Programmer le parallélisme avec des futures en Heptagon un langage synchrone flot de données et étude des réseaux de Kahn en vue d'une compilation synchrone

Léonard Gérard

Université Paris Sud
École Doctorale d'informatique
Équipe Parkas, Inria / ENS

Thèse de doctorat
discipline informatique
soutenue le 25 septembre 2013 par

Léonard Gérard

Programmer le parallélisme avec des *futures* en Heptagon un langage synchrone flot de données et étude des réseaux de Kahn en vue d'une compilation synchrone

Directeur de thèse	Marc Pouzet	Prof. Université Paris 6
Rapporteurs	Alain Girault	D.R. Inria Rhône-Alpes
	Dumitru Potop-Butucaru	C.R. Inria Rocquencourt
	Yves Sorel	D.R. Inria Rocquencourt
Examinateurs	Albert Cohen	D.R. Inria Rocquencourt
	Sylvain Conchon	Prof. Université Paris 11

À ceux qui y ont cru,
ceux qui m'ont soutenu,
qui l'ont lue,
la liront,
à ceux qui l'utiliseront.

Introduction

Les langages synchrones ont été fondés dans les années 1980 pour modéliser, concevoir et implémenter les systèmes réactifs temps réel critiques. Avec la complexité toujours croissante des systèmes contrôlés, la vitesse d'exécution devient un critère important. Dans le même temps, les processeurs gagnent plus en nombre de cœurs qu'en vitesse. Nous sommes donc à la recherche d'une exécution parallèle, combinant efficacité et sûreté.

Les langages synchrones ont toujours intégré la notion de parallélisme pour l'expressivité de la modélisation. Leurs compilations visent principalement les circuits ou la génération de code séquentiel. Tous ont une sémantique formelle qui rend possible la distribution correcte du code. Mais la préservation de cette sémantique peut être un obstacle à l'efficacité du code généré, particulièrement s'il est nécessaire de préserver une notion d'instant global au système.

Le modèle sémantique qui nous intéresse est celui, fonctionnel flot de données, des réseaux de Kahn [Kah74]. Ces réseaux modélisent des calculateurs distribués, communiquant au travers de files de taille non bornée. Dans ce cadre, la distribution ne demande aucune synchronisation ni information extérieur au modèle. En considérant l'histoire des files de communication, la sémantique de Kahn permet de s'abstraire de l'exécution effective, tout en garantissant le déterminisme fonctionnel du calcul.

Parmi les trois langages synchrones originaux, seul Esterel [BC84] est impératif et non flots de données. Le langage Signal [Le +91], orienté modélisation, est relationnel. Lustre [Cas+87] est le premier langage synchrone fonctionnel à flots de données. Sa première sémantique dénotationnelle [Ber86] ressemblait déjà à une sémantique de Kahn. Ce sont les travaux de Caspi et Pouzet sur les réseaux de Kahn synchrones [CP96] qui ont définitivement lié Lustre à la sémantique de Kahn, en remplaçant la primitive `current` par `merge`. Cela a mené au langage synchrone fonctionnel d'ordre supérieur Lucid Synchrone [Pou02].

De cette lignée est né Heptagon, dont la première version s'appelait MiniLustre [Bie+08] et fut développée dans l'optique de compiler source à source les automates, avec un langage minimaliste [CPP05]. Heptagon est un langage fonctionnel du premier ordre, dont le compilateur est un prototype universitaire, apparenté à l'outil industriel Scade [Tec13]. Ce dernier est largement utilisé dans les industries concevant des systèmes embarqués critiques.

Grâce à sa sémantique de Kahn, un programme Heptagon s'assimile à un réseau de Kahn. Typiquement, chaque appel de fonction en Heptagon peut se comprendre comme un processus indépendant, communiquant avec le reste du programme aux travers de files. Dans le modèle général, il est impossible de borner la taille des files. Les langages synchrones assurent, grâce à des horloges marquant la présence des flots à des instants logiques, que toute valeur produite est consommée dans le même instant. Ainsi, un programme Heptagon a une sémantique de Kahn synchrone, assurant que les files du réseau sont vides entre les instants. Dans ce cas,

il est assuré que le réseau, même désynchronisé, peut s'exécuter avec des files d'une seule place [Pla10]. Ainsi, une fois la découpe du programme effectuée, la distribution en tant que telle ne pose plus de question, son efficacité beaucoup plus [GN03].

Assurer l'efficacité revêt deux aspects non indépendants. Avoir un découplage suffisant des calculs : il y a des délais dans les dépendances entre calculs. Avoir une granularité importante des calculs : un fort ratio temps de calcul sur fréquence de communication. Or la sémantique synchrone et les horloges d'un programme Heptagon reflètent exactement l'inverse. Elles permettent au programmeur de se contenter d'un découplage d'un instant et à chaque instant, au maximum une valeur est calculée. De plus, les instants sont typiquement courts, pour assurer que le système réagit rapidement.

Pour obtenir un gain de performance, l'outil Ocrep [Gir94 ; GNP06] a recours à une analyse statique flots de données pour optimiser le contrôle et les communications du code généré. Dans ce cadre, le programmeur décide de l'emplacement des calculs, mais le découplage et les synchronisations entre calculateurs sont automatiques. La distribution de certains programmes Signal peut être faite à l'aide de l'outil généraliste SynDEx [KS08] ou avec Polychrony [Ma10]. De ces travaux nous faisons deux constats.

Le premier est que nous souhaitons le contrôle du parallélisme par le programmeur, directement dans le code source. Il doit pouvoir maîtriser à quels instants il y a communication ou synchronisation. La solution que nous proposons est l'utilisation des *futures* dans Heptagon. Ils fournissent ce pouvoir au programmeur, tout en restant des annotations qui peuvent être supprimées sans changer la sémantique dénotationnelle du programme.

Le deuxième constat est que la granularité des calculs est une problématique profonde, touchant en particulier aux questions de dépendance de données, de choix des horloges et de compilation modulaire. Heptagon, comme ses parents, restreint les réseaux de Kahn qui peuvent être écrits, de telle sorte que ces trois questions se traitent séparément. Pour mieux comprendre le lien entre ces éléments, nous revenons aux réseaux de Kahn. Notre principal résultat est la définition de la sous-classe des réseaux ordonnés réactifs. Ceux-ci sont les seuls pour lesquels nous pouvons décrire modulairement le comportement avec des horloges, sans restreindre les contextes d'appels. Ces réseaux ont une signature d'horloge en forme normale, qui maximise la granularité. Pour l'exprimer, nous introduisons les horloges entières, décrivant la communication de plusieurs valeurs en un seul instant. Nous appliquons ensuite nos résultats pour voir sous un nouveau jour Heptagon, Signal, les politiques de Lucid Synchrone [Cas+09], mais aussi proposer une analyse pleinement modulaire de Lucy-n [Pla10] le langage synchrone le plus fidèle aux réseaux de Kahn.

Guide de lecture et contributions

La première partie présente le langage Heptagon puis explique sa compilation en montrant le lien avec la sémantique de Kahn. La deuxième partie traite de l'ajout des *futures* dans Heptagon, de la motivation initiale à la génération de code à mémoire statique, en passant par de nombreux exemples, montrant l'expressivité des *futures* dans un cadre synchrone. La troisième partie étudie les dépendances de données dans les réseaux de Kahn, classifie ceux-ci, pour finalement arriver aux *réseaux ordonnés réactifs*, qui se décrivent fidèlement avec des horloges. Notre étude des réseaux de Kahn est orientée génération d'une fonction de transition, similairement à la compilation synchrone usuelle. Pour cela nous avons recours à une présentation *co-itérative* des réseaux. Finalement, nous revenons sur les langages synchrones et en particulier Lucy-*n*, pour lequel nos résultats et constructions sont entièrement calculables.

Nous proposons ci-dessous des lectures thématiques transversales. Elles devraient apporter au lecteur une vision d'ensemble, tout en mettant en valeur notre démarche et nos résultats.

Lectures des parties I et II

Lecture A : le compilateur Heptagon (parties I et II)

Une part importante de notre travail est le développement du compilateur Heptagon, principalement en collaboration avec A. Guatto et C. Pasteur, deux autres doctorants de l'équipe. Ce travail a débouché sur un article commun [Ger+12] qui a reçu le prix de meilleur article de LCTES'12, dont le sujet était la compilation efficace des tableaux dans un langage fonctionnel déclaratif synchrone. Ce sujet était initialement celui du stage de M2 de Pasteur, mais les bouleversements dans le compilateur ont été importants à tous les niveaux. Une fois ce travail commun fini, nous avons continué à maintenir la branche commune du compilateur, tout en développant activement les futures, cf. Lecture B. Pour ceux-ci, la gestion de la réinitialisation devait être faite avec soin, c'est pourquoi nous y consacrons une part importante de la partie I.

Le chapitre 1 est une présentation informelle d'Heptagon. Les concepts de base et la syntaxe y sont présentés. Ce chapitre peut être très rapidement parcouru par un lecteur familier de Lustre. Il notera cependant quelques différences et notamment l'utilisation de la primitive `merge` à la place de `current`. De plus, l'opérateur de réinitialisation de Lucid Synchrone est ajouté et étendu pour accepter la réinitialisation de fonctions dont les entrées sorties ne sont pas de mêmes horloges. Ce dernier point est en réalité crucial, puisque nous voulons un langage où tout appel de fonction peut être inliné.

Le chapitre 2 s'attache à lier la sémantique de Kahn à la compilation d'Heptagon, cf. lecture C. Cependant, si c'est uniquement la compilation qui l'intéresse, le lecteur pourra sauter à la section 2.5, en piochant au passage les informations nécessaires concernant les horloges en

figure 2.6. La présentation de la compilation d'Heptagon est proche de celle de ses articles fondateurs [CPP05 ; Bie+08]. Cependant, le noyau étudié est effectivement celui du compilateur, sans simplification, les nœuds ont des entrées et des sorties sur des horloges différentes. En particulier, la réinitialisation est traitée complètement et modulairement. Ceci, sans interférer avec la génération de code ne contenant pas de réinitialisation et en permettant d'inliner n'importe quel appel de fonction.

La compilation des futures, une fois le compilateur Heptagon écrit de manière robuste, ne requiert dans le cœur du compilateur que quelques lignes. Par contre, les générateurs sont plus touchés. Celui de Java prend un millier de lignes, adjoint d'une petite bibliothèque d'une centaine de lignes, utilisée par le code généré. Le générateur de C++ a pris le chemin inverse, en déléguant presque tout le travail à des templates et à la libraire.

En Java, la gestion de la mémoire est laissée à la machine virtuelle de Java. La compilation des futures se réduit à la génération des encapsulations des nœuds appelés de manière asynchrone, comme détaillé en sections 3.2.3 et 4.4.1. Cette compilation était évoquée dans notre article [CGP12], qui a été récompensé comme meilleur article de EMSOFT'12.

En C++, il faut en plus instrumenter le code généré pour permettre la gestion de la mémoire, qui est allouée de manière statique, comme détaillé en section 5.2. Ces travaux, implémentés et testés dans le compilateur Heptagon, n'ont pas encore donné lieu à publication.

Lecture B : expressivité des futures dans un langage synchrone fonctionnel (partie II)

L'intérêt principal des *futures* est leur simplicité. En lecture A, nous avons souligné que seulement la toute fin de la compilation nécessitait des changements. En chapitre 3, nous expliquons le principe élémentaire des futures et les différentes variantes qui ont existé depuis bientôt 40 ans. Dans un contexte purement fonctionnel comme en Heptagon, leur simplicité est aussi sémantique, puisqu'alors un future est équivalent à une indirection. En théorème 3.3.8, nous prouvons la préservation de la sémantique du programme quand les futures sont effacés. Ils peuvent donc être considérés comme de simples annotations du programmeur.

Malgré cette simplicité conceptuelle, les futures apportent une très grande expressivité opérationnelle. Ils fournissent au programmeur le moyen de lancer un calcul avant d'en avoir besoin et de récupérer uniquement les résultats nécessaires au moment voulu. Ceci, combiné à la programmation synchrone qui permet de parler du temps, rend possible une gestion précise des calculs parallèles. Nous donnons en chapitre 4 un certain nombre de possibilités mimant des squelettes de programmation classiques. Un point très intéressant est l'interaction entre la primitive de réinitialisation d'Heptagon et les appels asynchrones. La réinitialisation permet d'exploiter du parallélisme de données, de manière dynamique, cf. section 4.3. Finalement, nous montrons que les futures peuvent fournir un oracle au programme, le renseignant sur l'état réel d'avancement des calculs. Le programme peut alors être influencé par son exécution et des comportements dynamiques de gestion de la charge sont possibles, cf. section 4.6. Notre preuve de concept est la programmation d'un programme réagissant à sa première entrée prête et la programmation du *ou*-parallèle.

Tout ce surcroît d'expressivité opérationnelle requiert une gestion dynamique de la mémoire. En effet, le flot des données reste déterministe globalement, mais si l'on se place au niveau d'un instant, il y a des valeurs en cours de calcul dont l'existence et l'identité du destinataire ne sont pas connues. Toutefois, en interdisant l'échappement des futures, il est possible de générer du code sans allocation dynamique de mémoire, comme celui C++ de la lecture A. Notons que cette restriction entame très peu l'expressivité des futures. De tous nos exemple,

le seul concerné est la forme la plus dynamique du pipelining, appelée *promise pipelining*, cf. section 4.5.1.

Lecture C : sémantiques dénotationnelles d'Heptagon (chapitre 2)

Tout comme la compilation d'Heptagon vue en lecture A, la sémantique d'Heptagon a déjà été partiellement publiée par de précédents auteurs sur des langages similaires [CP98 ; HP00 ; Pou02]. La section 2.1 est une entrée en matière, donnant la sémantique de Kahn du noyau d'Heptagon. C'est l'occasion de souligner le déroulement du point fixe de la sémantique et de porter une attention particulière aux registres souvent mal définis. La sémantique de Kahn synchrone est présentée en section 2.2. Pour sa modularité, nous utilisons le concept d'*horloge d'activation d'un nœud*. Ce concept appartient au folklore du synchrone, bien qu'il soit au centre de la compilation modulaire efficace des langages synchrones. C'est en insistant sur ce point, que nous introduisons, en section 2.4, une forme originale de la sémantique. Nous l'appelons *sémantique synchrone implicite*, car contrairement aux sémantiques de Kahn synchrones usuelles, elle ne nécessite pas d'étendre le domaine des valeurs avec la valeur spéciale $\perp$ représentant l'absence. Pour ce faire, cette sémantique de Kahn manipule les environnements et mime le contrôle qui sera généré par la suite dans le code impératif séquentiel.

Nous avons traité la sémantique de la réinitialisation à part, en section 2.6.4. En effet, la sémantique de Kahn de la réinitialisation n'existe pas car elle nécessite de connaître les horloges des conditions de réinitialisation. Si elles sont données, nous proposons une traduction source à source de la réinitialisation, donc une sémantique de Kahn. En acceptant la réinitialisation de fonctions dont les entrées sorties ont des horloges différentes, cette traduction est plus complète que celle proposée dans [CPP05]. Pour donner la sémantique de Kahn synchrone, nous ne changeons pas le domaine des flots pour y adjoindre une composante dédiée, comme dans [HP00]. Nous proposons une sémantique synchrone plus efficace, qui a recours pour les appels de nœuds à une manipulation des environnements, comme notre sémantique synchrone implicite. Par ailleurs, cette manipulation se combine parfaitement pour donner la sémantique synchrone implicite de la réinitialisation section 2.6.5. Cette dernière a le mérite de correspondre à la compilation faite dans Heptagon.

Lectures de la partie III

Prérequis

La lecture de la partie III est indépendante de la partie II. De la partie I, il est nécessaire d'avoir intégré la sémantique de Kahn d'Heptagon avec les notations introduites. Il faut bien comprendre que la signature d'horloge d'un nœud, donnée modulairement par le compilateur, représente le comportement de la fonction de transition générée.

Motivations et historique

Contrairement à l'ordre de présentation de ce manuscrit, le développement du compilateur Heptagon et des futures n'ont pas été nos premiers travaux. Comme évoqué dans l'introduction, pour toute distribution d'un programme avec une sémantique de Kahn synchrone, il existe une exécution avec des files de communications de taille maximale 1. À partir de là, il y a deux questions : celle du découpage du code pour la distribution, qui est en grande partie une question métier, dépendant de multiples contraintes, l'autre concernant l'efficacité de la distribution. Cette dernière a été notre point de départ.

De prime abord, cela semble être une question d'optimisation : comment défaire le synchronisme du code source pour avoir assez de liberté permettant une exécution parallèle efficace. C'était l'approche de l'outil Ocrep [GNP06]. Au lieu de chercher à optimiser le code généré, nous avons voulu comprendre quoi changer dans le langage source, pour exprimer plus de parallélisme. Peut-on changer la granularité des calculs et/ou des communications ? Quelles sont les contraintes imposées par les dépendances de données ? Qu'est ce qu'une dépendance de donnée dans un réseau de Kahn ? Y-a-t-il une notion de causalité dans un réseau de Kahn ? Quel est le lien avec la compilation modulaire ? Avec les instants et l'hypothèse synchrone ? Avec les horloges ? Ce sont toutes ces questions qui nous ont suivi et auxquelles nous apportons des réponses dans cette partie III.

Nos tous premiers travaux portaient sur les *objets* de Lucid Synchrone V4 [Cas+09]. Ces objets sont associés à des politiques décrivant modulairement comment l'objet peut réagir, en décomposant chaque instant en sous-étapes. Bien que ces travaux n'aient pas entièrement porté leurs fruits, ils expliquent clairement que l'instant synchrone ne se réduit pas à une horloge d'activation binaire, activant ou pas un nœud.

Les travaux autour de Lucy-n, faits par Plateau et Mandel [Pla10] dans le cadre *n-synchrone* [Coh+06], ont ouvert la voie à une programmation plus proche des réseaux de Kahn, avec des restrictions moins ad-hoc qu'usuellement trouvées dans le synchrone. En effet, en Lucy-n, les files de communications doivent être bornées, mais ce, par un entier quelconque. Tout comme les horloges en Lucid Synchrone, les horloges et la taille des files sont calculées par le compilateur.

