w_3 & = & (w_1 \ \mathit{on}\ w_3) \sqcup (w_1 \ \mathit{on}\ w_2) \\ w_1 \ \mathit{on}\ (w_2 \sqcap w_3) & = & (w_1 \ \mathit{on}\ w_2) \sqcap (w_1 \ \mathit{on}\ w_3) \\ (w_1 \sqcap w_2) \ \mathit{on}\ w_3 & = & (w_1 \ \mathit{on}\ w_3) \sqcap (w_1 \ \mathit{on}\ w_2) \end{array}$$

Démonstration :

$$\begin{array}{ll} \forall j \geq 1,\ \mathcal{I}_{w_1 \ \mathit{on}\ (w_2 \sqcup w_3)}(j) & = \mathcal{I}_{w_1}(\mathcal{I}_{w_2 \sqcup w_3}(j)) = \mathcal{I}_{w_1}(\max(\mathcal{I}_{w_2}(j), \mathcal{I}_{w_3}(j))) \\ & = \max\left(\mathcal{I}_{w_1}(\mathcal{I}_{w_2}(j)), \mathcal{I}_{w_1}(\mathcal{I}_{w_3}(j))\right) \text{ par la remarque 3.1} \\ & = \max\left(\mathcal{I}_{w_1 \ \mathit{on}\ w_2}(j), \mathcal{I}_{w_1 \ \mathit{on}\ w_3}(j)\right) \\ & = (w_1 \ \mathit{on}\ w_2) \sqcup (w_1 \ \mathit{on}\ w_3) \\ \forall j \geq 1,\ \mathcal{I}_{(w_1 \sqcup w_2) \ \mathit{on}\ w_3}(j) & = \mathcal{I}_{w_1 \sqcup w_2}(\mathcal{I}_{w_3}(j)) = \max\left(\mathcal{I}_{w_1}(\mathcal{I}_{w_3}(j)), \mathcal{I}_{w_2}(\mathcal{I}_{w_3}(j))\right) \\ & = \max\left(\mathcal{I}_{w_1 \ \mathit{on}\ w_3}(j), \mathcal{I}_{w_2 \ \mathit{on}\ w_3}(j)\right) \\ & = \mathcal{I}_{(w_1 \ \mathit{on}\ w_3) \sqcup (w_2 \ \mathit{on}\ w_3)}(j) \end{array}$$

Les preuves sont similaires pour l'infimum. □

3.4.2 Relation de synchronisabilité

Deux mots w_1 et w_2 sont synchronisables si la différence entre le nombre d'occurrences de $\mathtt{1}$ dans w_1 et le nombre d'occurrences de $\mathtt{1}$ dans w_2 est bornée. Cette relation est une relation d'équivalence. ✿

Définition 3.9 (synchronisabilité $\bowtie$). ✿

$$w_1 \bowtie w_2 \overset{def}{\Leftrightarrow} \exists b_1, b_2 \in \mathbb{Z},\ \forall i \geq 0,\ b_1 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq b_2$$

Dans la figure 3.3, les courbes de w_1 et w_2 restent à une distance verticale bornée l'une de l'autre ($0 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq 2$). Si cette propriété continue à être vérifiée après le dernier instant représenté sur le graphique, alors $w_1 \bowtie w_2$. En revanche, dans la figure 3.5, celle de w_5 est de plus en plus haute par rapport à celle de w_6. Si ce comportement continue infiniment, alors $w_5 \not\bowtie w_6$.

Une première définition de la relation de synchronisabilité a été donnée dans [CDE+06] :

$$w_1 \bowtie w_2 \overset{def}{\Leftrightarrow} \exists d_1, d_2 \in \mathbb{N},\ w_1 \preceq 0^{d_2}.w_2 \wedge w_2 \preceq 0^{d_1}.w_1$$

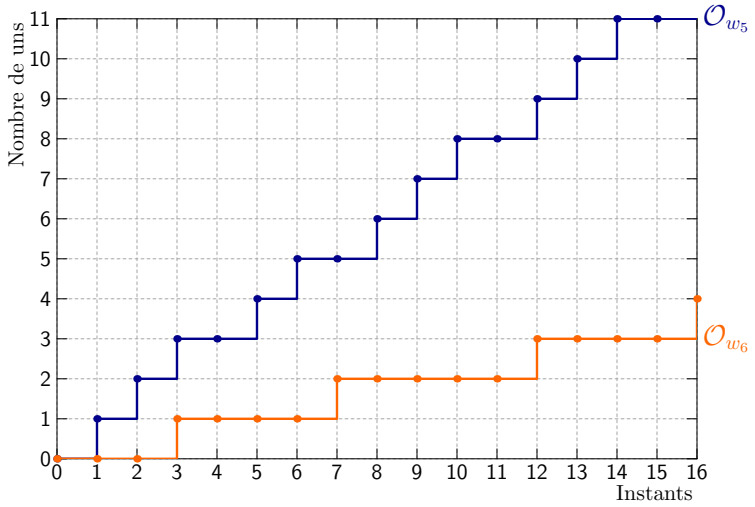


FIG. 3.5 – Fonctions de cumul des mots $w_5 = \texttt{1110110111011}\ldots$ et $w_6 = \texttt{0010001000010}\ldots$.

Elle peut être reformulée ainsi : $w_1 \bowtie w_2 \Leftrightarrow \exists d_1, d_2 \in \mathbb{N},\ -d_1 \leq \mathcal{I}_{w_1}(j) - \mathcal{I}_{w_2}(j) \leq d_2$. Cette première définition est plus restrictive que la définition 3.9, qui correspond exactement à ce que l'on veut exprimer ici : si $w_1 \bowtie w_2$, la taille de buffer nécessaire pour communiquer de w_1 à w_2 ou de w_2 à w_1 est bornée. Par exemple, suivant la définition choisie ici, le mot $w_1 = \texttt{1}^1\texttt{0}\texttt{1}^2\texttt{0}\texttt{1}^3\texttt{0}\texttt{1}^4\ldots\texttt{1}^n$ est synchronisable avec le mot $w_2 = \texttt{0}^1\texttt{1}\texttt{0}^2\texttt{1}\texttt{0}^3\texttt{1}\texttt{0}^4\texttt{1}\ldots\texttt{0}^n\texttt{1}$, et en effet la taille du buffer nécessaire pour communiquer de l'un à l'autre vaut 1 :

w_1	1	0	1	0	0	1	0	0	0	1	0	0	...
w_2	0	1	0	0	1	0	0	0	1	0	0	0	...
$\mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i)$	1	0	1	1	0	1	1	1	0	1	1	1	...

Ces deux mots ne sont pas synchronisables au sens de la définition de [CDE+06]. Il est à noter que dans le cas où les mots ne divergent pas, c'est-à-dire dans le cas où les distances entre deux 1 successifs et deux 0 successifs sont bornées, les deux définitions sont équivalentes.

Contrairement à l'intuition, il ne suffit pas que w_1 et w_2 aient le même taux asymptotique de 1 pour qu'ils soient synchronisables. Par exemple $w_1 = \texttt{1}^1\texttt{0}^1\texttt{1}^2\texttt{0}^2\texttt{1}^3\texttt{0}^3\ldots\texttt{1}^n\texttt{0}^n$ et $w_2 = \texttt{0}^1\texttt{1}^1\texttt{0}^2\texttt{1}^2\texttt{0}^3\texttt{1}^3\ldots\texttt{0}^n\texttt{1}^n$ ont tous deux un taux asymptotique valant $\frac{1}{2}$, mais la différence entre le nombre de 1 vus dans w_1 et le nombre de 1 vus dans w_2 croît à l'infini :

w_1	1	0	1	1	0	0	1	1	1	0	0	0	...
w_2	0	1	0	0	1	1	0	0	0	1	1	1	...
$\mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i)$	1	0	1	2	1	0	1	2	3	2	1	0	...

Ces deux mots ne sont donc pas synchronisables. Ceci est lié au fait qu'ils divergent. Si les mots ne divergent pas, alors il suffit qu'ils soient de même taux asymptotique pour être synchronisables.

Remarque 3.10. Comme pour $i = 0$ on a $\mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) = 0$, on sait que si $w_1 \bowtie w_2$, c'est-à-dire s'il existe des entiers b_1 et b_2 tels que $\forall i \geq 0,\ b_1 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq b_2$, alors $b_1 \leq 0$ et $b_2 \geq 0$. ◇

Effet des opérateurs sur la relation de synchronisabilité

L'opérateur $\mathtt{not}$ est compatible avec la relation $\bowtie$.

Lemme 3.11 ($\mathtt{not}$ et synchronisabilité). $w_1 \bowtie w_2 \Leftrightarrow \mathtt{not}\ w_1 \bowtie \mathtt{not}\ w_2$

Démonstration :

Par la remarque 3.4, $\mathcal{O}_{\mathtt{not}\ w_1}(i) - \mathcal{O}_{\mathtt{not}\ w_2}(i) = -(\mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i))$. Donc s'il existe des entiers b_1 et b_2 tels que $\forall i \geq 0,\ b_1 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq b_2$ alors il existe des entiers b'_1 et b'_2 tels que $b_{1'} \leq \mathcal{O}_{\mathtt{not}\ w_1}(i) - \mathcal{O}_{\mathtt{not}\ w_2}(i) \leq b_{2'}$ (on a $b'_1 = -b_1$ et $b'_2 = -b_2$), et réciproquement. On peut conclure par définition de $\bowtie$. □

L'opérateur $\mathtt{on}$ est lui aussi compatible avec la relation $\bowtie$.

Lemme 3.12 ($\mathtt{on}$ et synchronisabilité). $w_1 \bowtie w_3 \wedge w_2 \bowtie w_4 \Rightarrow w_1\ \mathtt{on}\ w_2 \bowtie w_3\ \mathtt{on}\ w_4$

Démonstration :

Montrons qu'il existe des entiers b_{12} et b_{34} tels que
$\forall i \geq 0,\ b_{12} \leq \mathcal{O}_{(w_1\ \mathtt{on}\ w_2)}(i) - \mathcal{O}_{(w_3\ \mathtt{on}\ w_4)}(i) \leq b_{34}$.
Par la proposition 3.2, $\forall i \geq 0,\ \mathcal{O}_{(w_1\ \mathtt{on}\ w_2)}(i) - \mathcal{O}_{(w_3\ \mathtt{on}\ w_4)}(i) = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i)) - \mathcal{O}_{w_4}(\mathcal{O}_{w_3}(i))$.
Comme $w_1 \bowtie w_3$, par la remarque 3.10, il existe des entiers naturels b_1 et b_3 tels que
$\forall i \geq 0,\ -b_1 + \mathcal{O}_{w_3}(i) \leq \mathcal{O}_{w_1}(i) \leq b_3 + \mathcal{O}_{w_3}(i)$.
La fonction $\mathcal{O}$ est croissante (voir remarque 3.2) donc $\forall i \geq 0$,
$\mathcal{O}_{w_2}(-b_1 + \mathcal{O}_{w_3}(i)) - \mathcal{O}_{w_4}(\mathcal{O}_{w_3}(i)) \leq \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i)) - \mathcal{O}_{w_4}(\mathcal{O}_{w_3}(i)) \leq$
$\mathcal{O}_{w_2}(b_3 + \mathcal{O}_{w_3}(i)) - \mathcal{O}_{w_4}(\mathcal{O}_{w_3}(i))$.
Par la remarque 3.2, et comme $b_1 \geq 0$ et $b_3 \geq 0$, on a
$\mathcal{O}_{w_2}(\mathcal{O}_{w_3}(i)) \leq b_1 + \mathcal{O}_{w_2}(-b_1 + \mathcal{O}_{w_3}(i))$ et $\mathcal{O}_{w_2}(b_3 + \mathcal{O}_{w_3}(i)) \leq b_3 + \mathcal{O}_{w_2}(\mathcal{O}_{w_3}(i))$.
Donc $-b_1 + \mathcal{O}_{w_2}(\mathcal{O}_{w_3}(i)) - \mathcal{O}_{w_4}(\mathcal{O}_{w_3}(i)) \leq \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i)) - \mathcal{O}_{w_4}(\mathcal{O}_{w_3}(i)) \leq$
$b_3 + \mathcal{O}_{w_2}(\mathcal{O}_{w_3}(i)) - \mathcal{O}_{w_4}(\mathcal{O}_{w_3}(i))$.
Comme $w_2 \bowtie w_4$, par la remarque 3.10, il existe des entiers naturels b_2 et b_4 tels que
$\forall i \geq 0, -b_2 \leq \mathcal{O}_{w_2}(i) - \mathcal{O}_{w_4}(i) \leq b_4$.
Étant donné que $\forall i \geq 0,\ \mathcal{O}_{w_3}(i) \geq 0$, on a $-b_2 \leq \mathcal{O}_{w_2}(\mathcal{O}_{w_3}(i)) - \mathcal{O}_{w_4}(\mathcal{O}_{w_3}(i)) \leq b_4$,
donc $-b_1 - b_2 \leq \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i)) - \mathcal{O}_{w_4}(\mathcal{O}_{w_3}(i)) \leq b_3 + b_4$.
Par conséquent, $b_{12} = -b_1 - b_2$ et $b_{34} = b_3 + b_4$ conviennent. □

Comme la relation de synchronisabilité est réflexive, on a donc en particulier les implications :

$$\begin{aligned} w_1 \bowtie w_2 \quad &\Rightarrow \quad w_3\ \mathtt{on}\ w_1 \bowtie w_3\ \mathtt{on}\ w_2 \\ w_1 \bowtie w_2 \quad &\Rightarrow \quad w_1\ \mathtt{on}\ w_3 \bowtie w_2\ \mathtt{on}\ w_3 \end{aligned}$$

Les réciproques sont vérifiées si w_3 contient une infinité de $\mathtt{1}$ et ne diverge pas :

Lemme 3.13 ($\mathtt{on}$, simplifications à gauche et à droite).
Si $\forall j,\ \mathcal{I}_{w_3}(j) \neq +\infty$, alors

$$w_3\ \mathtt{on}\ w_1 \bowtie w_3\ \mathtt{on}\ w_2 \Rightarrow w_1 \bowtie w_2$$

Si de plus $\exists b,\ \forall j,\ \mathcal{I}_{w_3}(j+1) - \mathcal{I}_{w_3}(j) \leq b$ alors

$$w_1\ \mathtt{on}\ w_3 \bowtie w_2\ \mathtt{on}\ w_3 \Rightarrow w_1 \bowtie w_2$$

Démonstration :

Montrons que $w_3 \texttt{ on } w_1 \bowtie w_3 \texttt{ on } w_2 \Rightarrow w_1 \bowtie w_2$

$$\begin{aligned} w_3 \texttt{ on } w_1 \bowtie w_3 \texttt{ on } w_2 &\Leftrightarrow \exists b_1, b_2,\ \forall i,\ b_1 \leq \mathcal{O}_{w_3 \texttt{ on } w_1}(i) - \mathcal{O}_{w_3 \texttt{ on } w_2}(i) \leq b_2 \\ &\Leftrightarrow \exists b_1, b_2,\ \forall i,\ b_1 \leq \mathcal{O}_{w_1}(\mathcal{O}_{w_3}(i)) - \mathcal{O}_{w_2}(\mathcal{O}_{w_3}(i)) \leq b_2 \end{aligned}$$

Comme $\forall j,\ \mathcal{I}_{w_3}(j) \neq +\infty$, w_3 contient une infinité de $\texttt{1}$,
donc $\forall i \in \mathbb{N}, \exists i' \in \mathbb{N},\ \mathcal{O}_{w_3}(i') = i$.
Par conséquent, $\exists b_1, b_2,\ \forall i,\ b_1 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq b_2$.

Montrons que $w_1 \texttt{ on } w_3 \bowtie w_2 \texttt{ on } w_3 \Rightarrow w_1 \bowtie w_2$

$$\begin{aligned} w_1 \texttt{ on } w_3 \bowtie w_2 \texttt{ on } w_3 &\Leftrightarrow \exists b_1, b_2,\ \forall i,\ b_1 \leq \mathcal{O}_{w_1 \texttt{ on } w_3}(i) - \mathcal{O}_{w_2 \texttt{ on } w_3}(i) \leq b_2 \\ &\Leftrightarrow \exists b_1, b_2,\ \forall i,\ b_1 \leq \mathcal{O}_{w_3}(\mathcal{O}_{w_1}(i)) - \mathcal{O}_{w_3}(\mathcal{O}_{w_2}(i)) \leq b_2 \end{aligned}$$

Comme w_3 contient une infinité de $\texttt{1}$, $\mathcal{O}_{w_3}$ est croissante, et comme w_3 ne diverge pas, la distance entre deux $\texttt{1}$ successifs est bornée, donc on peut en déduire que $\exists b_1, b_2,\ \forall i,\ b_1 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq b_2$.

□

3.4.3 Relation d'adaptabilité

La relation d'adaptabilité sur les mots est la conjonction des relations de précédence et de synchronisabilité. La première assure l'absence de lectures dans un buffer vide et la seconde assure que le nombre de valeurs en attente de lecture dans le buffer ne croît pas à l'infini. Par conséquent, si w_1 est adaptable à w_2, alors un flot de rythme w_1 peut être consommé sur le rythme w_2 par insertion d'un buffer borné.

Définition 3.10 (adaptabilité $<:$). ✿

$$w_1 <: w_2 \quad \overset{def}{\Leftrightarrow} \quad w_1 \preceq w_2 \ \wedge\ w_1 \bowtie w_2$$

Comme cette relation est définie par la conjonction des relations de précédence et de synchronisabilité, elle hérite des propriétés suivantes :

Proposition 3.8 ($<:$, simplifications structurelles).

$$\texttt{not } w_1 <: \texttt{not } w_2 \quad \Leftrightarrow \quad w_2 <: w_1$$

$$w_3 \texttt{ on } w_1 <: w_3 \texttt{ on } w_2 \quad \Leftrightarrow \quad w_1 <: w_2 \quad (\text{si } \forall j,\ \mathcal{I}_{w_3}(j) \neq +\infty)$$

$$w_1 \texttt{ on } w_3 <: w_2 \texttt{ on } w_3 \quad \begin{matrix} \not\Rightarrow \\ \not\Leftarrow \end{matrix} \quad w_1 <: w_2$$

Démonstration :

Par définition de $<:$, et par les lemmes 3.5, 3.11, 3.7, 3.8 et 3.13. □

On peut donc effectuer des simplifications structurelles des $\texttt{not}$, et à gauche des $\texttt{on}$ pour vérifier la relation d'adaptabilité. Par contre, les simplifications à droite des $\texttt{on}$ sont correctes mais pas complètes.

Taille du buffer Considérons un buffer qui prend en entrée un flot sur le rythme w_1, et fournit en sortie ce flot sur le rythme w_2. Le nombre d'éléments présents à chaque instant i dans le buffer est la différence entre le nombre de valeurs qui ont été écrites dans le buffer ($\mathcal{O}_{w_1}(i)$) et le nombre de valeurs qui y ont été lues ($\mathcal{O}_{w_2}(i)$) :

$$size_i(w_1, w_2) = \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i)$$

Une valeur négative pour cette différence signifie que plus de valeurs ont été lues qu'écrites, c'est-à-dire que des lectures ont été effectuées dans le buffer alors qu'il était vide.

La taille de buffer nécessaire et suffisante est le nombre maximal de valeurs présentes dans le buffer durant l'exécution :

$$size(w_1, w_2) = \max_{i \in \mathbb{N}^*}(\mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i))$$

Reprenons l'exemple de la figure 3.3. Si le comportement décrit ci-dessous continue à l'infini, la taille de buffer nécessaire est 2.

w_1	1	1	0	1	0	1	1	0	1	0	...	← Écriture dans le buffer
w_2	0	0	0	1	1	1	0	0	1	1	...	← Lecture dans le buffer
$\mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i)$	1	2	2	2	1	1	2	2	2	1	...	← Nombre d'éléments dans le buffer

Ralentir deux horloges de la même manière ne change pas la taille de buffer nécessaire :

Lemme 3.14 (*size*, simplification structurelle).
Si $\forall j,\ \mathcal{I}_w(j) \neq +\infty$, on a $size(w \ \texttt{on}\ w_1, w \ \texttt{on}\ w_2) = size(w_1, w_2)$.

Démonstration :

$$\begin{array}{rcl} size(w \ \texttt{on}\ w_1, w \ \texttt{on}\ w_2) & = & \max_{i \in \mathbb{N}^*}(\mathcal{O}_{w \ \texttt{on}\ w_1}(i) - \mathcal{O}_{w \ \texttt{on}\ w_2}(i)) \\ & = & \max_{i \in \mathbb{N}^*}(\mathcal{O}_{w_1}(\mathcal{O}_w(i)) - \mathcal{O}_{w_2}(\mathcal{O}_w(i))) \end{array}$$

Comme w contient une infinité de 1, $\forall i,\ \exists i',\ \mathcal{O}_w(i') = i$. Donc :
$size(w \ \texttt{on}\ w_1, w \ \texttt{on}\ w_2) = \max_{i \in \mathbb{N}^*}(\mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i)) = size(w_1, w_2)$ □

Par contre la taille de buffer nécessaire entre deux horloges échantillonnées de la même manière est en général bien inférieure à la taille de buffer nécessaire sans l'échantillonnage, car l'échantillonnage diminue le nombre de valeurs transitant sur les flots. Par exemple, $size(\texttt{(111000) on (100)}, \texttt{(000111) on (100)}) = 1 \ \geq \ size(\texttt{(111000)}, \texttt{(000111)}) = 3$.

Communication par buffers La relation d'adaptabilité garantit qu'une communication peut être effectuée de manière correcte à travers un buffer de taille bornée :

Théorème 3.1 (communication par buffers). Si w_1 est adaptable à w_2, la consommation d'un flot d'horloge w_1 peut être faite sur l'horloge w_2 par insertion d'un buffer de taille bornée et sans lectures dans un buffer vide :

$$w_1 <: w_2 \ \Rightarrow \ (size(w_1, w_2) < +\infty) \wedge (\forall i,\ size_i(w_1, w_2) \geq 0)$$

Démonstration :

La relation d'adaptabilité est la conjonction de la relation de précédence et de la relation de synchronisabilité. Par définition de la relation de synchronisabilité et par la formule donnant la taille du buffer, on a la garantie que $size(w_1, w_2) < +\infty$. Par définition de la relation de précédence, on a la garantie que $\forall i \geq 0,\ \mathcal{O}_{w_1}(i) \geq \mathcal{O}_{w_2}(i)$. □

3.5 Conclusion

Nous avons présenté dans ce chapitre un ensemble d'opérations algébriques permettant de décrire les mots binaires infinis, de les transformer et de les comparer. En particulier, nous avons défini une relation sur les mots binaires nommée relation d'adaptabilité, qui garantit qu'une communication à travers un buffer est correcte. Cependant, les calculs et relations présentés ne constituent pas des algorithmes qui terminent. Ils ne peuvent donc pas être mis en œuvre dans le cadre du calcul d'horloge de Lucy-n. Nous nous intéressons dans la suite à deux sous-ensembles des mots binaires infinis sur lesquels on sait effectuer les calculs et vérifier les relations.

Deuxième partie

Horloges périodiques

Chapitre 4

Mots binaires infinis ultimement périodiques

Dans ce chapitre, on s'intéresse au cas particulier des mots binaires qui ont un comportement périodique à partir d'un certain indice. Sur ces mots, on sait tester l'égalité, calculer les opérations et vérifier les relations décrites dans le chapitre 3. En particulier, on sait garantir statiquement qu'un mot ultimement périodique est adaptable à un autre mot ultimement périodique. Ces mots seront utilisés dans le chapitre 5 pour décrire des horloges sur lesquelles on sera en mesure d'assurer que la communication peut se faire à travers un buffer de taille bornée.

Nous reprenons l'ensemble des opérations et relations présentées précédemment, en donnant des algorithmes permettant de les calculer.

4.1 Mots binaires infinis ultimement périodiques

On appellera *mots binaires infinis ultimement périodiques*, ou plus simplement *mots binaires ultimement périodiques* et on notera $u(v)$, les mots constitués d'un préfixe fini u éventuellement vide (ε) suivi de la répétition infinie d'un mot fini v.

$$\begin{array}{lcl} p & ::= & u(v) \\ u & ::= & \varepsilon \mid \mathtt{0} \mid \mathtt{1} \mid \mathtt{0}.u \mid \mathtt{1}.u \\ v & ::= & \mathtt{0} \mid \mathtt{1} \mid \mathtt{0}.v \mid \mathtt{1}.v \end{array}$$

Définition 4.1 (mot binaire ultimement périodique p).

$$p = u(v) \overset{def}{\Leftrightarrow} p = u.w \text{ avec } w = v.w$$

Par exemple, $p = \mathtt{1101(110)} = \mathtt{1101\ 110\ 110\ 110}\ \ldots$

Ce sous-ensemble des mots binaires infinis sera noté $\mathbb{Q}_2$, et correspond aux nombres 2-adiques rationnels [Vui94]. Nous utiliserons aussi les termes raccourcis *mots binaires périodiques* et *mots périodiques* pour les désigner.

Lemme 4.1 (éléments de p).

$$p = u(v) \Leftrightarrow p[i] = \begin{cases} u[i] \text{ si } i \leq |u| \\ v[i'] \text{ si } i = |u| + x \times |v| + i' \text{ avec } x \in \mathbb{N} \text{ et } 1 \leq i' \leq |v| \end{cases}$$

Par conséquent, si $i > |u|$, alors $p[i] = v[1 + (i - |u| - 1) \bmod |v|]$.

Démonstration :

Par définition de la fonction d'accès aux éléments (définition 3.1 p. 50) et par définition des mots périodiques (définition 4.1).
Preuve de la deuxième formulation : appliquons ce lemme dans le cas où $i > |u|$.
On a $p[i] = v[i']$ si $i = |u| + x \times |v| + i'$ avec $x \in \mathbb{N}$ et $1 \leq i' \leq |v|$.
Posons $i' = 1 + i''$ avec $0 \leq i'' < |v|$. Alors $i = |u| + x \times |v| + 1 + i''$,
c'est-à-dire $i - |u| - 1 = i'' + x \times |v|$. Par conséquent, $i'' = (i - |u| - 1) \bmod |v|$ et on peut en conclure que $p[i] = v[1 + (i - |u| - 1) \bmod |v|]$. □

Un même mot ultimement périodique peut être représenté d'une infinité de façons différentes. Par exemple `(10)` = `(1010)` = `1(01)` = ... Il existe une forme normale qui est la représentation avec le préfixe le plus court possible, et la période la plus courte possible. Cependant, pour certaines opérations, il est plus pratique de mettre les opérandes sous une forme plus longue que leurs formes normales respectives. On pourra par exemple ajuster les formes des opérandes pour qu'elles soient de même taille, ou qu'elles contiennent le même nombre de `1`, ou encore que la taille de la première soit égale au nombre de `1` de la seconde.

Remarque 4.1. Soit un mot binaire ultimement périodique $p = u(v)$. Voici des manipulations possibles sur la forme de p.

Rallonger le préfixe :

- Pour toute taille $t \geq |u|$, p peut être mis sous la forme $u'(v')$ telle que $|u'| = t$.
  On a alors $u' = p[1..t]$ et $v' = p[t + 1..t + |v|]$.
  Par exemple, on peut rallonger le préfixe de $p =$ `1101(110)` de deux éléments. On obtient p sous la forme `1101 11(0 11)`.
- Si v contient au moins un `1`, pour tout nombre n de `1` supérieur ou égal à $|u|_1$, p peut être mis sous la forme $u'(v')$ telle que $|u'|_1 = n$.
  On a alors $u' = p[1..\mathcal{I}_p(n)]$ et $v' = p[(\mathcal{I}_p(n) + 1)..(\mathcal{I}_p(n) + |v|)]$.
  Par exemple, on peut rallonger le préfixe de $p =$ `1101(110)` de manière à ce qu'il contienne trois `1` supplémentaires. On obtient p sous la forme `1101 110 1(10 1)`.

Répéter le motif périodique :

- Pour toute taille t multiple de $|v|$, p peut être mis sous la forme $u(v')$ telle que $|v'| = t$.
  On a alors $v' = v^{t/|v|}$.
  Par exemple on peut tripler la taille du motif périodique de $p =$ `1101(110)`. On obtient p sous la forme `1101(110 110 110)`.
- Pour tout nombre n de `1` multiple de $|v|_1$, p peut être mis sous la forme $u(v')$ avec $|v'|_1 = n$. On a alors $v' = v^{n/|v|_1}$.
  Par exemple, on peut doubler le nombre de `1` du motif périodique de $p =$ `1101(110)`. On obtient p sous la forme $p =$ `1101(110 110)`.

◇

Ainsi, pour tester l'égalité de deux mots ultimement périodiques $p_1 = u_1(v_1)$ et $p_2 = u_2(v_2)$, il suffit d'ajuster leurs préfixes à la même taille, leurs parties périodiques à la même taille, et de tester l'égalité de manière structurelle. Il n'est en fait pas nécessaire de mettre explicitement les mots sous la même forme. Cet ajustement produisant des préfixes de taille $\max(|u_1|, |u_2|)$ et des suffixes de taille ppcm $(|v_1|, |v_2|)$, il suffit de tester l'égalité des éléments jusqu'à l'indice $\max(|u_1|, |u_2|) +$ ppcm $(|v_1|, |v_2|)$:

Lemme 4.2 (test d'égalité de p_1 et p_2). Soient $p_1 = u_1(v_1)$ et $p_2 = u_2(v_2)$.

$$p_1 = p_2 \Leftrightarrow \forall i,\ 1 \leq i \leq \max(|u_1|, |u_2|) + \text{ ppcm } (|v_1|, |v_2|),\ p_1[i] = p_2[i]$$

Démonstration :

Soient $p'_1 = u'_1(v'_1)$ et $p'_2 = u'_2(v'_2)$ tels que $p'_1 = p_1$ et $p'_2 = p_2$, et tels que $|u'_1| = |u'_2| = \max(|u_1|, |u_2|)$ et $|v'_1| = |v'_2| = \text{ ppcm } (|v_1|, |v_2|)$. D'après la remarque 4.1, p'_1 et p'_2 existent. Comme $\forall i \geq 1,\ p'_1[i] = p_1[i]$ et $p'_2[i] = p_2[i]$,
si $\forall i,\ 1 \leq i \leq \max(|u_1|, |u_2|) + \text{ ppcm } (|v_1|, |v_2|),\ p_1[i] = p_2[i]$,
alors $\forall i,\ 1 \leq i \leq \max(|u_1|, |u_2|) + \text{ ppcm } (|v_1|, |v_2|),\ p'_1[i] = p'_2[i]$. Par conséquent, d'après le lemme 4.1, $\forall i,\ 1 \leq i \leq |u'_1|,\ u'_1[i] = u'_2[i]$ et $\forall i,\ 1 \leq i \leq |v'_1|,\ v'_1[i] = v'_2[i]$.
On peut en déduire que p'_1 et p'_2 sont structurellement égaux et donc que que $p_1 = p_2$. □

Les indices des `1` dans les mots périodiques peuvent être calculés en fonction des indices des `1` dans leur préfixe et leur partie périodique. Si le préfixe contient au moins j `1`, alors l'indice du $j^{ième}$ `1` dans $p = u(v)$ est l'indice du $j^{ième}$ `1` dans u. Dans le cas où u contient moins de j `1`, l'indice du $j^{ième}$ `1` dans p est la taille de u additionnée à l'indice du $(j - |u|_1)^{ième}$ `1` dans (v), c'est-à-dire un certain nombre de fois $|v|$ plus l'indice du $(1 + (j - |u|_1 - 1) \bmod |v|_1)^{ième}$ `1` dans v.

Lemme 4.3 (indice du $j^{ième}$ `1` de p).

$$p = u(v) \Leftrightarrow \mathcal{I}_p(j) = \begin{cases} \mathcal{I}_u(j) \text{ si } j \leq |u|_1 \\ |u| + x \times |v| + \mathcal{I}_v(j') \text{ si } j = |u|_1 + x \times |v|_1 + j' \text{ avec } x \in \mathbb{N},\ 1 \leq j' \leq |v|_1 \end{cases}$$

Dans le cas particulier où $|v|_1 = 0$, on a $\mathcal{I}_p(j) = +\infty$ si $j \geq |u|_1$.

Une autre formulation de ce lemme, dans le cas où $j > |u|_1$ et $|v|_1 > 0$, sera utilisée dans certaines preuves :

$$\mathcal{I}_p(j) = |u| + \left\lfloor \frac{j - |u|_1 - 1}{|v|_1} \right\rfloor \times |v| + \mathcal{I}_v(1 + (j - |u|_1 - 1) \mod |v|_1)$$

Démonstration :

Par définition de la fonction qui donne l'indice des `1` (définition 3.2 p. 51) et des mots périodiques (définition 4.1).
Preuve de la deuxième formulation : appliquons ce lemme dans le cas où $j > |u|_1$ et $|v|_1 > 0$. On a $\mathcal{I}_p(j) = |u| + x \times |v| + \mathcal{I}_v(j')$ si $j = |u|_1 + x \times |v|_1 + j'$ avec $x \in \mathbb{N}$ et $1 \leq j' \leq |v|_1$. Posons $j' = 1 + j''$ avec $0 \leq j'' < |v|_1$. Alors $j = |u|_1 + x \times |v|_1 + 1 + j''$, c'est-à-dire $j - |u|_1 - 1 = x \times |v|_1 + j''$. Par conséquent, $j'' = (j - |u|_1 - 1) \bmod |v|_1$ et $x = \left\lfloor \frac{j - |u|_1 - 1}{|v|_1} \right\rfloor$ et on peut en conclure que $\mathcal{I}_p(j) = |u| + \left\lfloor \frac{j - |u|_1 - 1}{|v|_1} \right\rfloor \times |v| + \mathcal{I}_v(1 + (j - |u|_1 - 1) \mod |v|_1)$. □

Lemme 4.4 (fonction de cumul de p).

$$p = u(v) \Leftrightarrow \mathcal{O}_p(i) = \begin{cases} \mathcal{O}_u(i) \text{ si } i \leq |u| \\ |u|_1 + x \times |v|_1 + \mathcal{O}_v(i') \text{ si } i = |u| + x \times |v| + i' \text{ avec } x \in \mathbb{N} \text{ et } 0 \leq i' < |v| \end{cases}$$

Par conséquent, si $i \geq |u|$, $\mathcal{O}_p(i) = |u|_1 + \left\lfloor \frac{i-|u|}{|v|} \right\rfloor \times |v|_1 + \mathcal{O}_v((i-|u|) \mod |v|)$.
Remarquons aussi que $\forall i \geq |u|,\ \mathcal{O}_p(i+|v|) = \mathcal{O}_p(i) + |v|_1$.

Démonstration :

Par définition de la fonction de cumul (définition 3.3 p. 51) et des mots périodiques (définition 4.1).
Preuve de la deuxième formulation : appliquons ce lemme dans le cas où $i \geq |u|$.
$x = \left\lfloor \frac{i-|u|}{|v|} \right\rfloor$ et $i' = (i-|u|) \mod |v|$,
donc on a bien $\mathcal{O}_p(i) = |u|_1 + \left\lfloor \frac{i-|u|}{|v|} \right\rfloor \times |v|_1 + \mathcal{O}_v((i-|u|) \mod |v|)$. □

Enfin, à partir de la formule de la fonction de cumul, on peut déduire la formule du taux d'un mot périodique.

Lemme 4.5 (taux de 1 dans p). $p = u(v) \Leftrightarrow rate(p) = \frac{|v|_1}{|v|}$

Démonstration :

Par définition du taux et par le lemme 4.4. □

4.2 Opérateurs *not* et *on* sur $\mathbb{Q}_2$

L'ensemble des mots binaires ultimement périodiques est clos par application des opérations *not* et *on*. Pour calculer ces opérations, il suffit donc d'appliquer leur définition jusqu'à l'atteinte du comportement périodique du résultat. La difficulté consiste à trouver la taille du préfixe et de la partie périodique du résultat.

L'algorithme de l'opération *not* sur les mots binaires périodiques est trivialement défini car le préfixe et la partie périodique gardent la même taille :

Lemme 4.6 (calcul de *not* p).
Soit $p = u(v)$. Alors *not* $p = u'(v') \in \mathbb{Q}_2$ et on a $|u'| = |u|$ et $|v'| = |v|$.

En appliquant les définitions de *not* et des mots périodiques, on obtient les valeurs suivantes pour les éléments de u' et v' : $\forall i,\ 1 \leq i \leq |u'|,\ u'[i] = \text{ not } u[i]$
$\forall i,\ 1 \leq i \leq |v'|,\ v'[i] = \text{ not } v[i]$
On peut remarquer que $|u'|_1 = |u| - |u|_1$ et $|v'|_1 = |v| - |v|_1$.

Démonstration :

Soit $p' = u'(v')$. Par définition, $\forall i \geq 1,\ p'[i] = \text{ not } p[i]$.
Donc $\forall i,\ 1 \leq i \leq |u|,\ p'[i] = \text{ not } u[i]$ et $\forall i \geq |u|,\ p'[i] = \text{ not } v[1 + (i-|u|-1) \mod |v|]$.
Par le lemme 4.1, on peut conclure que $p' = u'(v')$ avec u' et v' comme spécifiés dans la proposition. □

Par exemple, *not* 1101(110) = 0010(001).

Le calcul des tailles du préfixe et de la partie périodique du résultat d'une opération `on` est plus compliqué. Le cas le plus simple est celui où le nombre de `1` du préfixe et de la partie périodique du premier mot correspond à la taille du préfixe et de la partie périodique du second :

p_1	1	1	0	1	(	1	1	1	0	0	1	1	0	)
p_2	1	0		1	(	1	0	0			1	0		)
p_1 `on` p_2	1	0	0	1	(	1	0	0	0	0	1	0	0	)

Dans ce cas, les tailles du préfixe et de la partie périodique du résultat sont celles du premier mot. En effet, le premier parcours de la partie périodique du premier mot commence en même temps que celui de la partie périodique du second, et c'est là que commence la partie périodique du résultat. De plus, les parcours suivants de la partie périodique des deux mots recommencent en même temps, et c'est aussi là que recommence le comportement périodique du résultat.

Nous avons vu dans la remarque 4.1 qu'il est possible d'augmenter la taille et le nombre de `1` du préfixe et de la partie périodique des mots. Ainsi, on peut toujours ajuster les opérandes du `on` afin qu'ils aient la forme de celles du cas simple ci-dessus.

Par exemple, si $p_1 =$ `1101(110)` et $p_2 =$ `1011(11110)`, alors on peut mettre p_1 sous la forme `1101 1(10 110 110 110 110 1)` et p_2 sous la forme `1011(11110 11110)` :

p_1	1	1	0	1	.	1	(	1	0	.	1	1	0	.	1	1	0	.	1	1	0	.	1	)
p_2	1	0		1	1		(	1		1	1		1	0	.		1	1		1	1		0	)
p_1 `on` p_2	1	0	0	1	1		(	1	0	1	1	0	1	0	0	1	1	0	1	1	0	0		)

Le préfixe et la partie périodique du résultat sont alors de la même taille que ceux de p_1, comme dans l'exemple précédent.

Il n'est en fait pas indispensable d'allonger explicitement la taille des opérandes, on peut se contenter de calculer la taille qu'ils auraient après cet ajustement pour avoir la taille du résultat. Si l'on veut effectuer l'opération p_1 `on` p_2, on suppose donc qu'on mette p_1 sous la forme $u'_1(v'_1)$ et p_2 sous la forme $u'_2(v'_2)$ telles que $|u'_1|_1 = |u'_2|$ et $|v'_1|_1 = |v'_2|$. Comme on veut obtenir des opérandes de taille minimale, et qu'on peut augmenter le préfixe d'un nombre quelconque d'éléments, on choisit $|u'_1|_1 = |u'_2| = \max(|u_1|_1, |u_2|)$. On a alors $|u'_1| = \max(|u_1|, \mathcal{I}_{p_1}(|u_2|))$. Comme on ne peut que répéter le comportement périodique, on choisit $|v'_1|_1 = |v'_2| =$ ppcm $(|v_1|_1, |v_2|)$. On a alors $|v'_1| = \frac{\text{ppcm}\,(|v_1|_1, |v_2|)}{|v_1|_1} \times |v_1|$. Le résultat de p_1 `on` p_2 est de la taille de $u'_1(v'_1)$.

L'algorithme de l'opération `on` sur les mots binaires périodiques est donc défini par :

Proposition 4.1 (calcul de p_1 `on` p_2).
Soient $p_1 = u_1(v_1)$ et $p_2 = u_2(v_2)$. Alors p_1 `on` $p_2 = u_3(v_3) \in \mathbb{Q}_2$ avec :

$$|u_3| = \begin{cases} |u_1| & \text{si } |u_2| = 0 \\ \max\left(|u_1|, \mathcal{I}_{p_1}(|u_2|)\right) & \text{sinon} \end{cases}$$

$$|v_3| = \begin{cases} 1 & \text{si } |v_1|_1 = 0 \\ \frac{\text{ppcm}\,(|v_1|_1, |v_2|)}{|v_1|_1} \times |v_1| & \text{sinon} \end{cases}$$

Une fois ces tailles calculées, il suffit d'appliquer la définition de l'opérateur pour calculer la valeur des éléments du préfixe et de la partie périodique de $u_3(v_3) = p_1 \ \texttt{on}\ p_2$:

$$\forall i,\ 1 \leq i \leq |u_3|,\ u_3[i] = \begin{cases} \texttt{0} & \text{si } p_1[i] = \texttt{0} \\ p_2[\mathcal{O}_{p_1}(i)] & \text{si } p_1[i] = \texttt{1} \end{cases}$$

$$\forall i,\ 1 \leq i \leq |v_3|,\ v_3[i] = \begin{cases} \texttt{0} & \text{si } p_1[|u_3| + i] = \texttt{0} \\ p_2[\mathcal{O}_{p_1}(|u_3| + i)] & \text{si } p_1[|u_3| + i] = \texttt{1} \end{cases}$$

Notons que si $p_1[i] = \texttt{1}$, $\mathcal{O}_{p_1}(i) \geq 1$ donc $p_2[\mathcal{O}_{p_1}(i)]$ est bien défini.

En remarquant que les 1 du résultat proviennent des 1 de p_2, on peut calculer que le nombre de 1 dans v_3 est $|v_3|_1 = \frac{\text{ppcm}\ (|v_1|_1, |v_2|)}{|v_2|} \times |v_2|_1$.

Démonstration :

Soit $p_3 = u_3(v_3)$.
Par définition de l'opérateur on, $p_3[i] = \texttt{0}$ si $p_1[i] = \texttt{0}$ et $p_3[i] = p_2[\mathcal{O}_{p_1}(i)]$ sinon.
Plusieurs cas de figure se présentent :

Cas où $i \leq |u_1|$ **:** $p_3[i] = \begin{cases} \texttt{0} & \text{si } u_1[i] = \texttt{0} \\ p_2[\mathcal{O}_{u_1}(i)] & \text{si } u_1[i] = \texttt{1} \end{cases}$

Deux sous-cas se présentent si $u_1[i] = \texttt{1}$:

Cas où $\mathcal{O}_{u_1}(i) \leq |u_2|$ **:** $p_3[i] = u_2[\mathcal{O}_{u_1}(i)]$

Cas où $\mathcal{O}_{u_1}(i) > |u_2|$ **:** $p_3[i] = v_2[1 + (\mathcal{O}_{u_1}(i) - |u_2| - 1) \mod |v_2|]$

Cas où $i > |u_1|$ **:** $p_3[i] = \begin{cases} \texttt{0} & \text{si } v_1[1 + (i - |u_1| - 1) \mod |v_1|] = \texttt{0} \\ p_2[\mathcal{O}_{p_1}(i)] & \text{si } v_1[1 + (i - |u_1| - 1) \mod |v_1|] = \texttt{1} \end{cases}$

Deux sous-cas se présentent si $v_1[1 + (i - |u_1| - 1) \mod |v_1|] = \texttt{1}$:

Cas où $\mathcal{O}_{p_1}(i) \leq |u_2|$ **:** $p_3[i] = u_2[\mathcal{O}_{p_1}(i)]$

Cas où $\mathcal{O}_{p_1}(i) > |u_2|$ **:** $p_3[i] = v_2[1 + (\mathcal{O}_{p_1}(i) - |u_2| - 1) \mod |v_2|]$

Donc pour tout $i > |u_1|$ tel que $\mathcal{O}_{p_1}(i) > |u_2|$, on a :

$$p_3[i] = \begin{cases} \texttt{0} & \text{si } v_1[1 + (i - |u_1| - 1) \mod |v_1|] = \texttt{0} \\ v_2[1 + (\mathcal{O}_{p_1}(i) - |u_2| - 1) \mod |v_2|] & \text{si } v_1[1 + (i - |u_1| - 1) \mod |v_1|] = \texttt{1} \end{cases}$$

On peut donc en déduire l'indice à partir duquel commence le comportement périodique selon que u_2 est de taille nulle ou pas :

Cas où $|u_2| = 0$ **:** pour tout i, $\mathcal{O}_{p_1}(i) > |u_2|$. Les formules ci-dessus sont donc vérifiées pour tout $i > |u_1|$. Pour $i = 1$ à $|u_1|$, on est donc en phase d'initialisation.

Cas où $|u_2| > 0$ **:** comme $\mathcal{O}_{p_1}(i) > |u_2| \Leftrightarrow i \geq \mathcal{I}_{p_1}(|u_2| + 1)$ (voir la remarque 3.3) et comme d'autre part
$\forall i \geq 1,\ \mathcal{I}_{p_1}(|u_2|) < i < \mathcal{I}_{p_1}(|u_2| + 1),\ p_1[i] = \texttt{0}$,
on peut en déduire : $\forall i > \max(|u_1|, \mathcal{I}_{p_1}(|u_2|))$,
$p_3[i] = \begin{cases} \texttt{0} & \text{si } v_1[1 + (i - |u_1| - 1) \mod |v_1|] = \texttt{0} \\ v_2[1 + (\mathcal{O}_{p_1}(i) - |u_2| - 1) \mod |v_2|] & \text{si } v_1[1 + (i - |u_1| - 1) \mod |v_1|] = \texttt{1} \end{cases}$
Pour $i = 1$ à $\max(|u_1|, \mathcal{I}_{p_1}(|u_2|))$, on est donc en phase d'initialisation. Par conséquent, $|u_3| = \max(|u_1|, \mathcal{I}_{p_1}(|u_2|))$.

Cherchons maintenant la longueur du comportement périodique, c'est-à-dire le plus petit $x \in \mathbb{N}^*$ tel que $\forall i > |u_3|,\ p_3[i+x] = p_3[i]$.
$\forall i > |u_3|,\ p_3[i+x] = p_3[i] \Leftrightarrow \forall i > |u_3|,$

$$v_1[1 + (i + x - |u_1| - 1) \mod |v_1|] = v_1[1 + (i - |u_1| - 1) \mod |v_1|] \quad (1)$$
$$\wedge$$
$$v_2[1 + (\mathcal{O}_{p_1}(i+x) - |u_2| - 1) \mod |v_2|] = v_2[1 + (\mathcal{O}_{p_1}(i) - |u_2| - 1) \mod |v_2|] \quad (2)$$

Comme on n'interprète pas les éléments de v_1,

$$(1) \Leftrightarrow \forall i > |u_3|,\ 1 + (i + x - |u_1| - 1) \mod |v_1| = 1 + (i - |u_1| - 1) \mod |v_1|$$
$$\Leftrightarrow x = x' \times |v_1| \text{ avec } x' \in \mathbb{N}^*$$

Comme on n'interprète pas les éléments de v_2,

$$(2) \Leftrightarrow \forall i > |u_3|,\ 1 + (\mathcal{O}_{p_1}(i+x) - |u_2| - 1) \mod |v_2| = 1 + (\mathcal{O}_{p_1}(i) - |u_2| - 1) \mod |v_2|$$
$$\Leftrightarrow \forall i > |u_3|,\ \mathcal{O}_{p_1}(i+x) \mod |v_2| = \mathcal{O}_{p_1}(i) \mod |v_2|$$

Donc $(1) \wedge (2) \Leftrightarrow \forall i > |u_3|,\ \mathcal{O}_{p_1}(i + x' \times |v_1|) \mod |v_2| = \mathcal{O}_{p_1}(i) \mod |v_2|$
$\Leftrightarrow (\mathcal{O}_{p_1}(i) + x' \times |v_1|_1) \mod |v_2| = \mathcal{O}_{p_1}(i) \mod |v_2|$
$\Leftrightarrow (x' \times |v_1|_1) \mod |v_2| = 0$

Le plus petit x' qui convienne est $\frac{\text{ppcm}\,(|v_1|_1, |v_2|)}{|v_1|_1}$.
Donc $|v_3| = x = \frac{\text{ppcm}\,(|v_1|_1, |v_2|)}{|v_1|_1} \times |v_1|$.

Calculons enfin le nombre de `1` présents dans le comportement périodique.
$|v_3|_1 = \mathcal{O}_{p_3}(|u_3| + |v_3|) - \mathcal{O}_{p_3}(|u_3|) = \mathcal{O}_{p_2}(\mathcal{O}_{p_1}(|u_3| + |v_3|)) - \mathcal{O}_{p_2}(\mathcal{O}_{p_1}(|u_3|))$.
On a montré dans la première partie de cette preuve que $|v_3| = \frac{\text{ppcm}\,(|v_1|_1, |v_2|)}{|v_1|_1} \times |v_1|$.
Par conséquent, $\mathcal{O}_{p_1}(|u_3| + |v_3|) = \mathcal{O}_{p_1}(|u_3| + \frac{\text{ppcm}\,(|v_1|_1, |v_2|)}{|v_1|_1} \times |v_1|)$.
Par la remarque qui suit le lemme 4.4, on peut en déduire que
$\mathcal{O}_{p_1}(|u_3| + |v_3|) = \mathcal{O}_{p_1}(|u_3|) + \frac{\text{ppcm}\,(|v_1|_1, |v_2|)}{|v_1|_1} \times |v_1|_1 = \mathcal{O}_{p_1}(|u_3|) + \text{ ppcm}\,(|v_1|_1, |v_2|)$.
Par conséquent, $\mathcal{O}_{p_2}(\mathcal{O}_{p_1}(|u_3| + |v_3|)) = \mathcal{O}_{p_2}(\mathcal{O}_{p_1}(|u_3|) + \text{ ppcm}\,(|v_1|_1, |v_2|)) = \mathcal{O}_{p_2}(\mathcal{O}_{p_1}(|u_3|) + \frac{\text{ppcm}\,(|v_1|_1, |v_2|)}{|v_2|} \times |v_2|)$.
Par la remarque qui suit le lemme 4.4, on peut en déduire que
$\mathcal{O}_{p_2}(\mathcal{O}_{p_1}(|u_3| + |v_3|)) = \mathcal{O}_{p_2}(\mathcal{O}_{p_1}(|u_3|)) + \frac{\text{ppcm}\,(|v_1|_1, |v_2|)}{|v_2|} \times |v_2|_1$.
Comme $|v_3|_1 = \mathcal{O}_{p_3}(|u_3| + |v_3|) - \mathcal{O}_{p_3}(|u_3|)$, on peut en conclure que
$|v_3|_1 = \mathcal{O}_{p_2}(\mathcal{O}_{p_1}(|u_3| + |v_3|)) - \mathcal{O}_{p_2}(\mathcal{O}_{p_1}(|u_3|)) = \frac{\text{ppcm}\,(|v_1|_1, |v_2|)}{|v_2|} \times |v_2|_1$. □

La taille de la partie périodique du résultat d'un `on` est donc de l'ordre du plus petit commun multiple des tailles de ses opérandes, c'est-à-dire beaucoup plus longue que celles-ci. Cette croissance rapide posera problème lors de l'application d'opérations dont la complexité dépend de la longueur du comportement périodique des opérandes, comme le calcul de l'opération `on` elle-même, le test d'égalité, le calcul du rythme commun présenté dans la section suivante, le test de précédence présenté dans la section 4.4 et surtout dans la résolution des contraintes d'égalité et d'adaptabilité sur les rythmes présentée dans le chapitre 5.

4.3 Relation $\leqslant$ et calcul du rythme commun sur $\mathbb{Q}_2$

Si p_1 et p_2 sont des mots périodiques, alors tous les mots permettant d'obtenir p_1 par ralentissement de p_2 (c'est-à-dire tous les mots de l'ensemble $S(p_1, p_2)$) sont périodiques. [1]

[1] Mis à part pour $S(\texttt{(0)}, \texttt{(0)})$ qui contient tous les mots binaires infinis car `(0)` est absorbant.

Dans le cas où la partie périodique de p_1 ou de p_2 vaut $\mathtt{0}$, le calcul de $S(p_1, p_2)$ suivant la formule donnée dans la proposition 3.5 (page 60) termine en un nombre fini d'étapes. Dans le cas contraire, on a besoin de connaître le nombre d'éléments de p_1 et p_2 à parcourir pour construire le préfixe des mots de $S(p_1, p_2)$, et le nombre d'éléments supplémentaires à parcourir afin de construire leur comportement périodique.

Calcul de $S(p_1, p_2)$. Quand v_1 et v_2 ne valent pas $\mathtt{0}$, on a :

$\forall p \in S(u_1\mathtt{(}v_1\mathtt{)}, u_2\mathtt{(}v_2\mathtt{)}), p = u\mathtt{(}v\mathtt{)}$ avec

$$\begin{array}{rclcrcl} |u| & = & \mathcal{I}_{p_1}(j+1) - 1 & & |v| & = & k_1 \times |v_1| \\ |u|_{\mathtt{1}} & = & \mathcal{I}_{p_2}(j+1) - 1 & & |v|_{\mathtt{1}} & = & k_2 \times |v_2| \end{array}$$

et $j = \max(|u_1|_{\mathtt{1}}, |u_2|_{\mathtt{1}})$, $j' = \text{ ppcm }(|v_1|_{\mathtt{1}}, |v_2|_{\mathtt{1}})$, $k_1 = \frac{j'}{|v_1|_{\mathtt{1}}}$, $k_2 = \frac{j'}{|v_2|_{\mathtt{1}}}$.

Une fois ces tailles calculées, il suffit d'appliquer l'algorithme de la proposition 3.5 pour obtenir les éléments de p.

Intuition de la démonstration :

Si $p_1 = u_1\mathtt{(}v_1\mathtt{)}$ et $p_2 = u_2\mathtt{(}v_2\mathtt{)}$, le nombre de $\mathtt{1}$ à parcourir dans p_1 et p_2 pour construire le préfixe des mots de $S(p_1, p_2)$ est $j = \max(|u_1|_{\mathtt{1}}, |u_2|_{\mathtt{1}})$. Comme on veut aussi parcourir la série de $\mathtt{0}$ qui suit ce $j^{ième}$ $\mathtt{1}$, le nombre d'éléments à parcourir dans p_1 et p_2 est donc l'indice précédent celui de leur $(j+1)^{ième}$ $\mathtt{1}$.

Le nombre de $\mathtt{1}$ supplémentaires à parcourir pour construire la partie périodique des mots de $S(p_1, p_2)$ est $j' = \text{ ppcm }(|v_1|_{\mathtt{1}}, |v_2|_{\mathtt{1}})$. Si $j' = k_1 \times |v_1|_{\mathtt{1}} = k_2 \times |v_2|_{\mathtt{1}}$, le nombre d'éléments à parcourir dans p_1 (resp. p_2) est donc $k_1 \times |v_1|$ (resp. $k_2 \times |v_2|$).

Comme le nombre d'éléments des mots de $S(p_1, p_2)$ correspond au nombre d'éléments parcourus dans p_1, on a $|u| = \mathcal{I}_{p_1}(j+1) - 1$ et $|v| = k_1 \times |v_1|$. Comme leur nombre de $\mathtt{1}$ correspond au nombre d'éléments parcourus dans p_2, on a $|u|_{\mathtt{1}} = \mathcal{I}_{p_2}(j+1) - 1$ et $|v|_{\mathtt{1}} = k_2 \times |v_2|$. □

De la même manière, le rythme commun le plus rapide de deux mots périodiques est un mot périodique. Si $v_1 = \mathtt{0}$ ou $v_2 = \mathtt{0}$, le rythme commun de $p_1 = u_1\mathtt{(}v_1\mathtt{)}$ et $p_2 = u_2\mathtt{(}v_2\mathtt{)}$ (c'est-à-dire $\inf(p_1, p_2)$) est calculé en un nombre fini d'étapes suivant la formule donnée dans la proposition 3.6 page 62. Si ce n'est pas les cas, il faut connaître le nombre d'éléments de p_1 et p_2 à parcourir pour construire son préfixe, et le nombre d'éléments supplémentaires à parcourir dans p_1 et p_2 pour construire son comportement périodique.

Calcul de $\inf(p_1, p_2)$. Quand v_1 et v_2 ne valent pas $\mathtt{0}$, on a :

$$\inf(u_1\mathtt{(}v_1\mathtt{)}, u_2\mathtt{(}v_2\mathtt{)}) = u\mathtt{(}v\mathtt{)} \text{ avec } |u|_{\mathtt{1}} = \max(|u_1|_{\mathtt{1}}, |u_2|_{\mathtt{1}}) \text{ et } |v|_{\mathtt{1}} = \text{ ppcm }(|v_1|_{\mathtt{1}}, |v_2|_{\mathtt{1}})$$

Par conséquent, pour obtenir le préfixe du rythme commun, il faut appliquer la formule de l'infimum pour la relation de ralentissement (voir proposition 3.6 page 62) jusqu'à l'indice précédent le $(|u|_{\mathtt{1}} + 1)^{ième}\mathtt{1}$ des opérandes, et pour obtenir la partie périodique, continuer à appliquer la formule sur les $\frac{|v|_{\mathtt{1}}}{|v_1|_{\mathtt{1}}} \times |v_1|$ éléments suivants de p_1 et les $\frac{|v|_{\mathtt{1}}}{|v_2|_{\mathtt{1}}} \times |v_2|$ éléments suivants de p_2.

4.4 Relations $\preceq$, $\bowtie$ et $<$: sur $\mathbb{Q}_2$

Nous montrons dans cette section comment vérifier les relations de précédence et de synchronisabilité sur des mots binaires périodiques, donc comment vérifier la relation d'adaptabilité.

Relation de synchronisabilité sur $\mathbb{Q}_2$. Deux mots binaires périodiques sont synchronisables si et seulement si leur partie périodique comporte la même proportion de 1.

Proposition 4.2 (test de synchronisabilité sur $\mathbb{Q}_2$). Soient $p_1 = u_1(v_1)$ et $p_2 = u_2(v_2)$.

$$p_1 \bowtie p_2 \Leftrightarrow \frac{|v_1|_1}{|v_1|} = \frac{|v_2|_1}{|v_2|}$$

Démonstration :

Soient $p_1 = u_1(v_1)$ et $p_2 = u_2(v_2)$. Grâce à la remarque 4.1, on peut toujours mettre p_1 sous la forme $u'_1(v'_1)$ et p_2 sous la forme $u'_2(v'_2)$ telles que $|u'_1| = |u'_2| = \max(|u_1|, |u_2|)$ et $|v'_1| = |v'_2| =$ ppcm $(|v_1|, |v_2|)$. On a alors $\frac{|v_1|_1}{|v_1|} = \frac{|v'_1|_1}{|v'_1|}$ et $\frac{|v_2|_1}{|v_2|} = \frac{|v'_2|_1}{|v'_2|}$.
On supposera donc sans perte de généralité que $|u_1| = |u_2|$ et $|v_1| = |v_2|$.
D'après le lemme 4.4, pour tout $i_x = |u_1| + x \times |v_1|$ $(= |u_2| + x \times |v_2|)$ avec $x \in \mathbb{N}$, on a :

$$\begin{aligned} \mathcal{O}_{p_1}(i_x) &= |u_1|_1 + x \times |v_1|_1 \\ \mathcal{O}_{p_2}(i_x) &= |u_2|_1 + x \times |v_2|_1 \end{aligned}$$

Montrons que $\frac{|v_1|_1}{|v_1|} \neq \frac{|v_2|_1}{|v_2|} \Rightarrow p_1 \not\bowtie p_2$

On a : $\mathcal{O}_{p_2}(i_x) - \mathcal{O}_{p_1}(i_x) = |u_2|_1 - |u_1|_1 + x \times (|v_2|_1 - |v_1|_1)$.
Si $\frac{|v_1|_1}{|v_1|} < \frac{|v_2|_1}{|v_2|}$ alors $|v_1|_1 < |v_2|_1$ et $\lim_{x\to+\infty}(\mathcal{O}_{p_2}(i_x) - \mathcal{O}_{p_1}(i_x)) = +\infty$.
Donc $\nexists b_2,\ \forall i \geq 0,\ \mathcal{O}_{p_2}(i) - \mathcal{O}_{p_1}(i) \leq b_2$.
Si $\frac{|v_1|_1}{|v_1|} > \frac{|v_2|_1}{|v_2|}$ alors $|v_1|_1 > |v_2|_1$ et $\lim_{x\to+\infty}(\mathcal{O}_{p_2}(i_x) - \mathcal{O}_{p_1}(i_x)) = -\infty$.
Donc $\nexists b_1,\ \forall i \geq 0,\ b_1 \leq \mathcal{O}_{p_2}(i) - \mathcal{O}_{p_1}(i)$.

Montrons que $\frac{|v_1|_1}{|v_1|} = \frac{|v_2|_1}{|v_2|} \Rightarrow p_1 \bowtie p_2$

Comme $\mathcal{O}$ est croissante, pour tout i tel que $i_x \leq i \leq i_{x+1}$ avec $x \in \mathbb{N}$,
$\mathcal{O}_{p_1}(i_x) \leq \mathcal{O}_{p_1}(i) \leq \mathcal{O}_{p_1}(i_{x+1})$ et $\mathcal{O}_{p_2}(i_x) \leq \mathcal{O}_{p_2}(i) \leq \mathcal{O}_{p_2}(i_{x+1})$.
Donc $\forall i,\ i_x \leq i \leq i_{x+1}$, $\mathcal{O}_{p_2}(i_x) - \mathcal{O}_{p_1}(i_{x+1}) \leq \mathcal{O}_{p_2}(i) - \mathcal{O}_{p_1}(i) \leq \mathcal{O}_{p_2}(i_{x+1}) - \mathcal{O}_{p_1}(i_x)$.
C'est-à-dire que $\forall i,\ i_x \leq i \leq i_{x+1}$,
$|u_2|_1 + x \times |v_2|_1 - (|u_1|_1 + (x+1) \times |v_1|_1) \leq \mathcal{O}_{p_2}(i) - \mathcal{O}_{p_1}(i) \leq$
$|u_2|_1 + (x+1) \times |v_2|_1 - (|u_1|_1 + x \times |v_1|_1)$.
Donc $\forall i,\ i_x \leq i \leq i_{x+1}$,
$|u_2|_1 - |u_1|_1 - |v_1|_1 + x \times (|v_2|_1 - |v_1|_1) \leq \mathcal{O}_{p_2}(i) - \mathcal{O}_{p_1}(i) \leq$
$|u_2|_1 - |u_1|_1 + |v_2|_1 + x \times (|v_2|_1 - |v_1|_1)$.
Si $\frac{|v_1|_1}{|v_1|} = \frac{|v_2|_1}{|v_2|}$ alors $|v_1|_1 = |v_2|_1$ et $\forall i,\ i_x \leq i \leq i_{x+1}$,
$|u_2|_1 - |u_1|_1 - |v_1|_1 \leq \mathcal{O}_{p_2}(i) - \mathcal{O}_{p_1}(i) \leq |u_2|_1 - |u_1|_1 + |v_2|_1$.
Par conséquent, $\exists b_1, b_2,\ \forall i \geq |u_1|,\ b_1 \leq \mathcal{O}_{p_2}(i) - \mathcal{O}_{p_1}(i) \leq b_2$.
Comme d'autre part $\forall i \leq |u_1|,\ -|u_1|_1 \leq \mathcal{O}_{p_2}(i) - \mathcal{O}_{p_1}(i) \leq |u_2|_1$, on peut en conclure que $\exists b_1, b_2,\ \forall i \geq 1,\ b_1 \leq \mathcal{O}_{p_2}(i) - \mathcal{O}_{p_1}(i) \leq b_2$ et donc que $p_1 \bowtie p_2$.

□

Par exemple, dans la figure 3.3 (page 64), $w_1 = \texttt{(11010)}$ et $w_2 = \texttt{0(00111)}$ comportent la même proportion de `1` ($\frac{3}{5}$), donc ils sont synchronisables. On peut vérifier dans le tableau ci-dessous que conformément à la définition 3.9, la différence entre leurs fonctions de cumul est bornée :

w_1	1	**1**	0	1	0	.	1	**1**	0	1	0	.	1	**1**	0	1	...
w_2	0	.	**0**	0	1	1	1	.	**0**	0	1	1	1	.	**0**	0	1	...
$\mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i)$	1	**2**	2	2	1	1	**2**	2	2	1	1	**2**	2	2	...			

(4.1)

Par contre, seulement $\frac{1}{5}$ des éléments de $w = \texttt{(00100)}$ sont des `1`, donc w n'est pas synchronisable avec w_1 et w_2. Par exemple, la différence entre la fonction de cumul de w_1 et celle de w croît à l'infini :

w_1	**1**	1	0	1	0	.	**1**	1	0	1	0	.	**1**	1	0	1	...
w	**0**	0	1	0	0	.	**0**	0	1	0	0	.	**0**	0	1	0	...
$\mathcal{O}_{w_1}(i) - \mathcal{O}_{w}(i)$	**1**	2	1	2	2	**3**	4	3	4	4	**5**	6	5	6	...		

Relation de précédence sur $\mathbb{Q}_2$. Pour tester la relation de précédence sur des mots périodiques synchronisables à l'aide de l'indice des `1`, il suffit de vérifier la condition de la définition 3.8 jusqu'à ce que le comportement périodique "commun" soit atteint.

Proposition 4.3 (test de précédence sur $\mathbb{Q}_2$ - indices des `1`).
Soient $p_1 = u_1\texttt{(}v_1\texttt{)}$ et $p_2 = u_2\texttt{(}v_2\texttt{)}$ tels que $p_1 \bowtie p_2$.

$$p_1 \preceq p_2 \Leftrightarrow \forall j,\ 1 \leq j \leq \max(|u_1|_1, |u_2|_1) + \text{ ppcm } (|v_1|_1, |v_2|_1),\ \mathcal{I}_{p_1}(j) \leq \mathcal{I}_{p_2}(j)$$

Par exemple, pour s'assurer que la relation de précédence est vérifiée entre $w_1 = \texttt{(11010)}$ et $w_2 = \texttt{0(00111)}$ (voir le tableau (4.1) et la figure 3.3), il suffit de comparer les indices des trois premiers `1` de w_1 et w_2.

Démonstration :

$\Rightarrow$) Trivial par définition de $\preceq$.
$\Leftarrow$) Montrons que
$(\forall j,\ 1 \leq j \leq \max(|u_1|_1, |u_2|_1) + \text{ ppcm } (|v_1|_1, |v_2|_1),\ \mathcal{I}_{p_1}(j) \leq \mathcal{I}_{p_2}(j))$
$\Rightarrow (\forall j \geq 1,\ \mathcal{I}_{p_1}(j) \leq \mathcal{I}_{p_2}(j))$.

Le cas où $|v_1|_1 = |v_2|_1 = 0$ est trivialement vérifié car dans ce cas où le mot comporte un nombre fini de `1`, le test va être effectué sur l'intégralité des `1` du mot.

On se place donc dans le cas $|v_1|_1 \neq 0$ et $|v_2|_1 \neq 0$ (car $p_1 \bowtie p_2$).

Soit $n = \max(|u_1|_1, |u_2|_1)$. Par la remarque 4.1, et comme $|v_1|_1 > 0$, on peut mettre p_1 sous la forme $u'_1\texttt{(}v'_1\texttt{)}$ avec $|u'_1|_1 = n$, $|v'_1| = |v_1|$ et $|v'_1|_1 = |v_1|_1$.
Soit $j > n + \text{ ppcm } (|v_1|_1, |v_2|_1)$.
Par le lemme 4.3, on a :

$$\begin{aligned}
\mathcal{I}_{p_1}(j) &= |u'_1| + \left\lfloor \tfrac{j-n-1}{|v_1|_1} \right\rfloor \times |v_1| + \mathcal{I}_{v'_1}(1 + (j-n-1) \mod |v_1|_1) \\
&= |u'_1| + \left\lfloor \frac{\left\lfloor \frac{j-n-1}{\text{ppcm}\,(|v_1|_1,|v_2|_1)} \right\rfloor \times \text{ppcm}\,(|v_1|_1,|v_2|_1) + (j-n-1) \mod \text{ppcm}\,(|v_1|_1,|v_2|_1)}{|v_1|_1} \right\rfloor \times |v_1| \\
&\quad + \mathcal{I}_{v'_1}(1 + (j-n-1) \mod |v_1|_1) \\
&= |u'_1| + \left(\left\lfloor \tfrac{j-n-1}{\text{ppcm}\,(|v_1|_1,|v_2|_1)} \right\rfloor \times \tfrac{\text{ppcm}\,(|v_1|_1,|v_2|_1)}{|v_1|_1} + \left\lfloor \tfrac{(j-n-1) \mod \text{ppcm}\,(|v_1|_1,|v_2|_1)}{|v_1|_1} \right\rfloor \right) \times |v_1| \\
&\quad + \mathcal{I}_{v'_1}(1 + (j-n-1) \mod |v_1|_1 + 1) \\
&= |u'_1| + \left\lfloor \tfrac{(j-n-1) \mod \text{ppcm}\,(|v_1|_1,|v_2|_1)}{|v_1|_1} \right\rfloor \times |v_1| + \mathcal{I}_{v'_1}(1 + (j-n-1) \mod |v_1|_1) \\
&\quad + \left\lfloor \tfrac{j-n-1}{\text{ppcm}\,(|v_1|_1,|v_2|_1)} \right\rfloor \times \tfrac{\text{ppcm}\,(|v_1|_1,|v_2|_1)}{|v_1|_1} \times |v_1| \\
&= \mathcal{I}_{p_1}(((j-n-1) \mod \text{ppcm}\,(|v_1|_1,|v_2|_1)) + n + 1) \\
&\quad + \left\lfloor \tfrac{j-n-1}{\text{ppcm}\,(|v_1|_1,|v_2|_1)} \right\rfloor \times \tfrac{\text{ppcm}\,(|v_1|_1,|v_2|_1)}{|v_1|_1} \times |v_1|
\end{aligned}$$

De la même manière, on met p_2 sous la forme $u'_2(v'_2)$ avec
$|u'_2|_1 = n$, $|v'_2| = |v_2|$ et $|v'_2|_1 = |v_2|_1$.
Soit $j > n + \text{ppcm}\,(|v_1|_1, |v_2|_1)$.

$$\begin{aligned}
\mathcal{I}_{p_2}(j) &= \mathcal{I}_{p_2}(((j-n-1) \mod \text{ppcm}\,(|v_1|_1,|v_2|_1)) + n + 1) \\
&\quad + \left\lfloor \tfrac{j-n-1}{\text{ppcm}\,(|v_1|_1,|v_2|_1)} \right\rfloor \times \tfrac{\text{ppcm}\,(|v_1|_1,|v_2|_1)}{|v_2|_1} \times |v_2|
\end{aligned}$$

Comme $((j-n-1) \mod \text{ppcm}\,(|v_1|_1,|v_2|_1)) + n + 1 \leq n + \text{ppcm}\,(|v_1|_1,|v_2|_1)$,
par hypothèse on a :
$\mathcal{I}_{p_1}(((j-n-1) \mod \text{ppcm}\,(|v_1|_1,|v_2|_1)) + n + 1) \leq$
$\mathcal{I}_{p_2}(((j-n-1) \mod \text{ppcm}\,(|v_1|_1,|v_2|_1)) + n + 1)$.

Comme $p_1 \bowtie p_2$, par la proposition 4.2, on a
$\left\lfloor \frac{j-n-1}{\text{ppcm}\,(|v_1|_1,|v_2|_1)} \right\rfloor \times \frac{\text{ppcm}\,(|v_1|_1,|v_2|_1)}{|v_1|_1} \times |v_1| = \left\lfloor \frac{j-n-1}{\text{ppcm}\,(|v_1|_1,|v_2|_1)} \right\rfloor \times \frac{\text{ppcm}\,(|v_1|_1,|v_2|_1)}{|v_2|_1} \times |v_2|$.

On peut conclure que $\forall j > n + \text{ppcm}\,(|v_1|_1,\ |v_2|_1),\ \mathcal{I}_{p_1}(j) \leq \mathcal{I}_{p_2}(j)$. □

La borne donnée correspond à ajuster les préfixes et les parties périodiques au même nombre de `1`, et effectuer un parcours des `1` jusqu'à la fin du premier cycle de la partie périodique.

Quand deux mots p_1 et p_2 sont synchronisables, il existe un instant à partir duquel toutes les valeurs possibles pour la différence entre $\mathcal{O}_{p_1}(i)$ et $\mathcal{O}_{p_2}(i)$ ont été rencontrées.

Lemme 4.7. Soient $p_1 = u_1(v_1)$ et $p_2 = u_2(v_2)$ tels que $p_1 \bowtie p_2$.
$\forall i \geq \max(|u_1|, |u_2|) + \text{ppcm}\,(|v_1|, |v_2|),\ \exists i' < \max(|u_1|, |u_2|) + \text{ppcm}\,(|v_1|, |v_2|)$,
$\mathcal{O}_{p_1}(i) - \mathcal{O}_{p_2}(i) = \mathcal{O}_{p_1}(i') - \mathcal{O}_{p_2}(i')$

Démonstration :

$\forall i \geq |u_1|,\ \mathcal{O}_{p_1}(i + x \times |v_1|) = \mathcal{O}_{p_1}(i) + x \times |v_1|_1$
et $\forall i \geq |u_2|,\ \mathcal{O}_{p_2}(i + x \times |v_2|) = \mathcal{O}_{p_2}(i) + x \times |v_2|_1$.
Donc $\forall i \geq \max(|u_1|, |u_2|),\ \mathcal{O}_{p_1}(i + x \times \text{ ppcm } (|v_1|, |v_2|)) = \mathcal{O}_{p_1}(i) + x \times \frac{\text{ppcm } (|v_1|, |v_2|)}{|v_1|} \times |v_1|_1$
et $\mathcal{O}_{p_2}(i + x \times \text{ ppcm } (|v_1|, |v_2|)) = \mathcal{O}_{p_2}(i) + x \times \frac{\text{ppcm } (|v_1|, |v_2|)}{|v_2|} \times |v_2|_1$.
Comme $p_1 \bowtie p_2$, on a $\frac{|v_1|_1}{|v_1|} = \frac{|v_2|_1}{|v_2|}$.
Par conséquent, $\forall i \geq \max(|u_1|, |u_2|)$,
$\mathcal{O}_{p_1}(i + x \times \text{ ppcm } (|v_1|, |v_2|)) - \mathcal{O}_{p_2}(i + x \times \text{ ppcm } (|v_1|, |v_2|)) = \mathcal{O}_{p_1}(i) - \mathcal{O}_{p_2}(i)$.
Donc $\forall i \geq \max(|u_1|, |u_2|) + \text{ ppcm } (|p_1|, |p_2|),\ \mathcal{O}_{p_1}(i) - \mathcal{O}_{p_2}(i) = \mathcal{O}_{p_1}(i') - \mathcal{O}_{p_2}(i')$
avec $i' = \max(|u_1|, |u_2|) + (i - \max(|u_1|, |u_2|)) \mod (\text{ ppcm } (|p_1|, |p_2|))$.
On a bien $i' < \max(|u_1|, |u_2|) + \text{ ppcm } (|p_1|, |p_2|)$. □

On peut donc aussi tester la relation de précédence sur les mots périodiques à l'aide des fonctions de cumul, en vérifiant la condition de la proposition 3.7 jusqu'à ce le comportement périodique "commun" soit atteint. Le calcul de cet instant où le comportement commun est atteint suit le même principe que celui de la proposition 4.3. Il s'agit ici d'ajuster les préfixes et parties périodiques à la même taille et de parcourir les mots jusqu'à la fin du premier cycle de leur partie périodique.

Proposition 4.4 (test de précédence sur $\mathbb{Q}_2$ - fonctions de cumul).
Soient $p_1 = u_1(v_1)$ et $p_2 = u_2(v_2)$ tels que $p_1 \bowtie p_2$.

$$p_1 \preceq p_2 \Leftrightarrow \forall i,\ 1 \leq i \leq \max(|u_1|, |u_2|) + \text{ ppcm } (|v_1|, |v_2|),\ \mathcal{O}_{p_1}(i) \geq \mathcal{O}_{p_2}(i)$$

Démonstration :

⇒) Trivial par la proposition 3.7

⇐) Par le lemme 4.7, on a $\forall i \geq 1,\ \mathcal{O}_{p_1}(i) - \mathcal{O}_{p_2}(i) = \mathcal{O}_{p_1}(i') - \mathcal{O}_{p_2}(i')$ avec $i' < \max(|u_1|, |u_2|) + \text{ ppcm } (|v_1|, |v_2|)$. Par hypothèse, on a $\mathcal{O}_{p_1}(i') \geq \mathcal{O}_{p_2}(i')$, donc on peut conclure que $\forall i \geq 1,\ \mathcal{O}_{p_1}(i) \geq \mathcal{O}_{p_2}(i)$, et donc que $p_1 \preceq p_2$.

□

Sur l'exemple du tableau (4.1) page 86, il suffit de vérifier que la différence entre les fonctions de cumul est positive jusqu'à l'instant 6, car la suite des valeurs prises par cette différence est cyclique.

On peut enfin noter que tout comme pour les opérations *not* et *on* présentés en section 4.2, on sait calculer le résultat des opérations de supremum et d'infimum pour la relation de précédence.

Lemme 4.8 (calcul de $p_1 \sqcup p_2$ et $p_1 \sqcap p_2$).
Soient $p_1 = u_1(v_1)$ et $p_2 = u_2(v_2)$ avec $p_1 \bowtie p_2$.
Alors $p_1 \sqcup p_2 = u_3(v_3)$ et $p_1 \sqcap p_2 = u_4(v_4)$ avec :

$$|u_3| = |u_4| = \max(|u_1|, |u_2|) \qquad |v_3| = |v_4| = \text{ ppcm } (|v_1|, |v_2|)$$

Les valeurs de u_3 et v_3 (resp. u_4 et v_4) sont calculées en appliquant la formule du supremum (resp. de l'infimum) sur les mots (lemme 3.3) :

$$\begin{array}{lcl}
\forall i,\ 1 \leq i \leq |u_3|,\ \mathcal{O}_{u_3}(i) & = & \min(\mathcal{O}_{p_1}(i), \mathcal{O}_{p_2}(i)) \\
\forall i,\ 1 \leq i \leq |v_3|,\ \mathcal{O}_{v_3}(i) & = & \min(\mathcal{O}_{p_1}(|u_3|+i), \mathcal{O}_{p_2}(|u_3|+i)) \\
\forall i,\ 1 \leq i \leq |u_4|,\ \mathcal{O}_{u_4}(i) & = & \max(\mathcal{O}_{p_1}(i), \mathcal{O}_{p_2}(i)) \\
\forall i,\ 1 \leq i \leq |v_4|,\ \mathcal{O}_{v_4}(i) & = & \max(\mathcal{O}_{p_1}(|u_4|+i), \mathcal{O}_{p_2}(|u_4|+i))
\end{array}$$

Démonstration :

Soit $p_3 = p_1 \sqcup p_2$. Par le lemme 3.4, $\forall i \geq 1,\ \mathcal{O}_{p_3}(i) = \min(\mathcal{O}_{p_1}(i), \mathcal{O}_{p_2}(i))$.
Or $\forall i \geq |u_1|,\ \mathcal{O}_{p_1}(i + |v_1|) = \mathcal{O}_{p_1}(i) + |v_1|_{\mathtt{1}}$ et $\forall i \geq |u_2|,\ \mathcal{O}_{p_2}(i + |v_2|) = \mathcal{O}_{p_2}(i) + |v_2|_{\mathtt{1}}$.
Donc $\forall i \geq \max(|u_1|, |u_2|),\ \mathcal{O}_{p_1}(i + \text{ ppcm } (|v_1|, |v_2|)) = \mathcal{O}_{p_1}(i) + \frac{\text{ppcm } (|v_1|,|v_2|)}{|v_1|} \times |v_1|_{\mathtt{1}}$
et $\mathcal{O}_{p_2}(i + \text{ ppcm } (|v_1|, |v_2|)) = \mathcal{O}_{p_2}(i) + \frac{\text{ppcm } (|v_1|,|v_2|)}{|v_2|} \times |v_2|_{\mathtt{1}}$.
Comme $p_1 \bowtie p_2$, on a $\frac{|v_1|_{\mathtt{1}}}{|v_1|} = \frac{|v_2|_{\mathtt{1}}}{|v_2|}$, donc $\forall i \geq \max(|u_1|, |v_1|)$,
$\min(\mathcal{O}_{p_1}(i + \text{ ppcm } (|v_1|, |v_2|)), \mathcal{O}_{p_2}(i + \text{ ppcm } (|v_1|, |v_2|)))$
$= \min(\mathcal{O}_{p_1}(i), \mathcal{O}_{p_2}(i)) + \frac{\text{ppcm } (|v_1|,|v_2|)}{|v_1|} \times |v_1|_{\mathtt{1}}$
et par conséquent $\mathcal{O}_{p_3}(i + \text{ ppcm } (|v_1|, |v_2|)) = \mathcal{O}_{p_3}(i) + \frac{\text{ppcm } (|v_1|,|v_2|)}{|v_1|} \times |v_1|_{\mathtt{1}}$.
Par le lemme 4.4, on peut conclure que $p_3 = u_3(v_3)$
avec $|u_3| = \max(|u_1|, |u_2|)$, $|v_3| = \text{ ppcm } (|v_1|, |v_2|)$ et $|v_3|_{\mathtt{1}} = \frac{\text{ppcm } (|v_1|,|v_2|)}{|v_1|} \times |v_1|_{\mathtt{1}}$.
Les valeur de $\mathcal{O}_{u_3}$ et $\mathcal{O}_{v_3}$ sont données par définition du supremum.
La preuve de la forme de l'infimum se fait de manière similaire. □

Nous avons fourni dans cette section des moyens de tester la relation de précédence (par les propositions 4.3 ou 4.4) et la relation de synchronisabilité (par la proposition 4.2). Quand on manipule des mots périodiques, on est donc en mesure de vérifier la relation d'adaptabilité. Nous montrons maintenant comment calculer la taille de buffer nécessaire à la communication entre deux horloges périodiques.

Calcul de la taille du buffer. On a vu dans le lemme 4.7 que quand deux mots sont synchronisables, il existe un indice à partir duquel toutes les valeurs possibles pour la différence entre leurs fonctions de cumul ont été rencontrées. Il suffit de calculer le maximum de cette différence jusqu'à cet indice afin de trouver la taille de buffer suffisante minimale :

Proposition 4.5 (taille du buffer). Soient $p_1 = u_1(v_1)$ et $p_2 = u_2(v_2)$ tels que $p_1 <: p_2$. La taille de buffer minimale pour communiquer de p_1 à p_2 est :

$$size(p_1, p_2) = \max_{1 \leq i < \max(|u_1|,|u_2|) \ + \ \text{ppcm } (|v_1|,|v_2|)} (\mathcal{O}_{p_1}(i) - \mathcal{O}_{p_2}(i))$$

Démonstration :

Nous avons vu en section 3.4 que $size(p_1, p_2) = \max_{i \in \mathbb{N}^*}(\mathcal{O}_{p_1}(i) - \mathcal{O}_{p_2}(i))$.
Soit $t = \max_{1 \leq i < \max(|u_1|,|u_2|) + \text{ ppcm } (|v_1|,|v_2|)}(\mathcal{O}_{p_1}(i) - \mathcal{O}_{p_2}(i))$.
Montrons que $\max_{i \in \mathbb{N}^*}(\mathcal{O}_{p_1}(i) - \mathcal{O}_{p_2}(i)) = t$,
Par le lemme 4.7, on a :
$\forall i \geq 1,\ \mathcal{O}_{p_1}(i) - \mathcal{O}_{p_2}(i) = \mathcal{O}_{p_1}(i') - \mathcal{O}_{p_2}(i')$ avec $i' < \max(|u_1|, |u_2|) + \text{ ppcm } (|v_1|, |v_2|)$.
Par hypothèse, $\mathcal{O}_{p_1}(i') - \mathcal{O}_{p_2}(i') \leq t$ donc $\forall i \geq 1,\ \mathcal{O}_{p_1}(i) - \mathcal{O}_{p_2}(i) \leq t$. □

Dans l'exemple (4.1), la taille de buffer nécessaire et suffisante jusqu'à l'instant 6 est suffisante à l'infini.

4.5 Mots affines

Les mots affines sont des cas particuliers de mots ultimement périodiques, de la forme $p_a = \mathtt{0}^d\mathtt{(10}^{l-1}\mathtt{)}$. Ils ont été utilisés pour améliorer le calcul d'horloge des langages Signal [Sma98] et Lustre [CCM+03, Cur05]. Les formules que nous venons d'énoncer pour les mots périodiques s'appliquent au sous-cas des mots affines. Étant donné la forme particulière de ces derniers, elles se simplifient dans leur cas. Nous listons donc ci-dessous l'ensemble des formules spécialisées pour ces mots.

Description des mots

Soit $p_a = \mathtt{0}^d\mathtt{(10}^{\ell-1}\mathtt{)}$. En remarquant que la taille du préfixe vaut d, que la taille du motif périodique vaut ℓ, que le nombre de `1` du motif périodique vaut 1, et que mis à part le premier élément du motif périodique, tous les éléments valent `0`, on obtient les quatre formules suivantes à partir de celles des horloges périodiques.

éléments de p_a	$p_a[i] = \begin{cases} \mathtt{1} \text{ si } i = d + x \times \ell + 1 \text{ avec } x \in \mathbb{N} \\ \mathtt{0} \text{ sinon} \end{cases}$
test d'égalité	$p_{a_1} = p_{a_2} \Leftrightarrow (d_1 = d_2) \wedge (\ell_1 = \ell_2)$
indice du $j^{ième}$ `1`	$\mathcal{I}_{p_a}(j) = d + (j-1) \times \ell + 1$
fonction de cumul	$\mathcal{O}_{p_a}(i) = \begin{cases} 0 \text{ si } i \leq d \\ \left\lceil \frac{(i-d)}{\ell} \right\rceil \text{ sinon} \end{cases}$

Ces formules sont utilisées dans les preuves des formules des opérateurs et relations sur les mots affines.

Opérateurs sur les mots

L'ensemble des mots affines n'est pas clos par application de l'opérateur ***not***. En effet, les seuls mots affines dont la négation est un mot affine sont `(10)` et `0(10)` : ***not*** `(10)` = `0(10)`. Par contre, ***not*** $\mathtt{0}^2\mathtt{(1000)} = \mathtt{1}^2\mathtt{(0111)}$. Cet opérateur ne doit donc pas être calculé si l'on souhaite rester dans le domaine des mots affines.