Dans le même temps, nous tentions, pour augmenter la granularité des calculs, de définir une opération de sur-échantillonnage qui soit vraiment le dual de l'échantillonnage. C'est-à-dire capable d'accélérer le calcul d'un flot et non pas, comme il est de coutume, de ralentir tout le reste du système. La base de ce travail est une extension des horloges en *horloges entières* [Ger08]. Au lieu d'indiquer simplement l'absence ou la présence d'une valeur, ces dernières donnent le nombre de valeurs du flot durant chaque instant.

Dans les langages synchrones, des restrictions permettent au compilateur de choisir les horloges et de vérifier la causalité entre les calculs, de manière indépendante. Il y a typiquement un *calcul d'horloge*, inférant les horloges [Pou02 ; ABL95], une *analyse de causalité*, utilisant des critères syntaxiques [HRR91] ou nettement plus avancés [ABL95 ; CP01 ; PR09].

Or avec les horloges entières, les possibilités sont beaucoup plus nombreuses. Choisir une approximation de la causalité, orthogonale aux horloges, n'est pas évident. Nous n'avons pas réussi à proposer un langage avec des restrictions suffisantes pour séparer ces analyses, tout en offrant une bonne expressivité et programmabilité. De plus, tous ces choix nous ont toujours semblé très ad-hoc, alors même qu'au moment de donner une sémantique, nous utilisons le cadre général et puissant des réseaux de Kahn. Cette frustration existait aussi dans Lucy-n, avec une analyse de causalité reposant sur l'existence d'un registre synchrone dans tout cycle, alors que, sémantiquement, il est possible de se passer du registre synchrone [Pla10].

Cette partie propose d'avoir toute l'information nécessaire dans les horloges, de telle sorte que l'analyse de causalité ne soit pas ad-hoc ni séparée. Nous donnons in fine une solution pour Lucy-n, acceptant les horloges entières sans les nécessiter et permettant de maximiser la granularité des calculs.

Lecture D : les dépendances de données dans les réseaux de Kahn (chapitre 7)

Le force du modèle des réseaux de Kahn est de représenter le comportement d'un réseau interagissant avec son environnement comme une fonction continue. Toute la structure est dans le domaine de cette fonction. Les entrées et les sorties d'une telle fonction sont des flots : la

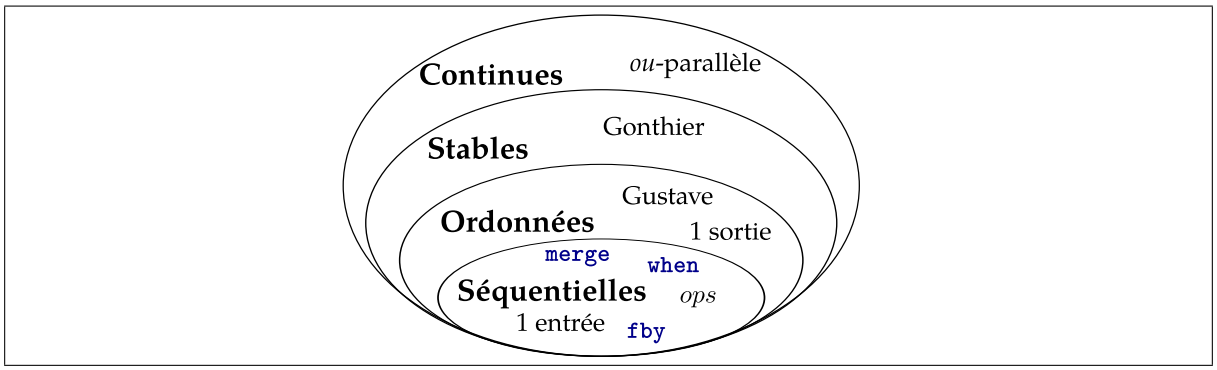


FIGURE 1: Inclusion des classes des réseaux de Kahn suivant que la fonction le représentant est continue (cf. section 2.1.2), stable (cf. section 7.2), ordonnée (cf. définition 7.3.4), séquentielle (cf. définition 7.4.10). La fonction du *ou*-parallèle n'est pas stable (cf. section 7.2.1), celle de Gonthier (cf. exemple 7.3.6) n'est pas ordonnée, celle de Gustave (cf. section 7.4.2 n'est pas séquentielle, quand finalement toutes les primitives d'Heptagon sont séquentielles. Toute fonction continue d'une sortie est ordonnée tandis que, si elle n'a qu'une sortie, elle est séquentielle.

suite finie ou infinie des valeurs vues au cours du temps. Ne reste donc que le comportement fonctionnel du réseau, représenté par une fonction f : si l'exécution du système donne le flot $\mathbf{x}$ en entrée, alors la sortie a le flot $\mathbf{y} = f(\mathbf{x})$.

Les dépendances de données, étudiées en section 7.2, posent la question en sens inverse : pour obtenir le flot de sortie $\mathbf{y}$, quel est le plus petit préfixe du flot d'entrée $\mathbf{x}$ nécessaire ? Cette question n'est pas toujours bien définie dès que le réseau a deux entrées ou plus, comme le montre l'exemple du réseau calculant le *ou*-parallèle de deux entrées, cf. section 7.2.1. Ce sont les réseaux dits *stables*, qui admettent une notion de dépendance de données. La stabilité d'une fonction a été définie par Berry [Ber78] dans le cadre plus général des fonctions continues.

Par la suite (cf. lecture F), nous voudrons être en mesure de connaître à tout instant quelle entrée doit être consommeée pour continuer à produire. Nous nommons *réseaux ordonnés* ceux qui le permettent. En effet, d'un autre point de vue, nous cherchons les réseaux pour lesquels les dépendances de données forment un ordre total entre les entrées et les sorties.

La dernière classe que nous considérons est celle des *réseaux séquentiels*. Ce sont ceux qui se programment entièrement dans un langage séquentiel impératif, muni, pour communiquer avec leur environnement, des primitives `get` et `set` de [KM76]. Cette classe est décrite par la définition générale d'une fonction continue séquentielle donnée par Vuillemin [Vui74].

La différence entre réseaux ordonnés et séquentiels est subtile et tient au fait que le réseau ne connaît pas la valeur des entrées qu'il n'a pas lue, alors qu'elles peuvent être utiles pour choisir l'entrée à lire. Les réseaux séquentiels (resp. ordonnés) sont ceux pour lesquels le réseau (resp. l'environnement) connaît quelle entrée doit être consommée pour continuer à produire. La génération de code pour un réseau ordonné mais pas séquentiel est coûteuse. Elle requiert une primitive telle que `select`, pour déterminer quelles sont les entrées disponibles, cf. l'exemple du réseau Gustave en section 7.4.2. Ces différentes classes sont résumées en figure 1. Notez que malheureusement, ni la classe des réseaux séquentiels, ni celle des réseaux ordonnés ne sont closes par produit, bien qu'elles le soient par composition et point fixe, cf. section 7.3.4.

Lecture E : représentation co-itérative des réseaux de Kahn (chapitres 6 et 7)

Tous nos travaux de la partie III ont pour but plus ou moins direct la compilation des réseaux de Kahn, donc le passage de la sémantique dénotationnelle à un code opérationnel : une

machine. Le principe de Kahn stipule que le point fixe d'une fonction continue dans le domaine de Kahn modélise l'exécution d'un réseau de machines communicant par des files, les machines ayant elles aussi une fonction continue pour sémantique. Ce principe a été prouvé pour divers modèles de machines, les automates d'entrées sorties [LS89], les CTS (*concurrent transition systems* [Sta87], plus largement dans un cadre de théorie des jeux [Fau82], etc. Cependant, aucun de ces modèles ne nous convient, les automates d'entrées sorties fournissent bien une fonction de transition mais ils font appel à des classes d'équivalence d'exécutions et nécessitent de se restreindre à des exécutions *équitables*. Cette contrainte externe à la fonction de transition n'est pas de manipulation aisée, bien que largement utilisée jusqu'à aujourd'hui [GB03]. Les CTS ou les automates d'entrées sorties monotones [PSS90] se dispensent de cette contrainte, mais considèrent que l'exécution du réseau est l'idéal engendré par toutes les exécutions possibles. Ceci est élégant mathématiquement mais nous éloigne de la compilation.

En chapitre 6, nous proposons une représentation dite *co-itérative*. La terminologie de co-itération est utilisée en référence à la compilation de Lucid Synchrone [CP98] sous la forme d'une fonction de transition dont la fonction d'exécution est co-itérative : à chaque pas, elle consomme une partie de ses entrées et produit des sorties. En repartant du calcul du point fixe des réseaux de Kahn, nous montrons comment ceci peut être fait de manière générique (cf. sections 6.1 et 6.2.1). Le système de transition que nous obtenons est une généralisation des transducteurs rationnels, avec une mémoire infinie nécessaire pour représenter tout réseau de Kahn. De plus, nous devons prendre en compte des exécutions infinies. Pour cette raison, la *confluence* du système de transition n'est pas suffisante pour assurer qu'il se comporte comme une fonction. Nous retombons sur l'approche classique en ajoutant une condition d'équité en théorème 6.2.14. Mais cela ne correspond pas à nos ambitions.

Le besoin d'équité provient de l'entrelacement nécessaire pour faire entrer un nombre fini d'infinis (plusieurs sorties avec un flot infini) dans un seul infini (celui de la suite de transition). Une solution mathématique élégante pourrait être de considérer des suites indexées par des nombres ordinaux [Par80]. Nous nous contentons de généraliser encore le système de transition en permettant qu'une transition produise un mot infini en sortie. Maintenant que nous avons assez d'infinis à disposition, nous forçons les systèmes à faire un nombre fini de pas si leur entrée est finie, ce que nous nommons *fortement normalisant*. Il suffit alors, que l'environnement fournisse les entrées petit à petit, pour que leur exécution soit une fonction continue propriété 6.3.8. Inversement, nous montrons que tout réseau peut s'écrire sous cette forme théorème 6.3.17.

Mieux, grâce à cette forme, nous sommes en mesure de séparer les réseaux en deux catégories, correspondant à deux formes canoniques, que nous appelons Moore et Mealy, cf. définition 6.3.12. Les réseaux Moore nécessitent une transition qui ne consomme pas de valeurs en entrée, quand les Mealy peuvent s'écrire sans aucune transition ne consommant rien. De plus, ces formes canoniques permettent de parler de granularité du calcul, que nous lions, en section 6.3.5, au coefficient de Lipschitz du réseau. Ces formes canoniques permettent en plus de caractériser un réseau stable avec une propriété de stabilité de la fonction de transition, cf. théorème 7.2.8.

Pour les réseaux ordonnés, nous donnons une forme normale au système de transition, cf. théorème 7.4.7. Elle maximise la granularité et son exécution est *déterministe* : pour des flots d'entrées fixés, il y a une unique suite de transition possibles. C'est, de notre point de vue, la meilleure compilation possible pour un réseau ordonné. En outre, elle discrimine simplement, parmi les réseaux ordonnés, ceux qui n'ont besoin que d'un seul infini : les *réseaux ordonnés réactifs*. Soit leur exécution déterministe est finie et seule la dernière transition peut produire une sortie infinie, soit leur exécution est infinie et ne contient pas de transition produisant une sortie infinie. Nous nommons ces derniers des *réseaux ordonnés interactifs*.

Lecture F : les réseaux de Kahn et la compilation synchrone (partie III)

La réussite des langages synchrones tient à la possibilité de composer des processus dont le comportement est infini, sans nécessiter de stockage intermédiaire. Dans les langages synchrones flot de données, ceci est assuré par la cohérence des horloges des flots, vérifiée par un calcul d'horloges. La compilation d'Heptagon l'illustre bien, toute valeur produite est consommée dans l'instant. En transposant cela aux réseaux de Kahn, cela signifie que les files de communications sont vides entre les instants et la compilation les transforme en variables locales à l'instant. C'est cela qui donne l'efficacité de la compilation des langages Lustre/Lucid Synchrone/Scade/Heptagon vers du code séquentiel. Dans ces langages, la modularité du calcul d'horloge est assurée par l'utilisation de signatures d'horloges. Celles-ci décrivent le comportement temporel des fonctions de transition générées. Ainsi, l'appelant peut être synchrone puisqu'il sait quand il y a consommation ou production de valeurs.

Quels sont les réseaux de Kahn qui admettent une description temporelle de leur fonction de transition ? Pour le comprendre, nous définissons, en section 6.1.5, la datation d'un réseau de Kahn. C'est une fonction qui associe une date réelle à chaque valeur consommée ou produite par le réseau. La densité de l'ensemble des réels permet de dater tous les réseaux, même ceux non-réactifs. Pour qu'une datation décrive fidèlement le réseau, elle doit être *principale* (cf. section 7.1.2). Ceci se comprend ainsi : toute valeur consommée avant la production d'une sortie, *doit* être nécessaire à la production de cette sortie. Nous montrons, en théorème 7.3.10 que l'existence d'une datation principale équivaut au fait que le réseau soit ordonné.

Le pointage, cf. chapitre 8, est l'abstraction des dates réelles en horloges, grâce au choix des instants. Les instants sont en nombre dénombrable et couvrent l'ensemble des réels. La date d'une valeur est abstraite par le numéro de l'instant la contenant. Par défaut, nous obtenons des horloges entières, puisque plusieurs valeurs d'un même flot peuvent avoir des dates appartenant à un même instant. La structure interne d'un instant détermine l'expressivité du pointage.

Le pointage en boucle simple [HRR91] associe un pour un les instants et les transitions : les entrées d'un instant sont consommées avant la production des sorties de l'instant. Il se représente avec une signature d'horloges. C'est le plus usuel, cf. Lustre, Lucid Synchrone, Lucy-*n*, Heptagon, etc. Nous l'étudions en détail en section 8.2. Nous montrons en théorème 8.2.10, qu'un réseau est ordonné réactif si et seulement si il admet une *signature d'horloge principale*. L'utilisation d'horloges entières dans la signature est facultatif, l'équivalence est aussi vraie avec des horloges booléennes puisqu'il suffit de choisir des instants plus petits, cf. théorème 8.2.11. Si au contraire, les instants sont choisis trop gros, la signature d'horloge obtenue n'est plus principale : elle ne correspond pas à une fonction de transition du réseau. La forme normale de la fonction de transition des réseaux ordonnés fournit une forme normale aux signatures principales. Une telle signature est dite *comblée* et correspond au choix des instants les plus gros possibles, pour maximiser la granularité. Cependant nous souhaitons des horloges entières et non infinies. Une distinction est donc à faire entre les réseaux interactifs qui ont une signature comblée, cf. théorème 8.2.17, et ceux simplement réactifs. Ces derniers se contentent d'une forme canonique presque normale, les signatures *quasi-comblées*. Finalement, en section 8.2.5, nous montrons comment une horloge d'activation à valeurs entières permet de faire du sur-échantillonnage, en fusionnant dans un même instant plusieurs activations d'une fonction de transition.

En chapitre 10, deux autres structurations de l'instant sont considérées, celui avec les *politiques* des *objets* de Lucid Synchrone V4 (sans existence publique outre notre article [Cas+09]) et celui, strictement plus expressif, avec les *graphes de dépendances conditionnées* [Le +91] de Signal. C'est l'occasion pour nous de revenir avec Kahn en tête sur un de nos premiers travaux. Nous montrons que la proposition [Cas+09] ne donne pas une sémantique de Kahn (non

synchrone) aux objets. Il manque de l'information pour que la sémantique soit fonctionnelle. Nous disons informellement qu'il y a des *fils cachés* puisqu'il faudrait rajouter des entrées sorties au programme pour avoir toute l'information nécessaire. Pour y remédier, nous proposons des modifications du langage des politiques. En SIGNAL, la structure de chaque instant est arbitrairement complexe et dépendante de la valeur *et de la présence* des flots. Ce dernier point est à la base de l'approche relationnelle de SIGNAL et est la première source de fils cachés. En se ramenant à notre modèle de transition, nous retrouvons qu'effectivement, les programmes SIGNAL *faiblement endochrones* [PCB04] admettent une sémantique de Kahn. Plus encore, nous obtenons qu'ils sont des réseaux *stables*.

Lecture G : LUCY-n (chapitre 9)

Le langage LUCY-n permet de programmer des réseaux de Kahn à mémoire bornée, dont le contrôle est ultimement périodique et connu statiquement. En ceci, il est très proche des modèles SDF [LM87] et CSDF [EBL94], qui fournissent une bibliographie très fournie. Les principaux thèmes de recherche tournent autour de l'ordonnancement du réseau, pour minimiser la mémoire ou maximiser le débit [SGB07 ; Ben+10 ; BTV12]. Mais LUCY-n est un langage de programmation, nous pensons qu'à ce titre, la compilation séparée et modulaire est très importante, alors qu'elle est délaissée dans les modèles comme les CSDF. En effet, l'optimisation de la mémoire ou du débit n'est pas modulaire. L'utilisation de signatures principales prend tout son sens pour LUCY-n.

La première remarque importante est que LUCY-n ne permet d'écrire que des réseaux stables. De plus, comme le contrôle est statique, les dépendances de données sont calculables à la compilation. Deuxièmement, les primitives du langage sont toutes ordonnées réactives. Nous leur donnons des signatures principales. Seule `when` a besoin d'horloges entières pour avoir une signature comblée (de granularité maximale). Nous retrouvons que la signature usuelle de `fby` n'est pas principale, correspondant au fait que sa compilation était traitée spécialement et associée à une analyse de causalité ad-hoc.

Le calcul de la signature principale peut se faire de proche en proche, en suivant les constructions proposées en section 7.3.4. La composition de réseaux ordonnés est ordonnée. Pour calculer une signature principale du résultat, nous utilisons l'ordonnancement « au plus tard » (ALAP cf. définition 9.3.1) de la composition, puis choisissons des instants (cf. section 9.3.1.3) pour retomber sur une signature d'horloge, principale. Le point fixe est traité de la même manière, sauf que l'ordonnancement ALAP est trivial. La mise en parallèle n'est pas toujours ordonnée. Nous en donnons un critère nécessaire et suffisant en section 9.5.

Le calcul de proche en proche de la signature se heurte au fait que le point fixe d'un réseau non ordonné peut l'être. Il y a donc des programmes LUCY-n qui admettent une signature principale dont le calcul doit être global, cf. section 9.6.2. Ce calcul reste un ordonnancement ALAP et peut donc se faire avec les techniques classiques, comme l'utilisation du solveur linéaire actuellement utilisé dans LUCY-n. En utilisant le fait qu'un programme LUCY-n à une seule sortie est obligatoirement ordonné réactif, nous obtenons une méthode systématique. Tout programme est projeté en autant de programmes qu'il a de sorties. Pour chaque projection, l'ordonnancement ALAP global donne une signature principale. Il suffit alors de mettre ces projections en parallèle. Si le résultat est ordonné réactif, nous obtenons une signature principale pour le programme entier. Cette signature est alors utilisée par le compilateur pour ordonnancer l'intérieur du programme. Si le résultat n'est pas ordonné, il faudra le reporter au programmeur. Il reste que, quand cela est possible, le calcul de proche en proche permettra probablement des messages d'erreur meilleurs et mieux localisés pour le programmeur.

La présentation exacte de tout ceci est grandement complexifiée par l'existence de réseaux qui produisent des flots finis. Ces flots finis ont des horloges ultimement nulles. Le fait que l'écriture de tels flots se fasse en Lucy-n est une vrai avancée, permettant de programmer uniformément le régime transitoire et celui périodique. Ce n'est pas le cas dans les SDF et CSDF. Par contre, cela requiert de pouvoir gérer des horloges d'activations dont la partie périodique est nulle, ce que nous appelons des taux nuls. Leur gestion ajoute des cas spéciaux à traiter dans presque toutes les formules que nous donnons.

Remarquez que nous n'avons pas parlé de causalité. En effet, la « causalité » du programme vu comme un réseau de Kahn n'a pas vraiment de sens. Cependant, en tant que programmeurs de langage synchrone, nous savons que manipuler les points fixes n'est pas évident et il y en a qui doivent être considérés comme suspicieux. Par exemple l'équation `y = y+1` est définie en Kahn, `y` vaut le flot vide, mais ce n'est probablement pas voulu. L'idée est que si un point fixe arrête de produire par manque d'entrée, le programmeur a probablement oublié de bien initialiser le point fixe (`y = 0 fby (y+1)`). Notre définition coïncide pour Lucy-n avec ce qui a été proposé dans une version expérimentale de Lucy-n [MPP11]. Mais, surtout, nous montrons en théorème 9.4.4, que cette notion de causalité ne dépend pas de la signature choisie si elle est principale. L'analyse de causalité est ainsi modulaire et nous aimons à dire que nous avons « mis la causalité dans les horloges », alors qu'en réalité, ce sont les dépendances de données que nous avons mises dans les signatures d'horloges.

Table des matières

Notations

x (police tt)	Syntaxe concrète / code source.

Notations générales mathématiques

true, false	Valeurs booléennes vraie et fausse.
$\wedge$, $\vee$, $\neg$	Conjonction, disjonction et négation booléennes.
∞	L'infini majorant (pour $\leq$) l'ensemble auquel il est ajouté.
$\mathbb{N}$, $\mathbb{N}^*$, $\overline{\mathbb{N}}$	Les entiers naturels, privés de 0, complétés avec le majorant ∞.
$\mathbb{R}$, $\mathbb{R}^*$, $\mathbb{R}_+$, $\overline{\mathbb{R}}$	Les réels, privés de 0, positifs, complétés avec le majorant ∞.
$\leq$	Ordre usuel des entiers et réels
$\preccurlyeq$	Relation partielle ordonnant les parties de $\overline{\mathbb{R}}$ (p.192).
$\cup$, $\cap$, $\subset$, $\setminus$	Union, intersection, inclusion et différence d'ensembles.
$\mathfrak{P}(X)$	Ensemble des parties de l'ensemble X.

Scalaires, mots et flots

Pour voir rassemblées toutes ces considération, il y a l'annexe IV.

V, V^*, V^ω	Alphabet dénombrable, ensembles des mots finis et infinis de V.
$V^\infty = V^* \cup V^\omega$	Ensemble des flots de V (p.41).
x (minuscule italique)	Scalaire, aussi mot à une seule lettre.
$\mathbf{x}$ (minuscule gras)	Flot (de scalaires), aussi appelé flot scalaire.
$\mathbf{X}$ (majuscule gras)	Vecteur de flots, aussi appelé flot.
$\mathbf{X} = (\mathbf{X}^{(1)}, \cdots, \mathbf{X}^{(n)})$	Décomposition du flot $\mathbf{X}$ en ses n composantes scalaires.
ε, $\boldsymbol{\varepsilon}$	Flot scalaire vide, flot vide.
$x.\mathbf{x}$	Concaténation d'une lettre en tête d'un flot.
$\mathbf{x}[k]$	$k$ième valeur du flot, avec $k \geq 1$.
$\mathbf{X}.\mathbf{Y}$	Concaténation étendue aux flots (p.140).
$\lvert\mathbf{x}\rvert$	Taille d'un flot scalaire, à valeurs dans $\overline{\mathbb{R}}_+$.
$\lvert\mathbf{X}\rvert$	Maximum des tailles des composantes de $\mathbf{X}$ (p.122).
$\lVert\mathbf{x}\rVert$	Somme des éléments du flots d'entiers $\mathbf{x}$ (p.211).
$\mathcal{D} = (T_1^\infty \times \cdots \times T_n^\infty, \sqsubseteq)$	Domaine de Kahn des vecteurs de flots.

$\mathbf{X} \sqsubseteq \mathbf{Y}$	Ordre partiel préfixe des flots, $\mathbf{X}$ est préfixe de $\mathbf{Y}$.
$\mathbf{X} \sqcup \mathbf{Y}$, $\mathbf{X} \sqcap \mathbf{Y}$	Plus petit majorant, respectivement plus grand minorant.
$\mathbf{X} \setminus \mathbf{Y}$, $\mathbf{X} \prec \mathbf{Y}$	Résidu de $\mathbf{X}$ moins le préfixe $\mathbf{Y}$, couverture de $\mathbf{X}$ par $\mathbf{Y}$.
$\mathcal{A}(\mathbf{X})$	Ensemble des approximants de $\mathbf{X}$ (flots finis préfixes de $\mathbf{X}$).
$\mathbf{X} \uparrow \mathbf{Y}$	Relation de compatibilité : il existe un majorant de $\mathbf{X}$ et de $\mathbf{Y}$.
$(\mathbf{X}_k)_k$	Suite de flots, implicitement avec $k \in \mathbb{N}^*$ et la convention $\mathbf{X}_0 = \boldsymbol{\varepsilon}$.
$(\mathbf{X}_{\delta k})_k$	Suite d'incréments, définissant la suite $(\mathbf{X}_k)_k$ avec $\mathbf{X}_{k-1}.\mathbf{X}_{\delta k} = \mathbf{X}_k$.

Les ʀKPN

Les réseaux de Kahn associés à une datation sont présentés en chapitre 6.

F	Fonction continue des entrées sorties d'un réseau de Kahn.
E, S, P	Ensemble des ports d'entrées, de sorties et de tous les ports.
$T_{\mathbf{D}}(p)[i]$	Fonction de datation pour le flot d'entrées $\mathbf{D}$, donnant la date de la $i$ème valeur passant au port p.
$T_{\mathbf{D}}(\mathbf{X})$	Maximum des dates des valeurs du flot $\mathbf{X}$, implicitement lié aux ports d'entrées ou de sorties suivant $\mathbf{X}$.
$R = (R^k)_k$	Les instants (ouverts de $\mathbb{R}$) numérotés dans l'ordre avec $k \in \mathbb{N}^*$.
$T^k_{\mathbf{D}}(p)$	Ensemble des dates des valeurs au port p inclues dans l'instant R^k.
$tF(s, \mathbf{X}) = (\mathbf{Y}, s')$	Fonction de transition et sa représentation (p.135, 141).
$s \xrightarrow{\mathbf{X}/\mathbf{Y}} s'$	
tnF	Forme normale de la fonction de transition (p.173).
$\mathrm{run}(tF)(s, \mathbf{X})$	Traces de tF à partir de l'état s avec l'entrée $\mathbf{X}$ (p.135).
$\mathrm{exec}(tF)(\mathbf{X})$	Trace maximale de tF fortement normalisante (p.142).

Dépendances de données et datations

$\mathbf{Y}^{F*\mathbf{D}}$	Inverse de $\mathbf{Y}$: plus petit préfixe de$\mathbf{D}$ produisant $\mathbf{Y}$ (p.160).
$\mathbf{Y}^{F**\mathbf{D}}$	Bi-inverse de $\mathbf{Y}$: plus grande sortie produite si $\mathbf{Y}$ l'est (p.160).
$\mathrm{Inv}(F, \mathbf{D})$	Ensemble des inverses finis de F pour l'entrée $\mathbf{D}$ (p.162).
$T \sqsubseteq\!\!\!\sim T'$	Pré-ordre d'inclusion des datations : T consomme avant et produit après T' (p.154).
$T \backsim T'$	Relation d'équivalence entre datations induit par $\sqsubseteq\!\!\!\sim$ (p.154).

Première partie

Heptagon

Le langage et compilateur Heptagon est à la fois objet et outil de cette thèse. La version actuelle est une réécriture que nous avons faite en collaboration avec G. Delaval, A. Guatto et C. Pasteur. Ce travail nous a permis de mener de nombreuses expérimentations de compilation et de programmation.

Dans le premier chapitre, nous présentons informellement Heptagon et expliquons les constructions de base du langage ainsi que les concepts primordiaux. En particulier, nous présentons la notion d'horloge d'un flot, essentielle à la compilation efficace [Bie+08].

Dans le second chapitre, nous donnons trois sémantiques dénotationnelles. La première se nomme usuellement sémantique de Kahn [Kah74]. Elle décrit un système purement flot de données. La seconde est une sémantique de Kahn synchrone [HP00], qui rajoute une notion d'instant global de calcul. Pour cela, elle ajoute une valeur spéciale dénotant l'absence d'un flot. Notre présentation a l'avantage de traiter tout le langage, y compris les définitions de fonctions. Ceci nous permet de souligner l'importance de la notion d'horloge d'activation d'un nœud.

La troisième sémantique, que nous nommons synchrone implicite, est originale. Elle permet, en ajoutant du contrôle, d'avoir une sémantique de Kahn synchrone sans manipuler de valeur spéciale. Tout en restant dénotationnelle, elle mime la structure de contrôle du code impératif que nous générons. Comme ce dernier, elle n'active un calcul que si son horloge l'indique et elle est modulaire sans nécessiter l'ajout d'entrée aux fonctions.

Nous présentons la génération de code avec une fonction de traduction vers un langage impératif nommé Obc, similairement à l'article fondateur d'Heptagon [Bie+08]. Obc est effectivement utilisé comme langage intermédiaire dans Heptagon. Il représente une fonction de flot du code source comme un objet contenant l'état de la fonction, une méthode de réinitialisation et une méthode de transition.

Nous finissons la présentation d'Heptagon en ajoutant la réinitialisation d'équations, introduite dans [HP00]. En Heptagon, pour préserver le principe de substitution, celle-ci a la particularité de ne pas avoir besoin d'être synchrone avec ce qu'elle touche. Mais la compilation source à source d'une telle réinitialisation est très coûteuse. Pour la traiter, le compilateur utilise de nouvelles constructions dont nous donnons la traduction vers Obc et les sémantiques. Jusqu'à présent, la réinitialisation était surtout utilisée pour la compilation des automates [CPP05]. Nous nous y intéressons plus particulièrement car elle apporte du parallélisme que nous exploitons avec les futures en deuxième partie.

Chapitre 1

Heptagon un langage synchrone fonctionnel flots de données

Dans ce chapitre, nous présentons Heptagon, un descendant de Lustre. Heptagon est un prototype universitaire de Scade [Tec13]. La principale évolution par rapport à Lustre est l'intégration d'automates. La première version d'Heptagon s'appelait MiniLustre [Bie+08] et fut développée par Marc Pouzet, dans l'optique de compiler source à source les automates, avec un langage minimaliste [CPP05].

1.1 Tour d'horizon

En Heptagon, un programme est un ensemble de déclarations de fonctions de flots, dont l'une d'elle est marquée comme principale. Cette dernière recevra ses entrées de l'environnement et ses sorties seront les réponses du programme. Considérons la déclaration de fonction `add33`, d'entrée `x`, de sortie `y` et de variable locale `t` :

```
fun add33 (x : int) returns (y : int)
var t : int;
let
  y = t + 3;
  t = x + 3;
tel
```

1.1.1 *Nota bene* Le point virgule n'en est pas un

Bien que la syntaxe utilise un point virgule (reliquat de Lustre) pour séparer les équations, le langage est flot de données, donc déclaratif et l'ordre des équations n'a pas d'importance. Celle de `t` peut tout aussi bien être avant celle de `y`.

Heptagon est utilisé pour programmer des systèmes réactifs. C'est un langage synchrone qui s'appuie sur une notion de temps logique. À chaque instant, le programme `add33` attend une valeur de `x` pour produire une valeur de `y`. C'est ainsi que pour expliquer le comportement d'un réseau, nous écrivons un *chronogramme* :

numéro de l'instant :	1	2	3	4	$\cdots$
`x`	x_1	x_2	x_3	x_4	$\cdots$
`t`	$x_1 + 3$	$x_2 + 3$	$x_3 + 3$	$x_4 + 3$	$\cdots$
`y`	$x_1 + 6$	$x_2 + 6$	$x_3 + 6$	$x_4 + 6$	$\cdots$

Toutes les constantes, variables et expressions ont pour valeur des *flots*. Un flot est un mot possiblement infini de valeurs appelées scalaires. Ainsi, la constante `3` représente le flot infini dont les éléments sont tous l'entier 3. Nous notons mathématiquement la valeur de ce flot avec le mot infini $3^\omega = 3.3.3.3.\cdots$. Ainsi, la sémantique de Kahn de la constante `3` est $[\![\texttt{3}]\!]^{\mathcal{K}} = 3^\omega$. De la même manière, les opérateurs scalaires usuels sont liftés point à point aux flots. Si l'entrée `x` vaut $x_1.x_2.x_3.\cdots$, nous avons $[\![\texttt{x + 3}]\!]^{\mathcal{K}} = (x_1+3).(x_2+3).(x_3+3).\cdots$ et le flot `t` est défini comme valant ce même flot. Similairement, nous avons $[\![\texttt{y}]\!]^{\mathcal{K}} = (x_1+6).(x_2+6).(x_3+6).\cdots$. La $k$ième valeur d'un mot `x` est notée $\texttt{x}[k]$ avec $1 \leq k$. Nous avons par exemple $([\![\texttt{x + 3}]\!]^{\mathcal{K}})[2] = x_2 + 3$.

Un flot est tel que tous ses éléments sont des scalaires de même type. Par abus de langage, un flot dit de type t signifie que ses scalaires le sont. L'annotation de type est donnée par deux points, par exemple une variable `x` porte un flot de type `int` est noté avec `x : int`.

|| 1.1.2 *Nota bene* Toutes les déclarations de variables doivent être annotées avec leur type.

1.1.1 Code généré et boucle de simulation

Pour chaque fonction déclarée, le compilateur produit ce que nous appelons une *fonction de transition*. Un appel à cette fonction effectue un pas d'exécution. Pour `add33`, la fonction de transition générée en C est `step_add33`. Si `add33` est de plus la fonction principale du programme, alors le compilateur va générer le code de simulation `main` qui gère l'interaction entre la fonction principale et l'environnement. *Chaque itération de la boucle de simulation correspond à un instant logique*, décomposé en trois phases, la lecture de la valeur courante des entrées, un appel à la fonction de transition principale et l'écriture des sorties :

```
//fonction de transition de add33
int step_add33(int x) {
  int t, y;
  t = x + 3;
  y = t + 3;
  return y;
}
```

```
int main() {
  int x, y;
  while(true) { //boucle de simulation
    x = read_input();
    y = step_add33(x);
    write_output(y);
  }
  return 0;
}
```

La fonction de transition travaille simplement sur des scalaires puisque la boucle de simulation lui demande d'effectuer un pas. De plus, comme la $i$ème valeur de sortie de `add33` dépend uniquement de la $i$ème valeur de l'entrée, la fonction de transition n'a pas d'état (de mémoire interne).

1.1.2 Le registre synchrone `fby`, aussi appelé délai unité

Il n'y a en Heptagon que la primitive `fby` (dite *followed by*) qui permet de décaler un flot et donc de nécessiter un état. Plus précisément, `x fby y` prend la première valeur de `x` et l'ajoute en tête du flot `y`. Si $[\![\texttt{x}]\!]^{\mathcal{K}} = x_1.x_2.x_3.\cdots$ et $[\![\texttt{y}]\!]^{\mathcal{K}} = y_1.y_2.y_3.\cdots$ alors $[\![x \texttt{ fby } y]\!]^{\mathcal{K}} = x_1.y_1.y_2.\cdots$.

L'exemple `sum` La fonction `sum` calcule la somme cumulée de son entrée.

```
node sum (x:int) returns (y:int)
var t : int;
let
  t = 0 fby y;
  y = t + x;
tel
```

x	x_1	x_2	x_3	x_4
t	0	x_1	$x_1 + x_2$	$x_1 + x_2 + x_3$
y	x_1	$x_1 + x_2$	$x_1 + x_2 + x_3$	$x_1 + x_2 + x_3 + x_4$

1.1.3 *Nota bene* **Fonction sans état, nœud, `fun` vs `node`**
Toute fonction à état est appelée un nœud et sa déclaration doit être faite avec le mot clef `node`. Une fonction a un état ssi elle contient un `fby`, ou elle appelle une fonction à état (déclarée avec `node`). La fonction `add33` n'a pas d'état, ce qui permet de la déclarer avec `fun`.
Si cela importe, nous précisons quand une fonction est sans état.