En revanche, les mots affines sont clos par application de l'opérateur ***on***. Il peut être calculé ainsi :

$$p_{a_1} \textbf{ on } p_{a_2} = \mathtt{0}^{d_1+d_2\times\ell_1}\mathtt{(10}^{\ell_1\times\ell_2-1}\mathtt{)}$$

Par exemple, $\mathtt{0}^5\mathtt{(10}^{4-1}\mathtt{)}$ ***on*** $\mathtt{0}^2\mathtt{(10}^{2-1}\mathtt{)} = \mathtt{0}^{5+2\times4}\mathtt{(10}^{4\times2-1}\mathtt{)} = \mathtt{0}^{13}\mathtt{(10}^{8-1}\mathtt{)}$.

Démonstration :

> La preuve de cette formule est fournie en annexe E. Elle utilise la formule du ***on*** sur les mots périodiques pour calculer $p_{a_3} = p_{a_1}$ ***on*** p_{a_2}, et montre que $p_{a_3} = u_3\mathtt{(}v_3\mathtt{)}$ est un mot affine égal à $\mathtt{0}^{d_1+d_2\times\ell_1}\mathtt{(10}^{\ell_1\times\ell_2-1}\mathtt{)}$. Pour cela, on calcule les tailles de u_3 et de v_3 à partir de la formule de la proposition 4.1, et on montre que u_3 n'est composé que de `0`, et que v_3 ne contient qu'un seul `1`. On allonge ensuite u_3 afin que le seul `1` de la partie périodique se trouve en première position de celle-ci. □

Le calcul des mots permettant d'obtenir un mot p_{a_1} par ralentissement d'un mot p_{a_2}, c'est-à-dire le calcul de $S(p_{a_1}, p_{a_2})$, peut lui aussi être spécialisé aux mots affines. On note

$S_a(p_{a_1}, p_{a_2})$ l'ensemble des mots affines de $S(p_{a_1}, p_{a_2})$. Cet ensemble est non-vide si et seulement si $\ell_2 \mid \ell_1$ et $d_1 \geq d_2 \times \frac{\ell_1}{\ell_2}$. Dans ce cas, il contient l'unique élément suivant :

$$S_a(0^{d_1}(10^{\ell_1-1}), 0^{d_2}(10^{\ell_2-1})) = 0^{d_1-d_2\times\frac{\ell_1}{\ell_2}}(10^{\frac{\ell_1}{\ell_2}-1})$$

Démonstration :

La preuve de cette formule est donnée en annexe E. Elle suit le raisonnement suivant. Les éléments de $S(p_{a_1}, p_{a_2})$ sont des mots périodiques $p = u(v)$. En appliquant la formule de S sur les mots périodiques, on obtient le nombre de 1 des u et des v, ainsi que le nombre d'éléments à parcourir dans p_{a_1} et p_{a_2} pour obtenir tous les éléments des mots p. En appliquant l'algorithme pour trouver ces éléments, on obtient la taille des u et des v. En contraignant la forme des u et des v pour que les mots p puissent être mis sous la forme de mots affines $0^d(10^{\ell-1})$, on peut en déduire que la seule valeur possible pour d est $d_1 - d_2 \times \frac{\ell_1}{\ell_2}$ et la seule valeur possible pour ℓ est $\frac{\ell_1}{\ell_2}$. Par conséquent, si $d \notin \mathbb{N}$ ou $\ell \notin \mathbb{N}^*$, il n'y a aucun mot affine dans $S(p_{a_1}, p_{a_2})$, dans le cas contraire il y a un seul mot affine dans cet ensemble, $0^d(10^{\ell-1})$. □

Le calcul du rythme commun le plus rapide diffère de celui du cas général des mots périodiques, si l'on ne veut utiliser que des mots affines pour ralentir les mots, c'est-à-dire si l'on cherche un p_a tel que $(S_a(p_a, p_{a_1}) \neq \varnothing) \wedge (S_a(p_a, p_{a_2}) \neq \varnothing)$ plutôt qu'un p tel que $(S(p, p_{a_1}) \neq \varnothing) \wedge (S(p, p_{a_2}) \neq \varnothing)$. Le rythme commun affine le plus rapide n'utilisant comme ralentisseurs que des mots affines est :

$$\inf_a(0^{d_1}(10^{\ell_1-1}), 0^{d_2}(10^{\ell_2-1})) = 0^d(10^{\ell-1}) \text{ avec } \ell = \text{ ppcm } (\ell_1, \ell_2) \text{ et } d = \max(d_2\times\frac{\ell}{\ell_2}, d_1\times\frac{\ell}{\ell_1})$$

Démonstration :

Soient $p_{a_1} = 0^{d_1}(10^{\ell_1-1})$ et $p_{a_2} = 0^{d_2}(10^{\ell_2-1})$.
On cherche le p_a plus plus rapide tel que $(S_a(p_a, p_{a_1}) \neq \varnothing) \wedge (S_a(p_a, p_{a_2}) \neq \varnothing)$.
$(S_a(p_a, p_{a_1}) \neq \varnothing) \wedge (S_a(p_a, p_{a_2}) \neq \varnothing) \Leftrightarrow (\ell_1 \mid \ell) \wedge (d \geq d_1 \times \frac{\ell}{\ell_1}) \wedge (\ell_2 \mid \ell) \wedge (d \geq d_2 \times \frac{\ell}{\ell_2})$. Comme on cherche le p_a le plus rapide, on choisit $\ell = \text{ ppcm } (\ell_1, \ell_2)$ et $d = \max(d_2 \times \frac{\ell}{\ell_2}, d_1 \times \frac{\ell}{\ell_1})$. □

Par exemple, le rythme commun le plus rapide de $0^2(10^3)$ et $0^8(10^5)$ est $0^{16}(10^{11})$, qui est égal à $0^{10}(10^2)$ *on* $0^2(10^3)$ et à (10) *on* $0^8(10^5)$.

Relations sur les mots

Par simple application des formules des tests sur les horloges périodiques, les relations définissant l'adaptabilité deviennent ici :

synchronisabilité	$0^{d_1}(10^{\ell_1-1}) \bowtie 0^{d_2}(10^{\ell_2-1}) \Leftrightarrow \ell_1 = \ell_2$
précédence	$0^{d_1}(10^{\ell-1}) \preceq 0^{d_2}(10^{\ell-1}) \Leftrightarrow d_1 \leq d_2$

Le supremum et l'infimum pour la relation de précédence se calculent ainsi sur les mots affines synchronisables :

$$p_{a_1} \sqcup p_{a_2} = 0^{\max(d_1,d_2)}(10^{\ell-1})$$
$$p_{a_1} \sqcap p_{a_2} = 0^{\min(d_1,d_2)}(10^{\ell-1})$$

Enfin, si $p_{a_1} <: p_{a_2}$, la taille de buffer pour consommer un flot d'horloge p_{a_1} sur une horloge p_{a_2} vaut :

$$size(0^{d_1}(10^{\ell-1}), 0^{d_2}(10^{\ell-1})) = \left\lceil \frac{d_2 - d_1}{\ell} \right\rceil$$

Démonstration :

Comme $p_{a_1} <: p_{a_2}$, on a $\ell_1 = \ell_2 = \ell$ et $d_1 \leq d_2$.
Donc $size(0^{d_1}(10^{\ell-1}), 0^{d_2}(10^{\ell-1})) = \max_{1 \leq i \leq d_2+l}(\mathcal{O}_{p_{a_1}}(i) - \mathcal{O}_{p_{a_2}}(i))$.
Pour $i = 1$ à d_1, $\mathcal{O}_{p_{a_1}}(i) = 0$ et $\mathcal{O}_{p_{a_2}}(i) = 0$. Pour $i = d_1 + 1$ à d_2, $\mathcal{O}_{p_{a_1}}(i) = \left\lceil \frac{i-d_1}{\ell} \right\rceil$ et $\mathcal{O}_{p_{a_2}}(i) = 0$. Pour $i = d_2 + 1$ à $d_2 + \ell$, $\mathcal{O}_{p_{a_1}}(i) = \left\lceil \frac{i-d_1}{\ell} \right\rceil$ et $\mathcal{O}_{p_{a_2}}(i) = 1$.
La valeur maximale est donc atteinte pour $i = d_2$ (et atteinte à nouveau pour $i = d_2 + \ell$).
Elle vaut : $\mathcal{O}_{p_{a_1}}(d_2) - \mathcal{O}_{p_{a_2}}(d_2) = \left\lceil \frac{d_2-d_1}{\ell} \right\rceil - \left\lceil \frac{d_2-d_2}{\ell} \right\rceil = \left\lceil \frac{d_2-d_1}{\ell} \right\rceil$. □

Propriété spécifique aux mots affines

La simplification structurelle à droite du *on* dans la relation d'adaptabilité, qui n'est pas complète dans le cas général (voir le lemme 3.8), est complète dans le cas des mots affines :

$$p_{a_1} \textit{ on } p_{a_3} <: p_{a_2} \textit{ on } p_{a_3} \Leftrightarrow p_{a_1} <: p_{a_2}$$

Démonstration :

$$\begin{aligned}
p_{a_1} \textit{ on } p_{a_3} <: p_{a_2} \textit{ on } p_{a_3} &\Leftrightarrow 0^{d_1+d_3\times\ell_1}(10^{\ell_1\times\ell_3-1}) <: 0^{d_2+d_3\times\ell_2}(10^{\ell_1\times\ell_3-1}) \\
&\Leftrightarrow (\ell_1 \times \ell_3 = \ell_2 \times \ell_3) \wedge (d_1 + d_3 \times \ell_1 \leq d_2 + d_3 \times \ell_2) \\
&\Leftrightarrow (\ell_1 = \ell_2) \wedge (d_1 + d_3 \times \ell_1 \leq d_2 + d_3 \times \ell_2) \\
&\Leftrightarrow (\ell_1 = \ell_2) \wedge (d_1 \leq d_2) \\
&\Leftrightarrow p_{a_1} <: p_{a_2}
\end{aligned}$$

□

Dans les prochains chapitres, nous prendrons soin de spécialiser les concepts présentés au cas des horloges affines. Ainsi, dans le chapitre 10, nous pourrons comparer nos travaux avec les règles du calcul d'horloge affines dans Signal.

4.6 Conclusion

Nous avons maintenant mis en place des algorithmes permettant de calculer les opérations et vérifier les relations utilisées dans le calcul d'horloge du chapitre 2. Le chapitre suivant montre donc comment appliquer ce calcul d'horloge à un langage d'horloges périodiques.

Chapitre 5

Typage des horloges périodiques

On définit dans ce chapitre un langage d'horloges périodiques pour Lucy-n, en s'appuyant sur l'algèbre des horloges périodiques présentée dans le chapitre précédent. La définition d'un langage d'horloges nécessite bien sûr d'établir la syntaxe et la sémantique des expressions d'horloges. Mais elle nécessite surtout de fournir des algorithmes de résolution des contraintes d'égalité et de sous-typage sur les types de ces horloges. En effet, ces algorithmes sont indispensables à l'inférence de types : une communication sans buffers impose que le type d'horloges du flot soit égal au type d'horloges attendu, et une communication à travers un buffer borné impose que le type d'horloges du flot soit un sous-type du type d'horloges attendu. Enfin, il faut être en mesure de calculer la taille des buffers nécessaires aux points d'application de la relation de sous-typage.

La section 5.1 décrit les expressions du langage d'horloges à travers leur syntaxe et leur interprétation vers des mots binaires infinis, ainsi qu'un test d'égalité et un test d'adaptabilité.

La section 5.2 définit plusieurs algorithmes d'unification des types d'horloges. Ces algorithmes ont pour but de trouver des instanciations pour les variables de types, telles que les contraintes d'égalité des types sont satisfaites. On montre qu'en interprétant les valeurs des horloges plutôt qu'en les comparant syntaxiquement, on peut augmenter le nombre de cas où le calcul d'horloge est en mesure de garantir l'égalité de deux types. On montre aussi que pour trouver une solution au système de contraintes à chaque fois qu'elle existe, il faut raisonner de manière globale à l'ensemble des contraintes, y compris les contraintes de sous-typage. Ce résultat est la première contribution à retenir de ce chapitre.

La section 5.3 décrit un algorithme de résolution des contraintes de sous-typage, c'est-à-dire un algorithme qui trouve des instanciations des variables de types, telles que les contraintes de sous-typage sont satisfaites. Cet algorithme est correct, mais nous n'avons pas la preuve qu'il mène toujours à une solution si elle existe. Il est d'autre part trop coûteux pour passer à l'échelle sur de grosses applications. Ces travaux constituent à nos yeux une avancée dans la compréhension du problème de la résolution correcte et complète des contraintes de sous-typage sur les horloges périodiques, mais pas un résultat abouti, contrairement à la méthode approchée décrite dans la partie III.

5.1 Un langage d'horloges périodiques

Le langage d'horloges proposé compose des horloges périodiques avec les opérateurs présentés dans le chapitre 3 :

$$\begin{array}{lcl} ce & ::= & c \mid p \mid \texttt{not}\ ce \mid ce\ \texttt{on}\ ce \\ p & ::= & u(v) \\ u & ::= & \varepsilon \mid b.u \\ v & ::= & b \mid b.v \\ b & ::= & 0 \mid 1 \end{array}$$

Une expression d'horloges est soit une variable d'horloge c, soit un mot binaire ultimement périodique p, soit la négation d'une expression d'horloges, soit la composition de deux expressions d'horloges par l'opérateur $\texttt{on}$.

L'interprétation d'une expression ce dans un environnement ρ est notée $[\![ce]\!]^{ce}_{\rho}$ et elle est définie ainsi :

$$\begin{array}{lcll} [\![u(v)]\!]^{ce}_{\rho} & = & u.(lim_{n\to+\infty}\, v^n) & \text{avec } v^0 = \varepsilon \text{ et } v^{n+1} = v.v^n \\ [\![c]\!]^{ce}_{\rho} & = & \rho(c) & \\ [\![\texttt{not}\ ce]\!]^{ce}_{\rho} & = & \texttt{not}\ [\![ce]\!]^{ce}_{\rho} & \text{(définition 3.5 page 53)} \\ [\![ce_1\ \texttt{on}\ ce_2]\!]^{ce}_{\rho} & = & [\![ce_1]\!]^{ce}_{\rho}\ \texttt{on}\ [\![ce_1]\!]^{ce}_{\rho} & \text{(définition 3.6 page 53)} \end{array}$$

Le chapitre 4 a fourni les algorithmes nécessaires à l'utilisation de ce langage d'horloges dans Lucy-n. La section 4.2 a présenté des algorithmes de calcul des mots périodiques résultats d'une opération $\texttt{on}$ ou $\texttt{not}$ sur les mots périodiques, que l'on note ci-dessous $\texttt{on}_p$ et $\texttt{not}_p$. On dispose donc d'une fonction *eval* qui calcule le mot périodique p résultat d'une expression ce.

$$\begin{array}{lcl} eval(u(v)) & = & u(v) \\ eval(\texttt{not}\ ce) & = & \texttt{not}_p\ eval(ce) \\ eval(ce_1\ \texttt{on}\ ce_2) & = & eval(ce_1)\ \texttt{on}_p\ eval(ce_2) \end{array}$$

La section 4.1 a fourni un test d'égalité entre les mots binaires périodiques, que l'on note ci-dessous $=_p$. Il est donc toujours possible de tester l'égalité de deux expressions d'horloges ce_1 et ce_2 en comparant $eval(ce_1)$ et $eval(ce_2)$: $ce_1 = ce_2 \Leftrightarrow eval(ce_1) =_p eval(ce_2)$. On peut cependant optimiser le test d'égalité des expressions d'horloges afin d'effectuer moins de calculs, en appliquant des simplifications structurelles. Nous noterons $=_{ce}$ le test d'égalité ainsi obtenu.

Définition 5.1 (test d'égalité). Soit $cl = p$ ou $\texttt{not}\ p$ ou c ou $\texttt{not}\ c$. Soient ce_1 et ce_2 des expressions quelconques du langage d'horloges périodiques.

$$\begin{array}{lcll} \texttt{not}\ ce_1 =_{ce} \texttt{not}\ ce_2 & \overset{def}{\Leftrightarrow} & ce_1 =_{ce} ce_2 & \\ cl\ \texttt{on}\ ce_1 =_{ce} cl\ \texttt{on}\ ce_2 & \overset{def}{\Leftrightarrow} & ce_1 =_{ce} ce_2 & \text{si } rate(cl) \neq 0 \\ ce_1 =_{ce} ce_2 & \overset{def}{\Leftrightarrow} & eval(ce_1) =_p eval(ce_2) & \text{dans tous les autres cas} \end{array}$$

Dans la deuxième règle, la restriction au sous-ensemble cl des expressions d'horloges vise à éviter de calculer des expressions intermédiaires afin de tester leur égalité.

Ce test est correct et complet : $ce_1 =_{ce} ce_2 \Leftrightarrow eval(ce_1) = eval(ce_2)$.

Démonstration :

Par la remarque 3.5 page 53, le lemme 3.1 page 57 et le lemme 4.2 page 79. □

De même, la section 4.4 a fourni un test de synchronisabilité et un test de précédence sur les mots périodiques. La relation d'adaptabilité étant la conjonction de ces deux relations, on dispose donc d'un test d'adaptabilité sur les mots périodiques, que l'on note $<:_p$. Il est par conséquent toujours possible de tester l'adaptabilité de deux expressions d'horloges en comparant $eval(ce_1)$ et $eval(ce_2)$: $ce_1 <: ce_2 \Leftrightarrow eval(ce_1) <:_p eval(ce_2)$. On peut ici aussi appliquer des simplifications structurelles pour optimiser le test d'adaptabilité des expressions d'horloges. Nous noterons $<:_{ce}$ le test d'adaptabilité ainsi obtenu.

Définition 5.2 (test d'adaptabilité). Soit $cl = p$ ou $\texttt{not}\ p$ ou c ou $\texttt{not}\ c$. Soient ce_1 et ce_2 des expressions quelconques du langage d'horloges périodiques.

$$\begin{array}{lcll} \texttt{not}\ ce_1 <:_{ce} \texttt{not}\ ce_2 & \stackrel{def}{\Leftrightarrow} & ce_2 <:_{ce} ce_1 & \\ cl\ \texttt{on}\ ce_1 <:_{ce} cl\ \texttt{on}\ ce_2 & \stackrel{def}{\Leftrightarrow} & ce_1 <:_{ce} ce_2 & \text{si } rate(cl) \neq 0 \\ ce_1 <:_{ce} ce_2 & \stackrel{def}{\Leftrightarrow} & eval(ce_1) <:_p eval(ce_2) & \text{dans tous les autres cas} \end{array}$$

Ce test est correct et complet : $ce_1 <:_{ce} ce_2 \Leftrightarrow eval(ce_1) <: eval(ce_2)$.

Démonstration :

Par la proposition 3.8. □

Enfin, le calcul de la taille de buffer nécessaire sur les mots périodiques a aussi été fourni dans la section 4.4, nous le notons $size_p$. Une fois de plus, pour calculer cette taille sur des expressions d'horloges, on commence par effectuer des simplifications structurelles avant d'évaluer les expressions et d'effectuer le calcul sur les mots périodiques obtenus.

Définition 5.3 (taille du buffer). Soit $cl = p$ ou $\texttt{not}\ p$ ou c ou $\texttt{not}\ c$. Soient ce_1 et ce_2 des expressions quelconques du langage d'horloges périodiques.

$$\begin{array}{lcll} size_{ce}(cl\ \texttt{on}\ ce_1, cl\ \texttt{on}\ ce_2) & \stackrel{def}{=} & size_{ce}(ce_1, ce_2) & \text{si } rate(cl) \neq 0 \\ size_{ce}(ce_1, ce_2) & \stackrel{def}{=} & size_p(eval(ce_1), eval(ce_2)) & \text{dans tous les autres cas} \end{array}$$

Ce calcul est correct et le plus précis : $size_{ce}(ce_1, ce_2) = size(eval(ce_1), eval(ce_2))$.

Démonstration :

Par le lemme 3.14. □

Remarquons que le langage d'horloges proposé ici peut être complété sans peine par tout opérateur des mots périodiques vers les mots périodiques, pour lequel on dispose d'un algorithme de calcul. Par exemple, l'algorithme de calcul des opérateurs $\sqcup$ et $\sqcap$ sur les mots périodiques synchronisables ayant été fourni dans le chapitre précédent, on pourrait les ajouter au langage. De même, l'ensemble des mots périodiques étant clos par les opérations de conjonction et de disjonction, ces dernières pourraient elles aussi être ajoutées. Pour cela, il faudrait compléter la fonction *eval*, et profiter le cas échéant des propriétés des opérations ajoutées pour effectuer des simplifications structurelles dans les tests et calculs. La conjonctions et la disjonction ne présentent pas de telles propriétés. Par contre, certaines simplifications pourraient être effectuées sur $\sqcup$ et $\sqcap$.

Les types d'horloges dans ce cadre

Nous avons vu dans la section 2.3 que les types des flots ont la forme suivante :

$$ck \quad ::= \quad \alpha \mid ck \texttt{ on } ce \mid ck \texttt{ on not } ce$$

L'égalité de deux types est définie par :

$$ck_1 = ck_2 \stackrel{def}{\Leftrightarrow} \forall x : ck_1,\ \forall y : ck_2,\ clock(x) = clock(y)$$

Le langage d'horloges ce disposant d'opérateurs *on* et *not* correspondant aux opérateurs on et not du langage des types (voir la proposition 3.4 page 57), on peut transformer un type de la manière suivante :

Définition 5.4 (simplification des types).

$$\begin{array}{lcl} simpl(\alpha) & \stackrel{def}{=} & \alpha \\ simpl(\alpha \texttt{ on } ce) & \stackrel{def}{=} & \alpha \texttt{ on } ce \\ simpl((ck \texttt{ on } ce_1) \texttt{ on } ce_2) & \stackrel{def}{=} & simpl(ck \texttt{ on } (ce_1 \textit{ on } ce_2)) \\ simpl(ck \texttt{ on not } ce) & \stackrel{def}{=} & simpl(ck \texttt{ on } \textit{not } ce) \end{array}$$

La simplification des types est correcte : $simpl(ck) = ck$.

Démonstration :

Les deux premières règles sont correctes par réflexivité de l'égalité des types.
Considérons la troisième règle. Pour tout flot x de type $(ck \texttt{ on } ce_1) \texttt{ on } ce_2$, $clock(x)$ est de la forme $(c \textit{ on } ce_1) \textit{ on } ce_2$ avec c l'horloge associée au type ck. Pour tout flot y de type $ck \texttt{ on } (ce_1 \textit{ on } ce_2)$, $clock(y)$ est de la forme $c \textit{ on } (ce_1 \textit{ on } ce_2)$. Par associativité du *on*, $clock(x) = clock(y)$ et par conséquent, par définition de l'égalité des types, $ck \texttt{ on } (ce_1 \textit{ on } ce_2) = (ck \texttt{ on } ce_1) \texttt{ on } ce_2$.
La preuve de la dernière règle suit le même principe. □

On peut donc toujours se ramener à un type de la forme :

$$ck \quad ::= \quad \alpha \mid \alpha \texttt{ on } ce$$

5.2 Unification

Considérons une contrainte d'égalité entre deux types : $\alpha_1 \texttt{ on } ce_1 = \alpha_2 \texttt{ on } ce_2$. En fonction des instanciations des variables α_1 et α_2, cette contrainte peut être ou ne pas être vérifiée. Pour pouvoir garantir que les horloges représentées par les types sont forcément égales, il faut que ceux-ci soient exprimés en fonction de la même variable. L'égalité de deux types de la forme α ou $\alpha \texttt{ on } ce$ est donc vérifiée par la relation :

$$\begin{array}{l} \alpha = \alpha \\ \alpha \texttt{ on } ce_1 = \alpha \texttt{ on } ce_2 \ \Leftrightarrow \ ce_1 = ce_2 \end{array}$$

Unifier les types ck_1 et ck_2 consiste à trouver une substitution θ telle que $\theta(ck_1) = \theta(ck_2)$. Il faut donc que $\theta(ck_1)$ et $\theta(ck_2)$ soient exprimés en fonction de la même variable de type.

Dans le cas où ck_1 et ck_2 sont déjà exprimés en fonction de la même variable, il suffit de vérifier l'égalité de ck_1 et ck_2. Nous considérerons dans toute la suite de cette section que $\alpha_1 \neq \alpha_2$. Dans le cas où un des types est une variable, il suffit de lui associer la valeur du second type. Dans le cas où $ck_1 = \alpha_1$ `on` ce_1 et $ck_2 = \alpha_2$ `on` ce_2, tout unificateur valide est d'une des formes suivantes :

- $\theta(\alpha_1) = \alpha$ `on` c_1 et $\theta(\alpha_2) = \alpha$ `on` c_2 avec c_1 ***on*** $ce_1 = c_2$ ***on*** ce_2
- $\theta(\alpha_1) = \alpha_2$ `on` c_1 avec c_1 ***on*** $ce_1 = ce_2$
- $\theta(\alpha_2) = \alpha_1$ `on` c_2 avec $ce_1 = c_2$ ***on*** ce_2

5.2.1 Unification structurelle

L'unification structurelle des types mis sous la forme α ou α `on` ce repose sur le même principe que l'unification structurelle présentée dans la section 2.3 (page 44), sauf que cette fois on analyse la structure de ce (modulo associativité de l'opérateur ***on***). Nous la noterons $\equiv_s$.

$$
\begin{array}{rcl}
\alpha_1 \equiv_s \alpha_2 & & \text{avec } \theta(\alpha_1) = \alpha_2 \\
\alpha_1 \equiv_s \alpha_2 \texttt{ on } ce_2 & & \text{avec } \theta(\alpha_1) = \alpha_2 \texttt{ on } ce_2 \\
\alpha_1 \texttt{ on } ce_1 \equiv_s \alpha_2 & & \text{avec } \theta(\alpha_2) = \alpha_1 \texttt{ on } ce_1 \\
\alpha_1 \texttt{ on } (ce_1 \texttt{ on } p_1) \equiv_s \alpha_2 \texttt{ on } (ce_2 \texttt{ on } p_2) & \overset{def}{\Leftrightarrow} & \alpha_1 \texttt{ on } ce_1 \equiv_s \alpha_2 \texttt{ on } ce_2 \text{ et } p_1 = p_2 \\
\alpha_1 \texttt{ on } (ce_1 \texttt{ on not } p_1) \equiv_s \alpha_2 \texttt{ on } (ce_2 \texttt{ on not } p_2) & \overset{def}{\Leftrightarrow} & \alpha_1 \texttt{ on } ce_1 \equiv_s \alpha_2 \texttt{ on } ce_2 \text{ et } p_1 = p_2
\end{array}
$$

Pour plus de concision, nous considérons ici et dans la suite de ce chapitre que toutes les variables d'horloges liées dans l'environnement sont remplacées par leur valeur.

5.2.2 Unification avec interprétation des expressions d'horloges

L'unification présentée ci-dessus ne fait qu'analyser la structure des expressions d'horloges, sans calculer leur valeur. Nous présentons dans cette section une unification qui trouve c_1 et c_2 tels que c_1 ***on*** $ce_1 = c_2$ ***on*** ce_2 en interprétant la valeur des expressions d'horloges ce_1 et ce_2. Nous noterons cette unification $\equiv_i$.

$$
\begin{array}{rl}
\alpha_1 \equiv_i \alpha_2 & \text{avec } \theta(\alpha_1) = \alpha_2 \\
\alpha_1 \equiv_i \alpha_2 \texttt{ on } ce_2 & \text{avec } \theta(\alpha_1) = \alpha_2 \texttt{ on } ce_2 \\
\alpha_1 \texttt{ on } ce_1 \equiv_i \alpha_2 & \text{avec } \theta(\alpha_2) = \alpha_1 \texttt{ on } ce_1 \\
\alpha_1 \texttt{ on } ce_1 \equiv_i \alpha_2 \texttt{ on } ce_2 & \text{avec } \theta(\alpha_1) = \alpha \texttt{ on } c_1,\ \theta(\alpha_2) = \alpha \texttt{ on } c_2, \\
 & \quad p_1 = eval(ce_1),\ p_2 = eval(ce_2),\ m = \inf(p_1, p_2) \\
 & \quad c_1 \in S(m, p_1),\ c_2 \in S(m, p_2)
\end{array}
$$

Les deux horloges ce_1 et ce_2 s'évaluent respectivement en les mots périodiques p_1 et p_2. On choisit de les ramener à leur rythme commun le plus rapide $m = \inf(p_1, p_2)$ (voir les sections 3.3 page 59 et 4.3 page 83). Pour cela, il suffit de choisir $c_1 \in S(m, p_1)$ et $c_2 \in S(m, p_2)$. Cependant, ceux-ci ne constituent pas un unificateur le plus général puisque tous les autres éléments de $S(m, p_1)$ et $S(m, p_2)$ conviennent et ne sont pas moins généraux.

Cette unification ne renvoyant pas d'unificateur le plus général et traitant les contraintes une à une, elle ne constitue pas un algorithme complet. Ceci signifie qu'elle peut échouer à trouver une solution à un système de contraintes d'unifications qui a des solutions. En ce qui concerne la résolution d'une contrainte, elle est toujours plus puissante que la résolution structurelle. En effet, contrairement à cette dernière, elle n'échoue jamais si les variables

d'horloges des termes à unifier sont différentes : α_1 `on (10) on (10)` $\equiv \alpha_2$ `on (1000)` échoue avec la méthode structurelle, et renvoie $\theta(\alpha_1) = \theta(\alpha_2) = \alpha$ `on (1)` avec la méthode interprétée.

On peut néanmoins se demander si l'unification interprétée est toujours plus puissante que l'unification structurelle en ce qui concerne la résolution d'un ensemble de contraintes. La réponse est non, en voici un contre-exemple :

$$\left\{ \begin{array}{lcl} \alpha_1 \texttt{ on } p_1 \texttt{ on } p_2 & \equiv & \alpha_2 \texttt{ on } p_2 \\ \alpha_1 \texttt{ on } p_1 & \equiv & \alpha_2 \end{array} \right\}$$

L'unification structurelle appliquée à la première contrainte calcule l'unificateur $\theta(\alpha_2) = \alpha_1$ `on` p_1 qui satisfait toujours la deuxième contrainte.
L'unification avec interprétation appliquée à la première contrainte calcule le rythme commun le plus rapide $\inf(p_1 \texttt{ on } p_2, p_2) = p_1 \texttt{ on } p_2$. Elle choisit un unificateur $\theta(\alpha_2) = \alpha_1$ `on` c_1 avec $c_1 \in S(p_1 \texttt{ on } p_2, p_2)$, qui contient p_1 mais pas seulement. Donc si le c_1 choisi n'est pas p_1, la deuxième contrainte qui devient : α_1 `on` $p_1 \equiv \alpha_1$ `on` c_1 n'est pas satisfaisable.

Par exemple, si p_1 = `(001)` et p_2 = `(10)`, on doit unifier α_1 `on (001) on (10)` = α_1 `on (001000)` avec α_2 `on (10)`. Le rythme commun le plus rapide est `(001000)`. On choisit alors c_1 dans l'ensemble $S(\texttt{(001000)}, \texttt{(10)}) = \{p \mid p = \texttt{(001.}v\texttt{)} \text{ avec } |v| = 3 \text{ et } |v|_1 = 1\}$. Prenons par exemple le mot au plus tôt selon la relation $\preceq$, c'est-à-dire c_1 = `(001100)`. On obtient alors l'instanciation $\alpha_2 = \alpha_1$ `on (001100)`, et la deuxième contrainte du système qui devient α_1 `on (001)` $\equiv \alpha_1$ `on (001100)` ne peut pas être satisfaite.

Unification semi-interprétée Afin d'obtenir un algorithme d'unification gloutonne strictement plus puissant que l'algorithme structurel, on peut appliquer l'unification structurelle tant qu'elle fonctionne, et lorsqu'elle échoue, unifier les types non unifiables structurellement avec la méthode interprétée :

$$\begin{array}{rcl} \alpha_1 \equiv_{si} \alpha_2 & \text{avec} & \theta(\alpha_1) = \alpha_2 \\ \alpha_1 \equiv_{si} \alpha_2 \texttt{ on } ce_2 & \text{avec} & \theta(\alpha_1) = \alpha_2 \texttt{ on } ce_2 \\ \alpha_1 \texttt{ on } ce_1 \equiv_{si} \alpha_2 & \text{avec} & \theta(\alpha_2) = \alpha_1 \texttt{ on } ce_1 \end{array}$$

$$\begin{array}{rcl} \alpha_1 \texttt{ on } (ce_1 \texttt{ on } ce) \equiv_{si} \alpha_2 \texttt{ on } (ce_2 \texttt{ on } ce) & \stackrel{def}{\Leftrightarrow} & \alpha_1 \texttt{ on } ce_1 \equiv_{si} \alpha_2 \texttt{ on } ce_2 \\ \text{avec } ce \text{ de la forme } p, c, \texttt{not } p \text{ ou } \texttt{not } c & & \end{array}$$

$$\begin{array}{rcl} \alpha_1 \texttt{ on } ce_1 \equiv_{si} \alpha_2 \texttt{ on } ce_2 & \text{avec} & \theta(\alpha_1) = \alpha \texttt{ on } c_1,\ \theta(\alpha_2) = \alpha \texttt{ on } c_2, \\ & & p_1 = eval(ce_1),\ p_2 = eval(ce_2), \\ & & m = \inf(p_1, p_2), \\ & & c_1 \in S(m, p_1),\ c_2 \in S(m, p_2) \end{array}$$

L'unification semi-interprétée reste incomplète. Par exemple, lors de la résolution de la première contrainte de l'ensemble $\{\alpha_1 \equiv \alpha_2;\ \alpha_1$ `on (01)` $\equiv \alpha_2$ `on (10)`$\}$, la résolution semi-interprétée choisit l'unificateur $\theta(\alpha_1) = \alpha_2$. En appliquant la substitution θ à la seconde contrainte, on obtient la contrainte α_2 `on (01)` $\equiv \alpha_2$ `on (10)` qui n'est pas satisfaisable. La résolution échoue donc, alors qu'il existe des solutions (par exemple la substitution $\theta(\alpha_1)$ = `(110)` et $\theta(\alpha_2)$ = `(011)`).

Cas des horloges affines Si l'on cherche à unifier deux types d'horloges affines α_1 `on` p_{a_1} et α_2 `on` p_{a_2} en instanciant α_1 et α_2 par des types d'horloges affines, il faut choisir $m = \inf_a(p_{a_1}, p_{a_2})$ comme rythme commun le plus rapide (défini page 91). Pour atteindre ce

rythme, on choisit $\alpha_1 = \alpha$ `on` c_{a_1} et $\alpha_2 = \alpha$ `on` c_{a_2} avec $c_{a_1} \in S_a(m, p_{a_1})$ et $c_{a_2} \in S_a(m, c_{a_2})$. Ces ensembles contiennent un unique élément, il n'y a donc qu'un seul choix d'unificateur. Par conséquent, l'unificateur choisi est le plus général et l'unification interprétée est correcte et complète dans le domaine des horloges affines.

5.2.3 Unification globale

Nous avons vu que nous ne disposons pas d'unificateur le plus général sur les types d'horloges. Le choix de l'unificateur de deux types dont les variables sont α_1 et α_2 dépend donc de l'ensemble des contraintes existant sur α_1 et α_2.

Par conséquent, pour que l'algorithme de résolution soit complet, il faut résoudre les contraintes de manière globale. Comme la relation de sous-typage est antisymétrique, $ck_1 = ck_2 \Leftrightarrow (ck_1 <: ck_2) \wedge (ck_2 <: ck_1)$. La contrainte d'unification $ck_1 \equiv ck_2$ est donc équivalente à la paire de contraintes $ck_1 <: ck_2$ et $ck_2 <: ck_1$. On peut collecter les contraintes d'unification dans le même ensemble que l'on collecte les contraintes de sous-typage, et résoudre le système ainsi obtenu de manière globale. Toutefois, pour éviter de faire trop grossir le système de contraintes à résoudre de manière globale, on peut, comme pour l'unification interprétée, choisir une méthode "semi-globale" qui ne collecte que les contraintes ne pouvant être résolues structurellement. L'algorithme redevient alors incomplet, mais il est plus puissant que l'algorithme semi-interprété, dans le cas où l'algorithme de résolution des contraintes de sous-typage est complet.

5.3 Résolution des contraintes de sous-typage

Pendant l'inférence de types d'un nœud, on collecte un ensemble de contraintes de sous-typage $\{ck_{x_i} <: ck_{y_i}\}_i$. Nous avons établi dans la section 5.1 que les types seraient sous la forme α ou α `on` ce. La résolution des contraintes interprétant les expressions d'horloges, on peut sans restriction décider de transformer les types se résumant à une variable α en le type équivalent α `on (1)`, afin de ne manipuler que des types sous la forme α `on` ce. Le système de contraintes de sous-typage à résoudre est donc de la forme :

$$C = \{\alpha_{x_i} \texttt{ on } ce_{x_i} <: \alpha_{y_i} \texttt{ on } ce_{y_i}\}_{i=1..\text{nombre de contraintes}}$$

Nous avons vu dans la section 2.3 que la relation de sous-typage est définie par :

$$ck_1 <: ck_2 \overset{def}{\Leftrightarrow} \forall x : ck_1, \forall y : ck_2,\ clock(x) <: clock(y)$$

Ainsi, sur les types de la forme considérée ici, elle est vérifiée par la relation :

$$\alpha \texttt{ on } ce_1 <: \alpha \texttt{ on } ce_2 \Leftrightarrow ce_1 <: ce_2$$

La relation de sous-typage est donc vérifiée si et seulement si les variables de type sont les mêmes et les expressions d'horloges ce_1 et ce_2 sont reliées par la relation d'adaptabilité (définie page 72).

Résoudre la contrainte de sous-typage $ck_1 <: ck_2$ consiste à trouver une substitution θ telle que $\theta(ck_1) <: \theta(ck_2)$. Il faut donc que $\theta(ck_1)$ et $\theta(ck_2)$ soient exprimés en fonction de la même variable de type. Dans le cas où ck_1 et ck_2 sont déjà exprimées en fonction de la même variable de type, quelle que soit l'instanciation, la contrainte prend la même valeur de vérité. Par conséquent, il s'agit seulement dans ce cas de vérifier que la contrainte est satisfaite, et la substitution inférée est vide. Dans le cas où $ck_1 = \alpha_1$ `on` ce_1 et $ck_2 = \alpha_2$ `on` ce_2 avec $\alpha_1 \neq \alpha_2$, toute substitution valide est d'une des formes suivantes :

- $\theta(\alpha_1) = \alpha$ `on` c_1 et $\theta(\alpha_2) = \alpha$ `on` c_2 avec c_1 *on* $ce_1 <: c_2$ *on* ce_2
- $\theta(\alpha_1) = \alpha_2$ `on` c_1 avec c_1 *on* $ce_1 <: ce_2$
- $\theta(\alpha_2) = \alpha_1$ `on` c_2 avec $ce_1 <: c_2$ *on* ce_2

Comme la deuxième et la troisième forme se ramènent à la première forme dans laquelle c_2 (resp. c_1) vaut l'élément neutre `(1)`, on peut chercher des solutions de la première forme seulement sans être incomplets.

Pour un taux donné, il n'existe pas un mot périodique qui soit plus petit que tous les autres (ou plus grand que tous les autres) pour la relation $\preceq$, [1] donc pour la relation $<:$. Par exemple :

$$\ldots <: \mathtt{1}^{999}\mathtt{(10)} <: \ldots <: \mathtt{1(10)} <: \mathtt{(10)} <: \mathtt{0(10)} <: \ldots <: \mathtt{0}^{999}\mathtt{(10)} <: \ldots$$

Par conséquent, si une substitution $\theta(\alpha_1) = \alpha$ `on` c_1 et $\theta(\alpha_2) = \alpha$ `on` c_2 convient pour satisfaire une contrainte α_1 `on` $ce_1 <: \alpha_2$ `on` ce_2, alors l'infinité de substitutions suivantes convient aussi :

- $\theta(\alpha_1) = \alpha$ `on` $\mathtt{0}^{d_1}.c_1$ et $\theta(\alpha_2) = \alpha$ `on` $\mathtt{0}^{d_2}.c_2$, quels que soient d_1 et d_2 tels que $d_1 \leq d_2$;
- $\theta(\alpha_1) = \alpha$ `on` $\mathtt{1}^{d_1}.c_1$ et $\theta(\alpha_2) = \alpha$ `on` $\mathtt{1}^{d_2}.c_2$, quels que soient d_1 et d_2 tels que $d_1 \geq d_2$;
- $\theta(\alpha_1) = \alpha$ `on` $\mathtt{1}^{d_1}.c_1$ et $\theta(\alpha_2) = \alpha$ `on` $\mathtt{0}^{d_2}.c_2$, quels que soient d_1 et d_2.

La technique consistant à choisir le type le plus grand selon la relation de sous-typage pour une variable située toujours à droite dans les contraintes, et le type le plus petit pour une variable située toujours à gauche ne peut donc pas s'appliquer dans notre cas. Par conséquent, l'inférence d'une substitution satisfaisant les contraintes doit forcément être faite de manière globale afin de choisir des types suffisamment grands, et/ou suffisamment petits pour satisfaire toutes les contraintes.

Remarque 5.1. Si l'on considère deux contraintes portant sur une même paire de variables, il n'existe pas forcément une contrainte plus stricte parmi les deux, c'est-à-dire un ordre de résolution tel que si la substitution inférée de manière gloutonne satisfait la première contrainte, alors elle satisfera la seconde. Par exemple, si l'on considère le système suivant :

$$\left\{ \begin{array}{lcl} \alpha_1 \texttt{ on (1001)} & <: & \alpha_2 \texttt{ on (100001)} \\ \alpha_1 \texttt{ on (0110)} & <: & \alpha_2 \texttt{ on (001100)} \end{array} \right\}$$

La substitution $\theta(\alpha_1) = \alpha_2$ `on (110011)` satisfait la première contrainte (en rendant les deux types égaux), mais pas la seconde car `(110011)` *on* `(0110)` = `(010010)` $\not<:$ `(001100)`. La substitution $\theta(\alpha_1) = \alpha_2$ `on (011110)` satisfait la seconde contrainte (là aussi en unifiant les deux types), mais pas la première car `(011110)` *on* `(1001)` = `(010010)` $\not<:$ `(100001)`. Il n'existe donc pas d'ordre de résolution des contraintes de ce système permettant de construire une

[1] Les mots `(1)` et `(0)` constituent une exception à cette règle. En effet, pour tout p de taux 1, `(1)` $<: p$ et pour tout p de taux 0, $p <:$ `(0)`.

solution de manière incrémentale. Enfin, notons que la substitution $\theta(\alpha_1) = \alpha_2$ `on (101101)` est une solution au système car elle satisfait les deux contraintes, mais elle ne satisfait pas la contrainte α_1 `on` $(\texttt{(1001)} \sqcup \texttt{(0110)}) <: \alpha_2$ `on` $(\texttt{(100001)} \sqcap \texttt{(001100)})$. Il n'est donc pas possible de simplifier le système en utilisant les opérations de supremum et d'infimum sans perdre la complétude de la résolution. ◇

5.3.1 Simplification du système

Nous avons vu qu'une contrainte portant sur deux types exprimés en fonction de la même variable α `on` $ce_1 <: \alpha$ `on` ce_2 est vérifiée si et seulement si $ce_1 <: ce_2$. La satisfaction d'une telle contrainte se résume donc en un test d'adaptabilité entre les expressions d'horloges impliquées (qui ne contiennent pas d'inconnue), comme décrit dans la définition 5.2. On commence par traiter toutes les contraintes de cette forme en testant leur valeur de vérité. Si l'une d'entre elles est fausse, le système n'a pas de solution. Par exemple, le système de contraintes généré par le typage du nœud `also_bad` du chapitre 2 (page 29) est $\{\alpha$ `on (1)` $<: \alpha$ `on (10)`$\}$. Ce système contient une contrainte fausse (car `(1)` $\not<:$ `(10)`). Le programme `also_bad` est donc mal typé.

Si toutes les contraintes qui ne nécessitent qu'une vérification ont la valeur vraie, on les supprime du système. Dans certains cas, le système est alors vide, et on l'a donc résolu en ne faisant que de la vérification. Le système de contraintes étant toujours vérifié, la substitution résultat est vide. C'est le cas pour le nœud `good` du chapitre 2 (page 29). Nous avons vu dans son arbre de typage (page 47) que l'ensemble de contraintes collectées est $\{\alpha$ `on (10)` $<: \alpha$ `on (01)`$\}$. La simplification de cet ensemble vérifie que `(10)` $<:$ `(01)`. Comme cette relation d'adaptabilité est vérifiée, la contrainte de sous-typage est toujours satisfaite et l'ensemble simplifié est vide.

Dans les autres cas, le système ne contient après simplification que des contraintes portant sur deux variables différentes, comme par exemple le système suivant :

$$C_1 = \left\{ \begin{array}{lcl} \alpha_1 \texttt{ on } \textit{not} \texttt{ (100)} & <: & \alpha_2 \texttt{ on (10)} \\ \alpha_1 \texttt{ on (1)} & <: & \alpha_3 \texttt{ on (01)} \\ \alpha_3 \texttt{ on (011) } \textit{on} \texttt{ (10)} & <: & \alpha_2 \texttt{ on (01)} \end{array} \right\}$$

On calcule alors les expressions d'horloges apparaissant dans le système, afin d'obtenir un système sous la forme $C = \{\alpha_{x_i}$ `on` $p_{x_i} <: \alpha_{y_i}$ `on` $p_{y_i}\}_i$ avec $\alpha_{x_i} \neq \alpha_{y_i}$ pour tout i.

Le système C_1 est donc mis sous la forme :

$$C_1 = \left\{ \begin{array}{lcl} \alpha_1 \texttt{ on (011)} & <: & \alpha_2 \texttt{ on (10)} \\ \alpha_1 \texttt{ on (1)} & <: & \alpha_3 \texttt{ on (01)} \\ \alpha_3 \texttt{ on (010)} & <: & \alpha_2 \texttt{ on (01)} \end{array} \right\}$$

5.3.2 Transformation en un système de contraintes d'adaptabilité

Nous avons vu que l'on peut chercher sans restriction une substitution résultat θ telle que pour toute contrainte α_x `on` $p_x <: \alpha_y$ `on` p_y, on ait $\theta(\alpha_x) = \alpha$ `on` c_x et $\theta(\alpha_y) = \alpha$ `on` c_y avec c_x *on* $p_x <: c_y$ *on* p_y. Si le système de contraintes est connexe, c'est-à-dire si tout couple de variables du système est relié soit par une contrainte directe soit par une contrainte par transitivité, alors la substitution résultat exprime toutes les variables α_x, α_y en fonction d'une

unique variable α. Si ce n'est pas le cas, on sépare le système en autant de sous-systèmes qu'il y a de composantes connexes, et on résout les sous-systèmes de manière indépendante.

Le système C_1 est connexe. Toute substitution θ solution est donc de la forme

$$\theta(\alpha_1) = \alpha \texttt{ on } c_1,\ \theta(\alpha_2) = \alpha \texttt{ on } c_2,\ \theta(\alpha_3) = \alpha \texttt{ on } c_3$$

avec c_1, c_2, c_3 tels que $\theta(C_1)$ est vérifié :

$$\theta(C_1) = \left\{ \begin{array}{lcl} \alpha \texttt{ on } c_1 \texttt{ on (011)} & <: & \alpha \texttt{ on } c_2 \texttt{ on (10)} \\ \alpha \texttt{ on } c_1 \texttt{ on (1)} & <: & \alpha \texttt{ on } c_3 \texttt{ on (01)} \\ \alpha \texttt{ on } c_3 \texttt{ on (010)} & <: & \alpha \texttt{ on } c_2 \texttt{ on (01)} \end{array} \right\}$$

$\theta(C_1)$ est vérifié si et seulement si l'ensemble de contraintes d'adaptabilité suivant est vérifié :

$$A_1 = \left\{ \begin{array}{lcl} c_1 \texttt{ on (011)} & <: & c_2 \texttt{ on (10)} \\ c_1 \texttt{ on (1)} & <: & c_3 \texttt{ on (01)} \\ c_3 \texttt{ on (010)} & <: & c_2 \texttt{ on (01)} \end{array} \right\}$$

On a donc transformé l'ensemble des contraintes de sous-typage $C = \{\alpha_{x_i} \texttt{ on } p_{x_i} <: \alpha_{y_i} \texttt{ on } p_{y_i}\}_i$ en un ensemble de contraintes d'adaptabilité $A = \{c_{x_i} \texttt{ on } p_{x_i} <: c_{y_i} \texttt{ on } p_{y_i}\}_i$ par composante connexe. On doit maintenant trouver des valeurs pour les variables c_{x_i} et c_{y_i} telles que A soit satisfait. Nous allons donc reprendre les définitions et propositions du chapitre 3 pour déterminer les propriétés qui doivent être satisfaites par les c_{x_i} et c_{y_i}.

Une contrainte d'adaptabilité $c_x \texttt{ on } p_x <: c_y \texttt{ on } p_y$ est définie par une contrainte de synchronisabilité et une contrainte de précédence (définition 3.10) :

$$c_x \texttt{ on } p_x <: c_y \texttt{ on } p_y \ \Leftrightarrow\ (c_x \texttt{ on } p_x \bowtie c_y \texttt{ on } p_y) \wedge (c_x \texttt{ on } p_x \preceq c_y \texttt{ on } p_y)$$

Les résultats des opérations $c_x \texttt{ on } p_x$ et $c_y \texttt{ on } p_y$ sont des mots périodiques que l'on notera respectivement $u_x\texttt{(}v_x\texttt{)}$ et $u_y\texttt{(}v_y\texttt{)}$. La relation de synchronisabilité sur ces mots périodiques peut être vérifiée de la manière suivante (proposition 4.2) :

$$c_x \texttt{ on } p_x \bowtie c_y \texttt{ on } p_y \Leftrightarrow \frac{|v_x|_1}{|v_x|} = \frac{|v_y|_1}{|v_y|}$$

La relation de précédence peut quant à elle être testée ainsi (proposition 4.3) :

$$\begin{array}{c} c_x \texttt{ on } p_x \preceq c_y \texttt{ on } p_y \\ \Leftrightarrow \\ \forall j,\ 1 \leq j \leq \max(|u_x|_1, |u_y|_1) + \text{ ppcm } (|v_x|_1, |v_y|_1),\ \mathcal{I}_{c_x \texttt{ on } p_x}(j) \leq \mathcal{I}_{c_y \texttt{ on } p_y}(j) \end{array}$$

Pour pouvoir parler indifféremment de x et de y, nous allons utiliser l'indice n. Autrement dit, $c_n \texttt{ on } p_n$ désigne dans la suite les $c_x \texttt{ on } p_x$ et $c_y \texttt{ on } p_y$. On s'intéresse donc à la taille et au nombre de 1 des préfixes u_n et parties périodiques v_n des $c_n \texttt{ on } p_n$. Nous noterons u_{c_n} et v_{c_n} le préfixe et la partie périodique de c_n et de même u_{p_n} et v_{p_n} pour les composantes de p_n. Par la proposition 4.1 (page 81), on a :

$$|u_n| = \left\{ \begin{array}{ll} |u_{c_n}| & \text{si } |u_{p_n}| = 0 \\ \max\left(|u_{c_n}|, \mathcal{I}_{c_n}(|u_{p_n}|)\right) & \text{sinon} \end{array} \right. \qquad |u_n|_1 = \mathcal{O}_{p_n}(\mathcal{O}_{c_n}(|u_n|))$$

$$|v_n| = \left\{ \begin{array}{ll} 1 & \text{si } |v_{c_n}|_1 = 0 \\ \frac{\text{ppcm } (|v_{c_n}|_1, |v_{p_n}|)}{|v_{c_n}|_1} \times |v_{c_n}| & \text{sinon} \end{array} \right. \qquad |v_n|_1 = \frac{\text{ppcm } (|v_{c_n}|_1, |v_{p_n}|)}{|v_{p_n}|} \times |v_{p_n}|_1$$

La contrainte de synchronisabilité se traduit donc en :

$$c_x \texttt{ on } p_x \bowtie c_y \texttt{ on } p_y \Leftrightarrow \frac{|v_{c_x}|_1}{|v_{c_x}|} \times \frac{|v_{p_x}|_1}{|v_{p_x}|} = \frac{|v_{c_y}|_1}{|v_{c_y}|} \times \frac{|v_{p_y}|_1}{|v_{p_y}|}$$

La contrainte de précédence se traduit quant à elle en :

$$\begin{array}{c} c_x \texttt{ on } p_x \preceq c_y \texttt{ on } p_y \\ \Leftrightarrow \\ \forall j,\ 1 \leq j \leq \max(|u_x|_1, |u_y|_1) + \ \text{ppcm}\ (|v_x|_1, |v_y|_1),\ \mathcal{I}_{c_x}(\mathcal{I}_{p_x}(j)) \leq \mathcal{I}_{c_y}(\mathcal{I}_{p_y}(j)) \end{array}$$

Pour pouvoir trouver les valeurs caractéristiques des mots c_n, c'est-à-dire leur taille, leur nombre de `1` et les indices de leurs `1`, on va ajuster les p_n du système sous une forme rendant plus simples les formules des contraintes de synchronisabilité et précédence énoncées ci-dessus.

Cas des horloges affines. Dans le cas des types d'horloges affines, les p_x sont de la forme $\texttt{0}^{d_{p_x}}\texttt{(10}^{\ell_{p_x}-1}\texttt{)}$ avec d_{p_x} et ℓ_{p_x} connus, et les c_x que l'on recherche sont de la forme $\texttt{0}^{d_{c_x}}\texttt{(10}^{\ell_{c_x}-1}\texttt{)}$. Une contrainte d'adaptabilité se traduit alors en une formule plus simple. En effet, si l'on applique les formules spécialisées énoncées dans la section 4.5, on obtient :

$$\begin{array}{cc} & \texttt{0}^{d_{c_x}}\texttt{(10}^{\ell_{c_x}-1}\texttt{)} \texttt{ on } \texttt{0}^{d_{p_x}}\texttt{(10}^{\ell_{p_x}-1}\texttt{)} <: \texttt{0}^{d_{c_y}}\texttt{(10}^{\ell_{c_y}-1}\texttt{)} \texttt{ on } \texttt{0}^{d_{p_y}}\texttt{(10}^{\ell_{p_y}-1}\texttt{)} \\ \Leftrightarrow & \texttt{0}^{d_{c_x}+d_{p_x}\times\ell_{c_x}}\texttt{(10}^{\ell_{c_x}\times\ell_{p_x}-1}\texttt{)} <: \texttt{0}^{d_{c_y}+d_{p_y}\times\ell_{c_y}}\texttt{(10}^{\ell_{c_y}\times\ell_{p_y}-1}\texttt{)} \\ \Leftrightarrow & (\ell_{c_x} \times \ell_{p_x} = \ell_{c_y} \times \ell_{p_y}) \ \wedge \ (d_{c_x} + d_{p_x} \times \ell_{c_x} \leq d_{c_y} + d_{p_y} \times \ell_{c_y}) \end{array}$$

Les contraintes de synchronisabilité $(\ell_{c_x} \times \ell_{p_x} = \ell_{c_y} \times \ell_{p_y})$ peuvent être rassemblées dans un système d'équations linéaires homogènes et résolues avec des méthodes classiques, en minimisant les ℓ_{c_n} pour maximiser le rythme. Une fois ces contraintes résolues, les ℓ_{c_n} sont connues donc les contraintes de précédence $(d_{c_x}+d_{p_x}\times\ell_{c_x} \leq d_{c_y}+d_{p_y}\times\ell_{c_y})$ deviennent des inéquations linéaires que l'on peut aussi résoudre par des méthodes classiques.

La suite de ce chapitre traite le cas général des horloges périodiques qui est beaucoup plus complexe.

5.3.3 Traitement des systèmes en forme ajustée

Un système de contraintes est dit *en forme ajustée* si :

1. Pour toute contrainte $c_x \texttt{ on } p_x <: c_y \texttt{ on } p_y$, $|u_{p_x}|_1 = |u_{p_y}|_1$ et $|v_{p_x}|_1 = |v_{p_y}|_1$.
2. Pour toute inconnue c_x, tous les mots p_x, p'_x, ... tels que $c_x \texttt{ on } p_x$, $c_x \texttt{ on } p'_x$, ... apparaissent dans le système ont la même taille de préfixe ($|u_{p_x}| = |u_{p'_x}| = \ldots$) et la même taille de partie périodique ($|v_{p_x}| = |v_{p'_x}| = \ldots$).

La forme ajustée du système de contraintes A_1 est:

$$A'_1 = \left\{ \begin{array}{lcl} c_1 \texttt{ on (011)} & <: & c_2 \texttt{ on (1010)} \\ c_1 \texttt{ on (111)} & <: & c_3 \texttt{ on (010101)} \\ c_3 \texttt{ on (010010)} & <: & c_2 \texttt{ on (0101)} \end{array} \right\}$$

Nous présentons ici une méthode de résolution des systèmes en forme ajustée. Cette méthode est correcte, mais sa complétude n'est à l'heure actuelle qu'une conjecture.

Comme les parties périodiques des deux mots p_x et p_y d'une même contrainte comprennent le même nombre de `1`, les contraintes de synchronisabilité pour les systèmes en forme ajustée se réduisent à :

$$c_x \texttt{ on } p_x \bowtie c_y \texttt{ on } p_y \Leftrightarrow \frac{|v_{c_x}|_1}{|v_{c_x}|} \times \frac{1}{|v_{p_x}|} = \frac{|v_{c_y}|_1}{|v_{c_y}|} \times \frac{1}{|v_{p_y}|}$$

Pour simplifier encore cette formule, on peut choisir $|v_{c_x}|_1 = |v_{p_x}|$ et $|v_{c_y}|_1 = |v_{p_y}|$. Ce choix est possible car tous les mots $p_x, p'_x, \ldots$ (resp. $p_y, p'_y, \ldots$) servant de condition d'échantillonnage à l'inconnue c_x (resp. c_y) ont une partie périodique de même taille. Ainsi cette simplification sera possible dans toutes les contraintes du système ($|v_{c_x}|_1 = |v_{p_x}| = |v_{p'_x}| = \ldots$ et $|v_{c_y}|_1 = |v_{p_y}| = |v_{p'_y}| = \ldots$). Les contraintes de synchronisabilité se résument alors à :

$$c_x \texttt{ on } p_x \bowtie c_y \texttt{ on } p_y \Leftrightarrow |v_{c_x}| = |v_{c_y}|$$

Remarque 5.2. Bien que nous n'ayons pas la preuve que ce choix pour le nombre de `1` des c_x et c_y ne mène jamais à un échec sur des systèmes ayant une solution, nous pensons que celui-ci est judicieux. En effet, comme les éléments des p_x et p_y sont parcourus au rythme des `1` dans les c_x et les c_y, si $|v_{c_x}|_1 = |v_{p_x}|$ et $|v_{c_y}|_1 = |v_{p_y}|$, on recommence le parcours de v_{p_x} en même temps que celui de v_{c_x} et le parcours de v_{p_y} en même temps que celui de v_{c_y}. Comme le v_{p_x} et le v_{p_y} d'une contrainte d'adaptabilité contiennent le même nombre de `1`, le comportement périodique des contraintes de précédence sur les indices des `1` de v_{c_x} et v_{c_y} est atteint dès le premier parcours de v_{p_x} et v_{p_y}. ◇

Ainsi, pour le système A'_1, si l'on choisit :

$$\begin{aligned}
|v_{c_1}|_1 &= |\texttt{(011)}| = |\texttt{(111)}| = 3\\
|v_{c_2}|_1 &= |\texttt{(1010)}| = |\texttt{(0101)}| = 4\\
|v_{c_3}|_1 &= |\texttt{(010101)}| = |\texttt{(010010)}| = 6
\end{aligned}$$

les contraintes de synchronisabilité se résument à $\{|v_{c_1}| = |v_{c_2}|;\ |v_{c_1}| = |v_{c_3}|;\ |v_{c_3}| = |v_{c_2}|\}$.

De la même manière, pour simplifier les formules de taille et de nombre de `1` des c_n `on` p_n données dans la section précédente, on choisit comme nombre de `1` des préfixes des c_n la taille des préfixes des p_n. Ce choix est possible car les préfixes de tous les mots $p_n, p'_n, \ldots$ servant de condition d'échantillonnage à une même inconnue c_n sont de même taille ($|u_{c_n}|_1 = |u_{p_n}| = |u_{p'_n}| = \ldots$). On obtient les valeurs suivantes :

$$|u_n| = |u_{c_n}| \qquad |u_n|_1 = |u_{p_n}|_1$$

$$|v_n| = \begin{cases} 1 & \text{si } |v_{p_n}| = 0 \\ |v_{c_n}| & \text{sinon} \end{cases} \qquad |v_n|_1 = |v_{p_n}|_1$$

Démonstration :

$$|v_n| = \begin{cases} 1 & \text{si } |v_{p_n}| = 0 \\ \frac{\text{ppcm}\,(|v_{p_n}|,|v_{p_n}|)}{|v_{c_n}|_1} \times |v_{c_n}| = |v_{c_n}| & \text{sinon} \end{cases}$$

$$|v_n|_1 = \frac{\text{ppcm}\,(|v_{p_n}|,|v_{p_n}|)}{|v_{p_n}|} \times |v_{p_n}|_1 = |v_{p_n}|_1$$

$$|u_n| = \begin{cases} |u_{c_n}| & \text{si } |u_{p_n}| = 0 \\ \max\,(|u_{c_n}|, \mathcal{I}_{c_n}(|u_{p_n}|)) = |u_{c_n}| & \text{sinon. En effet, } |u_{p_n}| = |u_{c_n}|_1 \text{ donc } \mathcal{I}_{c_n}(|u_{p_n}|) < |u_{c_n}|. \end{cases}$$

$$|u_n|_1 = \mathcal{O}_{p_n}(\mathcal{O}_{c_n}(|u_n|)) = \mathcal{O}_{p_n}(\mathcal{O}_{c_n}(|u_{c_n}|)) = \mathcal{O}_{p_n}(|u_{c_n}|_1) = \mathcal{O}_{p_n}(|u_{p_n}|) = |u_{p_n}|_1$$

□

Les contraintes de précédence deviennent donc :

$$c_x \texttt{ on } p_x \preceq c_y \texttt{ on } p_y \Leftrightarrow \forall j,\ 1 \leq j \leq |u_{p_x}|_1 + |v_{p_x}|_1,\ \mathcal{I}_{c_x}(\mathcal{I}_{p_x}(j)) \leq \mathcal{I}_{c_y}(\mathcal{I}_{p_y}(j))$$

Démonstration :

$c_x \texttt{ on } p_x \preceq c_y \texttt{ on } p_y$

$\Leftrightarrow \quad \forall j,\ 1 \leq j \leq \max(|u_x|_1, |u_y|_1) + \text{ ppcm } (|v_x|_1, |v_y|_1),\ \mathcal{I}_{c_x}(\mathcal{I}_{p_x}(j)) \leq \mathcal{I}_{c_y}(\mathcal{I}_{p_y}(j))$
$\Leftrightarrow \quad \forall j,\ 1 \leq j \leq \max(|u_{p_x}|_1, |u_{p_y}|_1) + \text{ ppcm } (|v_{p_x}|_1, |v_{p_y}|_1),\ \mathcal{I}_{c_x}(\mathcal{I}_{p_x}(j)) \leq \mathcal{I}_{c_y}(\mathcal{I}_{p_y}(j))$
$\Leftrightarrow \quad \forall j,\ 1 \leq j \leq |u_{p_x}|_1 + |v_{p_x}|_1,\ \mathcal{I}_{c_x}(\mathcal{I}_{p_x}(j)) \leq \mathcal{I}_{c_y}(\mathcal{I}_{p_y}(j))$

Car $|u_{p_x}|_1 = |u_{p_y}|_1$ et $|v_{p_x}|_1 = |v_{p_y}|_1$ □

Ces contraintes de précédence se décomposent en des contraintes sur les indices des `1` des préfixes et des contraintes sur les indices des `1` des motifs périodiques :

$$\begin{array}{rll} c_x \texttt{ on } p_x \preceq c_y \texttt{ on } p_y \Leftrightarrow & & \forall j,\ 1 \leq j \leq |u_{p_x}|_1,\ \mathcal{I}_{u_{c_x}}(\mathcal{I}_{u_{p_x}}(j)) \leq \mathcal{I}_{u_{c_y}}(\mathcal{I}_{u_{p_y}}(j)) \\ & \wedge & \forall j,\ 1 \leq j \leq |v_{p_x}|_1,\ |u_{c_x}| + \mathcal{I}_{v_{c_x}}(\mathcal{I}_{v_{p_x}}(j)) \leq |u_{c_y}| + \mathcal{I}_{v_{c_y}}(\mathcal{I}_{v_{p_y}}(j)) \end{array}$$

Ainsi, pour le système A'_1, comme $|u_{p_n}| = 0 = |u_{p_n}|_1$, on peut choisir des préfixes vides pour les c_n, et les contraintes de précédence deviennent :

$$\left\{ \begin{array}{l} \forall j,\ 1 \leq j \leq 2,\ \mathcal{I}_{v_{c_1}}(\mathcal{I}_{(011)}(j)) \leq \mathcal{I}_{v_{c_2}}(\mathcal{I}_{(1010)}(j)) \\ \forall j,\ 1 \leq j \leq 3,\ \mathcal{I}_{v_{c_1}}(\mathcal{I}_{(111)}(j)) \leq \mathcal{I}_{v_{c_3}}(\mathcal{I}_{(010101)}(j)) \\ \forall j,\ 1 \leq j \leq 2,\ \mathcal{I}_{v_{c_3}}(\mathcal{I}_{(010010)}(j)) \leq \mathcal{I}_{v_{c_2}}(\mathcal{I}_{(0101)}(j)) \end{array} \right\}$$

5.3.4 Résolution des contraintes

Les contraintes de précédence de A'_1 peuvent être détaillées en sept contraintes sur les indices des `1` dans les v_{c_n} :

$$\left\{ \begin{array}{lcl} \mathcal{I}_{v_{c_1}}(\mathcal{I}_{(011)}(1)) & \leq & \mathcal{I}_{v_{c_2}}(\mathcal{I}_{(1010)}(1)) \\ \mathcal{I}_{v_{c_1}}(\mathcal{I}_{(011)}(2)) & \leq & \mathcal{I}_{v_{c_2}}(\mathcal{I}_{(1010)}(2)) \\ \mathcal{I}_{v_{c_1}}(\mathcal{I}_{(111)}(1)) & \leq & \mathcal{I}_{v_{c_3}}(\mathcal{I}_{(010101)}(1)) \\ \mathcal{I}_{v_{c_1}}(\mathcal{I}_{(111)}(2)) & \leq & \mathcal{I}_{v_{c_3}}(\mathcal{I}_{(010101)}(2)) \\ \mathcal{I}_{v_{c_1}}(\mathcal{I}_{(111)}(3)) & \leq & \mathcal{I}_{v_{c_3}}(\mathcal{I}_{(010101)}(3)) \\ \mathcal{I}_{v_{c_3}}(\mathcal{I}_{(010010)}(1)) & \leq & \mathcal{I}_{v_{c_2}}(\mathcal{I}_{(0101)}(1)) \\ \mathcal{I}_{v_{c_3}}(\mathcal{I}_{(010010)}(2)) & \leq & \mathcal{I}_{v_{c_2}}(\mathcal{I}_{(0101)}(2)) \end{array} \right\}$$

Si l'on calcule les fonctions $\mathcal{I}$ sur les mots connus, on obtient un système d'inéquations linéaires

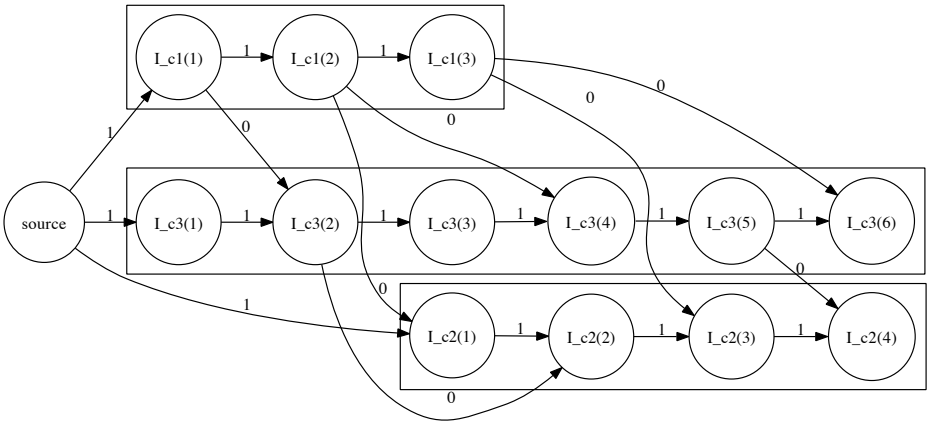


FIG. 5.1 – Graphe des contraintes sur les indices des 1. Le nœud I_c2(1) représente $\mathcal{I}_{v_{c_2}}(1)$. L'arc de poids 1 reliant ce nœud au nœud I_c2(2) indique que $\mathcal{I}_{v_{c_2}}(1)+1 \leq \mathcal{I}_{v_{c_2}}(2)$, c'est-à-dire que le premier 1 de v_{c_2} doit se situer strictement avant le deuxième 1 de v_{c_2}. L'arc de poids 0 qui relie I_c2(1) à I_c1(2) indique que le premier 1 de v_{c_2} doit arriver avant ou au même instant que le deuxième 1 de v_{c_1}.

sur les indices des 1 :

$$\left\{\begin{array}{lcl} \mathcal{I}_{v_{c_1}}(2) & \leq & \mathcal{I}_{v_{c_2}}(1) \\ \mathcal{I}_{v_{c_1}}(3) & \leq & \mathcal{I}_{v_{c_2}}(3) \\ \mathcal{I}_{v_{c_1}}(1) & \leq & \mathcal{I}_{v_{c_3}}(2) \\ \mathcal{I}_{v_{c_1}}(2) & \leq & \mathcal{I}_{v_{c_3}}(4) \\ \mathcal{I}_{v_{c_1}}(3) & \leq & \mathcal{I}_{v_{c_3}}(6) \\ \mathcal{I}_{v_{c_3}}(2) & \leq & \mathcal{I}_{v_{c_2}}(2) \\ \mathcal{I}_{v_{c_3}}(5) & \leq & \mathcal{I}_{v_{c_2}}(4) \end{array}\right\}$$

Par exemple, la première contrainte se lit "l'indice du 2e 1 dans v_{c_1} doit être inférieur ou égal à l'indice du 1er 1 dans v_{c_2}". Ce système doit être complété par des contraintes imposant qu'au sein d'un c_n, l'indice de chaque 1 précède strictement l'indice du 1 suivant. Dans notre exemple, des contraintes doivent donc être rajoutées entre les trois 1 de v_{c_1} ($\mathcal{I}_{v_{c_1}}(1)+1 \leq \mathcal{I}_{v_{c_1}}(2)$ et $\mathcal{I}_{v_{c_1}}(2)+1 \leq \mathcal{I}_{v_{c_1}}(3)$), les quatre 1 de v_{c_2} et les six 1 de v_{c_3}.

La résolution du système ainsi obtenu donne des indices corrects pour les 1 dans les c_n. Les inéquations de ce système sont un cas particulier d'inéquations linéaires, de la forme $x - y \leq n$ avec $n \in \mathbb{Z}$. Elles peuvent être exprimées sous la forme d'une *Difference Bound Matrix* [Dil90], qui peut être résolue aisément par l'algorithme de Bellman-Ford [CLR90]. Les contraintes sont exprimées dans un graphe où les nœuds représentent les indices des 1 à trouver et les poids des arcs la différence entre les valeurs des nœuds reliés. On ajoute un nœud source relié par un arc de poids un à tous les nœuds qui représentent le premier 1 d'un mot (car l'indice du premier 1 est supérieur ou égal à un). La résolution se résume alors à la recherche des plus longs chemins depuis ce nœud source vers chacun des nœuds. Les résultats obtenus minimisent les indices des 1 des mots inférés. Le graphe représentant les contraintes de l'exemple est donné dans la figure 5.1.

On trouve la solution suivante au système d'inéquations linéaires : [2]

$$\begin{array}{llllll}\mathcal{I}_{v_{c_1}}(1)=1, & \mathcal{I}_{v_{c_1}}(2)=2, & \mathcal{I}_{v_{c_1}}(3)=3 & & & \\ \mathcal{I}_{v_{c_2}}(1)=2, & \mathcal{I}_{v_{c_2}}(2)=3, & \mathcal{I}_{v_{c_2}}(3)=4, & \mathcal{I}_{v_{c_2}}(4)=5 & & \\ \mathcal{I}_{v_{c_3}}(1)=1, & \mathcal{I}_{v_{c_3}}(2)=2, & \mathcal{I}_{v_{c_3}}(3)=3, & \mathcal{I}_{v_{c_3}}(4)=4, & \mathcal{I}_{v_{c_3}}(5)=5, & \mathcal{I}_{v_{c_3}}(6)=6\end{array}$$

Nous disposons maintenant du nombre de `1` des parties périodiques des c_n ainsi que de l'indice de ces `1`, afin que les contraintes de précédence soient vérifiées. Il ne manque plus qu'à trouver une taille pour ces parties périodiques, telle que les contraintes de synchronisabilité soient vérifiées. Dans la section 5.3.3, nous avons réduit les contraintes de synchronisabilité au système suivant : $\{|v_{c_1}| = |v_{c_2}|;\ |v_{c_1}| = |v_{c_3}|;\ |v_{c_3}| = |v_{c_2}|\}$. Comme le système est connexe, toutes les tailles doivent être égales. Il suffit de choisir comme taille le plus grand indice des `1` de l'ensemble des mots, c'est-à-dire 6.

On obtient donc comme solution au système de contraintes d'adaptabilité :

$$c_1 = \texttt{(111000)}, \quad c_2 = \texttt{(011110)}, \quad c_3 = \texttt{(111111)}$$

Si l'on instancie c_1, c_2, c_3 par leurs valeurs dans A'_1, on obtient :

$$A'_1 = \left\{ \begin{array}{ll} \texttt{(111000) } \textit{on} \texttt{ (011)} & <: \texttt{(011110) } \textit{on} \texttt{ (1010)} \\ \texttt{(111000) } \textit{on} \texttt{ (111)} & <: \texttt{(111111) } \textit{on} \texttt{ (010101)} \\ \texttt{(111111) } \textit{on} \texttt{ (010010)} & <: \texttt{(011110) } \textit{on} \texttt{ (0101)} \end{array} \right\} = \left\{ \begin{array}{l} \texttt{(011000)} <: \texttt{(010100)} \\ \texttt{(111000)} <: \texttt{(010101)} \\ \texttt{(010010)} <: \texttt{(001010)} \end{array} \right\}$$

Les contraintes sont satisfaites.

L'exemple que l'on vient de traiter ne contient que des mots sans préfixes. Pour résoudre un système de contraintes sur des mots avec préfixes, il faut trouver les indices des `1` dans les préfixes en utilisant la même méthode que pour les parties périodiques. Contrairement au cas des motifs périodiques, les tailles de préfixe des mots inférés n'ont pas besoin d'être égales car le taux de `1` dans le préfixe n'influence pas la synchronisabilité. Par conséquent, il suffit de choisir pour chaque préfixe l'indice de son dernier `1` comme taille. Il faut ensuite prendre en compte la taille du préfixe lorsque l'on pose les contraintes sur les indices des `1` du motif périodique.

5.3.5 Retour sur la mise en forme ajustée

Nous venons de voir un algorithme permettant de résoudre les systèmes de contraintes d'adaptabilité qui sont en forme ajustée.

La mise en forme ajustée des parties périodiques d'un système peut être réalisée en exprimant les contraintes d'ajustement dans un système d'équations linéaires, et de résoudre ce système à l'aide d'un solveur de contraintes externe.

[2] L'outil Glpk [Glp] trouve la même solution lorsque la fonction objectif est la minimisation de la somme des indices des `1` (voir l'annexe F).

Par exemple, considérons le système A_1 sous sa forme initiale (c'est-à-dire non ajustée) :

$$A_1 = \left\{ \begin{array}{lcl} c_1 \texttt{ on (011)} & <: & c_2 \texttt{ on (10)} \\ c_1 \texttt{ on (1)} & <: & c_3 \texttt{ on (01)} \\ c_3 \texttt{ on (010)} & <: & c_2 \texttt{ on (01)} \end{array} \right\}$$

Ajuster les parties périodiques du système consiste à le mettre sous la forme

$$\left\{c_1 \texttt{ on } p_1 <: c_2 \texttt{ on } p_2;\ \ c_1 \texttt{ on } p'_1 <: c_3 \texttt{ on } p_3;\ \ c_3 \texttt{ on } p'_3 <: c_2 \texttt{ on } p'_2\right\}$$

avec $\texttt{(011)} = p_1$, $\texttt{(1)} = p'_1$, $\texttt{(10)} = p_2$, $\texttt{(01)} = p'_2$, $\texttt{(01)} = p_3$, $\texttt{(010)} = p'_3$, et :

1. $|p_1|_1 = |p_2|_1$, $|p'_1|_1 = |p_3|_1$, $|p'_3|_1 = |p'_2|_1$
2. $|p_1| = |p'_1|$, $|p_2| = |p'_2|$, $|p_3| = |p'_3|$.

Les tailles et nombre de `1` des parties périodiques des mots initiaux de A_1 peuvent être augmentés par répétition du motif périodique. Par conséquent, la taille et le nombre de `1` obtenus sont un multiple de la taille et du nombre de `1` initiaux. On cherche donc des facteurs entiers k_1, k'_1, k_2, k'_2, k_3, k'_3 strictement positifs tels que :

$$\begin{array}{lcllcl} |p_1| & = & k_1 \times |\texttt{(011)}| = k_1 \times 3 & |p_1|_1 & = & k_1 \times |\texttt{(011)}|_1 = k_1 \times 2 \\ |p'_1| & = & k'_1 \times |\texttt{(1)}| = k'_1 \times 1 & |p'_1|_1 & = & k'_1 \times |\texttt{(1)}|_1 = k'_1 \times 1 \\ |p_2| & = & k_2 \times |\texttt{(10)}| = k_2 \times 2 & |p_2|_1 & = & k_2 \times |\texttt{(10)}|_1 = k_2 \times 1 \\ |p'_2| & = & k'_2 \times |\texttt{(01)}| = k'_2 \times 2 & |p'_2|_1 & = & k'_2 \times |\texttt{(01)}|_1 = k'_2 \times 1 \\ |p_3| & = & k_3 \times |\texttt{(01)}| = k_3 \times 2 & |p_3|_1 & = & k_3 \times |\texttt{(01)}|_1 = k_3 \times 1 \\ |p'_3| & = & k'_3 \times |\texttt{(010)}| = k'_3 \times 3 & |p'_3|_1 & = & k'_3 \times |\texttt{(010)}|_1 = k'_3 \times 1 \end{array}$$

et tels que les contraintes 1. et 2. données ci-dessus sont vérifiées, c'est-à-dire tels que :

1. $k_1 \times 2 = k_2$, $k'_1 = k_3$, $k'_3 = k'_2$
2. $k_1 \times 3 = k'_1$, $k_2 \times 2 = k'_2 \times 2$, $k_3 \times 2 = k'_3 \times 3$.

Ces contraintes constituent un ensemble d'équations linéaires homogènes que l'on peut résoudre par des méthodes classiques. Comme on souhaite déplier le moins possible les mots périodiques, on utilise comme objectif la minimisation de la somme des inconnues. La solution trouvée est alors : $k_1 = 1$, $k'_1 = 3$, $k_2 = 2$, $k'_2 = 2$, $k_3 = 3$, $k'_3 = 2$ (voir l'annexe F).

Par conséquent, il faut répéter 1 fois le motif de `(011)` pour obtenir p_1, 3 fois le motif de `(1)` pour obtenir p'_1, 2 fois le motif de `(10)` pour obtenir p_2, et ainsi de suite. Le système A_1 ajusté est donc le suivant, comme nous avons vu page 103 :

$$A'_1 = \left\{ \begin{array}{lcl} c_1 \texttt{ on (011)} & <: & c_2 \texttt{ on (1010)} \\ c_1 \texttt{ on (111)} & <: & c_3 \texttt{ on (010101)} \\ c_3 \texttt{ on (010010)} & <: & c_2 \texttt{ on (0101)} \end{array} \right\}$$

Nous pensons que si le système de contraintes d'ajustement des parties périodiques n'a pas de solution, c'est que les contraintes de synchronisabilité n'ont pas de solution. Le système est donc rejeté dans ces cas. Nous n'avons cependant pas la preuve de cette affirmation.

En ce qui concerne l'ajustement de la taille des préfixes, nous n'avons pour le moment qu'un algorithme itératif, qui augmente les tailles des préfixes, alternativement pour respecter la première contrainte d'ajustement, puis pour respecter la seconde, puis pour rétablir

la première, jusqu'à l'obtention d'un système en forme ajustée. Cet algorithme n'est pas satisfaisant. Nous n'avons même pas la garantie que les préfixes de tout système ayant une solution peuvent être mis dans une forme ajustée. On peut donc considérer que la résolution des contraintes de sous-typage que nous fournissons ne fonctionne que sur des systèmes dont les préfixes sont déjà en forme ajustée, en particulier :

- les systèmes sans préfixes ;
- les systèmes avec des préfixes composés uniquement de `0`, en nombre égal sur tous les mots échantillonnant une même variable ;
- les systèmes qui ne contiennent qu'une contrainte.

5.4 Conclusion

Le système de types utilisant la résolution des contraintes sur les horloges périodiques a été implanté et a été utilisé pour typer les exemples du chapitre 2 (dont les types sont fournis dans l'annexe B), ainsi que pour les exemples d'applications plus conséquentes présentées dans le prochain chapitre. Ces applications illustrent l'utilité des buffers, ainsi que les forces et les faiblesses des algorithmes de résolution des contraintes d'égalité et de sous-typage que nous venons de présenter. Il serviront de référence pour la comparaison avec la méthode de résolution présentée dans la partie III.

Chapitre 6

Applications et discussion

Nous présentons dans ce chapitre deux exemples d'applications programmées en Lucy-n avec le langage d'horloges périodiques. La première est issue du protocole de communication GSM pour la téléphonie mobile. La seconde est issue du traitement vidéo dans les télévisions haute définition. Ces exemples illustrent l'utilité de l'opérateur de bufferisation, et servent de cas d'étude pour évaluer les algorithmes de résolution des contraintes d'égalité et de sous-typage sur les types d'horloges périodiques. Nous terminons par une discussion sur ces algorithmes.

6.1 Exemple d'un encodeur de canal GSM

Dans le protocole GSM [LGT00], la voix est divisée en échantillons de 20ms, qui sont chacun numérisés par une trame de 260 bits. Ces trames sont ensuite encodées, afin de pouvoir détecter et éventuellement corriger les erreurs qui pourraient les atteindre lors de la transmission radio. C'est à cet encodage que nous nous intéressons ici. Les bits d'une trame sont divisés en trois classes, selon l'importance qu'ils ont dans la qualité de la voix numérisée :

Classe Ia : les 50 premiers bits qui sont les bits les plus importants,

Classe Ib : les 132 bits suivants qui sont modérément importants,

Classe II : les 78 derniers bits qui sont les moins importants.

On applique aux 50 bits de classe Ia un encodeur cyclique qui rajoute un code de redondance cyclique de 3 bits. Les 53 bits obtenus sont concaténés aux 132 bits de classe Ib et à quatre autres bits (dont la valeur ne sera pas utilisée), pour construire un paquet de 189 bits, auquel on applique un encodeur convolutionnel. L'encodeur produit deux bits de sortie pour chaque bit d'entrée, calculés par combinaison des 4 bits d'entrée précédents. Enfin, on concatène les 378 bits ainsi obtenus avec les 78 bits de classe II qui ne seront pas encodés. L'encodeur est représenté graphiquement sur la figure 6.1.

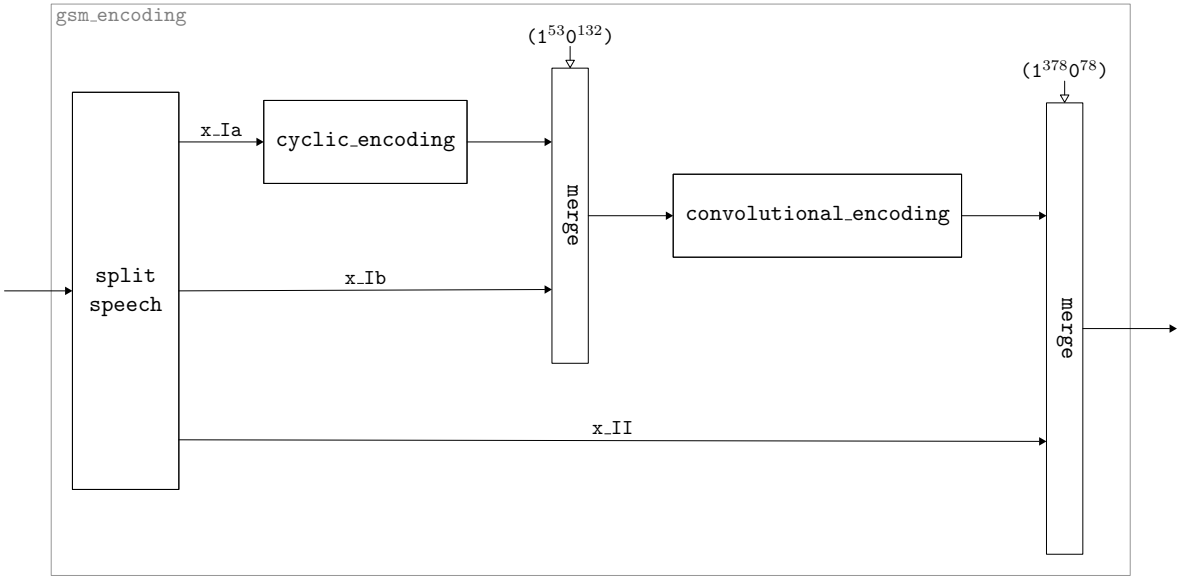


FIG. 6.1 – Encodeur de canal GSM

6.1.1 Programmation des nœuds de l'encodeur

Séparation des trames en trois parties

Le nœud `split_speech` sépare chaque trame de voix numérisée en trois sous-trames contenant les bits de classe Ia, Ib et II :

```
14 let node split_speech x = (x_Ia, x_Ib, x_II) where
15   rec x_Ia = x when (1^50 0^210)
16   and x_Ib = x when (0^50 1^132 0^78)
17   and x_II = x when (0^182 1^78)
```

Son type d'horloges est donc :[1]

```
split_speech ::
forall 'a. 'a ->
   ('a on (1^50 0^210 ) * 'a on (0^50 1^132 0^78 ) * 'a on (0^182 1^78 ))
```

Quelle que soit l'horloge du flot d'entrée, `split_speech` renvoie un triplet de flots contenant respectivement les 50 premières valeurs de chaque trame, les 132 valeurs suivantes et les 78 dernières valeurs. Les instants de présence de ces trois flots correspondent donc aux instants d'arrivée des trois parties de chaque trame.