Cette distinction syntaxique entre fonction combinatoire et nœud est un typage rudimentaire nécessaire à la génération de code. En effet le code généré pour `sum` doit stocker l'argument de `fby` (la valeur de `y`) entre les appels à sa fonction de transition. La présentation la plus simple de la compilation d'un nœud à état est la génération d'une classe, avec une méthode `step` qui joue le rôle de la fonction de transition et les variables d'instances qui stockent l'état du nœud. Une méthode `reset` est utilisée pour initialiser l'état.

Les équations de `t` et `y` sont mutuellement récursives, la génération de code séquentiel nécessite de trouver un *ordonnancement* de leurs évaluations. Si cet ordre n'existe pas, le nœud est rejeté comme *non causal*. Ici, comme `t` est en réalité la valeur stockée dans le registre, elle peut être lue avant d'être mise à jour.

Code C++ généré pour `sum` :

```
class Sum {
  int t;
public:
  Sum() { }
  int step(int x) {
    int y = this.t + x;
    this.t = y;
    return y;
  }
  void reset() {
    this.t = 0;
  }
};
```

En C, le code sépare explicitement l'état `Sum_mem` de la fonction de transition `Sum_step` et de la fonction d'initialisation `Sum_reset` :

```
typedef struct Sum_mem { int t; } Sum_mem;
void Sum_step(int x, int* y, Sum_mem* self) {
  (*y) = self->t + x;
  self->t = (*y);
}
void Sum_reset(Sum_mem* self) { self->t = 0; }
```

1.1.3 Opérateurs combinatoires

Outre l'addition, les opérateurs usuels sur les entiers et les booléens font partie des expressions du langage. Dans l'exemple ci-dessous, nous en utilisons deux, le test d'égalité (`=`) et la conditionnelle `if then else`. Respectivement, ce sont les opérateurs (`==`) et l'opérateur ternaire (`? :`) du langage C. Le compteur `cpt` est remis à 1 à chaque fois qu'il atteint 3 :

```
node period3() returns (c : bool)
var cpt, next : int;
let
  next = if (cpt = 3)
         then 1 else (cpt + 1);
  cpt = 1 fby next;
  c = (cpt = 1);
tel
```

cpt	1	2	3	1	2
next	2	3	1	2	3
c	true	false	false	true	false

Nous utiliserons aussi une version plus générique, dont la période change suivant une entrée. La valeur de cette entrée n'est utilisée que quand la sortie est vraie. Notez que `period3()` équivaut à `countdown(3)` :

```
node countdown(x :int)
returns (c : bool)
var cpt, next : int;
let
  next = if (cpt <= 1)
         then x else (cpt - 1);
  cpt = 1 fby next;
  c = (cpt = 1);
tel
```

x	2	x_2	3	x_4	x_5	1	3
cpt	1	2	1	3	2	1	1
next	2	1	3	2	1	1	3
c	true	false	true	false	false	true	true

Les valeurs x_2, x_4, x_5, x_8 et x_9 ne sont pas utilisées par `countdown` puisqu'il est déjà en train de compter à rebours.

1.1.4 Appel de fonction

Chaque appel d'une fonction utilise sa propre mémoire et est ainsi nommée une instance. Considérons la composition de `sum` avec `sum` :

```
node double_sum(x : int) returns (y : int)
var t : int;
let
  t = sum(x);
  y = sum(t);
tel
```

x	x_1	x_2	x_3
t	x_1	$x_1 + x_2$	$x_1 + x_2 + x_3$
y	x_1	$2x_1 + x_2$	$3x_1 + 2x_2 + x_3$

Chaque appel à `sum` a son propre état. Ces instances font partie de l'état de `double_sum`. Ainsi, la *mémoire est hiérarchique*, ce qui est une propriété très importante des programmes Heptagon. La méthode d'initialisation est en charge d'appeler récursivement celle des fils. Si

`double_sum` est la fonction principale, le code généré, y compris celui du simulateur, est :

```
class Double_sum {
  Sum sum1, sum2;
public:
  Double_sum(){ }
  int step(int x) {
    int t, y;
    t = sum1.step(x);
    y = sum2.step(t);
    return y;
  }
  void reset() {
    sum1.reset();
    sum2.reset();
  }
};
```

```
//Alloue récursivement tous les états
Double_sum main_node;

//Initialise récursivement tous les états
main_node.reset();

int main() {
  int x, y;
  while(true) {
    x = read_input();
    y = main_node.step();
    write_output(y);
  }
  return 0;
}
```

1.2 Rythmes différents et horloges des flots

1.2.1 La primitive `when`

Comme en Lustre, `x when c` est l'échantillonnage du flot `x` par le flot booléen `c`. Le flot résultant est présent et vaut la valeur courante de `x` si `c` est vrai, sinon il est absent. Pour être défini, `when` requiert que ses entrées soient présentes aux mêmes instants. Considérons les équations suivantes et le chronogramme correspondant :

```
c = period3();
y = x when c;
t = x when not c;
d = period3();
z = y when d;
```

c	true	false	false	true	false	false	true	false	false	true
x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
y	x_0			x_3			x_6			x_9
t		x_1	x_2		x_4	x_5		x_7	x_8	
d	true			false			false			true
z	x_0									x_9

Nous avons deux flots `c` et `d` qui ont la même sémantique $[\![\texttt{c}]\!]^{\mathcal{K}} = [\![\texttt{d}]\!]^{\mathcal{K}} = (\text{true.false.false})^{\omega}$. Mais les valeurs de `d` ne sont produites que quand `c` vaut true, puisqu'elles doivent arriver pour échantillonner `y`, lui-même présent que quand `c` est vrai. Ainsi, `z` n'est présent que quand `d` est vrai, c'est-à-dire une fois sur trois d'une fois sur trois. Finalement, `x when not c` échantillonne `x` quand `c` est faux et le compilateur accepte la syntaxe `x whenot c`.

1.2.2 La primitive `merge`

La primitive duale de l'échantillonneur n'est pas `current` comme en Lustre, mais `merge`. L'expression `merge c x y` fusionne les flots `x` et `y`. À tout instant, le flot `c` doit valoir true ssi `x` est présent et `y` est absent. Symétriquement, `c` doit valoir false ssi `x` est absent et `y` est présent.

Considérons le chronogramme suivant, où nous calculons l'identité `o = x` en passant par des flots intermédiaires plus lents `x1` et `x2` :

```
x1 = x when c;
x2 = x when not c;
o = merge c x1 x2;
```

c	true	false	false	true	false	false	true	···
x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	···
x1	x_0			x_3			x_6	···
x2		x_1	x_2		x_4	x_5		···
o	x_0	x_1	x_2	x_3	x_4	x_5	x_6	···

Nous observons bien que les arguments `x1` et `x2` ne sont présents que quand `merge c` en a besoin. Par contre, l'expression `merge c x x2` est rejetée par le compilateur car il y a des instants où `x` est présent alors que `c` vaut false (au 3ième, 4ième et 6ième instants).

1.2.3 Versions non booléennes

1.2.3.1 Types sommes

En Heptagon, il est possible de définir des *types sommes*, comme `abc`, dont les constructeurs sont `A`, `B` et `C` :

```
type abc = A | B | C
```

Si `c` est de type `abc`, alors l'échantillonnage de `x` quand `c` vaut `B` s'écrit `x when (B = c)`. Les booléens sont définis [1] par `type bool = true | false`. Cette approche permet d'adopter une notation uniforme qui ne recourt pas à `whenot`. Nous avons les équivalences (toutes les variantes sont acceptées par le compilateur) :

```
x when b      ≡  x when (true = b)
x whenot b    ≡  x when (false = b)
merge b x y   ≡  merge b (true -> x) (false -> y)
```

L'ordre des cas du `merge` n'a plus d'importance. Par contre, il en faut autant que de constructeurs du type. Si `c : abc` alors il faut trois cas. Ainsi, l'identité `y = x` peut aussi s'écrire :

```
y = merge c (B -> x when (B = c)) (A -> x when (A = c)) (C -> x when (C = c))
```

1.2.3.2 Entiers bornés

Il est aussi possible d'utiliser des entiers bornés. La syntaxe `int{<n}` dénote le type des entiers naturels bornés par `n`. Si l'argument du `merge` est de ce type, il nécessite `n` autres arguments. Si `n` vaut trois, l'identité s'écrit :

```
y = merge c (0 -> x when (0 = c)) (1 -> x when (1 = c)) (2 -> x when (2 = c))
```

1.2.4 Ne pas confondre `merge` et `if then else`

Montrons par l'exemple la différence entre `if then else` et `merge`, qui peuvent sembler similaires à première vue [2]. Considérons que nous avons une entrée `x` valant soit `Torchon` soit `Serviette`. La sortie du nœud `range` vaut -1 si l'entrée courante est une `Serviette`, sinon elle indique combien de `Torchon` sont passés. Le nœud `melange` ci-dessous indique bien -1 si

1. Normalement les constructeurs commencent par une majuscule, cette déclaration de type n'est donc pas valide et ces constructeurs sont traités de manière spéciale dans le compilateur.
2. Un peu comme les torchons et les serviettes.

l'entrée est une `Serviette`, mais il mélange les `Torchon` et les `Serviette` dans son compte :

```
type ts = Torchon | Serviette

node melange(x : ts) = (z : int)
var m : int;
let
  m = sum(1);
  z = if(Torchon = x) then m else -1;
tel
```

```
node range(x : ts) returns (y : int)
var torchons : int;
let
  torchons = sum(1);
  y = merge x (Torchon -> torchons)
              (Serviette -> -1);
tel
```

x	Torchon	Torchon	Serviette	Serviette	Torchon	Serviette	···
m	1	2	3	4	5	6	···
z	1	2	−1	−1	5	−1	···
torchons	1	2			3		···
y	1	2	−1	−1	3	−1	···

La différence entre `range` et `melange` tient aux instants de présence de l'expression `sum(1)`. Dans `range`, l'instance de `sum(1)` ne doit fournir des valeurs que quand `x` est un `Torchon`, alors que `melange` l'active tout le temps. Le code généré reflète précisément cette différence :

```
enum Ts {TORCHON, SERVIETTE};

class Melange {
  Sum sum;
public:
  Melange () {}
  int step (Ts x) {
    int z, m;
    m = sum.step(1);
    z = (TORCHON == x) ? m : -1;
    return z;
  }
  void reset() {sum.reset();}
};
```

```
class Range {
  Sum sum;
public :
  Range() {}
  int step (Ts x) {
    int y, torchons;
    switch (x) {
    case TORCHON:
      torchons = sum.step(1);
      y = torchons;
    case SERVIETTE:
      y = -1;
    }
    return y;
  }
  void reset() {sum.reset();}
};
```

1.2.1 *Remarque* **Échantillonner les constantes**
En Lustre, il aurait fallu écrire `torchons = sum(1 when (Trochon = x))`, car les constantes comme `1` sont présentes à tous les instants. Cette écriture a le mérite d'expliciter quand les torchons sont calculés, mais elle est très lourde. En Heptagon, les constantes sont automatiquement présentes quand nécessaire.

Traduction du `if then else` La construction `if c then e1 else e2` requiert une valeur pour tous ses arguments à chaque activation. Elle se réécrit en :
`merge c (e1 when c) (e2 when not c)`.

1.2.5 Horloges et pointeaux

Nous avons vu que `when` et `merge` imposent des contraintes sur la présence de leurs entrées. Cela est en fait le cas pour toutes les expressions. Par exemple, comme tous les opérateurs liftés, l'addition `x + y` requiert que `x` et `y` soient présents aux mêmes instants. Pour déterminer la présence des flots, le compilateur associe à chaque flot une *horloge*. L'horloge $\mathrm{clock}\,(\texttt{a})$ d'un flot `a` est un mot booléen infini dont la $i$ème valeur est true si `a` est présent au $i$ème instant du programme et false sinon. À cause de `when` et `merge`, la présence d'un flot peut dépendre de la valeur d'autres flots. En Heptagon, les flots ne sont pas interprétés pas le compilateur et les horloges ne sont donc pas calculées explicitement. La vérification de la cohérence des horloges est traditionnellement faite par une inférence de type appelée calcul d'horloge [Cas+87 ; CP03b].

$$ck := \mathbf{true} \mid \alpha \mid ck\ \texttt{on}\ (i = p)$$

Dans ce cadre, une horloge ck est soit l'horloge de base globale toujours vraie ($\mathbf{true} = \mathrm{true}^\omega$), soit une variable d'horloge α, soit une horloge *échantillonnée* par un flot booléen. Les flots booléens acceptés comme échantillonneurs sont d'une forme spéciale, testant l'égalité entre une constante i et un *pointeau* p. Un pointeau est un flot nommé du programme. Si l'horloge ck vaut le mot $\mathbf{b}$ et le pointeau p a pour sémantique le mot $\mathbf{p}$, alors l'horloge ck `on (`i `=` p`)` vaut $\mathrm{on}_i^{\mathcal{K}}(\mathbf{b}, \mathbf{p})$ défini par :

$$\mathrm{on}_i^{\mathcal{K}}(\mathrm{true}.\mathbf{b}, i.\mathbf{p}) = \mathrm{true}.(\mathrm{on}_i^{\mathcal{K}}(\mathbf{b}, \mathbf{p})) \qquad \mathrm{on}_i^{\mathcal{K}}(\mathrm{false}.\mathbf{b}, \mathbf{p}) = \mathrm{false}.(\mathrm{on}_i^{\mathcal{K}}(\mathbf{b}, \mathbf{p}))$$
$$\mathrm{on}_i^{\mathcal{K}}(\mathrm{true}.\mathbf{b}, j.\mathbf{p}) = \mathrm{false}.(\mathrm{on}_i^{\mathcal{K}}(\mathbf{b}, \mathbf{p})) \quad \text{avec } j \neq i$$

1.2.2 *Remarque* Il y a une valeur du pointeau p par instant où l'horloge ck est vraie.

L'horloge de base globale représente l'horloge d'activation du programme principal. Or les nœuds sont typés séparément, sans connaître le programme principal. Ainsi, les horloges données par le compilateur ont toujours une variable d'horloge comme racine.

1.2.3 *Nota bene* **Variable d'horloge unique « • »**
En Heptagon, toutes les horloges des flots d'un nœud doivent utiliser une même variable d'horloge. La syntaxe concrète pour cette variable d'horloge est un point, que nous grossissons dans le texte en « • ». Cette unique variable d'horloge est appelée *horloge de base du nœud* / *horloge d'activation du nœud*.

À la compilation de `sum`, tous ses flots sont sur l'horloge de base de `sum`. Une fois `sum` instancié dans `range`, tous ses flots se voient présents uniquement quand `x` est un `Torchon`. Leur horloge de base a été remplacée par • `on (Torchon = x)` où • dénote l'horloge de base locale à `range`. C'est ce point de vue de l'appelant qui explique l'appellation d'*horloge d'activation* du nœud.

Reprenons les flots de la section 1.2.1 et annotons le source avec leurs horloges. L'annotation d'horloge utilise deux doubles points, pour la différencier de l'annotation de type.

```
c = period3() :: . ;
y = (x when c) :: . on (true = c);
t = (x whenot c) :: . on (false = c);
d = period3() :: . on (true = c);
z = (y when d) :: . on (true = c) on (true = d);
```

Dans notre chronogramme, `c` était présent à « tous » les instants, car il n'y a pas de contraintes sur `c`. C'est pour cela qu'il est mis sur l'horloge de base du nœud, soit $\mathrm{clock}\,(\texttt{c}) = \bullet$. Le flot `y` est

présent quand `c` est présent et vrai, d'où son horloge $\mathrm{clock}\,(\texttt{c})$ `on (true = c)`. Le flot `z` vaut `y` quand `d` est vrai, $\mathrm{clock}\,(\texttt{y})$ `on (true = d)`. Ceci force `d` à être présent en même temps que `y`, ce qui est possible car `d` n'a pas d'autres contraintes.

1.2.4 *Nota bene* **Pointeaux et horloges**
En considérant `c` toujours présent (sur l'horloge de base globale **true**), nous avons $\mathrm{clock}\,(\texttt{y}) = (\mathrm{true.false.false})^\omega = [\![\texttt{c}]\!]^{\mathcal{K}}$. Il est alors tentant et courant de dire que « `c` est l'horloge de `y` ». Ceci est source de beaucoup de confusions et nous voulons insister sur la distinction entre l'horloge qui est un mot mathématique booléen et le flot `c` qui est calculé par le programme et dont la présence est relative à l'horloge de base.

Nous appelons *pointeau* le flot `c`, car il est nécessaire pour *pointer* (indiquer la présence) d'autres flots, dont `y`. En Heptagon, les pointeaux sont des flots de type finis. Ceci permet de bien faire la différence entre un pointeau qui peut être de type `ts`, `abc`, `int{<4}`, etc. tandis que l'horloge est toujours booléenne. C'est aussi pour souligner que `c` a le même rôle pour `t` et `y` que nous notons symétriquement leurs horloges • `on (false = c)` et • `on (true = c)`.

Dans Lucid Synchrone V3, les pointeaux sont déclarés avec le mot clef `clock` [CP03b].

1.2.5 *Définition* **Sous horloge**
L'horloge **ck**′ est une sous-horloge de **ck**, noté avec la relation d'ordre partiel $\mathbf{ck}' \leq \mathbf{ck}$ ssi à tout instant, l'horloge **ck**′ est vraie implique que **ck** soit vraie :

$$\mathbf{ck}' \leq \mathbf{ck} \qquad \Longleftrightarrow \qquad \forall k, \mathbf{ck}'[k] \implies \mathbf{ck}[k]$$

La valeur d'une horloge dépend possiblement des entrées du programme. En Heptagon, la relation de sous-horloge est approximée structurellement sur les horloges données par le compilateur en observant que pour tout pointeau `p` et valeur i :

$$ck\ \texttt{on (}i\texttt{ = p)} \leq ck$$

1.2.6 *Propriété* **Arbre d'horloges**
Les horloges données par le compilateur forment des arbres, tels que toute horloge est sous-horloge de ses parents.