[1] Nous indiquons la sortie fournie par notre typeur.
Dans la notation mathématique utilisée jusqu'alors, le type affiché pour `split_speech` correspond à :
$$split\_speech : \forall\alpha.\, \alpha \rightarrow (\alpha \texttt{ on } (1^{50}0^{210}) \times \alpha \texttt{ on } (0^{50}1^{132}0^{78}) \times \alpha \texttt{ on } (0^{182}1^{78}))$$

Encodage cyclique

Le nœud `cycling_encoding` concatène la sous-trame de classe Ia avec trois bits de redondance calculés en fonction des 50 bits de celle-ci :

```
22 let node join_50_3 (x, r0, r1, r2) = x012 where
23   rec x0 = merge (1^50 0) x r0
24   and x01 = merge (1^51 0) x0 r1
25   and x012 = merge (1^52 0) x01 r2

28 let node cyclic_encoding x = y where
29   rec u0 = false fby (xor(x, u2))
30   and u1 = false fby (xor(u0, u2))
31   and u2 = false fby u1
32   and y = join_50_3
33       ( x,
34         buffer(u0 when (0^49 1)),
35         buffer(u1 when (0^49 1)),
36         buffer(u2 when (0^49 1)) )
```

Lorsque la cinquantième valeur de la trame d'entrée arrive, les flots `u0`, `u1` et `u2` contiennent les bits de redondance. Ces bits de redondance sont donc obtenus en échantillonnant les flots `u0`, `u1` et `u2` avec l'horloge `(0^49 1)`, et conservés dans un buffer jusqu'à l'instant où ils doivent être insérés dans le flot de sortie. Le nœud `join_50_3` concatène les bits de classe Ia avec les 3 bits ainsi obtenus. Les types d'horloges sont les suivants :[2]

```
join_50_3 ::
   forall 'a. ( 'a on (1^52 0) on (1^51 0) on (1^50 0)
              * 'a on (1^52 0) on (1^51 0) on not (1^50 0)
              * 'a on (1^52 0) on not (1^51 0)
              * 'a on not (1^52 0))
             -> 'a

cyclic_encoding :: forall 'a. 'a on (1^52 0) on (1^51 0) on (1^50 0) -> 'a
Buffer line 34, characters 1-25: size = 1
Buffer line 35, characters 1-25: size = 1
Buffer line 36, characters 1-25: size = 1
```

Si l'on simplifie les expressions de ces types, on obtient :

$$\textit{join\_50\_3} : \forall\alpha.\,(\alpha \texttt{ on } (1^{50}000) * \alpha \texttt{ on } (0^{50}100) * \alpha \texttt{ on } (0^{50}010) * \alpha \texttt{ on } (0^{50}001)) \to \alpha$$
$$\textit{cyclic\_encoding} : \forall\alpha.\,\alpha \texttt{ on } (1^{50}0^3) \to \alpha$$

La partie 1^{50} correspond à la consommation des 50 bits de classe Ia de la trame, et la partie 0^3 corresponds aux instants où le nœud ne consomme rien car il est en train de produire les bits de redondance en sortie.

Remarque 6.1. Nous avons fait le choix de ne pas calculer les expressions d'horloges dans les types inférés pour plusieurs raisons. D'une part, le fait de conserver la structure des expressions peut permettre d'effectuer des simplifications structurelles dans les contraintes d'égalité ou de sous-typage apparaissant lors de l'instanciation du nœud. Il est toujours possible de calculer les expressions d'horloges à posteriori si aucune simplification n'est possible. D'autre part, la structure de l'expression aide souvent à comprendre la provenance des différentes horloges

[2] Notre typeur affiche aussi optionnellement les tailles des buffers.

étant impliquées dans le type. Si ce n'est pas le cas, il est toujours possible pour l'utilisateur de calculer ces expressions à l'aide des outils fournis avec le typeur. Enfin ce choix a aussi été réalisé pour des questions d'affichage. L'écriture de l'horloge résultat d'une expression est souvent moins compacte que l'expression elle même (car les `1` sont moins rassemblés après échantillonnage). Par exemple, $(1^{1000}0)$ *on* $(10) = (\underbrace{101010\ldots1010}_{1000}\ 0)$. ◇

Le nœud `cyclic_encoding` peut être écrit sans l'aide de l'opérateur de bufferisation, de la manière suivante :

```
let node current_50_3 x = y where
  rec y = merge (1^50 0^3) x ((false fby y) whenot (1^50 0^3))

let node cyclic_encoding x = y where
  rec u0 = false fby (xor(x, u2))
  and u1 = false fby (xor(u0, u2))
  and u2 = false fby u1
  and y = join_50_3
      ( x,
        (current_50_3 u0) when (0^50 100),
        (current_50_3 u1) when (0^50 010),
        (current_50_3 u1) when (0^50 001) )
```

Les valeurs des bits de redondance sont significatives simultanément à l'arrivée du $50^{ième}$ bit de chaque trame. Il faut ici maintenir la $50^{ième}$ valeur des flots `u0`, `u1` et `u2` durant les instants suivants (c'est le rôle du nœud `current_50_3`), puis échantillonner chacun de ces flots. Ainsi, chacune des trois valeurs de redondance à insérer dans la trame est présente uniquement et exactement à l'instant où l'on désire l'insérer. Cette version sans buffers du code, bien que presque aussi concise que la première, présente l'inconvénient d'être plus éloignée de la description fonctionnelle du nœud. En effet, le flot que l'on construit en maintenant la dernière valeur des flots de redondance n'a pas de sens, il sert juste à contourner la restriction imposée par la composition synchrone. Ce style de programmation est donc plus fastidieux et source d'erreurs.

Remarquons que cette version du code n'est bien typée que si l'on dispose d'une unification interprétant les valeurs des horloges. En effet, comme les flots `u0`, `u1` et `u2` sont passés argument au nœud `current_50_3`, ils sont de type α `on` $(1^{50}0^3)$. Comme le flot `x` est composé de manière synchrone avec ces trois flots, il est aussi de type α `on` $(1^{50}0^3)$. Un calcul d'horloge avec unification structurelle échoue, car `x` est aussi passé en argument au nœud `join_50_3` qui attend un flot de type α' `on` $(1^{52}0)$ `on` $(1^{51}0)$ `on` $(1^{50}0)$. Ces deux types ne sont pas unifiables structurellement, bien que la simple substitution $\theta(\alpha') = \alpha$ les rende égaux. Mis à part l'algorithme d'unification structurelle, tous les algorithmes présentés dans la section 5.2 typent avec succès toutes les déclarations effectuées ci-dessus, et calculent les mêmes types résultats.

Encodage convolutionnel

L'encodeur convolutionnel ajoute quatre bits non significatifs au flot d'entrée, calcule deux flots à partir de celui-ci, et produit en sortie la fusion de ces deux flots.

```
41 let node multiplexer (x1, x2) = o where
42   rec o = merge (10) x1 x2
```

```
45 let node convolutional_encoding x = y where
46   rec x' = merge (1^185 0^4) x false
47   and u1 = false fby x'
48   and u2 = false fby u1
49   and u3 = false fby u2
50   and u4 = false fby u3
51   and x1 = xor (xor (xor (x', u1), u3), u4)
52   and x2 = xor (xor (x', u3), u4)
53   and y = multiplexer (x1,buffer(x2))
```

Les types d'horloges sont donc :

```
multiplexer :: forall 'a. ('a on (10) * 'a on not (10)) -> 'a

convolutional_encoding :: forall 'a. 'a on (10) on (1^185 0^4 ) -> 'a
Buffer line 53, characters 26-36: size = 1
```

La partie `(10)` du type de `convolutional_encoding` dénote du fait que pour chaque entrée consommée, deux sorties sont produites. Les entrées doivent donc être consommées un instant sur deux. La partie `(1^185 0^4 )` dénote du fait qu'après chaque paquet de (50 + 3 + 132) bits, l'encodeur convolutionnel ajoute quatre bits non significatifs. Aucune entrée n'est consommée durant cet ajout.

Ici aussi, on aurait pu programmer ce nœud sans utiliser l'opérateur de bufferisation :

```
let node convolutional_encoding x = y where
  rec x' = merge (1^185 0^4) x false
  and u1 = false fby x'
  and u2 = false fby u1
  and u3 = false fby u2
  and u4 = false fby u3
  and x1 = xor (xor (xor (x', u1), u3), u4) (* x^4 + x^3 + x + 1 *)
  and x2 = xor (xor (x', u3), u4) (* x^4 + x^3 + 1 *)
  and delayed_x2 = (stutter x2) whenot (10)
  and y = multiplexer (x1, delayed_x2)
```

Pour cela, il faut faire en sorte que les valeurs du flot `x2` soient disponibles aux rang impairs au lieu des rangs pairs, afin que le nœud `multiplexer` qui fusionne de `x1` et `x2` reçoive ces flots exactement aux instants où il les utilise. Ce décalage est construit en faisant bégayer `x2` puis en échantillonnant sur les instants impairs. Ici aussi, la version sans buffer est plus éloignée de la spécification fonctionnelle : il n'est pas immédiat de prouver que la suite des valeurs prises par `delayed_x2` est identique à la suite des valeurs prises par `x2`.

6.1.2 Connexion des nœuds de l'encodeur

L'encodeur de canal GSM représenté sur la figure 6.1 peut être programmé en connectant les nœuds précédemment définis.

```
60 let node encode_without_buffer x = y where
61   rec (x_Ia, x_Ib, x_II) = split_speech x
62   and y1 = cyclic_encoding x_Ia
63   and y1_x_Ib = merge (1^53 0^132) y1 x_Ib
64   and y2 = convolutional_encoding y1_x_Ib
65   and y = merge (1^378 0^78) y2 x_II
```

L'unification structurelle échoue, car le type d'horloges du flot `x_Ia` n'est pas structurellement unifiable avec le type d'horloges d'entrée du nœud `cycling_encoding` :

```
line 62, characters 27-31:
Cannot unify clock 'a9 on (1^50 0^210 )
with clock 'a10 on (1^52 0) on (1^51 0) on (1^50 0)
```

L'unification semi-interprétée échoue elle aussi, car le choix de l'unificateur effectué lors de l'unification du type d'horloges du flot `x_Ia` avec le type d'entrée du nœud `cycling_encoding` rend fausse la contrainte d'égalité entre le type d'horloges du flot `x_Ib` et le type d'horloges de la deuxième branche du `merge`.

```
line 63, characters 38-42:
Cannot unify clock 'a13 on (0^50 1^132 0^78 )
with clock 'a13 on (1^185 0^75 ) on not (1^53 0^132 )
```

L'unification globale trouve une substitution permettant de satisfaire toutes les contraintes d'égalité. Le type d'horloges ainsi obtenu pour le nœud `encode_without_buffer` est :

```
encode_without_buffer ::
   forall 'a. 'a on ( 10101010101010101010101010101010101010101010101010
                      10101010101010101010101010101010101010101010101010^7
                      10101010101010101010101010101010101010101010101010
                      10101010101010101010101010101010101010101010101010
                      10101010101010101010101010101010101010101010101010
                      10101010101010101010101010101010101010101010101010
                      10101010101010101010101010101010101010101010101010
                      10101010101010^9
                      1^78 )
              -> 'a
```

Si l'on note $\{v\}^x$ la concaténation de x fois le mot v, ce type est égal à :

$$\forall \alpha.\, \alpha \texttt{ on } (\{10\}^{50}0^6\{10\}^{132}0^81^{78}) \rightarrow \alpha$$

Les 50 bits de classe Ia sont consommés un instant sur deux, car l'encodeur convolutionnel consomme ses entrées un instant sur deux. Aucune entrée n'est ensuite consommée pendant six instants, pendant que les trois bits de redondance produits par l'encodeur cyclique sont consommés (un instant sur deux) par l'encodeur convolutionnel. Suivant le même principe, les 132 bits de classe Ib sont consommés au rythme d'un instant sur deux, et aucune entrée n'est consommée durant les 8 instants suivants, pendant le traitement des quatre bits ajoutés par l'encodeur convolutionnel. Enfin, les 78 bits de classe II qui ne subissent aucun traitement sont consommés à raison d'un bit à chaque instant. Une valeur est produite en sortie à chaque instant. Le comportement que nous venons de décrire se reproduit périodiquement à l'infini.

6.2 Décodeur de canal GSM

Le décodeur associé à l'encodeur de canal GSM que nous venons de présenter est composé d'un décodeur convolutionnel et d'un décodeur cyclique. Il est représenté sur la figure 6.2.

6.2.1 Programmation des nœuds du décodeur

Le décodeur convolutionnel est défini par l'algorithme de Viterbi [LGT00] que nous n'avons pas programmé. Nous l'avons remplacé pour nos expériences par un nœud ayant le même type

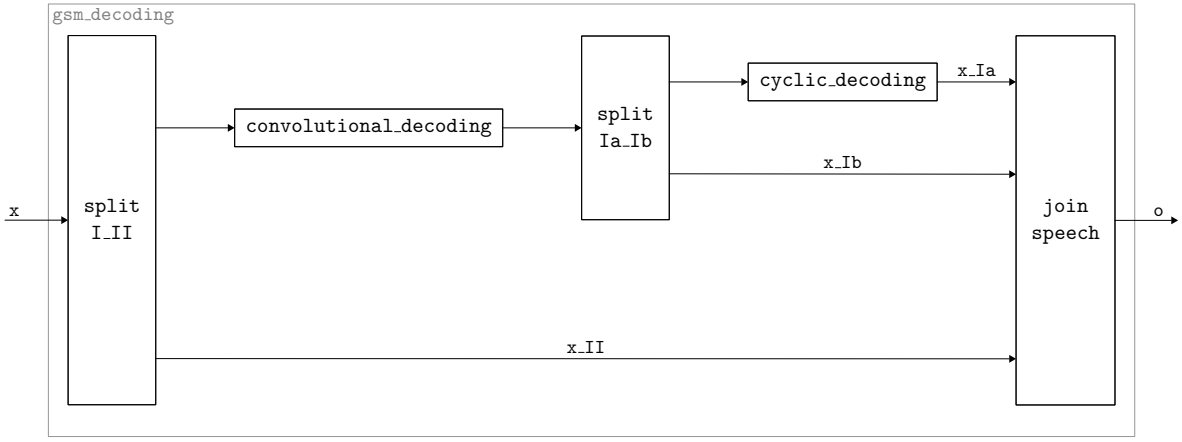


FIG. 6.2 – Décodeur de canal GSM

d'horloges. Le décodeur cyclique est programmé de la manière suivante et représenté sur la figure 6.3 :

```
56 let node cyclic_decoding x = o where
57   rec (data, redondancy) =
58     (x when (1^50 0^3) , x whenot (1^50 0^3))
59   and reencoded_data = cyclic_encoding data
60   and code = reencoded_data whenot (1^50 0^3)
61   and ok = (code = redondancy)
62   and frame_ok = and_by_3 ok
63   and data_ok = stutter50 frame_ok
64   and current_frame = buffer(data)
65   and last_frame = merge 1^50(0) true (buffer(current_frame))
66   and o = if data_ok then current_frame else last_frame
```

Les buffers sont indispensables pour programmer ce nœud. La sous-trame d'entrée est décomposée en deux flots : `data` contient les 50 bits de classe Ia, et `redondancy` contient les 3 bits de redondance. L'encodeur cyclique est appliqué aux bits de données, afin de comparer le code obtenu avec les trois bits de redondance. Si les trois bits du code sont égaux aux trois bits de redondance, la trame n'a pas été altérée et les bits de donnée sont produits en sortie. Sinon, c'est que la trame a été altérée. Dans ce cas, c'est la trame précédente qui est produite en sortie. Pendant le réencodage des données, la trame courante doit être conservée dans un buffer en attendant la décision de la produire en sortie ou non. La trame précédente doit elle aussi être conservée dans un buffer au cas où la trame courante serait erronée. La taille inférée pour chacun de ces buffers est 50 (c'est le nombre de bits dans une trame), et le type obtenu en utilisant l'unification interprétée ou semi-interprétée est :

```
cyclic_decoding ::
forall 'a. 'a -> 'a on 0^52 (1^50 0^3 )
Buffer line 64, characters 22-34: size = 50
Buffer line 65, characters 39-60: size = 50
```

Le préfixe 0^{52} du type de sortie rend compte du fait qu'il faut attendre les 50 bits de la première trame plus les trois bits de redondance pour vérifier qu'il n'y a pas eu d'erreur de transmission avant de commencer à produire des bits en sortie. Après cette phase d'initialisation le décodeur produit des trames en continu, avec des interruptions de trois instants puisque les bits de

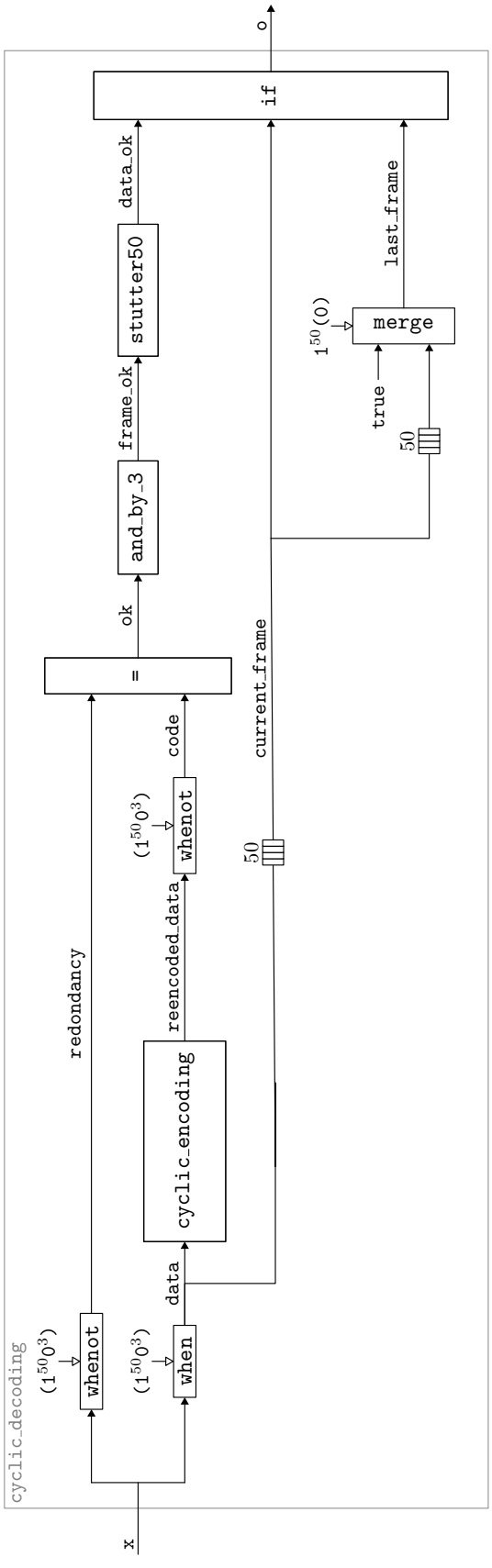


FIG. 6.3 – Décodeur cyclique

redondance sont supprimés du flot. L'unification globale ne produit pas un type aussi rapide, le type inféré est :

```
cyclic_decoding ::
forall 'a. 'a on (1^53 0^49 ) -> 'a on (0^52 1^50 )
Buffer line 64, characters 22-34: size = 50
Buffer line 65, characters 39-60: size = 50
```

Ici, le nœud attend d'avoir fini de produire une trame en sortie avant de commencer à consommer la trame suivante en entrée. Ce type aboutit à une exécution beaucoup moins rapide. Ce défaut est lié au fait que si les types à unifier sont exprimés en fonction d'horloges sans préfixes, alors les unificateurs inférés sont sans préfixes. Ils ne font donc pas forcément partie des unificateurs menant au rythme commun le plus rapide.

Remarque 6.2. Une première implémentation de ce nœud avait conduit à une taille minimale de 100 pour le buffer de la ligne 65. En effet, nous avions défini le flot de l'image précédente par l'équation suivante :

```
65      and last_frame = merge 1^50(0) true (buffer(data))
```

Graphiquement parlant (voir la figure 6.3), cela revient à brancher l'entrée de ce buffer sur le fil `data` plutôt que sur la sortie du buffer produisant la trame courante (`current_frame`). Deux trames devaient donc y être stockées : la trame courante en tant que prochaine trame précédente, et la trame précédente. Le compilateur permettant d'afficher les tailles de buffer inférées, nous avons pu nous rendre compte rapidement de cette inefficacité et avons pu la corriger facilement, en utilisant le buffer stockant la trame courante comme entrée du buffer stockant la trame précédente. ◇

Le système de contraintes généré par le typage de ce nœud (sans unification globale) est un exemple de système où l'étape de simplification décrite en section 5.3.1 aboutit à un système de contraintes vide :[3]

```
Constraints to simplify:
 { 'a20 on 0^52 (1^50 0^3 ) <: 'a20 on 0^52 (1^50 0^3 ) on not 1^50 (0)
     (* line 65, characters 39-60 *)
   'a20 on (1^50 0^3 ) <: 'a20 on 0^52 (1^50 0^3 )
     (* line 64, characters 22-34 *) }
No constraint to solve.
```

En effet, ici, les contraintes portent chacune sur une seule variable, donc il s'agit de vérifier que les contraintes sont satisfaites, il n'y a pas de contrainte à résoudre par substitution des variables.

6.2.2 Connexion des nœuds du décodeur

Le décodeur peut être programmé en connectant les nœuds précédemment définis. Malheureusement, les contraintes de types contenant des horloges avec préfixe, notre algorithme échoue, car il n'arrive pas à mettre les préfixes du système sous forme ajustée. On rencontre ici le problème que nous avons décrit dans la section 5.3.5.

[3] L'affichage des contraintes de sous-typage est disponible en option de notre typeur.

6.3 Exemple du *Picture In Picture*

L'application *Picture in Picture* permet d'incruster une image en petit format au sein d'une image en grand format, pour pouvoir visualiser deux vidéos simultanément. Cette application reçoit deux images en haute définition (HD), réduit la taille de la première vers une petite définition (SD), supprime une zone de pixels de la seconde et y incruste la première image. Elle est représentée sur la figure 6.4.

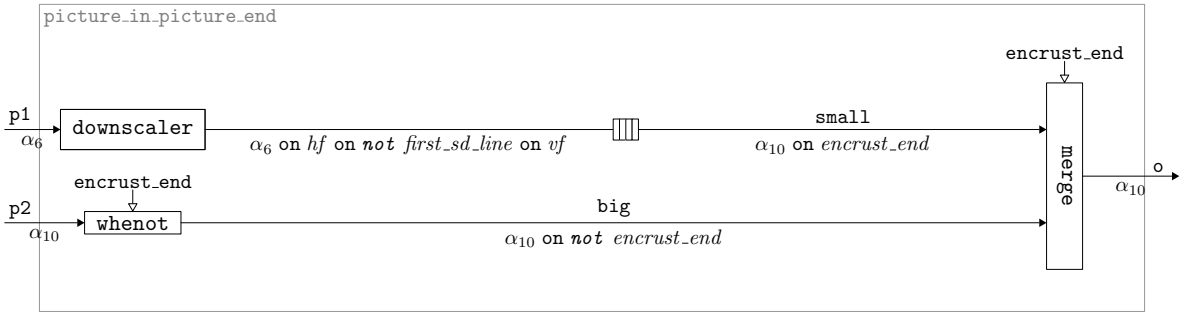


FIG. 6.4 – Le *Picture in Picture*

Le nœud `downscaler` réduit la taille de l'image. Son type est :

$$\forall\alpha.\, \alpha \to \alpha \texttt{ on } hf \texttt{ on not } \mathit{first\_sd\_line} \texttt{ on } vf$$

avec $hf = \texttt{(10100100)}$, $\mathit{first\_sd\_line} = \texttt{1}^{720}\texttt{(0)}$ et $vf = \texttt{(1}^{720}\texttt{0}^{720}\texttt{1}^{720}\texttt{0}^{720}\texttt{0}^{720}\texttt{1}^{720}\texttt{0}^{720}\texttt{0}^{720}\texttt{1}^{720}\texttt{)}$. La première entrée du nœud `merge` doit avoir un type de la forme α_2 `on` *encrust_end* avec *encrust_end* l'horloge qui définit les instants où c'est la petite image qui doit être affichée.

Le système de contraintes obtenu est :

```
picture_in_picture_end ::
forall 'a1, 'a2. ('a1 * 'a2) -> 'a2
with
 { 'a1 on hf on not first_sd_line on vf <: 'a2 on encrust_end
     (* line 56, characters 14-35 *) }
```

Ce système, bien que composé d'une seule contrainte, nécessite de l'ordre d'une journée de calculs pour être résolu. Cela est dû à la taille des horloges impliquées : le comportement périodique de `hf on not first_sd_line on vf` est de taille 17 280 et contient 2 880 fois l'élément `1`, celui de `encrust_end` est de taille 2 073 600 (la taille d'une image haute définition) et contient 345 600 fois l'élément `1` (la taille d'une image basse définition). Les instanciations recherchées pour `'a1` et `'a2` contiennent 2 073 600 fois l'élément `1`. Il y a donc $(2\,073\,600 - 1)$ contraintes sur les indices des `1` de chacun des mots qu'on infère pour assurer leur bonne formation (l'indice du $j^{ième}$ `1` est strictement inférieur à l'indice du $(j+1)^{ième}$ `1`). Il y a de plus 345 600 contraintes pour garantir la relation de précédence. On obtient donc un ensemble de 4 492 798 inéquations linéaires à satisfaire.

Lorsque l'horloge d'incrustation définit que la petite image doit être insérée en bas à droite de la grande, le type inféré est :

$$\begin{aligned}\forall\alpha.\quad & (\ \alpha \texttt{ on } \texttt{1}^{1919}\texttt{(1}^{2073600}\texttt{0}^{1918}\texttt{)} \ \times\ \alpha \texttt{ on (1}^{2071680}\texttt{0}^{720}\texttt{1}^{1200}\{\texttt{10100100}\}^{239}\texttt{101001)}) \\ & \quad \to \alpha \texttt{ on (1}^{2071680}\texttt{0}^{720}\texttt{1}^{1200}\{\texttt{10100100}\}^{239}\texttt{101001)}\end{aligned}$$

La taille de buffer inférée est 191 970, c'est-à-dire environ une demi-image en petite définition. On peut constater que le taux de sortie n'est pas égal à 1, alors qu'en commençant à produire plus tard, il serait possible de produire des pixels à chaque instant.

6.4 Discussion

Les exemples présentés ont permis de mettre en lumière les éléments annoncés dans les chapitres précédents. Nous avons pu constater du confort apporté par l'opérateur de bufferisation lors du codage des nœuds de l'encodeur GSM. Le code est plus facile à écrire, plus lisible et plus proche des spécifications donc moins sujet à erreurs.

La connexion des nœuds de l'encodeur a démontré que la résolution globale des contraintes d'égalité est parfois indispensable pour trouver une solution.

La partie décodage cyclique du décodeur GSM montre un exemple de programme où l'utilisation de buffers est essentielle. Dans un langage synchrone classique, ils auraient dû être encodés par le programmeur, à l'aide de tableaux par exemple, sur lesquels il aurait fallu effectuer des opérations temporelles fastidieuses afin de rendre les données disponibles aux instants adéquats.

Dans le cas de l'utilisation de l'unification interprétée ou semi-interprétée, le décodeur cyclique constitue aussi un exemple de nœud pour lequel le système de contraintes est vide après simplification : la résolution n'est donc composée que de cette phase qui effectue des vérifications. Contrairement à la phase de calcul d'une substitution satisfaisant le système, la phase de simplification est réalisée de manière gloutonne, sans ajuster la taille des horloges du système, et ne fait qu'appliquer le test d'adaptabilité sur les horloges périodiques, qui a été prouvé correct et complet. On est dans ce cas quand, pour chaque composante connexe du nœud, il existe un chemin sans buffers entre les entrées et les sorties, et il n'y a qu'un buffer sur chacun des autres chemins.

La connexion des nœuds du décodeur génère des contraintes sur des types d'horloges avec préfixes que l'on ne sait pas mettre en forme ajustée, et que l'on ne sait donc pas résoudre. Ce problème est une sérieuse limitation de l'algorithme.

Le *Picture in Picture* montre la seconde grosse limitation de la méthode présentée, son explosion combinatoire. Le temps de calcul nécessaire au typage de cette application est totalement rédhibitoire. Par ailleurs, c'est un exemple où les horloges inférées sont plus lentes que nécessaire. En effet, le type de sortie du nœud `downscaler` et de l'entrée du `merge` sont exprimés en fonction d'horloges synchronisables :

$$\texttt{hf on not first\_sd\_line on vf} \bowtie \texttt{encrust\_end}$$

Il suffirait ici d'ajouter un délai d'une ligne HD au début de l'horloge `encrust_end` afin de satisfaire la contrainte de précédence et donc la contrainte d'adaptabilité. Comme c'est la seule contrainte du système, cette instanciation constituerait une solution. Le type suivant serait donc un type correct pour le nœud `picture_in_picture` :

$$\forall\alpha.\,(\alpha \times \alpha \texttt{ on } 0^{1920}\texttt{(1)}) \rightarrow \alpha \texttt{ on } 0^{1920}\texttt{(1)}$$

Même si dans cet exemple on pouvait trouver une solution en ajoutant des délais, notons qu'il existe des cas où les types de tous les nœuds sont exprimés en fonction d'horloges synchronisables, mais il faut les ralentir pour satisfaire des contraintes de précédence.

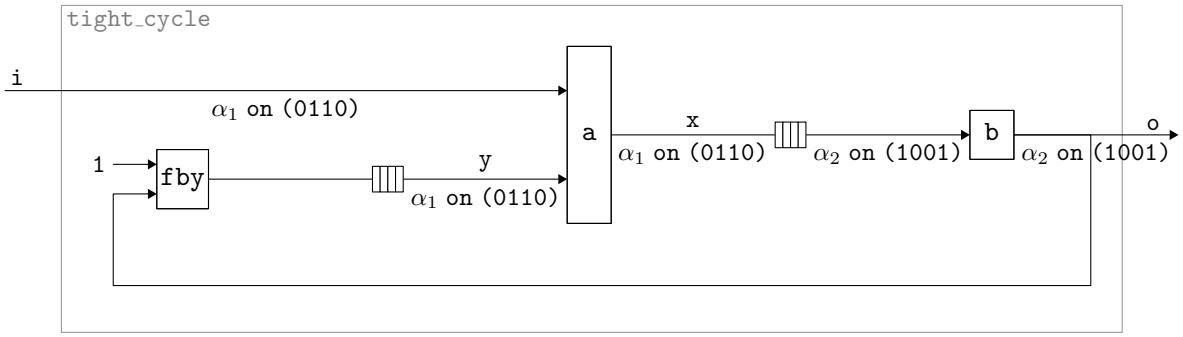


FIG. 6.5 – Un nœud dont le type d'horloges est plus lent que le plus lent des sous-nœuds.

Par exemple, considérons le nœud `tight_cycle` suivant :

```
let node tight_cycle i = o where
  rec x = a (i, buffer y)
  and y = (1 fby o)
  and o = b (buffer x)
```

avec `a` un nœud de type $\forall\alpha_1.\, \alpha_1$ `on (0110)` $\times\ \alpha_1$ `on (0110)` $\rightarrow\ \alpha_1$ `on (0110)` et `b` un nœud de type $\forall\alpha_2.\, \alpha_2$ `on (1001)` $\rightarrow\ \alpha_2$ `on (1001)`. Le nœud `tight_cycle` est représenté sur la figure 6.5. Le système de contraintes obtenu est :

$$\left\{ \begin{array}{lcl} \alpha_1 \texttt{ on (0110)} & <: & \alpha_2 \texttt{ on (1001)} \\ \alpha_2 \texttt{ on (1001)} & <: & \alpha_1 \texttt{ on (0110)} \end{array} \right\}$$

La substitution inférée est telle que :

$$\begin{array}{rcl} \theta(\alpha_1) & = & \alpha \texttt{ on (110011)} \\ \theta(\alpha_2) & = & \alpha \texttt{ on (011110)} \end{array}$$

Le type inféré pour le nœud est donc :

```
tight_cycle ::
forall 'a. 'a on (1^2 0^2 1^2 ) on (01^2 0) -> 'a on (01^4 0) on (10^2 1)
```

Les buffers résultants sont de taille nulle.

Bien que les contraintes du système ne portent que sur des horloges synchronisables (elles sont toutes de taux $\frac{1}{2}$), on est obligés de ralentir les rythmes des nœuds `a` et `b` pour satisfaire les contraintes de précédence. Il n'existe pas de solution de la forme $\theta(\alpha_1) = \alpha$ `on` 0^{d_1}`(1)` et $\theta(\alpha_2) = \alpha$ `on` 0^{d_2}`(1)`.

Étant donné le langage considéré, il n'existe donc pas d'algorithme correct et complet trouvant toujours des types tels que le réseau fonctionnera au rythme du nœud le plus lent, et jamais plus lentement. Une perspective intéressante est de se placer dans un langage où l'opérateur d'initialisation a le type $\forall\alpha.\, \alpha$ `on 0(1)` $\rightarrow \alpha$ plutôt que $\forall\alpha.\, \alpha \rightarrow \alpha$. Nous pensons que dans ce cas, tout programme causal peut être exécuté au rythme du nœud le plus lent.

Une manière de contourner les limitations de notre algorithme de résolution est d'annoter les programmes avec les types de certains flots. Cela guide l'algorithme de résolution des

contraintes, et réduit parfois le problème à de la vérification. Par exemple, le type de sortie inféré pour le nœud `f` suivant produit deux instants sur trois par rapport à son rythme d'activation :

```
let node f (x, y) = o where
  rec o = x when (10) + buffer (y when (01))

f :: forall 'a. ('a on (01^2 ) * 'a on (1^2 0)) -> 'a on (01^2 ) on (10)
```

Annotons la sortie pour stipuler que les valeurs commencent à être produites après au moins un instant de délai :

```
let node f' (x, y) = (o::'a on 0(1)) where
  rec o = x when (10) + buffer (y when (01))

f' :: forall 'a. ('a on 0(1^2 ) * 'a) -> 'a on 1(10) on 0(1)
```

Le type inféré indique alors que le nœud produit un instant sur deux par rapport à son rythme d'activation, au lieu d'un sur trois pour le programme non annoté.

6.5 Conclusion

Les travaux présentés dans cette partie nous ont convaincu que la résolution des contraintes sur les types d'horloges est un problème complexe. C'est une des raisons qui nous a poussé à nous tourner vers une méthode qui n'est pas complète sur les horloges périodiques, mais qui a l'avantage d'être simple et efficace. Elle a l'inconvénient de ne pas traiter le problème de l'unification (qui est donc réalisée structurellement), mais présente la force de pouvoir s'appliquer à un langage d'horloges plus large, contenant des horloges pas forcément périodiques mais encadrées par des horloges périodiques. Cette méthode est décrite dans la partie III, qui suit la même structure que la partie que nous sommes en train de clore : le chapitre 7 présente l'algèbre associée aux horloges manipulées, le chapitre 8 montre comment résoudre les contraintes de typage sur ces horloges, et le chapitre 9 présente l'application satisfaisante aux exemples que nous venons de présenter.

Troisième partie

Horloges abstraites

Chapitre 7

Horloges abstraites

Dans le chapitre 4, nous avons montré comment vérifier la relation d'adaptabilité sur les mots binaires ultimement périodiques, et comment calculer le mot binaire périodique résultat d'une horloge composée. On s'intéresse maintenant aux mots qui ne sont pas forcément exactement périodiques mais presque, et dont on connaît une spécification. Par exemple :

- les mots périodiques sur lesquels on autorise un décalage d'un nombre d'instants borné pour l'occurrence des `1`. Ces mots sont utiles pour modéliser la gigue. On peut ainsi spécifier l'horloge d'un flot dont les valeurs arrivent un instant sur quatre à plus où moins un instant près. Ces mots sont aussi utiles pour modéliser le temps de calcul variable des nœuds. On peut ainsi spécifier l'horloge de sortie d'un nœud dont les valeurs sont produites de deux à trois instants après l'instant où elles sont consommées.
- les mots infinis spécifiés par une expression régulière ou un automate, tels que l'ensemble des mots reconnus sont encadrés par des horloges périodiques de même taux asymptotique de `1`. Ces mots sont utiles pour spécifier la valeur d'une horloge calculée par une expression booléenne complexe : par exemple $w \in ((10) + (01))^\omega$.

Pour traiter ces horloges dont on ne connaît pas statiquement la valeur exacte, nous proposons d'utiliser les méthodes de l'interprétation abstraite [CC77, CH78]. Pour vérifier la relation d'adaptabilité sur deux mots w_1 et w_2 dont on ne connaît qu'une spécification, ou *abstraction*, nous vérifions que la relation tient pour tout couple de mots répondant aux spécifications respectives de w_1 et w_2. Si c'est le cas, on a alors l'assurance que la relation tient sur les deux mots qui nous intéressent. De la même manière, on calcule ensuite une taille de buffer suffisante pour tout couple de mots répondant aux spécifications de w_1 et w_2.

S'il s'agit de raisonner non plus sur des mots mais sur des horloges composées d'opérations *on* et *not*, il faudra trouver une spécification correcte pour l'horloge à partir des spécifications des mots qui la composent.

Il s'agit donc de définir :

- **l'ensemble des valeurs abstraites**, ou domaine abstrait, et **la fonction de concrétisation** Cette fonction associe à chaque valeur abstraite l'ensemble des mots qu'elle représente, appelé ensemble de concrétisation.
- **une relation abstraite** $<:^\sim$
  Cette relation doit être telle que pour toutes valeurs abstraites a_1 et a_2, $a_1 <:^\sim a_2$ implique $w_1 <: w_2$ pour tout mot w_1 représenté par a_1, et tout mot w_2 représenté par a_2. Il est ainsi correct de vérifier la relation d'adaptabilité sur une abstraction des mots.

- **une opération abstraite** $size^{\sim}$
  Cette opération doit être telle que pour toutes valeurs abstraites a_1 et a_2, $size^{\sim}(a_1, a_2) \geq size(w_1, w_2)$ pour tout mot w_1 représenté par a_1, et tout mot w_2 représenté par a_2. Il est ainsi correct de calculer une taille de buffer suffisante sur une abstraction des mots.
- **des opérateurs abstraits** $on^{\sim}$ **et** $not^{\sim}$
  Ces opérateurs doivent permettre de trouver une valeur abstraite correcte pour une horloge composée, à partir des valeurs abstraites des mots qui la composent.

La perte d'information engendrée par l'abstraction implique une perte de précision, mais elle permet de raisonner de manière correcte sur des mots pour lesquels on ne sait pas calculer de manière exacte. Nous verrons qu'elle permet aussi de raisonner de manière moins coûteuse sur les mots ultimement périodiques.

La section 7.1 donne l'intuition de la méthode proposée en présentant une première manière d'abstraire les mots, ainsi que les opérations et relations qui y sont associées. Cette abstraction a été présentée dans [CMPP08]. La section 7.2 raffine cette abstraction, et la suite du chapitre présente de manière formelle l'algèbre des horloges abstraites et leurs propriétés. Une version intermédiaire de l'abstraction est décrite dans [MP09] et sa forme actuelle a été présentée dans [CMPP09].

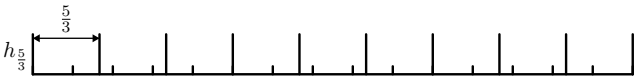
7.1 Première intuition

On propose d'abstraire un mot binaire infini par les deux informations suivantes :

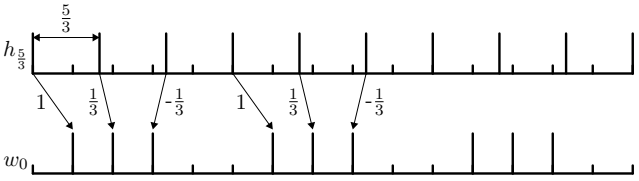
1. la distance moyenne T séparant deux `1` dans w
2. deux phases d et D qui bornent les indices des `1` dans w par rapport au rythme parfait d'un `1` tous les T instants

On note cette valeur abstraite $[d, D]\,(T)$, avec $d, D, T \in \mathbb{Q}$.

Imaginons une horloge idéale h_T qui produise un tick au premier instant puis un tick tous les T instants (ici $\frac{5}{3}$) :



Les mots abstraits par la valeur $[d, D]\,(T)$ sont les mots dont le $j^{ième}$ `1` arrive toujours avec un décalage de d à D instants par rapport au $j^{ième}$ tick de h_T (qui se trouve à l'instant $1 + T \times (j-1)$). Par exemple, le mot w_0 est abstrait par $\left[-\frac{1}{3}, 1\right]\left(\frac{5}{3}\right)$:



On peut observer que les `1` de w_0 arrivent avec un décalage d'au minimum $-\frac{1}{3}$ et d'au maximum 1 par rapports aux ticks correspondants de $h_{\frac{5}{3}}$.

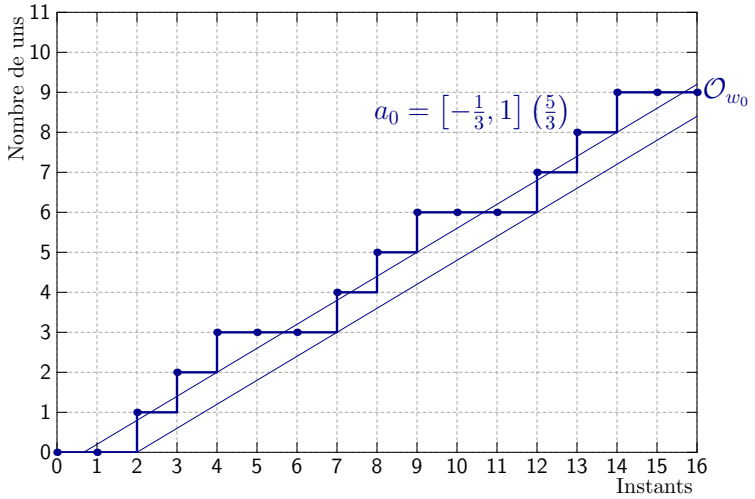
Plus formellement, une valeur $[d, D](T)$ représente l'ensemble de mots binaires infinis suivant, appelé ensemble de concrétisation :

$$concr([d, D](T)) \stackrel{def}{=} \{w,\ \forall j \geq 0,\ 1 + T \times j + d \leq \mathcal{I}_w(j+1) \leq 1 + T \times j + D\}$$

D sera toujours positif ou nul, car dans le cas contraire l'indice du premier `1` doit être inférieur à l'indice du premier instant ($\mathcal{I}_w(1) < 1$), contrainte satisfaisable par aucun mot binaire. T sera toujours supérieur ou égal à 1, car la distance moyenne entre deux `1` ne peut être inférieure à un instant (un seul `1` peut se produire à chaque instant).

On peut remarquer que l'indice du premier `1` doit être compris entre $1 + d$ et $1 + D$, donc les valeurs d et D donnent le nombre minimum et maximum de `0` en préfixe des mots représentés par $[d, D](T)$.

Dans les graphiques des fonctions de cumul, la valeur abstraite $[d, D](T)$ peut être représentée par deux droites qui encadrent les points de départ des fronts montants des mots qu'elle contient. Ces droites sont de pente $\frac{1}{T}$ et de décalage horizontal d et D par rapport à l'instant un. En référence à cette représentation graphique des valeurs abstraites, on les nommera aussi *enveloppes*, et lorsqu'un mot est abstrait par une valeur a, on dira qu'il est dans l'enveloppe a. Voici la fonction de cumul du mot w_0, et sa valeur abstraite $a_0 = \left[-\frac{1}{3}, 1\right]\left(\frac{5}{3}\right)$ représentée par deux droites de pente $\frac{3}{5}$ et de décalages horizontaux respectifs $-\frac{1}{3}$ et 1 par rapport à l'instant un :



L'enveloppe a_0 contient une infinité d'autres mots. Toutes les fonctions de cumul dont les fronts montants démarrent entre les deux droites représentent des mots de l'enveloppe. La figure 7.1 montre les différentes trajectoires possibles. On peut constater qu'il y a une infinité de points de choix (instants 1, 4, 6, 9, ...). Il y a donc une infinité de mots dans l'enveloppe, périodiques ou non. Ces mots sont les mots reconnus par l'expression régulière suivante : $(\ (\texttt{10} + \texttt{01})\ .\ \texttt{1}\ .\ (\texttt{10} + \texttt{01})\ )^{\omega}$.

Plus l'intervalle $[d, D]$ est large, c'est-à-dire plus les droites sont écartées, plus l'enveloppe $[d, D](T)$ contient de mots.

Nous noterons $abs(w)$ toute fonction d'abstraction correcte, c'est-à-dire toute fonction qui associe à un mot w une enveloppe qui le contient : [1]

$$abs(w) = a \Rightarrow w \in concr(a)$$

[1] Traditionnellement [CC77], les fonctions d'abstraction sont notées α et les fonctions de concrétisation γ.

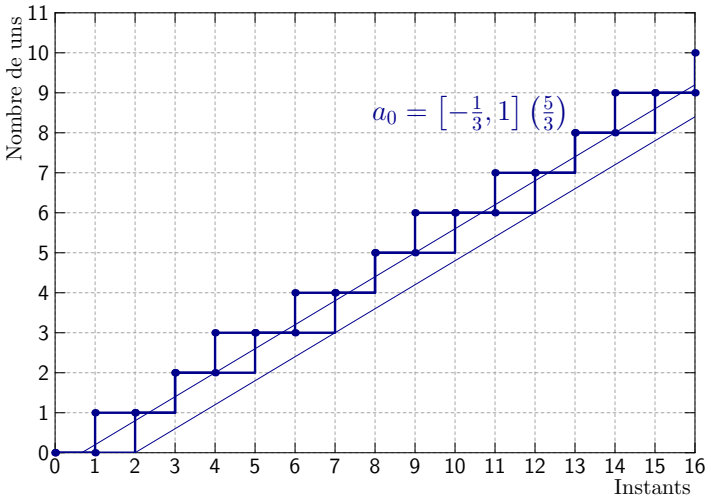


FIG. 7.1 – Trajectoires possibles pour les fonctions de cumul des mots de l'enveloppe a_0.

Relation d'adaptabilité sur les valeurs abstraites

On définit une relation $a_1 <:^{\sim} a_2$ qui est vérifiée si et seulement si pour tout mot w_1 dans a_1 et tout mot w_2 dans a_2, la relation $w_1 <: w_2$ est vérifiée. Ainsi, il est correct de vérifier la relation d'adaptabilité sur l'abstraction des deux mots plutôt que sur les mots eux-mêmes, c'est-à-dire que

$$abs\,(w_1) <:^{\sim} abs\,(w_2) \Rightarrow w_1 <: w_2$$

Tout comme pour la relation concrète $<:$, on définit la relation abstraite $<:^{\sim}$ par la conjonction des relations de synchronisabilité et de précédence :

$$a_1 <:^{\sim} a_2 \overset{def}{\Leftrightarrow} a_1 \bowtie^{\sim} a_2 \wedge a_1 \preceq^{\sim} a_2$$

Nous avons vu en section 3.4 que deux mots sont synchronisables si et seulement si leurs fonctions de cumul restent à une distance verticale bornée l'une de l'autre. Cette condition est vérifiée entre les mots d'une enveloppe a_1 et les mots d'une enveloppe a_2 si et seulement si a_1 et a_2 sont de même pente :

$$[d_1, D_1]\,(T_1) \bowtie^{\sim} [d_2, D_2]\,(T_2) \Leftrightarrow T_1 = T_2$$

Par exemple, dans la figure 7.2, l'enveloppe $a_1 = \left[-\frac{2}{3}, 0\right]\left(\frac{5}{3}\right)$ du mot w_1 est synchronisable avec l'enveloppe $a_2 = \left[-\frac{5}{3}, 3\right]\left(\frac{5}{3}\right)$ du mot w_2. Par contre, dans la figure 7.3, les enveloppes a_5 et a_6 ne sont pas synchronisables. Les mots contenus par a_5 s'éloignent infiniment des mots contenus par a_6.

Nous avons aussi vu en section 3.4 qu'un mot w_1 précède un mot w_2 si et seulement si le $j^{ième}$ front montant de sa fonction de cumul arrive à un instant antérieur ou égal au $j^{ième}$ front montant de celle de w_2. Une condition suffisante pour garantir que tous les mots d'une enveloppe a_1 précèdent tous ceux d'une enveloppe a_2 est que a_1 se situe à gauche de a_2. Lorsque les enveloppes sont de même pente, on sait le vérifier par comparaison des décalages :

$$D_1 \leq d_2 \Rightarrow [d_1, D_1]\,(T) \preceq^{\sim} [d_2, D_2]\,(T)$$

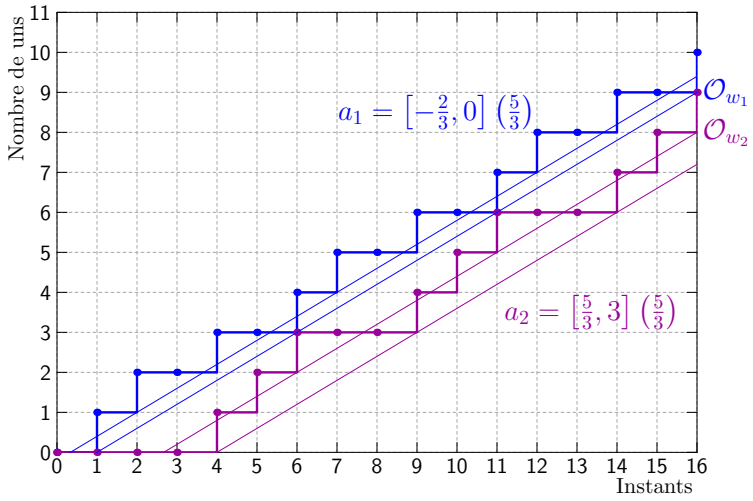


FIG. 7.2 – L'enveloppe a_1 est synchronisable avec l'enveloppe a_2 et elle la précède.

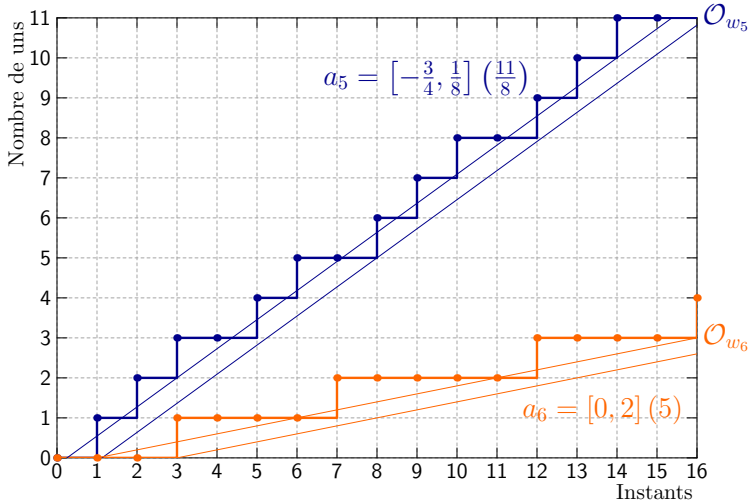


FIG. 7.3 – L'enveloppe a_5 n'est pas synchronisable avec l'enveloppe a_6.

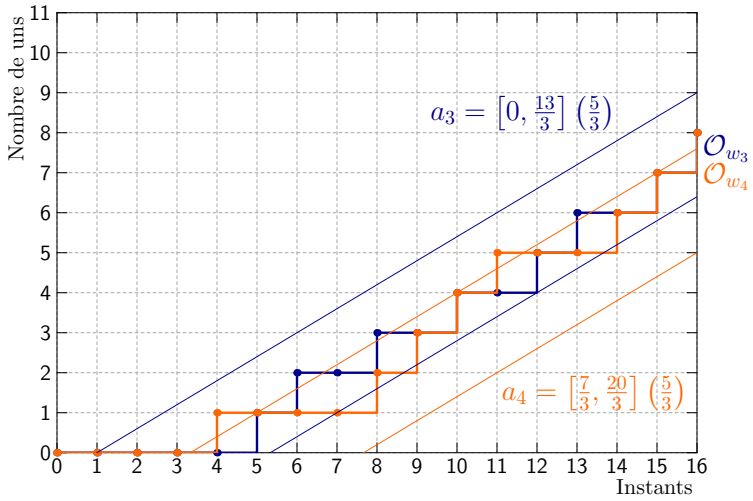


FIG. 7.4 – L'enveloppe a_3 ne précède pas a_4 et a_4 ne précède pas a_3.

Par exemple, dans la figure 7.2 l'enveloppe a_1 précède l'enveloppe a_2. Le $j^{ième}$ front montant des mots de a_1 arrive dans la zone entre les deux droites de a_1 à l'ordonnée $(j-1)$, donc forcément avant le $j^{ième}$ front montant des mots de a_2 qui arrive dans la zone entre les deux droites de a_2 à l'ordonnée $(j-1)$.

Par contre, dans la figure 7.4, l'enveloppe a_3 ne précède pas a_4 et a_4 ne précède pas a_3. Les fonctions de cumul contenues par ces deux enveloppes peuvent être entrelacées.

En fait, les deux enveloppes peuvent se chevaucher légèrement sans perdre la garantie que les mots de la première précèdent ceux de le seconde. La taille du chevauchement doit être suffisamment petite pour qu'il n'y aie qu'un instant discret dans cette zone pour toute ordonnée $j-1$, ce qui permet, au pire, l'occurrence du $j^{ième}$ front en même temps dans les deux mots. La condition $D_1 - d_2 < 1$ convient donc aussi et elle est plus précise. C'est le cas dans la figure 7.5, où l'enveloppe a'_1 précède l'enveloppe a_2, malgré le fait qu'elles se chevauchent légèrement.

Si une valeur abstraite a_1 est adaptable à une valeur abstraite a_2, alors on peut calculer une taille de buffer suffisante pour communiquer de tout mot de a_1 vers tout mot de a_2. La taille de buffer nécessaire pour communiquer d'un mot w_1 vers un mot w_2 est la distance maximale entre la courbe de cumul de w_1 et la courbe de cumul de w_2. Une sur-approximation de cette distance peut être calculée en fonction de la distance entre le haut de l'enveloppe de w_1 et le bas de l'enveloppe de w_2.

La relation d'adaptabilité définie sur les mots binaires se vérifie donc aisément sur l'enveloppe de ces mots par des tests simples sur des nombres rationnels.

Opérateur $on^{\sim}$ sur les valeurs abstraites

On veut définir un opérateur on abstrait noté $on^{\sim}$, tel que l'enveloppe résultat de a_1 $on^{\sim}$ a_2 contienne tous les mots que l'on peut obtenir en calculant w_1 on w_2 avec w_1 un mot de l'enveloppe a_1 et w_2 un mot de l'enveloppe a_2. Il est ainsi correct de calculer l'opération d'échantillonnage sur l'abstraction des mots plutôt que sur les mots eux-mêmes, car w_1 on $w_2 \in concr\,(abs\,(w_1)$ $on^{\sim}$ $abs\,(w_2))$.

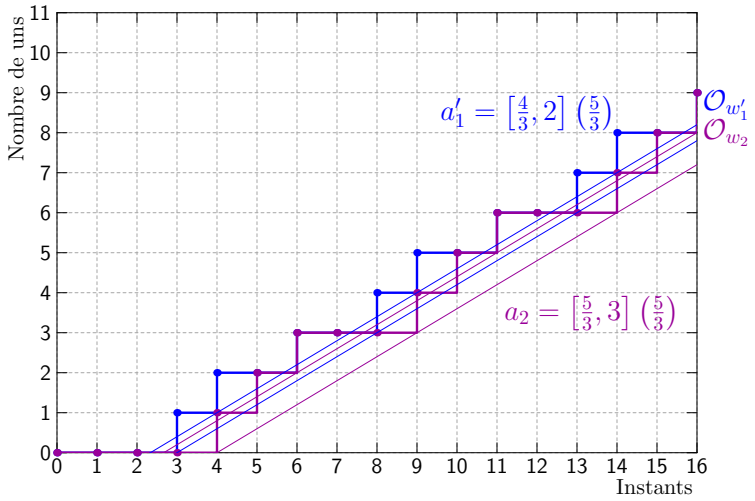


FIG. 7.5 – Bien que les enveloppes a'_1 et a_2 se chevauchent légèrement, a'_1 précède a_2, car tous les mots de a'_1 précédent les mots de a_2.

Cet opérateur peut être défini ainsi :

$$\begin{array}{lll} & [\quad d_1 \quad , \quad D_1 \quad]\,(\quad T_1 \quad) \\ \mathit{on}^{\sim} & [\quad d_2 \quad , \quad D_2 \quad]\,(\quad T_2 \quad) \\ \stackrel{def}{=} & [\,d_1 + d_2 \times T_1 \;,\; D_1 + D_2 \times T_1\,]\,(\,T_1 \times T_2\,) \end{array}$$

Nous avons vu en section 3.2 que les éléments de w_1 on w_2 sont les éléments de w_2, parcourus au rythme des `1` dans w_1. Donc si la distance entre les `1` de w_2 est en moyenne égale à T_2, et la distance entre ceux de w_1 est en moyenne égale à T_1, alors la distance entre les `1` de w_1 on w_2 est en moyenne égale à $T_1 \times T_2$. Échantillonner w_1 avec w_2 garde intact le délai de w_1, et y ajoute de délai de w_2 multiplié par T_1 car w_2 est traversé au rythme T_1.

Par exemple, si $a_1 = [1, 2]\,(2)$ et $a_2 = [2, 4]\,(3)$ alors a_1 $\mathit{on}^{\sim}$ $a_2 = [5, 10]\,(6)$. Vérifions que ce résultat est correct sur les mots les plus précoces (au sens de $\preceq$) de a_1 et a_2. Le mot le plus précoce de a_1 est composé d'un `0` ($d_1 = 1$) suivi d'un `1` tous les deux instants ($T_1 = 2$). Il s'agit donc du mot $w_1 =$ `0(10)`. Celui de a_2 est composé de deux `0` ($d_2 = 2$) suivis d'un `1` tous les trois instants ($T_2 = 3$). Il s'agit donc du mot $w_2 =$ `00(100)`. Calculons w_1 on w_2 :

	`0(10)`	0	1	0	1	0	1	0	1	0	1	0	1	0	...	$\in \mathit{concr}\,[\quad 1 \quad , \quad 2 \quad]\,(\quad 2 \quad)$
on	`00(100)`		0		0		1		0		0		1		...	$\in \mathit{concr}\,[\quad 2 \quad , \quad 4 \quad]\,(\quad 3 \quad)$
=	0^5`(100000)`	0	0	0	0	0	1	0	0	0	0	0	1	0	...	$\in \mathit{concr}\,[1{+}2{\times}2\;,\;2{+}4{\times}2\,]\,(\,2{\times}3\,)$

Les `1` de w_1 on w_2 sont effectivement espacés de 2×3 instants en moyenne. Le `0` en préfixe de w_1 est reporté dans le résultat et les deux `0` en préfixe de w_2 sont reportés au rythme des `1` dans w_1 (tous les deux instants). Il y a donc $1 + 2 \times 2$ occurrences de l'élément `0` en préfixe du résultat. On peut en conclure qu'on a bien `0(10)` on `00(100)` $\in \mathit{concr}\,([5, 10]\,(6))$.

Abstraire les horloges permet donc de vérifier la relation d'adaptabilité en temps constant, par calculs arithmétiques simples.

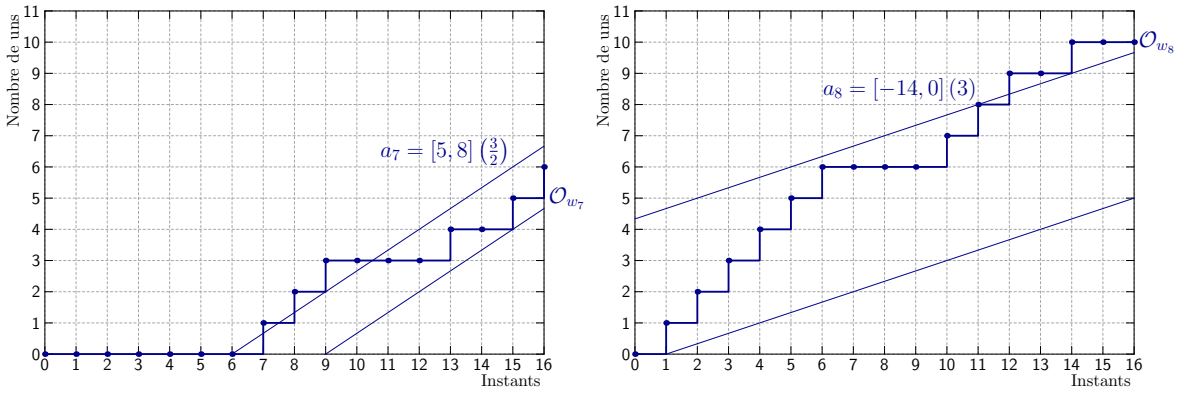


FIG. 7.6 – L'abstraction d'un mot commençant par une série de `0`, et l'abstraction de la négation de ce mot, qui commence par une série de `1`.

L'abstraction présentée dans cette section a néanmoins les deux inconvénients suivants :

1. Il est impossible de modéliser les mots ne contenant qu'un nombre fini de `1`, car dans ce cas la distance moyenne T vaut $+\infty$.
2. Il est possible de créer une enveloppe ne contenant que des mots qui commencent par une série de `0`, mais il est impossible de créer une enveloppe ne contenant que des mots qui commencent par une série de `1`. Sur la figure 7.6, on peut observer que l'enveloppe de w_7 est très précise et en particulier ne perd pas l'information de la présence d'une série de `0` en préfixe. Par contre, celle de $w_8 = \texttt{not}\ w_7$ est très large, et contient donc un grand nombre de mots, en particulier des mots qui ne commencent pas par une série de `1`.

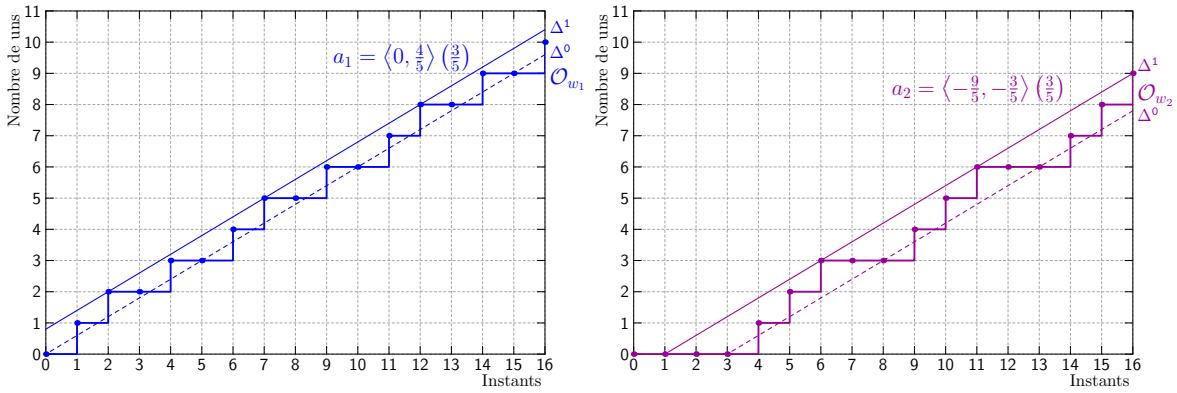
Ces deux inconvénients sont liés au fait que l'abstraction raisonne sur les indices des `1`, et ne contraint que ceux-ci (pas l'indice des `0`). Cette asymétrie mène à un opérateur abstrait $\texttt{not}^\sim$ peu intuitif :

$$\texttt{not}^\sim\left([d, D]\,(T)\right) \stackrel{def}{=} \left[\frac{-D+1}{T-1}, \max\left(0, 1-\frac{d}{T-1}\right)\right]\left(\frac{T}{T-1}\right) \qquad \text{avec } T > 1$$

Elle est responsable du fait que l'opérateur $\texttt{not}^\sim$ n'est pas défini quand le résultat concret contient un nombre fini de `1`. Par exemple, $\texttt{not}$ `000110(1)` $=$ `111001(0)` ne peut pas être abstrait. Enfin, cette asymétrie implique une perte d'information lors du calcul de $\texttt{not}^\sim$ quand l'opérande commence par une série de `0`. On peut observer sur la figure 7.6 que $a_8 = \texttt{not}^\sim\ a_7$ contient des mots qui ne correspondent pas à la négation de mots de a_7 (ceux qui ne commencent pas par une série de `1`). Si l'on calcule $\texttt{not}^\sim\ a_8$, on obtient donc une enveloppe plus large que a_7.

Pour résoudre ces deux inconvénients, il suffit de considérer la valeur de la fonction de cumul plutôt que l'indice des `1`, et contraindre ainsi à la fois l'occurrence des `1` et l'occurrence des `0`.

La suite de ce chapitre présente en détail l'abstraction retenue. La section 7.2 décrit les valeurs abstraites et les mots qu'elles contiennent, la section 7.3 montre comment obtenir la valeur abstraite d'un mot ou d'un ensemble de mots et les sections 7.4 et 7.5 décrivent respectivement les opérateurs et les relations sur ces valeurs abstraites. Pour conclure, la section 7.6 discute de la qualité de l'abstraction retenue, en ce qui concerne sa précision et le coût de ses calculs.



w_1 est dans $a_1 = \langle 0, \frac{4}{5} \rangle \left(\frac{3}{5}\right)$ w_2 est dans $a_2 = \langle -\frac{9}{5}, -\frac{3}{5} \rangle \left(\frac{3}{5}\right)$

FIG. 7.7 – Si les fronts montants d'un mot w aboutissent tous sous la droite Δ^1 d'équation $r \times i + b^1$ et l'absence de front ne se produit qu'au dessus de la droite Δ^0 d'équation $r \times i + b^0$, alors w est dans l'abstraction $\langle b^0, b^1 \rangle (r)$.

7.2 Les enveloppes

L'abstraction d'un mot binaire infini w que nous proposons ici consiste à conserver uniquement le taux asymptotique r de `1` dans w et deux valeurs b^0 et b^1 qui donnent les décalages minimum et maximum du nombre de `1` vus dans w par rapport au taux r. On note cette valeur abstraite $\langle b^0, b^1 \rangle (r)$.

7.2.1 Valeurs abstraites

Une valeur abstraite $\langle b^0, b^1 \rangle (r)$ représente l'ensemble de mots binaires infinis suivant :

Définition 7.1 (concrétisation). ✿

$$concr\left(\langle b^0, b^1 \rangle (r)\right) \stackrel{def}{=} \left\{ w \mid \forall i \geq 1, \quad \wedge \begin{array}{lcl} w[i] = \mathtt{1} & \Rightarrow & \mathcal{O}_w(i) \leq r \times i + b^1 \\ w[i] = \mathtt{0} & \Rightarrow & \mathcal{O}_w(i) \geq r \times i + b^0 \end{array} \right\}$$

avec $b^0, b^1, r \in \mathbb{Q}$ et $0 \leq r \leq 1$.

Un taux $r > 1$ représenterait des mots qui ont en moyenne plus qu'un `1` par instant, qui ne sont pas considérés pour l'instant (comme il est mentionné en section 3.1, $\mathcal{O}_w(i+1) - \mathcal{O}_w(i) \leq 1$). Nous reviendrons sur ce point dans la partie IV.

Dans les chronogrammes, la valeur abstraite $\langle b^0, b^1 \rangle (r)$ peut être représentée par deux droites $\Delta^1 : r \times i + b^1$ et $\Delta^0 : r \times i + b^0$ qui bornent les fronts montants et les absences de front des mots qu'elle contient. La définition 7.1 indique que tout front montant doit arriver sous la droite Δ^1 (représentée par une ligne continue) et toute absence de front doit se produire au dessus de la droite Δ^0 (représentée par une ligne en pointillés). Les valeurs b^0 et b^1 représentent donc des décalages verticaux.[2] Par exemple, dans la figure 7.7 le mot $w_1 =$ `(11010)` est dans $a_1 = \langle 0, \frac{4}{5} \rangle \left(\frac{3}{5}\right)$ et le mot $w_2 =$ `0(00111)` est dans $a_2 = \langle -\frac{9}{5}, -\frac{3}{5} \rangle \left(\frac{3}{5}\right)$.

Les droites Δ^0 et Δ^1 peuvent être interprétées comme les valeurs minimale et maximale désirées pour la fonction de cumul $\mathcal{O}_w$. Les mots appartenant à $\langle b^0, b^1 \rangle (r)$ sont les mots de

[2] Les valeurs d et D de l'abstraction de la section précédente représentent des décalages horizontaux.

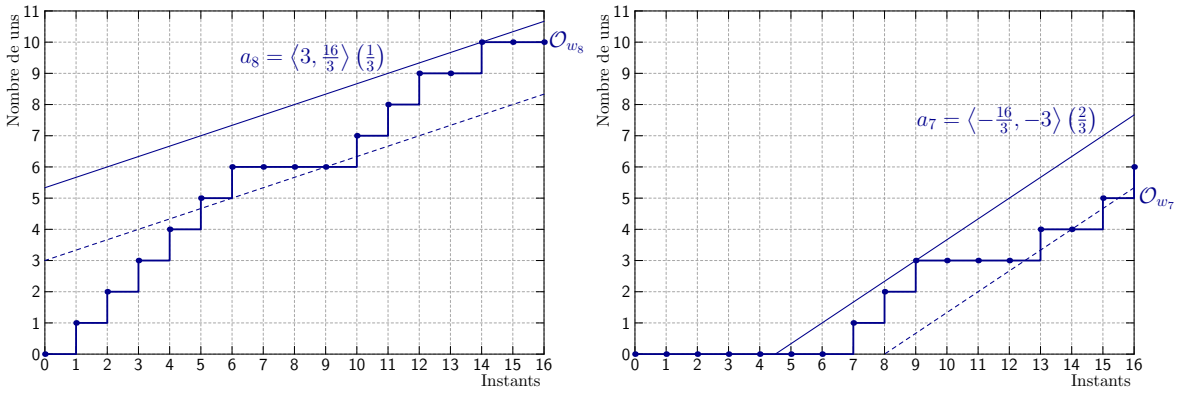


FIG. 7.8 – L'abstraction a_8 ne contient que des mots qui commencent par cinq 1 et l'abstraction a_7 ne contient que des mots qui commencent par cinq 0.

taux r dans lesquels l'occurrence d'un 1 à l'instant i ne peut se produire que si elle n'entraîne pas le dépassement du nombre de 1 maximal désiré $r \times i + b^1$, et l'occurrence d'un 0 à l'instant i ne peut se produire que si on est au dessus du nombre de 1 minimal désiré $r \times i + b^0$.

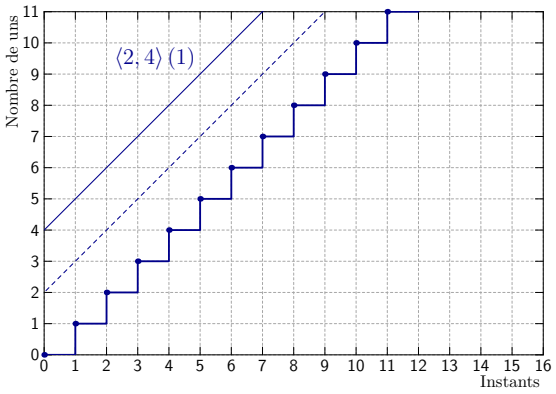
Une valeur abstraite peut donc maintenant représenter un ensemble de mots qui commencent tous par une série de 1. Par exemple, sur la figure 7.8, $a_8 = \langle 3, \frac{16}{3} \rangle \left(\frac{1}{3}\right)$ ne contient que des mots qui commencent par cinq 1. Les deux droites forment une zone qu'il faut chercher à atteindre, et dont on ne ressort pas une fois atteinte. Pour cette raison, nous reprendrons le terme *enveloppe* pour parler des valeurs abstraites, bien que la fonction de cumul ne soit pas forcément dans la zone entre les deux droites dès le premier instant. [3] Sur la figure 7.8, jusqu'à l'instant 5, les fonctions de cumul des mots représentés par a_8 sont sous cette zone et doivent donc obligatoirement monter pour l'atteindre (les 0 sont interdits sous Δ^0). À partir de l'instant 5, toutes les valeurs prises par la fonction de cumul sont dans cette zone. Notez que c'est bien le cas pour le mot représenté sur la figure, et ce même aux instants 5 et 10 car la valeur de la fonction de cumul correspond au haut du front montant.

Une valeur abstraite peut aussi représenter un ensemble de mots qui commencent tous par une série de 0. Par exemple, sur la figure 7.8, jusqu'à l'instant 5, les fonctions de cumul des mots représentés par a_7 sont au dessus des deux droites et doivent donc obligatoirement stagner pour rentrer dans la zone entre les droites (les 1 sont interdits si le haut du front n'est pas sous Δ^1). À partir de l'instant 5, toutes les valeurs prises par les fonctions de cumul sont dans cette zone.

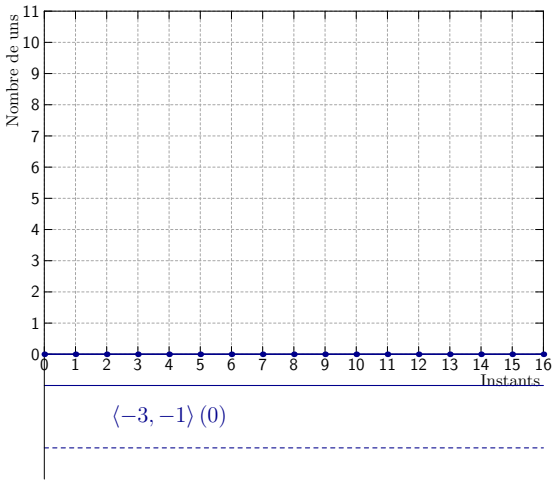
Plus formellement, si un mot appartenant à une enveloppe est sous Δ^1 à l'instant i, alors il reste sous Δ^1 pour toujours. De même, s'il est au dessus de Δ^0 à l'instant i, alors il y reste pour toujours.

Lemme 7.1. Soit $w \in \mathit{concr}\left(\langle b^0, b^1 \rangle (r)\right)$.
$\mathcal{O}_w(i) \leq r \times i + b^1 \quad \Rightarrow \quad \forall i' \geq i,\ \mathcal{O}_w(i') \leq r \times i' + b^1$
$\mathcal{O}_w(i) \geq r \times i + b^0 \quad \Rightarrow \quad \forall i' \geq i,\ \mathcal{O}_w(i') \geq r \times i' + b^0$

[3] On dira qu'un mot appartient à une enveloppe pour dire qu'il appartient à la concrétisation de celle ci.



Quand le taux vaut 1, les droites Δ^1 et Δ^0 peuvent être inutilement hautes.



Quand le taux vaut 0, les droites Δ^1 et Δ^0 peuvent être inutilement basses.

FIG. 7.9 – Cas particuliers.

Démonstration :

Supposons que $\mathcal{O}_w(i) \leq r \times i + b^1$.
Si $w[i+1] = \mathtt{1}$, comme $w \in \mathit{concr}\,(a)$, on a $\mathcal{O}_w(i+1) \leq r \times (i+1) + b^1$.
Si $w[i+1] = \mathtt{0}$, alors $\mathcal{O}_w(i+1) = \mathcal{O}_w(i)$, donc $\mathcal{O}_w(i+1) \leq r \times i + b^1$. Comme $r \geq 0$, on peut en déduire $\mathcal{O}_w(i+1) \leq r \times (i+1) + b^1$.

D'autre part, supposons que $\mathcal{O}_w(i) \geq r \times i + b^0$.
Si $w[i+1] = \mathtt{1}$, alors $\mathcal{O}_w(i+1) = \mathcal{O}_w(i) + 1$. Par hypothèse, on a $\mathcal{O}_w(i) \geq r \times i + b^0$.
Comme $r \leq 1$, on peut en déduire $\mathcal{O}_w(i+1) = \mathcal{O}_w(i)+1 \geq r \times i + b^0 + 1 \geq r \times (i+1) + b^0$.
Si $w[i+1] = \mathtt{0}$, comme $w \in \mathit{concr}\,(a)$, on a $\mathcal{O}_w(i+1) \geq r \times (i+1) + b^0$.

□

De plus, dans le cas général, la courbe de cumul finit toujours par rentrer dans la zone entre les droites. Il existe deux cas spéciaux où ce n'est pas le cas :

Remarque 7.1 (Cas particuliers)**.**

1. Le cas où le taux vaut 1 et la droite Δ^0 a un décalage vertical b^0 strictement positif, comme sur la figure 7.9. Dans ce cas, $\Delta^0(i)$ est toujours strictement supérieure à i, et comme la courbe de cumul ne peut monter que d'une unité par instant, elle n'atteint jamais Δ^0. Le seul mot respectant la contrainte imposée par Δ^0 est `(1)`.
2. Le cas où le taux vaut 0 et la droite Δ^1 a un décalage vertical b^1 strictement négatif, comme sur la figure 7.9. Le seul mot respectant la contrainte imposée par Δ^1 est `(0)`.

On peut aussi remarquer sur la figure 7.9 que si le taux vaut 1, une droite Δ^1 ayant un décalage vertical b^1 strictement positif est inutilement haute et ne pourra jamais être atteinte, et si le taux vaut 0, une droite Δ^0 avec un décalage vertical b^0 strictement négatif est inutilement basse et ne pourra jamais être atteinte non plus.

Ces cas ne nous intéressent pas pour le moment, car dans notre cadre, on peut remplacer l'enveloppe $\langle b^0, b^1 \rangle (1)$ par $\langle \min(b^0, 0), \min(b^1, 0) \rangle (1)$ et l'enveloppe $\langle b^0, b^1 \rangle (0)$ par $\langle \max(b^0, 0), \max(b^1, 0) \rangle (0)$ sans changer les ensembles de concrétisation (on baisse les droites inutilement hautes et on monte les droites inutilement basses).

Dans la suite du chapitre, on considérera (sauf mention contraire) que toute enveloppe $\langle b^0, b^1 \rangle (r)$ est telle que :

$$\begin{array}{l} r = 1 \;\Rightarrow\; b^0 \leq 0 \text{ et } b^1 \leq 0 \\ r = 0 \;\Rightarrow\; b^0 \geq 0 \text{ et } b^1 \geq 0 \end{array}$$

◇

Une fois que les cas spéciaux décris dans la remarque précédente sont écartés, on peut prouver qu'il existe forcément un instant à partir duquel $\mathcal{O}_w$ passe entre les deux droites. Avant cet instant, tous les éléments de w valent `1` si la courbe est initialement sous les droites ($0 < b^0$), et `0` si la courbe est initialement au dessus des droites ($b^1 < 0$).

Lemme 7.2. $\forall w \in \mathit{concr}\left(\langle b^0, b^1 \rangle (r)\right), \exists i',\ r \times i' + b^0 \leq \mathcal{O}_w(i') \leq r \times i' + b^1$.

Si $b^0 \leq 0 \leq b^1$, alors $i' = 0$.

Si $0 < b^0$, alors $i' = \left\lceil \frac{b^0}{1-r} \right\rceil$ et $\forall i \leq i',\ w[i] = \mathtt{1}$.

Si $b^1 < 0$, alors $i' = \lceil -\frac{b^1}{r} \rceil$ et $\forall i \leq i',\ w[i] = \mathtt{0}$.

Démonstration :

Si $b^0 \leq 0 \leq b^1$**,** alors $r \times 0 + b^0 \leq \mathcal{O}_w(0) = 0 \leq r \times 0 + b^1$.

Si $0 < b^0$**.** Par la remarque 7.1, $r \neq 1$. Pour tout $i = 1$ à $i' = \left\lceil \frac{b^0}{1-r} \right\rceil$, on a $i \leq \frac{b^0}{1-r}$ donc $i - 1 < r \times i + b^0$. Par conséquent, comme pour tout i, $\mathcal{O}_w(i-1) \leq i - 1$, on sait que pour tout $i = 1 \ldots i'$, on a $\mathcal{O}_w(i-1) < r \times i + b^0$. Comme $w \in \langle b^0, b^1 \rangle (r)$, cela implique que pour tout $i = 1 \ldots i'$, $w[i] \neq \mathtt{0}$. Donc pour tout $i = 1 \ldots i'$, $\mathcal{O}_w(i) = i$, et on peut conclure que $\mathcal{O}_w(i') = i' \geq r \times i' + b^0$. D'autre part, comme $w[i'] = \mathtt{1}$ et $w \in \langle b^0, b^1 \rangle (r)$, on a $\mathcal{O}_w(i') \leq r \times i' + b^1$.

Si $b^1 < 0$**.** Par la remarque 7.1, $r \neq 0$. De la même manière, pour tous les $i = 1$ à $i' = \lceil -\frac{b^1}{r} \rceil$, on a $r \times i + b^1 \leq 0$. Par conséquent, comme pour tout i, $\mathcal{O}_w(i) \geq 0$, on sait que pour tout $i = 1 \ldots i'$, on a $\mathcal{O}_w(i-1) \geq r \times i + b^1$, donc $\mathcal{O}_w(i-1) + 1 > r \times i + b^1$. Comme $w \in \langle b^0, b^1 \rangle (r)$, cela implique que pour tout $i = 1 \ldots i'$, $w[i] \neq \mathtt{1}$. Donc pour tout $i = 1 \ldots i'$, $\mathcal{O}_w(i) = 0$, et on peut conclure que $\mathcal{O}_w(i') = 0 \leq r \times i' + b^1$. D'autre part, comme $w[i'] = \mathtt{0}$ et $w \in \langle b^0, b^1 \rangle (r)$, on a $\mathcal{O}_w(i') \geq r \times i' + b^0$.

□

On peut en conclure que pour toute enveloppe a, à partir d'un certain instant, les fonctions de cumul des mots appartenant à a entrent dans la zone entre les deux droites et n'en ressortent jamais.

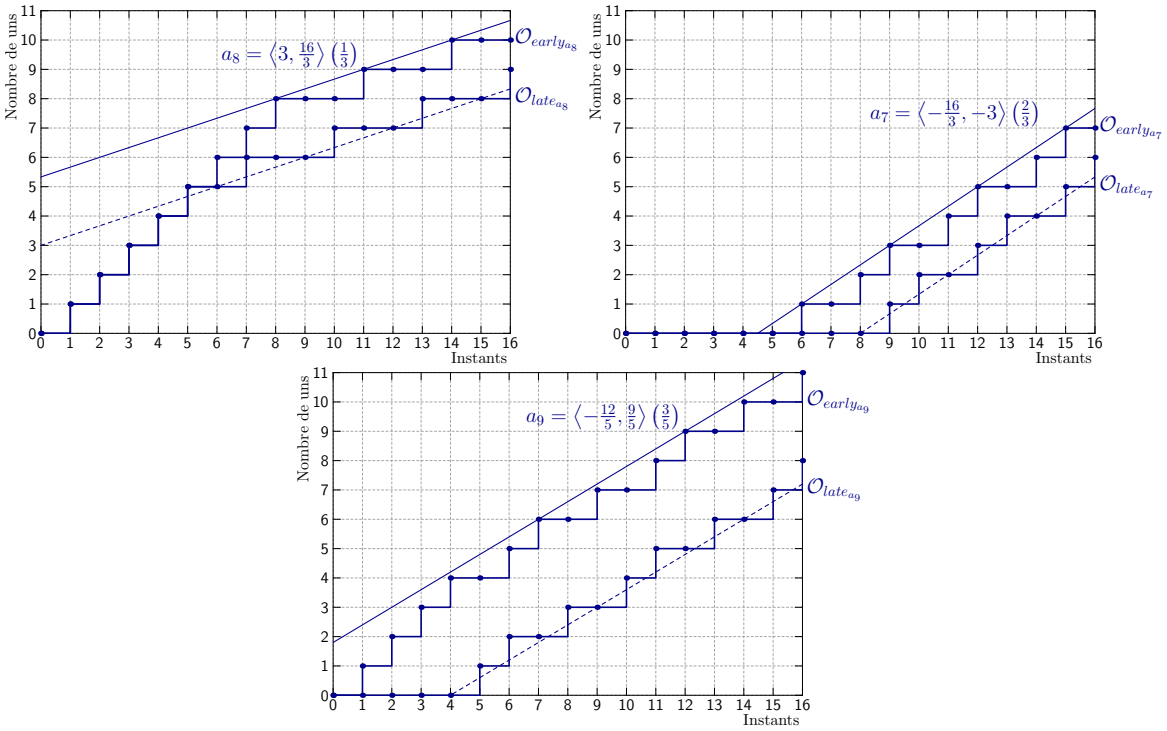


FIG. 7.10 – $early_a$ est le mot qui effectue ses 1 au plus tôt et $late_a$ celui qui effectue ses 1 au plus tard.

7.2.2 Mots au plus tôt et mots au plus tard

On définit $early_a$ comme le mot dont les occurrences des 1 sont effectuées dès que possible, sans violer la contrainte de l'enveloppe a sur l'occurrence des 1. De même, on définit $late_a$ comme le mot dont les occurrences de 1 sont effectuées le plus tard possible, sans violer la contrainte de a sur l'occurrence des 0 (on effectue des 0 tant que possible).

Définition 7.2 ($early_a$, $late_a$). Soit $a = \langle b^0, b^1 \rangle (r)$ une valeur abstraite.

$$early_a[i] \stackrel{def}{=} \begin{cases} 1 & \text{si } \mathcal{O}_{early_a}(i-1) + 1 \leq r \times i + b^1 \\ 0 & \text{sinon} \end{cases} \quad ✿$$

$$late_a[i] \stackrel{def}{=} \begin{cases} 0 & \text{si } \mathcal{O}_{late_a}(i-1) + 0 \geq r \times i + b^0 \\ 1 & \text{sinon} \end{cases} \quad ✿$$

La figure 7.10 représente les mots au plus tôt et au plus tard des enveloppes a_7, a_8 et a_9.

$early_a$ est le mot le plus précoce (au sens de $\preceq$) respectant la contrainte sur les 1 et $late_a$ est le mot le plus tardif respectant la contrainte sur les 0. Tous les mots appartenant à la concrétisation de a respectent la contrainte sur les 1, donc ils sont précédés de $early_a$. Ils respectent aussi la contrainte sur les 0, donc ils précèdent $late_a$. Réciproquement, tout mot compris entre $early_a$ et $late_a$ appartient à la concrétisation de a. On peut donc redéfinir ainsi l'ensemble de concrétisation :

Proposition 7.1 (ensemble de concrétisation).

$$concr(a) = \{w \mid early_a \preceq w \preceq late_a\}$$

Démonstration :

$\subseteq$) Montrons que $\forall w \in \mathit{concr}\,(a)\,,\ \mathit{early}_a \preceq w \preceq \mathit{late}_a$.✿ ✿

On démontre par induction que $\forall w \in \mathit{concr}\,(a)\,,\ \forall i \geq 0,\ \mathcal{O}_{\mathit{early}_a}(i) \geq \mathcal{O}_w(i)$.
Cas de base: $\mathcal{O}_{\mathit{early}_a}(0) = 0 = \mathcal{O}_w(0)$.
Supposons que $\mathcal{O}_{\mathit{early}_a}(i) \geq \mathcal{O}_w(i)$, et montrons que $\mathcal{O}_{\mathit{early}_a}(i+1) \geq \mathcal{O}_w(i+1)$.