1.2.6 Signature d'horloge

1.2.6.1 L'exemple de `current`

Pour assurer la modularité de la compilation, le calcul d'horloge assigne à chaque fonction une *signature d'horloge* σ de la grammaire suivante, avec x un nom de variable :

$$\sigma ::= \forall \bullet \forall x, \cdots x.(x :: ck, \cdots, x :: ck) \to (x :: ck, \cdots, x :: ck)$$

Considérons le nœud `current` inspiré de la primitive du même nom en Lustre. Sa sortie vaut la dernière valeur reçue pour son entrée `x`. La présence de `x` est donnée par une entrée supplémentaire `c` :

```
node current(x : int; c : bool)
returns (y : int)
var curr_y : int;
let
  curr_y = 0 fby y;
  y = merge c x
              (curr_y when not c);
tel
```

c	true	false	true	true	false	false
x	x_1		x_2	x_3		
curr_y	0	x_1	x_1	x_2	x_3	x_3
y	x_1	x_1	x_2	x_3	x_3	x_3

La signature donnée par le compilateur est :
`val current(x : int :: . on (true = c); c : bool :: .) returns (y : int :: .)`
Nous l'écrivons $\forall \bullet \forall c.(\bullet \texttt{ on } c, c :: \bullet) \to (\bullet)$ en ne gardant que les noms qui servent de pointeau.

1.2.7 *Remarque* En Lustre, `current x` est une expression valant `x` s'il est présent et sa dernière valeur sinon. L'horloge de `x` est implicitement utilisée pour donner une sémantique à cette expression. En Heptagon, la sémantique de Kahn ne dépend jamais des horloges, ce qui requiert l'ajout de l'entrée `c`.

1.2.6.2 Annotations

Les horloges et signatures sont inférées par Heptagon. Si le programmeur annote le code source, ses indications sont prises en compte et vérifiées. En première ligne de `current`, il peut écrire `node current(x : int :: . on c; c : bool :: .)`. Dans la suite, pour aider la lecture, nous annoterons les entrées et sorties qui ne sont pas sur l'horloge de base. De plus, nous utiliserons un sucre syntaxique accepté par le compilateur, omettant « `:: .` ». Ainsi nous écrirons : `node current(x : int on c; c : bool)`

1.2.6.3 Des pointeaux en sortie

Il est aussi permis d'avoir des sorties de rythme lent. Les pointeaux d'une sortie peuvent être des entrées mais aussi des sorties, comme dans le nœud `positifs` :

```
node positifs (x : int) returns (y : int on c; c : bool)
let
  c = (x >= 0);
  y = x when c;
tel
```

La signature d'horloge inférée par le compilateur est $\forall \bullet, \forall c.(\bullet) \to (\bullet \texttt{ on } c, c :: \bullet)$. Notez que le flot `c` doit être en sortie, sinon l'horloge de `y` ne pourrait être connue par l'appelant.

1.2.8 *Nota bene* Tandis qu'une sortie peut être pointée par des entrées et des sorties, une entrée ne peut l'être que par d'autres entrées.

1.2.6.4 Signature de `merge`

Tout comme les fonctions, les primitives peuvent être munies d'une signature d'horloges. Pour la mettre en évidence, il suffit de l'encapsuler dans une fonction. La spécialisation de `merge` à un pointeau de type `int{<2}` est :

```
fun merge2 (p : int{<2}; x0 : int; x1 : int) returns (y : int)
let y = merge p (0 -> x0) (1 -> x1); tel
```

Elle est de signature $\forall\bullet, \forall p.(p :: \bullet, \bullet \texttt{ on } (0 = p), \bullet \texttt{ on } (1 = p)) \to \bullet$.

1.3 Automates et réinitialisation

Bien que la possibilité de réinitialiser un morceau de programme soit intéressante, son développement a suivi l'intérêt pour les automates. Les travaux originaux de F. Maraninchi et Y. Rémond proposent avec Matou [MR98 ; MRR00] un langage d'automates hiérarchiques intégrants des équations à la Lustre. De cette inspiration, G. Hamon et M. Pouzet ont intégré des structures de contrôle, ainsi que la réinitialisation dans Lucid Synchrone [HP00]. Les automates dans Lucid Synchrone sont présentés avec une compilation source à source dans un article [CPP05] qui a servi aussi au compilateur Scade.

1.3.1 Automates

Considérons le code suivant à deux modes :

```
node updown (x : int)
returns (y :int)
let
  automaton
    state Up do
      y = (-2) fby (y + x);
    until y >= 1 then Down
    state Down do
      y = 2 fby (y - x);
    until y <= -1 then Up
  end;
tel
```

Nous notons U l'état `Up` et D pour `Down`. Avec l'entrée constante 1^ω :

état	U	U	U	U	D	D	D	D	U	U	$\cdots$
x	1	1	1	1	1	1	1	1	1	1	$\cdots$
y	-2	-1	0	1	2	1	0	-1	-2	-1	$\cdots$

Avec une entrée changeante :

état	U	U	U	D	D	D	U	U	U	D	$\cdots$
x	0	7	0	2	1	0	-3	4	2	1	$\cdots$
y	-2	-2	5	2	0	-1	-2	-5	-1	1	$\cdots$

L'automate débute dans son premier état (`Up`). Dans cet état, la sortie `y` est définie comme valant -2 suivie de l'entrée plus sa précédente valeur. Ainsi, si l'entrée est stationnaire à 1, `y` est incrémenté de 1 à chaque instant. Quand `y >=1` l'automate effectue une transition de `Up` vers `Down`. Le mot clef `until` indique une transition faible, c'est-à-dire effectuée à la fin de l'instant. Une transition forte, effectuée au début de l'instant serait indiquée avec le mot clef `unless`.

|| 1.3.1 *Nota bene* Les états sont exclusifs, seul le code de l'état actif est exécuté.

L'utilisation du mot clef `then` indique que la transition s'effectue en réinitialisant l'état dans lequel on arrive. Cela ne change rien à la première transition vers `Down`, puisque l'état n'a pas encore été activé, le registre est toujours dans son état initial. Par contre, ceci s'observe quand on revient dans `Up`, la sortie `y` prend alors la valeur d'initialisation du registre -2 et non la valeur précédemment stockée.

1.3.2 Réinitialisation

Une compilation possible de l'automate `updown` est le code ci-dessous.

```
type st_updown = Up | Down
node _updown (x : int) returns (y :int)
var to_up, to_down : bool; st : st_updown; yup : int; ydown : int;
let
```

```
  to_up = (ydown <= -1);
  to_down = (yup >= 1);
  st = Up fby merge st (Up -> if to_down then Down else Up)
                       (Down -> if to_up then Up else Down);
  reset
    yup = -2 fby (yup + (x when (Up=st)));
  every to_up;
  reset
    ydown = 2 fby (ydown - (x when (Down=st)));
  every to_down;
  y = merge st (Up -> yup) (Down -> ydown);
tel
```

L'état `st` est initialisé à `Up` et change si `to_up` ou `to_down` l'indiquent. La variable `yup` (resp. `ydown`) représente `y` quand l'automate est dans l'état `Up` (resp. `Down`). Son équation est comme celle de `y` dans la branche `Up`, sauf qu'elle n'est active que quand l'automate est dans cette branche (`Up=st`), ce qui requiert d'échantillonner l'entrée `x`. De plus, cette équation est réinitialisée à chaque fois que l'automate arrive dans l'état `Up`. Les instants où il y a transition allant dans l'état `Up` sont calculés avec `to_up`, respectivement `Down` avec `to_down`.

Le point qui nous intéressera est l'utilisation de la construction `reset` eqs `every c`, qui réinitialise l'ensemble d'équations eqs quand la condition `c` est vraie. Notez que notre exemple utilise une condition `c` qui n'a pas la même horloge que ce qu'elle réinitialise. Nous en parlons en détail en section 2.6.

1.4 Principe de substitution

Heptagon respecte le principe de substitution : tout appel de fonction peut être remplacé par sa définition sans changer la sémantique du programme. Le programmeur peut forcer cela en ajoutant le mot clef `inline` avant l'appel d'un nœud. Bien évidemment, dans ce cas le compilateur doit avoir accès au code source de la fonction à inliner.

1.5 Paramètres statiques

Les paramètres statiques, déclarés entre chevrons, sont des paramètres dont la valeur est connue statiquement à l'appel de la fonction. Ils permettent d'écrire du code générique sans changement profond dans la génération de code ou les diverses analyses.

L'exemple de `period` Le nœud `period3` est équivalent à l'instanciation `period<<3>>` de la déclaration suivante :

```
node period<< n : int | (n > 0) >> () returns (c : bool)
var cpt, next : int;
let
  next = if (cpt = n) then 1 else (cpt + 1);
  cpt = 1 fby next;
  c = (cpt = 1);
tel
```

Les paramètres statiques sont toujours entre double chevrons `<<` ... `>>`. Il est possible de rajouter des contraintes sur ces valeurs qui seront vérifiées pour chaque instanciation. Ici la

contrainte est `n>0` pour assurer que la période utilisée est strictement positive : l'expression `period<<-1>>()` lèvera une erreur.

Ces paramètres sont macro expansés quand nous générons du C. Une partie des paramètres peuvent être passés en argument au constructeur de la classe générée, ce que fait notre générateur Java.

Ordre supérieur statique Outre des constantes, nous utiliserons dans ce manuscrit la possibilité de passer des fonctions en paramètres :

```
node double<< node f(int) returns (int) >> (x : int) returns (y : int)
let
  y = f(f(x));
tel
```

Paramètre statique du registre synchrone `fby«n»` Pour exprimer du parallélisme, nous aurons besoin d'avoir des boucles de rétroaction avec un délai de plusieurs instants. Ceci nécessite l'utilisation de plusieurs `fby` composés. La primitive `fby` accepte un entier en paramètre statique, pour indiquer combien de fois il insère son premier argument en tête du deuxième. Par exemple : `0 fby (0 fby x)` ≡ `0 fby<<2>> x`. Cet opérateur existe aussi en SCADE.

1.6 Les tableaux

Finissons la présentation des constructions du langage avec les tableaux qui profitent grandement des paramètres statiques. Un tableau de taille `n`, dont les éléments sont de type `t`, a pour type `t^n`. Différentes constructions et itérateurs permettent de manipuler les tableaux.

Décrivons l'itérateur `map` qui nous sera utile par la suite. L'appel `map<<n>> f (x)` indique que `x` est un tableau de taille `n`. Il applique une instance du nœud `f` par élément du tableau. Ci-dessous, nous utilisons un tableau constant en argument, avec la liste des éléments écrite entre crochets :

`map<<2>> sum ([1, 3])`	$[1,3]$	$[2,6]$	$[3,9]$	$[4,12]$	$[5,15]$	...

Notez que cette expression est équivalente à l'écriture explicite du tableau : `[sum(1), sum(3)]`.

1.7 Conclusion

Pour résumer, Heptagon a une syntaxe et une sémantique proche de celle de LUSTRE. Comme LUSTRE, c'est une langage fonctionnel synchrone du premier ordre. Parmi les primitives, la seule différence notable est l'utilisation de `merge` à la place de `current` (cf. section 1.2.1). Nous avons surtout insisté sur les primitives `fby`, `when` et `merge` car elles forment le noyau du langage. Toutes les constructions plus complexes, tels les automates, sont compilées source à source. La génération de code (Java, C, etc.) s'effectue à partir du noyau. Cette approche nous permet de générer du code optimisé car le noyau flot de données est pour cela adéquat. L'analyse et la génération de code sont effectuées sur chaque fonction de manière séparée.

La modularité et le passage par un noyau réduit facilitent les modifications et extensions. Parmi les extensions intégrées dans le compilateur Heptagon, notons la synthèse de contrôleurs [DMR10], la génération de code VHDL [GP10], le suréchantillonnage LHO [Ger11], l'optimisation mémoire des tableaux [Ger+12] et l'intégration des futures [CGP12]. D'autres extensions sont dans des branches plus ou moins maintenues comme la génération de code

CUDA et OpenCL. Nous avons participé au développement du compilateur et à une majorité des expérimentations.

Outre des expériences de compilation, Heptagon est utilisé comme plateforme dans des projets de recherche. Il est aussi utile dans le cadre de stages de Master pour programmer divers exemples, de la programmation de protocoles de cohérence de caches à l'incrustation d'une vidéo dans une autre (*picture in picture*). Le but de ces exemples est de comprendre les différentes manières de programmer dans un langage synchrone flot de données, ainsi que les extensions et optimisations qui mériteraient d'être étudiées.

Notons finalement que les travaux sur la certification de la compilation, faits par Auger [Aug13], portent sur un langage et un compilateur jumeau d'Heptagon.

Chapitre 2

Sémantiques et compilation d'Heptagon

2.1 Sémantique de Kahn

Heptagon est un langage synchrone fonctionnel flots de données. Grâce à ceci, il est possible de lui donner une sémantique de Kahn. La première sémantique de LUSTRE [Ber86] était dénotationnelle, proche d'un sémantique de Kahn. Comme nous l'indiquions en remarque 1.2.7, l'utilisation de `merge` à la place de `current` nous permet de donner une sémantique de Kahn sans utiliser les horloges. La sémantique présentée suit celle de LUCID SYNCHRONE [Pou02]. Avant de présenter cette sémantique, nous proposons un noyau réduit, ne contenant que les primitives de bas niveau. Ce noyau correspond à un des langages intermédiaires du compilateur.

2.1.1 Noyau fonctionnel du langage

Le noyau fonctionnel du langage Heptagon est présenté en figure 2.1. Un programme est composé d'une suite de définitions de fonctions auxiliaires, puis d'une expression qualifiée de principale, définissant le comportement du programme. Les entrées et sorties d'un programme sont celles de son expression principale. Une définition de fonction est la définition de sorties, calculées à l'aide des entrées et d'un ensemble d'équations locales à la fonction. La syntaxe omet la déclaration des variables locales qui sont implicitement toutes les variables définies par les équations locales sauf les sorties. Les équations d'un nœud sont récursives, leur ordre n'est pas significatif.

2.1.2 Domaine de Kahn

Ce que nous appelons la sémantique de Kahn [Kah74] est une sémantique dénotationnelle à la Scott [Sco69], mais les valeurs utilisées sont dans des domaines de Kahn.

Les valeurs de ce domaine sont les *flots* (mots finis et infinis). Si T est un ensemble de valeurs appelées « scalaires », alors T^* est l'ensemble des mots finis de valeurs de T, T^ω l'ensemble des mots infinis et $T^\infty = T^* \cup T^\omega$ l'ensemble des flots d'éléments de T. Pour tout flot d'éléments scalaires $\mathbf{x}$, sa $k$ième valeur est notée $\mathbf{x}[k]$. Nous notons ε la séquence vide (de longueur zéro) et le point est l'opérateur de concaténation. Pour tout flot fini $\mathbf{u}$ de T^*, flot $\mathbf{v}$ de T^∞ et scalaire u de T, la concaténation est telle que :

$$(u.\mathbf{u}).\mathbf{v} = u.(\mathbf{u}.\mathbf{v}) \qquad\qquad \varepsilon.\mathbf{v} = \mathbf{v}$$

Programme :

$P ::=$	ds `;` e	définitions et l'expression principale
$ds ::=$	d `;` ds	séquence de définitions
$d ::=$	f `(` x `,` $\cdots$ `,` x `) = (` x `,` $\cdots$ `,` x `)` **`with`** eqs	définition

Équations et expressions :

$eqs ::=$	**`rec`** eq **`and`** $\cdots$ **`and`** eq	équations récursives
$eq ::=$	`(` x `,` $\cdots$ `,` x `) =` e	équation
$e ::=$	i	valeur instantanée
	$\mid x$	variable
	$\mid op(e, e)$	opérateur binaire instantanées (`+`, `-`, etc)
	$\mid e$ **`fby`** e	registre synchrone initialisé
	$\mid e$ **`when`** e	échantillonnage
	$\mid$ **`merge`** e `(`i `->` e`)` $\cdots$ `(`i `->` e`)`	complémentation
	$\mid f$ `(`e `,` $\cdots$ `,` e`)`	appel de fonction
$i ::=$	**`true`** $\mid$ **`false`** $\mid$ `0` $\mid$ `1` $\mid \ldots$	

FIGURE 2.1: Noyau synchrone flot de données

L'*ordre partiel* $\sqsubseteq$ est l'ordre préfixe :

$$\mathbf{v} \sqsubseteq \mathbf{v}' \iff \begin{cases} \exists \mathbf{q} \in T^\infty, \quad \mathbf{v}.\mathbf{q} = \mathbf{v}' & \text{si } \mathbf{v} \in T^* \\ \mathbf{v} = \mathbf{v}' & \text{sinon} \end{cases}$$

Le triplet $(T^\infty, \sqsubseteq, \varepsilon)$ est un *domaine de calcul* [KP78]. En particulier, c'est un ordre partiel complet, dont le plus petit élément est ε.

Pour de plus amples rappels sur les domaines et particulièrement celui de Kahn, l'annexe IV rappelle les principales définitions et propriétés. Nous entrerons dans les détails en partie III. Ici, seule la notion de chaîne et de fonction continue d'un domaine de calcul nous sont utiles.

Tout ensemble non vide totalement ordonné est appelé une *chaîne*. Elle est notée comme une suite ordonnée indexée par un entier i non nul : $(\mathbf{X}_i)_{i>0}$. La limite de cette chaîne $\lim_{i\to\infty} \mathbf{X}_i$ est la borne supérieure de la chaîne, qui existe par définition d'un ordre partiel complet.

Un domaine de Kahn est un domaine $(T^\infty, \sqsubseteq, \varepsilon)$ ou bien tout domaine construit par produit et/ou exponentiation d'un domaine de Kahn. Un produit de flots est noté avec une majuscule $\mathbf{X}$ et sa $i$ème composante est notée $\mathbf{X}^{(i)}$. L'ordre entre deux tuples de flots est calculé composantes par composantes : $\mathbf{X} \sqsubseteq \mathbf{Y} \iff \forall i, \mathbf{X}^{(i)} \sqsubseteq \mathbf{Y}^{(i)}$. L'ordre entre deux fonctions est extensionnel : $f \sqsubseteq g \iff \forall \mathbf{X}, f(\mathbf{X}) \sqsubseteq g(\mathbf{X})$.