Cas où $\mathit{early}_a[i+1] = \mathtt{1}$**,** c'est-à-dire $\mathcal{O}_{\mathit{early}_a}(i+1) = \mathcal{O}_{\mathit{early}_a}(i) + 1$.
Comme $\forall i,\ \mathcal{O}_w(i+1) \leq \mathcal{O}_w(i) + 1$,
par hypothèse d'induction on a bien $\mathcal{O}_{\mathit{early}_a}(i+1) \geq \mathcal{O}_w(i+1)$.

Cas où $\mathit{early}_a[i+1] = \mathtt{0}$**,** c'est-à-dire $\mathcal{O}_{\mathit{early}_a}(i+1) = \mathcal{O}_{\mathit{early}_a}(i)$.
Par définition de early_a, on sait que $\mathcal{O}_{\mathit{early}_a}(i) + 1 > r \times (i+1) + b^{\mathtt{1}}$.

Si $w[i+1] = \mathtt{0}$**,** c'est-à-dire $\mathcal{O}_w(i+1) = \mathcal{O}_w(i)$, on peut conclure par hypothèse d'induction.

Si $w[i+1] = \mathtt{1}$**,** c'est-à-dire $\mathcal{O}_w(i+1) = \mathcal{O}_w(i) + 1$, comme $w \in \mathit{concr}\,(a)$, on a $\mathcal{O}_w(i) + 1 \leq r \times (i+1) + b^{\mathtt{1}}$. Comme $\mathcal{O}_{\mathit{early}_a}(i)+1 > r\times(i+1)+b^{\mathtt{1}}$, on sait que $\mathcal{O}_{\mathit{early}_a}(i) > \mathcal{O}_w(i)$, c'est-à-dire que $\mathcal{O}_{\mathit{early}_a}(i) \geq \mathcal{O}_w(i) + 1$. On peut donc conclure que $\mathcal{O}_{\mathit{early}_a}(i+1) \geq \mathcal{O}_w(i+1)$.

On peut prouver de la même manière que $\forall w \in \mathit{concr}\,(a)\,,\ \forall i \geq 0,\ \mathcal{O}_w(i) \geq \mathcal{O}_{\mathit{late}_a}(i)$.
Cas de base: $\mathcal{O}_w(0) = 0 = \mathcal{O}_{\mathit{late}_a}(0)$.
Supposons que $\mathcal{O}_w(i) \geq \mathcal{O}_{\mathit{late}_a}(i)$, et montrons que $\mathcal{O}_w(i+1) \geq \mathcal{O}_{\mathit{late}_a}(i+1)$.

Cas où $\mathit{late}_a[i+1] = \mathtt{0}$**,** c'est-à-dire $\mathcal{O}_{\mathit{late}_a}(i+1) = \mathcal{O}_{\mathit{late}_a}(i)$.
Comme $\forall i,\ \mathcal{O}_w(i+1) \geq \mathcal{O}_w(i)$,
par hypothèse d'induction on a bien $\mathcal{O}_w(i+1) \geq \mathcal{O}_{\mathit{late}_a}(i+1)$.

Cas où $\mathit{late}_a[i+1] = \mathtt{1}$**,** c'est-à-dire $\mathcal{O}_{\mathit{late}_a}(i+1) = \mathcal{O}_{\mathit{late}_a}(i) + 1$.
Par définition de late_a, on sait que $\mathcal{O}_{\mathit{late}_a}(i) < r \times (i+1) + b^{\mathtt{0}}$.

Si $w[i+1] = \mathtt{1}$**,** c'est-à-dire $\mathcal{O}_w(i+1) = \mathcal{O}_w(i) + 1$, on peut conclure par hypothèse d'induction.

Si $w[i+1] = \mathtt{0}$**,** c'est-à-dire $\mathcal{O}_w(i+1) = \mathcal{O}_w(i)$, comme $w \in \mathit{concr}\,(a)$, on a $\mathcal{O}_w(i) \geq r \times (i+1) + b^{\mathtt{0}}$. Comme $\mathcal{O}_{\mathit{late}_a}(i) < r \times (i+1) + b^{\mathtt{0}}$, on sait que $\mathcal{O}_w(i) > \mathcal{O}_{\mathit{late}_a}(i)$, c'est-à-dire que $\mathcal{O}_w(i) \geq \mathcal{O}_{\mathit{late}_a}(i) + 1$. On peut donc conclure que $\mathcal{O}_w(i+1) \geq \mathcal{O}_{\mathit{late}_a}(i+1)$.

$\supseteq$) Montrons maintenant que $\forall w, \mathit{early}_a \preceq w \preceq \mathit{late}_a \Rightarrow w \in \mathit{concr}\,(a)$.
Soient $a = \langle b^{\mathtt{0}}, b^{\mathtt{1}} \rangle\,(r)$ et w tel que $\mathit{early}_a \preceq w \preceq \mathit{late}_a$, c'est-à-dire tel que $\forall i \geq 0,\ \mathcal{O}_{\mathit{early}_a}(i) \geq \mathcal{O}_w(i) \geq \mathcal{O}_{\mathit{late}_a}(i)$. Montrons que $w \in \mathit{concr}\,(a)$, c'est-à-dire que : $\forall i \geq 1,\ w[i] = \mathtt{1} \Rightarrow \mathcal{O}_w(i) \leq r \times i + b^{\mathtt{1}}$ et $w[i] = \mathtt{0} \Rightarrow \mathcal{O}_w(i) \geq r \times i + b^{\mathtt{0}}$.

Cas où $w[i] = \mathtt{1}$

Si $\mathit{early}_a[i] = \mathtt{1}$**,** par définition de early_a, $\mathcal{O}_{\mathit{early}_a}(i) \leq r \times i + b^{\mathtt{1}}$.
Par l'hypothèse que $\forall i \geq 0,\ \mathcal{O}_w(i) \leq \mathcal{O}_{\mathit{early}_a}(i)$, on peut conclure que $\mathcal{O}_w(i) \leq r \times i + b^{\mathtt{1}}$.

Si $\mathit{early}_a[i] = \mathtt{0}$**,** par définition de early_a, $\mathcal{O}_{\mathit{early}_a}(i-1) + 1 \geq r \times i + b^{\mathtt{1}}$.
On a par hypothèse $\mathcal{O}_w(i-1) \leq \mathcal{O}_{\mathit{early}_a}(i-1)$.
Si $\mathcal{O}_w(i-1) = \mathcal{O}_{\mathit{early}_a}(i-1)$, alors on obtient $\mathcal{O}_w(i) > \mathcal{O}_{\mathit{early}_a}(i)$ qui contredit

l'hypothèse, donc $\mathcal{O}_w(i-1) < \mathcal{O}_{early_a}(i-1)$. Comme $\mathcal{O}_w(i-1) \geq 0$, on a $\mathcal{O}_{early_a}(i-1) > 0$, donc $\exists i' \leq i-1,\ early_a[i'] = \mathtt{1}$. Par définition de $early_a$, $\mathcal{O}_{early_a}(i') \leq r \times i' + b^{\mathtt{1}}$. Par conséquent, d'après le lemme 7.1, $\forall i'' \geq i', \mathcal{O}_{early_a}(i'') \leq r \times i'' + b^{\mathtt{1}}$. En particulier, $\mathcal{O}_{early_a}(i) \leq r \times i + b^{\mathtt{1}}$ et par l'hypothèse $\mathcal{O}_w(i) \leq \mathcal{O}_{early_a}(i)$, on peut en conclure que $\mathcal{O}_w(i) \leq r \times i + b^{\mathtt{1}}$.

Cas où $w[i] = \mathtt{0}$

Si $late_a[i] = \mathtt{0}$**,** par définition de $late_a$, $\mathcal{O}_{late_a}(i) \geq r \times i + b^{\mathtt{0}}$.
Par l'hypothèse que $\forall i \geq 0,\ \mathcal{O}_w(i) \geq \mathcal{O}_{late_a}(i)$, on peut conclure que $\mathcal{O}_w(i) \geq r \times i + b^{\mathtt{0}}$.

Si $late_a[i] = \mathtt{1}$**,** par définition de $late_a$, $\mathcal{O}_{late_a}(i-1) < r \times i + b^{\mathtt{0}}$.
On a par hypothèse $\mathcal{O}_w(i-1) \geq \mathcal{O}_{late_a}(i-1)$.
Si $\mathcal{O}_w(i-1) = \mathcal{O}_{late_a}(i-1)$, alors on obtient $\mathcal{O}_w(i) < \mathcal{O}_{late_a}(i)$ qui contredit l'hypothèse, donc $\mathcal{O}_w(i-1) > \mathcal{O}_{late_a}(i-1)$. Comme $\mathcal{O}_w(i-1) \leq i-1$, on a $\mathcal{O}_{late_a}(i-1) < i-1$, donc $\exists i' \leq i-1,\ late_a[i'] = \mathtt{0}$. Par définition de $late_a$, $\mathcal{O}_{late_a}(i') \geq r \times i' + b^{\mathtt{0}}$. Par conséquent, d'après le lemme 7.1, $\forall i'' \geq i', \mathcal{O}_{late_a}(i'') \geq r \times i'' + b^{\mathtt{0}}$. En particulier, $\mathcal{O}_{late_a}(i) \geq r \times i + b^{\mathtt{0}}$ et par l'hypothèse $\mathcal{O}_w(i) \geq \mathcal{O}_{late_a}(i)$, on peut en conclure que $\mathcal{O}_w(i) \geq r \times i + b^{\mathtt{0}}$.

□

Cela implique que l'ensemble de concrétisation de a est non vide si et seulement si $early_a \preceq late_a$ (on a alors $early_a \in concr\,(a)$ et $late_a \in concr\,(a)$).

Proposition 7.2 (concrétisation non vide). ✿ $\forall a,\ concr\,(a) \neq \varnothing \Leftrightarrow early_a \preceq late_a$.

Démonstration :

$\Leftarrow$) Si $early_a \preceq late_a$,
alors $early_a \in \{w \mid early_a \preceq w \preceq late_a\}$ et $late_a \in \{w \mid early_a \preceq w \preceq late_a\}$.
Donc $early_a \in concr\,(a)$ et $late_a \in concr\,(s)$ et par conséquent $concr\,(a) \neq \varnothing$.

$\Rightarrow$) Si $concr\,(a) \neq \varnothing$,
alors $\{w \mid early_a \preceq w \preceq late_a\} \neq \varnothing$, c'est-à-dire $\exists w,\ early_a \preceq w \preceq late_a$.
Par transitivité de $\preceq$, on en conclu que $early_a \preceq late_a$.

□

On peut aussi déduire de la proposition 7.1 que si l'ensemble de concrétisation n'est pas vide, alors $early_a$ en est l'infimum par rapport à la relation $\preceq$, et $late_a$ en est le supremum.

Proposition 7.3 (infimum et supremum de l'ensemble de concrétisation). ✿ ✿ $\forall a,\ concr\,(a) \neq \varnothing \Rightarrow \sqcap concr\,(a) = early_a \wedge \sqcup concr\,(a) = late_a$

Démonstration :

Par la proposition 7.1, $\forall w \in concr\,(a)\,, early_a \preceq w \preceq late_a$.
Comme $concr\,(a) \neq \varnothing$, par la proposition 7.1, $early_a \in concr\,(a)$ et $late_a \in concr\,(a)$.
Donc $early_a$ est le minimum de l'ensemble de concrétisation de a et $late_a$ en est le maximum.
Par conséquent, $early_a = \sqcap concr\,(a)$ et $late_a = \sqcup concr\,(a)$. □

Dans le cas où la concrétisation de a n'est pas vide, $early_a$ et $late_a$ sont les deux mots "extrêmes" de cette concrétisation. Nous allons donc beaucoup les utiliser dans la suite pour raisonner sur l'ensemble de concrétisation. Pour cette raison, on s'intéresse à la valeur de leur fonction de cumul, en fonction des valeurs de b^0, b^1 et r. Celle-ci sera utilisée de manière déterminante dans les preuves.

Si à l'instant 1 la droite Δ^1 est supérieure à 1 (comme pour a_8 de la figure 7.10), alors la fonction de cumul de $early_a$ va monter à chaque instant, jusqu'à l'atteinte de Δ^1. Durant ce comportement initial $\mathcal{O}_{early_a}(i)$ est donc égal à i, et on reste dans ce comportement tant qu'on est sous Δ^1, c'est-à-dire tant que $i < r \times i + b^1$.

Si à l'instant 1 la droite Δ^1 est inférieure à 1 (comme pour a_7 de la figure 7.10), alors la fonction de cumul de $early_a$ va stagner tant que l'occurrence d'un front montant fait dépasser Δ^1. Durant ce comportement initial $\mathcal{O}_{early_a}(i)$ est donc égal à 0, et on reste dans ce comportement tant que l'occurrence d'un front fait passer au dessus de Δ^1, c'est-à-dire tant que $0 + 1 > r \times i + b^1$.

Les valeurs de $\mathcal{O}_{late_a}$ durant son comportement initial suivent le même principe. Si la fonction de cumul de $late_a$ est initialement sous Δ^0, elle va monter jusqu'à l'atteindre (comme pour a_8 de la figure 7.10), dans le cas contraire, elle va stagner jusqu'à l'atteindre (comme pour a_7 de la figure 7.10).

À partir d'un certain rang, on a $\mathcal{O}_{early_a}(i) = \lfloor r \times i + b^1 \rfloor$. Cela signifie que la courbe de cumul reste au plus près de la droite Δ^1, en restant en dessous.[4] De même, à partir d'un certain rang, $\mathcal{O}_{late_a}(i) = \lceil r \times i + b^0 \rceil$. Cela signifie que la courbe de cumul reste au plus près de la droite Δ^0, en restant au dessus. Si l'on ne considère pas leur préfixe, les mots $early_a$ et $late_a$ sont donc à rapprocher des mots mécaniques [Lot02] inférieurs et supérieurs, qui approchent une droite (ici Δ^1 ou Δ^0) par une courbe prenant des valeurs discrètes.

Les fonctions de cumul de $early_a$ et $late_a$ peuvent donc être calculées ainsi :

Proposition 7.4 (fonctions de cumul de $early_a$ et $late_a$).

Soit une enveloppe $a = \langle b^0, b^1 \rangle (r)$.

La fonction de cumul de $early_a$ prend les valeurs suivantes :

$$\forall i \geq 1,\ \mathcal{O}_{early_a}(i) = \begin{cases} i & \text{si } i < \frac{b^1}{1-r} \quad \text{avec } b^1 > 1 - r \quad \text{(préfixe)} \\ 0 & \text{si } i < -\frac{b^1}{r} \quad \text{avec } b^1 < -r \quad \text{(préfixe)} \\ \lfloor r \times i + b^1 \rfloor & \text{sinon} \end{cases}$$

La fonction de cumul de $late_a$ prend les valeurs suivantes :

$$\forall i \geq 1,\ \mathcal{O}_{late_a}(i) = \begin{cases} i & \text{si } i < \frac{b^0}{1-r} \quad \text{avec } b^0 > 1 - r \quad \text{(préfixe)} \\ 0 & \text{si } i < -\frac{b^0}{r} \quad \text{avec } b^0 < -r \quad \text{(préfixe)} \\ \lceil r \times i + b^0 \rceil & \text{sinon} \end{cases}$$

Démonstration :

Calcul de $\mathcal{O}_{early_a}(i)$:
Par définition de la fonction de cumul, $\mathcal{O}_{early_a}(0) = 0$.

[4] $\lfloor x \rfloor$ (resp. $\lceil x \rceil$) est la notation pour la partie entière par défaut (resp. par excès) de x.

Si $i < \frac{b^1}{1-r}$ **avec** $b^1 > 1 - r$ **:** Par la remarque 7.1, la condition $b^1 > 1 - r$ assure que $r \neq 1$. Pour tout $i < \frac{b^1}{1-r}$, on a $i < r \times i + b^1$. Comme pour tout i, $\mathcal{O}_{early_a}(i-1) \leq i-1$, on sait que pour tout $i < \frac{b^1}{1-r}$, on a $\mathcal{O}_{early_a}(i-1)+1 < r \times i + b^1$. Donc par définition de $early_a$, pour tout i tel que $1 \leq i < \frac{b^1}{1-r}$, $early_a[i] = \texttt{1}$ et donc pour tout $i < \frac{b^1}{1-r}$, $\mathcal{O}_{early_a}(i) = i$.

Si $i < -\frac{b^1}{r}$ **avec** $b^1 < -r$ **:** Pour tout $i < -\frac{b^1}{r}$, on a $0 > r \times i + b^1$. Comme pour tout i, $\mathcal{O}_{early_a}(i-1) \geq 0$, on sait que pour tout $i < -\frac{b^1}{r}$, on a $\mathcal{O}_{early_a}(i-1)+1 > r \times i + b^1$. Donc par définition de $early_a$, pour tout i tel que $1 \leq i < \frac{b^1}{1-r}$, $early_a[i] = \texttt{0}$ et donc pour tout $i < \frac{b^1}{1-r}$, $\mathcal{O}_{early_a}(i) = 0$.

Sinon :

Ici, trois cas sont possibles :

1. $-r \leq b^1 \leq 1 - r$
2. $b^1 > 1 - r$ et $i \geq \frac{b^1}{1-r}$
3. $b^1 < -r$ et $i \geq -\frac{b^1}{r}$

Montrons par récurrence que $\mathcal{O}_{early_a}(i) = \lfloor r \times i + b^1 \rfloor$.

Cas de base si $-r \leq b^1 \leq 1 - r$ **:** $i = 1$

Comme $i = 1$, $0 \leq \lfloor r \times i + b^1 \rfloor \leq 1$ et $\mathcal{O}_{early_a}(i-1) = 0$.

Cas où $\mathcal{O}_{early_a}(i-1) + 1 \leq r \times i + b^1$ **:**

par définition de $early_a$, $early_a[i] = \texttt{1}$. Donc $\mathcal{O}_{early_a}(i) = 1$.
Comme $\mathcal{O}_{early_a}(i-1) + 1 \leq r \times i + b^1$,
on a $1 \leq r \times i + b^1$ donc $1 \leq \lfloor r \times i + b^1 \rfloor$.
Par conséquent, $\lfloor r \times i + b^1 \rfloor = 1$ et $\mathcal{O}_{early_a}(i) = \lfloor r \times i + b^1 \rfloor$.

Cas où $\mathcal{O}_{early_a}(i-1) + 1 > r \times i + b^1$ **:**

par définition de $early_a$, $early_a[i] = \texttt{0}$. Donc $\mathcal{O}_{early_a}(i) = 0$.
Comme $\mathcal{O}_{early_a}(i-1) + 1 > r \times i + b^1$,
on a $1 > r \times i + b^1$ donc $1 > \lfloor r \times i + b^1 \rfloor$.
Par conséquent, $\lfloor r \times i + b^1 \rfloor = 0$ et $\mathcal{O}_{early_a}(i) = \lfloor r \times i + b^1 \rfloor$.

Cas de base si $b^1 > 1 - r$ **et** $i \geq \frac{b^1}{1-r}$ **:** $i = \left\lceil \frac{b^1}{1-r} \right\rceil$

Comme $b^1 > 1 - r$, par la remarque 7.1 on a $r \neq 1$.
Comme $i - 1 < \frac{b^1}{1-r}$, on a $\mathcal{O}_{early_a}(i-1) = i - 1 = \left\lceil \frac{b^1}{1-r} \right\rceil - 1$.

Cas où $\mathcal{O}_{early_a}(i-1) + 1 \leq r \times i + b^1$ **:**

par définition de $early_a$, $early_a[i] = \texttt{1}$.
Donc $\mathcal{O}_{early_a}(i) = \mathcal{O}_{early_a}(i-1) + 1 = i$.
D'une part, $\mathcal{O}_{early_a}(i) = \mathcal{O}_{early_a}(i-1) + 1 \leq r \times i + b^1$.
D'autre part, $i \geq \frac{b^1}{1-r}$, i.e. $i \geq r \times i + b^1$ et donc $\mathcal{O}_{early_a}(i) \geq r \times i + b^1$.
Par conséquent, $\mathcal{O}_{early_a}(i) = r \times i + b^1 = \lfloor r \times i + b^1 \rfloor$.

Cas où $\mathcal{O}_{early_a}(i-1) + 1 > r \times i + b^1$ **:**

par définition de $early_a$, $early_a[i] = \texttt{0}$. Donc $\mathcal{O}_{early_a}(i) = \mathcal{O}_{early_a}(i-1)$.
On sait que $\mathcal{O}_{early_a}(i-1) + 1 > \lfloor r \times i + b^1 \rfloor$,
donc $\mathcal{O}_{early_a}(i) \geq \lfloor r \times i + b^1 \rfloor$.
D'autre part, comme $i - 1 \leq \frac{b^1}{1-r}$,

on a $(i-1) \leq r \times (i-1) + b^1 \leq r \times i + b^1$.
Donc $\mathcal{O}_{early_a}(i) \leq r \times i + b^1$.
Comme $\mathcal{O}_{early_a}(i) \in \mathbb{N}$, on a bien $\mathcal{O}_{early_a}(i) \leq \lfloor r \times i + b^1 \rfloor$.
On peut en conclure que $\mathcal{O}_{early_a}(i) = \lfloor r \times i + b^1 \rfloor$.

Cas de base si $b^1 < -r$ et $i \geq -\frac{b^1}{r}$: $i = \left\lceil -\frac{b^1}{r} \right\rceil$

Comme $b^1 < -r$, par la remarque 7.1 on a $r \neq 0$.
Comme $i - 1 \leq -\frac{b^1}{r}$, on a $\mathcal{O}_{early_a}(i-1) = 0$.

Cas où $\mathcal{O}_{early_a}(i-1) + 1 \leq r \times i + b^1$:

par définition de $early_a$, $early_a[i] = \mathtt{1}$.
Donc $\mathcal{O}_{early_a}(i) = \mathcal{O}_{early_a}(i-1) + 1 \leq r \times i + b^1$.
Comme $\mathcal{O}_{early_a}(i) \in \mathbb{N}$, on a bien $\mathcal{O}_{early_a}(i) \leq \lfloor r \times i + b^1 \rfloor$.
D'autre part, $r \times i + b^1 = r \times \left\lceil -\frac{b^1}{r} \right\rceil + b^1 \leq r \times -\frac{b^1}{r} + 1 + b^1 = 1$.
Comme $\mathcal{O}_{early_a}(i) = \mathcal{O}_{early_a}(i-1) + 1 = 1$,
on a bien $\mathcal{O}_{early_a}(i) \geq r \times i + b^1 \geq \lfloor r \times i + b^1 \rfloor$.
On peut en conclure que $\mathcal{O}_{early_a}(i) = \lfloor r \times i + b^1 \rfloor$.

Cas où $\mathcal{O}_{early_a}(i-1) + 1 > r \times i + b^1$:

par définition de $early_a$, $early_a[i] = \mathtt{0}$.
Donc $\mathcal{O}_{early_a}(i) = \mathcal{O}_{early_a}(i-1) = 0$.
On sait que $\mathcal{O}_{early_a}(i-1) + 1 > r \times i + b^1 \geq \lfloor r \times i + b^1 \rfloor$,
donc comme $\mathcal{O}_{early_a}(i-1) \in \mathbb{N}$, $\mathcal{O}_{early_a}(i) \geq \lfloor r \times i + b^1 \rfloor$.
D'autre part, comme $-\frac{b^1}{r} \leq i$, on a $0 \leq r \times i + b^1$,
donc $\mathcal{O}_{early_a}(i) \leq r \times i + b^1$.
Comme $\mathcal{O}_{early_a}(i) \in \mathbb{N}$, on a bien $\mathcal{O}_{early_a}(i) \leq \lfloor r \times i + b^1 \rfloor$.
On peut en conclure que $\mathcal{O}_{early_a}(i) = \lfloor r \times i + b^1 \rfloor$.

Induction :

Supposons que $\mathcal{O}_{early_a}(i) = \lfloor r \times i + b^1 \rfloor$
et montrons que $\mathcal{O}_{early_a}(i+1) = \lfloor r \times (i+1) + b^1 \rfloor$.

Cas où $\mathcal{O}_{early_a}(i) + 1 \leq r \times (i+1) + b^1$:

par définition de $early_a$, $early_a[i+1] = \mathtt{1}$.
Donc $\mathcal{O}_{early_a}(i+1) = \mathcal{O}_{early_a}(i) + 1 = \lfloor r \times i + b^1 \rfloor + 1$.
D'une part, $\mathcal{O}_{early_a}(i) + 1 \leq \lfloor r \times (i+1) + b^1 \rfloor$
donc $\mathcal{O}_{early_a}(i+1) \leq \lfloor r \times (i+1) + b^1 \rfloor$.
D'autre part, $\mathcal{O}_{early_a}(i+1) = \lfloor r \times i + b^1 \rfloor + 1 \geq \lfloor r \times (i+1) + b^1 \rfloor$.
Par conséquent, on a bien $\mathcal{O}_{early_a}(i+1) = \lfloor r \times (i+1) + b^1 \rfloor$.

Cas où $\mathcal{O}_{early_a}(i) + 1 > r \times (i+1) + b^1$:

par définition de $early_a$, $early_a[i+1] = \mathtt{0}$.
Donc $\mathcal{O}_{early_a}(i+1) = \mathcal{O}_{early_a}(i) = \lfloor r \times i + b^1 \rfloor$.
D'une part, $\mathcal{O}_{early_a}(i+1) = \lfloor r \times i + b^1 \rfloor \leq \lfloor r \times (i+1) + b^1 \rfloor$.
D'autre part, $\mathcal{O}_{early_a}(i) + 1 > \lfloor r \times (i+1) + b^1 \rfloor$,
donc $\mathcal{O}_{early_a}(i+1) \geq \lfloor r \times (i+1) + b^1 \rfloor$.
Par conséquent, on a bien $\mathcal{O}_{early_a}(i+1) = \lfloor r \times (i+1) + b^1 \rfloor$.

Calcul de $\mathcal{O}_{late_a}(i)$:

La preuve de $\mathcal{O}_{late_a}$ suit le même schéma.

□

Les formules données par la proposition 7.4 sont équivalentes aux formules suivantes, qui sont les formules utilisées dans le développement Coq :

$$\begin{array}{lrcl} \forall i, & \mathcal{O}_{early_a}(i) & = & \max(0, \min(i, \lfloor r \times i + b^1 \rfloor)) \\ & \mathcal{O}_{late_a}(i) & = & \max(0, \min(i, \lceil r \times i + b^0 \rceil)) \end{array}$$

Remarque 7.2. Les mots $early_a$ et $late_a$ sont périodiques. Si $a = \langle b^0, b^1 \rangle \left(\frac{n}{\ell}\right)$, leur motif périodique est de longueur ℓ et contient n 1. ◇

Démonstration :

Montrons que $\exists i', \forall i > i', \mathcal{O}_{early_a}(i+l) = \mathcal{O}_{early_a}(i) + n$. Si $-r \leq b^1 \leq 1 - r$, alors $i' = 1$. Si $b^1 > 1 - r$, alors $i' = \left\lceil \frac{b^1}{1-r} \right\rceil$. Si $b^1 < -r$, alors $i' = \left\lceil -\frac{b^1}{r} \right\rceil$. Par la proposition 7.4,
$\forall i > i', \mathcal{O}_{early_a}(i+\ell) = \lfloor r \times (i+\ell) + b^1 \rfloor = \lfloor r \times i + b^1 \rfloor + n = \mathcal{O}_{early_a}(i) + n$ □

Une valeur abstraite peut toujours être normalisée vers la forme $\left\langle \frac{k^0}{\ell}, \frac{k^1}{\ell} \right\rangle \left(\frac{n}{\ell}\right)$ avec $\text{pgcd}(n, \ell) = 1$ sans modifier son ensemble de concrétisation.

Proposition 7.5 (forme normale).
$\forall a = \langle b^0, b^1 \rangle (r)\,,\ \exists a' = \left\langle \frac{k^0}{\ell}, \frac{k^1}{\ell} \right\rangle \left(\frac{n}{\ell}\right)$ avec $\text{pgcd}(n, \ell) = 1$ telle que $concr(a') = concr(a)$.
On a : [5] $\frac{n}{\ell} = \text{red}(r), \qquad k^0 = \lceil b^0 \times \ell \rceil \quad \text{et} \quad k^1 = \lfloor b^1 \times \ell \rfloor$.

Intuitivement, si on a une valeur abstraite $a = \langle b^0, b^1 \rangle (r)$ on réduit la fraction r en $\frac{n}{\ell}$, on remplace b^0 par la plus petite valeur $\frac{k^0}{\ell}$ qui lui est supérieure, et b^1 par la plus grande valeur $\frac{k^1}{\ell}$ qui lui est inférieure.

Par exemple dans la figure 7.11, $a = \left\langle \frac{1}{4}, \frac{7}{4} \right\rangle \left(\frac{1}{2}\right)$ contient le même ensemble de mots que sa forme normale $a' = \left\langle \frac{1}{2}, \frac{3}{2} \right\rangle \left(\frac{1}{2}\right)$. L'ensemble de valeurs discrètes situées entre les deux droites est identique, donc être situé sous Δ^1 est toujours équivalent à être situé sous $\Delta^{1\prime}$, et être situé au dessus de Δ^0 est toujours équivalent à être situé sous $\Delta^{0\prime}$.

Démonstration :

$$concr\left(\langle b^0, b^1 \rangle (r)\right) \stackrel{def}{=} \left\{ w \mid \forall i \geq 1, \quad \wedge \begin{array}{rcl} w[i] = 1 & \Rightarrow & \mathcal{O}_w(i) \leq r \times i + b^1 \\ w[i] = 0 & \Rightarrow & \mathcal{O}_w(i) \geq r \times i + b^0 \end{array} \right\}$$

D'une part, $\mathcal{O}_w(i) \leq r \times i + b^1 \Leftrightarrow \mathcal{O}_w(i) \leq \lfloor \frac{n}{\ell} \times i + b^1 \rfloor$ car $r = \frac{n}{\ell}$ et $\mathcal{O}_w(i) \in \mathbb{N}$.

D'autre part, $\lfloor \frac{n}{\ell} \times i + \frac{k^1}{\ell} \rfloor = \left\lfloor \frac{n \times i + k^1}{\ell} \right\rfloor = \left\lfloor \frac{\lfloor n \times i + b^1 \times \ell \rfloor}{\ell} \right\rfloor = \left\lfloor \frac{n \times i + b^1 \times \ell}{\ell} \right\rfloor = \lfloor \frac{n}{\ell} \times i + b^1 \rfloor$.

De même, $\mathcal{O}_w(i) \geq r \times i + b^0 \Leftrightarrow \mathcal{O}_w(i) \geq \lceil \frac{n}{\ell} \times i + b^0 \rceil$

et $\lceil \frac{n}{\ell} \times i + \frac{k^0}{\ell} \rceil = \left\lceil \frac{n \times i + \lceil b^0 \times \ell \rceil}{\ell} \right\rceil = \left\lceil \frac{\lceil n \times i + b^0 \times \ell \rceil}{\ell} \right\rceil = \left\lceil \frac{n \times i + b^0 \times \ell}{\ell} \right\rceil = \lceil \frac{n}{\ell} \times i + b^0 \rceil$.

Donc

$$\left\{ w \mid \forall i \geq 1, \quad \wedge \begin{array}{rcl} w[i] = 1 & \Rightarrow & \mathcal{O}_w(i) \leq \frac{n'}{\ell'} \times i + b^1 \\ w[i] = 0 & \Rightarrow & \mathcal{O}_w(i) \geq \frac{n'}{\ell'} \times i + b^0 \end{array} \right\}$$

$$= \left\{ w \mid \forall i \geq 1, \quad \wedge \begin{array}{rcl} w[i] = 1 & \Rightarrow & \mathcal{O}_w(i) \leq \frac{n}{\ell} \times i + \frac{k^1}{\ell} \\ w[i] = 0 & \Rightarrow & \mathcal{O}_w(i) \geq \frac{n}{\ell} \times i + \frac{k^0}{\ell} \end{array} \right\}$$

□

[5] $\text{red}(r)$ avec $r \in \mathbb{Q}$ correspond à la forme irréductible de r.

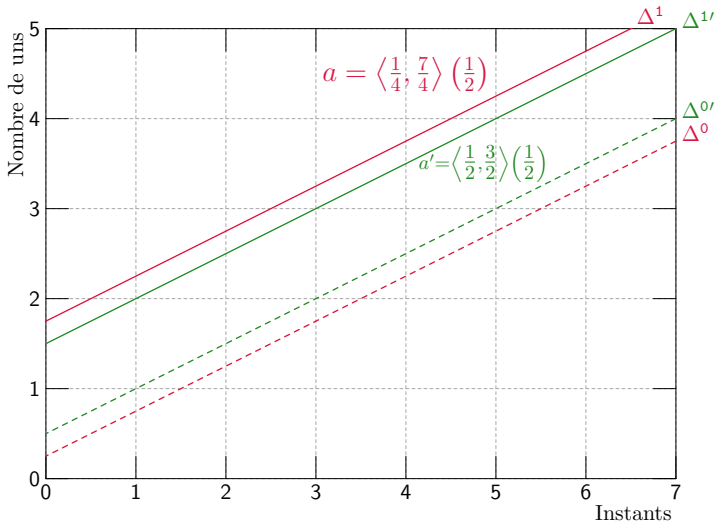


FIG. 7.11 – Normalisation de la valeur abstraite $a = \langle \frac{1}{4}, \frac{7}{4} \rangle (\frac{1}{2})$.

Deux valeurs abstraites de concrétisations non vides ont la même forme normale si et seulement si leurs ensembles de concrétisation sont égaux.

Démonstration :

Soient a_1 et a_2 et leurs formes normales $a'_1 = \left\langle \frac{k^0_1}{\ell_1}, \frac{k^1_1}{\ell_1} \right\rangle \left(\frac{n_1}{\ell_1}\right)$ et $a'_2 = \left\langle \frac{k^0_2}{\ell_2}, \frac{k^1_2}{\ell_2} \right\rangle \left(\frac{n_2}{\ell_2}\right)$.

$\Rightarrow$) Supposons que $a'_1 = a'_2$ et montrons que $concr\,(a_1) = concr\,(a_2)$.
Par la proposition 7.5, $concr\,(a_1) = concr\,(a'_1)$ et $concr\,(a_2) = concr\,(a'_2)$. Comme $concr\,(a'_1) = concr\,(a'_2)$, on peut en déduire que $concr\,(a_1) = concr\,(a_2)$.

$\Leftarrow$) Supposons que $a'_1 \neq a'_2$ et montrons que $concr\,(a_1) \neq concr\,(a_2)$.
Pour cela, il suffit de montrer que $(early_{a'_1} \neq early_{a'_2})$ ou que $(late_{a'_1} \neq late_{a'_2})$. En effet, $(early_{a'_1} \neq early_{a'_2}) \Leftrightarrow early_{a'_1} \not\preceq early_{a'_2} \vee early_{a'_2} \not\preceq early_{a'_1}$. Par la proposition 7.1, cela implique que $early_{a'_1} \notin concr\,(a'_2)$ ou $early_{a'_2} \notin concr\,(a'_1)$. Comme a'_1 et a'_2 sont de concrétisation non vides, $early_{a'_1} \in concr\,(a'_1)$ et $early_{a'_2} \in concr\,(a'_2)$. Donc $concr\,(a'_1) \neq concr\,(a'_2)$ et on peut en conclure que $concr\,(a_1) \neq concr\,(a_2)$. De la même manière, $(late_{a'_1} \neq late_{a'_2})$ implique que $concr\,(a_1) \neq concr\,(a_2)$.
Montrons donc que $(early_{a'_1} \neq early_{a'_2})$ ou que $(late_{a'_1} \neq late_{a'_2})$.
Par la proposition 7.4, à partir d'un certain rang i',

$$\mathcal{O}_{early_{a'_1}}(i) = \left\lfloor \frac{n_1}{\ell_1} \times i + \frac{k^1_1}{\ell_1} \right\rfloor \qquad \mathcal{O}_{late_{a'_1}}(i) = \left\lceil \frac{n_1}{\ell_1} \times i + \frac{k^0_1}{\ell_1} \right\rceil$$

$$\mathcal{O}_{early_{a'_2}}(i) = \left\lfloor \frac{n_2}{\ell_2} \times i + \frac{k^1_2}{\ell_2} \right\rfloor \qquad \mathcal{O}_{late_{a'_2}}(i) = \left\lceil \frac{n_2}{\ell_2} \times i + \frac{k^0_2}{\ell_2} \right\rceil$$

Si $\frac{n_1}{\ell_1} \neq \frac{n_2}{\ell_2}$**,** il existe forcément un $i \geq i'$ tel que $\left\lfloor \frac{n_1}{\ell_1} \times i + \frac{k^1_1}{\ell_1} \right\rfloor \neq \left\lfloor \frac{n_2}{\ell_2} \times i + \frac{k^1_2}{\ell_2} \right\rfloor$.
Donc $\mathcal{O}_{early_{a'_1}}(i) \neq \mathcal{O}_{early_{a'_2}}(i)$ et donc $early_{a'_1} \neq early_{a'_2}$.

Si $\frac{k^1_1}{\ell_1} \neq \frac{k^1_2}{\ell_2}$**,** il existe forcément un $i \geq i'$ tel que $\left\lfloor \frac{n_1}{\ell_1} \times i + \frac{k^1_1}{\ell_1} \right\rfloor \neq \left\lfloor \frac{n_2}{\ell_2} \times i + \frac{k^1_2}{\ell_2} \right\rfloor$.
Donc $early_{a'_1} \neq early_{a'_2}$.

Si $\frac{k^0_1}{\ell_1} \neq \frac{k^0_2}{\ell_2}$**,** il existe forcément un $i \geq i'$ tel que $\left\lceil \frac{n_1}{\ell_1} \times i + \frac{k^0_1}{\ell_1} \right\rceil \neq \left\lceil \frac{n_2}{\ell_2} \times i + \frac{k^0_2}{\ell_2} \right\rceil$.
Donc $late_{a'_1} \neq late_{a'_2}$.

Par conséquent, $concr\,(a'_1) \neq concr\,(a'_2)$.
On peut en conclure que $concr\,(a_1) \neq concr\,(a_2)$.

□

7.2.3 Taille des enveloppes et opérations ensemblistes

On peut déduire des formules des fonctions de cumul de $early_a$ et $late_a$ un test assurant que $early_a \preceq late_a$, c'est-à-dire que l'ensemble de concrétisation de a est non vide [6] :

Proposition 7.6 (test de non vacuité). ✿ ✿

$$\forall a = \left\langle \frac{k^0}{\ell}, \frac{k^1}{\ell} \right\rangle \left(\frac{n}{\ell}\right),\ \frac{k^1}{\ell} - \frac{k^0}{\ell} \geq 1 - \frac{1}{\ell} \Rightarrow concr\,(a) \neq \varnothing$$

De plus, la réciproque est vraie si a est en forme normale (ici c'est-à-dire si $\text{pgcd}(n, \ell) = 1$).

Démonstration :

⇒) Par la proposition 7.4, à partir d'un certain rang i', $\mathcal{O}_{early_a}(i) = \lfloor \frac{n}{\ell} \times i + \frac{k^1}{\ell} \rfloor$ et $\mathcal{O}_{late_a}(i) = \lceil \frac{n}{\ell} \times i + \frac{k^0}{\ell} \rceil$. Or $\lfloor \frac{n}{\ell} \times i + \frac{k^1}{\ell} \rfloor \geq \lfloor \frac{n}{\ell} \times i + \frac{k^0}{\ell} + 1 - \frac{1}{\ell} \rfloor = \lceil \frac{n}{\ell} \times i + \frac{k^0}{\ell} \rceil$.
Donc $\forall i \geq i',\ \mathcal{O}_{early_a}(i) \geq \mathcal{O}_{late_a}(i)$.
La preuve complète (traitant les comportements initiaux) a été faite en Coq.

⇐) Preuve de la réciproque :
Soit $a = \left\langle \frac{k^0}{\ell}, \frac{k^1}{\ell} \right\rangle \left(\frac{n}{\ell}\right)$ avec $\text{pgcd}(n, \ell) = 1$.
$concr\,(a) \neq \varnothing \Rightarrow early_a \preceq late_a \Leftrightarrow \forall i,\ \mathcal{O}_{early_a}(i) \geq \mathcal{O}_{late_a}(i)$.
Par la proposition 7.4, il existe un rang i' tel que $\forall i \geq i',\ \mathcal{O}_{early_a}(i) = \lfloor \frac{n}{\ell} \times i + \frac{k^1}{\ell} \rfloor$ et $\mathcal{O}_{late_a}(i) = \lceil \frac{n}{\ell} \times i + \frac{k^0}{\ell} \rceil$.
Donc $concr\,(a) \neq \varnothing \Rightarrow \forall i \geq i',\ \left\lfloor \frac{n \times i + k^1}{\ell} \right\rfloor \geq \left\lceil \frac{n \times i + k^0}{\ell} \right\rceil$.
Si $\ell = 1$, $\left\lfloor \frac{n \times i + k^1}{\ell} \right\rfloor \geq \left\lceil \frac{n \times i + k^0}{\ell} \right\rceil \Rightarrow k^1 \geq k^0$ et la propriété est vérifiée.
Si $\ell > 1$, comme $\text{pgcd}(n, \ell) = 1$, par le théorème de Bézout il existe $x, y \in \mathbb{Z}$ tels que $n \times y + k^0 = \ell \times x + 1$. Posons $i = y + z \times \ell$ et $x' = x + z \times n$, avec $z \in \mathbb{N}$ tel que $i \geq i'$ et $x' \geq 0$. Pour ce i, on a $n \times i + k^0 = \ell \times x' + 1$.
Comme $\forall i \geq i',\ \left\lfloor \frac{n \times i + k^1}{\ell} \right\rfloor \geq \left\lceil \frac{n \times i + k^0}{\ell} \right\rceil \Leftrightarrow \forall i \geq i',\ \left\lfloor \frac{n \times i + k^0 + (k^1 - k^0)}{\ell} \right\rfloor \geq \left\lceil \frac{n \times i + k^0}{\ell} \right\rceil$,
pour ce i, on a $\left\lfloor \frac{\ell \times x' + 1 + (k^1 - k^0)}{\ell} \right\rfloor \geq \left\lceil \frac{\ell \times x' + 1}{\ell} \right\rceil$, c'est-à-dire $x' + \left\lfloor \frac{1 + (k^1 - k^0)}{\ell} \right\rfloor \geq x' + 1$, c'est-à-dire $\left\lfloor \frac{1 + (k^1 - k^0)}{\ell} \right\rfloor \geq 1$, c'est-à-dire $\frac{1 + (k^1 - k^0)}{\ell} \geq 1$, c'est-à-dire $\frac{k^1}{\ell} - \frac{k^0}{\ell} \geq 1 - \frac{1}{\ell}$ et la propriété est donc vérifiée.

□

[6] On appellera dans la suite *enveloppe non vide* les enveloppes dont l'ensemble de concrétisation n'est pas vide.

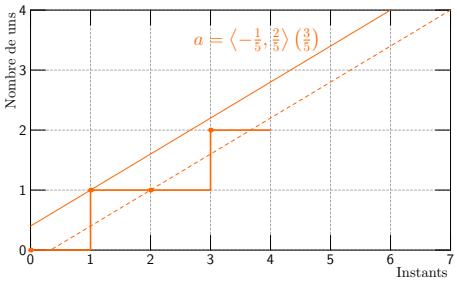


FIG. 7.12 – L'ensemble de concrétisation de a est vide.

En effet, supposons que la valeur abstraite est exprimée sous la forme $\left\langle \frac{k^0}{\ell}, \frac{k^1}{\ell} \right\rangle \left(\frac{n}{\ell}\right)$ de sorte que les valeurs prises par Δ^0 et Δ^1 aux instants discrets sont multiples de $\frac{1}{\ell}$, et supposons que la distance entre Δ^1 et Δ^0 est supérieure ou égale à $1 - \frac{1}{\ell}$, comme pour les enveloppes de la figure 7.7 page 135. Si la courbe de cumul se trouve sous Δ^0, l'occurrence d'un `0` est interdite mais comme on se trouve à une distance d'au moins $\frac{1}{\ell}$ sous Δ^0, on se trouve forcément à une distance d'au moins 1 sous Δ^1 et l'occurrence d'un `1` est donc autorisée. De même, si la courbe se trouve trop près de Δ^1 pour pouvoir effectuer un front montant (c'est-à-dire à une distance de moins de 1), l'occurrence d'un `1` est interdite, mais comme on se trouve à une distance d'au maximum $1 - \frac{1}{\ell}$ sous Δ^1, alors on se trouve forcément au dessus de Δ^0, et l'occurrence d'un `0` est alors autorisée.

La réciproque est vraie lorsque a est en forme normale : si $\frac{k^1}{\ell} - \frac{k^0}{\ell} < 1 - \frac{1}{\ell}$, alors l'ensemble de concrétisation est vide. Par exemple, sur la figure 7.12, il n'y a pas de choix valide à l'instant 4. La courbe se trouve sous Δ^0, donc l'occurrence d'un `0` est interdite, et l'occurrence d'un `1` entraînerait de dépassement de Δ^1, donc elle est aussi interdite.

La réciproque est fausse lorsque a n'est pas en forme normale. Par exemple, l'enveloppe $a_1'' = \langle 0, \frac{8}{10} \rangle \left(\frac{6}{10}\right)$ à le même ensemble de concrétisation que $a_1 = \langle 0, \frac{4}{5} \rangle \left(\frac{3}{5}\right)$, qui est non vide, mais $\frac{8}{10} - 0 \not\geq 1 - \frac{1}{10}$.

Remarque 7.3. Ce test n'est correct que si les valeurs de l'abstraction sont au même dénominateur. Si elles ne sont pas au même dénominateur, on peut utiliser le test correct mais moins précis suivant: $b^1 - b^0 \geq 1 \Rightarrow concr\left(\langle b^0, b^1 \rangle (r)\right) \neq \varnothing$ ◇

La concrétisation ne contient qu'un mot (comme pour l'enveloppe a_1 de la figure 7.7) si $\frac{k^1}{\ell} - \frac{k^0}{\ell} = 1 - \frac{1}{\ell}$ et seulement dans ce cas si a est en forme normale. En effet dans ce cas, pour tout i, la courbe de cumul ne peut être à la fois au dessus de Δ^0 et à une distance de 1 sous Δ^1. Un seul choix est donc possible à chaque instant. On peut vérifier que pour tout i, $\mathcal{O}_{early_a}(i) = \mathcal{O}_{late_a}(i)$ et donc que $early_a = late_a$ est le seul mot appartenant à l'enveloppe.

Si $\frac{k^1}{\ell} - \frac{k^0}{\ell} > 1 - \frac{1}{\ell}$ et a est en forme normale, alors l'ensemble est infini (comme par exemple pour l'enveloppe a_2 de la figure 7.7 et pour les enveloppes de la figure 7.8).

En effet, dans ce cas il existe un i tel que $\mathcal{O}_{early}(i) > \mathcal{O}_{late}(i)$. Pour ce i, il y a plusieurs choix d'éléments possibles. De plus s'il existe un tel i, il en existe une infinité donc une infinité de points de choix. On peut vérifier sur l'enveloppe a_2 de la figure 7.7 que par exemple à l'instant 8, on a le choix entre effectuer ou non un front montant. Il en est de même pour tous les instants égaux à $8 + 5 * x$ avec $x \in \mathbb{N}$, comme l'instant 13.

Nous verrons dans la section 7.2.4 une représentation finie des ensembles de concrétisation.

Relation d'inclusion des enveloppes Définissons un ordre partiel sur les enveloppes, tel que l'inclusion des enveloppes corresponde à l'inclusion des ensembles de mots représentés :

Définition 7.3 (relation d'inclusion $\sqsubseteq^\sim$). $a_1 \sqsubseteq^\sim a_2 \overset{def}{\Leftrightarrow} concr(a_1) \subseteq concr(a_2)$

Cette relation d'ordre peut être testée efficacement :

Proposition 7.7 (test d'inclusion). Soient $a_1 = \langle {b^0}_1, {b^1}_1 \rangle (r_1)$ et $a_2 = \langle {b^0}_2, {b^1}_2 \rangle (r_2)$ deux enveloppes non vides. Alors :

$$r_1 = r_2 \text{ et } [{b^0}_1, {b^1}_1] \subseteq [{b^0}_2, {b^1}_2] \Rightarrow a_1 \sqsubseteq^\sim a_2$$

De plus, si a_1 est en forme normale alors la réciproque est vraie.
Si a_1 n'est pas en forme normale, une partie de la réciproque est vraie : $a_1 \sqsubseteq^\sim a_2 \Rightarrow r_1 = r_2$.

Démonstration :

Par la définition 7.3, $a_1 \sqsubseteq^\sim a_2 \Leftrightarrow concr(a_1) \subseteq concr(a_2)$.

⇒) Montrons que si $concr(a_1) \neq \varnothing$ et $concr(a_2) \neq \varnothing$,
alors $r_1 = r_2 \wedge [{b^0}_1, {b^1}_1] \subseteq [{b^0}_2, {b^1}_2] \Rightarrow concr(a_1) \subseteq concr(a_2)$.
Supposons que $r_1 = r_2 \wedge [{b^0}_1, {b^1}_1] \subseteq [{b^0}_2, {b^1}_2]$. On peut montrer en utilisant la proposition 7.4 que $\forall i \geq 1,\ \mathcal{O}_{early_{a_1}}(i) \geq \mathcal{O}_{early_{a_2}}(i)$ et $\mathcal{O}_{late_{a_1}}(i) \geq \mathcal{O}_{late_{a_2}}(i)$. Donc $early_{a_2} \preceq early_{a_1}$ et $late_{a_1} \preceq late_{a_2}$. Comme a_1 est non vide, on sait de plus par la proposition 7.2 que $early_{a_1} \preceq late_{a_1}$, donc $early_{a_2} \preceq early_{a_1} \preceq late_{a_1} \preceq late_{a_2}$ et on peut en conclure par la proposition 7.1 que tout mot appartenant à $concr(a_1)$ appartient à $concr(a_2)$.

⇐) Montrons que si $concr(a_1) \neq \varnothing$, $concr(a_2) \neq \varnothing$, et a_1 est en forme normale, alors $concr(a_1) \subseteq concr(a_2) \Rightarrow r_1 = r_2 \wedge [{b^0}_1, {b^1}_1] \subseteq [{b^0}_2, {b^1}_2]$.
Supposons que $concr(a_1) \subseteq concr(a_2)$.
Alors par les propositions 7.1 et 7.2, $\forall i \geq 1, early_{a_2} \preceq early_{a_1} \preceq late_{a_1} \preceq late_{a_2}$,
c'est-à-dire $\forall i \geq 1, \mathcal{O}_{early_{a_2}}(i) \geq \mathcal{O}_{early_{a_1}}(i) \geq \mathcal{O}_{late_{a_1}}(i) \geq \mathcal{O}_{late_{a_2}}(i)$.
Donc par la proposition 7.4, pour tout i supérieur à un certain i',
$\lfloor r_2 \times i + {b^1}_2 \rfloor \geq \lfloor r_1 \times i + {b^1}_1 \rfloor \geq \lceil r_1 \times i + {b^0}_1 \rceil \geq \lceil r_2 \times i + {b^0}_2 \rceil$. Par conséquent, $r_1 = r_2$. Soit $\frac{n}{\ell} = r_1$. Comme a_1 est en forme normale, $\text{pgcd}(n, \ell) = 1$, ${b^0}_1 = \frac{k^0_1}{\ell}$ et ${b^1}_1 = \frac{k^1_1}{\ell}$. Donc $\forall i \geq i',\ \lfloor \frac{n}{\ell} \times i + {b^1}_2 \rfloor \geq \left\lfloor \frac{n}{\ell} \times i + \frac{k^1_1}{\ell} \right\rfloor \geq \left\lceil \frac{n}{\ell} \times i + \frac{k^0_1}{\ell} \right\rceil \geq \lceil \frac{n}{\ell} \times i + {b^0}_2 \rceil$.
Tout comme pour la preuve du test de non vacuité, comme $\text{pgcd}(n, \ell) = 1$,
on peut utiliser le théorème de Bézout pour déduire qu'il existe un $i \geq i'$ tel que $\left\lfloor \frac{n}{\ell} \times i + \frac{k^1_1}{\ell} \right\rfloor = \frac{n}{\ell} \times i + \frac{k^1_1}{\ell}$. Pour ce i, $\lfloor \frac{n}{\ell} \times i + {b^1}_2 \rfloor \geq \frac{n}{\ell} \times i + \frac{k^1_1}{\ell}$ et on peut en déduire que ${b^1}_2 \geq {b^1}_1$. De même, il existe un $i \geq i'$ tel que $\left\lceil \frac{n}{\ell} \times i + \frac{k^0_1}{\ell} \right\rceil = \frac{n}{\ell} \times i + \frac{k^0_1}{\ell}$. Pour ce i, $\frac{n}{\ell} \times i + \frac{k^0_1}{\ell} \geq \lceil \frac{n}{\ell} \times i + {b^0}_2 \rceil$ et donc ${b^0}_1 \geq {b^0}_2$. Comme $\left\lfloor \frac{n}{\ell} \times i + \frac{k^1_1}{\ell} \right\rfloor \geq \left\lceil \frac{n}{\ell} \times i + \frac{k^0_1}{\ell} \right\rceil$, on sait que ${b^1}_1 \geq {b^0}_1$. On peut donc en conclure que ${b^1}_2 \geq {b^1}_1 \geq {b^0}_1 \geq {b^0}_2$.

□

Graphiquement parlant, $a_1 \sqsubseteq^\sim a_2$ si les lignes représentant l'enveloppe a_1 sont situées entre les lignes représentant l'enveloppe a_2, c'est-à-dire si elles ont la même pente que celles de a_2 ($r_1 = r_2$), et des décalages compris entre les décalages de a_2 ($[{b^0}_1, {b^1}_1] \subseteq [{b^0}_2, {b^1}_2]$).

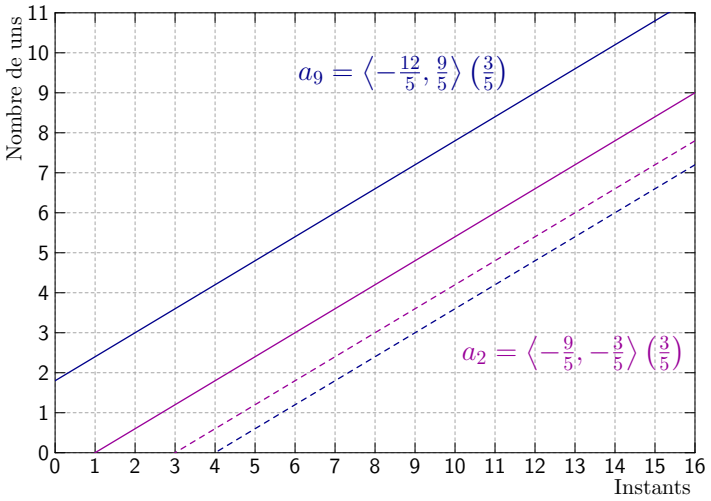


FIG. 7.13 – L'enveloppe a_2 est inclue dans l'enveloppe a_9.

Par exemple sur la figure 7.13, l'enveloppe a_2 est inclue dans l'enveloppe a_9. Tous les mots qui sont dans a_2 sont aussi dans a_9.

La réciproque de la proposition 7.7 est fausse dans le cas où la première enveloppe n'est pas en forme normale. On peut le constater sur la figure 7.11 : puisque les ensembles de concrétisation de a et de sa forme normale a' sont identiques $a \sqsubseteq^{\sim} a'$, mais le test échoue car l'enveloppe a est "artificiellement" plus large.

Remarque 7.4. Il est toujours correct de remplacer une enveloppe $a_1 = \langle b^0{}_1, b^1{}_1 \rangle (r)$ par une enveloppe plus large $a_2 = \langle b^0{}_2, b^1{}_2 \rangle (r)$ avec $[b^0{}_1, b^1{}_1] \subseteq [b^0{}_2, b^1{}_2]$, car dans ce cas le test d'inclusion assure que $a_1 \sqsubseteq^{\sim} a_2$. ◇

Union et intersection d'enveloppes Soit $\top$ une valeur abstraite dont l'ensemble de concrétisation contient tous les mots que l'on peut abstraire et $\bot$ une valeur abstraite de concrétisation vide :

$$\begin{aligned} &\mathit{concr}\,(\top) = \left\{ w \mid \left(\exists b^0, b^1, r,\ w \in \mathit{concr}\left(\langle b^0, b^1 \rangle (r)\right)\right)\right\} \\ &\mathit{concr}\,(\bot) = \varnothing \end{aligned}$$

Pour tout a, $\bot \sqsubseteq^{\sim} a \sqsubseteq^{\sim} \top$.

L'union $\cup^{\sim}$ et l'intersection $\cap^{\sim}$ de deux enveloppes correspondent à l'infimum et au supremum selon la relation d'inclusion $\sqsubseteq^{\sim}$. Ainsi, $a_1 \cup^{\sim} a_2$ est la plus petite enveloppe contenant à la fois les mots de a_1 et les mots de a_2, et $a_1 \cap^{\sim} a_2$ est la plus grande enveloppe dont les mots appartiennent à la fois à a_1 et à a_2.

Proposition 7.8 (union $\cup^{\sim}$, intersection $\cap^{\sim}$).
L'union et l'intersection de deux enveloppes sont définies par :

$$\langle b^0{}_1, b^1{}_1 \rangle (r_1) \cup^{\sim} \langle b^0{}_2, b^1{}_2 \rangle (r_2) = \begin{cases} \langle \min(b^0{}_1, b^0{}_2), \max(b^1{}_1, b^1{}_2) \rangle (r) & \text{si } r_1 = r_2 = r \\ \top & \text{sinon} \end{cases}$$

$$\langle b^0{}_1, b^1{}_1 \rangle (r_1) \cap^{\sim} \langle b^0{}_2, b^1{}_2 \rangle (r_2) = \begin{cases} \langle \max(b^0{}_1, b^0{}_2), \min(b^1{}_1, b^1{}_2) \rangle (r) & \text{si } r_1 = r_2 = r \\ \bot & \text{sinon.} \end{cases}$$

Ces opérations sont correctes et les plus précises.

Démonstration :

Correction D'après le test d'inclusion (proposition 7.7), dans le cas où $r_1 = r_2$ on a : $a_1 \sqsubseteq^\sim (a_1 \cup^\sim a_2)$, $a_2 \sqsubseteq^\sim (a_1 \cup^\sim a_2)$, $(a_1 \cap^\sim a_2) \sqsubseteq^\sim a_1$ et $(a_1 \cap^\sim a_2) \sqsubseteq^\sim a_2$. Comme $\forall a,\ \bot \sqsubseteq^\sim a \sqsubseteq^\sim \top$, ces relation sont aussi vérifiées quand $r_1 \neq r_2$. Donc les opérateurs $\cup^\sim$ et $\cap^\sim$ sont corrects.

Précision

Supposons que $a_1 \cup^\sim a_2 = \langle b^0, b^1 \rangle (r)$. D'après la proposition 7.7, $a_1 \sqsubseteq^\sim (a_1 \cup^\sim a_2) \Rightarrow r_1 = r$ et $a_2 \sqsubseteq^\sim (a_1 \cup^\sim a_2) \Rightarrow r_2 = r$. Si $r_1 \neq r_2$, il n'existe pas d'enveloppe de la forme $(\langle b^0, b^1 \rangle (r))$ telle que $a_1 \sqsubseteq^\sim (a_1 \cup^\sim a_2)$ et $a_2 \sqsubseteq^\sim (a_1 \cup^\sim a_2)$ donc $a_1 \cup^\sim a_2 = \top$ est le résultat le plus précis.
Si $r_1 = r_2$, alors $r = r_1 = r_2$ et :

Si a_1 et a_2 sont en forme normale, d'après la proposition 7.7, on a :
$a_1 \sqsubseteq^\sim (a_1 \cup^\sim a_2) \Rightarrow [b^0{}_1, b^1{}_1] \subseteq [b^0, b^1]$ et $a_2 \sqsubseteq^\sim (a_1 \cup^\sim a_2) \Rightarrow [b^0{}_2, b^1{}_2] \subseteq [b^0, b^1]$.
Donc $[\min(b^0{}_1, b^0{}_2), \max(b^1{}_1, b^1{}_2)] \subseteq [b^0, b^1]$, et la formule donnée pour $a_1 \cup^\sim a_2$ quand $r_1 = r_2$ est bien la plus précise.
On peut en conclure que $\cup^\sim$ est l'opérateur le plus précis sur des opérandes en forme normale. La preuve que $\cap^\sim$ est le plus précis suit le même principe.

Si a_1 et a_2 ne sont pas en forme normale, comparons la forme normale de $a_1 \cup^\sim a_2$ avec le résultat le plus précis, c'est-à-dire avec l'union appliquée aux formes normales de a_1 et a_2.
Les formes normales de a_1 et a_2 sont :
$\left\langle \left\lceil \frac{b^0{}_1 \times \ell_1}{\ell_1} \right\rceil , \left\lfloor \frac{b^1{}_1 \times \ell_1}{\ell_1} \right\rfloor \right\rangle \left(\frac{n_1}{\ell_1}\right)$ avec $\frac{n_1}{\ell_1} = \mathrm{red}(r_1)$
et $\left\langle \left\lceil \frac{b^0{}_2 \times \ell_2}{\ell_2} \right\rceil , \left\lfloor \frac{b^1{}_2 \times \ell_2}{\ell_2} \right\rfloor \right\rangle \left(\frac{n_2}{\ell_2}\right)$ avec $\frac{n_2}{\ell_2} = \mathrm{red}(r_2)$.
Leur union abstraite est : $\left\langle \min(\left\lceil \frac{b^0{}_1 \times \ell_1}{\ell_1} \right\rceil , \left\lceil \frac{b^0{}_2 \times \ell_2}{\ell_2} \right\rceil), \max(\left\lfloor \frac{b^1{}_1 \times \ell_1}{\ell_1} \right\rfloor , \left\lfloor \frac{b^1{}_2 \times \ell_2}{\ell_2} \right\rfloor) \right\rangle (r)$.
D'après le cas précédent de la preuve, il s'agit du résultat le plus précis.
Le résultat de l'union abstraite de a_1 et a_2 est $\langle \max(b^0{}_1, b^0{}_2), \min(b^1{}_1, b^1{}_2) \rangle (r)$.
Sa forme normale est $\left\langle \left\lceil \frac{\max(b^0{}_1, b^0{}_2) \times \ell}{\ell} \right\rceil , \left\lfloor \frac{\min(b^1{}_1, b^1{}_2) \times \ell}{\ell} \right\rfloor \right\rangle \left(\frac{n}{\ell}\right)$ si $\frac{n}{\ell} = \mathrm{red}(r)$.
Comme $\mathrm{red}(r) = \mathrm{red}(r_1) = \mathrm{red}(r_2)$, on a $\ell_1 = \ell_2 = \ell$. Donc la forme normale du résultat est égale au résultat de l'opération sur les formes normales. L'union abstraite est donc l'opération la plus précise, même sur des enveloppes qui ne sont pas en forme normale. La preuve que l'intersection abstraite est la plus précise dans ce cas suit le même principe.

□

Graphiquement parlant, $a_1 \cup^\sim a_2$ correspond à choisir la plus haute des deux droites Δ^1 et la plus basse des droites Δ^0. Par exemple sur la figure 7.21 page 172, l'union des enveloppes a_1 et a_2 vaut $\langle -\frac{9}{5}, \frac{4}{5} \rangle \left(\frac{3}{5}\right)$. L'enveloppe résultat est délimitée par la droite Δ^1 de a_1 et la droite Δ^0 de a_2.

Si les enveloppes n'ont la même pente, il n'y a pas forcément de droite Δ^1 (resp. Δ^0) qui soit plus haute (resp. plus basse) pour tout i. Si toutefois il y a une droite plus haute pour tout i et une droite plus basse pour tout i, alors ces deux droites n'ont pas la même pente et il n'y a donc pas de pente correcte pour que l'enveloppe résultat contienne à la fois les mots de a_1 et les mots de a_2. L'union est donc dans ce cas la valeur abstraite $\top$.

Symétriquement, $a_1 \cap^\sim a_2$ correspond à choisir la plus basse des deux droites Δ^1 et la plus haute des droites Δ^0. Si les enveloppes n'ont pas la même pente, il n'y a aucun mot en commun dans les deux enveloppes, et l'intersection est dans ce cas la valeur abstraite $\perp$.

7.2.4 Enveloppes et automates

Nous avons vu qu'une enveloppe représente un ensemble de mots qui peut être vide, contenir un seul mot ou contenir une infinité de mots. Cet ensemble peut être représenté par un automate fini déterministe qui reconnaît tous les mots de l'ensemble et seulement eux. Nous commençons par définir un automate infini tel que le langage reconnu soit l'ensemble de concrétisation. Puis, nous montrons qu'il est équivalent à un automate fini.

Définition 7.4 (automate associé à une enveloppe).
Soit une enveloppe $a = \langle b^0, b^1 \rangle (r)$. L'automate infini associé à a est $I_a \stackrel{def}{=} \langle Q, \Sigma, \delta, q_o \rangle$ avec :
- l'ensemble d'états Q est un ensemble de paires $(i, j) \in \mathbb{N}^* \times \mathbb{N}$;
- l'état initial q_o est $(1, 0)$;
- l'alphabet Σ est $\{\mathtt{0}, \mathtt{1}\}$;
- la fonction de transition δ est définie par :
  $\delta(\mathtt{1}, (i, j)) = (i + 1, j + 1)$ si $j + 1 \leq r \times i + b^1$
  $\delta(\mathtt{0}, (i, j)) = (i + 1, j + 0)$ si $j + 0 \geq r \times i + b^0$
  la fonction est non définie sinon.

Proposition 7.9. Soit une enveloppe $a = \langle b^0, b^1 \rangle (r)$, et I_a son automate infini associé. Le langage reconnu par I_a est $concr\,(a)$.

Les étiquettes des états correspondent aux coordonnées dans les chronogrammes. La valeur i est l'indice de l'instant courant, et j est le nombre de $\mathtt{1}s$ vus avant l'instant courant ($\mathcal{O}_w(i-1)$). Une transition de l'état (i, j) à l'état $(i+1, j+1)$ correspond à un front montant dans la courbe $\mathcal{O}_w$, c'est-à-dire à l'occurrence d'un $\mathtt{1}$ à l'indice i. Cette transition peut être prise si le haut du front montant, c'est-à-dire le point $(i, j+1)$ est sous la droite Δ^1 d'équation $r \times i + b^1$. Une transition de (i, j) à $(i + 1, j)$ correspond à l'absence de front dans la courbe, c'est-à-dire à l'occurrence d'un $\mathtt{0}$ à l'indice i. Elle est autorisée si le point (i, j) est au-dessus de la droite Δ^0 d'équation $r \times i + b^0$.

Définissons maintenant l'ensemble des états atteignables. Un état (i, j) est atteignable si j est une valeur possible pour $\mathcal{O}_w(i - 1)$, c'est-à-dire (d'après les lemmes 7.2 et 7.1) si une des conditions suivantes est vérifiée :
- Il s'agit d'un instant où la fonction de cumul est dans la zone entre les deux droites : $r \times (i - 1) + b^0 \leq \mathcal{O}_w(i - 1) \leq r \times i - 1 + b^1$.
- Il s'agit d'un instant où la fonction de cumul est sous la zone entre les deux droites : $i - 1 \leq \left\lceil \frac{b^0}{1-r} \right\rceil$ et $\mathcal{O}_w(i - 1) = i - 1$.
- Il s'agit d'un instant où la fonction de cumul est au dessus de la zone entre les deux droites : $i - 1 \leq \lceil -\frac{b^1}{r} \rceil$ et $\mathcal{O}_w(i - 1) = 0$.

On peut donc restreindre l'ensemble d'états à l'ensemble des états atteignables :

$$\begin{aligned} Q \quad = \quad & \left\{(i, j) \in \mathbb{N}^* \times \mathbb{N} \text{ avec } r \times (i - 1) + b^0 \leq j \leq r \times (i - 1) + b^1\right\} \\ & \cup \left\{(i, i - 1) \text{ avec } 1 \leq i \leq \left\lceil \frac{b^0}{1-r} \right\rceil + 1\right\} \\ & \cup \left\{(i, 0) \text{ avec } 1 \leq i \leq \lceil -\frac{b^1}{r} \rceil + 1\right\} \end{aligned}$$

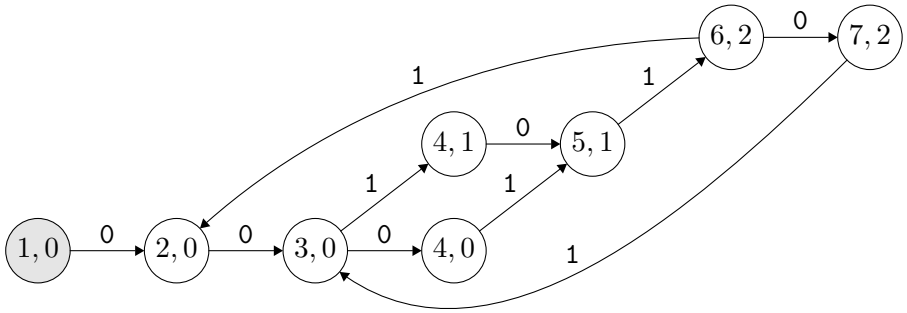


FIG. 7.14 – Automate représentant les mots de la concrétisation de $a_2 = \langle -\frac{9}{5}, -\frac{3}{5} \rangle \left(\frac{3}{5}\right)$.

Cet automate infini peut être transformé en automate fini si l'on remarque que la fonction de transition ne dépend que de la valeur de $j - r \times i$. Tous les états (i, j), (i', j') tels que $j - r \times i = j' - r \times i'$ sont donc équivalents. Comme la première composante de l'ensemble des états atteignables est telle que $b^0 - r \leq j - r \times i \leq b^1 - r$, et que les deuxième et troisième composantes sont des ensembles finis, il existe un nombre fini de classes d'états équivalents. La fonction de transition de l'automate fini A_a équivalent à l'automate infini I_a est :

$$\begin{array}{rcl} \delta(\mathtt{1}, (i, j)) & = & norm(i + 1, j + 1) \text{ si } j + 1 \leq r \times i + b^{\mathtt{1}} \\ \delta(\mathtt{0}, (i, j)) & = & norm(i + 1, j) \text{ si } j \geq r \times i + b^{\mathtt{0}} \end{array}$$

avec $r = \frac{n}{\ell}$, $norm(i, j) = (i - x \times \ell, j - x \times n)$, $x = \max\{x \in \mathbb{N}, (i - x \times \ell \geq 1) \wedge (j - x \times n \geq 0)\}$.

On peut donc représenter par un automate fini les ensembles de concrétisation infinis. La figure 7.14 montre l'automate associé à a_2 de la figure 7.7, et la figure 7.15 montre l'automate associé à a_8 de la figure 7.10.

L'infimum de l'ensemble de concrétisation (selon $\preceq$) correspond au chemin dans l'automate où les transitions `1` sont prises prioritairement. Pour le supremum, ce sont les transitions `0` qui sont prises prioritairement. On peut vérifier sur l'automate associé à a_8 que le chemin prenant les transitions `1` dès que possible correspond à $early_{a_8}$ et le chemin prenant les transitions `0` dès que possible correspond à $late_{a_8}$ (voir figure 7.10).

Dans l'automate représentant une valeur abstraite de taux r, tous les chemins contiennent le même taux asymptotique de `1` (r), c'est-à-dire que prendre une transition `1` ne fait que retarder la transition `0` correspondante et vice-versa. En particulier tous les cycles sont de taux r.

Ces automates sont intéressants car ils permettent de vérifier dynamiquement ou statiquement (avec du *model checking*) qu'une horloge est dans la concrétisation d'une valeur abstraite. Il est à remarquer qu'il n'est pas nécessaire de construire explicitement l'automate. Voici un exemple de programme Lucid Synchrone [Pou06] qui vérifie qu'un mot `w` est abstrait par la valeur $\langle b^0, b^1 \rangle (r)$.

```
let norm { num = n; den = l; } i j =
  if i > l && j >= n then (i - l, j - n) else (i, j)

let node check b0 b1 r w = ok where
  rec i, j = (1,0) fby norm r (i+1) (if w then j + 1 else j)
  and ok = if w then rat_of_int (j + 1) <=.. r *.. (rat_of_int i) +.. b1
           else rat_of_int j >=.. r *.. (rat_of_int i) +.. b0
```

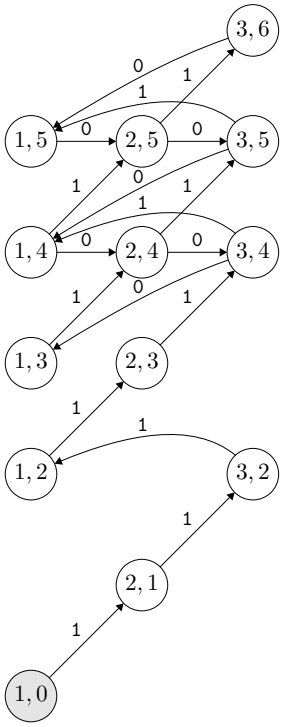


FIG. 7.15 – Automate représentant les mots de la concrétisation de $a_8 = \langle 3, \frac{16}{3} \rangle \left(\frac{1}{3}\right)$.

Les opérateurs suffixés par « `..` » (`+..`, `*..`, `<=..`, etc.) sont les opérateurs sur les rationnels. La fonction `norm` calcule de façon incrémentale la forme normale de l'état `(i,j)` en utilisant le taux $r = \frac{n}{\ell}$. Le nœud `check` prend en entrée une valeur abstraite et un flot de booléens `w`. Il maintient la valeur courante de l'état. L'état est initialisé à la valeur `(1,0)`, puis à chaque instant, le compteur d'indices `i` est incrémenté, et le compteur de la fonction de cumul `j` est incrémenté si l'horloge `w` était vraie à l'instant précédent. La fonction de normalisation est appliquée à ce nouvel état. Puis, suivant la valeur courante de `w`, on vérifie que l'on va faire une transition valide. En suivant le même principe, on peut également générer des horloges contenues dans une enveloppe, par exemple pour faire une simulation.

```
let node generate choice b0 b1 r = w where
  rec i, j = (1,0) fby norm r (i + 1) (if w then j + 1 else j)
  and one = rat_of_int (j + 1) <=.. r *.. (rat_of_int i) +.. b1
  and zero = rat_of_int j >=.. r *.. (rat_of_int i) +.. b0
  and w = choice one zero
```

Le nœud `generate` est paramétré par une fonction `choice` qui choisit quelle transition prendre lorsqu'une transition `1` et une transition `0` sont possibles. Ainsi, on peut par exemple définir les nœuds qui génèrent $early_a$ et $late_a$:

```
let node early b0 b1 r = generate (fun x y -> x) b0 b1 r
let node late b0 b1 r = generate (fun x y -> not y) b0 b1 r
```

`early` choisit une transition `1` dès qu'elle est autorisée, et `late` ne choisit une transition `1` que lorsque la transition `0` est interdite. Il est aussi possible de définir une fonction de choix qui génère des horloges aléatoires dans `a`.

Remarque 7.5. Les nœuds que nous venons de décrire ne peuvent pas encore être programmés en Lucy-n car notre langage ne permet pas de manipuler les horloges comme des flots de données quelconques. En particulier, on ne peut pas prendre une horloge en paramètre d'un nœud. ◇

En utilisant les techniques de *model-checking* fournies par le compilateur Lustre dans la gestion des assertions, on peut écrire un programme Lustre générant des horloges dans une enveloppe donnée en paramètre. Cette expérience réalisée par Pascal Raymond est présentée en annexe G.

7.3 Fonctions d'abstraction

Tout mot binaire infini w restant à une distance bornée de son taux rationnel r peut être abstrait par une enveloppe.

Proposition 7.10 (abstraction d'un mot).
L'abstraction la plus précise de w est $abs\,(w) = \langle b^{\mathtt{0}}, b^{\mathtt{1}} \rangle\,(r)$ avec :

$$\begin{array}{lcl} r & = & rate(w) \;=\; \lim_{i\to+\infty} \frac{\mathcal{O}_w(i)}{i} \\ b^{\mathtt{0}} & = & \min_{i=1..+\infty \text{ avec } w[i]=\mathtt{0}} (\mathcal{O}_w(i) - r \times i) \\ b^{\mathtt{1}} & = & \max_{i=1..+\infty \text{ avec } w[i]=\mathtt{1}} (\mathcal{O}_w(i) - r \times i) \end{array}$$

Démonstration :

L'abstraction est correcte :

Pour tout i tel que $w[i] = \mathtt{0}, b^{\mathtt{0}} \leq \mathcal{O}_w(i) - r \times i$ donc $w[i] = \mathtt{0} \Rightarrow \mathcal{O}_w(i) \geq r \times i + b^{\mathtt{0}}$.

Pour tout i tel que $w[i] = \mathtt{1}, b^{\mathtt{1}} \geq \mathcal{O}_w(i) - r \times i$ donc $w[i] = \mathtt{1} \Rightarrow \mathcal{O}_w(i) \leq r \times i + b^{\mathtt{1}}$.

De plus, elle est la plus précise :

Comme $b^{\mathtt{0}} = \min_{i=1..+\infty \text{ avec } w[i]=\mathtt{0}}(\mathcal{O}_w(i) - r \times i)$,
on sait que $\exists i' \in i = 1..+\infty$ avec $w[i] = \mathtt{0}$ tel que $\mathcal{O}_w(i') - r \times i' = b^{\mathtt{0}}$,
donc $\forall b^{\mathtt{0}\prime} > b^{\mathtt{0}}$, $\exists i' \in i = 1..+\infty$ avec $w[i] = \mathtt{0}$ tel que $\mathcal{O}_w(i') - r \times i' < b^{\mathtt{0}}$.
$b^{\mathtt{0}\prime}$ est par conséquent incorrect.

Le même raisonnement permet de montrer que $b^{\mathtt{1}}$ est le plus petit décalage correct.

□

Pour pouvoir abstraire un mot binaire, il faut donc que $\frac{\mathcal{O}_w(i)}{i}$ admette une limite rationnelle r, et que la distance entre $\mathcal{O}_w(i)$ et $r \times i$ soit bornée.

Par exemple, les mots $\mathtt{1}^1\mathtt{0}^1\,\mathtt{1}^2\mathtt{0}^2\,\mathtt{1}^3\mathtt{0}^3 \ldots \mathtt{1}^n\mathtt{0}^n \ldots$ et $\mathtt{1}^1\mathtt{0}^1\,\mathtt{1}^2\mathtt{0}^2\,\mathtt{1}^4\mathtt{0}^4 \ldots \mathtt{1}^{2^n}\mathtt{0}^{2^n} \ldots$ (représenté sur la figure 7.16) sont de taux asymptotique $\frac{1}{2}$, mais s'en éloignent de manière non bornée. Il ne peuvent donc pas être abstraits avec notre méthode.

7.3.1 Abstraction de mots binaires périodiques

L'abstraction d'un mot binaire ultimement périodique peut être automatiquement calculée. Le taux asymptotique r correspond au rapport entre le nombre de $\mathtt{1}$ du motif périodique

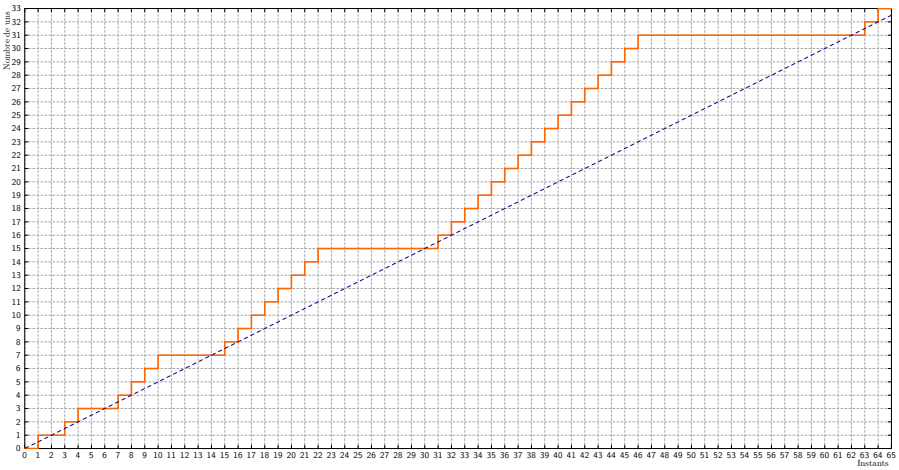


FIG. 7.16 – Le mot $\mathtt{1}^1\mathtt{0}^1\,\mathtt{1}^2\mathtt{0}^2\,\mathtt{1}^4\mathtt{0}^4\ldots\mathtt{1}^{2^n}\mathtt{0}^{2^n}$ ne peut pas être abstrait avec notre méthode.

et sa longueur, comme nous l'avons vu dans le chapitre 4. Il suffit ensuite de parcourir le mot en retenant les décalages minimum (lors de l'occurrence d'un 0) et maximum (lors de l'occurrence d'un 1) du nombre de 1 vus à l'instant i par rapport au nombre de 1 idéal $r \times i$. Ces décalages minimum et maximum sont atteints après le parcours du préfixe et un parcours du motif périodique.

Définition 7.5 (fonction d'abstraction des mots binaires périodiques).
Soit $p = u(v)$ un mot binaire périodique. $abs\,(p) \stackrel{def}{=} \langle b^0, b^1 \rangle\,(r)$ avec :

$$\begin{array}{lcl} r & = & rate(p) \;=\; \frac{|v|_1}{|v|} \\ b^0 & = & \min_{i=1..|u|+|v| \text{ avec } p[i]=0}\,(\mathcal{O}_p(i) - r \times i) \\ b^1 & = & \max_{i=1..|u|+|v| \text{ avec } p[i]=1}\,(\mathcal{O}_p(i) - r \times i) \end{array}$$

Proposition 7.11 (correction et précision).
Cette fonction d'abstraction est correcte : $p \in concr\,(abs\,(p))$.
De plus, il s'agit de la plus précise : $\forall a,\ p \in concr\,(a)\,,\ abs\,(p) \sqsubseteq^{\sim} a$.

Démonstration :

Comme p est périodique, d'après la remarque 4.4, si $i = |u| + x \times |v| + i'$ avec $x \in \mathbb{N},\ i' \leq |v|$, alors on a $\mathcal{O}_p(i) = |u|_1 + x \times |v|_1 + \mathcal{O}_v(i')$.

Par conséquent, d'une part
$$\begin{array}{lcl} r & = & \lim_{i \to +\infty} \frac{\mathcal{O}_p(i)}{i} \\ & = & \lim_{i \to +\infty} \frac{|u|_1 + \left\lfloor \frac{i-|u|}{|v|} \right\rfloor \times |v|_1 + y}{i} \text{ avec } 0 \leq y \leq |v|_1 \\ & = & \lim_{i \to +\infty} \frac{\left\lfloor \frac{i}{|v|} \right\rfloor \times |v|_1}{i} = \frac{|v|_1}{|v|} \end{array}$$

D'autre part
$$\begin{array}{lcl} \forall i \geq |u|, \mathcal{O}_p(i) - r \times i & = & |u|_1 + x \times |v|_1 + \mathcal{O}_v(i') - r \times (|u| + x \times |v| + i') \\ & = & |u|_1 + \mathcal{O}_v(i') - r \times (|u| + i') + x \times |v|_1 - r \times x \times |v| \\ & = & \mathcal{O}_p(|u| + i') - r \times (|u| + i') \end{array}$$

Donc $b^0 = \min_{i=1..+\infty \text{ avec } p[i]=0}(\mathcal{O}_p(i) - r \times i) = \min_{i=1..|u|+|v| \text{ avec } p[i]=0}(\mathcal{O}_p(i) - r \times i)$
et $b^1 = \max_{i=1..+\infty \text{ avec } p[i]=1}(\mathcal{O}_p(i) - r \times i) = \max_{i=1..|u|+|v| \text{ avec } p[i]=1}(\mathcal{O}_p(i) - r \times i)$. □

Par exemple, les mots w_1 et w_2 représentés sur la figure 7.7 (page 135) sont périodiques : $w_1 = \texttt{(11010)}$ et $w_2 = \texttt{0(00111)}$. L'enveloppe a_1 est l'abstraction la plus précise de w_1 et a_2 est l'abstraction la plus précise de w_2. En effet, le motif périodique de w_1 est de taille cinq et contient trois $\texttt{1}$, donc $abs\,(w_1) = \langle {b^{\texttt{0}}}_1, {b^{\texttt{1}}}_1 \rangle\,(r_1)$ avec :

$$\begin{array}{lcllcll} r_1 & = & \frac{3}{5} & & & & \\ {b^{\texttt{0}}}_1 & = & \min_{i=1..5 \text{ avec } w_1[i]=\texttt{0}} & \left(\mathcal{O}_{w_1}(i) - \frac{3}{5} \times i\right) & = & 0 & \text{(atteint à l'instant 5)} \\ {b^{\texttt{1}}}_1 & = & \max_{i=1..5 \text{ avec } w_1[i]=\texttt{1}} & (\mathcal{O}_{w_1}(i) - \frac{3}{5} \times i) & = & \frac{4}{5} & \text{(atteint à l'instant 2)} \end{array}$$

De même, le motif périodique de w_2 est de taille cinq et contient trois $\texttt{1}$, et son préfixe est de taille un, donc $abs\,(w_2) = \langle {b^{\texttt{0}}}_2, {b^{\texttt{1}}}_2 \rangle\,(r_2)$ avec :

$$\begin{array}{lcllcll} r_2 & = & \frac{3}{5} & & & & \\ {b^{\texttt{0}}}_2 & = & \min_{i=1..1+5 \text{ avec } w_2[i]=\texttt{0}} \left(\mathcal{O}_{w_2}(i) - \frac{3}{5} \times i\right) & & = & -\frac{9}{5} & \text{(atteint à l'instant 3)} \\ {b^{\texttt{1}}}_2 & = & \max_{i=1..1+5 \text{ avec } w_2[i]=\texttt{1}} (\mathcal{O}_{w_2}(i) - \frac{3}{5} \times i) & & = & -\frac{3}{5} & \text{(atteint à l'instant 6)} \end{array}$$

Mots périodiques équilibrés Les mots équilibrés ont été introduits par Morse et Hedlund en 1940 [MH40]. Ce sont des mots w, tels que la différence entre le nombre de $\texttt{1}$ dans toute paire de sous-chaînes de w de même taille n'excède pas 1 :

$$\forall i_1, i_2, k \in \mathbb{N}^*,\ |w[i_1..(i_1+k)]|_{\texttt{1}} - |w[i_2..(i_2+k)]|_{\texttt{1}} \leq 1$$

Dit autrement, ce sont des mots dans lesquels les $\texttt{1}$ sont le plus espacés possible sur les instants discrets, tout en respectant un taux donné. Si ce taux est irrationnel, alors le mot équilibré est apériodique, et il est appelé mot de Sturm [Lot02]. Si ce taux est rationnel, alors le mot équilibré est ultimement périodique [AS03]. Ces mots ont été récemment utilisés pour décrire des ordonnancements statiques de systèmes insensibles à la latence en minimisant la taille des ressources de stockage [Mil08].