Toute fonction f d'un domaine $\mathcal{D}_e$ dans un domaine $\mathcal{D}_s$ est *continue* ssi elle préserve la limite des chaînes : $\forall (\mathbf{X}_i)_i, \lim_{i\to\infty} f(\mathbf{X}_i) = f(\lim_{i\to\infty} \mathbf{X}_i)$. Une fonction continue est aussi *monotone*, c'est-à-dire qu'elle préserve l'ordre. Dans un domaine de Kahn, si un flot $\mathbf{X}$ est préfixe d'un autre $\mathbf{X} \sqsubseteq \mathbf{X}'$, il correspond moralement à son passé. La monotonie est la préservation de cet ordre, c'est-à-dire $f(\mathbf{X}) \sqsubseteq f(\mathbf{X}')$. Pour toute fonction continue, nous notons le plus petit point

Fonctions auxiliaires, définies comme les plus petites fonctions telles que :

$$\begin{aligned}
\text{const}_i^{\mathcal{K}}() &= i^\omega \\
op^{\mathcal{K}}(v_1.\mathbf{v_1}, v_2.\mathbf{v_2}) &= v.\, op^{\mathcal{K}}(\mathbf{v_1}, \mathbf{v_2}) \text{ avec } v = op(v_1, v_2) \\
\text{fby}^{\mathcal{K}}(v.\mathbf{v}, \mathbf{u}) &= v.\mathbf{u} \\
\text{when}^{\mathcal{K}}(v.\mathbf{v}, \text{true}.\mathbf{p}) &= v.\,\text{when}_i^{\mathcal{K}}(\mathbf{v}, \mathbf{p}) \\
\text{when}^{\mathcal{K}}(v.\mathbf{v}, \text{false}.\mathbf{p}) &= \text{when}_i^{\mathcal{K}}(\mathbf{v}, \mathbf{p}) \\
\text{merge}^{\mathcal{K}}(k.\mathbf{k}, \mathbf{v_0}, \cdots, v.\mathbf{v}_k, \cdots, \mathbf{v_n}) &= v.\,\text{merge}^{\mathcal{K}}(\mathbf{k}, \mathbf{v_0}, \ldots, \mathbf{v}_k, \cdots, \mathbf{v_n})
\end{aligned}$$

Interprétation des expressions :

$$\begin{aligned}
[\![i]\!]^{\mathcal{K}}_{H,\rho} &= \text{const}_i^{\mathcal{K}}() & [\![x]\!]^{\mathcal{K}}_{H,\rho} &= \rho(x) \\
[\![op(e_1, e_2)]\!]^{\mathcal{K}}_{H,\rho} &= op^{\mathcal{K}}\Big([\![e_1]\!]^{\mathcal{K}}_{H,\rho}, [\![e_2]\!]^{\mathcal{K}}_{H,\rho}\Big) & [\![f\texttt{(}e_1\texttt{, }\cdots\texttt{, }e_n\texttt{)}]\!]^{\mathcal{K}}_{H,\rho} &= H(f)\Big([\![e_1]\!]^{\mathcal{K}}_{H,\rho}, \cdots, [\![e_n]\!]^{\mathcal{K}}_{H,\rho}\Big) \\
[\![e_1 \texttt{ when } e_2]\!]^{\mathcal{K}}_{H,\rho} &= \text{when}^{\mathcal{K}}\Big([\![e_1]\!]^{\mathcal{K}}_{H,\rho}, [\![e_2]\!]^{\mathcal{K}}_{H,\rho}\Big) & [\![e_1 \texttt{ fby } e_2]\!]^{\mathcal{K}}_{H,\rho} &= \text{fby}^{\mathcal{K}}\Big([\![e_1]\!]^{\mathcal{K}}_{H,\rho}, [\![e_2]\!]^{\mathcal{K}}_{H,\rho}\Big) \\
\multicolumn{4}{r}{[\![\texttt{merge } e' \texttt{ (0 -> } e_0\texttt{)} \cdots \texttt{(}n\texttt{ -> } e_n\texttt{)}]\!]^{\mathcal{K}}_{H,\rho} = \text{merge}^{\mathcal{K}}\Big([\![e']\!]^{\mathcal{K}}_{H,\rho}, [\![e_0]\!]^{\mathcal{K}}_{H,\rho}, \cdots, [\![e_n]\!]^{\mathcal{K}}_{H,\rho}\Big)}
\end{aligned}$$

Interprétation des équations :

$$\begin{aligned}
[\![\texttt{(}x_1\texttt{, }\cdots\texttt{, }x_n\texttt{) = }e]\!]^{\mathcal{K}}_{H,\rho} &= [x_i \mapsto ([\![e]\!]^{\mathcal{K}}_{H,\rho})^{(i)} \mid 1 \leq i \leq n] \\
[\![\texttt{rec } eq_1 \texttt{ and } \cdots \texttt{ and } eq_n]\!]^{\mathcal{K}}_{H,\rho} &= \text{fix}\,(\lambda\rho'.\, \uplus_{1\leq i\leq n} [\![eq_i]\!]^{\mathcal{K}}_{H,\rho\uplus\rho'})
\end{aligned}$$

Interprétation d'un programme :

$$\begin{aligned}
[\![ds\texttt{;}e]\!]^{\mathcal{K}}_{\rho_0} &= [\![e]\!]^{\mathcal{K}}_{H,\rho_0} \qquad \text{avec } H = [\![ds]\!]^{\mathcal{K}}_{\emptyset} \\
[\![d\texttt{;}ds]\!]^{\mathcal{K}}_{H} &= H' \uplus [\![ds]\!]^{\mathcal{K}}_{H\uplus H'} \qquad \text{avec } H' = [\![d]\!]^{\mathcal{K}}_{H} \\
[\![f\texttt{(}x_1\texttt{, }\cdots\texttt{, }x_n\texttt{) = (}y_1\texttt{, }\cdots\texttt{, }y_m\texttt{) with } eqs]\!]^{\mathcal{K}}_{H} &= [f \mapsto \lambda(\mathbf{x_1}, \cdots, \mathbf{x_n}).(\rho(y_1), \cdots, \rho(y_m))] \\
&\quad \text{avec } \rho = [\![eqs]\!]^{\mathcal{K}}_{H,\rho_{in}} \quad \text{et} \quad \rho_{in} = [x_1 \mapsto \mathbf{x_1}, \cdots, x_n \mapsto \mathbf{x_n}]
\end{aligned}$$

FIGURE 2.2: Sémantique de Kahn

fixe $\text{fix}(f)$, il se calcule d'après le théorème de Knaster-Tarski (cf. théorème 0.1) avec l'égalité $\text{fix}(f) = \lim_{i\to\infty} f^i(\boldsymbol{\varepsilon})$.

2.1.3 Sémantique de Kahn

La sémantique de Kahn du noyau fonctionnel de la figure 2.1 est donnée en figure 2.2. Le cœur de la sémantique est porté par les fonctions auxiliaires calculant la sémantique des primitives. Dans la figure, nous ne fournissons que les filtrages qui produisent une valeur, dans tous les autres cas, comme nous prenons la plus petite fonction, elle renvoie ε. Par exemple pour le registre synchrone, si sa première entrée a au moins un scalaire v, alors il le met en tête de son deuxième argument. Par contre, si sa première entrée est vide, il n'y a pas de filtrage explicite, donc son résultat est vide ($\text{fby}^{\mathcal{K}}(\varepsilon, \mathbf{u}) = \varepsilon$). Les opérateurs binaires op sont liftés point à point. L'échantillonnage (`when`) consomme une valeur de ses deux entrées, quand la complémentation

(`merge`) ne consomme une valeur que de sa première et sa $k+1$ ième, avec k la valeur lue à sur la première entrée. Il est immédiat de constater que ces fonctions auxiliaires sont continues.

L'interprétation d'un programme collecte en premier lieu les définitions de fonction dans H, puis interprète l'expression principale e avec l'environnement ρ_0 qui contient les valeurs des variables libres de e qui sont les entrées du programme.

L'interprétation d'une expression $[\![e]\!]^{\mathcal{K}}_{H,\rho}$ renvoie une valeur et se fait avec l'environnement ρ qui est une fonction des noms de variable vers leur valeur. L'environnement H associe aux noms de fonctions l'interprétation de leur définition. Cette dernière est une fonction prenant en argument la valeur des entrées et retournant celle des sorties. L'interprétation d'une équation est un environnement associant à chaque variable du membre gauche sa valeur.

Le point fixe pour les équations récursives est bien défini puisqu'il s'applique à la composition de fonctions continues, comme nous le montrons ci-dessous.

Les interprétations des expressions et des équations sont des fonctions continues de ρ, si nous munissons le domaine des environnements avec l'ordre partiel complet extensionnel :

$$\rho \sqsubseteq \rho' \quad \Longleftrightarrow \quad \forall x, \rho(x) \sqsubseteq \rho'(x)$$

Le plus petit environnement est l'environnement vide noté $[\,]$, qui associe à toute variable la valeur ε. Pour tout environnement ρ_1 et ρ_2 disjoints (pour tout x, $\rho_1(x) = \varepsilon \vee \rho_2(x) = \varepsilon$), leur union disjointe est définie par :

$$\rho_2 \uplus \rho_1 = \rho_1 \uplus \rho_2 = x \mapsto \begin{cases} \rho_2(x) & \text{si } \rho_1(x) = \varepsilon \\ \rho_1(x) & \text{si } \rho_2(x) = \varepsilon \end{cases}$$

L'union disjointe est une fonction continue de ses deux arguments, la seule précaution à prendre est d'assurer la disjonction des environnements. Ceci est le cas car une variable est soit une entrée, soit définie par une unique équation, ce qui est vérifié par le compilateur.

2.1.1 *Exemple* Exemple de calcul du point fixe avec le nœud `sum`

Considérons le programme suivant dans la syntaxe du noyau :

```
sum(x) = (y) with rec t = 0 fby y and y = t + x;
sum(v)
```

Ce programme est évalué dans l'environnement $\rho_0 = [\mathtt{v} \mapsto \mathbf{v}]$ avec $\mathbf{v} = v_1.v_2.v_3.v_4.\cdots$. Le point qui nous intéresse est le calcul du point fixe des équations dans l'application de `sum`, c'est-à-dire $\mathrm{fix}\,(\lambda\rho'.[\![\mathtt{t = 0 fby y}]\!]^{\mathcal{K}}_{H,\rho'\uplus\rho_0} \uplus [\![\mathtt{y = t + x}]\!]^{\mathcal{K}}_{H,\rho'\uplus\rho_0})$. Le point fixe est calculé en prenant un premier environnement vide ρ'_0 et chaque itération nous fait passer de l'environnement n à celui $n+1$:

$$\rho'_0 = [\mathtt{t} \mapsto \varepsilon, \mathtt{y} \mapsto \varepsilon] \qquad \rho'_1 = [\mathtt{t} \mapsto 0, \mathtt{y} \mapsto \varepsilon]$$
$$\rho'_2 = [\mathtt{t} \mapsto 0, \mathtt{y} \mapsto 0 + v_1] \qquad \rho'_3 = [\mathtt{t} \mapsto 0.(0 + v_1), \mathtt{y} \mapsto 0 + v_1]$$
$$\rho'_4 = [\mathtt{t} \mapsto 0.(0 + v_1), \mathtt{y} \mapsto (0 + v_1).(0 + v_1 + v_2)] \qquad \rho'_5 = \cdots \text{ etc.}$$

Notez qu'il est primordial que le registre puisse fournir une sortie alors même que son deuxième argument est ε, sinon, ρ'_1 serait égal à ρ'_0 et le point fixe s'arrêterait là. Notez aussi que le point fixe ne produit pas une valeur de sortie à chaque itération (ici, il en faut deux pour une valeur). La sortie `y` est bien le flot attendu $v_1.(v_1 + v_2).(v_1 + v_2 + v_3).\cdots$.

La mise en parallèle par point fixe est la clef de voûte de la sémantique de Kahn. C'est ce qui lui confère son élégance et sa robustesse. Le principe de Kahn [LS89] stipule que le point

fixe de fonctions continues modélise un ensemble de nœuds (continus) communiquant avec des files de taille non bornée. La continuité des fonctions permet de considérer que chaque nœud produit petit à petit ses sorties, de manière concurrente des autres. En effet, ceci est une autre manière de calculer le point fixe de multiples équations. Nous en discuterons en détail en partie III. Avoir une sémantique de Kahn permet un raisonnement mathématique concis et est l'assurance de pouvoir répartir les calculs sans surcoût.

Pourtant, deux questions ne sont pas étudiées par la sémantique de Kahn, la productivité du réseau et la quantité de mémoire nécessaire. Les deux viennent de la même difficulté. Dans la sémantique de Kahn, la longueur des flots n'est pas contrôlée.

2.2 Sémantique de Kahn Synchrone

La réponse des langages synchrones aux problèmes des réseaux de Kahn est de restreindre les réseaux que l'on peut écrire à un sous-ensemble de ceux qui sont productifs et ont besoin de files de taille au maximum 1. Pour ce faire, une notion d'instant global est définie. À chaque instant, tous les flots doivent porter une valeur scalaire. Grâce à ceci, le point fixe va être calculé instant par instant, sur les scalaires, au lieu d'être calculé sur les flots en entier. Chaque élément du réseau va consommer une et une seule valeur de chacune de ses entrées et produire une et une seule valeur pour chacune de ses sorties.

Notre langage source a des flots sur des rythmes différents. Par exemple, l'opérateur de filtrage `when` ne produit pas forcément une valeur pour chaque valeur consommée. Pour mettre tous les flots sur le même rythme, « les *synchroniser* », une valeur scalaire spéciale abs est ajoutée. Elle représente l'absence de valeur. Un flot qui était de type T^∞ dans la sémantique de Kahn devient un flot de type $(T + \{abs\})^\infty$ dans la sémantique synchrone. Cette approche vient des travaux sur la co-itération [CP98] de Lucid Synchrone et est utilisée dans [HP00].

2.2.1 *Nota bene* **Notations avec abs explicite et w**
Par la suite, les lettres comme v, u, p, c, etc sont réservées pour spécifier un scalaire différent de abs. Seule la lettre w sera utilisée pour dénoter un scalaire pouvant être égal à abs.

Les opérateurs binaires liftés requièrent une valeur de leurs deux entrées :

$$\begin{aligned} op^{\mathcal{S}}(v_1.\mathbf{v_1}, v_2.\mathbf{v_2}) &= v.\, op^{\mathcal{S}}(\mathbf{v_1}, \mathbf{v_2}) \text{ avec } v = op(v_1, v_2) \\ op^{\mathcal{S}}(abs.\mathbf{v_1}, abs.\mathbf{v_2}) &= abs.\, op^{\mathcal{S}}(\mathbf{v_1}, \mathbf{v_2}) \qquad (*) \end{aligned}$$

Si une entrée est absente, l'autre doit l'être aussi, sinon il faudrait stocker la valeur en attendant l'autre entrée. Or nous voulons que seul l'opérateur `fby` ait un état.

2.2.2 *Nota bene* **Plus petites fonctions acceptant ces filtrages**
Comme toujours, nous prenons la plus petite fonction qui satisfait les filtrages donnés. Ainsi, par exemple $op^{\mathcal{S}}(abs.\mathbf{v_1}, v_2.\mathbf{v_2}) = \varepsilon$ ou bien $op^{\mathcal{S}}(v_1.\mathbf{v_1}, \varepsilon) = \varepsilon$.

L'utilité de abs apparait quand l'échantillonnage filtre une valeur. Dans ce cas la sortie est abs bien que les entrées soient présentes. Pour ne pas avoir à stocker des valeurs, si une des entrées est absente, l'autre est requise de l'être aussi :

$$\begin{aligned} \text{when}^{\mathcal{S}}(v.\mathbf{v}, \text{true}.\mathbf{p}) &= v.\, \text{when}^{\mathcal{S}}(\mathbf{v}, \mathbf{p}) \\ \text{when}^{\mathcal{S}}(v.\mathbf{v}, \text{false}.\mathbf{p}) &= abs.\, \text{when}^{\mathcal{S}}(\mathbf{v}, \mathbf{p}) \\ \text{when}^{\mathcal{S}}(abs.\mathbf{v}, abs.\mathbf{p}) &= abs.\, \text{when}^{\mathcal{S}}(\mathbf{v}, \mathbf{p}) \qquad (*) \end{aligned}$$

Pour la complémentation, les entrées non nécessaires doivent être absentes pour ne pas avoir besoin de stockage. Si la première entrée est absente, aucun choix ne peut être fait, donc toutes les autres doivent aussi l'être, ainsi que la sortie :

$$\begin{aligned} \mathrm{merge}^{\mathcal{S}}(k.\mathbf{k}, \mathit{abs}.\mathbf{v_0}, \cdots, v.\mathbf{v}_k, \cdots, \mathit{abs}.\mathbf{v_n}) &= v.\,\mathrm{merge}^{\mathcal{S}}(\mathbf{k}, \mathbf{v_0}, \cdots, \mathbf{v}_k, \cdots, \mathbf{v_n}) \\ \mathrm{merge}^{\mathcal{S}}(\mathit{abs}.\mathbf{k}, \mathit{abs}.\mathbf{v_0}, \cdots, \mathit{abs}.\mathbf{v_n}) &= \mathit{abs}.\,\mathrm{merge}^{\mathcal{S}}(\mathbf{k}, \mathbf{v_0}, \cdots, \mathbf{v_n}) \qquad (*) \end{aligned}$$

2.2.1 Le registre synchrone

Contrairement à `when` et `merge`, la sémantique de Kahn synchrone de `fby` est délicate. Il doit consommer sa première entrée jusqu'à avoir un scalaire présent, le fournir, puis fournir sa deuxième entrée. Le but est d'avoir une mémoire d'un scalaire, similairement à un registre synchrone matériel. La technique usuelle est de requérir que les deux entrées aient la même horloge. De cette façon, au moment où la première entrée est présente, la seconde est stockée et sera utilisé comme prochaîne sortie. Nous notons en indice la valeur stockée. À l'initialisation, la mémoire est vide, pour le représenter nous mettons *abs* dans la mémoire. Si une des entrées est absente, l'autre l'est aussi et la mémoire est intouchée :

$$\begin{aligned} \mathrm{fby}^{\mathcal{S}}_{\mathit{abs}}(v.\mathbf{v}, u.\mathbf{u}) &= v.\,\mathrm{fby}^{\mathcal{S}}_{u}(\mathbf{v}, \mathbf{u}) \\ \mathrm{fby}^{\mathcal{S}}_{u'}(v.\mathbf{v}, u.\mathbf{u}) &= u'.\,\mathrm{fby}^{\mathcal{S}}_{u}(\mathbf{v}, \mathbf{u}) \\ \mathrm{fby}^{\mathcal{S}}_{w}(\mathit{abs}.\mathbf{v}, \mathit{abs}.\mathbf{u}) &= \mathit{abs}.\,\mathrm{fby}^{\mathcal{S}}_{w}(\mathbf{v}, \mathbf{u}) \qquad (*) \end{aligned}$$

2.2.3 *Remarque* Au deuxième filtrage, nous forçons la première entrée à être présente quand la seconde l'est, même si nous n'avons plus besoin de ses valeurs. Ceci est une simplification bienvenue même si la sémantique pourrait s'en passer.