L'abstraction d'un mot périodique équilibré p produit une valeur abstraite dont la concrétisation ne contient que p. Par exemple, le mot équilibré de taux $\frac{n}{\ell}$ (avec $\text{pgcd}(n, \ell) = 1$) qui n'est jamais en retard par rapport à l'occurrence d'un $\texttt{1}$ tous les $\frac{n}{\ell}$ instants est abstrait par l'enveloppe $\langle 0, 1 - \frac{1}{\ell} \rangle\,(\frac{n}{\ell})$. Elle ne contient que ce mot. Abstraire un tel mot ne constitue donc pas une perte d'information.

C'est le cas de $w_1 = \texttt{(11010)}$ (de la figure 7.7), qui est le mot équilibré de taux $\frac{3}{5}$ ayant ses $\texttt{1}$ au plus tôt. Son abstraction vaut bien $\langle 0, \frac{4}{5} \rangle\,(\frac{3}{5})$ et ne contient que w_1 (car $\frac{4}{5} - 0 = 1 - \frac{1}{5}$, comme expliqué dans la section 7.2.3).

De même, le mot équilibré de taux $\frac{n}{\ell}$ qui n'est jamais en avance par rapport à l'occurrence d'un $\texttt{1}$ tous les $\frac{n}{\ell}$ instants est abstrait par l'enveloppe $\langle -(1 - \frac{1}{\ell}), 0 \rangle\,(\frac{n}{\ell})$ qui ne contient que ce mot. Plus généralement, l'enveloppe $\langle -\frac{k}{\ell}, 1 - \frac{1}{\ell} - \frac{k}{\ell} \rangle\,(\frac{n}{\ell})$ avec $0 \leq k < \ell$ est le mot obtenu par $k \times n$ rotations arrière[7] du mot de l'enveloppe $\langle 0, 1 - \frac{1}{\ell} \rangle\,(\frac{n}{\ell})$ évoquée plus haut. Par exemple, l'enveloppe $\langle -\frac{1}{5}, \frac{3}{5} \rangle\,(\frac{3}{5})$ ne contient que le mot $\texttt{(10110)}$ obtenu par 3 rotations arrière de w_1.

Nous reviendrons sur les mots équilibrés dans la section 7.6, et sur les liens existant entre nos travaux et les ordonnancements statiques de systèmes insensibles à la latence dans le chapitre 10.

[7] La rotation arrière d'un mot périodique $\texttt{(}b.v\texttt{)}$ est le mot $\texttt{(}v.b\texttt{)}$ avec $b \in \{\texttt{0}, \texttt{1}\}$ et v un mot fini. Pour plus de détails, voir [Mil08].

Mots dont le préfixe est uniforme À partir de l'abstraction d'un mot w, on peut calculer facilement l'abstraction des mots de la forme $\mathtt{0}^d.w$ et $\mathtt{1}^d.w$. D'une part, on a :

$$\text{Si } abs\,(w) = \left\langle b^{\mathtt{0}}, b^{\mathtt{1}} \right\rangle (r) \text{ alors } abs\left(\mathtt{0}^d.w\right) = \left\langle \min(b^{\mathtt{0}}, 0) - (d \times r),\ b^{\mathtt{1}} - (d \times r) \right\rangle (r)$$

En effet, ajouter d éléments $\mathtt{0}$ au début d'un mot translate de d instants vers la droite sa fonction de cumul. Il suffit de translater les droites de l'enveloppe de d unités vers la droite, pour que les fronts montants et les absences de front de w restent positionnés correctement par rapport aux droites constituant l'enveloppe. Cette translation horizontale correspond à une translation verticale de $d \times r$ unités vers le bas, d'où la soustraction appliquée aux décalages verticaux initiaux $b^{\mathtt{0}}$ et $b^{\mathtt{1}}$. Par ailleurs, on ajoute des $\mathtt{0}$ au début du mot. Par conséquent, il faut que la droite $\Delta^{\mathtt{0}}$ de l'enveloppe résultat autorise leur occurrence aux instants 1 à d, c'est-à-dire qu'elle prenne des valeurs négatives jusqu'à l'instant d. Elle doit donc avoir un décalage vertical inférieur à $-d \times r$, qui explique le minimum dans la première composante de la formule du résultat.

D'autre part, on a :

$$\text{Si } abs\,(w) = \left\langle b^{\mathtt{0}}, b^{\mathtt{1}} \right\rangle (r) \text{ alors } abs\left(\mathtt{1}^d.w\right) = \left\langle b^{\mathtt{0}} + (d \times (1-r)), \max(b^{\mathtt{1}}, 0) + (d \times (1-r)) \right\rangle (r)$$

Suivant le même principe, ajouter d éléments $\mathtt{1}$ au début d'un mot translate de d instants vers la droite et de d unités vers le haut sa fonction de cumul. Cette translation doit donc être aussi appliquée aux droites constituant l'enveloppe. Les d fronts montants ajoutés doivent avoir lieu sous la droite $\Delta^{\mathtt{1}}$, ce qui explique le maximum dans la formule.

Horloges affines Nous avons vu à la fin du chapitre 4 que les opérations et relations sur les mots binaires sont particulièrement simples à calculer sur les horloges affines, c'est-à-dire les horloges de la forme $\mathtt{0}^d(\mathtt{10}^{\ell-1})$. Ces mots ont aussi des caractéristiques intéressantes dans le cadre de l'abstraction d'horloges. L'abstraction d'une horloge affine $p_a = \mathtt{0}^d(\mathtt{10}^{\ell-1})$ est :

$$\left\langle -\frac{d}{\ell},\ -\frac{d}{\ell} + 1 - \frac{1}{\ell} \right\rangle \left(\frac{1}{\ell}\right)$$

En effet, le mot $(\mathtt{10}^{\ell-1})$ est le mot équilibré de taux $\frac{1}{\ell}$ qui n'est jamais en retard par rapport à l'occurrence d'un $\mathtt{1}$ tous les $\frac{1}{\ell}$ instants. Son abstraction la plus précise est donc $\left\langle 0, 1 - \frac{1}{\ell} \right\rangle \left(\frac{1}{\ell}\right)$. On peut en déduire qu'en lui ajoutant le préfixe uniforme $\mathtt{0}^d$, ce mot est abstrait par $\left\langle 0 - (d \times \frac{1}{\ell}),\ 1 - \frac{1}{\ell} - (d \times \frac{1}{\ell}) \right\rangle \left(\frac{1}{\ell}\right)$.

Tout comme les mots équilibrés, les horloges affines sont abstraites par des enveloppes les contenant comme unique élément.

Nous reviendrons sur l'abstraction d'horloges affines dans la section 7.6.

7.3.2 Abstraction d'un ensemble de mots binaires

Étant donnée une fonction d'abstraction des mots, on peut abstraire un ensemble de mots de la manière suivante :

Définition 7.6 (abstraction d'un ensemble de mots).
$abs\,(\{w_1, \ldots, w_n\}) \stackrel{def}{=} \cup^{\sim}(abs\,(w_1), \ldots, abs\,(w_n))$

Si la fonction d'abstraction des mots est la plus précise, la fonction d'abstraction des ensembles l'est aussi.

Proposition 7.12 (correction et précision).
L'abstraction d'un ensemble de mots est correcte : $\{w_1, \ldots, w_n\} \in concr\,(abs\,(\{w_1, \ldots, w_n\}))$. De plus, cette fonction d'abstraction est la plus précise.

Démonstration :

Si pour tout x de 1 à n, $a_x = abs\,(w_x)$ est la plus petite enveloppe telle que $w_x \in concr(ax_x)$, alors comme l'union d'enveloppes est la plus précise, $a = \cup^\sim(abs\,(w_1), \ldots, abs\,(w_n))$ est la plus petite enveloppe telle que $\forall x = 1..n,\ w_x \in concr(a)$. □

Proposition 7.13. L'abstraction d'un ensemble de mots est égale à l'union des abstractions du supremum et de l'infimum de cet ensemble : $abs\,(W) = abs\,(\sqcap W)\ \cup^\sim abs\,(\sqcup W)$

Démonstration :

Soit $a = abs\,(\sqcap W) \cup^\sim abs\,(\sqcup W)$. D'une part $\sqcap W \in abs\,(\sqcap W)$ donc par définition de $\cup^\sim$, $\sqcap W \in a$. De même, $\sqcup W \in a$. Donc par la proposition 7.1, $early_a \preceq \sqcap W \preceq late_a$ et $early_a \preceq \sqcup W \preceq late_a$. D'autre part, par définition de l'infimum et du supremum, $\forall w \in W,\ \sqcap W \preceq w \preceq \sqcup W$. Donc par transitivité de $\preceq$, $\forall w \in W,\ early_a \preceq w \preceq late_a$. Donc par la proposition 7.1, $\forall w \in W,\ w \in concr\,(a)$.
Par conséquent, a est une enveloppe correcte pour l'ensemble W.

Cette enveloppe est la plus précise, parce que si $abs\,(\sqcap W)$ est la plus petite enveloppe contenant $\sqcap W$ et $abs\,(\sqcup W)$ est la plus petite enveloppe contenant $\sqcup W$, alors $abs\,(\sqcap W) \cup^\sim abs\,(\sqcup W)$ est la plus petite enveloppe contenant ces deux mots. □

On peut remarquer que l'abstraction d'un ensemble de mots contient tous les mots compris entre l'infimum et le supremum de cet ensemble. Elle en contient aussi d'autres. En effet si l'infimum (resp. le supremum) n'est pas l'unique élément de son enveloppe, l'abstraction contient des mots plus petits (resp. plus grand) que celui-ci.

Proposition 7.14. La fonction d'abstraction des ensembles de mots est croissante :

$$W_1 \subseteq W_2 \Rightarrow abs\,(W_1) \sqsubseteq^\sim abs\,(W_2)$$

Démonstration :

Soient W_1 et W_2 des ensembles de mots tels que $W_1 \subseteq W_2$. Par définition, on a :
$abs\,(W_1) = \cup^\sim \{abs\,(w_1)$ avec $w_1 \in W_1\}$
$abs\,(W_2) = \cup^\sim \{abs\,(w_2)$ avec $w_2 \in W_2\}$
$= \cup^\sim(\cup^\sim \{abs\,(w_2)$ avec $w_2 \in W_2 \backslash W_1\}),\ abs\,(W_1)$
Par définition de $\cup^\sim$, $abs\,(W_1) \sqsubseteq^\sim abs\,(W_2)$. □

Étant donnée une fonction d'abstraction des mots correcte et la plus précise (comme la fonction d'abstraction des mots périodiques) et les deux ordres partiels $\subseteq$ et $\sqsubseteq^\sim$, la fonction d'abstraction des ensembles de mots et la fonction de concrétisation forment une correspondance de Galois.

Proposition 7.15 (correspondance de Galois). La fonction d'abstraction des ensembles de mots est monotone, la fonction de concrétisation l'est aussi, et pour tout ensemble de mots W, pour toute enveloppe a, on a :

$$abs\,(W) \sqsubseteq^{\sim} a \Leftrightarrow W \subseteq concr\,(a)$$

Démonstration :

En effet, *abs* est monotone par la proposition 7.14, *concr* est monotone par définition de $\sqsubseteq^{\sim}$.

Montrons que $abs\,(W) \sqsubseteq^{\sim} a \Rightarrow W \subseteq concr\,(a)$:

$$\begin{aligned} abs\,(W) \sqsubseteq^{\sim} a &\Leftrightarrow concr\,(abs\,(W)) \subseteq concr\,(a) \text{ par définition de } \sqsubseteq^{\sim} \\ &\Rightarrow W \subseteq concr\,(a) \text{ car par la proposition 7.12,} W \subseteq concr\,(abs\,(W)) \end{aligned}$$

Montrons que $abs\,(W) \sqsubseteq^{\sim} a \Leftarrow W \subseteq concr\,(a)$:

$W \subseteq concr\,(a) \Rightarrow abs\,(W) \sqsubseteq^{\sim} abs\,(concr\,(a))$ car *abs* est monotone. Comme l'abstraction des ensembles de mots est la plus précise, $abs\,(concr\,(a))$ est la plus petite enveloppe dont la concrétisation contient $concr\,(a)$. Par conséquent, comme a est une enveloppe dont la concrétisation contient $concr\,(a)$, $abs\,(concr\,(a)) \sqsubseteq^{\sim} a$.
Par transitivité de $\sqsubseteq^{\sim}$, on obtient $abs\,(W) \sqsubseteq^{\sim} a$.

□

7.3.3 Abstraction d'un ensemble de mots représentés par un automate

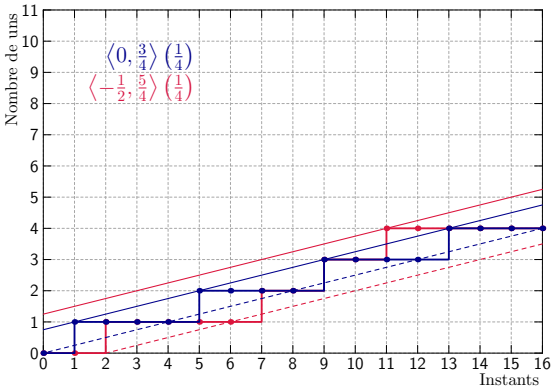
On peut abstraire un ensemble éventuellement infini de mots W si l'on dispose d'un automate fini déterministe A reconnaissant W. L'automate ne doit reconnaître que des mots infinis, c'est-à-dire que tous ses états doivent avoir une transition sortante. De plus, tous les mots reconnus doivent être de même taux asymptotique, c'est-à-dire que tous les cycles de A doivent contenir la même proportion de `1`. Dans ce cas, le mot correspondant au chemin où l'on choisit les transitions à `1` en priorité est l'infimum de l'ensemble de mots reconnus, et le mot correspondant au chemin où l'on choisit les transitions à `0` en priorité en est le supremum. Ces mots sont périodiques, donc on sait calculer leurs valeurs abstraites $a_{\sqcap}$ et $a_{\sqcup}$. Par conséquent, d'après la proposition 7.13, l'abstraction de W peut être calculée par l'union abstraite de $a_{\sqcap}$ et $a_{\sqcup}$.

7.3.4 Les enveloppes en tant que spécifications

Les enveloppes peuvent être obtenues à partir de spécifications fournies par le programmeur, décrivant les horloges concrètes.

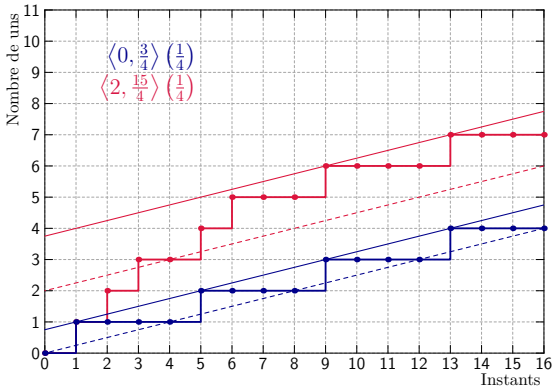
Par exemple, on peut imaginer un opérateur *jitter* tel que $jitter(\texttt{(1000)}, 2)$ représente l'horloge `(1000)` soumise à une gigue de plus ou moins 2 instants, c'est-à-dire telle que les éléments `1` surviennent chacun de 2 instants avant à 2 instants après les instants d'occurrence prévus par l'horloge `(1000)`. Si l'on sait calculer l'abstraction d'un mot, on sait calculer l'abstraction de ce mot soumis à de la gigue :

$$abs\,(jitter(w,d)) = \left\langle b^0 - d \times r,\ b^1 + d \times r \right\rangle (r) \qquad \text{avec } \left\langle b^0, b^1 \right\rangle (r) = abs\,(w)$$



$abs\,(jitter(\texttt{(1000)}, 2)) = \langle -\frac{1}{2}, \frac{5}{4}\rangle\,(\frac{1}{4})$

Rythme de flots présents 1 instant sur 4 et soumis à une gigue de 2 instants.



$abs\,(init(\texttt{(1000)}, 2, 3)) = \langle 2, \frac{15}{4}\rangle\,(\frac{1}{4})$

Rythme de flots présents 1 instant sur 4 avec une avance de 2 à 3 valeurs.

FIG. 7.17 – Représentation de l'avance ou du retard variables d'une horloge, en terme de nombre d'instant ou en terme de nombres de `1`.

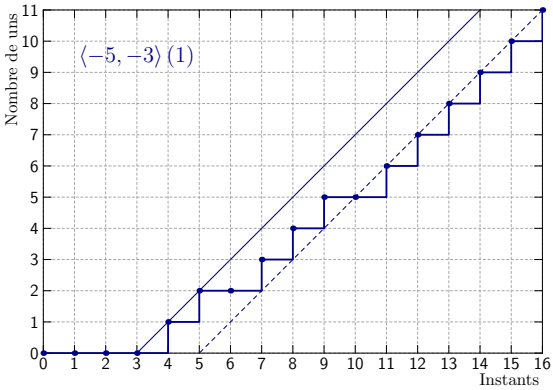
L'enveloppe des mots spécifiés par $jitter(\texttt{(1000)}, 2)$ est donc $\langle 0 - 2 \times \frac{1}{4},\ \frac{3}{4} + 2 \times \frac{1}{4}\rangle\,(\frac{1}{4}) = \langle -\frac{1}{2}, \frac{5}{4}\rangle\,(\frac{1}{4})$. Elle est représentée sur la figure 7.17. On peut introduire un opérateur similaire *delay* qui ajoute un retard variable à une horloge en termes de nombre d'instants. Les éléments `1` de $delay(w, d, D)$ arrivent avec un retard de d à D instants par rapport aux `1` de w. Si $abs\,(w) = \langle b^0, b^1\rangle\,(r)$, alors $abs\,(delay(w, d, D)) = \langle b^0 - D \times r,\ b^1 - d \times r\rangle\,(r)$. La figure 7.18 représente le cas particulier de l'enveloppe $abs\,(delay(\texttt{(1)}, 3, 5)) = \langle -5, -3\rangle\,(1)$, qui contient tous les mots commençant par trois `0`, suivis de `1` à l'infini sauf éventuellement à une ou deux reprises. Ces mots sont utiles pour représenter le rythme de flots qui ne sont pas disponibles dès le premier instant, et qui peuvent être absents à quelques instants au cours de l'exécution.

On peut aussi définir un opérateur *init* tel que $init(w, b, B)$ désigne le rythme d'un flot d'horloge connue w, auquel on a ajouté entre b et B valeurs d'initialisation, disponibles dès le premier instant. Par exemple, sur la figure 7.17, $init(\texttt{(1000)}, 2, 3)$ représente l'horloge `(1000)`, avec de 2 à 3 occurrences de `1` d'avance. Si $abs\,(w) = \langle b^0, b^1\rangle\,(r)$, alors :

$$abs\,(init(w, b, B)) = \langle b^0 + b,\ b^1 + B\rangle\,(r)$$

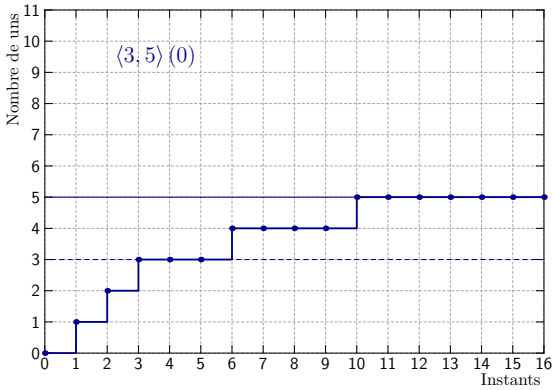
Donc $abs\,(init(\texttt{(1000)}, 2, 3)) = \langle 0 + 2,\ \frac{3}{4} + 3\rangle\,(\frac{1}{4}) = \langle 2, \frac{15}{4}\rangle\,(\frac{1}{4})$. La figure 7.18 représente le cas particulier de l'enveloppe $abs\,(init(\texttt{(0)}, 3, 5)) = \langle 3, 5\rangle\,(0)$, qui contient tous les mots de taux 0, qui commencent par trois `1` et qui valent ensuite `0` pour toujours, sauf éventuellement à deux instants. Ces mots sont utiles pour représenter le rythme de flots qui servent à fournir des valeurs d'initialisation.

Quand les enveloppes sont calculées à partir de spécifications, le typage assure que si l'horloge correspond à sa spécification, alors le programme peut s'exécuter de manière sûre. Les travaux décrits dans la section 7.2.4 prennent tout leur sens dans le cadre des horloges spécifiées par le programmeur, car ils permettent de vérifier statiquement ou dynamiquement qu'une horloge effective est dans l'enveloppe prévue. Ils permettent aussi de générer des horloges dans les enveloppes spécifiées, à des fins de simulation.



$abs\,(delay(\texttt{(1)}, 3, 5)) = \langle -5, -3\rangle\,(1)$

Enveloppe spécifiant les mots de taux 1, qui commencent forcément par trois `0` puis valent `1` à l'infini, sauf éventuellement à 1 ou 2 instants.



$abs\,(init(\texttt{(0)}, 3, 5)) = \langle 3, 5\rangle\,(0)$

Enveloppe spécifiant les mots de taux 0, qui commencent forcément par 3 `1` puis valent `0` à l'infini, sauf éventuellement à 1 ou 2 instants.

FIG. 7.18 – Cas particuliers d'horloges spécifiées de taux 1 et 0

Nous avons présenté dans cette section différentes manières d'obtenir l'abstraction d'un mot ou d'un ensemble de mots. Nous présentons dans la section suivante les opérateurs abstraits, qui permettent de calculer l'abstraction d'une horloge exprimée par des opérations *not* et *on*, à partir de l'abstraction des mots qui la composent.

7.4 Opérateurs abstraits

Dans cette section, nous définissons les opérateurs sur les enveloppes, correspondant aux opérateurs sur les mots définis dans le chapitre 3. Calculer ces opérations dans le domaine abstrait se fait en temps et mémoire constants. Le résultat dans le domaine abstrait contient le résultat de l'opération dans le domaine concret.

7.4.1 Opérateur $on^{\sim}$

Définissons un opérateur *on* abstrait, noté $on^{\sim}$.

Définition 7.7 (opérateur $on^{\sim}$). ✿ Soient [8] ${b^0}_1 \leq 0$ et ${b^0}_2 \leq 0$.

$$\begin{array}{ll} & \langle\ {b^0}_1\ ,\ {b^1}_1\ \rangle\,(\ r_1\ ) \\ on^{\sim} & \langle\ {b^0}_2\ ,\ {b^1}_2\ \rangle\,(\ r_2\ ) \\ \stackrel{def}{=} & \langle\, {b^0}_1 \times r_2 + {b^0}_2\ ,\ {b^1}_1 \times r_2 + {b^1}_2\,\rangle\,(\, r_1 \times r_2\,) \end{array}$$

Cette formule suit le même principe que celle de l'opérateur $on^{\sim}$ présenté en section 7.1. Les éléments de w_1 *on* w_2 sont les éléments de w_1, filtrés par les éléments de w_2. Le taux de `1` présents dans w_1 *on* w_2 est donc le produit du taux de w_1 et du taux de w_2.

[8] Cette contrainte est expliquée section 7.6.2.

Par exemple, $a_2 \ \mathit{on}^{\sim}\ a_7 = \langle -\frac{9}{5}, -\frac{3}{5} \rangle \left(\frac{3}{5}\right) \ \mathit{on}^{\sim}\ \langle -\frac{16}{3}, -3 \rangle \left(\frac{2}{3}\right) = \langle -\frac{103}{15}, -\frac{17}{5} \rangle \left(\frac{2}{5}\right)$. Considérons le mot $w_2 =$ `0(00111)` qui appartient à a_2, et le mot $w'_7 =$ `00000(101)` qui appartient à a_7. Il sont respectivement de taux $\frac{3}{5}$ et $\frac{2}{3}$. Le taux de $w_2 \ \mathit{on}\ w'_7$ est bien $\frac{2}{5}$:

w_2	0.	0	0	1	1	1.	0	0	0	1	1	(	1.	0	0	0	1	1	)
w'_7				0	0	0				0	0	(	1				0	1	)
$w_2 \ \mathit{on}\ w'_7$	0	0	0	0	0	0	0	0	0	0	0	(	1	0	0	0	0	1	)

Les décalages sont eux aussi calculés selon une formule similaire à la formule présentée pour l'abstraction de la section 7.1. Comme on considère ici des décalages en nombre de `1` plutôt qu'en nombre d'instants, ce sont cette fois les décalages du premier mot qui sont multipliés par le taux du mot qui l'échantillonne, et les décalages de ce dernier qui sont reportés tels quels dans le résultat.

Cet opérateur abstrait $\mathit{on}^{\sim}$ a la propriété attendue : pour tout couple de mots des concrétisations respectives, le résultat de l'opération sur les mots appartient à la concrétisation du résultat de l'opération sur les enveloppes.

Proposition 7.16 ($\mathit{on}^{\sim}$ est correct). ✿
$\forall w_1 \in \mathit{concr}\,(a_1)\,,\ \forall w_2 \in \mathit{concr}\,(a_2)\,,\ w_1 \ \mathit{on}\ w_2 \in \mathit{concr}\,(a_1 \ \mathit{on}^{\sim}\ a_2)$

Démonstration :

Soient $w_1 \in \mathit{concr}\,(a_1)$ et $w_2 \in \mathit{concr}\,(a_2)$. Par la proposition 7.1, $\mathit{early}_{a_1} \preceq w_1 \preceq \mathit{late}_{a_1}$ et $\mathit{early}_{a_2} \preceq w_2 \preceq \mathit{late}_{a_2}$. Par le lemme 3.6, l'opérateur on est monotone par rapport à $\preceq$, donc $\mathit{early}_{a_1} \ \mathit{on}\ \mathit{early}_{a_2} \preceq w_1 \ \mathit{on}\ w_2 \preceq \mathit{late}_{a_1} \ \mathit{on}\ \mathit{late}_{a_2}$.
Si l'on montre que $\mathit{early}_{a_1 \ \mathit{on}^{\sim}\ a_2} \preceq \mathit{early}_{a_1} \ \mathit{on}\ \mathit{early}_{a_2}$ et $\mathit{late}_{a_1} \ \mathit{on}\ \mathit{late}_{a_2} \preceq \mathit{late}_{a_1 \ \mathit{on}^{\sim}\ a_2}$, on peut en déduire que $\mathit{early}_{a_1 \ \mathit{on}^{\sim}\ a_2} \preceq w_1 \ \mathit{on}\ w_2 \preceq \mathit{late}_{a_1 \ \mathit{on}^{\sim}\ a_2}$, c'est-à-dire par la proposition 7.1 que $w_1 \ \mathit{on}\ w_2 \in \mathit{concr}\,(a_1 \ \mathit{on}^{\sim}\ a_2)$.

Montrons qu'à partir d'un certain rang $\mathit{early}_{a_1 \ \mathit{on}^{\sim}\ a_2} \preceq \mathit{early}_{a_1} \ \mathit{on}\ \mathit{early}_{a_2}$ c'est-à-dire que $\forall i \geq i'$, $\mathcal{O}_{\mathit{early}_{a_1 \ \mathit{on}^{\sim}\ a_2}}(i) \geq \mathcal{O}_{\mathit{early}_{a_1} \ \mathit{on}\ \mathit{early}_{a_2}}(i)$. Par la proposition 7.4, à partir d'un certain rang, $\mathcal{O}_{\mathit{early}_{a_1 \ \mathit{on}^{\sim}\ a_2}}(i) = \lfloor r_1 \times r_2 \times i + b^1{}_1 \times r_2 + b^1{}_2 \rfloor$. Par la proposition 3.2, $\mathcal{O}_{\mathit{early}_{a_1} \ \mathit{on}\ \mathit{early}_{a_2}}(i) = \mathcal{O}_{\mathit{early}_{a_2}}(\mathcal{O}_{\mathit{early}_{a_1}}(i))$. Donc par la proposition 7.4, à partir d'un certain rang, $\mathcal{O}_{\mathit{early}_{a_1} \ \mathit{on}\ \mathit{early}_{a_2}}(i) = \lfloor r_2 \times (\lfloor r_1 \times i + b^1{}_1 \rfloor) + b^1{}_2 \rfloor$. Comme $\lfloor r_1 \times i + b^1{}_1 \rfloor \leq r_1 \times i + b^1{}_1$, on a $\lfloor r_2 \times (\lfloor r_1 \times i + b^1{}_1 \rfloor) + b^1{}_2 \rfloor \leq \lfloor r_1 \times r_2 \times i + b^1{}_1 \times r_2 + b^1{}_2 \rfloor$. Par conséquent, à partir d'un certain rang, $\mathcal{O}_{\mathit{early}_{a_1 \ \mathit{on}^{\sim}\ a_2}}(i) \geq \mathcal{O}_{\mathit{early}_{a_1} \ \mathit{on}\ \mathit{early}_{a_2}}(i)$.

La preuve qu'à partir d'un certain rang, $\mathit{late}_{a_1} \ \mathit{on}\ \mathit{late}_{a_2} \preceq \mathit{late}_{a_1 \ \mathit{on}^{\sim}\ a_2}$ suit le même principe. La preuve complète (traitant les comportements initiaux) a été faite en Coq. □

Préservation des formes normales

L'opérateur $\mathit{on}^{\sim}$ ne préserve pas la forme normale. On peut remarquer dans l'exemple donné précédemment que $a_2 = \langle -\frac{9}{5}, -\frac{3}{5} \rangle \left(\frac{3}{5}\right)$ et $a_7 = \langle -\frac{16}{3}, -3 \rangle \left(\frac{2}{3}\right)$ sont en forme normale, mais pas $a_2 \ \mathit{on}^{\sim}\ a_7 = \langle -\frac{103}{15}, -\frac{17}{5} \rangle \left(\frac{2}{5}\right)$. Il convient donc après chaque application de $\mathit{on}^{\sim}$ de calculer la forme normale de l'enveloppe résultat (de nombreuses opérations ne sont les plus précises que si les opérandes sont en forme normale).

L'opérateur $\mathit{on}^{\sim}$ préserve la forme prescrite dans la remarque 7.1 (page 137) pour les enveloppes de taux 1, qui doivent avoir des décalages b^0 et b^1 négatifs ou nuls. En effet :

$$\begin{array}{c} a_1 \ \mathit{on}^{\sim} \ a_2 = \left\langle b^0, b^1 \right\rangle (1) \\ \Leftrightarrow \\ a_1 = \left\langle {b^0}_1, {b^1}_1 \right\rangle (1) \ \wedge \ a_2 = \left\langle {b^0}_2, {b^1}_2 \right\rangle (1) \text{ avec } b^0 = {b^0}_1 + {b^0}_2 \text{ et } b^1 = {b^1}_1 + {b^1}_2 \end{array}$$

Par conséquent, si a_1 et a_2 sont dans la forme prescrite, c'est-à-dire si ${b^0}_1 \leq 0$, ${b^1}_1 \leq 0$, ${b^0}_2 \leq 0$ et ${b^1}_2 \leq 0$, alors $a_1 \ \mathit{on}^{\sim} \ a_2$ l'est aussi, c'est-à-dire que $b^0 \leq 0$ et $b^1 \leq 0$.

En ce qui concerne la forme prescrite sur les enveloppes de taux 0, qui doivent avoir des décalages positifs ou nuls, elle n'est que partiellement préservée. En effet :

$$\begin{array}{c} a_1 \ \mathit{on}^{\sim} \ a_2 = \left\langle b^0, b^1 \right\rangle (0) \\ \Leftrightarrow \\ a_1 = \left\langle {b^0}_1, {b^1}_1 \right\rangle (r_1) \ \wedge \ a_2 = \left\langle {b^0}_2, {b^1}_2 \right\rangle (r_2) \quad \begin{array}{ll} \text{avec} & r_1 = 0 \text{ ou } r_2 = 0 \\ \text{et} & b^0 = {b^0}_1 \times r_2 + {b^0}_2,\ b^1 = {b^1}_1 \times r_2 + {b^1}_2 \end{array} \end{array}$$

Dans le cas où $r_2 = 0$, on a $b^0 = {b^0}_2$ et $b^1 = {b^1}_2$. Si a_2 est dans la forme prescrite, c'est-à-dire si ${b^0}_2 \geq 0$ et ${b^1}_2 \geq 0$, alors on peut en déduire que $a_1 \ \mathit{on}^{\sim} \ a_2$ l'est aussi.[9]
Dans le cas où $r_1 = 0$, même si a_1 est dans la forme prescrite, c'est-à-dire si ${b^0}_1 \geq 0$ et ${b^1}_1 \geq 0$, la condition $b^0 \geq 0$ et $b^1 \geq 0$ n'est pas nécessairement vérifiée. Il faut donc dans ce cas rétablir la bonne formation de l'enveloppe avec la formule fournie dans la remarque 7.1.

Éléments neutre et absorbant

L'élément neutre pour l'opérateur on sur les mots binaires est le mot `(1)`. L'abstraction de ce mot est la valeur abstraite $abs\,(\texttt{(1)}) = \langle 0, 0\rangle\,(1)$, qui est neutre à droite et à gauche pour l'opérateur $\mathit{on}^{\sim}$:

$$\begin{array}{rcccl} \left\langle b^0, b^1 \right\rangle (r) \ \mathit{on}^{\sim} \ \langle 0, 0\rangle\,(1) & = & \left\langle b^0 \times 1 + 0,\ b^1 \times 1 + 0 \right\rangle (r \times 1) & = & \left\langle b^0, b^1 \right\rangle (r) \\ \langle 0, 0\rangle\,(1) \ \mathit{on}^{\sim} \ \left\langle b^0, b^1 \right\rangle (r) & = & \left\langle 0 \times r + b^0,\ 0 \times r + b^1 \right\rangle (1 \times r) & = & \left\langle b^0, b^1 \right\rangle (r) \end{array}$$

Abstraire une expression w `on (1)` par $abs\,(w)$ $\mathit{on}^{\sim}$ $abs\,(\texttt{(1)})$ ne fait donc perdre aucune précision par rapport à abstraire w, le résultat de l'expression. Il en est de même pour l'abstraction de `(1) on` w.

L'élément absorbant pour l'opérateur on sur les mots binaires est le mot `(0)`. L'abstraction de ce mot est la valeur abstraite $abs\,(\texttt{(0)}) = \langle 0, 0\rangle\,(0)$ qui est absorbante à droite, mais pas à gauche :

$$\begin{array}{rcccl} \left\langle b^0, b^1 \right\rangle (r) \ \mathit{on}^{\sim} \ \langle 0, 0\rangle\,(0) & = & \left\langle b^0 \times 0 + 0,\ b^1 \times 0 + 0 \right\rangle (r \times 0) & = & \langle 0, 0\rangle\,(0) \\ \langle 0, 0\rangle\,(0) \ \mathit{on}^{\sim} \ \left\langle b^0, b^1 \right\rangle (r) & = & \left\langle 0 \times r + b^0,\ 0 \times r + b^1 \right\rangle (0 \times r) & = & \left\langle b^0, b^1 \right\rangle (0) \end{array}$$

Abstraire une expression w `on (0)` plutôt que son résultat `(0)` n'est donc pas source d'imprécision. Par contre, abstraire une expression `(0) on` w plutôt que son résultat `(0)` fait perdre de l'information. Ceci est dû à une imprécision de notre opérateur. En effet, l'ensemble de concrétisation de $\langle 0, 0\rangle\,(0)$ est le singleton $\{\texttt{(0)}\}$, donc $\forall w_1 \in concr\,(\langle 0, 0\rangle\,(0))$, $\forall w_2 \in concr\left(\left\langle b^0, b^1 \right\rangle (r)\right)$,

[9] Comme ${b^0}_2 \leq 0$ est un pré-requis à l'application de $\mathit{on}^{\sim}$, on a en fait ${b^0}_2 = 0$.

w_1 $\mathtt{on}$ $w_2 = \mathtt{(0)}$. Par conséquent, le résultat le plus précis de $\langle 0,0\rangle(0)$ $\mathtt{on}^\sim$ $\langle b^0, b^1\rangle(r)$ est $\langle 0,0\rangle(0)$, et non pas $\langle b^0, b^1\rangle(0)$.[10]

Cette imprécision de la formule peut être facilement corrigée en rajoutant la règle spécifique suivante à $\mathtt{on}^\sim$: si $a_1 = \langle 0,0\rangle(0)$, alors a_1 $\mathtt{on}^\sim$ $a_2 = \langle 0,0\rangle(0)$. Cette règle améliore la précision de l'opérateur.

Domaine de définition

On peut voir dans la définition 7.7 que l'opérateur $\mathtt{on}^\sim$ que nous proposons n'est applicable qu'à des enveloppes $\langle b^0{}_1, b^1{}_1\rangle(r_1)$ et $\langle b^0{}_2, b^1{}_2\rangle(r_2)$ telles que les décalages $b^0{}_1$ et $b^0{}_2$ sont négatifs ou nuls. Cette restriction est indispensable, car la formule fournie n'est pas correcte lorsque cette condition n'est pas vérifiée.

Notons qu'on peut toujours se ramener à des décalages négatifs ou nuls en élargissant les enveloppes, comme expliqué dans la remarque 7.4. Pour cela, il suffit de remplacer b^0 par $\min(b^0, 0)$. Par exemple, pour calculer la valeur de $\langle 0, \frac{4}{5}\rangle\left(\frac{3}{5}\right)$ $\mathtt{on}^\sim$ $\langle 3, \frac{16}{3}\rangle\left(\frac{1}{3}\right)$, on sur-approxime $\langle 3, \frac{16}{3}\rangle\left(\frac{1}{3}\right)$ par $\langle 0, \frac{16}{3}\rangle\left(\frac{1}{3}\right)$, et on peut alors utiliser la formule donnée dans la définition de $\mathtt{on}^\sim$. Dans la suite, on considérera que cet élargissement est effectué de manière implicite, afin de pouvoir appliquer l'opérateur $\mathtt{on}^\sim$ à toutes les enveloppes.

Cet élargissement constitue une grosse source d'imprécision, car il perd de l'information lorsque le préfixe commence par une série de `1`. Il sera question de ce problème dans la section 7.6. Notons néanmoins que cet élargissement n'est nécessaire qu'à la première application de l'opérateur, car le résultat de $\mathtt{on}^\sim$ appartient à son domaine de définition.

Nous étudions ci-dessous la précision de l'opérateur $\mathtt{on}^\sim$ sur les enveloppes de son domaine de définition, c'est-à-dire sans compter la perte initiale due à un éventuel élargissement.

Précision de l'opérateur abstrait $\mathtt{on}^\sim$

Le résultat correct le plus précis pour a_1 $\mathtt{on}^\sim$ a_2 est la plus petite enveloppe qui contienne l'ensemble de mots $W = \{w$ tels que $w = w_1$ $\mathtt{on}$ w_2 avec $w_1 \in \mathit{concr}(a_1)$ et $w_2 \in \mathit{concr}(a_2)\}$. Contrairement aux opérateurs présentés jusqu'ici (propositions 7.6, 7.7, 7.8, 7.11, 7.12 et 7.13), qui étaient les plus précis sur des opérandes en forme normale, l'opérateur $\mathtt{on}^\sim$ que nous proposons n'est pas l'opérateur correct le plus précis.

Le résultat de a $\mathtt{on}^\sim$ a' se trouve être le résultat le plus précis pour toutes les enveloppes a et a' choisies parmi les exemples donnés depuis le début de ce chapitre et qui appartiennent au domaine de définition de $\mathtt{on}^\sim$ ($b^0 \leq 0$, i.e. a_1, a_2, a_7 et a_9). Cependant, la figure 7.19 exhibe un exemple où a $\mathtt{on}^\sim$ a' n'est pas le résultat le plus précis. Si $a = \langle -\frac{2}{3}, 0\rangle\left(\frac{2}{3}\right)$ et $a' = \langle -\frac{5}{6}, 0\rangle\left(\frac{5}{6}\right)$, alors la forme normale de a $\mathtt{on}^\sim$ a' est $\langle -\frac{12}{9}, 0\rangle\left(\frac{5}{9}\right)$ alors que la plus petite enveloppe contenant l'ensemble des mots w $\mathtt{on}$ w' tels que $w \in \mathit{concr}(a)$ et $w' \in \mathit{concr}(a')$ est $\langle -\frac{11}{9}, 0\rangle\left(\frac{5}{9}\right)$. La partie verte de l'automate représente l'ensemble des mots w $\mathtt{on}$ w' avec $w \in \mathit{concr}(a)$ et $w' \in \mathit{concr}(a')$ (c'est ici un singleton, car les enveloppes opérandes ne contiennent qu'un mot chacune). Les états bleus sont les états rajoutés par l'abstraction de cet ensemble de mots. L'automate formé par les états verts et bleu est donc l'automate associé à la plus petite enveloppe correcte pour a $\mathtt{on}^\sim$ a'. Les états rouges sont les états supplémentaires rajoutés par l'imprécision du calcul abstrait a $\mathtt{on}^\sim$ a'. Le chronogramme montre en bleu les droites Δ^1 et Δ^0 correspondant à l'enveloppe la plus précise. La droite Δ^1 de a $\mathtt{on}^\sim$ a' est la droite la plus

[10] Comme $b^0 \leq 0$ est un pré-requis à l'application de $\mathtt{on}^\sim$ et d'après la remarque 7.1, l'enveloppe $\langle b^0, b^1\rangle(0)$ contient les mêmes mots que l'enveloppe $\langle 0, b^1\rangle(0)$.

précise. Par contre la droite Δ^0 de a $\mathit{on}^{\sim}$ a' est légèrement trop basse, elle est représentée en rouge. Contrairement à l'enveloppe la plus précise, elle permet l'occurrence d'un `0` à l'instant 6, lorsque la fonction de cumul vaut 2.

On sait quantifier l'imprécision du $\mathit{on}^{\sim}$: la droite Δ^1 (resp. Δ^0) calculée peut être au dessus (resp. en dessous) de la droite la plus précise, mais à une distance de moins de 1.

Proposition 7.17 (imprécision de $\mathit{on}^{\sim}$).
Soit $a_1 = \left\langle {b^0}_1, {b^1}_1 \right\rangle (r_1)$ une enveloppe non vide, en forme normale, avec ${b^0}_1 \leq 0$ et $r_1 \neq 0$.
Soit $a_2 = \left\langle {b^0}_2, {b^1}_2 \right\rangle (r_2)$ une enveloppe non vide, avec ${b^0}_2 \leq 0$.
Soit $W = \{w \text{ tels que } w = w_1 \ \mathit{on}\ w_2 \text{ avec } w_1 \in \mathit{concr}\,(a_1) \text{ et } w_2 \in \mathit{concr}\,(a_2)\}$.
Si $\mathit{abs}\,(W) = \left\langle b^0, b^1 \right\rangle (r)$ et $a_1\ \mathit{on}^{\sim}\ a_2 = \left\langle b^{0\prime}, b^{1\prime} \right\rangle (r)$, alors $b^0 - b^{0\prime} < 1$ et $b^{1\prime} - b^1 < 1$.

Démonstration :

Montrons que pour une infinité de i, $\mathcal{O}_{early_{a_1}\ \mathit{on}^{\sim}\ a_2}(i) - \mathcal{O}_{early_{abs(W)}}(i) \leq 0$

À partir d'un certain rang i' :
$\mathcal{O}_{early_{a_1}}(i) = \lfloor r_1 \times i + {b^1}_1 \rfloor$ et $\mathcal{O}_{early_{a_2}}(i) = \lfloor r_2 \times i + {b^1}_2 \rfloor$.

Donc pour tout i à partir de i' :
$\mathcal{O}_{early_{a_1}\ \mathit{on}\ early_{a_2}}(i) = \mathcal{O}_{early_{a_2}}(\mathcal{O}_{early_{a_1}}(i)) = \lfloor r_2 \times \lfloor r_1 \times i + {b^1}_1 \rfloor + {b^1}_2 \rfloor$.

Comme a_1 est en forme normale, il existe une infinité de i tels que
$\lfloor r_1 \times i + {b^1}_1 \rfloor = r_1 \times i + {b^1}_1$ (voir lemme D.2 page 245).
Donc il existe une infinité de i après i' tels que :
$\mathcal{O}_{early_{a_1}\ \mathit{on}\ early_{a_2}}(i) = \lfloor r_2 \times (r_1 \times i + {b^1}_1) + {b^1}_2 \rfloor$.

Or, à partir d'un certain rang i'', on a :
$\mathcal{O}_{early_{(a_1\ \mathit{on}^{\sim}\ a_2)}}(i) = \lfloor r_1 \times r_2 \times i + {b^1}_1 \times r_2 + {b^1}_2 \rfloor$. Donc il existe une infinité de i après $\max(i', i'')$ tels que $\mathcal{O}_{early_{a_1}\ \mathit{on}\ early_{a_2}}(i) = \mathcal{O}_{early_{a_1}\ \mathit{on}^{\sim}\ a_2}(i)$.
Comme $early_{a_1}\ \mathit{on}\ early_{a_2} \in W$, $early_{abs(W)} \preceq early_{a_1}\ \mathit{on}\ early_{a_2}$ et donc $\mathcal{O}_{early_{a_1}\ \mathit{on}\ early_{a_2}}(i) \leq \mathcal{O}_{early_{abs(W)}}(i)$.

Par transitivité, pour une infinité de i après $\max(i', i'')$ on a donc :
$\mathcal{O}_{early_{a_1}\ \mathit{on}^{\sim}\ a_2}(i) \leq \mathcal{O}_{early_{abs(W)}}(i)$, c'est-à-dire $\mathcal{O}_{early_{a_1}\ \mathit{on}^{\sim}\ a_2}(i) - \mathcal{O}_{early_{abs(W)}}(i) \leq 0$.

Montrons que $b^{1\prime} - b^1 - 1 < \mathcal{O}_{early_{a_1}\ \mathit{on}^{\sim}\ a_2}(i) - \mathcal{O}_{early_{abs(W)}}(i)$

Comme à partir du rang i'', $\mathcal{O}_{early_{a_1}\ \mathit{on}^{\sim}\ a_2}(i) = \left\lfloor r \times i + b^{1\prime} \right\rfloor$, on a :
$r \times i + b^{1\prime} - 1 < \mathcal{O}_{early_{a_1}\ \mathit{on}^{\sim}\ a_2}(i) \leq r \times i + b^{1\prime}$.
Comme à partir d'un rang i''', $\mathcal{O}_{early_{abs(W)}}(i) = \lfloor r \times i + b^1 \rfloor$, on a :
$r \times i + b^1 - 1 < \mathcal{O}_{early_{abs(W)}}(i) \leq r \times i + b^1$.

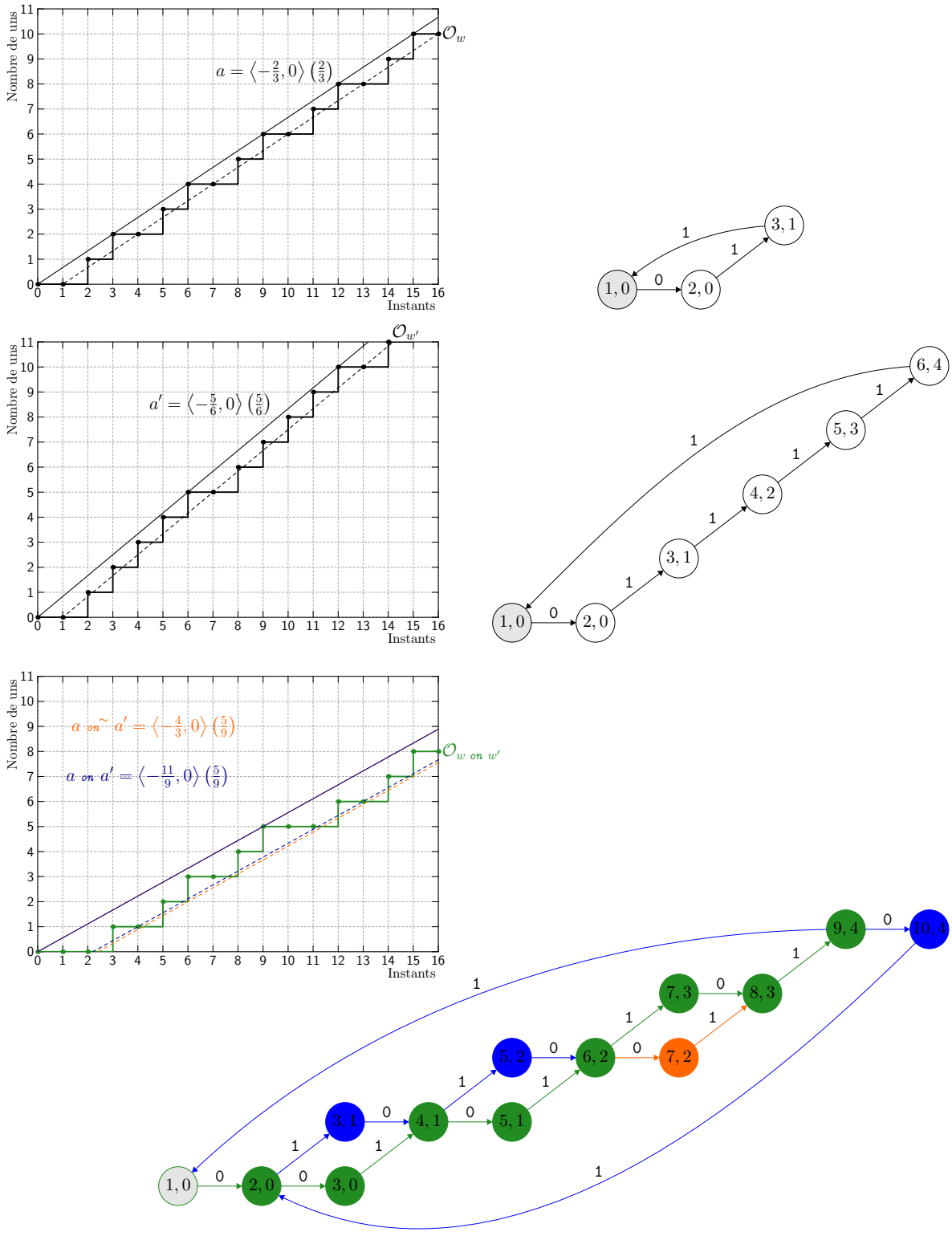
Donc, à partir de $\max(i'', i''')$, on a :
$r \times i + b^{1\prime} - 1 - (r \times i + b^1) < \mathcal{O}_{early_{a_1}\ \mathit{on}^{\sim}\ a_2}(i) - \mathcal{O}_{early_{abs(W)}}(i)$,
c'est-à-dire $b^{1\prime} - b^1 - 1 < \mathcal{O}_{early_{a_1}\ \mathit{on}^{\sim}\ a_2}(i) - \mathcal{O}_{early_{abs(W)}}(i)$.

On obtient donc pour une infinité de i à partir de $\max(i'', i''')$:

$$b^{1\prime} - b^1 - 1 < \mathcal{O}_{early_{a_1}\ \mathit{on}^{\sim}\ a_2}(i) - \mathcal{O}_{early_{abs(W)}}(i) \leq 0$$

On peut en conclure que $b^{1\prime} - b^1 < 1$. La preuve que $b^0 - b^{0\prime} < 1$ suit le même principe.

□



En vert: Ensemble des mots w on w' avec $w \in \mathit{concr}\,(a)$ et $w' \in \mathit{concr}\,(a')$

En vert et bleu: Abstraction de l'ensemble vert

En vert, bleu et rouge: Mots de l'enveloppe a $\mathit{on}^{\sim}$ a'

FIG. 7.19 – Précision de l'opération abstraite $\mathit{on}^{\sim}$.

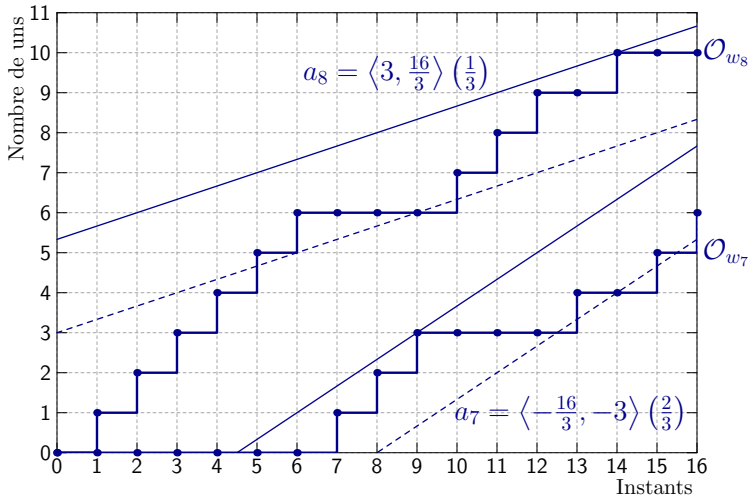


FIG. 7.20 – $a_8 = \mathit{not}^{\sim}\ a_7$.

7.4.2 Opérateur $\mathit{not}^{\sim}$

Définissons un opérateur not abstrait, écrit $\mathit{not}^{\sim}$.

Définition 7.8 (opérateur $\mathit{not}^{\sim}$). ✿

$$\mathit{not}^{\sim} \langle b^0, b^1 \rangle (r) \stackrel{def}{=} \langle -b^1, -b^0 \rangle (1 - r)$$

L'intuition derrière la formule de cet opérateur abstrait est la suivante. Soit $a = \langle b^0, b^1 \rangle \left(\frac{n}{\ell}\right)$ et soit $w \in \mathit{concr}\,(a)$. Sur ℓ éléments de w, il y aura en moyenne n occurrences de l'élément $\mathtt{1}$, donc $\ell - n$ occurrences de l'élément $\mathtt{0}$. Par conséquent, sur ℓ éléments de $\mathit{not}\ w$, il y aura en moyenne $\ell - n$ occurrences de $\mathtt{1}$. Le taux de $\mathtt{1}$ dans $\mathit{not}\ w$ est donc $\frac{\ell - n}{\ell}$, c'est-à-dire $1 - r$.

En ce qui concerne les décalages, rappelons que le nombre maximal de $\mathtt{0}$ avant le premier $\mathtt{1}$ dans w est une fonction de b^0, et que le nombre maximal de $\mathtt{1}$ avant le premier $\mathtt{0}$ dans w est une fonction de b^1. Il n'est donc pas étonnant que dans la négation de w, la borne inférieure du décalage soit définie en fonction de b^1 et la borne supérieure en fonction de b^0.

Sur la figure 7.20, la valeur abstraite a_8 est la négation de la valeur abstraite a_7.

Cet opérateur abstrait est correct, et c'est le plus précis :

Proposition 7.18 ($\mathit{not}^{\sim}$ est correct ✿ et le plus précis).
$\mathit{concr}\,(\mathit{not}^{\sim}\ a) = \{w' \mid w' = \mathit{not}\ w \text{ avec } w \in \mathit{concr}\,(a)\}$

Démonstration :

Correction: Montrons que $\{w' \mid w' = \mathit{not}\ w$ **avec** $w \in \mathit{concr}\,(a)\} \subseteq \mathit{concr}\,(\mathit{not}^{\sim}\ a)$

Soit $w \in \mathit{concr}\,(a)$. Par la remarque 3.4, on a $\mathcal{O}_{\mathit{not}\ w}(i) = i - \mathcal{O}_w(i)$.
D'une part, $\mathit{not}\ w[i] = \mathtt{1} \Rightarrow w[i] = \mathtt{0} \Rightarrow \mathcal{O}_w(i) \geq r \times i + b^0$.
Donc $\mathit{not}\ w[i] = \mathtt{1} \Rightarrow i - \mathcal{O}_{\mathit{not}\ w}(i) \geq r \times i + b^0$.
Donc $\mathcal{O}_{\mathit{not}\ w}(i) \leq i - (r \times i + b^0)$, i.e. $\mathcal{O}_{\mathit{not}\ w}(i) \leq (1 - r) \times i - b^0$.
D'autre part, $\mathit{not}\ w[i] = \mathtt{0} \Rightarrow w[i] = \mathtt{1} \Rightarrow \mathcal{O}_w(i) \leq r \times i + b^1$.
Donc $\mathit{not}\ w[i] = \mathtt{0} \Rightarrow i - \mathcal{O}_{\mathit{not}\ w}(i) \leq r \times i + b^1$.
Donc $\mathcal{O}_{\mathit{not}\ w}(i) \geq i - r \times i - b^1$, i.e. $\mathcal{O}_{\mathit{not}\ w}(i) \geq (1 - r) \times i - b^1$.
Donc $\mathit{not}\ w \in \mathit{concr}\,(\mathit{not}^{\sim}\ a)$.

Précision: Montrons que $concr\,(\texttt{not}^{\sim}\ a) \subseteq \{w' \mid w' = \texttt{not}\ w$ **avec** $w \in concr\,(a)\}$

Soit $w' \in concr\,(\texttt{not}^{\sim}\ a)$. Montrons qu'il existe un $w \in concr\,(a)$ tel que $w' = \texttt{not}\ w$, c'est-à-dire montrons que $\texttt{not}\ w' \in concr\,(a)$.

On a $\mathcal{O}_{\texttt{not}\ w'}(i) = i - \mathcal{O}_{w'}(i)$.

D'une part, $\texttt{not}\ w'[i] = \texttt{1} \Rightarrow w'[i] = \texttt{0} \Rightarrow \mathcal{O}_{w'}(i) \geq (1-r) \times i + (-b^{\texttt{1}})$.

Donc $\texttt{not}\ w'[i] = \texttt{1} \Rightarrow i - \mathcal{O}_{\texttt{not}\ w'}(i) \geq (1-r) \times i - b^{\texttt{1}}$.

Donc $\mathcal{O}_{\texttt{not}\ w'}(i) \leq r \times i + b^{\texttt{1}}$.

D'autre part, $\texttt{not}\ w'[i] = \texttt{0} \Rightarrow w'[i] = \texttt{1} \Rightarrow \mathcal{O}_{w'}(i) \leq (1-r) \times i + (-bzero)$.

Donc $\texttt{not}\ w'[i] = \texttt{0} \Rightarrow i - \mathcal{O}_{\texttt{not}\ w'}(i) \leq (1-r) \times i + (-b^{\texttt{0}})$.

Donc $\mathcal{O}_{\texttt{not}\ w'}(i) \geq r \times i + b^{\texttt{0}}$.

Donc $\texttt{not}\ w' \in concr\,(a)$, c'est-à-dire $w' = \texttt{not}\ w$ avec $w \in concr\,(a)$.

Par conséquent, $w' \in \{w' \mid w' = \texttt{not}\ w$ avec $w \in concr\,(a)\}$. □

Remarque 7.6. Si a est en forme normale, alors par définition de l'opérateur $\texttt{not}^{\sim}$, $\texttt{not}^{\sim}\ a$ est aussi en forme normale. De plus, l'opérateur $\texttt{not}^{\sim}$ préserve la condition imposée par la remarque 7.1 sur les décalages des enveloppes de taux 1 et 0. ◇

Démonstration :

Préservation de la forme normale

Soit $a = \left\langle \frac{k^{\texttt{0}}}{\ell}, \frac{k^{\texttt{1}}}{\ell} \right\rangle \left(\frac{n}{\ell}\right)$ avec $\text{pgcd}(n, \ell) = 1$. Soit $a' = \texttt{not}^{\sim}\ a$. $a' = \left\langle -\frac{k^{\texttt{1}}}{\ell}, -\frac{k^{\texttt{0}}}{\ell} \right\rangle \left(\frac{\ell - n}{\ell}\right)$.

Montrons que $\text{pgcd}(\ell - n, \ell) = 1$. Soit $g = \text{pgcd}(\ell - n, \ell)$. On a $\ell = g \times x$ avec $x \in \mathbb{N}$, et $\ell - n = g \times y$ avec $y \in \mathbb{N}$, $y < x$ (car $\ell - n < \ell$) et $\text{pgcd}(x, y) = 1$. Donc $n = g \times x - g \times y = g \times (x - y)$. Donc g divise ℓ et g divise n. Comme $\text{pgcd}(n, \ell) = 1$, on peut en conclure que $g = 1$.

Préservation des décalages acceptés pour les taux à 1 et 0

D'autre part, soit $a' = \texttt{not}^{\sim}\ a = \left\langle b^{\texttt{0}}, b^{\texttt{1}} \right\rangle (0)$. Alors $a = \left\langle -b^{\texttt{0}}, -bun \right\rangle (1)$. Comme on suppose que a respecte la condition sur les décalages des enveloppes de taux 1, c'est-à-dire que $-b^{\texttt{0}} \leq 0$ et $-b^{\texttt{1}} \leq$, on peut en déduire que a' respecte la condition sur les décalages des enveloppes de taux 0, c'est-à-dire que $b^{\texttt{0}} \geq 0$ et $b^{\texttt{1}} \geq 0$. On peut montrer de la même manière que si $a' = \texttt{not}^{\sim}\ a = \left\langle b^{\texttt{0}}, b^{\texttt{1}} \right\rangle (1)$, alors $b^{\texttt{0}} \leq 0$ et $b^{\texttt{1}} \leq 0$.

□

On peut remarquer en appliquant la définition de $\texttt{not}^{\sim}$ que $\texttt{not}^{\sim}\ \texttt{not}^{\sim}\ a = a$.

7.4.3 Opérateurs $\sqcup^{\sim}$ et $\sqcap^{\sim}$

Définissons des opérateurs $\sqcup$ et $\sqcap$ abstraits qui opèrent sur des enveloppes de même taux. Ils seront notés $\sqcap^{\sim}$ et $\sqcup^{\sim}$.

Définition 7.9 (opérateurs $\sqcup^{\sim}$ et $\sqcap^{\sim}$).

$$\left\langle b^{\texttt{0}}{}_1, b^{\texttt{1}}{}_1 \right\rangle (r) \sqcup^{\sim} \left\langle b^{\texttt{0}}{}_2, b^{\texttt{1}}{}_2 \right\rangle (r) \stackrel{def}{=} \left\langle \min(b^{\texttt{0}}{}_1, b^{\texttt{0}}{}_2), \min(b^{\texttt{1}}{}_1, b^{\texttt{1}}{}_2) \right\rangle (r)$$

$$\left\langle b^{\texttt{0}}{}_1, b^{\texttt{1}}{}_1 \right\rangle (r) \sqcap^{\sim} \left\langle b^{\texttt{0}}{}_2, b^{\texttt{1}}{}_2 \right\rangle (r) \stackrel{def}{=} \left\langle \max(b^{\texttt{0}}{}_1, b^{\texttt{0}}{}_2), \max(b^{\texttt{1}}{}_1, b^{\texttt{1}}{}_2) \right\rangle (r)$$

En effet, la fonction de cumul du supremum vaut à chaque instant le minimum des fonctions de cumul des opérandes. À partir d'un certain rang, elle se situe donc sous les droites Δ^1 des opérandes, et au dessus de la plus basse des droites Δ^0. Par exemple, pour les enveloppes de la figure 7.13, $a_9 \sqcup^\sim a_2 = \left\langle -\frac{12}{5}, \frac{9}{5} \right\rangle \left(\frac{3}{5}\right) \sqcup^\sim \left\langle -\frac{9}{5}, -\frac{3}{5} \right\rangle \left(\frac{3}{5}\right) = \left\langle -\frac{12}{5}, -\frac{3}{5} \right\rangle \left(\frac{3}{5}\right)$.

De même, la fonction de cumul de l'infimum vaut à chaque instant le maximum des fonctions de cumul des opérandes. À partir d'un certain rang, elle se situe donc au dessus des droites Δ^0 des opérandes, et sous la plus haute des droites Δ^1.

Les opérateurs abstraits $\sqcup^\sim$ et $\sqcap^\sim$ sont donc corrects : le supremum (resp. l'infimum) dans le domaine concret est dans la concrétisation du supremum (resp. de l'infimum) dans le domaine abstrait. Ce sont les opérateurs les plus précis.

Proposition 7.19 ($\sqcap^\sim$ et $\sqcup^\sim$ sont corrects et les plus précis).
Les opérateurs $\sqcap^\sim$ et $\sqcup^\sim$ sont corrects :

$$\begin{array}{rcl} \forall w_1 \in \mathit{concr}\,(a_1)\,,\ \forall w_2 \in \mathit{concr}\,(a_2)\,, & w_1 \sqcup w_2 & \in \quad \mathit{concr}\,(a_1 \sqcup^\sim a_2) \\ & w_1 \sqcap w_2 & \in \quad \mathit{concr}\,(a_1 \sqcap^\sim a_2) \end{array}$$

De plus, si a_1 et a_2 sont en forme normale, $a_1 \sqcup^\sim a_2$ et $a_1 \sqcap^\sim a_2$ sont les plus petites enveloppes au sens de $\sqsubseteq^\sim$ qui vérifient cette propriété.

Démonstration :

Soient $a_1 = \left\langle {b^0}_1, {b^1}_1 \right\rangle (r)$ et $a_2 = \left\langle {b^0}_2, {b^1}_2 \right\rangle (r)$. Soient $w_1 \in \mathit{concr}\,(a_1)$ et $w_2 \in \mathit{concr}\,(a_2)$.

Montrons que $\sqcup^\sim$ est correct, c'est-à-dire que $w_1 \sqcup w_2 \in \mathit{concr}\,(a_1 \sqcup^\sim a_2)$.

$\mathcal{O}_{(w_1 \sqcup w_2)}(i) = \min(\mathcal{O}_{w_1}(i), \mathcal{O}_{w_2}(i))$, donc :

$$\begin{array}{rcl} (w_1 \sqcup w_2)[i] = \mathtt{1} & \Rightarrow & (w_1[i] = \mathtt{1} \wedge \mathcal{O}_{w_2}(i) \geq \mathcal{O}_{w_1}(i)) \ \vee \ (w_2[i] = \mathtt{1} \wedge \mathcal{O}_{w_1}(i) \geq \mathcal{O}_{w_2}(i)) \\ & \Rightarrow & (\mathcal{O}_{w_2}(i) \geq \mathcal{O}_{w_1}(i) \geq r \times i + {b^1}_1) \ \vee \ (\mathcal{O}_{w_1}(i) \geq \mathcal{O}_{w_2}(i) \geq r \times i + {b^1}_2) \\ & \Rightarrow & \mathcal{O}_{w_1 \sqcup w_2}(i) = \min(\mathcal{O}_{w_1}(i), \mathcal{O}_{w_2}(i)) \leq r \times i + \min({b^1}_1, {b^1}_2) \\ (w_1 \sqcup w_2)[i] = \mathtt{0} & \Rightarrow & w_1[i] = \mathtt{0} \wedge w_2[i] = \mathtt{0} \\ & \Rightarrow & (\mathcal{O}_{w_1}(i) \geq r \times i + {b^0}_1) \ \wedge \ (\mathcal{O}_{w_2}(i) \geq r \times i + {b^0}_2) \\ & \Rightarrow & \mathcal{O}_{w_1 \sqcup w_2}(i) = \min(\mathcal{O}_{w_1}(i),\ \mathcal{O}_{w_2}(i)) \geq r \times i + \min({b^0}_1, {b^0}_2) \end{array}$$

Par conséquent, par définition de la fonction de concrétisation,
$w_1 \sqcup w_2 \in \mathit{concr}\,(a_1 \sqcup^\sim a_2)$.
La preuve que $w_1 \sqcap w_2 \in \mathit{concr}\,(a_1 \sqcap^\sim a_2)$ suit le même principe.

Montrons que $\sqcup^\sim$ est le plus précis c'est-à-dire que $a_1 \sqcup^\sim a_2$ est la plus petite enveloppe correcte au sens de $\sqsubseteq^\sim$ (proposition 7.7).

Cas où ${b^1}_1 \geq {b^1}_2$:
Dans ce cas, $\mathit{early}_{a_1} \preceq \mathit{early}_{a_2}$. Donc $\mathit{early}_{a_1} \sqcup \mathit{early}_{a_2} = \mathit{early}_{a_2}$.
Par conséquent, $\forall {b^1}' \leq {b^1}_2,\ \mathit{early}_{a_1} \sqcup \mathit{early}_{a_2} \notin \left\langle \min({b^0}_1, {b^0}_2), {b^1}' \right\rangle (r)$.

Cas où ${b^1}_1 \leq {b^1}_2$:
Ici, $\mathit{early}_{a_1} \sqcup \mathit{early}_{a_2} = \mathit{early}_{a_1}$.
Par conséquent, $\forall {b^1}' \leq {b^1}_1,\ \mathit{early}_{a_1} \sqcup \mathit{early}_{a_2} \notin \left\langle \min({b^0}_1, {b^0}_2), {b^1}' \right\rangle (r)$.

Donc $\min({b^1}_1, {b^1}_2)$ est le plus petit décalage correct pour Δ^1 de $a_1 \sqcup a_2$. On peut prouver de la même manière que $\min({b^0}_1, {b^0}_2)$ est le plus grand décalage correct pour Δ^0 :

Cas où ${b^0}_1 \geq {b^0}_2$:
Dans ce cas, $late_{a_1} \preceq late_{a_2}$. Donc $late_{a_1} \sqcap late_{a_2} = late_{a_2}$.
Par conséquent, $\forall {b^0}' \geq {b^0}_2,\ late_{a_1} \sqcup late_{a_2} \notin \left\langle {b^0}', \min({b^1}_1, {b^1}_2) \right\rangle (r)$.

Cas où ${b^0}_1 \leq {b^0}_2$:
Ici, $late_{a_1} \sqcap late_{a_2} = late_{a_1}$.
Par conséquent, $\forall {b^0}' \leq {b^0}_1,\ late_{a_1} \sqcap late_{a_2} \notin \left\langle {b^0}', \min({b^1}_1, {b^1}_2) \right\rangle (r)$.

La preuve que $\sqcap^\sim$ est le plus précis est similaire.

□

7.4.4 Abstraction d'une expression d'horloges par une enveloppe

Étant donnée une abstraction des mots ($abs\,(w)$), on peut calculer l'abstraction d'expressions composées par ces mots. Elle est définie ainsi :

Définition 7.10 (fonction d'abstraction des expressions d'horloges).

$$\begin{array}{lcl} abs\,(\texttt{not}\ ce) & \stackrel{def}{=} & \texttt{not}^\sim\ abs\,(ce) \\ abs\,(ce_1\ \texttt{on}\ ce_2) & \stackrel{def}{=} & abs\,(ce_1)\ \texttt{on}^\sim\ abs\,(ce_2) \\ abs\,(ce_1 \sqcup ce_2) & \stackrel{def}{=} & abs\,(ce_1) \sqcup^\sim abs\,(ce_2) \\ abs\,(ce_1 \sqcap ce_2) & \stackrel{def}{=} & abs\,(ce_1) \sqcap^\sim abs\,(ce_2) \end{array}$$

Cette abstraction des horloges est correcte :

Proposition 7.20 (correction). $ce \in concr\,(abs\,(ce))$

Démonstration :

Par induction structurelle sur ce :

Si $ce = w$ alors par hypothèse $abs\,(w)$ est une abstraction correcte de w, i.e. $ce \in concr\,(abs\,(ce))$.

Si $ce = \texttt{not}\ ce'$ avec $ce' \in concr\,(abs\,(ce'))$ alors par la proposition 7.18 $\texttt{not}\ ce' \in concr\,(\texttt{not}^\sim\ abs\,(ce'))$, i.e. $ce \in concr\,(abs\,(ce))$.

Si $ce = ce_1\ \texttt{on}\ ce_2$ avec $ce_1 \in concr\,(abs\,(ce_1))$ et $ce_2 \in concr\,(abs\,(ce_2))$, alors par la proposition 7.16 $ce_1\ \texttt{on}\ ce_2 \in concr\,(abs\,(ce_1)\ \texttt{on}^\sim\ abs\,(ce_2))$, i.e. $ce \in concr\,(abs\,(ce))$.

Si $ce = ce_1 \sqcup ce_2$ avec $ce_1 \in concr\,(abs\,(ce_1))$ et $ce_2 \in concr\,(abs\,(ce_2))$, alors par la proposition 7.19 $ce_1 \sqcup ce_2 \in concr\,(abs\,(ce_1) \sqcup^\sim abs\,(ce_2))$, i.e. $ce \in concr\,(abs\,(ce))$.

Si $ce = ce_1 \sqcap ce_2$ avec $ce_1 \in concr\,(abs\,(ce_1))$ et $ce_2 \in concr\,(abs\,(ce_2))$, alors par la proposition 7.19 $ce_1 \sqcap ce_2 \in concr\,(abs\,(ce_1) \sqcap^\sim abs\,(ce_2))$, i.e. $ce \in concr\,(abs\,(ce))$.

□

Par contre, comme l'opérateur $\texttt{on}$ n'est pas le plus précis, cette fonction d'abstraction n'est pas la plus précise. L'abstraction des ensembles d'expressions d'horloges associée à la concrétisation n'est donc pas une correspondance de Galois.

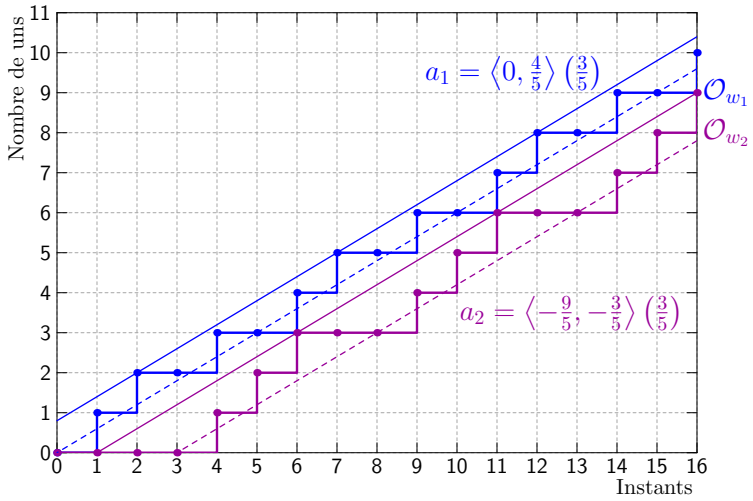


FIG. 7.21 – L'enveloppe a_1 est synchronisable avec l'enveloppe a_2 et elle la précède.

7.5 Relations abstraites

Nous avons décrit dans les deux sections précédentes comment trouver l'enveloppe d'une expression d'horloges. Définissons maintenant les relations sur les enveloppes, correspondant aux relations sur les mots définies en section 3.4. Une relation est vérifiée sur des enveloppes si elle est vérifiée sur tout couple de mots appartenant aux concrétisations respectives de celles-ci. La relation de synchronisabilité abstraite, notée $\bowtie^\sim$, est définie de la manière suivante.

Définition 7.11 (relation $\bowtie^\sim$). ✿

$$a_1 \bowtie^\sim a_2 \overset{def}{\Leftrightarrow} \forall w_1 \in \mathit{concr}\,(a_1)\,, w_2 \in \mathit{concr}\,(a_2)\,,\ w_1 \bowtie w_2$$

On peut vérifier simplement la relation de synchronisabilité sur les enveloppes :

Proposition 7.21 (test de synchronisabilité). ✿ ✿

$$\langle {b^0}_1, {b^1}_1 \rangle\,(r_1) \bowtie^\sim \langle {b^0}_2, {b^1}_2 \rangle\,(r_2) \Leftrightarrow r_1 = r_2$$

Par exemple, sur la figure 7.21 l'enveloppe a_1 est synchronisable avec l'enveloppe a_2. Géométriquement parlant, les chronogrammes des mots représentés par a_1 restent à une distance bornée des droites Δ_1, et les horloges représentées par a_2 restent à une distance bornée des droites Δ_2. Par conséquent, les horloges de a_1 restent à une distance bornée de celles de a_2 si et seulement si les droites Δ_1 restent à une distance bornée des droites Δ_2, c'est-à-dire si et seulement si la pente r_1 des droites Δ_1 est égale à la pente r_2 des droites Δ_2.

Démonstration :

Soient $w_1 \in \mathit{concr}\,(a_1)$ et $w_2 \in \mathit{concr}\,(a_2)$.
Par la proposition 7.1, $\mathit{early}_{a_1} \preceq w_1 \preceq \mathit{late}_{a_1}$ et $\mathit{early}_{a_2} \preceq w_2 \preceq \mathit{late}_{a_2}$.
Par conséquent, par la proposition 3.7 (page 64) :
$\forall i,\ \mathcal{O}_{\mathit{early}_{a_1}}(i) \geq \mathcal{O}_{w_1}(i) \geq \mathcal{O}_{\mathit{late}_{a_1}}(i)$ et $\forall i,\ \mathcal{O}_{\mathit{early}_{a_2}}(i) \geq \mathcal{O}_{w_2}(i) \geq \mathcal{O}_{\mathit{late}_{a_2}}(i)$.
Donc $\mathcal{O}_{\mathit{early}_{a_1}}(i) - \mathcal{O}_{\mathit{late}_{a_2}}(i) \geq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \geq \mathcal{O}_{\mathit{late}_{a_1}}(i) - \mathcal{O}_{\mathit{early}_{a_2}}(i)$.

Par la proposition 7.4, à partir d'un certain indice, on a
$\mathcal{O}_{early_{a_1}}(i) = \lfloor r_1 \times i + b^1{}_1 \rfloor$, $\mathcal{O}_{late_{a_1}}(i) = \lceil r_1 \times i + b^0{}_1 \rceil$,
$\mathcal{O}_{early_{a_2}}(i) = \lfloor r_2 \times i + b^1{}_2 \rfloor$ et $\mathcal{O}_{late_{a_2}}(i) = \lceil r_2 \times i + b^0{}_2 \rceil$.

Avant cet indice, le nombre de résultats obtenus pour $\mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i)$ est fini, et les résultats sont donc bornés par leur minimum et leur maximum.

À partir de cet indice, on a :
$\lfloor r_1 \times i + b^1{}_1 \rfloor - \lceil r_2 \times i + b^0{}_2 \rceil \geq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \geq \lceil r_1 \times i + b^0{}_1 \rceil - \lfloor r_2 \times i + b^1{}_2 \rfloor$,
c'est-à-dire $(r_1 \times i + b^1{}_1) - (r_2 \times i + b^0{}_2) \geq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \geq (r_1 \times i + b^0{}_1) - (r_2 \times i + b^1{}_2)$
(car $x - y \geq \lfloor x \rfloor - \lceil y \rceil$ et $\lceil x \rceil - \lfloor y \rfloor \geq x - y$, voir lemme D.3).
On obtient donc $(r_1 - r_2) \times i + b^1{}_1 - b^0{}_2 \geq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \geq (r_1 - r_2) \times i + b^0{}_1 - b^1{}_2$.

Analysons ces bornes en fonction de la comparaison de r_1 et r_2 :

Si $r_1 = r_2$ **alors** $b^1{}_1 - b^0{}_2 \geq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \geq b^0{}_1 - b^1{}_2$.
Par définition de $\bowtie$ (définition 3.9 page 69), $w_1 \bowtie w_2$.
Par définition de $\bowtie^\sim$, $a_1 \bowtie^\sim a_2$.

Si $r_1 > r_2$ **alors** $lim_{i \to +\infty}((r_1 - r_2) \times i + b^0{}_1 - b^1{}_2) = +\infty$.
Donc $\nexists b,\ \forall i \geq 0, b \geq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i)$. Par conséquent, par définition de $\bowtie$, $w_1 \not\bowtie w_2$.
Comme $w_1 \in concr\,(a_1)$ et $w_2 \in concr\,(a_2)$, par définition de $\bowtie^\sim$ on peut en déduire que $a_1 \not\bowtie^\sim a_2$.

Si $r_1 < r_2$ **alors** de manière similaire, $lim_{i \to +\infty}((r_1 - r_2) \times i + b^1{}_1 - b^0{}_2) = -\infty$.
Donc $\nexists b,\ \forall i \geq 0, \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \geq b$. Par conséquent, $w_1 \not\bowtie w_2$ et $a_1 \not\bowtie^\sim a_2$.

□

Remarque 7.7. Un corollaire de cette proposition est que $\bowtie^\sim$ est réflexive, donc toute valeur abstraite est synchronisable avec elle même. Cela signifie qu'au sein d'un ensemble de concrétisation tout couple de mots est synchronisable. La relation est aussi symétrique et transitive, c'est donc une relation d'équivalence. ◇

Les valeurs abstraites contiennent toutes les informations nécessaires pour vérifier de manière complète la synchronisabilité de deux expressions d'horloges sur leur enveloppe :

Proposition 7.22 (correction et complétude). $abs\,(ce_1) \bowtie^\sim abs\,(ce_2) \Leftrightarrow ce_1 \bowtie ce_2$

Démonstration :

Par la proposition 7.20, on a $ce_1 \in concr\,(abs\,(ce_1))$ et $ce_2 \in concr\,(abs\,(ce_2))$.
$\Rightarrow$) Si $abs\,(ce_1) \bowtie^\sim abs\,(ce_2)$, alors par définition de $\bowtie^\sim$, $ce_1 \bowtie ce_2$.
$\Leftarrow$) Supposons que $ce_1 \bowtie ce_2$.
Par la remarque 7.7, on a de plus :
$\forall w \in concr\,(abs\,(ce_1)), w \bowtie ce_1$ et $\forall w \in concr\,(abs\,(w_2)), w \bowtie ce_2$.
La transitivité de $\bowtie$ implique que $\forall w \in concr\,(abs\,(ce_1)), \forall w' \in concr\,(abs\,(ce_2)), w \bowtie w'$.
Enfin, par définition de $\bowtie^\sim$, $abs\,(ce_1) \bowtie^\sim abs\,(ce_2)$. □

Définissons maintenant la relation de précédence sur les enveloppes.

Définition 7.12 (relation $\preceq^{\sim}$). ✿

$$a_1 \preceq^{\sim} a_2 \overset{def}{\Leftrightarrow} \forall w_1 \in concr\,(a_1)\,, w_2 \in concr\,(a_2)\,,\ w_1 \preceq w_2$$

La relation de précédence est vérifiée sur deux enveloppes si et seulement si le supremum (au sens de $\preceq$) de la concrétisation de la première enveloppe précède l'infimum de la concrétisation de la seconde :

Proposition 7.23. ✿ ✿
Soient $a_1 = \langle {b^0}_1, {b^1}_1 \rangle\,(r_1)$ et $a_2 = \langle {b^0}_2, {b^1}_2 \rangle\,(r_2)$ telles que $concr\,(a_1) \neq \varnothing$ et $concr\,(a_2) \neq \varnothing$.

$$\begin{aligned} a_1 \preceq^{\sim} a_2 &\Leftrightarrow \sqcup\,(concr\,(a_1)) \preceq \sqcap\,(concr\,(a_2)) \\ &\Leftrightarrow late_{a_1} \preceq early_{a_2} \end{aligned}$$

En effet, quand le plus grand mot de la première enveloppe précède le plus petit mot de la seconde, tous les mots de la première enveloppe précèdent tous ceux de la seconde.
Démonstration :

> Par définition de $\preceq^{\sim}$, $a_1 \preceq^{\sim} a_2 \Leftrightarrow \forall w_1 \in concr\,(a_1)\,, \forall w_2 \in concr\,(a_2)\,, w_1 \preceq w_2$. Par définition de $\sqcap$ et $\sqcup$ et par transitivité de $\preceq$, ceci équivaut à $\sqcup concr\,(a_1) \preceq \sqcap\,(concr\,(a_2))$. Comme $concr\,(a_1) \neq \varnothing$, $\sqcup concr\,(a_1) = late_{a_1}$ et comme $concr\,(a_2) \neq \varnothing$, $\sqcap concr\,(a_2) = early_{a_2}$. □

Quand deux enveloppes sont synchronisables, la propriété 7.23 peut être vérifiée par une condition suffisante. De plus, si les enveloppes sont en forme normale, cette condition est aussi nécessaire.

Proposition 7.24 (test de précédence). ✿ ✿ Soient $a_1 = \langle {b^0}_1, {b^1}_1 \rangle\,(r)$ et $a_2 = \langle {b^0}_2, {b^1}_2 \rangle\,(r)$.

$${b^1}_2 - {b^0}_1 < 1 \Rightarrow \langle {b^0}_1, {b^1}_1 \rangle\,(r) \preceq^{\sim} \langle {b^0}_2, {b^1}_2 \rangle\,(r)$$

De plus, la réciproque est vraie si a_1 et a_2 sont en forme normale.

Un chevauchement de moins de 1 entre les enveloppes a_1 et a_2 assure que $late_{a_1} \preceq early_{a_2}$. Sur la figure 7.21, les enveloppes a_1 et a_2 ne se chevauchent pas, donc a_1 précède a_2.
Démonstration :

> ⇒) Soient $a_1 = \langle {b^0}_1, {b^1}_1 \rangle\,(r)$ et $a_2 = \langle {b^0}_2, {b^1}_2 \rangle\,(r)$. Exprimons a_1 et a_2 sous la forme $a_1 = \left\langle \frac{k^0_1}{\ell}, \frac{k^1_1}{\ell} \right\rangle \left(\frac{n}{\ell}\right)$ et $a_2 = \left\langle \frac{k^0_2}{\ell}, \frac{k^1_2}{\ell} \right\rangle \left(\frac{n}{\ell}\right)$. Cette opération est toujours possible, il suffit de choisir pour ℓ le ppcm des dénominateurs de r, ${b^0}_1$, ${b^1}_1$, ${b^0}_2$ et ${b^1}_2$.
> $${b^1}_2 - {b^0}_1 < 1 \Rightarrow r \times i + {b^1}_2 < 1 + r \times i + {b^0}_1 \Rightarrow \frac{n\times i + k^1_2}{\ell} < \frac{\ell + n\times i + k^0_1}{\ell}$$
> $$\Rightarrow \frac{n\times i + k^1_2}{\ell} \leq \frac{\ell - 1 + n\times i + k^0_1}{\ell} \Rightarrow \left\lfloor \frac{n\times i + k^1_2}{\ell} \right\rfloor \leq \left\lfloor \frac{\ell - 1 + n\times i + k^0_1}{\ell} \right\rfloor .$$
> D'après le lemme D.1, cela implique que $\left\lfloor \frac{n\times i + k^1_2}{\ell} \right\rfloor \leq \left\lceil \frac{n\times i + k^0_1}{\ell} \right\rceil$. Par la proposition 7.4, à partir d'un certain rang i', $\mathcal{O}_{early_{a_2}}(i) = \left\lfloor \frac{n\times i + k^1_2}{\ell} \right\rfloor$ et $\mathcal{O}_{late_{a_1}}(i) = \left\lceil \frac{n\times i + k^0_1}{\ell} \right\rceil$. Donc $\forall i \geq i'$, $\mathcal{O}_{early_{a_2}}(i) \leq \mathcal{O}_{late_{a_1}}(i)$. Par conséquent, à partir du rang i', $late_{a_1} \preceq early_{a_2}$ et donc par la proposition 7.23, à partir du rang i' on a $a_1 \preceq^{\sim} a_2$. La preuve complète (traitant les comportements initiaux) a été faite en Coq.