Toutefois ces filtrages ne suffisent pas. En effet, `fby` ne peut pas être une fonction stricte, au sens où elle requiert une valeur de chacune de ses entrées avant de fournir une sortie. Comme nous l'avons vu en exemple 2.1.1, le registre doit fournir une valeur en sortie avant d'en avoir une sur sa deuxième entrée, sinon le point fixe reste bloqué à ε.

La solution est de rajouter le filtrage acceptant le flot vide en deuxième argument et fournissant la valeur stockée, ou la valeur d'initialisation si elle n'a pas déjà été fournie :

$$\mathrm{fby}^{\mathcal{S}}_{u}(\mathbf{v}, \varepsilon) = u \qquad\qquad \mathrm{fby}^{\mathcal{S}}_{\mathit{abs}}(v.\mathbf{v}, \varepsilon) = v$$

2.2.2 Les constantes

Autant les constantes paraissent triviales dans la sémantique de Kahn autant elles sont cruciales dans la sémantique synchrone. L'interprétation d'une constante nécessite de connaître les instants où elle doit être absente. Cette connaissance est la donnée de son *horloge* dont nous avons déjà discuté en section 1.2.5. L'horloge de la constante est un mot infini booléen tel que s'il vaut true à la $i$ème position, la constante doit être présente au $i$ème instant du programme.

L'interprétation synchrone d'une constante est dépendante de l'horloge qui lui est associée. Pour cette raison, nous allons décorer avec leurs horloges les constantes du noyau. L'idée est d'avoir la syntaxe i^{ck} avec ck l'horloge donnée par le compilateur à la constante i. Une fois que nous connaissons le mot $\mathbf{c}$ associé à l'horloge ck, nous le fournissons en argument à $\mathrm{const}^{\mathcal{S}}_{i}(\mathbf{c})$, définie par :

Interprétation des expressions :

$$\left[\!\left[i^{ck} \right]\!\right]^{\mathbf{b}}_{H,\rho} = \mathrm{const}^{\mathcal{S}}_i\left([\![ck]\!]^{\mathbf{b}}_{H,\rho}\right) \qquad [\![x]\!]^{\mathbf{b}}_{H,\rho} = \rho(x) \qquad [\![op(e_1,e_2)]\!]^{\mathbf{b}}_{H,\rho} = op^{\mathcal{S}}\left([\![e_1]\!]^{\mathbf{b}}_{H,\rho}, [\![e_2]\!]^{\mathbf{b}}_{H,\rho}\right)$$

$$[\![e_1 \texttt{ when } e_2]\!]^{\mathbf{b}}_{H,\rho} = \mathrm{when}^{\mathcal{S}}\left([\![e_1]\!]^{\mathbf{b}}_{H,\rho}, [\![e_2]\!]^{\mathbf{b}}_{H,\rho}\right) \qquad [\![e_1 \texttt{ fby } e_2]\!]^{\mathbf{b}}_{H,\rho} = \mathrm{fby}^{\mathcal{S}}\left([\![e_1]\!]^{\mathbf{b}}_{H,\rho}, [\![e_2]\!]^{\mathbf{b}}_{H,\rho}\right)$$

$$\left[\!\left[\texttt{merge } e' \texttt{ (0 -> } e_0\texttt{)} \cdots \texttt{(}n\texttt{ -> } e_n\texttt{)}\right]\!\right]^{\mathbf{b}}_{H,\rho} = \mathrm{merge}^{\mathcal{S}}\left([\![e']\!]^{\mathbf{b}}_{H,\rho}, [\![e_0]\!]^{\mathbf{b}}_{H,\rho}, \cdots, [\![e_n]\!]^{\mathbf{b}}_{H,\rho}\right)$$

$$\left[\!\left[f^{ck}\texttt{(}e_1\texttt{, }\cdots\texttt{, }e_n\texttt{)} \right]\!\right]^{\mathbf{b}}_{H,\rho} = H(f)([\![ck]\!]^{\mathbf{b}}_{H,\rho})\left([\![e_1]\!]^{\mathbf{b}}_{H,\rho}, \cdots, [\![e_n]\!]^{\mathbf{b}}_{H,\rho}\right)$$

Interprétation des équations :

$$[\![\texttt{(}x_1\texttt{, }\cdots\texttt{, }x_n\texttt{) = }e]\!]^{\mathbf{b}}_{H,\rho} = [x_i \mapsto ([\![e]\!]^{\mathbf{b}}_{H,\rho})^{(i)} \mid 1 \leq i \leq n]$$

$$[\![\texttt{rec } eq_1 \texttt{ and } \cdots \texttt{ and } eq_n]\!]^{\mathbf{b}}_{H,\rho} = \mathrm{fix}\,(\lambda\rho'.\, \uplus_{1\leq i\leq n} [\![eq_i]\!]^{\mathbf{b}}_{H,\rho\uplus\rho'})$$

Interprétation d'un programme :

$$[\![ds\texttt{;}e]\!]^{\mathcal{S}}_{\rho_0} = [\![e]\!]^{\mathbf{true}}_{H,\rho_0} \qquad \text{avec } H = [\![ds]\!]^{\mathcal{S}}_{\emptyset}$$

$$[\![d\texttt{;}ds]\!]^{\mathcal{S}}_{H} = H' \uplus [\![ds]\!]^{\mathcal{S}}_{H\uplus H'} \qquad \text{avec } H' = [\![d]\!]^{\mathcal{S}}_{H}$$

$$[\![f\texttt{(}x_1\texttt{, }\cdots\texttt{, }x_n\texttt{) = (}y_1\texttt{, }\cdots\texttt{, }y_m\texttt{) with } eqs]\!]^{\mathcal{S}}_{H} = [f \mapsto \lambda\mathbf{b}.\lambda(\mathbf{x_1}, \cdots, \mathbf{x_n}).(\rho(y_1), \cdots, \rho(y_m))]$$
$$\text{avec } \rho = [\![eqs]\!]^{\mathbf{b}}_{H,\rho_{in}} \quad \text{et} \quad \rho_{in} = [x_1 \mapsto \mathbf{x_1}, \cdots, x_n \mapsto \mathbf{x_n}]$$

FIGURE 2.3: Sémantique de Kahn synchrone

$$\begin{aligned} \mathrm{const}^{\mathcal{S}}_i(\mathrm{true}.\mathbf{c}) &= i.\,\mathrm{const}^{\mathcal{S}}_i(\mathbf{c}) \\ \mathrm{const}^{\mathcal{S}}_i(\mathrm{false}.\mathbf{c}) &= \mathit{abs}.\,\mathrm{const}^{\mathcal{S}}_i(\mathbf{c}) \end{aligned} \qquad (*)$$

Il reste donc à être capable d'interpréter une horloge.

2.2.3 Les horloges

Rappelons que les horloges ne sont pas des flots du programme mais des mots binaires, donc sans valeur *abs*. Nous avons présenté informellement en section 1.2.5 le langage utilisé par le compilateur pour décrire les horloges. Sa grammaire et sa sémantique sont données en figure 2.4.

Nous avions souligné le fait que les horloges, données par le compilateur, sont relatives à l'horloge de base locale $\bullet$. Celle-ci est inconnue tant que le nœud n'est pas instanciée. C'est pourquoi la sémantique que nous donnons est relative à une horloge de base $\mathbf{b}$, ajoutée en exposant de l'interprétation $[\![\ ]\!]^{\mathbf{b}}_{H,\rho}$. C'est cette base relative qui permet d'avoir une compilation et une sémantique modulaire des fonctions.

2.2.4 Programme et fonctions

En figure 2.3, nous donnons l'interprétation d'un programme. Il y a peu de différences par rapport à la version non synchrone. L'horloge d'activation $\mathbf{b}$ est le premier argument de

La grammaire :

$$
\begin{array}{rll}
\sigma ::= & \forall \bullet, \forall x, \cdots, x, (x :: ck, \cdots, x :: ck) \to (x :: ck, \cdots, x :: ck) & \text{signature d'horloge} \\
ck ::= & \mathbf{true} \quad \text{avec} \quad \mathbf{true} = \text{true}.\mathbf{true} & \text{horloge de base globale} \\
 & \mid \bullet & \text{horloge de base locale} \\
 & \mid ck \texttt{ on } c & \text{sous-horloge} \\
c ::= & \texttt{(}i \texttt{ = } p\texttt{)} & \text{échantillonneur} \\
p ::= & x & \text{pointeau}
\end{array}
$$

L'interprétation dans un contexte où l'horloge de base locale vaut $\mathbf{b}$:

$$
\begin{array}{ll}
[\![\bullet]\!]^{\mathbf{b}}_{H,\rho} = \mathbf{b} & [\![ck \texttt{ on (}i \texttt{ = } p\texttt{)}]\!]^{\mathbf{b}}_{H,\rho} = \text{on}^{\mathcal{S}}_i\left([\![ck]\!]^{\mathbf{b}}_{H,\rho}, \rho(p)\right) \\
\text{on}^{\mathcal{S}}_i(\text{true}.\mathbf{ck}, j.\mathbf{p}) = \text{false}.\,\text{on}^{\mathcal{S}}_i(\mathbf{ck}, \mathbf{p}) & \text{on}^{\mathcal{S}}_i(\text{false}.\mathbf{ck}, \mathit{abs}.\mathbf{p}) = \text{false}.\,\text{on}^{\mathcal{S}}_i(\mathbf{ck}, \mathbf{p}) \\
\text{on}^{\mathcal{S}}_i(\text{true}.\mathbf{ck}, i.\mathbf{p}) = \text{true}.\,\text{on}^{\mathcal{S}}_i(\mathbf{ck}, \mathbf{p}) &
\end{array}
$$

Figure 2.4: Le langage d'horloges d'Heptagon et son interprétation

l'interprétation d'une définition de fonction. Elle sert d'horloge de base aux équations de la fonction. À chaque instanciation, l'horloge d'activation doit être fournie. Tout comme pour les constantes, nous annotons les instances des fonctions avec l'horloge d'activation donnée par le compilateur. Un appel est alors $f^{ck}(x, \cdots, x)$.

Finalement, l'interprétation de l'expression principale du programme est faite avec l'horloge de base globale, c'est-à-dire le mot **true**.

2.2.5 Lien avec la sémantique non synchrone

La sémantique synchrone que nous donnons est un raffinement de la sémantique de Kahn que nous avons donnée précédemment. Le lien entre les deux sémantiques est donné par la fonction d'effacement des absences clean, de $(T + \{\mathit{abs}\})^\infty$ dans $(T)^\infty$:

$$\text{clean}(\varepsilon) = \varepsilon \qquad \text{clean}(\mathit{abs}.\mathbf{x}) = \text{clean}(\mathbf{x}) \qquad \text{clean}(v.\mathbf{x}) = v.\,\text{clean}(\mathbf{x})$$

Supprimer les absences du résultat de la sémantique synchrone revient à supprimer les absences des entrées et à les fournir à la sémantique non synchrone. En étendant clean aux environnements, cette propriété s'écrit :

$$\text{clean}([\![P]\!]^{\mathcal{S}}_{\rho_{in}}) = [\![P]\!]^{\mathcal{K}}_{\text{clean}(\rho_{in})}$$

La sémantique synchrone dépend des horloges données aux constantes et aux appels de nœuds, ce qui inclut le calcul des horloges dans la preuve.

2.3 Noyau équationnel et pointage

Le noyau fonctionnel accepte une expression comme argument de toute autre expression. Cependant, le langage d'horloges accepte uniquement des noms de flots comme pointeaux.

Pour l'expression `x when (0 = c)`, l'horloge sera clock (`x`) `on (0 = c)`, la variable `c` étant utilisée comme pointeau. Par contre, l'expression `x when (true fby c)` pose problème. Si le

Programme :

$$P ::= ds; eq \qquad ds ::= d; ds \qquad d ::= f\texttt{(}x\texttt{, }\cdots\texttt{, }x\texttt{) = (}x\texttt{, }\cdots\texttt{, }x\texttt{) with }eqs$$

Équations :

$$\begin{aligned} eqs &::= \texttt{rec}\ eq\ \texttt{and}\ \cdots\ \texttt{and}\ eq \\ eq &::= x\ \texttt{=}^{ck}\ ex \quad | \quad \texttt{(}x\texttt{, }\cdots\texttt{, }x\texttt{) =}^{ck}\ app \\ app &::= f\texttt{(}x\texttt{, }\cdots\texttt{, }x\texttt{)} \\ ex &::= i \quad | \quad op(x,x) \quad | \quad i\ \texttt{fby}\ x \quad | \quad x\ \texttt{when}\ c \quad | \quad \texttt{merge}\ p\ \texttt{(}i\ \texttt{->}\ x\texttt{)}\ \cdots\ \texttt{(}i\ \texttt{->}\ x\texttt{)} \end{aligned}$$

Figure 2.5: Noyau équationnel

programmeur veut une telle expression, il devra introduire une nouvelle variable `d` égale à `true fby c` et l'utiliser comme pointeau `x when (true = d)`. Le premier argument de `merge` doit lui aussi être un pointeau, car chaque branche du `merge` est sur une horloge échantillonnée avec lui. C'est aussi le cas pour les fonctions dont des entrées servent de pointeaux à d'autres entrées/sorties.

Les sorties aussi peuvent être des pointeaux, ce qui nécessite de les nommer. Par exemple le nœud `positif` (cf. section 1.2.6.1) est de signature $\forall\bullet, \forall c, \bullet \rightarrow (c :: \bullet, \bullet\ \texttt{on}\ c)$. À l'instanciation, `(d, y) = positif(x)`, l'horloge de `y` est $\mathrm{clock}\ (\texttt{x})\ \texttt{on}\ \texttt{(true = d)}$. Pour être définie, l'horloge de la deuxième sortie requiert de connaître le nom de la variable qui stocke la première sortie.

Nous pourrions présenter le calcul des horloges sur un noyau qui requiert des variables uniquement là où cela est nécessaire. Mais nous allons profiter du changement de noyau pour en présenter un très proche de celui utilisé dans le compilateur. Outre la possibilité de calculer les horloges, ce noyau nous simplifiera la génération de code.

2.3.1 Noyau équationnel

Nous passons à un noyau que nous qualifions d'équationnel, présenté en figure 2.5. La structure d'un programme n'a pas changée. Elle commence par des définitions de nœuds et finit avec une équation principale (à la place d'une expression). Cette dernière joue le même rôle, les variables libres sont les entrées du programme, les variables définies sont les sorties. Par contre, le membre droit d'une équation n'est plus une expression arbitrairement profonde. Cette forme normale est proche du format trois adresses SSA. Tous les arguments des pseudo expressions ex sont des variables.

Nous en avons profité pour simplifier le registre synchrone. Il prend maintenant en premier argument une valeur instantanée. Le langage d'horloge utilisé est le même qu'en figure 2.4. La primitive `when` prend directement un échantillonneur en deuxième argument et `merge` un pointeau en premier argument. De plus, au lieu d'annoter les constantes et les appels de nœuds avec leurs horloges, nous annotons les équations.

2.3.1.1 Traduction du `fby` avec initialisation statique

Le registre synchrone dans sa forme la plus simple n'est pas initialisé dynamiquement avec la première valeur d'un flot mais par une constante. La traduction utilise l'équivalence :

```
e1 fby e2     ≡     if (true fby false) then e1 else (42 fby e2)
```

La valeur d'initialisation 42 est prise au hasard, puisqu'inutile. Par contre, elle doit être de même type de données que `e1` et `e2`, ici considérées `int`.

2.3.1.2 Passage d'un noyau à l'autre

Le gros de la traduction est l'introduction de variables pour chaque sous-expression. Les horloges des équations sont les horloges d'activation de l'équation. Elles correspondent exactement à l'horloge de l'expression de leur membre droit, en particulier, si nous avions une constante i^{ck} alors l'équation introduite pour cette constante va porter la même horloge ($x = ^{ck} i$). Similairement, si nous avions un appel de fonction $f^{ck}(x, \cdots, x)$, alors l'équation introduite porte aussi l'horloge ck.

2.3.2 Pointage (*clocking* ou *calcul d'horloges*)

Le pointage est un typage qui associe à chaque flot une horloge. Il permet de rejeter les programmes qui utilisent incorrectement les primitives, par exemple `x+y` si `x` et `y` ne peuvent pas être de même horloge. Ce typage est traditionnellement appelé *calcul d'horloge* [Ber86 ; Ham02 ; Pou02], cependant, en anglais, il est plus aisément appelé *clocking*, permettant de parler de *clocked stream* et de l'utiliser comme un verbe désignant le fait de typer. Pour cette raison, nous préférons utiliser le terme *pointage*, permettant de parler de *flot pointé* et d'utiliser le verbe *pointer* comme on utiliserait le verbe *typer*. Ce changement de terminologie est cohérent avec la notion de pointeau qu'il nous semble important de séparer de la notion d'horloge (cf. note 1.2.4). Finalement, nous verrons en chapitre 8 une approche générique du pointage, indépendamment de la manière dont sont calculées les horloges.