⇐) Soient $a_1 = \left\langle \frac{k_1^0}{\ell}, \frac{k_1^1}{\ell} \right\rangle \left(\frac{n}{\ell}\right)$ et $a_2 = \left\langle \frac{k_2^0}{\ell}, \frac{k_2^1}{\ell} \right\rangle \left(\frac{n}{\ell}\right)$ deux enveloppes en forme normale, c'est-à-dire telles que $\text{pgcd}(n, \ell) = 1$. Montrons que $a_1 \preceq^\sim a_2 \Rightarrow {b^1}_2 - {b^0}_1 < 1$. Par la proposition 7.23, $a_1 \preceq^\sim a_2 \Rightarrow late_{a_1} \preceq early_{a_2}$. Donc $\forall i, \mathcal{O}_{late_{a_1}}(i) \geq \mathcal{O}_{early_{a_2}}(i)$. Par la proposition 7.4, il existe un rang i' à partir duquel $\mathcal{O}_{late_{a_1}}(i) = \left\lceil \frac{n \times i + k_1^0}{\ell} \right\rceil$ et $\mathcal{O}_{early_{a_2}}(i) = \left\lfloor \frac{n \times i + k_2^1}{\ell} \right\rfloor$. Donc $\forall i \geq i', \left\lceil \frac{n \times i + k_1^0}{\ell} \right\rceil \geq \left\lfloor \frac{n \times i + k_2^1}{\ell} \right\rfloor$.
Si $\ell = 1$, $\left\lceil \frac{n \times i + k_1^0}{\ell} \right\rceil \geq \left\lfloor \frac{n \times i + k_2^1}{\ell} \right\rfloor \Rightarrow k_2^1 - k_1^0 \leq 0$ et la propriété est vérifiée.
Si $\ell > 1$, comme $\text{pgcd}(\ell, n) = 1$, par le théorème de Bézout, on a :
$\exists x, y \in \mathbb{Z},\ n \times y + k_1^0 = \ell \times x + 1$. Posons $i = y + z \times \ell$ et $x' = x + z \times n$, avec $z \in \mathbb{N}$ tel que $i \geq i'$ et $x' \geq 0$. Pour ce i, on a $n \times i + k_1^0 = \ell \times x' + 1$.
Comme $\forall i \geq i', \left\lceil \frac{n \times i + k_1^0}{\ell} \right\rceil \geq \left\lfloor \frac{n \times i + k_2^1}{\ell} \right\rfloor \Leftrightarrow \forall i \geq i', \left\lceil \frac{n \times i + k_1^0}{\ell} \right\rceil \geq \left\lfloor \frac{n \times i + k_1^0 + (k_2^1 - k_1^0)}{\ell} \right\rfloor$, pour ce i on a $\left\lceil \frac{\ell \times x' + 1}{\ell} \right\rceil \geq \left\lfloor \frac{\ell \times x' + 1 + (k_2^1 - k_1^0)}{\ell} \right\rfloor$, c'est-à-dire $x' + 1 \geq x' + \left\lfloor \frac{1 + (k_2^1 - k_1^0)}{\ell} \right\rfloor$, c'est-à-dire $1 \geq \left\lfloor \frac{1 + (k_2^1 - k_1^0)}{\ell} \right\rfloor$, c'est-à-dire $1 \geq \frac{1 + (k_2^1 - k_1^0)}{\ell}$, et par conséquent $1 > {b^1}_2 - {b^0}_1$. □

On a vu dans la proposition 7.23 que pour vérifier la relation de précédence entre deux expressions d'horloges sur leur abstraction, le manque d'informations sur la position des 1 nous oblige à considérer le pire cas de concrétisation. Cette vérification sur les enveloppes est donc correcte mais pas complète par rapport à la vérification sur les horloges concrètes :

Proposition 7.25 (correction). $abs\,(ce_1) \preceq^\sim abs\,(ce_2) \Rightarrow ce_1 \preceq ce_2$

Démonstration :

Par définition de $\preceq^\sim$ et parce que l'abstraction des expressions d'horloges est correcte (proposition 7.20). □

On peut maintenant définir la relation d'adaptabilité sur les enveloppes :

Définition 7.13 (relation $<:^\sim$). ✿

$$a_1 <:^\sim a_2 \overset{def}{\Leftrightarrow} \forall w_1 \in concr\,(a_1), w_2 \in concr\,(a_2),\ w_1 <: w_2$$

Tout comme dans le domaine concret, cette relation est la conjonction des relations de synchronisabilité et de précédence :

Proposition 7.26 (test d'adaptabilité). ✿ ✿ $a_1 <:^\sim a_2 \Leftrightarrow a_1 \bowtie^\sim a_2 \wedge a_1 \preceq^\sim a_2$

Démonstration :

Par définition de $<:^\sim$, $<:$, $\bowtie^\sim$ et $\preceq^\sim$. □

Comme les relations de synchronisabilité et de précédence abstraites sont correctes, la relation d'adaptabilité l'est aussi :

Proposition 7.27 (correction). $abs\,(ce_1) <:^\sim abs\,(ce_2) \Rightarrow ce_1 <: ce_2$

Démonstration :

Par définition de $<:^\sim$ et par la proposition 7.20. □

Il est donc possible de tester efficacement la relation d'adaptabilité dans le domaine abstrait, en utilisant les tests de synchronisabilité et de précédence abstraits (propositions 7.21 et 7.24).

Calcul de la taille du buffer

Lorsque la relation d'adaptabilité entre deux horloges est vérifiée, on peut consommer au rythme de la seconde horloge un flot produit sur le rythme de la première horloge en introduisant un buffer de taille bornée. Nous avons vu dans cette section que la relation d'adaptabilité peut être vérifiée dans le domaine abstrait. Une taille correcte pour le buffer établissant la communication peut elle aussi être calculée dans le domaine abstrait.

Définition 7.14 (taille du buffer). ✿

Soient $a_1 = \langle {b^0}_1, {b^1}_1 \rangle (r_1)$ et $a_2 = \langle {b^0}_2, {b^1}_2 \rangle (r_2)$ deux enveloppes non vides telles que $a_1 <:^{\sim} a_2$.

$$size^{\sim}(a_1, a_2) = \lfloor {b^1}_1 - {b^0}_2 \rfloor$$

En effet, la taille de buffer nécessaire pour communiquer de toute horloge de a_1 vers toute horloge de a_2 est la taille de buffer nécessaire pour communiquer de l'horloge la plus précoce de a_1 vers l'horloge la plus tardive de a_2 (qui correspond approximativement à la distance entre la droite Δ^1 de a_1 et la droite Δ^0 de a_2). La formule vient donc du calcul de cette taille sur les horloges concrètes $early_{a_1}$ et $late_{a_2}$.

Proposition 7.28 (correction ✿ et précision de $size^{\sim}$).

$size^{\sim}$ calcule une taille de buffer suffisante et qui surévalue de 1 au maximum la taille nécessaire $s = \max_{w_1 \in concr(a_1), w_2 \in concr(a_2)}(size(w_1, w_2))$:

$$0 \leq size^{\sim}(a_1, a_2) - s \leq 1$$

Démonstration :

Soit s la taille de buffer nécessaire et suffisante.

$$\begin{aligned} s &= \max_{w_1 \in concr(a_1), w_2 \in concr(a_2)}(size(w_1, w_2)) \\ &= \max_{w_1 \in concr(a_1), w_2 \in concr(a_2), i \in \mathbb{N}} (\mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i)) \\ &= \max_{i \in \mathbb{N}}(\max_{w_1 \in concr(a_1)} \mathcal{O}_{w_1}(i) - \min_{w_2 \in concr(a_2)} \mathcal{O}_{w_2}(i)) \\ &= \max_{i \in \mathbb{N}}(\mathcal{O}_{early_{a_1}}(i) - \mathcal{O}_{late_{a_2}}(i)) \\ &= size(early_{a_1}, late_{a_2}) \end{aligned}$$

Par la proposition 7.4, $\mathcal{O}_{early_{a_1}}(i) = \max(0, \min(i, \lfloor r \times i + {b^1}_1 \rfloor))$ et
$\mathcal{O}_{late_{a_1}}(i) = \max(0, \min(i, \lceil r \times i + {b^0}_2 \rceil))$.
Donc $s = \max_{i \in \mathbb{N}^*}(\max(0, \min(i, \lfloor r \times i + {b^1}_1 \rfloor)) - \max(0, \min(i, \lceil r \times i + {b^0}_2 \rceil)))$.
La différence maximale est atteinte quand $\mathcal{O}_{early_{a_1}}(i) = \lfloor r \times i + {b^1}_1 \rfloor$ (c'est toujours vrai à partir d'un rang) et $\mathcal{O}_{late_{a_2}}(i) = \lceil r \times i + {b^0}_2 \rceil$ (toujours vrai à partir d'un certain rang).
Donc $s = \max_{i \in \mathbb{N}^*}(\mathcal{O}_{early_{a_1}}(i) - \mathcal{O}_{late_{a_2}}(i)) = \max_{i \in \mathbb{N}^*}(\lfloor r \times i + {b^1}_1 \rfloor - \lceil r \times i + {b^0}_2 \rceil)$.
Par le lemme D.4, on a $\lfloor x - y \rfloor - 1 \leq \lfloor x \rfloor - \lceil y \rceil \leq \lfloor x - y \rfloor$.
Comme $\lfloor r \times i + {b^1}_1 - (r \times i + {b^0}_2) \rfloor = \lfloor {b^1}_1 - {b^0}_2 \rfloor$,
on peut en déduire que $\lfloor {b^1}_1 - {b^0}_2 \rfloor - 1 \leq \lfloor r \times i + {b^1}_1 \rfloor - \lceil r \times i + {b^0}_2 \rceil \leq \lfloor {b^1}_1 - {b^0}_2 \rfloor$.
Par conséquent, $\max_{i \in \mathbb{N}^*}(\lfloor {b^1}_1 - {b^0}_2 \rfloor - 1) \leq s \leq \max_{i \in \mathbb{N}^*}(\lfloor {b^1}_1 - {b^0}_2 \rfloor)$,
c'est-à-dire $\lfloor {b^1}_1 - {b^0}_2 \rfloor - 1 \leq s \leq \lfloor {b^1}_1 - {b^0}_2 \rfloor$.

Comme $size^{\sim}(a_1, a_2) = \lfloor {b^1}_1 - {b^0}_2 \rfloor$, on peut en conclure que
$size^{\sim}(a_1, a_2) - 1 \leq s \leq size^{\sim}(a_1, a_2)$. L'opérateur $size^{\sim}(a_1, a_2)$ calcule donc une taille de buffer correcte, et surévaluée de 1 au maximum. □

7.6 Discussion

7.6.1 Comparaison avec l'abstraction proposée en première approche

Nous avions proposé en première approche une abstraction basée sur des décalages des indices d'occurrence des 1 (autrement dit sur des distances horizontales dans les graphiques). On peut toujours traduire sans perte de précision une enveloppe de cette abstraction vers une enveloppe de l'abstraction basée sur les fonctions de cumul présentée dans le reste du chapitre :

$$[d, D]\,(T) \rightsquigarrow \left\langle -\frac{D}{T}, -\frac{d}{T} \right\rangle \left(\frac{1}{T}\right)$$

Par contre, la traduction d'une enveloppe de l'abstraction présentée dans le reste du chapitre vers une enveloppe de la première approche perd de l'information si le décalage vertical minimum est positif ($b^0 > 0$) et ne peut pas être réalisée si le taux est nul :

$$\left\langle b^0, b^1 \right\rangle (r) \rightsquigarrow \left[-\frac{b^1}{r},\ \max(-\frac{b^0}{r}, 0) \right] \left(\frac{1}{r}\right) \quad \text{avec } r \neq 0$$

Ce gain de précision et d'expressivité est lié à la différence technique suivante. L'approche retenue contraint les instants d'occurrence des 1 et les instants d'occurrence des 0 de manière symétrique alors que dans la première approche, seules les occurrences des 1 sont contraintes. Les contraintes sur les occurrences des 0 n'en sont qu'une conséquence.

La deuxième différence technique entre les deux approches est que l'approche retenue manipule la fonction de cumul plutôt que la fonction donnant les indices des 1. La fonction de cumul a l'avantage de fournir autant d'information sur les occurrences des 1 que sur les occurrences des 0 (si l'on connaît la fonction de cumul des 1, on peut en déduire la fonction de cumul des 0). Le lien entre la fonction d'indice des 0 et la fonction d'indice des 1 n'est pas aussi clair. Pour contraindre à la fois les 0 et les 1 en manipulant les fonctions d'indices, on devrait donc manipuler en parallèle la fonction donnant l'indice des 1 et la fonction donnant l'indice des 0.

Notons qu'on peut tout à fait intégrer ces deux améliorations techniques à l'abstraction proposée en première approche. La définition de l'ensemble de concrétisation ressemblerait à :

$$concr\,([d, D]\,(T)) = \left\{ w \mid \forall i \geq 1, \quad \wedge \begin{array}{lcl} w[i] = \mathtt{1} & \Rightarrow & i \geq T \times (\mathcal{O}_w(i)) + d \\ w[i] = \mathtt{0} & \Rightarrow & i \leq T \times (\mathcal{O}_w(i) + 1) + D \end{array} \right\}$$

On obtient une abstraction équivalente à l'abstraction retenue, mais exprimée en terme de décalages temporels plutôt que de décalages sur les fonctions de cumul. Les fonctions de cumul étant au cœur des calculs, nous avons choisi de retenir la seconde.

7.6.2 Domaine de définition du $on^{\sim}$

Nous avons vu que l'opérateur $on^{\sim}$ n'est pas défini sur les enveloppes ayant un décalage vertical b^0 positif. Pour appliquer l'opérateur $on^{\sim}$ sur ces enveloppes ne contenant que des mots qui commencent par une série de 1, il faut donc commencer par les élargir. Cela signifie qu'il faut perdre l'information que les mots représentés commencent nécessairement par un ensemble de 1. Par exemple, avant d'appliquer l'opérateur $on^{\sim}$ à $a_8 = \left\langle 3, \frac{16}{3} \right\rangle \left(\frac{1}{3}\right)$, on élargit pour obtenir $\left\langle 0, \frac{16}{3} \right\rangle \left(\frac{1}{3}\right)$, comme montré sur la figure 7.22.

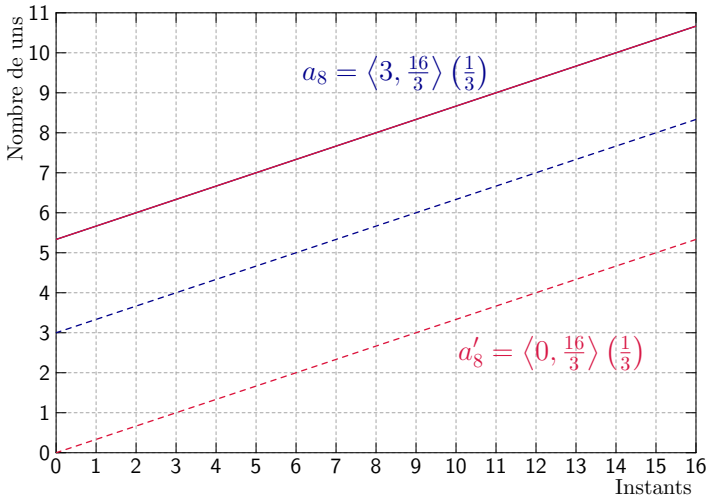


FIG. 7.22 – Élargissement de l'enveloppe a_8 avant application de l'opérateur *on*

Illustrons sur un exemple que la formule de $on^\sim$ est incorrecte en dehors de son domaine de définition. Considérons $a = \langle 6, \frac{20}{3} \rangle \left(\frac{1}{3}\right)$ et $a' = \langle \frac{3}{2}, 2 \rangle \left(\frac{1}{2}\right)$. Le résultat le plus précis pour a $on^\sim$ a' est l'abstraction de l'ensemble des mots égaux à w *on* w' avec w un mot de a et w' un mot de a'. Comme a contient l'unique mot $w =$ `111111111(100)` et a' contient l'unique mot $w' =$ `111(10)`, l'enveloppe la plus précise pour a $on^\sim$ a' est l'enveloppe la plus précise de :

`111111111(100)` *on* `111(10)` = `111101010(100000)`

c'est-à-dire $\langle \frac{19}{6}, \frac{16}{3} \rangle \left(\frac{1}{6}\right)$.

Si l'on applique la formule de $on^\sim$ sur a et a' sans les ramener au domaine de définition de l'opérateur, on obtient :

$$\left\langle 6 \times \frac{1}{2} + \frac{3}{2}, \frac{20}{3} \times \frac{1}{2} + 2 \right\rangle \left(\frac{1}{3} \times \frac{1}{2}\right) = \left\langle \frac{9}{2}, \frac{16}{3} \right\rangle \left(\frac{1}{6}\right)$$

Cette enveloppe, représentée sur la figure 7.23 ne contient pas w *on* w'. Par exemple à l'instant 5, elle interdit à tord l'occurrence d'un `0` dans le mot. Elle est donc incorrecte. En appliquant correctement l'opérateur, c'est-à-dire en élargissant les opérandes au préalable, on obtient un résultat correct :

$$\left\langle 0 \times \frac{1}{2} + \frac{3}{2}, 0 \times \frac{1}{2} + 2 \right\rangle \left(\frac{1}{3} \times \frac{1}{2}\right) = \left\langle 0, \frac{16}{3} \right\rangle \left(\frac{1}{6}\right)$$

Ce résultat, représenté sur la figure 7.23, est correct car il contient le résultat correct le plus précis : $\langle \frac{19}{6}, \frac{16}{3} \rangle \left(\frac{1}{6}\right) \sqsubseteq^\sim \langle 0, \frac{16}{3} \rangle \left(\frac{1}{6}\right)$. On peut cependant observer qu'il ne s'agit pas du résultat le plus précis, car il ne force pas le préfixe à commencer par quatre `1`.

Le traitement des b^0 positifs, c'est-à-dire des enveloppes ne contenant que des mots qui commencent par une série de `1`, est difficile. Les séries de `0` sont reportées dans le résultat du *on*, intactes pour celles de la première opérande, et allongées selon le rythme de la première opérande pour celles de la seconde. À l'inverse, qu'elle soit présente en début du premier mot

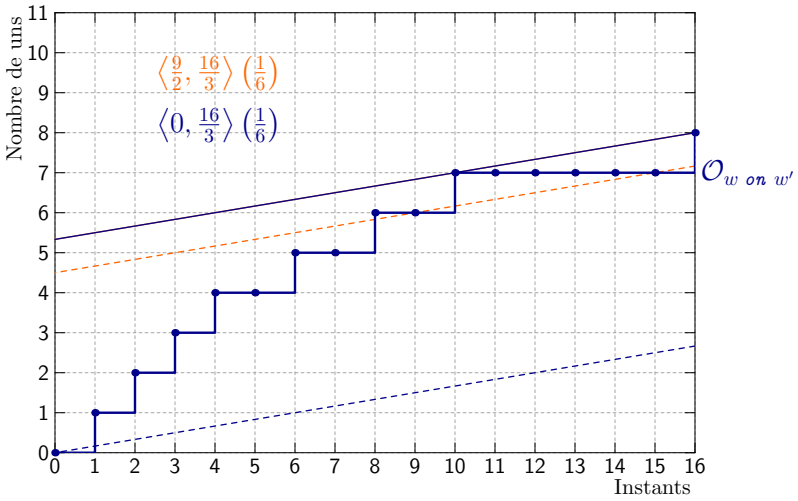


FIG. 7.23 – La formule de $on^{\sim}$ est erronée en dehors de son domaine de définition

ou du second, une série de 1 n'est pas forcément reportée dans le résultat du on.
Par exemple, $1^9(100) \; on \; (10) = 101010101(000100)$:

	1	1	1	1	1	1	1	1	1	(	1	0	0.	1	0	0	)
on	1	0.	1	0.	1	0.	1	0.	1	(	0.			1			)
=	1	0	1	0	1	0	1	0	1	(	0	0	0	1	0	0	)

et $(100) \; on \; 1^3(10) = 100100100(10^5)$:

	1	0	0.	1	0	0.	1	0	0.	(	1	0	0.	1	0	0	)
on	1			1			1.			(	1			0			)
=	1	0	0	1	0	0	1	0	0	(	1	0	0	0	0	0	)

Il existe cependant des cas où l'on sait caractériser le résultat d'une opération on sur des mots commençant par une série de 1.
Par exemple, $1^31^6(100) \; on \; 1^3(10) = 1^3101010(100000)$:

	1	1	1	1	1	1	1	1	1	(	1	0	0.	1	0	0	)
on	1	1	1.	1	0.	1	0.	1	0.	(	1			0			)
=	1	1	1.	1	0.	1	0.	1	0.	(	1	0	0	0	0	0	)

Ici, on peut observer que sur la portion des débuts de préfixe où les deux mots valent 1, le résultat vaut 1. Similairement, on peut remarquer que dans $1^31^6(100) \; on \; 0^3(10) = 0^3101010(100000)$, sur la portion des débuts de préfixe où le premier mot vaut 1 et le second vaut 0, le résultat vaut 0.

Ces remarques se formalisent ainsi :

$$\begin{aligned} 1^d.w \; on \; 1^d.w' &= 1^d.(w \; on \; w') \\ 1^d.w \; on \; 0^d.w' &= 0^d.(w \; on \; w') \end{aligned}$$

On peut améliorer le calcul de la valeur abstraite en exploitant ces formules et la remarque suivante. Si l'on a une enveloppe a avec $concr\,(a) = \{w \mid w = 1^d.w_s\}$, on peut calculer l'enveloppe a' telle que $concr\,(a') = \{w_s \mid 1^d.w_s \in concr\,(a)\}$. Dans les graphiques des fonctions de cumul, cette opération revient à effectuer un changement de repère plaçant l'origine au point (d, d). Inversement, si l'on a une enveloppe a', on peut obtenir l'enveloppe a telle que $concr\,(a) = \{1^d.w_s \mid w_s \in concr\,(a')\}$ en effectuant le changement de repère inverse, c'est-à-dire en plaçant l'origine au point $(-d, -d)$.

Supposons que l'on désire calculer a_1 $\texttt{on}^\sim$ a_2, avec a_1 et a_2 des enveloppes de décalage b^0 positif, c'est-à-dire ne contenant que des mots commençant par une série de `1`. On peut éviter la perte due à un élargissement de a_1 et a_2 en appliquant les étapes suivantes :

1. Calculer le nombre commun d de `1` au début des mots de a_1 et des mots de a_2. Les mots de a_1 $\texttt{on}^\sim$ a_2 commenceront tous par d éléments `1`.
2. Effectuer des changements de repères sur a_1 et a_2 pour manipuler les abstractions a'_1 et a'_2 des mots tronqués de leurs d premiers éléments `1` ;
3. Appliquer la formule de l'opérateur $\texttt{on}^\sim$ initial, en élargissant préalablement a'_1 ou a'_2 au besoin. On obtient une enveloppe a' contenant les mots résultat de l'opération `on` sur les mots opérandes tronqués ;
4. Effectuer le changement de repère inverse sur a', pour ajouter la série de d éléments `1` présents au début des mots résultats de l'opération sur les mots non tronqués.

Le traitement du cas où a_2 ne représente que des mots commençant par une série de `0` peut-être réalisé de manière similaire. Pour supprimer (resp. ajouter) des séries de `0`, on effectue des changements de repère plaçant l'origine au point $(d, 0)$ (resp. $(-d, 0)$).

L'algorithme que nous venons d'expliquer a été implémenté. Cependant, ce n'est pas l'opérateur utilisé dans les exemples et les applications, car l'ensemble des preuves présentées dans cette thèse portent sur l'opérateur n'appliquant pas l'optimisation décrite.

7.6.3 Précision de la fonction d'abstraction

La valeur abstraite d'un mot est d'autant plus précise que les `1` et `0` du mot sont équitablement répartis. Nous abordons ci-dessous les deux cas de figure extrêmes : les mots contenant des séries de `0` et de `1`, c'est-à-dire dont les éléments sont mal répartis, et les mots bien équilibrés.

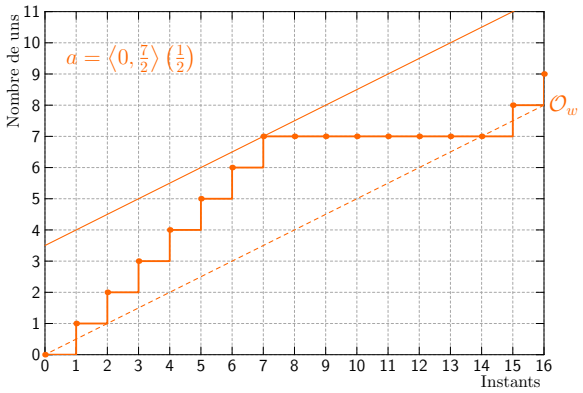
Séries de `0` et de `1`

Mis à part en début de préfixe, on ne peut pas forcer les mots d'une enveloppe à contenir des suites d'éléments identiques. La présence de telles séries dans un mot implique que son enveloppe contient une quantité importante d'autres mots. Graphiquement parlant, cela signifie que son enveloppe est large.

Considérons par exemple les enveloppes des mots qui sont constitués d'une alternance entre une série de `1` et une série de `0` :

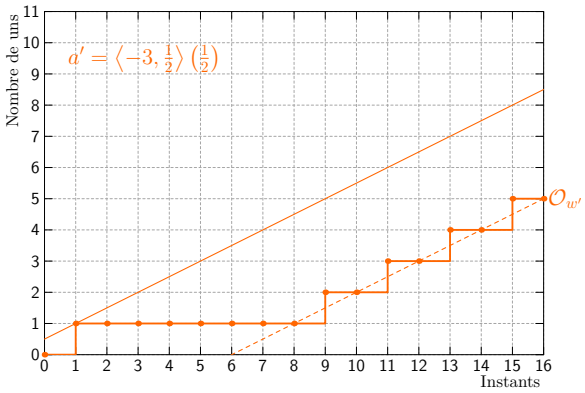
$$a = abs\,(\texttt{(}1^n0^m\texttt{)}) = \left\langle 0, \frac{n \times m}{n+m} \right\rangle \left(\frac{n}{n+m}\right)$$

L'enveloppe a est d'autant plus large que n et m sont grands.



$w = (1^7 0^7)$

L'occurrence des paquets de `1` et de `0` produit une enveloppe moins précise que pour un mot dans lequel les `1` et les `0` sont mieux répartis.



$w' = 10^7(10)$

L'occurrence d'un `1` à l'instant 1 empêche de garder l'information sur la série de `0` qui suit.

FIG. 7.24 – Exemples de mots dont l'enveloppe est large

La figure 7.24 montre l'abstraction du mot $(1^7 0^7)$, qui contient tous les mots compris entre $1^7(01)$ et (10). Cette figure montre aussi l'abstraction du mot $10^7(10)$ qui est elle aussi large, à cause du `1` en début de préfixe. La droite Δ^1 (en trait plein) doit autoriser son occurrence, ce qui empêche de conserver l'information de la présence d'une suite de `0` juste après.

Mots périodiques équilibrés

Nous avons annoncé dans la section 7.3.1 que l'abstraction d'un mot périodique équilibré ne contient que ce mot et ne constitue donc pas une perte d'information. On pourrait donc croire que si les horloges utilisées pour échantillonner les flots ne sont que des mots équilibrés, alors il est équivalent de vérifier les relations dans le domaine abstrait et dans le domaine concret. Pourtant, la composition de mots équilibrés par l'opération *on* n'est pas forcément un mot équilibré. Par exemple, `(111110)` et `(11110)` sont des mots équilibrés mais

$$(111110) \textit{ on } (11110) = (111100)$$

n'est pas un mot équilibré (les deux `0` sont côte à côte alors qu'ils pourraient être séparés par deux `1`). Par conséquent, ce mot n'est pas seul dans son enveloppe et il est moins précis de vérifier les relations sur sa valeur abstraite que sur sa valeur concrète.

Par contre, l'application de l'opération *not* à un mot équilibré est un mot équilibré. Nous avons vu que l'opérateur $not^{\sim}$ est le plus précis, donc la négation de la valeur abstraite d'un mot équilibré est une enveloppe ne contenant que la négation de ce mot.

Mots affines

Nous avons aussi vu dans la section 7.3.1 que comme l'enveloppe d'un mot équilibré, l'enveloppe d'un mot affine ne contient que ce mot. Contrairement au cas des mots équilibrés, la négation d'un mot affine n'est pas un mot affine, et composition par l'opérateur *on* de deux

mots affines est un mot affine. Considérons les horloges composées seulement d'opérations on. L'opérateur $on^{\sim}$ est suffisamment précis pour que la composition de deux enveloppes de mots affines soit une enveloppe ne contenant que le mot affine résultat de l'opération concrète. En d'autres termes, si $p_a = p_{a_1}\ on\ p_{a_2}$, alors :

$$concr\,(abs\,(p_{a_1})\ on^{\sim}\ abs\,(p_{a_2})) = concr\,(abs\,(p_a)) = \{p_a\}$$

Il n'y a donc aucune perte d'information à l'application de l'opérateur $on^{\sim}$. Comme les tests de précédence et de synchronisabilité sur les enveloppes en forme normale sont les plus précis, on peut en déduire qu'il est équivalent de tester la relation d'adaptabilité sur les mots affines et sur leur enveloppe. De même, le calcul de la taille du buffer sur les enveloppes de mots affines produit le même résultat que le calcul sur les valeurs concrètes. Pour ce cas particulier où la manipulation des valeurs concrètes est simple, le raisonnement dans le domaine abstrait est donc équivalent.

7.7 Conclusion

Le coût de l'abstraction d'un mot périodique est linéaire en la taille de ce mot. Parmi les opérations et relations sur les valeurs abstraites, la plus coûteuse est le calcul du $on^{\sim}$, qui ne nécessite que trois multiplications et deux additions sur les nombres rationnels. L'abstraction choisie atteint donc bien son objectif de vérifier la relation d'adaptabilité à peu de frais. On peut se demander si elle donne des résultats suffisamment précis pour être utiles. Les sources d'imprécision sont de deux natures. La première est la perte d'information intrinsèque à l'abstraction des mots. La seconde est l'imprécision des opérations et relations abstraites. En ce qui concerne la première, elle est d'autant plus importante que les instants de présence des horloges sont mal répartis. Nous verrons dans le chapitre 9 sur l'exemple de l'encodage GSM que même dans ces cas défavorables où les horloges sont "par rafales", les résultats obtenus sont raisonnables. Nous verrons également que dans le cas de l'incrustation d'images où cette fois-ci les horloges sont assez régulières, les résultats sont très bons. En ce qui concerne la seconde source d'imprécision, une grosse partie des opérations sont les plus précises. Nous avons quantifié l'imprécision des autres, et donné des idées d'amélioration pour l'opérateur $on^{\sim}$ qui est à l'origine de la principale perte d'information. Une fois de plus, les résultats du chapitre 9 montrent que cette imprécision n'est pas critique en pratique. Pour conclure, nous pensons avoir trouvé un bon compromis entre coût et précision, pour la vérification de la relation d'adaptabilité sur les horloges.

Chapitre 8

Typage des horloges abstraites

On définit dans ce chapitre un langage d'horloges abstraites pour Lucy-n, fondé sur l'algèbre des horloges abstraites présentée dans le chapitre précédent. Comme nous l'avons fait pour le langage d'horloges périodiques de la partie II, nous définissons les expressions d'horloges, ainsi que des algorithmes de résolution des contraintes d'égalité et de sous-typage sur les types de ces expressions. Ce chapitre fournit donc les outils nécessaires à l'inférence de types sur le langage d'horloges abstraites.

La résolution des contraintes d'égalité doit être effectuée de manière structurelle et repose essentiellement sur l'égalité des noms. En effet, la manipulation d'approximations des horloges ne permet pas de garantir leur égalité de manière plus puissante. Par contre, l'approximation permet une résolution des contraintes de sous-typage beaucoup plus légère que celle du langage d'horloges périodiques. Nous verrons qu'elle se résume en la résolution de systèmes d'inégalités linéaires de taille comparable aux nombre de contraintes de sous-typage.

8.1 Un langage d'horloges abstraites

Les horloges élémentaires du langage d'horloges proposé sont des horloges que l'on sait abstraire. Tout comme dans le langage d'horloges périodiques, celles-ci peuvent être composées avec les opérateurs de négation et d'échantillonnage :

$$
\begin{array}{lcl}
ce & ::= & c \mid p \mid \texttt{random in}\ a \mid \textit{not}\ ce \mid ce\ \textit{on}\ ce \\
p & ::= & u(v) \\
u & ::= & \varepsilon \mid b.u \\
v & ::= & b \mid b.v \\
b & ::= & \texttt{0} \mid \texttt{1} \\
a & ::= & \langle rat, rat \rangle\, (rat) \\
rat & ::= & int \mid int/int
\end{array}
$$

Les expressions d'horloges élémentaires sont soit une variable d'horloge c, soit mot binaire ultimement périodique p, soit une horloge inconnue `random in` a spécifiée par son abstraction. Les expressions d'horloges peuvent être composées avec les opérateurs *not* et *on*.

L'interprétation d'une expression ce dans un environnement ρ est définie ainsi :

$$\begin{array}{lcll}
[\![u(v)]\!]^{ce}_{\rho} & = & u.(lim_{n\to+\infty}\, v^n) & \text{avec } v^0 = \varepsilon \text{ et } v^{n+1} = v.v^n \\
[\![c]\!]^{ce}_{\rho} & = & \rho(c) & \\
[\![\texttt{random in}\, \langle b^0, b^1\rangle\,(r)]\!]^{ce}_{\rho} & = & w \text{ avec } w \in concr\left(\langle b^0, b^1\rangle\,(r)\right) & \\
[\![\mathit{not}\ ce]\!]^{ce}_{\rho} & = & \mathit{not}\ [\![ce]\!]^{ce}_{\rho} & \text{(définition 3.5 page 53)} \\
[\![ce_1\ \mathit{on}\ ce_2]\!]^{ce}_{\rho} & = & [\![ce_1]\!]^{ce}_{\rho}\ \mathit{on}\ [\![ce_1]\!]^{ce}_{\rho} & \text{(définition 3.6 page 53)}
\end{array}$$

Le seul ajout par rapport au langage d'horloges périodiques est l'opérateur `random in` a. On peut voir dans l'interprétation de cet opérateur qu'il désigne une horloge quelconque dans l'enveloppe a. Cette horloge est choisie au hasard dans leur enveloppe lors de la génération de code.

On ne sait généralement pas affirmer l'égalité de deux horloges définies par le constructeur `random in`, car elles sont choisies de manière aléatoire dans un ensemble d'horloges. Nous avons donc choisi de considérer ces horloges toujours différentes. Les mots périodiques sont quant à eux comparés avec le test d'égalité défini dans le lemme 4.2 (page 79). Les expressions composées sont comparées de manière structurelle.

La fonction d'abstraction des horloges périodiques est donnée dans la définition 7.5, nous la notons ci-dessous abs_p. L'abstraction d'une horloge construite par la primitive `random in` a est a, et la fonction d'abstraction d'expressions d'horloges composées avec les opérateurs not et on est définie dans la définition 7.10 en utilisant les opérateurs abstraits not et on. Nous disposons donc d'une fonction d'abstraction des expressions de notre langage d'horloges :

$$\begin{array}{lcl}
abs\,(p) & = & abs_p\,(p) \\
abs\,(\texttt{random in}\, a) & = & a \\
abs\,(ce_1\ \mathit{on}\ ce_2) & = & ce_1\ \mathit{on}^{\sim}\ ce_2 \\
abs\,(\mathit{not}\ ce) & = & \mathit{not}^{\sim}\ ce
\end{array}$$

La proposition 7.26 fournit un test d'adaptabilité sur les horloges abstraites. Il est donc toujours possible de tester la relation d'adaptabilité entre deux expressions d'horloges en calculant leur abstraction. Cependant, on peut tout d'abord appliquer des simplifications structurelles, permettant non seulement d'alléger les calculs, mais surtout de diminuer le nombre de cas où le test échoue alors que la relation est vérifiée. Nous noterons $<:_{ce}$ le test d'adaptabilité ainsi obtenu.

Définition 8.1 (test d'adaptabilité). Soit $cl = p$ ou $\mathit{not}\ p$ ou c ou $\mathit{not}\ c$. Soient ce_1 et ce_2 des expressions quelconques du langage d'horloges abstraites.

$$\begin{array}{lcll}
cl <:_{ce} cl & & & \\
\mathit{not}\ ce_1 <:_{ce} \mathit{not}\ ce_2 & \overset{def}{\Leftrightarrow} & ce_2 <:_{ce} ce_1 & \\
cl\ \mathit{on}\ ce_1 <:_{ce} cl\ \mathit{on}\ ce_2 & \overset{def}{\Leftrightarrow} & ce_1 <:_{ce} ce_2 & \text{avec } rate(cl) \neq 0 \\
ce_1\ \mathit{on}\ cl <:_{ce} ce_2\ \mathit{on}\ cl & \overset{def}{\Leftrightarrow} & ce_1 <:_{ce} ce_2 & \text{avec } rate(cl) \neq 0 \\
ce_1 <:_{ce} ce_2 & \overset{def}{\Leftrightarrow} & abs\,(ce_1) <:^{\sim} abs\,(ce_2) & \text{dans tous les autres cas}
\end{array}$$

Les trois premières règles sont correctes et complètes, et doivent donc être appliquées en priorité. La quatrième règle effectue une simplification à droite du on. Elle est correcte mais

pas complète. Malgré son incomplétude, cette règle rejette moins de cas corrects que la dernière règle qui vérifie la relation d'adaptabilité sur l'abstraction des expressions d'horloges. Cette dernière règle ne doit donc être appliquée que lorsque les autres règles ne s'appliquent pas.

Ce test d'adaptabilité est donc correct : $ce_1 <:_{ce} ce_2 \Leftarrow ce_1 <: ce_2$.

Démonstration :

$1^{ère}$ **règle :** Par définition de $<:$, on a $w <: w$. Donc si ce ne contient pas de sous-expression `random in` a qui représente un mot différent à chaque occurrence, on a $ce <: ce$. Pour éviter de tester l'égalité d'expressions intermédiaires ce trop complexes, on restreint l'application de ce cas à des ce simples à comparer, c'est-à-dire p, `not` p, c, `not` c.

$2^{ième}$ **règle :** Par la proposition 3.8 (page 72), `not` $ce_1 <:_{ce}$ `not` $ce_2 \Leftrightarrow ce_2 <:_{ce} ce_1$

$3^{ième}$ **règle :** Par la proposition 3.8 (page 72),
si $rate(ce) \neq 0$,
on a ce `on` $ce_1 <: ce$ `on` $ce_2 \Leftrightarrow ce_1 <: ce_2$.
Même remarque que pour la règle précédente en ce qui concerne la forme de ce.

$4^{ième}$ **règle :** Par la proposition 3.8 (page 72), si $rate(ce) \neq 0$,
on a ce_1 `on` $ce <:_{ce} ce_2$ `on` $ce \Leftarrow ce_1 <:_{ce} ce_2$.
Même remarque que pour les règles précédentes en ce qui concerne la forme de ce.

$5^{ième}$ **règle :** Par la proposition 7.27 (page 175), $ce_1 <:_{ce} ce_2 \Leftarrow abs(ce_1) <:^{\sim} abs(ce_2)$.

□

Enfin, le calcul de la taille de buffer nécessaire sur les horloges abstraites a aussi été fourni dans la section 7.5. Une fois de plus, pour calculer cette taille sur des expressions d'horloges, on commence par effectuer des simplifications structurelles avant d'abstraire les expressions et d'effectuer le calcul sur les valeurs abstraites obtenues.

Définition 8.2 (taille du buffer). Soit $cl = p$ ou `not` p ou c ou `not` c. Soient ce_1 et ce_2 des expressions quelconques du langage d'horloges abstraites.

$$\begin{array}{lcll} size_{ce}(cl \texttt{ on } ce_1, cl \texttt{ on } ce_2) & \stackrel{def}{=} & size_{ce}(ce_1, ce_2) & \text{si } rate(cl) \neq 0 \\ size_{ce}(ce_1, ce_2) & \stackrel{def}{=} & size^{\sim}(abs(ce_1), abs(ce_2)) & \text{dans tous les autres cas} \end{array}$$

Ce calcul est correct : $size(ce_1, ce_2) \leq size_{ce}(ce_1, ce_2) \leq size(ce_1, ce_2) + 1$.

Démonstration :

Par le lemme 3.14 et la proposition 7.28. □

Extensions du langage

Remarquons que le langage d'horloges abstraites peut être complété par n'importe quelle construction introduisant une horloge dont on connaît l'abstraction, soit parce qu'on sait la calculer, soit qu'elle est fournie par le programmeur. Par exemple, on a défini dans la

section 7.3.4 un opérateur *jitter* sur les mots périodiques, permettant de désigner une horloge périodique soumise à une certaine gigue. La fonction d'abstraction de cet opérateur a été définie, donc on peut introduire dans le langage un opérateur `jitter`. Il en est de même pour les opérateurs *delay* et *init* décrits dans cette même section. Tout comme pour l'opérateur `random in`, les instants de présence des expressions d'horloges utilisant ces opérateurs ne sont pas complètement définis. Deux choix sont possibles pour la génération de code :

1. Générer une horloge au hasard répondant à la spécification (c'est-à-dire dans l'enveloppe). Ceci peut être utile à des fins de simulation.
2. Utiliser une horloge complètement explicitée, fournie par le programmeur. Dans ce cas, les garanties données par le calcul d'horloge sont soumises à la condition que l'horloge fournie correspond à sa spécification. Par exemple, si l'horloge a été spécifiée par `jitter((10),2)`, il faut que sa valeur effective corresponde à l'horloge périodique `(10)`, soumise à une gigue d'au maximum 2 instants.

En ce qui concerne le second choix, on peut imaginer prendre l'horloge en entrée du programme, ou donner la possibilité au programmeur de définir la valeur concrète d'une horloge par un nœud écrit en Lucy-n, ne prenant pas d'argument et produisant une valeur booléenne à chaque activation. Le calcul d'horloge assure alors que si le nœud produit effectivement des valeurs dans l'enveloppe donnée comme spécification, alors le programme peut s'exécuter avec des buffers bornés et calcule une taille suffisante pour ceux-ci.

Les types d'horloges dans ce cadre
Tout comme dans le chapitre 5, on peut simplifier les types en les mettant sous la forme $ck ::= \alpha \mid \alpha \texttt{ on } ce$ à l'aide de la fonction *simpl* définie page 96 qui, grâce à l'associativité de $\texttt{on}$, peut rassembler au sein d'une même expression d'horloges les échantillonnages successifs exprimés dans le type (par exemple $(\alpha \texttt{ on } ce_1) \texttt{ on } ce_2 = \alpha \texttt{ on } (ce_1 \texttt{ on } ce_2)$).

Nous présentons dans la section suivante un algorithme de résolution des contraintes utilisant le mécanisme d'abstraction du chapitre 7.

8.2 Résolution des contraintes de sous-typage

L'algorithme de résolution des contraintes que nous allons présenter interprète les horloges par la valeur de leur abstraction. L'abstraction de l'horloge `(1)` étant neutre pour l'opération d'échantillonnage, on peut sans restriction transformer les types de la forme α vers le type équivalent α `on (1)`.

L'ensemble de contraintes de sous-typage collecté durant l'inférence de types peut donc être mis sous la forme $C = \{\alpha_{x_i} \texttt{ on } ce_{x_i} <: \alpha_{y_i} \texttt{ on } ce_{y_i}\}_{i=1..\text{nombre de contraintes}}$.

Résoudre un système de contraintes C consiste à trouver une instanciation θ pour les variables de type telle que le système $\theta(C)$ est satisfait. Rappelons que sur les types de la forme α `on` ce, la relation de sous-typage est vérifiée si et seulement si les variables de type sont les mêmes et les expressions d'horloges sont reliées par la relation d'adaptabilité (voir la section 2.3) :

$$\alpha \texttt{ on } ce_1 <: \alpha \texttt{ on } ce_2 \Leftrightarrow ce_1 <: ce_2$$

8.2.1 Simplification du système

Une contrainte α `on` $ce_1 <: \alpha$ `on` ce_2 portant sur deux types exprimés en fonction de la même variable d'horloge est vérifiée si et seulement si $ce_1 <: ce_2$.

On teste donc la valeur de vérité des contraintes n'impliquant qu'une variable d'horloge en vérifiant la relation d'adaptabilité à l'aide du test de la définition 8.1. Si le test renvoie vrai, la relation d'adaptabilité est vérifiée et la contrainte de sous-typage est toujours vérifiée. On peut donc supprimer cette contrainte du système. Si le test renvoie faux, alors on ne peut pas garantir que la relation d'adaptabilité est vérifiée, et donc que la relation de sous-typage est vérifiée. Le programme est alors rejeté par le système de types.

Une fois que les contraintes à vérifier ont été supprimées, si le système est vide c'est que les contraintes sont toujours satisfaites et la substitution résultat est vide. Si le système contient encore des contraintes, alors elles portent toutes sur deux variables différentes. On les transforme alors en un système de contraintes d'adaptabilité.

8.2.2 Transformation en un système de contraintes d'adaptabilité abstraites

On peut chercher sans restriction une substitution résultat θ telle que pour toute contrainte α_x `on` $ce_x <: \alpha_y$ `on` ce_y, on ait :

$$\left.\begin{array}{rcl}\theta(\alpha_x) & = & \alpha \texttt{ on } c_x \\ \theta(\alpha_y) & = & \alpha \texttt{ on } c_y\end{array}\right\} \text{ avec } c_x \texttt{ on } ce_x <: c_y \texttt{ on } ce_y$$

Si le système de contraintes est connexe, alors la substitution résultat exprime toutes les variables α_{x_i}, α_{y_i} en fonction d'une unique variable α. Si ce n'est pas le cas, on sépare le système en autant de sous-systèmes qu'il y a de composantes connexes, et on résout les sous-systèmes de manière indépendante. Le système de contraintes de sous-typage devient :

$$\theta(C) = \{\alpha \texttt{ on } c_{x_i} \texttt{ on } ce_{x_i} <: \alpha \texttt{ on } c_{y_i} \texttt{ on } ce_{y_i}\}_{i=1..\text{nombre de contraintes}}$$

Par définition de la relation de sous-typage, ce système est vérifié si et seulement si l'ensemble de contraintes d'adaptabilité suivant est vérifié :

$$A = \{c_{x_i} \texttt{ on } ce_{x_i} <: c_{y_i} \texttt{ on } ce_{y_i}\}_{i=1..\text{nombre de contraintes}}$$

On doit maintenant trouver une instanciation pour les variables c_{x_i} et c_{y_i} telle que A soit vérifié. En appliquant les règles du test d'adaptabilité 8.1, on obtient un ensemble de contraintes d'adaptabilité abstraites :

$$A^{\sim} = \{abs\,(c_{x_i} \texttt{ on } ce_{x_i}) <:^{\sim} abs\,(c_{y_i} \texttt{ on } ce_{y_i})\}_{i=1..\text{nombre de contraintes}}$$

En appliquant la définition de la fonction d'abstraction, on obtient :

$$\begin{array}{rcl} A^{\sim} & = & \{abs\,(c_{x_i}) \texttt{ on}^{\sim} abs\,(ce_{x_i}) <:^{\sim} abs\,(c_{y_i}) \texttt{ on}^{\sim} abs\,(ce_{y_i})\}_{i=1..\text{nombre de contraintes}} \\ & = & \{h_{x_i} \texttt{ on}^{\sim} a_{x_i} <:^{\sim} h_{y_i} \texttt{ on}^{\sim} a_{y_i}\}_{i=1..\text{nombre de contraintes}} \end{array}$$

avec $h_{x_i} = abs\,(c_{x_i}) = \left\langle {b^0}_{h_{x_i}}, {b^1}_{h_{x_i}} \right\rangle \left(r_{h_{x_i}}\right)$ et $a_{x_i} = abs\,(ce_{x_i}) = \left\langle {b^0}_{a_{x_i}}, {b^1}_{a_{x_i}} \right\rangle (r_{a_{x_i}})$, et de même pour h_{y_i} et a_{y_i}.

Une contrainte d'adaptabilité abstraite $h_x \ \mathbf{on}^\sim \ a_x <:^\sim h_y \ \mathbf{on}^\sim \ a_y$ est définie par une contrainte de synchronisabilité et une contrainte de précédence (proposition 7.26) :

$$h_x \ \mathbf{on}^\sim \ a_x <:^\sim h_y \ \mathbf{on}^\sim \ a_y$$
$$\Leftrightarrow$$
$$(h_x \ \mathbf{on}^\sim \ a_x \bowtie^\sim h_y \ \mathbf{on}^\sim \ a_y) \ \wedge \ (h_x \ \mathbf{on}^\sim \ a_x \preceq^\sim h_y \ \mathbf{on}^\sim \ a_y)$$

Les résultats des opérations $h_x \ \mathbf{on}^\sim \ a_x$ et $h_y \ \mathbf{on}^\sim \ a_y$ sont des valeurs abstraites que l'on notera respectivement $\left\langle {b^0}_x, {b^1}_x \right\rangle (r_x)$ et $\left\langle {b^0}_y, {b^1}_y \right\rangle (r_y)$. La relation de synchronisabilité abstraite peut être vérifiée de la manière suivante (proposition 7.21) :

$$h_x \ \mathbf{on}^\sim \ a_x \bowtie^\sim h_y \ \mathbf{on}^\sim \ a_y \quad \Leftrightarrow \quad r_x = r_y$$

La relation de précédence abstraite peut quand à elle être testée ainsi (proposition 7.24) :

$$h_x \ \mathbf{on}^\sim \ a_x \preceq^\sim h_y \ \mathbf{on}^\sim \ a_y \quad \Leftarrow \quad {b^1}_y - {b^0}_x < 1$$

On s'intéresse donc aux valeurs $\left\langle b^0, b^1 \right\rangle (r)$ des $h \ \mathbf{on}^\sim \ a$ (i.e. des $h_x \ \mathbf{on}^\sim \ a_x$ et $h_y \ \mathbf{on}^\sim \ a_y$). Par la définition 7.7, on a :

$$\begin{aligned} h \ \mathbf{on}^\sim \ a &= \left\langle {b^0}_h, {b^1}_h \right\rangle (r_h) \ \mathbf{on}^\sim \left\langle {b^0}_a, {b^1}_a \right\rangle (r_a) \\ &= \left\langle \min(0, {b^0}_h) \times r_a + \min(0, {b^0}_a), \ {b^1}_h \times r_a + {b^1}_a \right\rangle (r_h \times r_a) \end{aligned}$$

Les relations de synchronisabilité et de précédence abstraites se réécrivent donc ainsi :

$$h_x \ \mathbf{on}^\sim \ a_x \bowtie^\sim h_y \ \mathbf{on}^\sim \ a_y \quad \Leftrightarrow \quad r_{h_x} \times r_{a_x} = r_{h_y} \times r_{a_y}$$
$$h_x \ \mathbf{on}^\sim \ a_x \preceq^\sim h_y \ \mathbf{on}^\sim \ a_y \quad \Leftarrow \quad ({b^1}_{h_y} \times r_{a_y} + {b^1}_{a_y}) - (\min(0, {b^0}_{h_x}) \times r_{a_x} + \min(0, {b^0}_{a_x})) < 1$$

Le système de contraintes d'adaptabilité abstraites a été décomposé en un système assurant la satisfaction des contraintes de synchronisabilité, et un système assurant la satisfaction des contraintes de précédence :

$$\left\{ r_{h_{x_i}} \times r_{a_{x_i}} = r_{h_{y_i}} \times r_{a_{y_i}} \right\}_{i=1..\text{nombre de contraintes}}$$

$$\left\{ ({b^1}_{h_{y_i}} \times r_{a_{y_i}} + {b^1}_{a_{y_i}}) - (\min(0, {b^0}_{h_{x_i}}) \times r_{a_{x_i}} + \min(0, {b^0}_{a_{x_i}})) < 1 \right\}_{i=1..\text{nombre de contraintes}}$$

8.2.3 Résolution des contraintes de synchronisabilité abstraites

On cherche à résoudre le système suivant :

$$\left\{ r_{h_{x_i}} \times r_{a_{x_i}} = r_{h_{y_i}} \times r_{a_{y_i}} \right\}_i$$

Pour cela, on va exprimer l'exprimer sous la forme $\left\{ r_{h_{x_i}} = k_i \times r_{h_{y_i}} \right\}_i$ telle que :

- $k_i \leq 1$;
- chaque variable apparaît au maximum une fois à gauche ;
- toute variable apparaissant à droite n'apparaît jamais à gauche.

Dans ce cas, la substitution $[r_{h_{x_i}} \leftarrow k_i ; \ r_{h_{y_i}} \leftarrow 1]_i$ est la solution du système qui maximise la valeur des taux.

Détail de la transformation du système

La transformation du système est faite en deux étapes. Premièrement, pour chaque contrainte $r_{h_x} \times r_{a_x} = r_{h_y} \times r_{a_y}$, si $r_{a_x} > r_{a_y}$, on exprime r_{h_x} en fonction de r_{h_y} ($r_{h_x} = r_{h_y} \times \frac{r_{a_y}}{r_{a_x}}$). Dans le cas contraire, on exprime r_{h_y} en fonction de r_{h_x} ($r_{h_y} = r_{h_x} \times \frac{r_{a_x}}{r_{a_y}}$). On obtient donc un système sous la forme :

$$\left\{ r_{h_{x_i}} = k_i \times r_{h_{y_i}} \right\}_i \text{ avec } k_i \leq 1$$

Une contrainte $r_{h_x} = k \times r_{h_y}$ définit maintenant une substitution $r_{h_x} \leftarrow k \times r_{h_y}$.

La seconde transformation est la saturation du système. Elle consiste à appliquer la substitution associée à chaque contrainte à toutes les variables apparaissant dans les autres contraintes. La contrainte obtenue après substitution doit être soumise aux traitements suivants :

- Lorsqu'on substitue la variable située à gauche du signe d'égalité, on obtient une contrainte sous la forme $k \times r_{h_x} = k' \times r_{h_y}$ qu'il faut remettre sous la forme $r_{h_x} = k'' \times r_{h_y}$ avec $k'' \leq 1$, en utilisant la méthode expliquée plus haut.
- Lorsqu'on obtient une contrainte de la forme $r_{h_x} = k \times r_{h_x}$, il faut vérifier que $k = 1$. Dans ce cas, la contrainte est toujours vérifiée et on peut donc la supprimer du système. Dans le cas contraire, le système de contraintes de synchronisabilité n'a pas de solution. Le programme est alors rejeté par le système de types, car les contraintes de sous-typage ne peuvent pas être satisfaites.

Une fois qu'on a appliqué toutes les substitutions définies par les contraintes, on obtient un système où les variables apparaissant à gauche sont toutes différentes et n'apparaissent pas à droite. De plus, comme le système est connexe, la variable de droite est la même dans toutes les équations.

Ce système exprime donc toutes les variables de taux en fonction d'une unique variable non contrainte. Comme on désire obtenir des taux maximaux, on choisit d'affecter la valeur 1 à cette dernière variable, et on obtient ainsi la valeur de toutes les autres.

8.2.4 Résolution des contraintes de précédence abstraites

Durant l'étape de résolution des contraintes de synchronisabilité que nous avons présentée dans la section précédente, nous avons inféré des valeurs rationnelles pour les r_{h_x} et les r_{h_y} (nous les supposerons sous forme irréductible). Les inconnues restantes pour trouver les valeurs des h_x et des h_y sont donc : ${b^0}_{h_x}$, ${b^1}_{h_x}$, ${b^0}_{h_y}$ et ${b^1}_{h_y}$.

Ces variables sont contraintes par le système suivant :

$$\left\{ ({b^1}_{h_{y_i}} \times r_{a_{y_i}} + {b^1}_{a_{y_i}}) - (\min(0, {b^0}_{h_{x_i}}) \times r_{a_{x_i}} + \min(0, {b^0}_{a_{x_i}})) < 1 \right\}_{i=1..\text{nombre de contraintes}}$$

Lemme 8.1. Si le système de contraintes d'adaptabilité abstraites a une solution, alors il a une solution telle que toutes les horloges inférées sont en forme normale.

Démonstration :

Soit $A^\sim$ le système de contraintes d'adaptabilité. Supposons que $A^\sim$ admet comme solution la substitution ϕ, c'est-à-dire $\phi(A^\sim)$ est satisfait. Soit ϕ' la substitution telle que pour toute variable d'horloge h, $\phi'(h)$ est la forme normale de $\phi(h)$. Alors, par la proposition 7.5, pour tout h, $concr\,(\phi(h)) = concr\,(\phi'(h))$. Cela signifie tout

d'abord que si $concr\,(\phi(h)) \neq \varnothing$, alors $concr\,(\phi'(h)) \neq \varnothing$. D'autre part, pour tout h, $\phi'(h)$ $\mathtt{on}^{\sim}$ $a \sqsubseteq^{\sim} \phi(h)$ $\mathtt{on}^{\sim}$ a (par définition de $\mathtt{on}^{\sim}$). Par définition de $<:^{\sim}$, cela signifie que pour toute contrainte h_x $\mathtt{on}^{\sim}$ $a_x <:^{\sim} h_y$ $\mathtt{on}^{\sim}$ a_y de $A^{\sim}$, si $\phi(h_x)$ $\mathtt{on}^{\sim}$ $a_x <:^{\sim} \phi(h_y)$ $\mathtt{on}^{\sim}$ a_y est vérifiée, alors $\phi'(h_x)$ $\mathtt{on}^{\sim}$ $a_x <:^{\sim} \phi'(h_y)$ $\mathtt{on}^{\sim}$ a_y l'est aussi. On peut en conclure que ϕ' est une solution de $A^{\sim}$. □

On peut par conséquent décider sans restriction de chercher des h_x et des h_y en forme normale, c'est-à-dire tels que les taux sont sous forme irréductible, et les décalages rationnels sont au même dénominateur que le taux. Pour cela, on pose :

$$\begin{array}{rclcrcl} {b^{\mathtt{0}}}_{h_x} & = & \dfrac{k^{\mathtt{0}}_{h_x}}{l_{h_x}} & & {b^{\mathtt{0}}}_{h_y} & = & \dfrac{k^{\mathtt{0}}_{h_y}}{l_{h_y}} \\ {b^{\mathtt{1}}}_{h_x} & = & \dfrac{k^{\mathtt{1}}_{h_x}}{l_{h_x}} & \text{et} & {b^{\mathtt{1}}}_{h_y} & = & \dfrac{k^{\mathtt{1}}_{h_x}}{l_{h_x}} \end{array}$$

avec l_{h_x} le dénominateur de r_{h_x} et l_{h_y} celui de r_{h_y}, qui sont connus.

Il s'agit donc de trouver des valeurs pour $k^{\mathtt{0}}_{h_x}$, $k^{\mathtt{1}}_{h_x}$, $k^{\mathtt{0}}_{h_y}$ et $k^{\mathtt{1}}_{h_y}$, telles que les contraintes de précédence sont vérifiées, c'est-à-dire telles que :

$$\left(\frac{k^{\mathtt{1}}_{h_y}}{l_{h_y}} \times r_{a_y} + {b^{\mathtt{1}}}_{a_y}\right) - \left(\min(0, \frac{k^{\mathtt{0}}_{h_x}}{l_{h_x}}) \times r_{a_x} + \min(0, {b^{\mathtt{0}}}_{a_x})\right) < 1$$

Pour se ramener à des contraintes d'inégalités linéaires à coefficients entiers, on met les coefficients sous forme irréductible :

$$\frac{n_x}{l_x} = red(\frac{r_{a_x}}{l_{h_x}}) \quad \text{et} \quad \frac{n_y}{l_y} = red(\frac{r_{a_y}}{l_{h_y}})$$

puis on multiplie les inéquations par le plus petit commun multiple des dénominateurs de ces coefficients, noté *den*. On obtient alors les coefficients entiers suivants :

$$z_x = n_x \times \frac{den}{l_x} \text{ et } z_y = n_y \times \frac{den}{l_y} \quad \text{avec } den = \text{ ppcm } (l_x, l_y)$$

Chaque contrainte se réécrit en :

$$\left(k^{\mathtt{1}}_{h_y} \times z_y + {b^{\mathtt{1}}}_{a_y} \times den\right) - \left(\min(0, k^{\mathtt{0}}_{h_x}) \times z_x + (\min(0, {b^{\mathtt{0}}}_{a_x}) \times den)\right) < 1 \times den$$

On obtient un système de la forme :

$$\left\{k^{\mathtt{1}}_{h_{y_i}} \times z_{y_i} - \min(0, k^{\mathtt{0}}_{h_{x_i}}) \times z_{x_i} < (\min(0, {b^{\mathtt{0}}}_{a_{x_i}}) - {b^{\mathtt{1}}}_{a_{y_i}} + 1) \times den\right\}_{i=1..\text{nombre de contraintes}}$$

Comme les parties gauches des inégalités prennent des valeurs entières, ce système est équivalent à :

$$\left\{k^{\mathtt{1}}_{h_{y_i}} \times z_{y_i} - \min(0, k^{\mathtt{0}}_{h_{x_i}}) \times z_{x_i} \leq \left\lceil (\min(0, {b^{\mathtt{0}}}_{a_{x_i}}) - {b^{\mathtt{1}}}_{a_{y_i}} + 1) \times den \right\rceil - 1\right\}_{i=1..\text{nombre de contraintes}}$$

et finalement, il est équivalent à :

$$\left\{ \begin{array}{rcl} k^{\mathtt{1}}_{h_{y_i}} \times z_{y_i} - k^{\mathtt{0}}_{h_{x_i}} \times z_{x_i} & \leq & \left\lceil (\min(0, {b^{\mathtt{0}}}_{a_{x_i}}) - {b^{\mathtt{1}}}_{a_{y_i}} + 1) \times den \right\rceil - 1 \\ k^{\mathtt{1}}_{h_{y_i}} \times z_{y_i} & \leq & \left\lceil (\min(0, {b^{\mathtt{0}}}_{a_{x_i}}) - {b^{\mathtt{1}}}_{a_{y_i}} + 1) \times den \right\rceil - 1 \end{array} \right\}_{i=1..\text{nombre de contraintes}}$$

Ce système d'inéquations linéaires sur les nombres entiers peut-être résolu à l'aide d'un outil externe comme Glpk. Avant cela, le système doit être complété par d'autres contraintes. En effet, comme $h_x = abs\,(c_x)$ et $h_y = abs\,(c_y)$, il faut de plus que $concr\,(h_x) \neq \varnothing$ et $concr\,(h_y) \neq \varnothing$ pour toutes les variables h_x et h_y inférées. Par la proposition 7.6, c'est le cas si et seulement si :

$$\left\{k^{\mathtt{0}}_{h_i} - k^{\mathtt{1}}_{h_i} \leq 1 - l_{h_i}\right\}_{i=1..\text{nombre de variables d'horloges}}$$

Ces contraintes sur les $k^{\mathtt{0}}_{h_i}$ et les $k^{\mathtt{1}}_{h_i}$ doivent donc être ajoutées au système, qui peut enfin être résolu avec l'outil Glpk.

8.2.5 Solution du système concret

Nous avons montré comment calculer un ensemble de valeurs abstraites h_{x_i} et h_{y_i} telles que le système de contraintes d'adaptabilité abstraites $A^{\sim}$est vérifié. Toute instanciation associant aux c_{x_i} un mot de l'enveloppe h_{x_i} et aux c_{y_i} un mot de l'enveloppe h_{y_i} satisfait le système de contraintes d'adaptabilité concrètes A. Nous avons donc décidé d'associer respectivement les horloges `random in` h_{x_i} et `random in` h_{y_i} aux c_{x_i} et c_{y_i}. Nous n'avons pas encore réfléchi au choix d'horloges effectives les plus judicieuses lors de la génération de code.

Cas des horloges affines

Si l'on restreint les expressions élémentaires du langage aux horloges affines, et que l'on aménage la résolution abstraite de manière à n'inférer que des horloges affines, celle-ci est équivalente à la résolution concrète. En effet, nous avons vu dans le chapitre précédent qu'il est équivalent de tester la relation d'adaptabilité sur les valeurs abstraites des horloges affines et sur leurs valeurs concrètes. Nous avons aussi vu que pour ces horloges, la simplification structurelle à droite du *on* est correcte et complète pour la relation d'adaptabilité. Le test d'adaptabilité abstrait de la définition 8.1 est donc équivalent au test sur les valeurs concrètes. L'algorithme de résolution des contraintes de synchronisabilité doit être adapté, afin de n'inférer que des taux de la forme $\frac{1}{\ell}$, et des décalages tels que les enveloppes contiennent un unique élément (qui sera forcément une horloge affine). Pour cela, il suffit de diviser les taux obtenus avec l'algorithme fourni par le ppcm des numérateurs. La résolution des contraintes de précédence peut être réalisée en suivant l'algorithme fourni, mais en imposant que les décalages verticaux soient négatifs, afin d'assurer que les préfixes des mots inférés ne seront composés que de `0`, et en transformant l'inégalité assurant que l'enveloppe contient au moins un élément en une égalité assurant que l'enveloppe contient exactement un élément.

8.3 Conclusion

Le système de types utilisant la résolution des contraintes sur les horloges abstraites a été implanté et a été utilisé pour typer les exemples du chapitre 2. Il calcule sur ces exemples les mêmes types que le système de types utilisant la résolution des contraintes sur les horloges périodiques. Nous verrons dans le prochain chapitre que la résolution abstraite apporte un gain important sur le temps de calcul nécessaire pour le typage de l'application *Picture in Picture*, et sur la qualité du type obtenu. En ce qui concerne l'application d'encodage GSM, le résultat obtenu est moins bon que le résultat de la résolution concrète. Nous discuterons dans

la partie IV de l'idée de combiner les deux méthodes dans le cas de systèmes n'impliquant que des horloges périodiques. Pour finir, l'intérêt indubitable de la résolution abstraite est de permettre de modéliser des systèmes impliquant des horloges qui ne sont pas exactement périodiques, comme c'est le cas en présence de gigue. Cette méthode permet d'obtenir des garanties fortes sur un système, même en présence d'incertitudes sur son rythme d'exécution exact.

Chapitre 9

Applications et discussion

Ce chapitre applique la méthode de résolution avec abstraction des horloges à nos cas d'étude du chapitre 6. Nous la nommerons résolution abstraite. La méthode de la partie II, qui raisonne sur les horloges périodiques, sera nommée résolution concrète.

Nous reprenons dans la section 9.1 l'exemple de l'application *Picture in Picture* sur lequel la résolution concrète avait fourni des résultats insatisfaisants. Après une description approfondie du code de l'application, nous analysons les types d'horloges inférés par la résolution abstraite, et montrons que les résultats sont cette fois satisfaisants.

La section 9.2 détaille pas à pas l'algorithme de résolution des contraintes de typage sur notre deuxième cas d'étude, l'encodage de canal GSM. Cet exemple, bien que mal adapté à notre méthode d'abstraction, donne un résultat convenable.

Enfin, la section 9.3 montre que la résolution abstraite n'est pas seulement utile pour traiter des horloges périodiques trop longues pour être manipulées efficacement de manière exacte, mais aussi pour traiter des systèmes dont les horloges ne sont pas exactement périodiques.

9.1 Exemple du *Picture in Picture*

Nous étudions ici plus en détails le code du nœud `picture_in_picture`. Ce nœud reçoit en entrée les flots de pixels de deux images en haute définition (HD: 1920×1080 pixels), réduit la taille de la première vers une basse définition (SD: 720×480 pixels) en utilisant un sous-nœud `downscaler`, et incruste la petite image obtenue dans la seconde image d'entrée (voir la figure 9.1 et la figure 1.1 page 12).

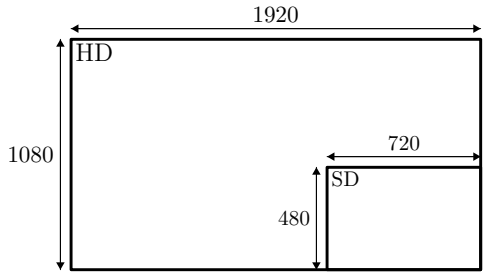


Fig. 9.1 – Image produite par le *Picture in Picture*

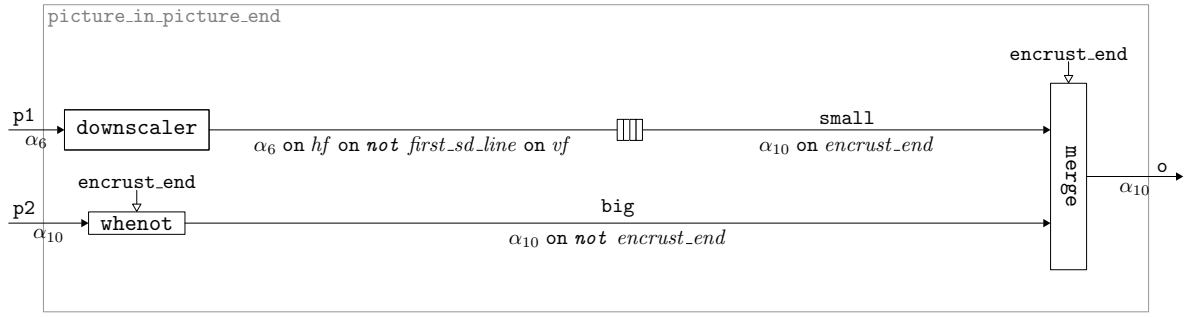


FIG. 9.2 – L'application *Picture in Picture*. Les types d'horloges avant résolution des contraintes de sous-typage sont indiqués sous les fils.

```
51 (* picture in picture *)
52 let clock encrust_end =
53   (0^(1920 * (1080 - 480)) {0^1200 1^720}^480)
54
55 let node picture_in_picture_end (p1, p2) = o where
56   rec small = buffer(downscaler p1)
57   and big = (p2 whenot encrust_end)
58   and o = merge encrust_end small big
```

L'horloge `encrust_end` détermine la provenance des pixels de l'image résultat, dans le cas où l'on veut incruster la petite image en bas à droite de la grande, comme sur la figure 9.1. Elle vaut `1` quand c'est la petite image qu'il faut afficher, et `0` quand c'est la grande. La notation `{0^1200 1^720}^480` est un raccourci pour la répétition de `480` fois le motif `0^1200 1^720`. Si l'on désire incruster la petite image en haut à gauche de la grande, on utilise l'horloge :

```
let clock encrust_begin =
  ({1^720 0^1200}^480 0^(1920 * (1080 - 480)))
```

La petite image est obtenue par application du nœud `downscaler`, dont le code est fourni dans la figure 9.3. Il est composé d'un filtre horizontal, qui applique une convolution puis un échantillonnage réduisant la taille des lignes de 1920 vers 720, et d'un filtre vertical effectuant lui aussi une convolution et réduisant par échantillonnage la taille des colonnes de 1080 vers 480. Pour effectuer la convolution, le filtre vertical a besoin de disposer des pixels situés au dessus et en dessous du pixel traité. Pour cette raison, il applique d'abord un nœud `reorder` qui produit les pixels de trois lignes d'images. Le premier élément du triplet retourné par `reorder` (`p0`) contient les pixels de la ligne précédente, sauf sur la première ligne de chaque image, où il contient les pixels de la ligne courante. Ces pixels sont conservés dans un buffer initialisé, à l'aide du nœud `my_fby_sd_line` semblable au nœud `my_fby3` de la page 31. Le deuxième élément du triplet (`p1`) contient les pixels de la ligne courante. Ceux-ci doivent eux aussi être conservés dans un buffer en attendant l'arrivée des pixels de `p2`. Le troisième élément (`p2`) contient les pixels de la ligne suivante, sauf sur la dernière ligne de chaque image, où il contient les pixels de la ligne courante.

Tout comme lors de l'utilisation de la résolution concrète, le type inféré pour le nœud `downscaler` est :

```
downscaler ::
forall 'a. 'a -> 'a on hf on not first_sd_line on vf
```

```
1  (* convolutions *)
2  let node convo (c0, c1, c2) = (c0 + c1 + c2) / 3
3
4  let node convolution (p0, p1, p2) = p where
5    rec p = (r,g,b)
6    and r = convo (p0r, p1r, p2r)
7    and g = convo (p0g, p1g, p2g)
8    and b = convo (p0b, p1b, p2b)
9    and p0r, p0g, p0b = p0
10   and p1r, p1g, p1b = p1
11   and p2r, p2g, p2b = p2
12
13 (* horizontal filter *)
14 let clock hf = (10100100)
15
16 let node horizontal_filter p = o where
17   rec p0 = p fby p1
18   and p1 = p fby p2
19   and p2 = p
20   and o = (convolution (p0, p1, p2)) when hf
21
22 (* vertical sliding_window *)
23 let clock first_sd_line = 1^720 (0)
24 let clock first_line_of_img = (1^720 0^(720*1079))
25 let clock last_line_of_img = (0^(720*1079) 1^720)
26
27 let node my_fby_sd_line (p1,p2) =
28   merge first_sd_line (p1 when first_sd_line) (buffer(p2))
29
30 let node reorder p = ((p0,p1,p2)::'a) where
31   rec p0 =
32     merge first_line_of_img
33       (p1 when first_line_of_img)
34       ((my_fby_sd_line (p1, p1)) whenot first_line_of_img)
35   and p1 = buffer(p)
36   and p2 =
37     merge last_line_of_img
38       (p1 when last_line_of_img)
39       ((p whenot first_sd_line) whenot last_line_of_img)
40
41 (* vertical filter *)
42 let clock vf = (1^720 0^720 1^720 0^720 0^720 1^720 0^720 0^720 1^720)
43
44 let node vertical_filter p = o where
45   rec (p0,p1,p2) = reorder p
46   and o = (convolution (p0, p1, p2)) when vf
47
48 (* downscaler *)
49 let node downscaler p = vertical_filter (horizontal_filter p)
```

FIG. 9.3 – Le code du *Downscaler*

Comme la sortie du nœud `downscaler` est connectée à l'entrée du nœud de fusion par l'intermédiaire d'un buffer, le type du nœud `picture_in_picture_end` avant généralisation est `('a6 * 'a10) -> 'a10`, avec les contraintes suivantes sur `'a6` et `'a10` :

```
{ 'a6 on hf on not first_sd_line on vf <: 'a10 on encrust_end
    (* line 56, characters 14-35 *) }
```

Le type inféré est :

```
picture_in_picture_end ::
forall 'a. ('a * 'a on <-4315, -4315>(1)) -> 'a on <-4315, -4315>(1)
Buffer line 56, characters 14-35: size = 193079
```

L'unique horloge dans l'enveloppe $\langle -4315, -4315\rangle$ (1) est $\mathtt{0}^{4315}\mathtt{(1)}$, donc le type obtenu est équivalent à :

$$\mathit{picture\_in\_picture\_end} : \forall\alpha.\,(\alpha \times \alpha \texttt{ on } \mathtt{0}^{4315}\mathtt{(1)}) \rightarrow \alpha \texttt{ on } \mathtt{0}^{4315}\mathtt{(1)}$$

Cela signifie que si l'image destinée à être réduite arrive en continu à partir du premier instant, l'image destinée à recevoir l'incrustation doit arriver en continu après un délai de 4315 instants, et l'image résultat sera produite en continu après ce même délai, soit approximativement le temps de recevoir deux lignes d'images haute définition. Pour mieux comprendre ce résultat, observons d'une part le type de sortie du nœud `downscaler` après le calcul de l'expression d'horloges `hf on not first_sd_line on vf` :

downscaler :
$\forall\alpha.\,\alpha \rightarrow \alpha \texttt{ on } \mathtt{0}^{1920}(\{\mathtt{10100100}\}^{240}\mathtt{0}^{1920}\{\mathtt{10100100}\}^{240}\mathtt{0}^{3840}\{\mathtt{10100100}\}^{240}\mathtt{0}^{3840}\{\mathtt{10100100}\}^{240})$
Le délai introduit par le *downscaler* est de 1920 instants, c'est-à-dire le temps de recevoir une ligne HD de pixels : le filtre vertical doit attendre l'arrivée des pixels de la deuxième ligne HD avant de commencer à produire l'image réduite, à cause de la convolution à appliquer verticalement.
D'autre part, le nœud de fusion n'introduit pas de délai. Le délai introduit par le *Picture in Picture* est donc de 1920 instants. Le type obtenu surévalue d'un peu plus d'une ligne HD le nombre d'instants nécessaires avant de pouvoir commencer à produire des sorties ($4315 = 1920 + 1920 + 475$). Ce résultat n'est pas étonnant, car les `1` du type de sortie du *downscaler* sont bien répartis à un peu plus d'une ligne HD près. Il est satisfaisant que la précision des calculs sur l'abstraction soit de cet ordre.

La taille de buffer calculée est aussi très satisfaisante : 193 079 places nécessaires contre 191 970 pour la résolution concrète, soit un surcoût de stockage d'environ une ligne et demi d'image SD.

Nous avons vu dans le chapitre 6 que le système de contraintes de type du *Picture in Picture* nécessite une journée de calculs pour être résolu avec la résolution concrète. La méthode de résolution sur les horloges abstraites répond en quelques secondes.

Sur cet exemple problématique pour la résolution concrète, la résolution abstraite infère donc des types et des tailles de buffers très satisfaisants, en un temps de calcul lui aussi satisfaisant.

9.2 Exemple du codage de canal GSM

Les programmes des nœuds de l'encodeur de canal GSM ont été donnés dans le chapitre 6, et l'intégralité des programmes de l'encodeur et du décodeur sont rassemblés dans l'annexe H.

9.2.1 Typage des nœuds de l'encodeur

Les nœuds `split_speech`, `join_50_3`, et `multiplexer`, qui ne contiennent pas de buffer, reçoivent ici le même type qu'avec la méthode concrète, car l'unification structurelle est suffisante pour résoudre les contraintes d'égalité.

Les nœuds `cyclic_encoding` et `convolutionnal_encoding`, qui contiennent des buffers, reçoivent eux aussi le même type, car la résolution des contraintes se résume à de la vérification. Illustrons cela sur le cas du nœud `convolutionnal_encoding`. Le système de contraintes à résoudre pour typer ce nœud est :

$$\{\alpha \texttt{ on (10)} <: \alpha \texttt{ on not (10)}\}$$

Comme les deux types impliqués dans la contrainte sont exprimés en fonction de la même variable de type, cette contrainte est traitée lors de l'étape de simplification du système décrite dans la section 8.2.1. Il s'agit donc d'utiliser le test d'adaptabilité de la définition 8.1 pour vérifier que `(10)` $<:$ `not (10)`. Comme aucune simplification structurelle n'est possible, c'est la dernière règle du test qui est appliquée, c'est-à-dire $ce_1 <: ce_2 \Leftarrow abs\,(ce_1) <:^{\sim} abs\,(ce_2)$. Pour cela, les expressions d'horloges sont abstraites :

$$\begin{array}{lcl} abs\,(\texttt{(10)}) & = & \langle 0, \frac{1}{2}\rangle\left(\frac{1}{2}\right) \\ abs\,(\texttt{not (10)}) & = & \texttt{not}^{\sim}\, abs\,(\texttt{(10)}) \;=\; \texttt{not}^{\sim}\left(\langle 0, \frac{1}{2}\rangle\left(\frac{1}{2}\right)\right) \;=\; \langle -\frac{1}{2}, 0\rangle\left(\frac{1}{2}\right) \end{array}$$

Comme les taux sont égaux et les enveloppes ne se chevauchent pas, d'après les tests de synchronisabilité et de précédence abstraits (propositions 7.21 et 7.24 pages 172 et 174), la relation d'adaptabilité est vérifiée sur les enveloppes des expressions :

$$\left\langle 0, \frac{1}{2}\right\rangle\left(\frac{1}{2}\right) <:^{\sim} \left\langle -\frac{1}{2}, 0\right\rangle\left(\frac{1}{2}\right)$$

Par conséquent, la relation d'adaptabilité `(10)` $<:$ `not (10)` est vérifiée. La contrainte peut donc être enlevée du système. Une fois simplifié, ce système est vide :

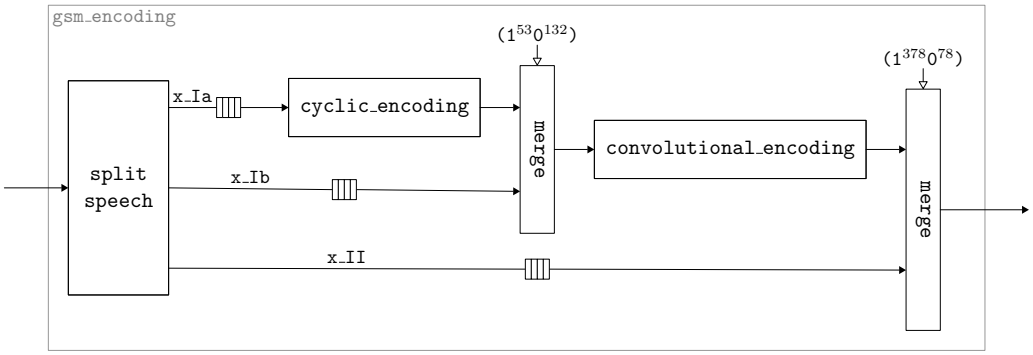
```
Constraints to simplify:
 { 'a21 on (10) <: 'a21 on not (10) (* line 53, characters 26-36 *) }
No constraint to solve.
convolutional_encoding ::
forall 'a. 'a on (10) on (1^185 0^4 ) -> 'a
Buffer line 53, characters 26-36: size = 1
```

La taille de buffer suffisante calculée est $size^{\sim}(\langle 0, \frac{1}{2}\rangle\left(\frac{1}{2}\right), \langle -\frac{1}{2}, 0\rangle\left(\frac{1}{2}\right)) = \left\lfloor \frac{1}{2} - (-\frac{1}{2})\right\rfloor = 1$.

Les enveloppes $\langle 0, \frac{1}{2}\rangle\left(\frac{1}{2}\right)$ et $\langle -\frac{1}{2}, 0\rangle\left(\frac{1}{2}\right)$ ne contenant qu'une horloge, le test d'adaptabilité abstrait est ici équivalent au test d'adaptabilité concret. Il n'y a pas eu besoin d'appliquer de substitution pour satisfaire les contraintes. Il n'est donc pas étonnant d'obtenir le même type et la même taille de buffer que lors de la résolution concrète.

9.2.2 Typage de l'encodeur

La connexion des nœuds de l'encodeur ne peut ici être réalisée sans buffer.



En effet, l'unification structurelle échoue à satisfaire la contrainte d'égalité entre le type de la première sortie du nœud `split_speech` qui est de la forme α_1 `on` $(1^{50}0^{210})$ et le type d'entrée du nœud `cyclic_encoding` qui est de la forme α_2 `on` $(1^{52}0)$ `on` $(1^{51}0)$ `on` $(1^{50}0)$. Il en est de même pour la connexion des deux autres sorties de `split_speech`. Il est donc obligatoire d'établir la communication à travers des buffers, afin d'obtenir des contraintes de sous-typage plutôt que des contraintes d'égalité.

```
69 let node encode_with_3_buffers x = y where
70   rec (x_Ia, x_Ib, x_II) = split_speech x
71   and y1 = cyclic_encoding (buffer(x_Ia))
72   and y1_x_Ib = merge (1^53 0^132) y1 (buffer(x_Ib))
73   and y2 = convolutional_encoding y1_x_Ib
74   and y = merge (1^378 0^78) y2 (buffer(x_II))
```

Le type du nœud avant la généralisation est : $\alpha_9 \to \alpha_{15}$, avec les contraintes suivantes sur α_9 et α_{15} (notées `'a9` et `'a15`) :

```
Constraints to simplify:
 { 'a9 on (0^182 1^78 ) <:
     'a15 on not (1^378 0^78 )
     (* line 74, characters 33-45 *)
   'a9 on (0^50 1^132 0^78 ) <:
     'a15 on (1^378 0^78 ) on (10) on (1^185 0^4 ) on not (1^53 0^132 )
     (* line 72, characters 39-51 *)
   'a9 on (1^50 0^210 ) <:
     'a15 on (1^378 0^78 ) on (10) on (1^185 0^4 ) on (1^53 0^132 )
                                        on (1^52 0) on (1^51 0) on (1^50 0)
     (* line 71, characters 28-40 *) }
```

Nous détaillons ci-dessous la résolution abstraite de ce système, en suivant les étapes décrites dans les sous-sections de la section 8.2.

Simplification des contraintes

Comme toutes les contraintes portent sur deux variables distinctes, il n'y a pas de simplification à réaliser. Les contraintes à résoudre sont donc les suivantes :

$$C = \left\{ \begin{array}{lcl} \alpha_9 \texttt{ on } (1^{50}0^{210}) & <: & \alpha_{15} \texttt{ on } (1^{378}0^{78}) \texttt{ on } (10) \texttt{ on } (1^{185}0^{4}) \texttt{ on } (1^{53}0^{132}) \\ & & \qquad \texttt{on } (1^{52}0) \texttt{ on } (1^{51}0) \texttt{ on } (1^{50}0) \\ \alpha_9 \texttt{ on } (0^{50}1^{132}0^{78}) & <: & \alpha_{15} \texttt{ on } (1^{378}0^{78}) \texttt{ on } (10) \texttt{ on } (1^{185}0^{4}) \texttt{ on not } (1^{53}0^{132}) \\ \alpha_9 \texttt{ on } (0^{182}1^{78}) & <: & \alpha_{15} \texttt{ on not } (1^{378}0^{78}) \end{array} \right\}$$

Ces contraintes sont issues (dans l'ordre) des buffers des lignes 71, 72 et 74.

Transformation en un système de contraintes d'adaptabilité abstraites

Ce système est tout d'abord transformé en un système de contraintes d'adaptabilité concrètes :

$$A = \left\{ \begin{array}{lcl} c_9 \ \mathit{on}\ (1^{50}0^{210}) & <: & c_{15} \ \mathit{on}\ (1^{378}0^{78}) \ \mathit{on}\ (10) \ \mathit{on}\ (1^{185}0^{4}) \ \mathit{on}\ (1^{53}0^{132}) \\ & & \qquad \mathit{on}\ (1^{52}0) \ \mathit{on}\ (1^{51}0) \ \mathit{on}\ (1^{50}0) \\ c_9 \ \mathit{on}\ (0^{50}1^{132}0^{78}) & <: & c_{15} \ \mathit{on}\ (1^{378}0^{78}) \ \mathit{on}\ (10) \ \mathit{on}\ (1^{185}0^{4}) \ \mathit{on\ not}\ (1^{53}0^{132}) \\ c_9 \ \mathit{on}\ (0^{182}1^{78}) & <: & c_{15} \ \mathit{on\ not}\ 1^{378}0^{78} \end{array} \right\}$$

Si les horloges c_9 et c_{15} satisfont ces contraintes, alors la substitution θ telle que $\theta(\alpha_9) = \alpha$ on c_9 et $\theta(\alpha_{15}) = \alpha$ on c_{15} est solution du système de contraintes de sous-typage C.

On traduit ensuite A en un système de contraintes d'adaptabilité abstraites :

$$A^{\sim} = \left\{ \begin{array}{lcl} h_9 \ \mathit{on}^{\sim}\ \mathit{abs}\left((1^{50}0^{210})\right) & <:^{\sim} & h_{15} \ \mathit{on}^{\sim}\ \mathit{abs}\left((1^{378}0^{78})\right) \\ & & \qquad \mathit{on}^{\sim}\ \mathit{abs}\left((10)\right) \ \mathit{on}^{\sim}\ \mathit{abs}\left((1^{185}0^{4})\right) \ \mathit{on}^{\sim}\ \mathit{abs}\left((1^{53}0^{132})\right) \\ & & \qquad\quad \mathit{on}^{\sim}\ \mathit{abs}\left((1^{52}0)\right) \ \mathit{on}^{\sim}\ \mathit{abs}\left((1^{51}0)\right) \ \mathit{on}^{\sim}\ \mathit{abs}\left((1^{50}0)\right) \\ h_9 \ \mathit{on}^{\sim}\ \mathit{abs}\left((0^{50}1^{132}0^{78})\right) & <:^{\sim} & h_{15} \ \mathit{on}^{\sim}\ \mathit{abs}\left((1^{378}0^{78})\right) \ \mathit{on}^{\sim}\ \mathit{abs}\left((10)\right) \ \mathit{on}^{\sim}\ \mathit{abs}\left((1^{185}0^{4})\right) \\ & & \qquad \mathit{on}^{\sim}\ \mathit{not}^{\sim}\ \mathit{abs}\left((1^{53}0^{132})\right) \\ h_9 \ \mathit{on}^{\sim}\ \mathit{abs}\left((0^{182}1^{78})\right) & <:^{\sim} & h_{15} \ \mathit{on}^{\sim}\ \mathit{not}^{\sim}\ \mathit{abs}\left((1^{378}0^{78})\right) \end{array} \right\}$$

Si les horloges abstraites h_9 et h_{15} satisfont ces contraintes, alors pour tout $c_9 \in \mathit{concr}\,(h_9)$, pour tout $c_{15} \in \mathit{concr}\,(h_{15})$, le système A est vérifié. Une fois les expressions abstraites calculées, $A^{\sim}$ vaut :

$$A^{\sim} = \left\{ \begin{array}{lcl} h_9 \ \mathit{on}^{\sim}\ \left\langle 0, \frac{525}{13} \right\rangle \left(\frac{5}{26}\right) & <:^{\sim} & h_{15} \ \mathit{on}^{\sim}\ \left\langle 0, \frac{5501}{114} \right\rangle \left(\frac{25}{228}\right) \\ h_9 \ \mathit{on}^{\sim}\ \left\langle -\frac{330}{13}, \frac{198}{5} \right\rangle \left(\frac{33}{65}\right) & <:^{\sim} & h_{15} \ \mathit{on}^{\sim}\ \left\langle -\frac{718}{19}, \frac{977}{38} \right\rangle \left(\frac{11}{38}\right) \\ h_9 \ \mathit{on}^{\sim}\ \left\langle -\frac{273}{5}, 0 \right\rangle \left(\frac{3}{10}\right) & <:^{\sim} & h_{15} \ \mathit{on}^{\sim}\ \left\langle -\frac{2457}{38}, 0 \right\rangle \left(\frac{13}{76}\right) \end{array} \right\}$$

On pose $h_9 = \left\langle {b^0}_{h_9}, {b^1}_{h_9} \right\rangle (r_{h_9})$ et $h_{15} = \left\langle {b^0}_{h_{15}}, {b^1}_{h_{15}} \right\rangle (r_{h_{15}})$, ainsi que ${b^{0\prime}}_{h_9} = \min(0, {b^0}_{h_9})$ et ${b^{0\prime}}_{h_{15}} = \min(0, {b^0}_{h_{15}})$,[1] et on calcule les $\mathit{on}^{\sim}$ restant :

$$\left\{ \begin{array}{lcl} \left\langle {b^{0\prime}}_{h_9} \times \frac{5}{26} + 0,\ {b^1}_{h_9} \times \frac{5}{26} + \frac{525}{13} \right\rangle \left(r_{h_9} \times \frac{5}{26}\right) & <:^{\sim} & \left\langle {b^{0\prime}}_{h_{15}} \times \frac{25}{228} + 0,\ {b^1}_{h_{15}} \times \frac{25}{228} + \frac{5501}{114} \right\rangle \left(r_{h_{15}} \times \frac{25}{228}\right) \\ \left\langle {b^{0\prime}}_{h_9} \times \frac{33}{65} - \frac{330}{13},\ {b^1}_{h_9} \times \frac{33}{65} + \frac{198}{5} \right\rangle \left(r_{h_9} \times \frac{33}{65}\right) & <:^{\sim} & \left\langle {b^{0\prime}}_{h_{15}} \times \frac{11}{38} - \frac{718}{19},\ {b^1}_{h_{15}} \times \frac{11}{38} + \frac{977}{38} \right\rangle \left(r_{h_{15}} \times \frac{11}{38}\right) \\ \left\langle {b^{0\prime}}_{h_9} \times \frac{3}{10} - \frac{273}{5},\ {b^1}_{h_9} \times \frac{3}{10} + 0 \right\rangle \left(r_{h_9} \times \frac{3}{10}\right) & <:^{\sim} & \left\langle {b^{0\prime}}_{h_{15}} \times \frac{13}{76} - \frac{2457}{38},\ {b^1}_{h_{15}} \times \frac{13}{76} + 0 \right\rangle \left(r_{h_{15}} \times \frac{13}{76}\right) \end{array} \right\}$$

Il s'agit maintenant de trouver des valeurs pour r_{h_9} et $r_{h_{15}}$, telles que les contraintes de synchronisabilité sont vérifiées, et des valeurs pour ${b^0}_{h_9}$, ${b^1}_{h_9}$, ${b^0}_{h_{15}}$, ${b^1}_{h_{15}}$, telles que les contraintes de précédence sont vérifiées.

Résolution des contraintes de synchronisabilité abstraites

Les contraintes de synchronisabilité sont satisfaites si et seulement si les taux sont égaux :

$$\left\{ \begin{array}{lcl} r_{h_9} \times \frac{5}{26} & = & r_{h_{15}} \times \frac{25}{228} \\ r_{h_9} \times \frac{33}{65} & = & r_{h_{15}} \times \frac{11}{38} \\ r_{h_9} \times \frac{3}{10} & = & r_{h_{15}} \times \frac{13}{76} \end{array} \right\} \text{ c'est-à-dire } \left\{ \begin{array}{lcl} r_{h_9} & = & r_{h_{15}} \times \frac{65}{114} \\ r_{h_9} & = & r_{h_{15}} \times \frac{65}{114} \\ r_{h_9} & = & r_{h_{15}} \times \frac{65}{114} \end{array} \right\}$$

[1] Les variables ${b^{0\prime}}_{h_9}$ et ${b^{0\prime}}_{h_{15}}$ servent à représenter l'élargissement éventuel préalable à l'application du on si ${b^0}_{h_9}$ ou ${b^0}_{h_{15}}$ sont positifs.

Après saturation du système, on obtient $\left\{ \; r_{h_9} = r_{h_{15}} \times \frac{65}{114} \; \right\}$. Pour obtenir des taux maximaux, on choisit $r_{h_{15}} = 1$ et $r_{h_9} = \frac{65}{114}$.

Résolution des contraintes de précédence abstraites

Les contraintes de précédence sont quant à elles satisfaites si et seulement si le chevauchement entre les enveloppes est inférieur à 1 :

$$\left\{ \begin{array}{lcl} (b^{\mathsf{1}}{}_{h_{15}} \times \frac{25}{228} + \frac{5501}{114}) - (\min(0, b^{\mathsf{0}}{}_{h_9}) \times \frac{5}{26} + 0) & < & 1 \\ (b^{\mathsf{1}}{}_{h_{15}} \times \frac{11}{38} + \frac{977}{38}) - (\min(b^{\mathsf{0}}{}_{h_9}) \times \frac{33}{65} - \frac{330}{13}) & < & 1 \\ (b^{\mathsf{1}}{}_{h_{15}} \times \frac{13}{76} + 0) - (\min(0, b^{\mathsf{0}}{}_{h_9}) \times \frac{3}{10} - \frac{273}{5}) & < & 1 \end{array} \right\}$$

Comme on recherche des valeurs abstraites en forme normale, on sait d'après les solutions trouvées pour les taux ($r_{h_9} = \frac{65}{114}$ et $r_{h_{15}} = 1$) que les décalages sont de la forme a:

$$b^{\mathsf{0}}{}_{h_9} = \frac{k^{\mathsf{0}}_{h_9}}{114} \quad b^{\mathsf{1}}{}_{h_9} = \frac{k^{\mathsf{1}}_{h_9}}{114} \qquad b^{\mathsf{0}}{}_{h_{15}} = \frac{k^{\mathsf{0}}_{h_{15}}}{1} \quad b^{\mathsf{1}}{}_{h_{15}} = \frac{k^{\mathsf{1}}_{h_{15}}}{1} \qquad \text{avec } k^{\mathsf{0}}_{h_9}, k^{\mathsf{1}}_{h_9}, k^{\mathsf{0}}_{h_{15}}, k^{\mathsf{1}}_{h_{15}} \in \mathbb{N}^*$$

Si l'on instancie ces valeurs dans le système de contraintes, on obtient des contraintes sur $k^{\mathsf{0}}_{h_9}, k^{\mathsf{1}}_{h_9}, k^{\mathsf{0}}_{h_{15}}$ et $k^{\mathsf{1}}_{h_{15}}$:

$$\left\{ \begin{array}{lcl} k^{\mathsf{1}}_{h_{15}} \times \frac{25}{228} - \min(0, k^{\mathsf{0}}_{h_9} \times \frac{5}{114 \times 26}) & < & -\frac{5387}{114} \\ k^{\mathsf{1}}_{h_{15}} \times \frac{11}{38} - \min(0, k^{\mathsf{0}}_{h_9} \times \frac{33}{114 \times 65}) & < & -\frac{24747}{494} \\ k^{\mathsf{1}}_{h_{15}} \times \frac{13}{76} - \min(0, k^{\mathsf{0}}_{h_9} \times \frac{3}{114 \times 10}) & < & -\frac{268}{5} \end{array} \right\}$$

On se ramène à un système d'inéquations ne contenant que des coefficients entiers en multipliant chaque inéquation par le plus petit commun multiple des dénominateurs des deux coefficients. On obtient :

$$\left\{ \begin{array}{lcl} k^{\mathsf{1}}_{h_{15}} \times 325 - \min(0, k^{\mathsf{0}}_{h_9} \times 5) & < & -140062 \\ k^{\mathsf{1}}_{h_{15}} \times 715 - \min(0, k^{\mathsf{0}}_{h_9} \times 11) & < & -123735 \\ k^{\mathsf{1}}_{h_{15}} \times 65 - \min(0, k^{\mathsf{0}}_{h_9} \times 1) & < & -20368 \end{array} \right\}$$

Ce système est équivalent à :

$$\left\{ \begin{array}{lcl} k^{\mathsf{1}}_{h_{15}} \times 325 - \min(0, k^{\mathsf{0}}_{h_9} \times 5) & \leq & -140063 \\ k^{\mathsf{1}}_{h_{15}} \times 715 - \min(0, k^{\mathsf{0}}_{h_9} \times 11) & \leq & -123736 \\ k^{\mathsf{1}}_{h_{15}} \times 65 - \min(0, k^{\mathsf{0}}_{h_9}) & \leq & -20369 \end{array} \right\}$$

c'est-à-dire à :

$$\left\{ \begin{array}{lcl} k^{\mathsf{1}}_{h_{15}} \times 325 - k^{\mathsf{0}}_{h_9} \times 5 & \leq & -140063 \\ k^{\mathsf{1}}_{h_{15}} \times 325 & \leq & -140063 \\ k^{\mathsf{1}}_{h_{15}} \times 715 - k^{\mathsf{0}}_{h_9} \times 11 & \leq & -123736 \\ k^{\mathsf{1}}_{h_{15}} \times 715 & \leq & -123736 \\ k^{\mathsf{1}}_{h_{15}} \times 65 - k^{\mathsf{0}}_{h_9} & \leq & -20369 \\ k^{\mathsf{1}}_{h_{15}} \times 65 & \leq & -20369 \end{array} \right\}$$

À cela s'ajoutent les contraintes de non-vacuité, destinées à assurer que les valeurs abstraites h_9 et h_{15} contiennent au moins une horloge :

$$\begin{array}{lcl} k^{\mathsf{1}}_{h_9} - k^{\mathsf{0}}_{h_9} & \geq & 114 - 1 \\ k^{\mathsf{1}}_{h_{15}} - k^{\mathsf{0}}_{h_{15}} & \geq & 1 - 1 \end{array}$$

Le système contenant les contraintes de précédence et de non vacuité peut être résolu à l'aide d'un solveur de contraintes linéaires comme Glpk.[2] Lorsque l'on maximise $k^{\mathtt{1}}_{h_{15}}$, le délai de sortie est minimisé et on obtient :

$$k^{\mathtt{0}}_{h_9} = 0 \qquad k^{\mathtt{1}}_{h_9} = 113 \qquad k^{\mathtt{1}}_{h_{15}} = -431 \qquad k^{\mathtt{0}}_{h_{15}} = -431$$

Donc $h_9 = \langle \frac{0}{114}, \frac{113}{114} \rangle \left(\frac{65}{114}\right) = \langle 0, \frac{113}{114} \rangle \left(\frac{65}{114}\right)$ et $h_{15} = \langle -431, -431 \rangle (1)$.

Le typeur renvoie effectivement le type suivant :

```
encode_with_3_buffers ::
forall 'a. 'a on <0, 113/114>(65/114) -> 'a on <-431, -431>(1)
```

Comme les enveloppes apparaissant dans le type ne contiennent qu'un mot,[3] ce type est équivalent à :

$$\mathit{encode\_with\_3\_buffers} : \forall \alpha.\, \alpha \texttt{ on } (\{\texttt{1101010}\}^{16}\texttt{10}) \rightarrow \alpha \texttt{ on } \texttt{0}^{431}\texttt{(1)}$$

La sortie est disponible après un délai de 431 instants. Ce délai est absent dans le type de sortie calculé par la résolution concrète. Il correspond au nombre d'éléments d'un peu moins d'une trame de sortie.

Les tailles de buffers calculées sont les suivantes :

```
Buffer line 71, characters 28-40: size = 87
Buffer line 72, characters 39-51: size = 202
Buffer line 74, characters 33-45: size = 138
```

Ces tailles correspondent au double de la taille des sous-trames à stocker. Ce n'est pas un résultat très satisfaisant, nous aurions pu espérer des tailles égales aux tailles des sous-trames. Cependant, rappelons que les horloges utilisées dans cette application sont les horloges les moins favorables à l'opération d'abstraction, car elles contiennent de grosses séries de `0` et de `1`.

9.2.3 Typage du décodeur

Contrairement à la méthode concrète de la partie II, la méthode présentée dans cette partie permet de traiter le cas du décodeur de canal GSM décrit dans la section 6.2 :

```
71 let node decode_with_3_buffers x = y where
72   rec x_I_enc, x_II = split_I_II x
73   and x_I = convolutional_decoding x_I_enc
74   and x_Ia_enc, x_Ib = split_Ia_Ib x_I
75   and x_Ia = cyclic_decoding x_Ia_enc
76   and y = join_speech (buffer(x_Ia), buffer(x_Ib), buffer(x_II))
```

Le type obtenu est :

```
decode_with_3_buffers ::
forall 'a. 'a -> 'a on <-2911/3, -36835/38>(65/114)
Buffer line 76, characters 23-35: size = 85
Buffer line 76, characters 37-49: size = 412
Buffer line 76, characters 51-63: size = 355
```

[2] Le système généré par notre typeur est disponible en annexe F.

[3] Dans le cas contraire, le rythme effectivement choisi par le typeur est une horloge de l'enveloppe, que nous n'avons pas encore déterminée. On peut par exemple choisir la plus précoce ou la plus tardive.

9.3 Modélisation de rythmes variables

Nous avons vu dans les sections précédentes que l'abstraction des horloges permet de résoudre plus efficacement les contraintes sur les horloges périodiques. L'introduction d'horloges abstraites dans le langage ouvre aussi de nouvelles possibilités par rapport aux langages synchrones classiques : savoir donner, grâce au calcul d'horloge, des garanties sur des rythmes pas totalement spécifiés. Nous illustrons cela dans cette section en montrant comment modéliser les temps de calcul et de communication.

Modélisation du temps de calcul

Considérons le nœud `f` suivant :

```
1  let node f x = x + x
```

À l'aide des annotations de types, on peut spécifier que ce nœud sera exécuté au rythme d'un instant sur six :

```
let node compute_f (x :: 'a on (1 0^5)) = (y :: 'a on (1 0^5))
  where rec y = f x
```

Grâce aux horloges abstraites, on peut modéliser un temps de calcul variable pour le nœud. Par exemple, ici, on spécifie que la sortie est produite avec un retard de 0 à 3 instants après l'arrivée de l'entrée :

```
3  let clock computation_time_f = random in <-3,0>(1)
4
5  let node compute_f (x :: 'a on (1 0^5)) = (y :: 'a on computation_time_f on (1 0^5))
6    where rec y = buffer (f x)
```

Les horloges répondant à la spécification de `computation_time_f` sont :

$$concr\left(\langle -3,0\rangle\left(1\right)\right) = \left\{w \mid \texttt{(1)} <: w <: 0^3\texttt{(1)}\right\}$$

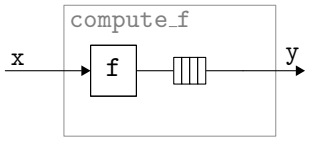
Étant données les annotations de types fournies sur l'entrée et la sortie de `compute_f`, le calcul d'horloge considère que les horloges de sortie possibles pour ce nœud sont les horloges de : $concr\left(\langle -3,0\rangle\left(1\right) \mathbin{on^{\sim}} abs\left(\texttt{(}10^5\texttt{)}\right)\right) = concr\left(\langle -3,0\rangle\left(1\right) \mathbin{on^{\sim}} \langle 0,\frac{5}{6}\rangle\left(\frac{1}{6}\right)\right) = concr\left(\langle -\frac{1}{2},\frac{5}{6}\rangle\left(\frac{1}{6}\right)\right)$ c'est-à-dire de l'ensemble $\left\{w \mid \texttt{(}10^5\texttt{)} <: w <: 0^3\texttt{(}10^5\texttt{)}\right\}$. Ces horloges et le programme `compute_f` sont représentés sur la figure 9.4. Le buffer placé sur le fil de sortie est nécessaire pour conserver les valeurs produites par le nœud `f` qui, suivant le modèle synchrone, sont disponibles simultanément à l'arrivée de `x`. Le type obtenu pour `compute_f` est le suivant :

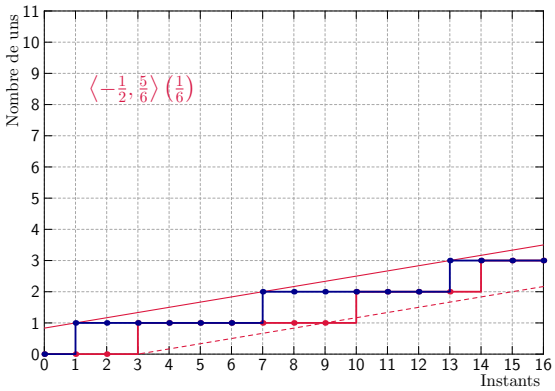
```
compute_f :: forall 'a. 'a on (10^5 ) -> 'a on computation_time_f on (10^5 )
Buffer line 6, characters 10-22: size = 1
```

Comme prévu, la taille nécessaire pour le buffer est d'une place, pour conserver la valeur calculée par `f` entre l'instant où elle est produite, c'est-à-dire l'instant d'arrivée de l'entrée, et l'instant où il a été déclaré que le calcul est terminé.

Composons ce nœud `f` avec un nœud `g` exécuté lui-aussi un instant sur six, et nécessitant un temps de calcul de 2 à 6 instants :

```
8  let node g x = x * x * x * x * x
9
10 let clock computation_time_g = random in <-6,-2>(1)
11
12 let node compute_g (x :: 'a on (1 0^5)) = (y :: 'a on computation_time_g on (1 0^5))
13   where rec y = buffer (g x)
```





La courbe de (10^5) représente le rythme d'entrée et sortie de `f`.
Celle de $0^2 10^3 0^3 10^2 010^4 0^3 10^2 \ldots$ représente un des rythmes possibles en sortie de `compute_f`.
La distance horizontale entre ces courbes est de 0 à 3 instants, qui représentent le temps de calcul que l'on désire modéliser. La distance verticale est de 0 à 1 éléments, qui correspondent au nombre d'éléments dans le buffer.

FIG. 9.4 – Modélisation du temps de calcul de `f`

```
15 let node fg x = z
16   where rec z = compute_g (compute_f x)
```

Les types obtenus sont :

```
compute_g :: forall 'a. 'a on (10^5 ) -> 'a on computation_time_g on (10^5 )
Buffer line 13, characters 10-22: size = 1
```

```
fg ::
forall 'a. 'a on (10^5 ) -> 'a on computation_time_f on computation_time_g on (10^5 )
```

On peut observer que le délai entre l'arrivée des entrées et la production des sorties du nœud composant `f` et `g` est défini par `computation_time_f on computation_time_g`, c'est-à-dire un mot appartenant à $\langle -3, 0 \rangle (1)$ $on^{\sim}$ $\langle -6, -2 \rangle (1) = \langle -9, -2 \rangle (1)$. Le temps de calcul du nœud `fg` est donc de 2 à 9 instants, ce qui correspond bien à la somme des temps de calcul de `f` et de `g`.

Modélisation du temps de communication

Suivant le même principe, on peut modéliser des temps de communication variables. Supposons que l'on veuille appliquer à un flot d'entrée trois fonctions `f`, `g` et `h`, de type $\forall \alpha.\, \alpha \to \alpha$.

```
1 let node f x = x + x
2 let node g x = x * x * x * x * x
3 let node h x = x + 3
```

On considère ici que les fonctions calculent leur sortie instantanément, mais on veut spécifier que :

- la communication sur le fil reliant `f` à `g` peut prendre de zéro à trois instants ;
- la communication sur le fil reliant `g` à `h` peut prendre de quatre à huit instants.

Pour cela, on utilise la méthode usuelle de modéliser les temps de transfert par des nœuds appliquant la fonction identité mais avec un "temps de calcul" s'élevant au temps de communication souhaité.

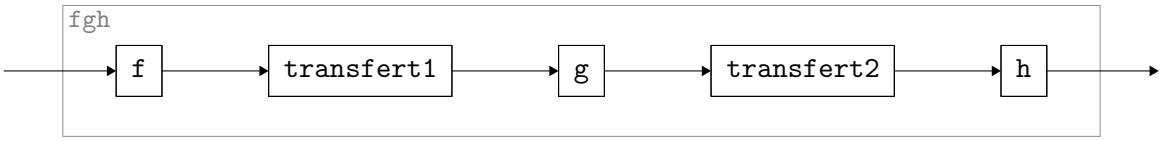


FIG. 9.5 – Modélisation du temps de communication

```
5  let clock communication_time1 = random in <-3,0>(1)
6  let clock communication_time2 = random in <-8,-4>(1)
7
8  let node transfert1 (x::'a) = (buffer(x) :: 'a on communication_time1)
9
10 let node transfert2 (x::'a) = (buffer(x) :: 'a on communication_time2)
11
12 let node fgh x = t where
13   rec y = transfert1 (f x)
14   and z = transfert2 (g y)
15   and t = h z
```

Le nœud `fgh` est représenté sur la figure 9.5 et les types inférés sont les suivants :

```
transfert1 :: forall 'a. 'a -> 'a on communication_time1
Buffer line 8, characters 33-42: size = 3

transfert2 :: forall 'a. 'a -> 'a on communication_time2
Buffer line 10, characters 33-42: size = 8

fgh :: forall 'a. 'a -> 'a on communication_time1 on communication_time2
```

Le temps de transfert spécifié sur le premier fil étant d'au maximum 3 instants, et les sorties de `f` étant produites à chaque instant, il y a au maximum 3 valeurs sur ce fil. La taille de buffer nécessaire pour stocker ces valeurs dans le nœud `transfer1` afin de simuler le temps de transfert est donc de 3 places. Suivant le même principe, le buffer placé dans `transfer2` doit contenir 8 places. On peut observer dans le type du nœud `fgh` que le délai entre l'arrivée des entrées et la production des sorties est défini par `communication_time1 on communication_time2`, c'est-à-dire un mot inconnu appartenant à $\langle -3, 0\rangle\,(1)$ $on^{\sim}$ $\langle -8, -4\rangle\,(1) = \langle -11, -4\rangle\,(1)$. Il y a donc de 4 à 11 instants de délai, ce qui correspond bien à la somme des temps de transfert des deux fils.

9.4 Conclusion

Les exemples de ce chapitre ont montré que la méthode avec abstraction permet de traiter toutes les horloges périodiques. Par rapport à la méthode de la partie II, on a l'avantage de savoir traiter les horloges de préfixe quelconque (comme dans le décodeur GSM), et les horloges dont le motif périodique est de taille trop longue pour supporter l'explosion combinatoire de la méthode concrète (comme dans l'incrustation d'images). La méthode présentée a aussi l'avantage de s'appliquer à des horloges qui ne sont pas exactement périodiques. Nous avons montré comment utiliser ces horloges pour modéliser des aspects habituellement absents dans les langages synchrones, comme les temps de communication et les temps de calcul variables.

Quatrième partie

Conclusion

Chapitre 10

Travaux connexes

Avant de conclure ce mémoire en présentant nos perspectives de recherches, nous présentons dans ce chapitre des travaux connexes aux nôtres. Dans la section 10.1, nous décrirons les travaux existant autour de l'interprétation d'horloges dans les langages synchrones. Dans la section 10.2, nous présenterons d'autres modèles pour la description de réseaux de processus à mémoire bornée. Dans la section 10.3, nous montrerons que l'approximation des rythmes à déjà été utilisée dans d'autres cadres. Enfin, dans la section 10.4, nous aborderons la question de la conception de circuits insensibles à la latence.

10.1 Horloges affines dans le modèle synchrone

L'idée d'utiliser des horloges suffisamment particulières pour être interprétables dans le calcul d'horloge n'est pas nouvelle. La principale piste explorée par le passé est l'utilisation d'horloges affines, c'est-à-dire constituées d'un délai initial puis d'un comportement régulier d'un instant de présence tous les ℓ instants. Ces horloges sont un cas particulier d'horloges ultimement périodiques. Nous leur avons porté attention au cours de ce document en leur consacrant des algorithmes spécialisés. Précisons que ces algorithmes ne sont pas encore implémentés dans Lucy-n. Nous décrivons dans la section 10.1.1 les travaux réalisés autour de ces horloges pour améliorer le calcul d'horloge du langage Signal et dans la section 10.1.2 les travaux réalisés dans le cadre du langage Lustre. Tous ces travaux augmentent le nombre de cas où le compilateur est en mesure de garantir l'égalité de deux types d'horloges.

10.1.1 Horloges affines en Signal

Les horloges affines ont été introduites en Signal dans le contexte du codesign Signal et Alpha [Sma98]. L'utilisation de nœuds programmés en Alpha dans un programme Signal produit pour ces nœuds différentes horloges reliées par des relations affines. Le besoin s'est donc fait sentir d'interpréter ces horloges afin de pouvoir prouver des égalités non syntaxiques lors de la connexion des nœuds. Les types d'horloges ck_1 et ck_2 de la figure 10.1 sont en relation affine de paramètres (ℓ_1, d_2, ℓ_2) (noté $ck_1 \ \mathcal{R}_{(\ell_1,d_2,\ell_2)} \ ck_2$).[1] Exprimé dans nos termes, cela revient à dire que $ck_1 \ \mathcal{R}_{(\ell_1,d_2,\ell_2)} \ ck_2$ si :

$$\text{Il existe un } \alpha \text{ tel que } ck_1 = \alpha \texttt{ on (10}^{\ell_1-1}\texttt{)} \text{ et } ck_2 = \alpha \texttt{ on 0}^{d_2}\texttt{(10}^{\ell_2-1}\texttt{)}$$

[1] Les lettres utilisées traditionnellement [Sma98] sont (n, φ, d).

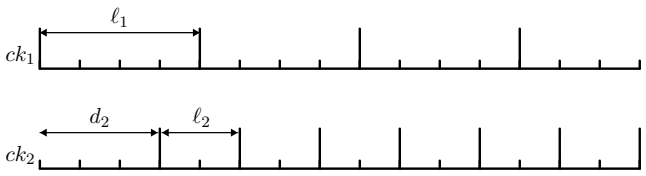


FIG. 10.1 – ck_1 $\mathcal{R}_{(\ell_1,d_2,\ell_2)}$ ck_2. Un type d'horloges ck_1 est en relation affine de paramètres (ℓ_1, d_2, ℓ_2) avec ck_2 si l'on peut obtenir ck_2 en insérant $\ell_1 - 1$ instants entre les `1` des horloges représentées par ck_1, et en positionnant sur l'ensemble d'instants obtenus un `1` tous les ℓ_2 instants, après un délai de d_2 instants.

En particulier, ck_1 $\mathcal{R}_{(1,d_2,\ell_2)}$ ck_2 signifie que $ck_2 = ck_1$ `on` $\texttt{0}^{d_2}\texttt{(10}^{\ell_2-1}\texttt{)}$.

Un ensemble d'une dizaine de règles est donné dans [Sma98], pour établir que deux types d'horloges affines sont égaux ou peuvent être rendus égaux. Ces règles sont présentées en annexe I, en donnant pour chacune d'entre elle sa traduction dans notre cadre. On constate que notre calcul d'horloge de la partie II spécialisé aux horloges affines donne la même réponse que celui de [Sma98]. Pour cela, nous utilisons les opérations, tests et algorithmes spécialisés sur les horloges affines, définis dans la section 4.5 et tout au long du chapitre 5.

10.1.2 Horloges affines et contraintes de temps réel en Lustre

Les horloges affines ont aussi été utilisées comme une extension au langage Lustre pour permettre la spécification d'applications temps-réel distribuées [Cur05]. Comme en Signal, cette extension permet de pouvoir tester sémantiquement l'égalité des horloges. Mais les horloges affines sont dans ces travaux aussi mises à profit à d'autres fins. Leur caractère régulier et suffisamment simple pour être interprété est utilisé pour ordonnancer finement les calculs des nœuds sur les différentes ressources de calcul disponibles. Par exemple, deux tâches nécessitant chacune toute la puissance de calcul peuvent utiliser la même ressource si elles sont exécutées sur des horloges complémentaires.

Nous ne nous sommes pas penchés sur ces questions de répartition des tâches temps réel, mais nous pensons que les travaux de [Cur05] pourraient être combinés aux nôtres. En effet, ils portent tous deux sur un langage à la Lustre et reposent sur des idées similaires. Notamment, la formule permettant de calculer l'horloge affine résultat d'échantillonnages successifs par des horloges affines est la même que la formule présentée dans la section 4.5 de ce mémoire (page 90). Elle correspond aussi à la formule proposée dans [Sma98] pour le langage Signal.

10.1.3 Horloges affines et communications par buffer

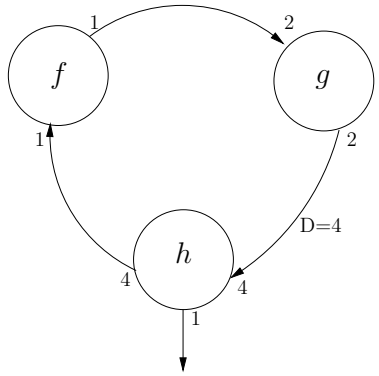
La communication par buffers n'ayant pas été étudiée par le passé dans le cadre des langages synchrones, nous n'avons pas de point de comparaison pour la résolution des contraintes de sous-typage sur des horloges affines. Nous avons vu dans les parties II et III que cette résolution est réalisée facilement dans ce cas particulier. Nous comptons intégrer les horloges affines dans Lucy-n, sous la forme d'un nouveau langage d'horloges. Notons qu'il est déjà possible d'utiliser des horloges affines pour l'échantillonnage des flots puisqu'elles sont un cas particulier de mots ultimement périodiques. Cependant, dans les langages d'horloges exis-

tant, les rythmes inférés pour les nœuds ne sont pas des horloges affines. Les résultats de complétude propres au domaine des horloges affines ne s'appliquent donc pas pour l'instant.

10.2 Programmation de réseaux de Kahn à mémoire bornée

10.2.1 Modèles de réseaux de processus

Le modèle des *Synchronous Dataflow* (SDF) [LM87a, LM87b] a été créé pour la modélisation d'applications de traitement du signal. Ce modèle est un cas particulier des réseaux de Kahn où le nombre de données produites et consommées à chaque exécution des nœuds est connu statiquement. Par exemple ci-dessous, le nœud h consomme à chaque activation quatre valeurs sur son canal d'entrée, et produit respectivement quatre et une valeurs sur ses canaux de sortie.



L'information du nombre de données produites et consommées par les nœuds permet de calculer statiquement un ordonnancement périodique. On peut vérifier l'absence de blocage et calculer les tailles de buffers nécessaires pour cet ordonnancement. Le nombre d'activations de chaque nœud durant un cycle de l'ordonnancement est contraint par des équations, appelées équations d'équilibre (ou *balance equations*). Ces équations garantissent que le nombre d'activations d'un nœud producteur multiplié par le nombre de valeurs produites à chaque activation est égal au nombre d'activations du nœud consommateur multiplié par le nombre de valeurs consommées.

Dans l'exemple précédent, si n_f, n_g et n_h représentent respectivement le nombre d'activations des différents nœuds, ces valeurs sont soumises aux équations d'équilibre suivantes :

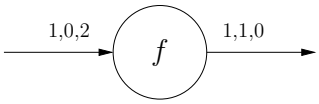
$$\begin{aligned} n_f \times 1 &= n_g \times 2 \\ n_g \times 2 &= n_h \times 4 \\ n_h \times 4 &= n_f \times 1 \end{aligned}$$

Ainsi, à chaque cycle de l'ordonnancement, le nombre de valeurs produites sur un canal est égal au nombre de valeurs consommées sur ce canal. Si les équations d'équilibre ont une solution, celle-ci fournit le nombre d'activations de chaque nœud durant un cycle d'ordonnancement, et le réseau est exécutable avec des buffers bornés.

Un ordonnancement périodique des nœuds ainsi que la taille des buffers peuvent ensuite être calculés par simulation. Pour cela, on active les nœuds dès que possible. Un nœud peut être activé s'il y a suffisamment de données sur ses canaux d'entrée, et si son nombre d'activations en un cycle n'est pas encore atteint. L'ordonnancement n'existe pas en cas d'incohérence

dans les contraintes d'ordre d'exécution des nœuds. La simulation aboutit dans ce cas à un blocage avant d'avoir activé chaque nœud autant de fois que prévu par la solution aux équations d'équilibre.

Les Cyclo-Static Dataflow (CSDF) [BELP96, PPL95] sont une généralisation des SDF où le nombre de valeurs produites et consommées par chaque nœud à chaque activation peut varier selon un motif périodique sans préfixe. Par exemple, le nœud ci-dessous consomme cycliquement 1 puis 0 puis 2 valeurs et produit 1 puis 1 puis 0 valeurs.



Là encore, un ordonnancement périodique sans préfixe peut être calculé statiquement. Pour cela, il faut tout d'abord calculer la taille du motif périodique représentant l'ordonnancement. Cette taille doit être un multiple des tailles des motifs de consommation et production des nœuds. Ensuite, le nombre d'activations de chaque nœud est calculé en résolvant des équations d'équilibre, suivant le même principe que pour les SDF : le nombre de valeurs produites sur un canal durant un cycle de l'ordonnancement doit être égal au nombre de valeurs consommées. Enfin, un ordonnancement est calculé par simulation, de la même manière que pour les SDF.

Par rapport aux SDF, les CSDF permettent une granularité plus fine dans l'expression du nombre de valeurs consommées et produites. Cela permet de mettre en place des ordonnancements nécessitant moins de buffers et de valeurs d'initialisation sur les cycles.

Dans le cas où le nombre de valeurs consommées ou produites à chaque activation d'un nœud est 0 ou 1, les CSDF correspondent à des programmes dans notre langage. Les motifs de consommation et production sont traduits par les types d'horloges des nœuds et l'ordonnancement du réseau correspond au typage du programme. La résolution concrète des contraintes de sous-typage du chapitre 5, bien que présentée différemment, est similaire à la méthode utilisée dans les CSDF. La mise en forme ajustée du système de contraintes est à rapprocher du calcul de la taille du motif périodique d'ordonnancement et de la résolution des équations d'équilibre. La résolution du graphe de contraintes sur les indices des `1` avec recherche de solutions au plus tôt est quant à elle à rapprocher du calcul de l'ordonnancement par simulation avec activation au plus tôt.

Rappelons que dans le cas où les horloges sont sans préfixe comme pour les CSDF, nous pensons que notre algorithme de résolution concrète est correct et complet. Nous pensons que les types que nous inférons dans ces cas sont comparables aux ordonnancements calculés pour les CSDF correspondants.

On peut noter qu'il existe des travaux sur l'approximation des rythmes dans les CSDF, qui semblent proches des nôtres [WBS07]. Nous n'avons pas encore fait d'étude comparative avec ces travaux récents.

10.2.2 Implémentations

Plusieurs langages et plateformes logicielles sont basés sur les modèles décris précédemment.

La plateforme logicielle Ptolemy **[BHLM94]** est un environnement de programmation et de prototypage. Les nœuds sont programmés dans un langage hôte. Les branchements du

réseau sont quant à eux programmés dans un langage de coordination, suivant un des modèles de calcul disponibles. Ces modèles, appelés *domaines*, définissent la manière dont les nœuds interagissent. Les SDF constituent un domaine, pour lequel la génération de code est de plus proposée [PHLB93]. Le nombre de données consommées et produites par chaque nœud est déclaré par l'utilisateur. Si ces valeurs ne sont pas correctes, une erreur se produit à l'exécution. Notre approche est donc différente. En effet, elle est basée sur l'utilisation d'un langage de programmation unique, et le nombre de données consommées et produites est calculé statiquement par typage (dans le cas du langage d'horloges périodiques).

Le langage StreamIt **[TKA02, KTA03]** implémente lui aussi le modèle des SDF, mais en suivant une approche plus proche de la notre, basée sur un langage de programmation unique. Dans ce langage, les nœuds ont une seule entrée et une seule sortie et doivent être connectés avec des combinateurs prédéfinis, qui sont les suivants :

- Pipeline : pour connecter plusieurs nœuds à la suite ;
- SplitJoin : pour séparer un flot en deux flots, auxquels on applique un nœud chacun, avant de les rassembler. La manière dont les données sont séparées entre les deux flots doit être choisie parmi deux possibilités : dupliquer les données sur les deux branches, ou les répartir selon un motif à fournir ;
- FeedBackLoop : pour créer un cycle dans le réseau, en fournissant des valeurs d'initialisation.

La connexion d'un ensemble de sous-nœuds produit donc toujours un nœud à une entrée et une sortie. Comme le langage restreint la forme du réseau et la communication entre les nœuds, des optimisations peuvent être effectuées sur le code généré. Ici encore, si les taux déclarés ne correspondent pas au taux réel du nœud, une erreur se produit à l'exécution.

Notons que les trois manières de connecter les nœuds en StreamIt sont utilisables en Lucy-n, par composition des nœuds, échantillonnage sur des rythmes complémentaires puis fusion (opérateurs `when/whenot`, `merge`) et insertion de délais initialisés (opérateur `fby`). Le langage Lucy-n offre cependant une plus grande liberté dans la forme du réseau.

10.3 Approximation de rythmes

La méthode consistant à encadrer les rythmes en bornant la fonction de cumul est aussi utilisée dans le domaine des réseaux de communication. Elle a été introduite par le *calcul réseau* (ou *Network Calculus*) [Cru91], pour obtenir des bornes sur les délais et les besoins de bufferisation dans les réseaux. Elle utilise une notion de *courbe d'arrivée* qui limite le nombre de valeurs pouvant arriver par rafales sur un flot. La courbe associe à chaque entier Δt le nombre maximum de valeurs qui peuvent arriver en Δt instants. Cette contrainte est nommée *burstiness constraint*. Cette méthode fait aussi intervenir des *courbes de service*, qui suivent le même principe que les courbes d'arrivées, mais pour décrire le nombre de valeurs pouvant être traitées par un nœud du réseau dans un intervalle de temps donné. La courbe décrivant les flots de sortie d'un nœud est calculée par convolution entre la courbe d'arrivée et la courbe de service. Le calcul réseau a été décrit en détail dans [BT01].

Les idées du calcul réseau ont été étendues dans la théorie du *calcul temps-réel* (*Real-time Calculus*) [TCN00], utilisée dans le domaine des systèmes embarqués distribués. Le calcul temps-réel utilise à la fois des courbes d'arrivée supérieures et inférieures pour analyser les performances et les besoins en buffers de systèmes temps réels distribués sur plusieurs proces-

seurs. Ainsi, les rythmes d'arrivée des données reçoivent des bornes supérieure et inférieure plutôt que seulement une borne supérieure. Il en est de même pour les courbes de service.

Tout comme dans nos travaux, la taille des buffers nécessaire est bornée par la plus grande distance verticale entre la courbe d'arrivée et la courbe de service. Le délai introduit est borné par la plus grande distance horizontale entre ces deux courbes. Cependant, les méthodes d'abstraction sont différentes, bien que toutes deux basées sur l'encadrement des fonctions de cumul. Dans le calcul temps-réel, l'encadrement est fait sur des fenêtres glissantes. Cela signifie que chaque taille d'intervalle de temps est associée à un nombre maximal et un nombre minimal d'arrivées de données. À l'inverse, nous donnons des bornes absolues. Autrement dit, nous considérons toujours des intervalles de temps commençant au début de l'exécution. Cela produit des valeurs abstraites ne contenant pas les mêmes ensembles de rythmes, et difficilement comparables. Dans le calcul temps-réel, toutes les permutations d'un rythme périodique sont encadrées par les mêmes courbes. Ce n'est pas le cas dans nos travaux. Nos enveloppes permettent de garder des informations sur un comportement initial différent de la suite de l'exécution, ce qui n'est pas possible si l'on considère des fenêtres glissantes.

Le principal apport de notre théorie des horloges abstraites par rapport au calcul temps-réel est la possibilité de décrire des nœuds de calcul qui ne conservent pas la longueur des flots. Le calcul de l'opérateur `on` est au cœur de l'algèbre des horloges abstraites, et permet d'obtenir les rythmes de sorties de tels nœuds. Une tentative d'extension du calcul temps-réel pour pouvoir modéliser des nœuds qui échantillonnent a été proposée dans [WT05]. Cependant, elle sort du cadre des courbes d'arrivée et de service. Des automates décrivent les types de données qui peuvent arriver sur les flux, et l'absence de donnée est représentée par un type vide. La théorie proposée est donc peu uniforme.

10.4 Conception insensible à la latence

La taille des fils dans les circuits devenant de plus en plus longue, les données mettent souvent plus d'un cycle d'horloge pour transiter entre les nœuds de calcul. La théorie de la conception insensible à la latence (ou *Latency Insensitive Design*) [CMSV01, CSV02] a été inventée pour répondre à ce problème, dit de *timing closure*, dans la conception des circuits.

L'idée est de concevoir un système sous l'hypothèse synchrone, puis de transformer chaque nœud de calcul en un nœud patient, c'est-à-dire un nœud qui n'est pas sensible à l'instant d'arrivée de ses entrées. Pour cela, on encapsule le nœud dans un processus qui sert d'interface de communication et décide de l'activation du nœud quand toutes les entrées sont disponibles. Si certaines entrées ne sont pas disponibles, la capsule (ou *shell wrapper*) stocke dans des buffers les entrées présentes. On partitionne ensuite les longs fils en segments en plaçant des nœuds "relais", appelés *relay stations*, jouant le rôle de délai unitaire. Ainsi, les données peuvent transiter en un cycle entre chacun des nœuds. En séparant la conception des nœuds de calcul et la communication entre ces nœuds, cette méthodologie permet de concevoir des systèmes sur puce corrects par construction en ce qui concerne l'insensibilité à la latence.

Différents protocoles d'ordonnancement dynamique des nœuds par les capsules ont été proposés [CSV02, CKG06]. Ces protocoles utilisent un mécanisme de rétroaction (ou *back-pressure*) pour informer le nœud producteur que le nœud consommateur n'a plus de place pour recevoir des entrées à mettre en attente. Lorsque qu'un nœud producteur n'est pas exécuté, la capsule doit aussi informer le consommateur sur un canal de contrôle qu'aucune donnée valide ne lui a été envoyée.

Pour éviter le surcoût de communications induit par ces protocoles dynamiques, des méthodes d'ordonnancement statique ont été proposées [Bou07, BdSM07, Mil08]. Elles consistent à ajouter des registres sur certains fils plus courts que les autres, afin que les nœuds reçoivent toutes leurs entrées en même temps, qu'elles viennent "de loin" ou pas. Cette opération s'appelle l'*égalisation*. Parfois, l'égalisation n'est pas possible sans utiliser des *registres fractionnaires*, qui bloquent le passage des données à certains instants mais pas à d'autres. Un ordonnancement des nœuds est ensuite calculé. Il est représenté sous la forme d'une horloge ultimement périodique par nœud, indiquant les instants où le nœud doit être exécuté.[2] Cet ordonnancement est calculé par simulation de l'exécution du réseau jusqu'à l'atteinte d'un comportement périodique, en exécutant les nœuds aussi tôt que possible. L'instant où le comportement périodique est atteint n'est pas calculé avant la simulation : il est détecté en stockant la liste des états (nombre de valeurs dans les différents buffers) déjà atteints au cours de la simulation.

Un réseau ordonnancé statiquement constitue un nœud de calcul, semblable aux nôtres, avec la particularité que les horloges des entrées et sorties d'un même nœud ont toutes la même longueur et le même nombre de `1` (donc le même taux). Ces nœuds sont appelés *SSIP*, pour *Statically Scheduled IP*. La question de la connexion de plusieurs SSIP entre eux a été abordée dans [BM07]. Cependant, bien que des tailles de buffers suffisantes soient calculées statiquement, l'algorithme d'ordonnancement proposé est un algorithme d'ordonnancement dynamique. Les travaux que nous avons présentés dans cette thèse peuvent donc être utilisés de manière complémentaire aux travaux sur l'ordonnancement statique de circuits insensibles à la latence, pour connecter les SSIP obtenus et calculer statiquement des rythmes d'activation adéquats pour chaque nœud. Nous décrirons nos perspectives de recherche sur ces questions dans le chapitre 10.

[2] À chaque activation, un nœud consomme une valeur sur chaque entrée et produit une valeur sur chaque sortie.

Chapitre 11

Perspectives et discussion

Nous avons étudié dans ce mémoire l'expression des horloges par des mots binaires. Nous pensons que les méthodes présentées, en particulier l'abstraction des horloges, peuvent s'adapter à des horloges plus expressives et détaillons comment dans la section 11.1. La résolution des contraintes de sous-typage sur les types d'horloges périodiques mérite encore d'être travaillée. Nous expliquons dans la section 11.2 les pistes que nous aimerions explorer. Dans les sections 11.3 et 11.4, nous montrons que les outils théoriques présentés constituent un cadre uniforme permettant de proposer différentes sortes de buffers au programmeur, comme les buffers de taille limite imposée et les buffers insérant un instant de délai. Dans la section 11.5, nous montrons une utilisation possible des seconds. Enfin, dans la section 11.6, nous rappelons que nous n'avons pas traité la question de la génération de code. Une version naïve pourra facilement être mise en place, mais ce sujet constitue une question importante si l'on s'intéresse à l'optimisation du code généré.

11.1 Un langage, des horloges

11.1.1 Horloges entières

Les horloges entières ont été introduites [CDDP07] pour pouvoir représenter la lecture ou l'écriture de plusieurs valeurs sur un même flot au sein du même instant. Ces horloges peuvent représenter des *salves* d'instants [Gér08] qui rassemblent plusieurs instants successifs en un ensemble d'instants insécable. Les *salves* d'instants définissent des séquences inobservables de calculs et sont donc utiles pour la compilation optimisante avec des boucles et des tableaux, ainsi que vers des architectures réparties.

Les horloges entières sont de la forme suivante : $w ::= n.w$ avec $n \in \mathbb{N}$. Par exemple, un flot d'horloge `31(052)` contiendra trois valeurs au premier instant observable, une valeur au suivant, puis cycliquement zéro puis cinq puis deux valeurs.

Le modèle n-synchrone, décrit dans le chapitre 3 sur des horloges binaires, peut facilement être étendu aux horloges entières, si l'on utilise les expressions des définitions et propositions en fonction de la fonction de cumul. Ainsi, la formule $\forall i \geq 0,\ \mathcal{O}_{w_1 \ \mathtt{on} \ w_2}(i) = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i))$ donnée dans la proposition 3.2 permet de calculer le résultat de l'opération $\mathtt{on}$ sur les horloges entières définie dans [CDDP07, Gér08].✿

En effet, tout comme sur les mots binaires, l'opération $w_1 \ \mathtt{on} \ w_2$ peut être vue comme un

parcourt de w_2 au rythme défini par w_1. Par exemple, considérons l'opération suivante :

w_1	3	1	0	5	2
w_2	2 1 5	4		2 0 4 3 0	5 2
w_1 *on* w_2	8	4	0	9	7

Au premier instant, l'horloge w_1 indique qu'il faut consulter trois valeurs dans w_2. Ces valeurs (2, 1 et 5) sont sommées, pour obtenir en résultat l'élément 8. Au second instant, l'élément 1 de w_1 indique qu'une valeur doit être consultée dans w_2 et l'élément 4 est donc reporté dans le résultat. Au troisième instant, w_1 indique qu'aucune valeur ne doit être consommée dans w_2 et ainsi de suite.

De même, la formule de $\preceq$ donnée dans la proposition 3.7 s'applique directement à ces horloges : $w_1 \preceq w_2 \Leftrightarrow \forall i > 0,\ \mathcal{O}_{w_1}(i) \geq \mathcal{O}_{w_2}(i)$. En effet, même si elles sont produites par paquets, si toujours plus de valeurs ont été écrites dans le buffer que de valeurs ont été lues, alors il n'y a jamais de lecture dans un buffer vide. De même, la formule de la relation de synchronisabilité donnée dans la définition 3.9 s'applique directement aux horloges entières : $w_1 \bowtie w_2 \overset{def}{\Leftrightarrow} \exists b_1, b_2 \in \mathbb{Z},\ \forall i \geq 0,\ b_1 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq b_2$. Même si elles sont produites par paquets, si la différence entre le nombre de valeurs écrites et le nombre de valeurs lues est bornée, alors la taille de buffer nécessaire est bornée.

Grâce à son expression par des opérations et relations sur les fonctions de cumul des mots manipulés, la théorie du modèle n-synchrone s'applique donc facilement aux horloges à valeurs entières. Considérons maintenant sa mise en œuvre, c'est-à-dire les algorithmes de vérification des relations et d'inférence d'horloges satisfaisant les relations, à travers les méthodes présentées dans les parties II et III.

Mots entiers ultimement périodiques

Considérons les mots entiers ultimement périodiques. La principale adaptation à appliquer aux formules du chapitre 4 est d'interpréter le nombre de 1 d'un mot fini ($|v|_1$) comme la somme de ses éléments ($\mathcal{O}_v(|v|)$). Ces valeurs coïncident pour les mots binaires.

L'ensemble des mots entiers périodiques est clos par application de l'opérateur *on*. La taille du préfixe et de la partie périodique du résultat doivent être calculées avec une version modifiée de la formule de la proposition 4.1. En effet, pour le calcul de la taille du préfixe du résultat, il faut trouver pour les opérandes une forme telle que le nombre de 1 du préfixe du premier mot soit égal au nombre d'éléments du préfixe du second mot. Contrairement au cas des horloges binaires, pour mettre les horloges entières sous cette forme, on doit parfois augmenter la taille des deux préfixes, car le nombre de 1 des mots augmente par paliers.

Le test de ralentissement et le calcul des ralentisseurs peuvent être effectués dans même esprit que dans le cas binaire, mais les possibilités offertes par les mots entiers ouvrent de nouvelles perspectives.

La formule de la relation de synchronisabilité reste un test d'égalité des taux. Le test de précédence sur les fonctions de cumul est inchangé, ainsi que le calcul de la taille du buffer.

En ce qui concerne les algorithmes de résolution des contraintes, nous pensons qu'ils peuvent être eux aussi adaptés au cas des mots entiers. Pour les contraintes d'égalité, les différents choix possibles sont ceux listés dans le chapitre 5. Mis à part la relation de ralentissement utilisée, le passage au cas entier ne change rien. Pour les contraintes d'adaptabilité, le choix d'horloges entières pertinentes satisfaisant les contraintes demande réflexion. Nous

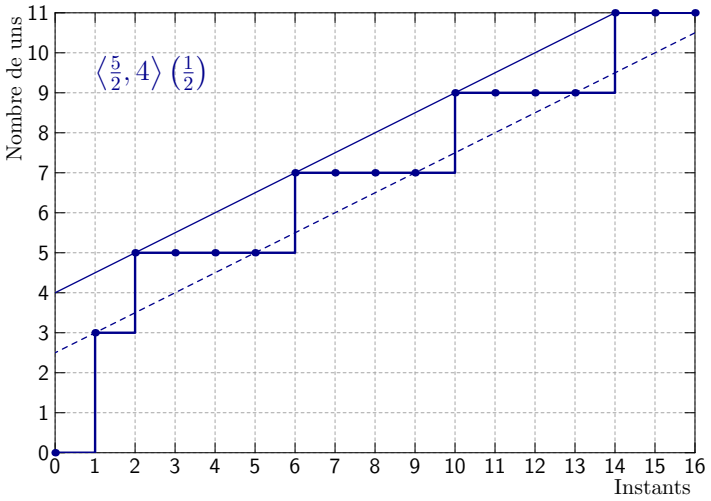


FIG. 11.1 – Mot entier `3(2000)` et son enveloppe $\langle \frac{5}{2}, 4 \rangle (\frac{1}{2})$.

pourrons peut-être nous inspirer des techniques utilisées pour l'ordonnancement des *cyclo-static dataflow* (voir section 10.2) qui sont similaires à des réseaux n-synchrones avec horloges entières sans préfixes.

Mots entiers abstraits

Adaptons la définition de la fonction de concrétisation au cas des mots entiers prenant des valeurs comprises entre `0` et `N` :

$$concr\left(\langle b^{\mathtt{0}}, b^{\mathtt{1}} \rangle (r)\right) = \left\{ w,\ \forall i \geq 1, \quad \wedge \begin{array}{lcl} w[i] > \mathtt{0} & \Rightarrow & \mathcal{O}_w(i) \leq r \times i + b^{\mathtt{1}} \\ w[i] < \mathtt{N} & \Rightarrow & \mathcal{O}_w(i) \geq r \times i + b^{\mathtt{0}} \end{array} \right\} \text{ avec } r \leq \mathtt{N}$$

On peut remarquer que dans le cas des mots binaires, $\mathtt{N} = \mathtt{1}$ et la définition ci-dessus est équivalente à la définition donnée dans le chapitre 7. Le principe reste le même quelle que soit la borne `N`. Les fonctions de cumul doivent monter au maximum quand elles sont sous la droite pointillée $\Delta^{\mathtt{0}}$ afin d'atteindre ce minimum demandé au plus vite. Lorsqu'elles sont au dessus de la droite $\Delta^{\mathtt{1}}$, les fonctions de cumul doivent stagner pour passer sous ce maximum autorisé au plus vite. Lorsqu'elles sont entre les deux droites, les fonctions de cumul peuvent évoluer comme désiré du moment qu'elles restent dans l'enveloppe.

Si l'on ne borne pas les valeurs qui peuvent être prises par le mot entier et que l'enveloppe se trouve au dessus de zéro au premier instant, la fonction de cumul doit prendre au premier instant une valeur suffisante pour rentrer directement dans l'enveloppe.

La figure 11.1 présente la fonction de cumul d'un mot entier et son enveloppe.

Après relecture du chapitre 7 à la lumière de cette nouvelle définition de la fonction de concrétisation, il nous semble que l'ensemble des travaux décrits peuvent être appliqués tels quels au cas des horloges entières. Le point délicat a été de trouver une fonction de concrétisation pour les horloges entières, telle que le test de précédence reste valide. En effet, si l'on n'avait pas imposé que les fonctions de cumul se rapprochent de leur enveloppe au plus vite, le choix de fronts montants plus ou moins hauts pour rejoindre les enveloppes aurait pu mener à des entrelacements.

En ce qui concerne l'algorithme de résolution des contraintes d'adaptabilité décrit dans le chapitre 8, il ne nécessite qu'une légère attention au moment de la résolution des contraintes sur les taux : pour maximiser le rythme, on peut donner la valeur `N` aux taux des horloges non contraintes, au lieu de la valeur 1 donnée dans le cas binaire (qui correspond effectivement à la valeur de `N` dans ce cas). Cependant, un tel choix crée des ensembles d'instants insécables. La question se pose de ne pas introduire de dépendances cycliques entre les valeurs calculées au sein de différents ensembles d'instants.

11.1.2 Horloges énumérées

Les horloges énumérées, introduites dans [BCP07], sont de la forme suivante :

$$w ::= e.w \text{ avec } e \in E \text{ où } E \text{ est un type énuméré}$$

Par exemple, si $E =$ `Class_Ia` | `Class_Ib` | `Class_II`, on peut construire une horloge (`Class_Ia`^{50} `Class_Ib`^{132} `Class_II`^{78}) qui peut être utile pour programmer l'encodeur et le décodeur de canal GSM présentés précédemment. Les nœuds qui séparent et rassemblent les différentes parties d'une trame peuvent être réécrits ainsi :

```
let clock c = (Class_Ia^50 Class_Ib^132 Class_II^78)

let node split_speech x = (x_Ia, x_Ib, x_II) where
  rec x_Ia = x when Class_Ia(c)
  and x_Ib = x when Class_Ib(c)
  and x_II = x when Class_II(c)

let node join_speech (x_Ia, x_Ib, x_II) = x where
  rec x = merge c (Class_Ia -> x_Ia) (Class_Ib -> x_Ib) (Class_II -> x_II)
```

L'horloge `c` définit les instants où la trame contient des bits de classe Ia, Ib, ou II. Dans le nœud `split_speech`, l'échantillonnage `x when Class_Ia(c)` conserve la valeur de `x` quand l'horloge `c` prend la valeur `Class_Ia`. Dans le nœud `join_speech`, la fusion `merge c (Class_Ia -> x_Ia) (...) (...)` prend les valeurs du flot `x_Ia` lorsque l'horloge `c` prend la valeur `Class_Ia`. Les types d'horloges de ces nœuds sont :

$$\begin{array}{rcl} \textit{split\_speech} & : & \forall\alpha.\ \alpha \to (\alpha \texttt{ on Class\_Ia}(c) \times \alpha \texttt{ on Class\_Ib}(c) \times \alpha \texttt{ on Class\_II}(c)) \\ \textit{join\_speech} & : & \forall\alpha.\ (\alpha \texttt{ on Class\_Ia}(c) \times \alpha \texttt{ on Class\_Ib}(c) \times \alpha \texttt{ on Class\_II}(c)) \to \alpha \end{array}$$

Dans ce cadre, il faut redéfinir la fonction de cumul d'une horloge de la manière suivante :

$$\begin{array}{rcl} \mathcal{O}_{\texttt{Class\_Ia}(w)}(0) & = & 0 \\ \mathcal{O}_{\texttt{Class\_Ia}(e.w)}(i) & = & \begin{cases} 1 + \mathcal{O}_{\texttt{Class\_Ia}(w)}(i-1) & \text{si } e = \texttt{Class\_Ia} \\ \mathcal{O}_{\texttt{Class\_Ia}(w)}(i-1) & \text{sinon} \end{cases} \end{array}$$

Les horloges binaires sont un cas particulier des horloges énumérées sur le type $B = \texttt{1} \mid \texttt{0}$. Ici encore, il nous semble qu'il suffit de considérer les définitions et propositions en fonction des fonctions de cumul pour utiliser les méthodes présentées sur ces nouvelles horloges. En ce qui concerne l'inférence de rythmes nous pensons qu'une première étape serait de n'inférer que des horloges du type B ci-dessus.

Pour le moment Lucy-n est un prototype qui permet d'expérimenter des calculs d'horloge pour la programmation dans le modèle n-synchrone. Cet aspect expérimental se reflète dans

son implémentation sous forme de foncteur. Dans ce mémoire, nous avons présenté un langage d'horloges périodiques et un langage d'horloges abstraites pour Lucy-n. L'architecture choisie nous permettra de mettre en place facilement d'autres langages d'horloges et systèmes de résolution associés.

11.2 Résolution des contraintes sur les mots périodiques

11.2.1 Résolution correcte et complète

Comme nous l'avons dit dans la partie II, l'algorithme de résolution de contraintes sur les mots périodiques n'est pas encore satisfaisant. Nous avons donc l'intention d'y travailler encore, sur deux points de vue. D'une part, du point de vue théorique, en trouvant comment traiter tous les systèmes, y compris ceux que l'on ne sait pas mettre en forme ajustée pour l'instant. Cela nous permettrait de disposer d'un algorithme correct et complet de référence. Même s'il ne passe pas à l'échelle, cet algorithme serait utile pour servir de point de comparaison pour des algorithmes utilisant des approximations comme celui de la partie III. D'autre part, du point de vue pratique, nous souhaitons optimiser l'implémentation de l'algorithme. On peut penser à l'utilisation de structures de données plus efficaces comme les *bit vectors* pour représenter les horloges, actuellement représentées par des paquets de `0` et de `1`. On peut aussi penser à une résolution plus efficace de nos systèmes d'inéquations linéaires (comme celui représenté sur la figure 5.1) en prenant partie de leur forme particulière. Une première optimisation par compression du graphe a déjà été implémentée.[1]

11.2.2 Résolution correcte

On peut imaginer d'autres algorithmes de résolution des contraintes d'adaptabilité sur les mots périodiques qui soient corrects mais pas complets, c'est-à-dire qui ne trouvent pas forcément les solutions si elles existent. Par exemple, quand le langage d'horloges utilisé est le langage d'horloges périodiques, on peut combiner la résolution concrète de la partie II avec la résolution abstraite de la partie III. Pour la phase de simplification du système, nous préconisons l'utilisation de l'algorithme de la partie II. Ainsi, on vérifie que les contraintes sur deux types exprimés en fonction de la même variable sont satisfaites et on calcule les mots périodiques résultats des expressions d'horloges en utilisant toutes les informations disponibles. Cette étape de l'algorithme de résolution concrète est correcte et complète, et elle n'est pas trop coûteuse car le traitement n'est pas global au système : les contraintes sont traitées une à une. Pour la résolution du système simplifié, nous préconisons l'utilisation de l'algorithme de la partie III, car c'est cette étape qui est trop coûteuse et ne traite pas tous les systèmes dans la résolution concrète. Un fois le système résolu, on peut repasser facilement dans le monde des horloges périodiques en choisissant n'importe quelle horloge périodique des enveloppes inférées. Le calcul de la taille des buffers peut alors être réalisé avec l'opération concrète, car ce calcul est effectué contrainte par contrainte, donc n'implique pas d'explosion combinatoire, et il donne des résultats plus précis que son analogue abstrait. Grâce aux différentes options disponibles dans notre typeur, nous pouvons utiliser cette combinaison des deux méthodes de résolution. Nous l'avons testée sur l'exemple du *Picture in Picture*, et les résultats sont satisfaisants : le temps de calcul est comparable à celui de la résolution

[1] Ces deux idées nous ont été proposées par Jean-Christophe Filliâtre qui a implémenté le calcul de la taille du buffer sur les *bit vectors*.

abstraite, le type inféré est le même que celui de la résolution abstraite, mais la taille du buffer calculée est de 192 240, au lieu de 193 079 pour la méthode abstraite et 191 970 pour la méthode concrète. L'application de cette méthode à l'encodeur GSM calcule des buffers dont la taille correspond exactement à la taille des sous-trames.

Nous aimerions aussi étudier la question d'un autre algorithme de résolution sur les horloges périodiques, correct mais pas complet. Celui-ci n'utiliserait pas la méthode d'abstraction elle même, mais suivrait le même principe : ralentir les horloges au minimum, juste pour les ramener au même taux, puis satisfaire les contraintes de précédence en procédant par décalages, c'est-à-dire en ajoutant des délais au début des horloges.

Étudions cette idée plus en détail. Après simplification, les systèmes à résoudre sont de la forme : $C = \{\alpha_{x_i} \texttt{ on } p_{x_i} <: \alpha_{y_i} \texttt{ on } p_{y_i}\}_i$ avec $\alpha_{x_i} \neq \alpha_{y_i}$ pour tout i. Comme dans les autres méthodes présentées, pour tout i, on pose $\alpha_{x_i} = \alpha \texttt{ on } c_{x_i}$ et $\alpha_{y_i} = \alpha \texttt{ on } c_{y_i}$ pour se ramener à des contraintes d'adaptabilité sur les mots :

$$A = \{c_{x_i} \textit{ on } p_{x_i} <: c_{y_i} \textit{ on } p_{y_i}\}_i$$

On cherche des taux pour les c_{x_i} et les c_{y_i} tels que les contraintes de synchronisabilité sont vérifiées, en utilisant la méthode présentée dans la partie III. Si r_{x_i} sont les taux inférés pour les c_{x_i} et r_{y_i} sont ceux des c_{y_i}, on peut choisir l'instanciation suivante :

$$c_{x_i} = 0^{d_{x_i}}\texttt{(1) } \textit{on } p'_{x_i}$$

avec p'_{x_i} un mot équilibré de taux r_{x_i} et d_{x_i} une valeur entière à trouver. On fait de même pour les c_{y_i}.

Le système obtenu est le suivant, il vérifie les contraintes de synchronisabilité.

$$A = \left\{0^{d_{x_i}}\texttt{(1) } \textit{on } p'_{x_i} \textit{ on } p_{x_i} <: 0^{d_{y_i}}\texttt{(1) } \textit{on } p'_{y_i} \textit{ on } p_{y_i}\right\}_i$$

Pour vérifier les contraintes des précédence, les d_{x_i} et d_{y_i} doivent être tels que

$$D = \left\{d_{x_i} + \textit{compute\_delay}(p'_{x_i} \textit{ on } p_{x_i},\ p'_{y_i} \textit{ on } p_{y_i}) \leq d_{y_i}\right\}_i$$

L'opération $\textit{compute\_delay}(p'_{x_i} \textit{ on } p_{x_i},\ p'_{y_i} \textit{ on } p_{y_i})$, définie dans le chapitre 3, calcule le plus petit délai d à appliquer à $p'_{y_i} \textit{ on } p_{y_i}$ pour que $p'_{x_i} \textit{ on } p_{x_i} \preceq 0^d.(p'_{y_i} \textit{ on } p_{y_i})$. Ce délai peut être calculé car à cette étape de la résolution, les p'_{x_i}, p_{x_i}, p'_{y_i} et p_{y_i} sont connus. Si $d_{x_i} + d \leq d_{y_i}$, on a $0^{d_{x_i}}\texttt{(1) } \textit{on } p'_{x_i} \textit{ on } p_{x_i} \preceq 0^{d_{y_i}}\texttt{(1) } \textit{on } p'_{y_i} \textit{ on } p_{y_i}$. On peut donc choisir pour les d_{x_i} et d_{y_i} n'importe quelle solution au système d'inéquations linéaires D.

11.3 Contraindre la taille des buffers

Une extension aisée de l'opérateur `buffer` serait de donner la possibilité au programmeur d'indiquer des limites de taille pour les buffers qu'il utilise. Pour cela on peut introduire un opérateur `buffer_n`, qui prend comme premier argument la taille maximale désirée. Par exemple, `buffer_n(3, x)` signifie que l'on désire stocker le flot `x` dans un buffer d'une taille maximale de `3`. Pour typer les programmes utilisant cette construction, il faut étendre la relation de sous-typage afin de pouvoir exprimer l'information de taille du buffer. On définit que $ck_1 <:^n ck_2$ si tout flot de type ck_1 peut être utilisé sur le rythme défini par ck_2 en

étant stocké dans un buffer de taille inférieure ou égale à n. Ainsi, si `x` est de type ck_1, `buffer_n(3, x)` peut recevoir tout type ck_2 tel que $ck_1 <:^3 ck_2$. On appellera "sous-typage limité" cette nouvelle relation. La relation d'adaptabilité associée au sous-typage limité est définie par :

$$w_1 <:^n w_2 \overset{def}{\Leftrightarrow} w_1 \bowtie w_2 \ \wedge\ w_1 \preceq w_2 \ \wedge\ size(w_1, w_2) \leq n$$

Rappelons que $size(w_1, w_2) = \max_{i \in \mathbb{N}^*}(\mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i))$. Donc :

$$size(w_1, w_2) \leq n \overset{def}{\Leftrightarrow} \forall i \geq 0,\ \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq n$$

Pour pouvoir intégrer cette nouvelle contrainte dans les systèmes de la partie II, il suffit de remarquer que $\mathcal{O}_{(w_1 \texttt{ on } 0^n\texttt{(1)})}(i) = \max(0, \mathcal{O}_w(i) - n)$. On a donc :

$$size(w_1, w_2) \leq n \Leftrightarrow \mathcal{O}_{w_1 \texttt{ on } 0^n\texttt{(1)}}(i) \leq \mathcal{O}_{w_2}(i) \Leftrightarrow w_2 \preceq w_1 \texttt{ on } 0^n\texttt{(1)}$$

La contrainte de taille limitée s'exprime donc comme une contrainte de précédence supplémentaire.

L'algorithme de la partie III s'adapte lui aussi facilement aux contraintes de sous-typage limité. Au lieu de seulement imposer que les enveloppes ne doivent pas être trop rapprochées, on impose aussi qu'elles ne doivent pas être trop éloignées :

$$\begin{aligned}
\left\langle {b^0}_1, {b^1}_1 \right\rangle (r_1) <:^{n\sim} \left\langle {b^0}_2, {b^1}_2 \right\rangle (r_2) \quad &\overset{def}{\Leftrightarrow} \quad \left\langle {b^0}_1, {b^1}_1 \right\rangle (r_1) \bowtie^{\sim} \left\langle {b^0}_2, {b^1}_2 \right\rangle (r_2) \\
&\qquad \wedge\ \left\langle {b^0}_1, {b^1}_1 \right\rangle (r_1) \preceq^{\sim} \left\langle {b^0}_2, {b^1}_2 \right\rangle (r_2) \\
&\qquad\quad \wedge\ size^{\sim}(\left\langle {b^0}_1, {b^1}_1 \right\rangle (r_1), \left\langle {b^0}_2, {b^1}_2 \right\rangle (r_2)) \leq n \\
&\Leftrightarrow \quad (r_1 = r_2) \ \wedge\ ({b^1}_2 - {b^0}_1 < 1) \ \wedge\ (\lfloor {b^1}_1 - {b^0}_2 \rfloor \leq n) \\
&\Leftrightarrow \quad (r_1 = r_2) \ \wedge\ ({b^1}_2 - {b^0}_1 < 1) \ \wedge\ ({b^1}_1 - {b^0}_2 < n + 1)
\end{aligned}$$

Les contraintes de taille maximale pour les buffers étant de la même forme que les contraintes de précédence, elles peuvent aisément être exprimées comme des contraintes d'inéquations linéaires supplémentaires à traiter simultanément à celles issues des contraintes de précédence. C'est ce point qui rend l'ajout de l'opérateur de bufferisation avec limite imposée satisfaisant. Les contraintes de sous-typage limité sont collectées avec les contraintes de sous-typage, et l'ensemble obtenu est résolu de manière uniforme.

11.4 Analyse de causalité

L'analyse de causalité du langage Lucid Synchrone peut être appliquée aux programmes Lucy-n. Cette analyse vérifie que la valeur courante de chaque flot ne dépend que des valeurs précédentes du flot. Pour cela, elle vérifie que la définition du flot n'utilise sa propre valeur qu'à travers un délai initialisé introduit par l'opérateur `fby`. Cependant, nous avons vu page 30 que l'on peut encoder l'opérateur `fby` à l'aide d'un buffer. Ainsi le nœud `sum` de la page 26 peut être réécrit de la manière suivante :

```
let clock one = 1(0)

let node sum x = o where
  rec o = x + zero_fby_o
  and zero_fby_o = merge one 0 (buffer(o))
```

Bien que calculable de manière séquentielle, ce nœud est rejeté par l'analyse de causalité car il n'utilise pas l'opérateur `fby` pour construire son accumulateur.

Nous aimerions donc étudier une analyse de causalité tenant compte de l'utilisation de buffers initialisés. Pour cela, il faudrait introduire un nouvel opérateur `buffer` (noté `strict_buffer`), qui n'autorise pas une donnée entrante à ressortir du buffer au sein du même instant. Ainsi, si tous les cycles du système traversent un tel buffer, on sait que le réseau de nœuds ne peut pas contenir de boucle de causalité. Comme dans la section 11.3, on peut mettre en œuvre cette idée en adaptant la relation de sous-typage. Ici, il s'agit de rendre "stricte" cette relation d'ordre. Si `x` est de type ck_1, `strict_buffer(x)` peut recevoir tout type ck_2 tel que $ck_1 \ll: ck_2$. La relation d'adaptabilité associée à cette relation de "sous-typage strict" est définie par :

$$w_1 \ll: w_2 \overset{def}{\Leftrightarrow} w_1 \bowtie w_2 \wedge w_1 \prec w_2$$
$$\text{avec } w_1 \prec w_2 \overset{def}{\Leftrightarrow} \forall j \geq 1,\ \mathcal{I}_{w_1}(j) < \mathcal{I}_{w_2}(j)$$

Il nous a suffit ici de modifier la relation de précédence en une relation de précédence stricte, c'est-à-dire assurant que l'indice du $j^{ième}$ élément `1` dans w_1, qui donne l'instant d'arrivée de la $j^{ième}$ valeur dans le buffer, est strictement inférieur à l'indice du $j^{ième}$ élément `1` dans w_2, qui donne l'instant de sortie de cette dernière.

La relation de sous-typage strict peut être utilisée dans l'un ou l'autre des langages d'horloges que nous avons présentés. Dans le cadre des horloges périodiques avec résolution concrète, il s'agit juste de traduire les contraintes de précédence stricte par des inéquations linéaires strictes sur les indices des `1`. Dans le graphe page 106, le passage à des inéquations strictes revient juste à donner la valeur 1 (au lieu de 0) aux labels des transitions reliant les deux mots. La résolution du système d'inéquations peut donc être réalisé comme précédemment. Dans le cadre des horloges abstraites, la relation de précédence stricte est impliquée par :

$$\left\langle {b^0}_1, {b^1}_1 \right\rangle (r) \prec^{\sim} \left\langle {b^0}_2, {b^1}_2 \right\rangle (r) \quad \Leftarrow \quad {b^1}_2 < {b^0}_1 \ \wedge\ {b^1}_2 < 1 - r$$

En effet, si les enveloppes ne se chevauchent pas (condition ${b^1}_2 < {b^0}_1$), l'occurrence du $j^{ième}$ front montant se produit toujours plus tôt dans les mots de la première enveloppe que dans les mots de la seconde. Cette affirmation est invalidée lorsque les deux enveloppes contiennent des mots commençant par une série de `1`. La condition ${b^1}_2 < 1 - r$ garantit que la deuxième enveloppe ne contient pas de tels mots. Ce test est donc correct. Cependant, il n'est pas le plus précis (on pourrait autoriser un petit chevauchement dans certains cas). Les contraintes de précédence stricte abstraite sont de la même forme que les contraintes de précédence abstraite. La résolution du système de contraintes peut donc là encore être réalisée de la même manière que précédemment.

L'avantage d'insérer des délais initialisés en utilisant un opérateur de bufferisation strict est de pouvoir donner plus d'une valeur d'initialisation. Ainsi si la chaîne de traitements à appliquer à un flot insère un délai de plusieurs instants, on peut utiliser en entrée du système les valeurs produites, en plaçant sur la boucle de rétroaction plusieurs valeurs d'initialisation. Il est possible d'obtenir le même résultat en plaçant plusieurs nœuds `fby`, mais cet opérateur ayant le type $\forall \alpha.\, \alpha \to \alpha$, le placement d'un buffer peut tout de même être nécessaire pour adapter le rythme des sorties au rythme où l'on désire les utiliser en entrée. Le programme devient alors peu lisible.

Toute l'élégance de la méthode que nous proposons ici réside dans le fait que les buffers stricts cohabitent sans peine avec les autres buffers. Les deux genres de contraintes de sous-typage sont collectées dans un même ensemble et résolues dans un cadre unique.

11.5 Ordonnancement de circuits insensibles à la latence

Notons que si l'on dispose de l'opérateur `strict_buffer` décrit dans la section précédente, on peut modéliser les circuits insensibles à la latence dans notre langage. En effet chaque nœud de calcul du circuit est un nœud du type $\forall\alpha.\ \alpha \to \alpha$, et les temps de transport peuvent être modélisés en plaçant un buffer strict pour jouer le rôle de station relais.

Remarque 11.1. Après lecture de la section 9.3, on pourrait penser utiliser un nœud de type $\forall\alpha.\ \alpha \to \alpha$ `on 0(1)` pour la modélisation d'un temps de transport :

```
let transport (x::'a) = (buffer x)::'a on 0(1)

transport :: forall 'a. 'a -> 'a on 0(1)
```

Cependant, l'utilisation d'un tel nœud n'aurait pas l'effet désiré ici. En effet, ce nœud de transport ne libère la valeur qu'il contient qu'à l'arrivée de la donnée suivante. Ce comportement ne correspond pas forcément à l'insertion d'un délai d'un seul instant comme on veut le faire dans les circuits considérés. ◇

Une fois les buffers stricts placés pour modéliser le temps de transport, il suffit de placer un buffer sur chaque fil pour conserver les données qui arrivent trop vite en entrée d'un nœud. S'il trouve une solution, l'algorithme de résolution des contraintes fournit des tailles suffisantes pour les buffers et l'instanciation des variables de type des nœuds de calcul et de transport fournit un ordonnancement valide du réseau. Le SSIP est donc simplement construit par composition des nœuds de calcul et de transport du circuit, et son ordonnancement statique est obtenu par typage. Cependant, nos algorithmes de résolution échouent dans ces cas particuliers. En effet, les systèmes de contraintes de ces circuits contiennent des horloges avec préfixe. Ces horloges proviennent du type des flots produisant seulement les valeurs d'initialisation pour les cycles, qui sont de la forme α `on` 1^d`(0)` avec d le nombre de valeurs d'initialisation. Notre algorithme de résolution concrète échoue donc à mettre les systèmes en forme ajustée et ne peut par conséquent pas les traiter. En ce qui concerne la résolution abstraite, elle recherche une solution sur le rythme du nœud le plus lent. Comme tous les nœuds sont de type $\forall\alpha.\ \alpha \to \alpha$, l'instanciation recherchée ne ralenti aucun nœud, et il n'existe généralement pas de telle solution en présence de cycles.

Cette application possible de nos travaux constitue une des motivations à continuer à travailler sur l'algorithme de résolution concrète. Pour cela, nous pensons nous inspirer des techniques décrites dans [Bou07, BdSM07, Mil08]. Cependant, le problème que nous cherchons à résoudre est plus général, puisque les nœuds que nous composons dans nos programmes ne préservent pas la longueur. Une fois notre problème résolu, nous aimerions tenter d'utiliser les résultats de [Mil08] pour calculer des ordonnancements réguliers les plus rapides possibles, ce qui n'est pour l'instant pas le cas dans l'algorithme de résolution concrète, comme le montre l'exemple du *Picture in Picture* 6.4.

11.6 Génération de code

Le générateur de code pour Lucy-n n'a pas encore été implémenté. Grâce aux informations de typage, la traduction des programmes n-synchrones en des programmes purement synchrones sera aisée. Après l'étape du calcul d'horloge, les buffers pourront être implémentés comme des nœuds synchrones qui sont paramétrés par leur taille et leurs horloges. Néanmoins, dans le cas des horloges périodiques comme dans celui des horloges abstraites, on dispose de plus d'informations à la compilation que dans le cas général des horloges non interprétées. Il serait intéressant d'utiliser ces informations pour générer du code optimisé.

Enfin, soulignons que la compilation de l'opérateur `cond` mérite une attention particulière. Cet opérateur, implicite dans la syntaxe du langage, est explicité dans le langage traité par la sémantique n-synchrone. Il transforme les horloges utilisées comme condition d'échantillonnage ou de fusion en séquences booléennes avec absence. Une compilation naïve de cet opérateur mènerait à générer pour certains programmes du code moins efficace que celui du programme synchrone équivalent. Par exemple, considérons le nœud `sum_by_4` :

```
let node sum_by_4 x = o where
  rec x1 = x when (1000)
  and x2 = x when (0100)
  and x3 = x when (0010)
  and x4 = x when (0001)
  and o = buffer(x1) + buffer(x2) + buffer(x3) + buffer(x4)
```

Si les quatre flots booléens correspondant aux horloges `(1000)`, `(0100)`, `(0010)` et `(0001)` sont générés de manière indépendante, cette implémentation est moins efficace que son analogue synchrone suivant, qui n'utilise qu'un flot booléen d'échantillonnage :

```
let node sum_by_4 x = o where
  rec x4 = x
  and x3 = 0 fby x4
  and x2 = 0 fby x3
  and x1 = 0 fby x2
  and o = (x1 + x2 + x3 + x4) when (0001)
```

La réponse vient peut être des horloges énumérées, bien adaptées pour faire ce genre de filtres et sûrement compilables plus efficacement. Ici par exemple, on utiliserait une unique horloge `(One Two Three Four)` :

```
let clock c = (One Two Three Four)

let node sum_by_4 x = o where
  rec x1 = x when One(c)
  and x2 = x when Two(c)
  and x3 = x when Three(c)
  and x4 = x when Four(c)
  and o = buffer(x1) + buffer(x2) + buffer(x3) + buffer(x4)
```

Chapitre 12

Conclusion

Les horloges sont le principal objet d'étude de la thèse que nous concluons ici. Ces suites booléennes sont associées aux flots de données dans les langages synchrones, pour marquer les instants de présence d'une valeur sur les flots. La connexion de deux nœuds de calcul peut être réalisée de manière synchrone, c'est-à-dire sans buffers et sans perte de valeurs, si l'horloge du flot produit sur le canal de communication est *égale* à l'horloge du flot attendu.

Le modèle n-synchrone permet une connexion plus souple des nœuds de calcul, en plaçant des buffers bornés sur les canaux de communication. Il n'est alors plus nécessaire que l'horloge du flot produit sur le canal soit strictement égale à l'horloge du flot attendu. Afin de garantir que les données sont produites à temps et que le nombre de données en attente d'être consommées est borné, on vérifie cependant que l'horloge de production est *adaptable* à l'horloge de consommation. Si c'est le cas, alors le flot produit peut être stocké dans un buffer de taille bornée en attendant d'être consommé, sans risque de perte de données, d'écriture dans un buffer plein ou de lecture dans un buffer vide.

Dans les langages synchrones flot de données, on vérifie que les programmes ne contiennent que des communications synchrones par une analyse de types, appelée *calcul d'horloge*. Chaque nœud reçoit un type d'horloges, qui décrit les horloges des sorties du nœud en fonction des horloges de ses entrées. L'application d'un nœud est bien typée si le type des flots que l'on connecte aux entrées du nœud est égal au type attendu. Deux types sont égaux si quelle que soit l'instanciation des variables qu'ils contiennent, les horloges représentées sont égales.

Pour s'assurer qu'une communication est n-synchrone, on demande que le type des flots que l'on connecte aux entrées du nœud soit un sous-type du type attendu. C'est le cas si quelle que soit l'instanciation des variables qu'ils contiennent, les horloges représentées par le premier sont adaptables aux horloges représentées par le second. Le calcul d'horloge nécessite donc de savoir vérifier qu'une horloge est égale à une autre et qu'une horloge est adaptable à une autre.

Nous nous sommes intéressés à deux classes d'horloges, sur lesquelles on sait vérifier la relation d'adaptabilité : les horloges ultimement périodiques et les horloges abstraites. Les horloges ultimement périodiques ont un comportement répétitif à partir d'un certain rang. Pour avoir la garantie que la relation est satisfaite à l'infini, il suffit donc de vérifier qu'elle l'est jusqu'à un certain instant où les horloges comparées reprennent toutes les deux leur comportement périodique. En ce qui concerne les horloges abstraites, elles sont encadrées par des horloges périodiques. Bien que l'on ne connaisse pas leur évolution à l'instant près, on peut avoir la garantie que la relation est satisfaite en considérant les pires cas possibles.

Dans le calcul d'horloge que nous avons présenté, les types sont *inférés*. Cela signifie que le programmeur n'a pas besoin d'annoter son programme avec les types des flots d'entrées, de sortie et des flots intermédiaires. Dans ce cadre, le calcul d'horloge doit collecter les contraintes d'égalité et de sous-typage. Si ces contraintes ne sont pas vérifiées quelle que soit l'instanciation des variables contenues dans les types, il faut trouver des instanciations les satisfaisant. Cette étape s'appelle la résolution des contraintes et constitue la partie difficile dans le cas des horloges périodiques, si l'on vise à accepter tous les programmes bien typés. Dans le cas des horloges abstraites, cette étape est traitée de manière simple et élégante : la difficulté se trouve plutôt en amont dans la manipulation des horloges, si l'on désire trouver des encadrements précis pour les horloges abstraites issues d'échantillonnages successifs.

Le langage n-synchrone que nous avons défini nous a permis d'expérimenter différents langages d'horloges, et différents algorithmes de résolution des contraintes de sous-typage associés. La communication n-synchrone est explicitée par le programmeur qui dispose pour cela d'un opérateur de bufferisation. À l'usage, nous trouvons cette solution agréable pour un programmeur habitué aux langages synchrones. La localisation explicite des buffers permet d'avoir une meilleure compréhension du programme, mais aussi d'aider et de diriger l'algorithme de résolution des contraintes qui calcule la taille des buffers nécessaires.

Une fois la génération de code mise en place, nous pourrons adapter les concepts et méthodes décris dans cette thèse à un langage plus riche comme Lucid Synchrone, et intégrer les améliorations décrites dans le chapitre précédent.

Index thématique

Bibliographie

[AS03] Jean-Paul Allouche et Jeffrey Shallit. *Automatic Sequences. Theory, Applications, Generalizations*. Cambridge University Press, 2003.

[BB91] Albert Benveniste et Gérard Berry. The Synchronous Approach to Reactive and Real-Time Systems. *In Proceedings of the IEEE*, pages 1270–1282, 1991.

[BC04] Yves Bertot et Pierre Castéran. *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions*. Springer-Verlag. Series: Texts in Theoretical Computer Science. An EATCS Series, 2004.

[BCE+03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic et Robert de Simone. The Synchronous Languages 12 Years Later. *Proceedings of the IEEE*, 91(1), Janvier 2003.

[BCP07] Darek Biernacki, Jean-Louis Colaco et Marc Pouzet. Clock-directed Modular Compilation from Synchronous Block-diagrams. *In Workshop on Automatic Program Generation for Embedded Systems (APGES)*, Salzburg, Austria, Octobre 2007. Embedded System Week.

[BdSM07] Julien Boucaron, Robert de Simone et Jean-Vivien Millo. Formal Methods for Scheduling of Latency-Insensitive Designs. *EURASIP Journal on Embedded Systems*, 2007 , Issue 1(ISSN:1687-3955):8–8, Janvier 2007.

[BELP96] Greet Bilsen, Marc Engels, Rudy Lauwereins et Jean Peperstraete. Cyclo-Static Dataflow. *Signal Processing, IEEE Transactions on*, 44(2):397–408, Février 1996.

[BGJ91] Albert Benveniste, Paul Le Guernic et Christian Jacquemot. Synchronous Programming with Events and Relations: the SIGNAL language and its Semantics. *Sci. Comput. Program.*, 16(2):103–149, 1991.

[BHLM94] Joseph Buck, Soonhoi Ha, Edward A. Lee et David G. Messerschmitt. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. *International Journal of Computer Simulation*, 4:155–182, Avril 1994. special issue on Simulation Software Development.

[BM07] Julien Boucaron et Jean-Vivien Millo. Compositionality of Statically Scheduled IP. *Electronic Notes in Theoretical Computer Science*, 200(1):71–87, Mai 2007. Proceedings of the Third International Workshop on Formal Methods for Globally Asynchronous Locally Synchronous Design (FMGALS 2007).

[Bou07] Julien Boucaron. *Modélisation formelle de systèmes Insensibles à la Latence et ordonnancement*. Thèse de doctorat, Université de Nice Sophia-Antipolis, 2007.

[BT01] Jean-Yves Le Boudec et Patrick Thiran. *Network Calculus : A Theory of Deterministic Queuing Systems for the Internet*. LNCS 2050. Springer Verlag, 2001.

[Buc93] Joseph T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. Thèse de doctorat, EECS Department, University of California, Berkeley, 1993.

[CC77] Patrick Cousot et Radhia Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. *In POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 238–252, New York, NY, USA, 1977.

[CCM+03] Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, Stavros Tripakis et Peter Niebert. From Simulink to SCADE/Lustre to TTA: a Layered Approach for Distributed Embedded Applications. *SIGPLAN Not.*, 38(7):153–162, 2003.

[CDDP07] Albert Cohen, Philippe Dumont, Marc Duranton et Marc Pouzet. Horloges entières. drafts, Août - Octobre 2007.

[CDE+05] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau et Marc Pouzet. Synchronizing Periodic Clocks. *In ACM International Conference on Embedded Software (EMSOFT'05)*, Jersey city, New Jersey, USA, Septembre 2005.

[CDE+06] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau et Marc Pouzet. N-Synchronous Kahn Networks: a Relaxed Model of Synchrony for Real-Time Systems. *In ACM International Conference on Principles of Programming Languages (POPL'06)*, Janvier 2006.

[CGHP04] Jean-Louis Colaço, Alain Girault, Grégoire Hamon et Marc Pouzet. Towards a Higher-order Synchronous Data-flow Language. *In ACM Fourth International Conference on Embedded Software (EMSOFT'04)*, Pisa, Italy, Septembre 2004.

[CH78] Patrick Cousot et Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. *In POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 84–96, New York, NY, USA, 1978. ACM.

[CKG06] Jordi Cortadella, Mike Kishinevsky et Bill Grundmann. SELF: Specification and design of synchronous elastic circuits. *In Proc. International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU)*, pages 16–21, Février 2006.

[CLR90] Thomas Cormen, Charles Leiserson et Ronald Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[CMPP08] Albert Cohen, Louis Mandel, Florence Plateau et Marc Pouzet. Abstraction of Clocks in Synchronous Data-flow Systems. *In The Sixth ASIAN Symposium on Programming Languages and Systems (APLAS 08)*, Bangalore, India, Décembre 2008.

[CMPP09] Albert Cohen, Louis Mandel, Florence Plateau et Marc Pouzet. Relaxing Synchronous Composition with Clock Abstraction. *In Hardware Design using Functional Languages (HFL 09)*, York, UK, Mars 2009.

[CMSV01] Luca P. Carloni, Kenneth L. McMillan et Alberto L. Sangiovanni-Vincentelli. Theory of Latency-Insensitive Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits ans Systems*, 20(9):1059–1076, Septembre 2001.

[CP96] Paul Caspi et Marc Pouzet. Synchronous Kahn Networks. *In ACM SIGPLAN International Conference on Functional Programming*, Philadelphia, Pensylvania, Mai 1996.

[CP01] Pascal Cuoq et Marc Pouzet. Modular Causality in a Synchronous Stream Language. *In European Symposium on Programming (ESOP'01)*, Genova, Italy, Avril 2001.

[CP03] Jean-Louis Colaço et Marc Pouzet. Clocks as First Class Abstract Types. *In Third International Conference on Embedded Software (EMSOFT'03)*, Philadelphia, Pennsylvania, USA, Octobre 2003.

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs et John A. Plaice. LUSTRE: A declarative language for programming synchronous systems. *In POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 178–188, New York, NY, USA, 1987.

[Cpl] Cplex. Cplex. http://www.decf.berkeley.edu/help/apps/ampl/cplex-doc/.

[Cru91] Rene L. Cruz. A Calculus for Network Delay. I. Network Elements in Isolation. *Information Theory, IEEE Transactions on*, 37(1):114–131, 1991.

[CSV02] Luca P. Carloni et Alberto L. Sangiovanni-Vincentelli. Coping with Latency in SOC Design. *IEEE Micro*, 22(5):24–35, 2002.

[Cuo02] Pascal Cuoq. *Ajout de synchronisme dans les langages fonctionnels typés*. Thèse de doctorat, Université Pierre et Marie Curie, Paris, France, Octobre 2002.

[Cur05] Adrian Curic. *Implementing Lustre Programs on Distributed Platforms with Real-time Constraints*. Thèse de doctorat, Université Joseph Fourier, 2005.

[Dil90] David L. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. *In Proceedings of the international workshop on Automatic verification methods for finite state systems*, pages 197–212, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[Gér08] Léonard Gérard. Des horloges entières pour la répartition de programmes synchrones flot de données. Rapport de master, Université Paris-Sud 11, 2008.

[Glp] Glpk. Gnu linear programming kit. http://www.gnu.org/software/glpk/.

[Hal84] Nicolas Halbwachs. *Modélisation et analyse du comportement des systèmes informatiques temporisés*. Thèse d'etat, Institut National Polytechnique de Grenoble, Juin 1984.

[Hal93] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Pub., 1993.

[Ham02] Grégoire Hamon. *Calcul d'horloge et structures de contrôle dans Lucid Synchrone, un langage de flots synchrone à la ML*. Thèse de doctorat, Université Paris 6, Novembre 2002.

[HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond et Daniel Pilaud. The Synchronous Data Flow Programming Language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, Septembre 1991.

[HR99] Nicolas Halbwachs et Pascal Raymond. Validation of Synchronous Reactive Systems: from Formal Verification to Automatic Testing. *In ASIAN'99, Asian Computing Science Conference*, Phuket (Thailand), Décembre 1999. LNCS 1742, Springer Verlag.

[Kah74] Gilles Kahn. The Semantics of a Simple Language for Parallel Programming. *In* J. L. Rosenfeld, éditeur. *Information processing*, pages 471–475, Stockholm, Sweden, Août 1974. North Holland Publishing Company.

[KTA03] Michal Karczmarek, William Thies et Saman Amarasinghe. Phased Scheduling of Stream Programs. *In Languages, Compilers, and Tools for Embedded Systems*, San Diego, CA, Juin 2003.

[LGT00] Xavier Lagrange, Philippe Godlewski et Sami Tabbane. *Réseaux GSM : des principes à la norme*. Hermès Science, Paris, 5ème édition édition, 2000.

[LM87a] Edward A. Lee et David G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, 36(1), Janvier 1987.

[LM87b] Edward A. Lee et David G. Messerschmitt. Synchronous Data Flow. *IEEE Transactions on Computers*, 75(9), Septembre 1987.

[Lot02] M. Lothaire. *Algebraic Combinatorics on Words*, chapitre 2. Cambridge University Press, 2002.

[MH40] Marston Morse et Gustav A. Hedlund. Symbolic Dynamics II. Sturmian Trajectories. *American Journal of Mathematics*, 62(1):1–42, 1940.

[Mil08] Jean-Vivien Millo. *Ordonnancements Périodiques dans les Réseaux de Processus : Application à la Conception Insensible aux Latences*. Thèse de doctorat, Université de Nice-Sophia Antipolis, Décembre 2008.

[MP09] Louis Mandel et Florence Plateau. Abstraction d'horloges dans les systèmes synchrones flot de données. *In Vingtièmes Journées Francophones des Langages Applicatifs (JFLA 09)*, Saint-Quentin sur Isère, France, Février 2009.

[Par95] Thomas Martyn Parks. *Bounded Scheduling of Process Networks*. Thèse de doctorat, EECS Department, University of California, Berkeley, Berkeley, CA, USA, 1995.

[PHLB93] José Luis Pino, Soonhoi Ha, Edward A. Lee et Joseph T. Buck. Software Synthesis for DSP Using Ptolemy. *Journal of VLSI Signal Processing*, 9:7–21, 1993.

[Pie05] Benjamin C. Pierce, éditeur. *Advanced Topics in Types and Programming Languages*, chapitre 10. MIT Press, 2005.

[Pou02] Marc Pouzet. *Lucid Synchrone: un langage synchrone d'ordre supérieur*. Paris, France, 14 novembre 2002. Habilitation à diriger les recherches.

[Pou06] Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, Avril 2006. Distribution available at: www.lri.fr/~pouzet/lucid-synchrone.

[PPL95] Thomas M. Parks, José Luis Pino et Edward A. Lee. A Comparison of Synchronous and Cyclo-Static Dataflow. *In ASILOMAR '95: Proceedings of the 29th Asilomar Conference on Signals, Systems and Computers (2-Volume Set)*, page 204, Washington, DC, USA, 1995. IEEE Computer Society.

[Pto] Ptolemy. The Ptolemy Project. http://ptolemy.eecs.berkeley.edu/.

[SBT96] Thomas R. Shiple, Gérard Berry et Hervé Touati. Constructive Analysis of Cyclic Circuits. *In EDTC '96: Proceedings of the 1996 European conference on Design and Test*, page 328, Washington, DC, USA, 1996. IEEE Computer Society.

[Sma98] Irina Smarandache. Transformations affines d'horloges: application au codesign de systèmes temps-réel en utilisant les langages SIGNAL et ALPHA. *Université de Rennes 1, France*, Octobre 1998.

[TCN00] Lothar Thiele, Samarjit Chakraborty et Martin Naedele. Real-Time Calculus for Scheduling Hard Real-Time Systems. *In International Symposium on Circuits and Systems ISCAS 2000*, volume 4, pages 101–104, Geneva, Switzerland, Mars 2000.

[TKA02] William Thies, Michal Karczmarek et Saman P. Amarasinghe. StreamIt: A Language for Streaming Applications. *In CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, London, UK, 2002. Springer-Verlag.

[TN98] Jean-Pierre Talpin et David Nowak. A Synchronous Semantics of Higher-Order Processes for Modeling Reconfigurable Reactive Systems. *In Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 78–89. Springer-Verlag, 1998.

[Vui94] Jean E. Vuillemin. On Circuits and Numbers. *IEEE Transactions on Computers*, 43(8):868–879, Août 1994.

[WBS07] Maarten Wiggers, Marco Bekooij et Gerard J. M. Smit. Efficient Computation of Buffer Capacities for Cyclo-Static Dataflow Graphs. *In DAC '07: Proceedings of the 44th annual conference on Design Automation*, pages 658–663, 2007.

[WT05] Ernesto Wandeler et Lothar Thiele. Abstracting Functionality for Modular Performance Analysis of Hard Real-Time Systems. *In Asia and South Pacific Desing Automation Conference (ASP-DAC)*, pages 697–702, Shanghai, P.R. China, Janvier 2005.

Annexes

Annexe A

Documentation des outils autour de Lucy-n

Par Louis Mandel et Florence Plateau

A.1 `lucync` : le typeur de Lucy-n

`lucync` est le typeur pour le langage Lucy-n. Il prend en entrée des fichiers sources Lucy-n contenant des définitions de nœuds et d'horloges et affiche sur la sortie standard les types de données et d'horloges de chaque définition.

À partir d'un fichier `file.ls`, le typeur produit le fichier `file.annot` qui contient les annotations de type d'horloges de chaque expression du programme et le `file.bannot` qui contient la taille de chaque buffer du programme. Ces fichiers sont utilisés pour afficher les types et les tailles de buffer dans emacs avec les commandes `C-c C-t` et `C-c C-b`.

L'utilisation classique du typeur est :

```
lucync [options] file.ls
```

Les principales options reconnues par le typeur sont :

`-dbuff` Affiche la taille des buffers.

`-cstrs` Affiche les systèmes de contraintes de sous-typage à résoudre.

`-verbose` Affiche les différentes étapes de la résolution des contraintes de sous-typage.

`-ce <lang>` Sélectionne le langage d'horloges à utiliser.

- `pbw` le langage d'horloges périodiques. Les algorithmes d'unification et de résolution peuvent être choisis avec les options `-unif` et `-solver`. Il s'agit du langage d'horloges sélectionné par défaut.
- `abs` le langage d'horloges mélangeant horloges périodiques et abstraites. L'algorithme d'unification est structurel. La résolution des contraintes de sous-typage se fait dans le domaine abstrait.
- `names` un autre langage d'horloges mélangeant horloges périodiques et abstraites, où l'égalité ne se fait que par noms. L'algorithme d'unification est structurel. Une horloge est un sous-type d'une autre uniquement si elles sont égales.

`-unif <algo>` Sélectionne l'algorithme d'unification à utiliser avec le langage d'horloges périodiques.

`s`, `struct` unification structurelle.

`i`, `interp` unification où les horloges sont interprétées avant d'être unifiées.

`si`, `semii` unification semi-interprétée. Chaque contrainte est unifiée de manière structurelle si possible et de manière interprétée sinon. Il s'agit de l'algorithme d'unification sélectionné par défaut.

`g`, `global` unification globale. Les contraintes d'unification sont collectées et résolues avec les contraintes de sous-typage.

`sg`, `semig` unification semi-globale. Chaque contrainte est unifiée de manière structurelle si possible et collectée pour unification globale sinon.

`-solver <algo>` Sélectionne l'algorithme de résolution des contraintes de sous-typage à utiliser avec le langage d'horloges périodiques.

`concr` les contraintes de sous-typage sont résolues dans le domaine concret. Il s'agit de l'algorithme sélectionné par défaut.

`abs` les contraintes de sous-typage sont résolues dans le domaine abstrait.

`-glpsol <cmd>` Change le nom de la commande à appeler pour résoudre les contraintes au format Cplex. Par défaut la commande est `glpsol`.

A.2 `eval_clock` : évaluer une expression d'horloges

Nous fournissons en plus du typeur de programmes Lucy-n un petit utilitaire `eval_clock` qui permet d'évaluer les expressions d'horloges. L'utilisation classique de cette commande est

```
eval_clock [options] "expr"
```

où l'expression `expr` doit être de la forme suivante :

expr ::= `(` *expr* `)` | *pbword*
| *abstraction-phd* | *abstraction-aplas* | *abstraction-jfla* | *abstraction-bounds*
| `not` *expr* | `not~` *expr* | *expr* `on` *expr* | *expr* `on~` *expr* | `nf` *expr*
| `abs` *expr* | `early` *expr* | `late` *expr* | `eval` *expr*

pbword ::= *word* `(` *word* `)` | `(` *word* `)`
word ::= {*baseword*}$^+$
baseword ::= `0` | `1` | `?` `(` *integer* `,` *integer* `)` | `0^` *pow* | `1^` *pow* | `{` *word* `}^` *pow*

abstraction-phd ::= `<` *rational* `,` *rational* `>` `(` *rational* `)`
abstraction-aplas ::= `[` *rational* `,` *rational* `]` `(` *rational* `)`
abstraction-jfla ::= `(` *rational* `,` *rational* `,` *rational* `)`
abstraction-bounds ::= `(` *pbword* `,` *pbword* `)`

rational ::= *integer* | *integer* `/` *integer*
pow ::= *integer* | `(` *arith* `)`
arith ::= *integer* | `(` *arith* `)`
| *arith* `+` *arith* | *arith* `-` *arith* | *arith* `*` *arith* | *arith* `/` *arith*

La principale option est :

`-abs` Sélectionne le domaine abstrait à utiliser.

- `phd` le domaine abstrait présenté dans la thèse de Florence Plateau. Les valeurs abstraites de la forme *abstraction-aplas*, *abstraction-jfla* et *abstraction-bounds* sont implicitement traduites en valeurs abstraites de la forme *abstraction-phd*. Il s'agit de l'option par défaut.
- `aplas` le domaine abstrait introduit dans l'article APLAS 2008.
- `jfla` le domaine abstrait introduit dans l'article JFLA 2009.
- `bounds` le domaine abstrait définit par l'ensemble des mots compris entre deux mots ultimement périodiques. Cette abstraction sert de référence pour étudier la précision des autres abstractions.

Annexe B

Types d'horloges des exemples du chapitre 2

Pour les exemples du chapitre 2, la résolution de la partie II et la résolution de la partie III renvoient le même résultat. Nous fournissons ci-dessous les sorties du typeur, obtenues avec les commandes :

```
lucync -ce pbw -dbuff nom_fichier.ls
lucync -ce abs -dbuff nom_fichier.ls
```

L'ajout de l'option `-dcstrs` sur la ligne de commance permet de visualiser les constraintes de sous-typage collectées lors du typage de chaque nœud.

B.1 Exemples sans buffers

fichier `average.ls` :

```
average :: forall 'a. ('a * 'a) -> 'a
average2 :: forall 'a. ('a * 'a) -> 'a
```

fichier `sum.ls` :

```
sum :: forall 'a. 'a -> 'a
sum_odd :: forall 'a. 'a -> 'a on (10)
sum_odd_bis :: forall 'a. 'a -> 'a on half
sum_odd_even :: forall 'a. 'a -> 'a
sum_sum_odd :: forall 'a. 'a -> 'a on (10) on (10)
foo :: forall 'a. 'a -> 'a on (10)
```

fichier `bad.ls` :

```
File "bad.ls", line 1, characters 17-33:
Cannot unify clock 'a2 on (10) with clock 'a2
```

fichier `not_so_bad.ls` :

```
File "not_so_bad.ls", line 4, characters 10-17:
Cannot unify clock 'a6 on (01) with clock 'a6 on (10)
```

B.2 Exemples avec buffers

fichier `buffers.ls` :

```
good :: forall 'a. 'a -> 'a on (01)
Buffer line 4, characters 10-20: size = 1

first_one :: forall 'a. 'a

my_fby :: forall 'a. ('a * 'a) -> 'a
Buffer line 9, characters 38-47: size = 1

first_three :: forall 'a. 'a

my_fby3 :: forall 'a. ('a * 'a) -> 'a
Buffer line 12, characters 66-75: size = 3

stutter :: forall 'a. 'a on (10) -> 'a

new_stutter :: forall 'a. 'a on (10) -> 'a
Buffer line 18, characters 39-48: size = 1

sum_by_4 :: forall 'a. 'a -> 'a on (0^3 1)
Buffer line 25, characters 10-20: size = 1
Buffer line 25, characters 23-33: size = 1
Buffer line 25, characters 36-46: size = 1
Buffer line 25, characters 49-59: size = 0

sum_by_4_opt :: forall 'a. 'a -> 'a on (0^3 1)
Buffer line 32, characters 24-34: size = 1
Buffer line 32, characters 17-40: size = 1
Buffer line 32, characters 10-46: size = 1

join4 :: forall 'a.
( 'a on (1^3 0) on (1^2 0) on (10)
* 'a on (1^3 0) on (1^2 0) on not (10)
* 'a on (1^3 0) on not (1^2 0)
* 'a on not (1^3 0)) -> 'a
```

NB: avec la résolution abstraite, le type de `sum_by_4` est :

```
sum_by_4 :: forall 'a. 'a -> 'a on <-3/4, 0>(1/4)
```

Ce type est équivalent au type indiqué plus haut, obtenu avec la résolution concrète, car $\mathit{concr}\left(\langle -\frac{3}{4}, 0\rangle\left(\frac{1}{4}\right)\right) = \left\{(0^3 1)\right\}$.

Annexe C

Sémantique du nœud sum du chapitre 2

Cette annexe illustre l'évaluation d'un programme selon la sémantique de Kahn présentée dans la section 2.2.1. On considère la déclaration du nœud `sum` de la section 2.1.2 :

```
let node sum x = o where
  rec o = x + (0 fby o)
```

Calculons sa sémantique :

$[\![\texttt{let node } sum\ x = o \texttt{ where rec } o = x + (0 \texttt{ fby } o)]\!]_\rho$
$= \rho + [sum \leftarrow \lambda s.[\![o \texttt{ where rec } o = x + (0 \texttt{ fby } o)]\!]_{\rho+[x\leftarrow s]}]$

Il faut calculer la sémantique de o `where rec` $o = x + (0$ `fby` $o)$:
$[\![o \texttt{ where rec } o = x + (0 \texttt{ fby } o)]\!]_{\rho+[x\leftarrow s]} = [\![o]\!]_{\rho+[x\leftarrow s]+\rho'}$ où $\rho' = [\![o = x + (0 \texttt{ fby } o)]\!]_{\rho+[x\leftarrow s]}$

Calculons ρ' :
$\rho' = [\![o = x + (0 \texttt{ fby } o)]\!]_{\rho+[x\leftarrow s]} = [o \leftarrow o^\sharp]$ où $o^\sharp = fix(\lambda d.[\![x + (0 \texttt{ fby } o)]\!]_{\rho+[x\leftarrow s]+[o\leftarrow d]})$

Calculons $o^\sharp$.
Soit $f = (\lambda d.[\![x + (0 \texttt{ fby } o)]\!]_{\rho+[x\leftarrow s]+[o\leftarrow d]})$. On a $o^\sharp = fix(f)$. On doit donc calculer $fix(f)$.

1. Calcul de $f\varepsilon$:
   $$\begin{array}{rcl} f\varepsilon & = & [\![x + (0 \texttt{ fby } o)]\!]_{\rho+[x\leftarrow s]+[o\leftarrow \varepsilon]} \\ & = & +^\sharp([\![x]\!]_{\rho+[x\leftarrow s]+[o\leftarrow \varepsilon]}, [\![0 \texttt{ fby } o]\!]_{\rho+[x\leftarrow s]+[o\leftarrow \varepsilon]}) \end{array}$$
   – D'une part, on a $[\![x]\!]_{\rho+[x\leftarrow s]+[o\leftarrow \varepsilon]} = s$.
   – D'autre part, on calcule $[\![0 \texttt{ fby } o]\!]_{\rho+[x\leftarrow s]+[o\leftarrow \varepsilon]}$:
   $$\begin{array}{rcl} [\![0 \texttt{ fby } o]\!]_{\rho+[x\leftarrow s]+[o\leftarrow \varepsilon]} & = & \texttt{fby}^\sharp([\![0]\!]_{\rho+[x\leftarrow s]+[o\leftarrow \varepsilon]}, [\![o]\!]_{\rho+[x\leftarrow s]+[o\leftarrow \varepsilon]}) \\ & = & \texttt{fby}^\sharp(0.[\![0]\!]_{\rho+[x\leftarrow s]+[o\leftarrow \varepsilon]}, \varepsilon) \\ & = & 0.\varepsilon \end{array}$$
   Donc
   $$\begin{array}{rcl} f\varepsilon & = & +^\sharp([\![x]\!]_{\rho+[x\leftarrow s]+[o\leftarrow \varepsilon]}, [\![0 \texttt{ fby } o]\!]_{\rho+[x\leftarrow s]+[o\leftarrow \varepsilon]}) \\ & = & +^\sharp(s, 0.\varepsilon) \\ & = & (v_1 + 0).+^\sharp(s', \varepsilon) \text{ si } s = v_1.s' \\ & = & v_1.\varepsilon \end{array}$$
2. Calcul de $f(f\varepsilon)$:
   $$\begin{array}{rcl} f(f\varepsilon) & = & [\![x + (0 \texttt{ fby } o)]\!]_{\rho+[x\leftarrow v_1.s']+[o\leftarrow v_1.\varepsilon]} \\ & = & +^\sharp([\![x]\!]_{\rho+[x\leftarrow v_1.s']+[o\leftarrow v_1.\varepsilon]}, [\![0 \texttt{ fby } o]\!]_{\rho+[x\leftarrow v_1.s']+[o\leftarrow v_1.\varepsilon]}) \end{array}$$

– D'une part, on a $[\![x]\!]_{\rho+[x\leftarrow v_1.s']+[o\leftarrow v_1.\varepsilon]} = v_1.s'$.
– D'autre part, on calcule $[\![0 \texttt{ fby } o]\!]_{\rho+[x\leftarrow v_1.s']+[o\leftarrow v_1.\varepsilon]}$:

$$\begin{array}{rcl} [\![0 \texttt{ fby } o]\!]_{\rho+[x\leftarrow v_1.s']+[o\leftarrow v_1.\varepsilon]} & = & \texttt{fby}^\sharp([\![0]\!]_{\rho+[x\leftarrow v_1.s']+[o\leftarrow v_1.\varepsilon]}, [\![o]\!]_{\rho+[x\leftarrow v_1.s]+[o\leftarrow v_1.\varepsilon]}) \\ & = & \texttt{fby}^\sharp((0.[\![0]\!]_{\rho+[x\leftarrow v_1.s']+[o\leftarrow v_1.\varepsilon]}, v_1.\varepsilon) \\ & = & 0.v_1.\varepsilon \end{array}$$

Donc

$$\begin{array}{rcl} f(f\varepsilon) & = & +^\sharp([\![x]\!]_{\rho+[x\leftarrow v_1.s']+[o\leftarrow v_1.\varepsilon]}, [\![0 \texttt{ fby } o]\!]_{\rho+[x\leftarrow v_1.s']+[o\leftarrow v_1.\varepsilon]}) \\ & = & +^\sharp(v_1.s', 0.v_1.\varepsilon) \\ & = & (v_1+0).(v_2+v_1).+^\sharp(s'', \varepsilon) \text{ si } s = v_1.v_2.s'' \\ & = & v_1.(v_2+v_1).\varepsilon \end{array}$$

3. Calcul de $f(f(f\varepsilon))$:
   En suivant le même principe, on obtient :
   $f(f(f\varepsilon)) = v_1.(v_2+v_1).(v_3+v_2+v_1).\varepsilon$ si $s = v_1.v_2.v_3.s'''$.

On peut en déduire que $\mathit{fix}(f) = v_1.(v_2+v_1).(v_3+v_2+v_1).\ldots$
et donc que $o^\sharp = v_1.(v_2+v_1).(v_3+v_2+v_1).\ldots.$

Par conséquent, $\rho' = [o \leftarrow v_1.(v_2+v_1).(v_3+v_2+v_1).\ldots]$.
Comme $[\![o \texttt{ where rec } o = x + (0 \texttt{ fby } o)]\!]_{\rho+[x\leftarrow s]} = [\![o]\!]_{\rho+[x\leftarrow s]+\rho'}$,
on en déduit que $[\![o \texttt{ where rec } o = x + (0 \texttt{ fby } o)]\!]_{\rho+[x\leftarrow s]} = v_1.(v_2+v_1).(v_3+v_2+v_1).\ldots$

Par conséquent, $[\![\texttt{let node } sum\ x = o \texttt{ where rec } o = x + (0 \texttt{ fby } o)]\!]_\rho$
$= \rho + [sum \leftarrow \lambda s.\ v_1.(v_2+v_1).(v_3+v_2+v_1).\ldots]$ si $s = v_1.v_2.v_3.s'''$.

Soit $\rho_1 = (\varnothing, sum \leftarrow \lambda(v_1.v_2.v_2.s''').\ v_1.(v_2+v_1).(v_3+v_2+v_1).\ldots, \varnothing)$.
L'application du nœud `sum` à la constante 1 donne la séquence des entiers naturels :

$$\begin{array}{rl} & [\![sum\ 1]\!]_{\rho_1} \\ = & (\rho_1(sum))[\![1]\!]_{\rho_1} \\ = & (\lambda(v_1.v_2.v_2.s''').\ v_1.(v_2+v_1).(v_3+v_2+v_1).\ldots)(1.1.1.[\![1]\!]_{\rho_1}) \\ = & 1.2.3.\ldots \end{array}$$

Annexe D

Propriétés sur les parties entières

Lemme D.1. $\forall x \in \mathbb{Z}, \forall n \in \mathbb{N}^*,\ \left\lceil \frac{x}{n} - \frac{n-1}{n} \right\rceil = \left\lfloor \frac{x}{n} \right\rfloor$ et $\left\lfloor \frac{x}{n} + \frac{n-1}{n} \right\rfloor = \left\lceil \frac{x}{n} \right\rceil$.

Démonstration :

Soit $x = y \times n + r$ avec $y \in \mathbb{N}$ et $0 \leq r < n$. On $\left\lfloor \frac{x}{n} \right\rfloor = y$.
D'autre part, $x-(n-1) = y \times n + r - n + 1 = y \times n - (-r+n-1)$ avec $0 \leq -r+n-1 < n$.
Donc $\left\lceil \frac{x-(n-1)}{n} \right\rceil = y = \left\lfloor \frac{x}{n} \right\rfloor$.
La deuxième équation se déduit de la première en instanciant x par $x + (n-1)$. □

Lemme D.2. Si $a = \left\langle b^0, b^1 \right\rangle (r)$ est en forme normale, il existe une infinité d'instants i tels que $\left\lfloor r \times i + b^1 \right\rfloor = r \times i + b^1$ et une infinité d'instants i tels que $\left\lceil r \times i + b^0 \right\rceil = r \times i + b^0$. Il existe donc une infinité d'instants tels que $\mathcal{O}_{early_a}(i) = r \times i + b^1$ et une infinité d'instants tels que $\mathcal{O}_{late_a}(i) = r \times i + b^0$.

Démonstration :

Soit $a = \left\langle \frac{k^0}{\ell}, \frac{k^1}{\ell} \right\rangle \left(\frac{n}{\ell}\right)$ avec $\text{pgcd}(n, \ell) = 1$.
Si $\ell = 1$, la propriété est vérifiée pour tout i.
Si $\ell > 1$, comme $\text{pgcd}(n, \ell) = 1$, par le théorème de Bézout, il existe $x, y \in \mathbb{Z}$ tels que $n \times y + \ell \times x = 1$. Donc il existe $x, y \in \mathbb{Z}$ tels que $n \times (y \times -k^1) + \ell \times (x \times -k^1) = -k^1$ c'est-à-dire tels que $n \times (y \times -k^1) + k^1 = \ell \times (x \times k^1)$.
Donc il existe un $i = (y \times -k^1) + z \times \ell$ et un $x' = (x \times k^1) + z \times n$ (avec $z \in \mathbb{N}$) tels que $i \geq 0$, $x' \geq 0$ et $n \times i + k^1 = \ell \times x'$. Pour ce i, on a $\frac{n \times i + k^1}{\ell} = x' = \left\lfloor \frac{n \times i + k^1}{\ell} \right\rfloor$.
C'est aussi le cas pour tous les $i' = i + x \times \frac{\ell}{n}$.
La preuve qu'il existe une infinité de i tels que $\left\lceil r \times i + b^0 \right\rceil = r \times i + b^0$ suit le même principe. □

Le lemme D.2 est utilisé dans la preuve de la propriété 7.17 page 166.

Lemme D.3. $\lfloor x \rfloor - \lceil y \rceil \leq x - y \leq \lceil x \rceil - \lfloor y \rfloor$

Démonstration :

$\lfloor x \rfloor \leq x \leq \lceil x \rceil$ et $-\lceil y \rceil \leq -y \leq -\lfloor y \rfloor$, donc $\lfloor x \rfloor - \lceil y \rceil \leq x - y \leq \lceil x \rceil - \lfloor y \rfloor$. □

Le lemme D.3 est utilisé dans la preuve de la propriété 7.21 page 172.

Lemme D.4. $\lfloor x - y \rfloor - 1 \leq \lfloor x \rfloor - \lceil y \rceil \leq \lfloor x - y \rfloor$

Démonstration :

D'après le lemme D.3, $\lfloor x \rfloor - \lceil y \rceil \leq x - y$. Donc $\lfloor \lfloor x \rfloor - \lceil y \rceil \rfloor \leq \lfloor x - y \rfloor$. Comme $\lfloor x \rfloor \in \mathbb{N}$ et $\lceil y \rceil \in \mathbb{N}$, on peut en conclure que $\lfloor \lfloor x \rfloor - \lceil y \rceil \rfloor = \lfloor x \rfloor - \lceil y \rceil$, donc que $\lfloor x \rfloor - \lceil y \rceil \leq \lfloor x - y \rfloor$.

D'autre part, $x-1 < \lfloor x \rfloor$ et $\lceil y \rceil < y+1$ c'est-à-dire $-(y+1) < -\lceil y \rceil$. Donc $x-1-(y+1) < \lfloor x \rfloor - \lceil y \rceil$. Comme $(\lfloor x \rfloor - \lceil y \rceil) \in \mathbb{N}$, on a $\lfloor x - 1 - y - 1 \rfloor + 1 \leq \lfloor x \rfloor - \lceil y \rceil$ c'est-à-dire $\lfloor x - y \rfloor - 1 \leq \lfloor x \rfloor - \lceil y \rceil$. □

Le lemme D.4 est utilisé dans la preuve de la propriété 7.28 page 176.

Annexe E

Horloges affines

E.1 Preuve du calcul de p_{a_1} *on* p_{a_2}

Lemme E.1 (Calcul de p_{a_1} *on* p_{a_2}). Soient $p_{a_1} = 0^{d_1}(10^{l_1-1})$ et $p_{a_2} = 0^{d_2}(10^{l_2-1})$.

$$p_{a_1} \textit{ on } p_{a_2} = 0^{d_1+d_2\times l_1}(10^{l_1\times l_2-1})$$

Démonstration :

Soit $p_{a_3} = p_{a_1} \textit{ on } p_{a_2} = u_3(v_3)$.

Tailles du préfixe et du motif périodique :

En appliquant la formule du calcul du *on* sur les mots périodiques (proposition 4.1), on obtient :

$$d_3 = |u_3| = \begin{cases} d_1 & \text{si } d_2 = 0 \\ \max(d_1, d_1 + (d_2-1)\times l_1) = d_1 + (d_2-1)\times l_1 + 1 & \text{sinon} \end{cases}$$
$$l_3 = |v_3| = l_2 \times l_1$$

Éléments du préfixe de p_{a_3} :

En appliquant la formule des éléments du résultat d'un *on* sur des mots périodiques, on obtient :

$$\forall i,\ 1 \le i \le d_3,\ u_3[i] = \begin{cases} 0 & \text{si } p_{a_1}[i] = 0 \\ p_{a_2}[\mathcal{O}_{p_{a_1}}(i)] & \text{si } p_{a_1}[i] = 1 \end{cases}$$

D'une part, d'après la formule sur les éléments d'une horloge affine, on a :
$p_{a_1}[i] = 1 \Leftrightarrow i = d_1 + x \times l_1 + 1$ avec $x \in \mathbb{N}$.
D'autre part, d'après la formule sur la fonction de cumul d'une horloge affine, on a :
$\forall i > d_1, \mathcal{O}_{p_{a_1}}(i) = \left\lceil \frac{(i-d_1)}{l_1} \right\rceil$.
De ces deux remarques, la formule des éléments du préfixe du résultat du *on* devient :

$$\forall i,\ 1 \le i \le d_3,\ u_3[i] = \begin{cases} 0 & \text{si } p_{a_1}[i] = 0 \\ p_{a_2}[\left\lceil \frac{(i-d_1)}{l_1} \right\rceil] & \text{si } i = d_1 + x \times l_1 + 1 \text{ avec } x \in \mathbb{N} \end{cases}$$

Si $d_2 = 0$, alors $d_3 = d_1$. Donc $i \le d_1$, et on est toujours dans le premier cas de la formule, c'est-à-dire $p_{a_1}[i] = 0$. Par conséquent, $u_3[i] = 0$.

Sinon, $d_3 = d_1 + (d_2 - 1) \times l_1 + 1$, donc $\forall i \leq d_3$ tel que $i = d_1 + x \times l_1 + 1$, $i - d_1 \leq d_3 - d_1$, c'est-à-dire $i - d_1 \leq (d_2 - 1) \times l_1 + 1$. Donc $\left\lceil \frac{(i-d_1)}{l_1} \right\rceil \leq d_2$, et par conséquent, $p_{a_2}[\left\lceil \frac{(i-d_1)}{l_1} \right\rceil] = \mathtt{0}$.

Donc dans ce cas, on a aussi toujours $u_3[i] = \mathtt{0}$.

Le préfixe est donc uniquement composé de $\mathtt{0}$.

Éléments de la partie périodique :

En appliquant la formule sur les mots périodiques, on obtient :

$$\forall i,\ 1 \leq i \leq l_3,\ v_3[i] = \begin{cases} \mathtt{0} & \text{si } p_{a_1}[d_3 + i] = \mathtt{0} \\ p_{a_2}[\mathcal{O}_{p_{a_1}}(d_3 + i)] & \text{si } p_{a_1}[d_3 + i] = \mathtt{1} \end{cases}$$

D'après la formule donnant les éléments d'une horloge affine, on a :
$p_{a_1}[d_3 + i] = \mathtt{1} \Leftrightarrow d_3 + i = d_1 + x_1 \times l_1 + 1$ avec $x_1 \in \mathbb{N}$.

Si $d_2 = 0$, alors $d_3 = d_1$. Donc $p_{a_1}[d_3+i] = \mathtt{1} \Leftrightarrow i = x \times l_1 + 1$ avec $x \in \mathbb{N}$. Dans ce cas, $v_3[i] = p_{a_2}[\mathcal{O}_{p_{a_1}}(d_3+i)]$. D'après la formule sur la fonction de cumul des horloges affines, on a $\mathcal{O}_{p_{a_1}}(d_3+i) = \mathcal{O}_{p_{a_1}}(d_1+i) = \left\lceil \frac{i}{l_1} \right\rceil$. Donc $p_{a_2}[\mathcal{O}_{p_{a_1}}(d_3+i)] = p_{a_2}[\left\lceil \frac{i}{l_1} \right\rceil]$. $p_{a_2}[\left\lceil \frac{i}{l_1} \right\rceil] = \mathtt{1} \Leftrightarrow \left\lceil \frac{i}{l_1} \right\rceil = x \times l_2 + 1$ avec $x \in \mathbb{N}$. Les seuls $i \leq l_3 = \ell_1 \times l_2$ tels que cette équation soit vérifiée sont les $i \leq l_1$. Pour ces i, $\left\lceil \frac{i}{l_1} \right\rceil = 1$, et donc le seul élément valant $\mathtt{1}$ de la partie périodique est le premier.

Sinon , $d_3 = d_1 + (d_2 - 1) \times l_1 + 1$. Donc $p_{a_1}[d_3 + i] = \mathtt{1} \Leftrightarrow d_1 + (d_2 - 1) \times l_1 + 1 + i = d_1 + x_1 \times l_1 + 1 \Leftrightarrow i = x_1 \times l_1 - (d_2 - 1) \times l_1$ (avec $x_1 \leq l_2 + d_2 - 1$ car $i \leq l_3 = l_1 \times l_2$). Dans ce cas, $v_3[i] = p_{a_2}[\mathcal{O}_{p_{a_1}}(d_3 + i)]$. D'après la formule sur la fonction de cumul des horloges affines, on a $\mathcal{O}_{p_{a_1}}(d_3 + i) = \left\lceil \frac{(d_3+i)-d_1}{l_1} \right\rceil = \left\lceil \frac{(d_1+(d_2-1)\times l_1+1+i)-d_1}{l_1} \right\rceil = \left\lceil \frac{(d_2-1)\times l_1+1+i}{l_1} \right\rceil = \left\lceil \frac{(d_2-1)\times l_1+1+(x_1\times l_1-(d_2-1)\times l_1)}{l_1} \right\rceil = \left\lceil \frac{1+x_1\times l_1}{l_1} \right\rceil = x_1 + \left\lceil \frac{1}{l_1} \right\rceil = x_1 + 1$

Donc $p_{a_2}[\mathcal{O}_{p_{a_1}}(d_3 + i)] = p_{a_2}[x_1 + 1]$.

D'après la formule sur les éléments d'un horloge affine, $p_{a_2}[x_1+1] = \mathtt{1} \Leftrightarrow x_1 + 1 = d_2 + x_2 \times l_2 + 1 \Leftrightarrow x_1 = d_2 + x_2 \times l_2$ Comme $x_1 \leq l_2 + d_2 - 1$, $p_{a_2}[x_1 + 1] = \mathtt{1} \Leftrightarrow x_1 = d_2$.

Donc $p_{a_1}[d_3 + i] = \mathtt{1} \Leftrightarrow d_3 + i = d_1 + d_2 \times l_1 + 1$

Au sein d'un motif périodique, il n'y a donc qu'un élément $\mathtt{1}$, à l'indice $d_1 + d_2 \times l_1 + 1 - d_3 = d_1 + d_2 \times l_1 + 1 - (d_1 + (d_2 - 1) \times l_1 + 1) = l_1$.

Retour sur la taille du préfixe : Afin que l'élément $\mathtt{1}$ soit situé en première position du motif périodique, on allonge donc le préfixe de $(l_1 - 1)$ éléments dans le cas où $d_2 \neq 0$:

$$d'_3 = |u_3| + (l_1 - 1) = \begin{cases} d_1 & \text{si } d_2 = 0 \\ d_1 + (d_2 - 1) \times l_1 + 1 + l_1 - 1 = d_1 + d_2 \times l_1 & \text{sinon} \end{cases}$$

Cette formule est équivalente à : $d'_3 = d_1 + d_2 \times l_1$. Notons que cet allongement ne rajoute que des $\mathtt{0}$ dans le préfixe car les $(l_1 - 1)$ premiers éléments du motif périodique sont des $\mathtt{0}$. On peut donc conclure que $p_{a_3} = \mathtt{0}^{d_1+d_2\times l_1}(\mathtt{10}^{l_1\times l_2-1})$

□

E.2 Preuve du calcul de $S_a(p_{a_1}, p_{a_2})$

Lemme E.2 (Calcul de $S_a(p_{a_1}, p_{a_2})$).
Soient $p_{a_1} = 0^{d_1}(10^{l_1-1})$ et $p_{a_2} = 0^{d_2}(10^{l_2-1})$.
Soit $S_a(p_{a_1}, p_{a_2})$ l'ensemble des horloges affines de $S(p_{a_1}, p_{a_2})$.
Si $l_2 \mid l_1$ et $d_1 \geq d_2 \times \frac{l_1}{l_2}$, alors :

$$S_a(p_{a_1}, p_{a_2}) = \left\{ 0^{d_1 - d_2 \times \frac{l_1}{l_2}} (10^{\frac{l_1}{l_2}-1}) \right\}$$

Si $l_2 \not| \, l_1$ ou $d_1 < d_2 \times \frac{l_1}{l_2}\ \frac{l_1}{l_2}$, alors $S_a(p_{a_1}, p_{a_2}) = \varnothing$.

Démonstration :

On cherche $S_a(p_{a_1}, p_{a_2}) = \{p \in S(p_{a_1}, p_{a_2}),\ p = 0^d(10^{l-1})\}$.

En appliquant la formule de $S(p_{a_1}, p_{a_2})$ sur les mots périodiques, on obtient : $\forall p \in S(u_1(v_1), u_2(v_2)), p = u(v)$ avec $|u|_1 = \max(0, d_2) = d_2$ et $|v|_1 = \text{ ppcm }(1, l_2) = l_2$.

Pour que p soit une horloge affine $0^d(10^{l-1})$, il faut que u soit de la forme $0^d.(10^{l-1})^x$ et v de la forme $(10^{l-1})^y$ avec x, y et d des entiers quelconques, et $l \neq 0$.

Si l'on prend en compte que $|u|_1 = d_2$ et $|v|_1 = l_2$, on obtient $u = 0^d.(10^{l-1})^{d_2}$ et $v = (10^{l-1})^{l_2}$.

En appliquant l'algorithme pour trouver les éléments des mots de $S(p_{a_1}, p_{a_2})$ (voir proposition 3.5 page 60), on obtient les informations suivantes :

- u est tel que $|u| = d_1$ et $|u|_1 = d_2$
- $v = 1.v'$ avec $|v'| = l_1 - 1$ et $|v'|_1 = l_2$

Par conséquent, on a $d + d_2 \times l = d_1$ et $l \times l_2 = l_1$, c'est-à-dire $l = \frac{l_1}{l_2}$ et $d = d_1 - d_2 \times \frac{l_1}{l_2}$.

On peut en déduire que :

$$\begin{aligned} S_a(p_{a_1}, p_{a_2}) &= \{p \in S(p_{a_1}, p_{a_2}),\ p = 0^d(10^{l-1})\} \\ &= \left\{p,\ p = 0^d(10^{l-1}) \text{ avec } l = \tfrac{l_1}{l_2} \text{ et } d = d_1 - d_2 \times \tfrac{l_1}{l_2}\right\}. \end{aligned}$$

Si $\frac{l_1}{l_2} \in \mathbb{N}^*$ et $d_1 - d_2 \times \frac{l_1}{l_2} \in \mathbb{N}$, alors $S_a(p_{a_1}, p_{a_2})$ contient l'unique élément $0^{d_1 - d_2 \times \frac{l_1}{l_2}}(10^{\frac{l_1}{l_2}-1})$. Comme $l_1 \in \mathbb{N}^*$ et $l_2 \in \mathbb{N}^*$, c'est le cas si et seulement si $l_2 \mid l_1$ et $d_1 \geq d_2 \times \frac{l_1}{l_2}$. Dans le cas contraire, $S_a(p_{a_1}, p_{a_2})$ est vide. □

Annexe F

Résolution de contraintes linéaires avec Glpk

Les systèmes de contraintes sont exprimés au format Cplex LP [Cpl], et résolus avec la command `glpsol` de l'outil Glpk [Glp]. La section F.1 fournit les systèmes de la partie II et la section F.2 les systèmes de la partie III. Tous les fichiers ci-dessous ont été générés automatiquement par le typeur.

F.1 Systèmes de contraintes pour la résolution concrète

F.1.1 Résolution du système de contraintes de précédence

Le fichier `prece.lp` suivant contient le système de contraintes de précédence présenté dans la section 5.3.4 :

```
Minimize
I_c3_6 + I_c3_5 + I_c3_4 + I_c3_3 + I_c3_2 + I_c3_1 + I_c1_3 + I_c1_2 + I_c1_1 + I_c2_4 + I_c2_3 + I_c2_2 + I_c2_1

Subject To
1 I_c2_1 - 1 I_c2_2 <= -1
1 I_c2_2 - 1 I_c2_3 <= -1
1 I_c2_3 - 1 I_c2_4 <= -1
1 I_c1_1 - 1 I_c1_2 <= -1
1 I_c1_2 - 1 I_c1_3 <= -1
1 I_c3_1 - 1 I_c3_2 <= -1
1 I_c3_2 - 1 I_c3_3 <= -1
1 I_c3_3 - 1 I_c3_4 <= -1
1 I_c3_4 - 1 I_c3_5 <= -1
1 I_c3_5 - 1 I_c3_6 <= -1
1 I_c1_3 - 1 I_c2_3 <= 0
1 I_c1_2 - 1 I_c2_1 <= 0
1 I_c3_5 - 1 I_c2_4 <= 0
1 I_c3_2 - 1 I_c2_2 <= 0
1 I_c1_3 - 1 I_c3_6 <= 0
1 I_c1_2 - 1 I_c3_4 <= 0
1 I_c1_1 - 1 I_c3_2 <= 0

Bounds
1 <= I_c2_1 <= +inf
1 <= I_c2_2 <= +inf
1 <= I_c2_3 <= +inf
1 <= I_c2_4 <= +inf
1 <= I_c1_1 <= +inf
```

```
1 <= I_c1_2 <= +inf
1 <= I_c1_3 <= +inf
1 <= I_c3_1 <= +inf
1 <= I_c3_2 <= +inf
1 <= I_c3_3 <= +inf
1 <= I_c3_4 <= +inf
1 <= I_c3_5 <= +inf
1 <= I_c3_6 <= +inf

Integer
I_c3_6
I_c3_5
I_c3_4
I_c3_3
I_c3_2
I_c3_1
I_c1_3
I_c1_2
I_c1_1
I_c2_4
I_c2_3
I_c2_2
I_c2_1

end
```

Les dix premières contraintes sont les contraintes qui garantissent qu'au sein d'un mot, l'indice d'un `1` est strictement inférieur à l'indice du `1` suivant. Les contraintes suivantes sont les contraintes de précédence. Par exemple, `1 I_c1_3 - 1 I_c3_6 <= 0` est la contrainte $\mathcal{I}_{c_1}(3) \leq \mathcal{I}_{c_3}(6)$.
La commande `glpsol -o prece.out --cpxlp prece.lp` produit le fichier `prece.out` suivant :

```
Problem:
Rows:       17
Columns:    13 (13 integer, 0 binary)
Non-zeros:  34
Status:     INTEGER OPTIMAL
Objective:  obj = 41 (MINimum)

   No.   Row name        Activity     Lower bound   Upper bound
------ ------------    ------------- ------------- -------------
     1 r.5                        -1                          -1
     2 r.6                        -1                          -1
     3 r.7                        -1                          -1
     4 r.8                        -1                          -1
     5 r.9                        -1                          -1
     6 r.10                       -1                          -1
     7 r.11                       -1                          -1
     8 r.12                       -1                          -1
     9 r.13                       -1                          -1
    10 r.14                       -1                          -1
    11 r.15                       -1                           0
    12 r.16                        0                           0
    13 r.17                        0                           0
    14 r.18                       -1                           0
    15 r.19                       -3                           0
    16 r.20                       -2                           0
    17 r.21                       -1                           0

   No. Column name       Activity     Lower bound   Upper bound
------ ------------    ------------- ------------- -------------
     1 I_c3_6       *              6             1
     2 I_c3_5       *              5             1
```

```
     3 I_c3_4       *              4             1
     4 I_c3_3       *              3             1
     5 I_c3_2       *              2             1
     6 I_c3_1       *              1             1
     7 I_c1_3       *              3             1
     8 I_c1_2       *              2             1
     9 I_c1_1       *              1             1
    10 I_c2_4       *              5             1
    11 I_c2_3       *              4             1
    12 I_c2_2       *              3             1
    13 I_c2_1       *              2             1

Integer feasibility conditions:

INT.PE: max.abs.err. = 0.00e+00 on row 0
        max.rel.err. = 0.00e+00 on row 0
        High quality

INT.PB: max.abs.err. = 0.00e+00 on row 0
        max.rel.err. = 0.00e+00 on row 0
        High quality

End of output
```

La colonne `Activity` donne les valeurs trouvées pour les variables de la colonne `Column name`.

F.1.2 Résolution du système de contraintes d'ajustement

Le fichier `adjust.lp` suivant contient le système de contraintes d'ajustement présenté dans la section 5.3.5 :

```
Minimize
k1 + k1' + k2 + k2' + k3 + k3'

Subject To
3 k3' - 2 k3 = 0
3 k1 - 1 k1' = 0
1 k1' - 1 k3 = 0
2 k2' - 2 k2 = 0
2 k1 - 1 k2 = 0
1 k3' - 1 k2' = 0

Bounds
1 <= k1 <= +inf
1 <= k1' <= +inf
1 <= k2 <= +inf
1 <= k2' <= +inf
1 <= k3 <= +inf
1 <= k3' <= +inf

Integer
k1
k1'
k2
k2'
k3
k3'

end
```

La commande `glpsol -o adjust.out --cpxlp adjust.lp` produit le fichier `adjust.out` suivant :

```
Problem:
Rows:       6
Columns:    6 (6 integer, 0 binary)
Non-zeros:  12
Status:     INTEGER OPTIMAL
Objective:  obj = 13 (MINimum)

   No.   Row name        Activity     Lower bound   Upper bound
------ ------------    ------------- ------------- -------------
     1 r.5                        0             0             =
     2 r.6                        0             0             =
     3 r.7                        0             0             =
     4 r.8                        0             0             =
     5 r.9                        0             0             =
     6 r.10                       0             0             =

   No. Column name       Activity     Lower bound   Upper bound
------ ------------    ------------- ------------- -------------
     1 k1           *             1             1
     2 k1'          *             3             1
     3 k2           *             2             1
     4 k2'          *             2             1
     5 k3           *             3             1
     6 k3'          *             2             1

Integer feasibility conditions:

INT.PE: max.abs.err. = 0.00e+00 on row 0
        max.rel.err. = 0.00e+00 on row 0
        High quality

INT.PB: max.abs.err. = 0.00e+00 on row 0
        max.rel.err. = 0.00e+00 on row 0
        High quality

End of output
```

F.2 Système de contraintes pour la résolution abstraite

Le fichier `encode_with_3_buffers.lp` contient le système de contraintes de précédence abstraites du nœud `encode_with_3_buffers` étudié dans la section 9.2.2.

```
Maximize
k1_h15

Subject To
1 k0_h15 - 1 k1_h15 <= 0
1 k0_h9 - 1 k1_h9 <= -113
65 k1_h15 - 0 k0_h9 <= -20369
65 k1_h15 - 1 k0_h9 <= -20369
325 k1_h15 - 0 k0_h9 <= -140063
325 k1_h15 - 5 k0_h9 <= -140063
715 k1_h15 - 0 k0_h9 <= -123736
715 k1_h15 - 11 k0_h9 <= -123736

Bounds
-inf <= k1_h15 <= 0
-inf <= k1_h9 <= +inf
-inf <= k0_h15 <= 0
-inf <= k0_h9 <= 0

Integer
```

```
k1_h15
k1_h9
k0_h15
k0_h9

end
```

La commande `glpsol -o encode_with_3_buffers.out --cpxlp encode_with_3_buffers.lp` produit le fichier `encode_with_3_buffers.out` qui contient la solution suivante :

```
Problem:
Rows:       8
Columns:    4 (4 integer, 0 binary)
Non-zeros:  13
Status:     INTEGER OPTIMAL
Objective:  obj = -431 (MAXimum)

   No.   Row name        Activity     Lower bound   Upper bound
------ ------------    ------------- ------------- -------------
     1 r.5                        0                           0
     2 r.6                     -113                        -113
     3 r.7                   -28015                      -20369
     4 r.8                   -28015                      -20369
     5 r.9                  -140075                     -140063
     6 r.10                 -140075                     -140063
     7 r.11                 -308165                     -123736
     8 r.12                 -308165                     -123736

   No. Column name       Activity     Lower bound   Upper bound
------ ------------    ------------- ------------- -------------
     1 k1_h15       *          -431                           0
     2 k0_h15       *          -431                           0
     3 k0_h9        *             0                           0
     4 k1_h9        *           113

Integer feasibility conditions:

INT.PE: max.abs.err. = 0.00e+00 on row 0
        max.rel.err. = 0.00e+00 on row 0
        High quality

INT.PB: max.abs.err. = 0.00e+00 on row 0
        max.rel.err. = 0.00e+00 on row 0
        High quality

End of output
```

La colonne `Activity` donne les valeurs trouvées pour les variables de la colonne `Column name`.

Annexe G

Générer des horloges dans une enveloppe grâce au compilateur Lustre

Pascal Raymond nous a montré comment générer des horloges dans une enveloppe en utilisant le mécanisme d'assertions du compilateur Lustre.

Le rôle premier des assertions en Lustre est de fournir des informations permettant au compilateur d'optimiser le code généré [HCRP91, HR99]. Les assertions étant considérées comme des hypothèses, les transitions invalidant une assertion sont supprimées de l'automate correspondant à la compilation du code source, car elles sont supposées superflues.

Une utilisation détournée de cette compilation optimisante permet de forcer certaines propriétés à être vérifiées, comme ici la propriété pour un mot binaire d'appartenir à une enveloppe.

Le code est spécialisé pour l'enveloppe $a_2 = \left\langle \frac{k^0}{\ell}, \frac{k^1}{\ell} \right\rangle \left(\frac{n}{\ell}\right) = \left\langle -\frac{9}{5}, -\frac{3}{5} \right\rangle \left(\frac{3}{5}\right)$:

```
const k0 = -9; const k1 = -3; const n = 3; const l = 5;
```

On reconnaît ci-dessous la fonction de normalisation des nœuds de l'automate associé à a_2 et la fonction qui vérifie que son entrée appartient à a_2.

```
node norm(i, j : int) returns (new_i, new_j: int);
let
  (new_i,new_j) = if ((i > l) and (j >= n)) then (i-l, j-n) else (i,j);
tel

node check(w : bool) returns (ok: bool);
var
   i,j,new_i,new_j : int;
let
  -- l'automate :
  i = 1 -> pre new_i;
  j = 0 -> pre new_j;
  (new_i,new_j) = norm (i+1, if w then (j+1) else j);
```

```
   -- la transition est valide si :
  ok =
    if w then new_j * l <= n * new_i + k1
    else new_j * l >= n * i + k0;
tel
```

Le code Lustre ci-dessus est analogue au code Lucid Synchrone présenté section 7.2.4 page 153.

Pour générer une horloge dans a_2, on se contente de recopier une entrée booléenne quelconque sur la sortie, en prenant soin de poser l'assertion que l'entrée est dans l'enveloppe a_2, à l'aide du nœud de vérification `check` :

```
node generate(x : bool) returns (w: bool);
let
  -- on se contente de recopier l'entree sur la sortie :
  w = x;
  -- mais on contraint x a etre correct !!
  assert check(x);
tel
```

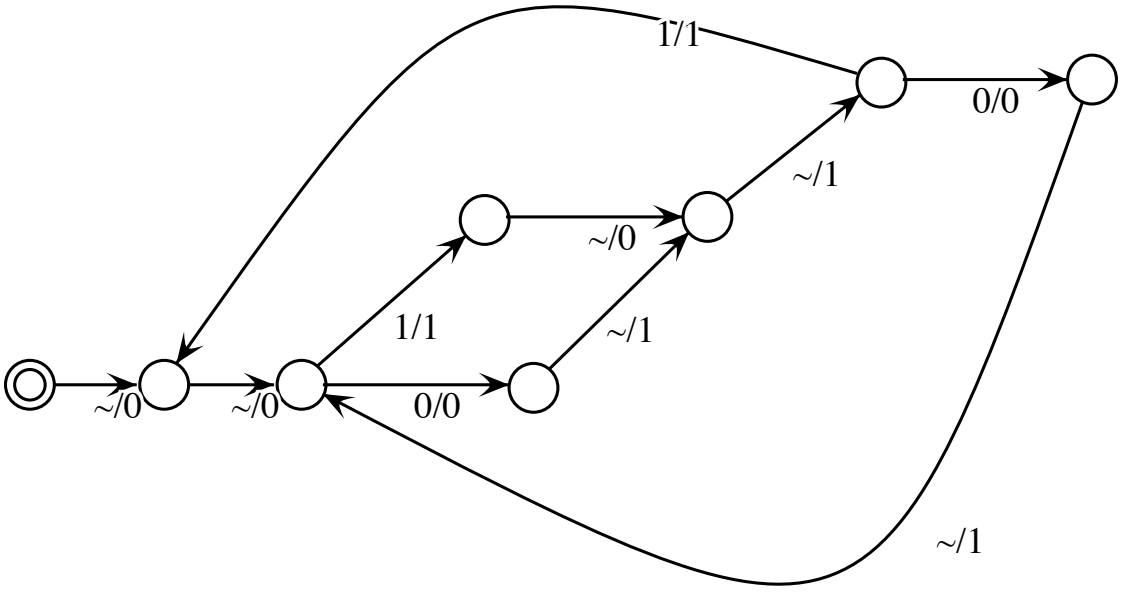
Le compilateur utilise les assertions pour simplifier le code généré. Les assertions étant considérées comme des hypothèses toujours vérifiées, les transitions invalidant une assertion sont supprimées de l'automate correspondant à la compilation du code source, car elles sont supposées superflues.

Par conséquent, si à un instant donné, `check(x);` n'est vrai que si `x` vaut *true* (resp. $false$), le compilateur donne la valeur *true* (resp. $false$) à `w`, sans même vérifier que l'entrée `x` prend effectivement cette valeur. En quelque sorte, il "force" les assertions à être vérifiées. La valeur du flot d'entrée n'est donc utilisée que lorsque l'assertion est vérifiée quelle que soit la valeur de `x`. Quand il n'y a pas le choix, c'est la seule valeur satisfaisant l'assertion qui est choisie et la valeur de l'entrée n'est pas consultée. Ainsi, pour générer une horloge qui vaut *true* dès que possible, on donne en paramètre un flot qui vaut toujours *true*, et pour générer une horloge qui vaut $false$ dès que possible, on donne en paramètre un flot qui vaut toujours $false$:

```
node early() returns (w: bool);
let
  w = generate(true);
tel

node late() returns (w: bool);
let
  w = generate(false);
tel
```

On peut générer une horloge aléatoire en donnant en paramètre un flot aléatoire. Le com-

pilateur Lustre calcule l'automate suivant pour générer les valeurs de sortie de `generate` en fonction de son entrée et en tirant profit de son assertion :



Le label $\sim/0$ sur la première transition signifie que l'entrée n'est pas consultée ($\sim$) et que la sortie vaut 0. L'entrée a donc été considérée comme valant 0, seule valeur satisfaisant l'assertion. La deuxième transition suit le même principe. Pour la troisième transition, l'assertion est toujours vérifiée, donc si l'entrée vaut 1, elle est recopiée sur la sortie (1/1) et si elle vaut 0, elle est aussi recopiée sur la sortie (0/0), comme exprimé dans le code du programme.

On peut constater que l'automate donnant les sorties de `generate` correspond exactement à l'automate associé à a_2, représenté sur la figure 7.14 page 153. `generate` ne produit donc que des mots appartenant à l'enveloppe a_2.

Annexe H

Encodeur et décodeur de canal GSM

Nous fournissons ici le code de l'encodeur et du décodeur de canal GSM. Le code de l'encodeur est celui qui a été utilisé pour les chapitres 6 et 9. Le code du décodeur est celui utilisé dans le chapitre 9, c'est-à-dire pour illustrer les résultats de la résolution abstraite. Il est très légèrement différent de celui utilisé pour le chapitre 6 sur l'application de la résolution concrète : le nœud `cyclic_decoding` a été adapté pour compenser les faiblesses de l'unification structurelle du typage sur les horloges abstraites. Le code du nœud utilisé pour illustrer le typage sur les horloges périodiques est quant à lui fourni dans le chapitre 6.

H.1 Encodeur de canal GSM

```
1  (* D'apres le code Lucid Synchrone de Gwenael Delaval *)
2
3  (************************ ENCODEUR GSM AVEC BUFFERS ****************************)
4  (*******************************************************************************)
5  (* compiler avec les options -dcstrs -dbuff
6     pour voir les contraintes a resoudre et la taille des buffers necessaires
7  *)
8
9  (* xor *)
10 let node xor (a, b) = (a && (not b)) || (b && (not a))
11
12 (*************** split_speech ***************)
13
14 let node split_speech x = (x_Ia, x_Ib, x_II) where
15   rec x_Ia = x when (1^50 0^210)
16   and x_Ib = x when (0^50 1^132 0^78)
17   and x_II = x when (0^182 1^78)
18
19 (************* cyclic_encoding **************)
20
21 (* fusion des donnees et de la redondance *)
22 let node join_50_3 ( (x(* ::'a on (1^50 0^3) *)) , r0, r1, r2) = x012 where
23   rec x0 = merge (1^50 0) x r0
```

```
24    and x01 = merge (1^51 0) x0 r1
25    and x012 = merge (1^52 0) x01 r2
26
27  (* encodeur cyclique : x^3 + x + 1 *)
28  let node cyclic_encoding x = y where
29    rec u0 = false fby (xor(x, u2))
30    and u1 = false fby (xor(u0, u2))
31    and u2 = false fby u1
32    and y = join_50_3
33        ( x,
34          buffer(u0 when (0^49 1)),
35          buffer(u1 when (0^49 1)),
36          buffer(u2 when (0^49 1)) )
37
38  (********** convolutional_encoding **********)
39
40  (* pour fusionner les deux flots de redondance *)
41  let node multiplexer (x1, x2) = o where
42    rec o = merge (10) x1 x2
43
44  (* encodeur convolutionnel : x^4 + x^3 + 1 and x^4 + x^3 + x + 1 *)
45  let node convolutional_encoding x = y where
46    rec x' = merge (1^185 0^4) x false
47    and u1 = false fby x'
48    and u2 = false fby u1
49    and u3 = false fby u2
50    and u4 = false fby u3
51    and x1 = xor (xor (xor (x', u1), u3), u4) (* x^4 + x^3 + x + 1 *)
52    and x2 = xor (xor (x', u3), u4) (* x^4 + x^3 + 1 *)
53    and y = multiplexer (x1, buffer(x2))
54
55  (************** gsm_encoding ****************)
56
57  (* sans buffers *)
58  (* ne fonctionne qu'avec la methode concrete avec unification globale *)
59  (*
60  let node encode_without_buffer x = y where
61    rec (x_Ia, x_Ib, x_II) = split_speech x
62    and y1 = cyclic_encoding x_Ia
63    and y1_x_Ib = merge (1^53 0^132) y1 x_Ib
64    and y2 = convolutional_encoding y1_x_Ib
65    and y = merge (1^378 0^78) y2 x_II
66  *)
67
68  (* avec buffers *)
69  let node encode_with_3_buffers x = y where
70    rec (x_Ia, x_Ib, x_II) = split_speech x
71    and y1 = cyclic_encoding (buffer(x_Ia))
72    and y1_x_Ib = merge (1^53 0^132) y1 (buffer(x_Ib))
73    and y2 = convolutional_encoding y1_x_Ib
74    and y = merge (1^378 0^78) y2 (buffer(x_II))
```

H.2 Décodeur de canal GSM

```
(* D'apres le code Lucid Synchrone de Gwenael Delaval *)

(********************** DECODEUR GSM AVEC BUFFERS ***************************)
(*****************************************************************************)

open Gsm_encoding

(************* cyclic_decoding **************)

(* stutter50 *)
let node stutter x =
  merge (10) x (buffer(x))

let node stutter16 x =
  stutter (stutter (stutter (stutter x)))

let node stutter32 x =
  stutter (stutter (stutter (stutter (stutter x))))

let node stutter48 x =
  merge (1^32 0^16) (stutter32 x) (buffer(stutter16 x))

let node stutter49 x =
 merge (1^48 0) (stutter48 x) (buffer(x))

let node stutter50 x =
  merge (1^49 0) (stutter49 x) (buffer(x))

let node split3 x = x when (100), x when (010), x when (001)

let node and_by_3 x = y where
  rec x1, x2, x3 = split3 x
  and y = ((buffer ((buffer x1) && x2)) && x3)

let node cyclic_decoding x = o where
  rec (data, redondancy) =
    (x when (1^52 0) when (1^51 0) when (1^50 0) , x whenot (1^50 0^3))
  and reencoded_data = cyclic_encoding data
  and code = reencoded_data whenot (1^50 0^3)
  and ok = (code = redondancy)
  and frame_ok = and_by_3 ok
  and data_ok = (stutter50 (buffer(frame_ok)))
  and current_frame = buffer(data)
  and last_frame = merge 1^50(0) true (buffer(current_frame))
  and o = if data_ok then current_frame else last_frame

(********** convolutional_decoding **********)
(* distance de hamming *)
let node hamming_dist ((x1,x2),(y1,y2)) =
  (if x1 = y1 then 0 else 1) + (if x2 = y2 then 0 else 1)
```

```
51
52  let node convolutional_decoding x = (* non implemente *)
53    (x when (01)) when 0^188(1^185 0^4)
54  (* code factice, avec le bon type de sortie *)
55
56  (*************** join_speech ****************)
57  let node join_speech (x_Ia, x_Ib, x_II) = x where
58    rec x = merge (1^50 0^210) x_Ia (merge (1^132 0^78) x_Ib x_II)
59
60  (*********** split encoded speech ***********)
61  let node split_I_II x = (x when (1^378 0^78), x whenot (1^378 0^78))
62  let node split_Ia_Ib x_I = (x_I when (1^53 0^132), x_I whenot (1^53 0^132))
63
64  (*********** GSM speech decoding ************)
65
66  (* les buffers sont indispensables si l'on utilise la resolution abstraite :
67     le typage echoue sur la version sans buffers *)
68
69  (* version retenue *)
70  (* l'ajustement des prefixes echoue dans la resolution concrete *)
71  let node decode_with_3_buffers x = y where
72    rec x_I_enc, x_II = split_I_II x
73    and x_I = convolutional_decoding x_I_enc
74    and x_Ia_enc, x_Ib = split_Ia_Ib x_I
75    and x_Ia = cyclic_decoding x_Ia_enc
76    and y = join_speech (buffer(x_Ia), buffer(x_Ib), buffer(x_II))
77
78
79  (***************** identity *****************)
80  let node id_speech x = y where
81    rec y = decode_with_3_buffers(encode_with_3_buffers(x))
```

Annexe I

Règles d'unification des types d'horloges affines en Signal

Les relations affines sur les horloges du langage Signal ont été expliquées dans la section 10.1. Un ensemble de règles est présenté dans [Sma98] pour établir que deux types d'horloges affines sont égaux ou peuvent être rendus égaux (noté $ck_1 \uplus ck_2$). Ces règles sont nommées *règles de synchronisabilité*, à ne pas confondre avec la notion de synchronisabilité utilisée dans ce document. Nous présentons ici un parallèle entre ces règles et la résolution des contraintes d'égalité sur les types d'horloges affines dans notre cadre.

Les hypothèses des règles sont présentées sous deux formes, de la manière suivante :

Horloges affines en Signal	Horloges affines en Lucy-n

Règle 1.

Si	$ck\ \mathcal{R}_{(1,d_1,\ell_1)}\ ck_1$	$\mid$	$ck_1 = ck$ `on` $0^{d_1}(10^{\ell_1-1})$
	$ck\ \mathcal{R}_{(1,d_2,\ell_2)}\ ck_2$	$\mid$	$ck_2 = ck$ `on` $0^{d_2}(10^{\ell_2-1})$
Alors	$ck_1 \uplus ck_2 \Leftrightarrow d_1 = d_2 \wedge \ell_1 = \ell_2$		

Dans notre cadre, comme ck_1 et ck_2 sont tous les deux exprimés en fonction de ck, le test d'égalité est réduit au test d'égalité des mots affines $0^{d_1}(10^{\ell_1-1})$ et $0^{d_2}(10^{\ell_2-1})$, qui est identique à la condition donnée dans la règle ci-dessus.

Règle 2.

Si	$ck\ \mathcal{R}_{(1,0,1)}\ ck_1$	$\mid$	$ck_1 = ck$ `on (1)`
Alors	$ck \uplus ck_1 \Leftrightarrow ck = ck_1$		

Dans notre cadre, cela correspond au fait que `(1)` est neutre pour l'opérateur *on*.

Règle 3.

Si	$ck\ \mathcal{R}_{(1,d_1,\ell_1)}\ ck_1$	$\mid$	$ck_1 = ck$ `on` $0^{d_1}(10^{\ell_1-1})$
	$ck_1\ \mathcal{R}_{(1,d_2,\ell_2)}\ ck_2$	$\mid$	$ck_2 = ck_1$ `on` $0^{d_2}(10^{\ell_2-1})$
	$ck\ \mathcal{R}_{(1,d_3,\ell_3)}\ ck_3$	$\mid$	$ck_3 = ck$ `on` $0^{d_3}(10^{\ell_3-1})$
Alors	$ck_2 \uplus ck_3 \Leftrightarrow (d_3 = d_1 + \ell_1 \times d_2) \wedge (\ell_3 = \ell_1 \times \ell_2)$		

Dans notre cadre, cette règle correspond au test d'égalité sur les horloges affines résultat d'une opération *on* :

$$\begin{aligned} ck_2 &= (ck \text{ on } 0^{d_1}(10^{\ell_1-1})) \text{ on } 0^{d_2}(10^{\ell_2-1}) \\ &= ck \text{ on } (0^{d_1}(10^{\ell_1-1}) \text{ on } 0^{d_2}(10^{\ell_2-1})) \\ &= ck \text{ on } 0^{d_1+d_2\times\ell_1}(10^{\ell_1\times\ell_2-1}) \end{aligned}$$

Donc $ck_2 \equiv ck_3 \Leftrightarrow 0^{d_1+d_2\times\ell_1}(10^{\ell_1\times\ell_2-1}) = 0^{d_3}(10^{\ell_3-1}) \Leftrightarrow (d_1+d_2\times\ell_1 = d_3)\wedge(\ell_1\times\ell_2 = \ell_3)$.
On peut remarquer que la condition est identique à celle de la règle ci-dessus.

Règles 4 à 6. Si $ck\ \mathcal{R}_{(1,d_1,\ell_1)}\ ck_1$ | $ck_1 = ck$ on $0^{d_1}(10^{\ell_1-1})$
$ck\ \mathcal{R}_{(1,d_2,\ell_2)}\ ck_2$ | $ck_2 = ck$ on $0^{d_2}(10^{\ell_2-1})$
$ck\ \mathcal{R}_{(1,d_3,\ell_3)}\ ck_3$ | $ck_3 = ck$ on $0^{d_3}(10^{\ell_3-1})$
Alors $ck_1\ op\ ck_2 = ck_3 \Leftrightarrow d_1, \ell_1$ et d_2, l_2 vérifient les conditions pour que $ck_1\ op\ ck_2$ soit affine, et d_3, ℓ_3 sont égaux aux valeurs attendues. L'opération *op* peut être une conjonction (règle 4.), une disjonction (règle 5.) ou une différence (règle 6.). Ces conditions sont définies formellement dans [Sma98].

Nous n'avons pas traité la question de savoir si la composition de deux horloges affines par les opérateurs de conjonction, de disjonction et de différence est une horloge affine. Les résultats de [Sma98] sont directement applicables à notre cas. Nous avons cependant mentionné dans le chapitre 5 qu'il est possible de calculer les mots périodiques résultats de telles opérations sur les mots périodiques. Il nous suffit donc de calculer l'horloge ultimement périodique résultat de l'opération et vérifier qu'elle est égale à l'horloge affine de ck_3 :

ck on $(0^{d_1}(10^{\ell_1-1})\ op\ 0^{d_2}(10^{\ell_2-1})) \equiv ck$ on $0^{d_3}(10^{\ell_3-1})$
$\Leftrightarrow\ 0^{d_1}(10^{\ell_1-1})\ op\ 0^{d_2}(10^{\ell_2-1}) = 0^{d_3}(10^{\ell_3-1})$

Règle 7. Si $ck\ \mathcal{R}_{(1,d_{1_i},\ell)}\ ck_{1_i},\quad i = 1,\ldots,p$ | $ck_{1_i} = ck$ on $0^{d_{1_i}}(10^{\ell-1})$
$ck\ \mathcal{R}_{(1,d_{2_j},\ell)}\ ck_{2_j},\quad j = 1,\ldots,q$ | $ck_{2_i} = ck$ on $0^{d_{2_i}}(10^{\ell-1})$
Alors $(\vee_{i=1\ldots p}\ ck_{1_i}) \ominus (\vee_{j=1\ldots q}\ ck_{2_j}) \Leftrightarrow \{d_{1_1},\ldots,d_{1_p}\} = \{d_{2_1},\ldots,d_{2_q}\}$

Cette règle est utilisée pour comparer des *formes normales relatives*. Comme l'ensemble des horloges affines n'est pas fermé par union, intersection et différence, il est proposé dans [Sma98] de mettre les expressions d'horloges en *forme normale relative*, c'est-à-dire une disjonction d'horloges affines dont le motif périodique est de même taille. Par exemple les formes normales relatives de l'ensemble d'horloges {0(10), 0(10), 00(100)} sont :

$0(10) = 0(10^5) \vee 0^3(10^5) \vee 0^5(10^5)$
$0(100) = 0(10^5) \vee 0^4(10^5)$
$00(100) = 0^2(10^5) \vee 0^5(10^5)$

Donc $0(10) \wedge (0(100) \vee 00(100)) = 0(10^5) \vee 0^5(10^5)$.
Ces disjonctions constituent une forme normale que l'on peut comparer pour tester l´égalité. Dans notre cadre, il suffit de calculer le mot utlimement périodique non nécessairement affine résultat de l'opération, et d'utiliser le test d'égalité sur les mots périodiques. Par exemple : $0(10) \wedge (0(100) \vee 00(100)) = 0(100001)$ et on sait comparer $0(100001)$ à un autre mot périodique. Dans la règle, le test d'égalité des ensembles de d_{1_i} et de d_{2_j} correspond à la comparaison des indices des 1 dans notre test d'égalité sur les périodiques, et le fait que les deux motifs périodiques doivent être de même longueur se retrouve aussi dans notre test d'égalité (même si nous ne les mettons pas à la même taille de manière explicite).

Règle 8. Si $ck_1\ \mathcal{R}_{(l_1,d_2,l_2)}\ ck_2$ | $ck_1 = \alpha$ on (10^{ℓ_1-1}) $ck_2 = \alpha$ on $0^{d_2}(10^{\ell_2-1})$
$ck'_1\ \mathcal{R}_{(l'_1,d'_2,l'_2)}\ ck'_2$ | $ck'_1 = \alpha'$ on $(10^{\ell'_1-1})$ $ck'_2 = \alpha'$ on $0^{d'_2}(10^{\ell'_2-1})$
et il n'y a pas de contrainte entre ck_1 et ck'_1
Alors $ck_2 \ominus ck'_2$

Dans notre cadre, cela correspond au fait que si deux types ne dépendent pas de la même variable de type, et qu'il n'y a aucune contrainte entre ces variables de types (en tenant compte des contraintes par transitivité), alors ces deux types sont unifiables et leur unification ne va pas créer d'echec dans la résolution du reste du système de contraintes d'égalité de types.

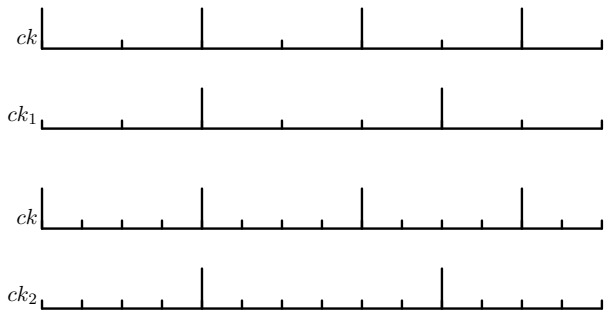
La règle 8' est la généralisation de la règle 8. au cas où ck_2 et ck'_2 sont chacune en relation avec plusieurs autres types (respectivement ck_{1_i} et ck'_{1_j}) et qu'aucun des types ck_{1_i} n'est relié par une contrainte à un type ck'_{1_j}.

Règle 9. Si $ck\ \mathcal{R}_{(\ell,d_1,\ell_1)}\ ck_1$ | $ck = \alpha\ \texttt{on}\ (10^{\ell-1})$ $ck_1 = \alpha\ \texttt{on}\ 0^{d_1}(10^{\ell_1-1})$
$ck\ \mathcal{R}_{(\ell',d_2,\ell_2)}\ ck_2$ | $ck = \alpha'\ \texttt{on}\ (10^{\ell'-1})$ $ck_2 = \alpha'\ \texttt{on}\ 0^{d_2}(10^{\ell_2-1})$
Alors $ck_1 \ominus ck_2 \Leftrightarrow (\ell, d_1, \ell_1) \sim (\ell', d_2, \ell_2)$

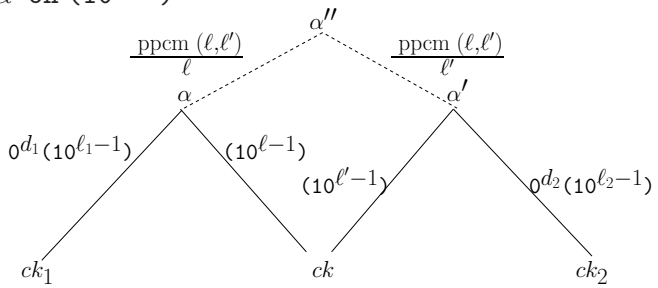
La relation $\sim$ est une équivalence entre les relations affines. Lorsque $g = \text{pgcd}(\ell, \ell_1)$ divise d_1 et $g' = \text{pgcd}(\ell', \ell_2)$ divise d_2, elle est définie ainsi :

$$(\ell, d_1, \ell_1) \sim (\ell', d_2, \ell_2) \Leftrightarrow (\frac{\ell}{g}, \frac{d_1}{g}, \frac{\ell_1}{g}) = (\frac{\ell'}{g'}, \frac{d_2}{g'}, \frac{\ell_2}{g'})$$

Par exemple ci-dessous, $ck\ \mathcal{R}_{(2,2,3)}\ ck_1$ et $ck\ \mathcal{R}_{(4,4,6)}\ ck_2$. Comme $(2,2,3) \sim (4,4,6)$, on peut en conclure que $ck_1 \ominus ck_2$.



Dans notre cadre, on doit commencer par unifier les deux contraintes sur ck :
$ck = \alpha\ \texttt{on}\ (10^{\ell-1}) = \alpha'\ \texttt{on}\ (10^{\ell'-1})$



La valeur inférée pour ck est $ck = \alpha''\ \texttt{on}\ (10^{\ \text{ppcm}\ (\ell,\ell')-1})$ et l'instanciation des variables α et α' pour s'y ramener est :

$$\alpha \leftarrow \alpha''\ \texttt{on}\ (10^{\frac{\text{ppcm}\ (\ell,\ell')}{\ell}-1})$$
$$\alpha' \leftarrow \alpha''\ \texttt{on}\ (10^{\frac{\text{ppcm}\ (\ell,\ell')}{\ell'}-1})$$

Si l'on applique cette substitution à ck_1 et ck_2, on obtient :

$$ck_1 = \alpha''\ \texttt{on}\ 10^{\frac{\text{ppcm}\ (\ell,\ell')}{\ell}}\ \texttt{on}\ 0^{d_1}(10^{\ell_1-1}) = \alpha''\ \texttt{on}\ 0^{d_1\times\frac{\text{ppcm}\ (\ell,\ell')}{\ell}}(10^{\frac{\text{ppcm}\ (\ell,\ell')}{\ell}\times\ell_1-1})$$
$$ck_2 = \alpha''\ \texttt{on}\ 0^{d_2\times\frac{\text{ppcm}\ (\ell,\ell')}{\ell'}}(10^{\frac{\text{ppcm}\ (\ell,\ell')}{\ell'}\times\ell_2-1}) = \alpha''\ \texttt{on}\ 0^{d_2\times\frac{\text{ppcm}\ (\ell,\ell')}{\ell'}}(10^{\frac{\text{ppcm}\ (\ell,\ell')}{\ell'}\times\ell_2-1})$$

Par conséquent, comme ils sont exprimés en fonction de la même variable, les types ck_1 et ck_2 sont unifiables si et seulement s'ils sont égaux, c'est-à-dire si :

$$d_1 \times \frac{\text{ppcm}\ (\ell, \ell')}{\ell} = d_2 \times \frac{\text{ppcm}\ (\ell, \ell')}{\ell'} \quad \text{et} \quad \frac{\text{ppcm}\ (\ell, \ell')}{\ell} \times \ell_1 = \frac{\text{ppcm}\ (\ell, \ell')}{\ell'} \times \ell_2$$

donc si $\frac{d_1}{\ell} = \frac{d_2}{\ell'}$ et $\frac{\ell_1}{\ell} = \frac{\ell_2}{\ell'}$. Cette condition est équivalente à $\frac{d_1}{d_2} = \frac{\ell}{\ell'} = \frac{\ell_1}{\ell_2}$ et donc à la condition $(\ell, d_1, \ell_1) \sim (\ell', d_2, \ell_2)$ définie plus haut.

La règle 9' est une généralisation au cas où ck_1 et ck_2 sont chacune en relation avec plusieurs autres types.

Règle 10. Si $ck\ \mathcal{R}_{(\ell,d_1,\ell_1)}\ ck_1$ | $ck = \alpha\ \texttt{on}\ (10^{\ell}-1)$ $\quad ck_1 = \alpha\ \texttt{on}\ 0^{d_1}(10^{\ell_1}-1)$

$ck_1\ \mathcal{R}_{(\ell'_1,d_2,\ell_2)}\ ck_2$ | $ck_1 = \alpha'\ \texttt{on}\ (10^{\ell'_1}-1)$ $\quad ck_2 = \alpha'\ \texttt{on}\ 0^{d_2}(10^{\ell_2}-1)$

$ck\ \mathcal{R}_{(\ell',d_3,\ell_3)}\ ck_3$ | $ck = \alpha''\ \texttt{on}\ (10^{\ell'}-1)$ $\quad ck_3 = \alpha''\ \texttt{on}\ 0^{d_3}(10^{\ell_3}-1)$

Alors $ck_2 \bigoplus ck_3 \Leftrightarrow \exists (\ell_s, d_{1_s}, \ell_{1_s}), (\ell'_{1_s}, d_{2_s}, \ell_{2_s})$ tels que $\ell_{1_s} = \ell'_{1_s}$ et :

$(\ell_s, d_{1_s}, \ell_{1_s}) \sim (\ell, d_1, \ell_1)$

$(\ell'_{1_s}, d_{2_s}, \ell_{2_s}) \sim (\ell'_1, d_2, \ell_2)$

$(\ell_s \times \ell'_{1_s}, \ell'_{1_s} \times d_{1_s} + \ell_{1_s} \times d_{2_s}, \ell_{1_s} \times \ell_{2_s}) \sim (\ell', d_3, \ell_3)$

Cette règle traite le cas où un type d'horloges ck est d'une part en relation affine avec un type ck_3, et d'autre part en relation avec un type ck_1 lui même en relation avec un type ck_2. Les types ck_3 et ck_2 sont unifiables si leurs relations respectives avec ck ne sont pas contradictoires. Il faut trouver les paramètres d'une relation affine liant ck à ck_2, à partir de la relation liant ck à ck_1 et de la relation liant ck_1 à ck_2 (on peut ensuite utiliser la règle 9.). Or on ne sait calculer une telle relation que si $l_1 = l'_1$, avec la formule suivante :

$$ck\ \mathcal{R}_{(\ell,d_1,\ell_1)}\ ck_1 \text{ et } ck_1\ \mathcal{R}_{(\ell'_1,d_2,\ell_2)}\ ck_2 \text{ avec } \ell_1 = \ell'_1 \Rightarrow ck\ \mathcal{R}_{(\ell'',d'_2,\ell'_2)}\ ck_2$$

avec $\ell'' = \ell \times \ell'_1$, $d'_2 = \ell'_1 \times d_1 + \ell_1 \times d_2$ et $\ell'_2 = \ell_1 \times \ell_2$.

Pour pouvoir appliquer ce calcul, il faut donc trouver une relation $(\ell_s, d_{1_s}, \ell_{1_s})$ équivalente à (ℓ, d_1, ℓ_1) et une relation $(\ell'_{1_s}, d_{2_s}, \ell_{2_s})$ équivalente à (ℓ'_1, d_2, ℓ_2), telles que $\ell_{1_s} = \ell'_{1_s}$. Trouver de telles relations est un problème NP-complet. Des heuristiques sont utilisées pour le résoudre de manière incomplète.

Dans notre cadre, cette règle correspond à deux unifications, en suivant le même principe que pour la règle 9. On commence par exemple par unifier les deux valeurs de ck_1, puis on applique la substitution résultat aux définitions des autres horloges. On unifie ensuite les deux valeurs de ck et on applique la substitution obtenue. Toutes les définitions sont alors exprimées en fonction d'une même variable d'horloges. Par conséquent, ck_2 et ck_3 sont unifiables si et seulement si elles sont égales.

La règle 10'. est une généralisation de la règle 10.