Les règles de pointage sont rassemblées en figure 2.6, elles suivent celles de [CPP05]. Les définitions du programme sont rassemblées dans l'environnement Γ qui associe à chaque fonction sa signature. L'équation principale est pointée avec ces définitions et un environnement γ_0 donnant les horloges des entrées et sorties du programme.

La règle (Def) de définition d'une fonction f ajoute sa signature à l'environnement. La signature est construite en généralisant les horloges des entrées et des sorties obtenues en pointant les équations de la fonction. La généralisation consiste à quantifier universellement l'horloge de base de la fonction, ainsi que les pointeaux. L'environnement γ est local aux équations d'un nœud, il est omniscient et associe à chaque variable son horloge.

L'horloge de base d'une équation est soit l'horloge de son membre droit pour une équation simple (Eq), soit l'horloge utilisée comme horloge de base pour l'instanciation de la fonction (App). L'instanciation d'une fonction effectue en plus la substitution des noms de pointeaux dans la signature pour obtenir les horloges des variables de l'équation. Le pointage des pseudo expressions suit ce que nous avons évoqué précédemment.

2.3.2.1 Structure des horloges et signatures

Les horloges et signatures inférées par le système sont contraintes d'être bien formées (cf. figure 2.6). Pour les horloges, les pointeaux utilisés doivent être sur l'horloge qu'ils servent à échantillonner. Pour les signatures, les pointeaux utilisés pour les horloges des entrées doivent être des entrées, tandis que les horloges des sorties peuvent utiliser tout pointeau de la signature. La propriété de bonne formation des horloges implique que les pointeaux forment un arbre d'horloge dont la racine est $\bullet$. En particulier, il n'y a pas de dépendances circulaires entre les pointeaux. Ceci associé à la bonne formation des signatures implique qu'il y a au moins une des entrées qui est sur l'horloge de base.

Manipulation d'une signature :
Si $\sigma = \forall \bullet, \forall x_1, \cdots, x_n, y_1, \cdots, y_m, (x_1 :: ck_1, \cdots, x_n :: ck_n) \to (y_1 :: ck'_1, \cdots, y_m :: ck'_m)$, alors

$$\mathrm{inst}(\sigma)(ck)(x'_1, \cdots, x'_n)(y'_1, \cdots, y'_m) = \Big((ck_1, \cdots, ck_n) \to \Big((ck'_1, \cdots, ck'_m)[y'_1/y_1] \cdots [y'_m/y_m]\Big)\Big)[ck/\bullet][x'_1/x_1] \cdots [x'_n/x_n]$$

$$\mathrm{gen}\Big((x_1 :: ck_1, \cdots, x_n :: ck_n) \to (y_1 :: ck'_1, \cdots, y_m :: ck'_m)\Big) = \sigma$$

Programme et définitions :

$$(\textsc{Prog})\ \frac{\vdash ds : \Gamma \qquad \Gamma, \gamma_0 \vdash eq}{\vdash ds\,\texttt{;}\,eq} \qquad\qquad (\textsc{SeqDef})\ \frac{\Gamma \vdash d : \Gamma' \qquad \Gamma' \vdash ds : \Gamma''}{\Gamma \vdash d\,\texttt{;}\,ds : \Gamma''}$$

$$(\textsc{Def})\ \frac{\begin{array}{c} s = (x_1 :: \gamma(x_1), \cdots, x_n :: \gamma(x_n)) \to (y_1 :: \gamma(y_1), \cdots, y_m :: \gamma(y_m)) \\ \overset{\text{SIG}}{\vdash} s \qquad \Gamma, \gamma \vdash eqs \qquad \Gamma' = \Gamma \uplus \{f \mapsto \mathrm{gen}(s)\} \end{array}}{\Gamma \vdash f\texttt{(}x_1\texttt{, }\cdots\texttt{, }x_n\texttt{) = (}y_1\texttt{, }\cdots\texttt{, }y_m\texttt{) with }eqs : \Gamma'}$$

Équations :

$$(\textsc{Eqs})\ \frac{\Gamma, \gamma \vdash eq_1 \qquad \cdots \qquad \Gamma, \gamma \vdash eq_n}{\Gamma, \gamma \vdash \texttt{rec}\ eq_1\ \texttt{and}\ \cdots\ \texttt{and}\ eq_n} \qquad\qquad (\textsc{Eq})\ \frac{\gamma \vdash ex :: ck \qquad \gamma(y) = ck}{\Gamma, \gamma \vdash y\ \texttt{=}^{ck}\ ex}$$

$$(\textsc{App})\ \frac{\begin{array}{c} \mathrm{inst}(\Gamma(f))(ck)(x_1, \cdots, x_n)(y_1, \cdots, y_m) = (ck_1, \cdots, ck_n) \to (ck'_1, \cdots, ck'_m) \\ \forall j \in [1; m], \gamma(y_j) = ck'_j \qquad \forall i \in [1; n], \gamma(x_i) = ck_i \end{array}}{\Gamma, \gamma \vdash \texttt{(}y_1\texttt{, }\cdots\texttt{, }y_m\texttt{) =}^{ck}\ f^{ck}\texttt{(}x_1\texttt{, }\cdots\texttt{, }x_n\texttt{)}}$$

Pseudo expressions :

$$(\textsc{Inst})\ \gamma \vdash i^{ck} :: ck \qquad (\textsc{Op})\ \frac{\gamma(x_1) = \gamma(x_2) = ck}{\gamma \vdash op(x_1, x_2) :: ck} \qquad (\textsc{When})\ \frac{\gamma(x) = \gamma(p) = ck}{\gamma \vdash x\ \texttt{when (}i\texttt{ = }p\texttt{)} :: ck\ \texttt{on (}i\texttt{ = }p\texttt{)}}$$

$$(\textsc{Fby})\ \frac{\gamma(x) = ck}{\gamma \vdash i^{ck}\ \texttt{fby}\ x :: ck} \qquad (\textsc{Merge})\ \frac{\gamma(p) = ck \qquad \forall i \in [1; n], \gamma(x_i) = ck\ \texttt{on (}i\texttt{ = }p\texttt{)}}{\gamma \vdash \texttt{merge}\ p\ \texttt{(0 -> }x_0\texttt{)} \cdots \texttt{(}n\texttt{ -> }x_n\texttt{)} :: ck}$$

Bonne formation des horloges :

$$(\textsc{Base})\ \gamma \overset{\text{CK}}{\vdash} \bullet \qquad (\textsc{On})\ \frac{\gamma(p) = ck \qquad \gamma \overset{\text{CK}}{\vdash} ck}{\gamma \overset{\text{CK}}{\vdash} ck\ \texttt{on (}i\texttt{ = }p\texttt{)}} \qquad \begin{array}{c} FV(\bullet) = \emptyset \\ FV(ck\ \texttt{on (}i\texttt{ = }p\texttt{)}) = \{p\} \cup FV(ck) \end{array}$$

$$(\textsc{Sig})\ \frac{\begin{array}{c} \forall i \in [1; n], \forall j \in [1; m], \\ FV(ck_i) \subseteq \{x_1, \cdots, x_n\} \quad FV(ck'_i) \subseteq \{x_1, \cdots, x_n, y_1, \cdots, y_m\} \quad \gamma \overset{\text{CK}}{\vdash} ck_i \quad \gamma \overset{\text{CK}}{\vdash} ck_j \end{array}}{\overset{\text{SIG}}{\vdash} (x_1 :: ck_1, \cdots, x_n :: ck_n) \to (y_1 :: ck'_1, \cdots, y_m :: ck'_m)}$$

Figure 2.6: Règles du pointage

Par construction, l'horloge de base de la signature est racine de *toutes* les horloges de flots internes à la fonction en question. Ainsi, à l'instanciation, nous savons que toutes les équations internes de la fonction ont pour racine l'horloge de l'instanciation.

2.3.2.2 Libertés dans le choix des horloges

Malgré toutes les contraintes du pointage, il est possible d'avoir des libertés dans le choix des horloges. C'est le cas du nœud suivant :

`f(c : int :: •) = (y : int ::` ck`) with rec y =`^{ck} `3`

L'horloge ck peut être • ou bien • `on (`i `= c)`. Dans ce cas, le compilateur Heptagon choisit par défaut l'horloge la plus rapide (ici •), sauf si des informations provenant du frontend existent. Ces informations peuvent être des annotations de l'utilisateur ou bien la position de l'équation dans une structure de contrôle comme un automate, avant la traduction de ceux-ci vers le noyau.

2.3.3 Cohérence du pointage

Le rôle des horloges est de définir les instants de présence des flots. Ainsi, les horloges inférées par le pointage doivent être cohérente avec la sémantique synchrone.

2.3.1 *Définition* **Cohérence d'un flot avec son horloge**
Un flot $\mathbf{x}$ synchrone est d'horloge **ck** ssi pour tout k inférieur à la longueur de $\mathbf{x}$,

$$\mathbf{x}[k] = abs \iff \mathbf{ck}[k] = \text{false}$$

2.3.2 *Propriété* **Cohérence d'un programme**
Si les entrées du programme sont cohérentes avec leurs horloges alors tous les flots définis par le programme le sont aussi.

Démonstration La preuve se fait par induction sur la structure du programme, en considérant cohérent l'environnement γ fourni.

- La mise en parallèle d'équations effectue le point fixe de l'interprétation des équations. À la toute première itération, l'environnement est celui fourni plus l'environnement vide. La cohérence est donc assurée car soit les flots sont dans l'environnement fourni, soit ils sont de taille nulle. La préservation de la propriété à chaque itération est prouvée en considérant les équations.
- Pour les équations `y =`^{ck} ex, nous avons toujours $\gamma(\mathbf{y}) = ck$. Nous devons alors prouver que $(\llbracket ex \rrbracket^{\mathbf{b}}_{H,\rho})[k] = abs$ ssi $(\llbracket ck \rrbracket^{\mathbf{b}}_{H,\rho})[k] = \text{false}$.
  - Si l'expression est une constante i^{ck}, le calcul de $\text{const}^{\mathcal{S}}_i\left(\llbracket ck \rrbracket^{\mathbf{b}}_{H,\rho}\right)$ (cf. section 2.2.2) assure la cohérence.
  - Les arguments des opérateurs sont de même horloge que le résultat. Le calcul $op^{\mathcal{S}}(\mathbf{x}, \mathbf{x}')$ consomme ses entrées au même rythme qu'il produit sa sortie. De plus, une valeur abs n'est produite que si les entrées le valent aussi (le filtrage marqué avec (*), cf. section 2.2).
  - Pour le registre, les choses sont comme pour les opérateurs. Il faut ajouter que la sortie est produite au même rythme que sont consommées les entrées, sauf une fois le deuxième argument égal à ε. Mais dans ce cas là, la valeur produite en sortie n'est pas abs (cf. section 2.2.1).

— L'horloge de la sortie est fausse, ssi la sortie vaut *abs*, grâce aux filtrages de $\text{when}^{\mathcal{S}}\left(\llbracket \mathtt{x} \rrbracket^{\mathbf{b}}_{H,\rho}, \llbracket i^{ck} = \mathtt{p} \rrbracket^{\mathbf{b}}_{H,\rho}\right)$ (cf. section 2.2) quand `p` vaut *abs* ou j avec $j \neq i$. Cela est cohérent avec le calcul $\text{on}^{\mathcal{S}}_{i}\left(\llbracket \gamma(\mathtt{p}) \rrbracket^{\mathbf{b}}_{H,\rho}, \llbracket \mathtt{p} \rrbracket^{\mathbf{b}}_{H,\rho}\right)$ (cf. figure 2.4) de l'horloge $\gamma(\mathtt{p})$ `on (i = p)` donnée au résultat.
— La complémentation consomme son pointeau au même rythme qu'elle produit. De plus, son résultat vaut *abs* ssi le pointeau est aussi absent.
— Pour l'équation d'appel de fonction, la cohérence vient directement de celle de l'évaluation des équations internes à la fonction.

Finalement, la cohérence du programme se résume à celle de son équation principale.

2.4 Sémantique de Kahn synchrone implicite (sans réaction à *abs*)

Nous présentons ici une sémantique originale qui se rapproche le plus possible du code impératif que l'on va générer. Son but est de rendre compte de la structure de contrôle qu'induisent les horloges et du gain primordial que cette structure apporte : les valeurs des flots n'ont pas besoin d'être dans un domaine étendu avec *abs*, car il n'y aura pas de calcul réagissant à cette valeur. Cette possibilité est d'une certaine manière évidente puisque notre langage a une sémantique de Kahn (nous verrons en partie III et plus particulièrement en section 7.4.2 que cela n'est pas si évident). Mais la sémantique synchrone a perdu cette possibilité. Par exemple l'addition `x + y` doit réagir quand ses deux entrées valent *abs* en renvoyant *abs*.

L'idée tient aux deux propriétés, de la sémantique synchrone, suivantes :

2.4.1 *Propriété* **Réaction à l'absence par l'absence sans changement d'état**
Dans la sémantique de Kahn synchrone, un calcul réagit à l'absence de toutes ses entrées, en déclarant l'absence de ses sorties et en gardant intact son état.

Démonstration Premièrement, les filtrages des primitives, marqués avec (*) dans la sémantique synchrone en section 2.2, correspondent au cas où toutes les entrées sont absentes. Dans tous ces cas, les sorties le sont aussi.

Deuxièmement, ces primitives n'ont pas d'état sauf le registre synchrone et il conserve sa mémoire dans le cas marqué par (*).

L'appel de fonction est le cas délicat. La section 2.3.2.1 nous donne le fait qu'il y a au moins une entrée sur l'horloge $\bullet$. Si toutes les entrées sont absentes, alors l'horloge de base est fausse. Si elle est fausse, comme les horloges sont en arbre avec $\bullet$ en racine, tous les flots du nœud sont absents (propriété 2.3.2). Par induction, tous les registres seront gardés intacts. Finalement, les sorties sont absentes puisque leurs horloges ont pour racine la base absente.

Intuitivement, si toutes les entrées sont absentes, comme l'état ne change pas et les sorties sont connues d'avance, il n'est pas utile d'effectuer le calcul. Ceci relie l'idée d'horloge d'activation d'une équation.

2.4.2 *Propriété* **Correction de l'horloge d'activation**
Aux instants où l'horloge d'activation d'une équation est fausse, il n'est pas utile d'effectuer son calcul, car nous savons que les flots définis valent *abs* et l'état du membre droit ne change pas.

Démonstration Quand l'horloge est fausse, toutes les entrées du membre droit sont absentes, exceptées pour `when` que nous traitons après. Si toutes les entrées sont absentes, la propriété 2.4.1 conclue que l'activation n'est pas utile. Pour `when`, le flot échantillonné n'est pas absent, mais comme nous sommes en forme trois adresses, l'équation n'a pas d'état. Exécuter ou pas le filtrage ne change rien, nous savons que la sortie sera absente.

Nous allons exhiber dans la sémantique l'utilisation de l'horloge d'activation ck de chaque équation. L'activation aura lieu à l'instant k ssi l'horloge $\mathbf{ck}$ associée est telle que $\mathbf{ck}[k] = \text{true}$. Dans notre contexte dénotationnel synchrone, ne pas activer une équation/expression à un instant k revient à supprimer la $k$ième valeur des flots du contexte. Nous allons donc filtrer l'environnement ρ dans lequel une équation est évaluée. Ce filtrage est effectué avec la fonction continue $\text{down}^{\mathcal{C}}(\mathbf{ck}, \rho)$. L'environnement résultant a uniquement les valeurs des instants où l'horloge $\mathbf{ck}$ est vraie. Le filtrage d'un environnement est le filtrage de chacune de ses variables. Si l'horloge d'activation est vraie, on laisse la valeur, que ce soit *abs* ou pas, sinon on enlève la valeur :

$$\text{down}^{\mathcal{C}}(\mathbf{ck}, \rho) = [\forall x, x \mapsto \text{down}^{\mathcal{C}}(\mathbf{ck}, \rho(x))] \qquad \begin{aligned} \text{down}^{\mathcal{C}}(\text{true}.\mathbf{ck}, w.\mathbf{x}) &= w.\, \text{down}^{\mathcal{C}}(\mathbf{ck}, \mathbf{x}) \\ \text{down}^{\mathcal{C}}(\text{false}.\mathbf{ck}, w.\mathbf{x}) &= \text{down}^{\mathcal{C}}(\mathbf{ck}, \mathbf{x}) \end{aligned}$$

Dans ce contexte, l'équation ne reçoit pas de valeur pour les instants où l'horloge est fausse, elle n'en produit donc pas non plus. Pour avoir les flots de sortie dans leur totalité, il faut insérer les valeurs qu'elle aurait produite si son environnement n'était pas filtré, ce que nous faisons avec la fonction continue $\text{up}^{\mathcal{C}}(\mathbf{ck}, \rho)$. Symétriquement à $\text{down}^{\mathcal{C}}(\mathbf{ck}, \rho)$, cette fonction laisse la valeur si l'horloge d'activation est vraie et rajoute *abs* sinon :

$$\text{up}^{\mathcal{C}}(\mathbf{ck}, \rho) = [\forall x, x \mapsto \text{up}^{\mathcal{C}}(\mathbf{ck}, \rho(x))] \qquad \begin{aligned} \text{up}^{\mathcal{C}}(\text{true}.\mathbf{ck}, w.\mathbf{x}) &= w.\, \text{up}^{\mathcal{C}}(\mathbf{ck}, \mathbf{x}) \\ \text{up}^{\mathcal{C}}(\text{false}.\mathbf{ck}, \mathbf{x}) &= \mathit{abs}.\, \text{up}^{\mathcal{C}}(\mathbf{ck}, \mathbf{x}) \end{aligned}$$

2.4.3 *Propriété* **Innocuité de la manipulation des environnements**
Si un flot $\mathbf{x}$ est sur une sous-horloge de l'horloge d'activation $\mathbf{ck}$, alors :

$$\text{up}^{\mathcal{C}}\Big(\mathbf{ck}, \text{down}^{\mathcal{C}}(\mathbf{ck}, \mathbf{x})\Big) = \mathbf{x}$$

Démonstration Posons $\mathbf{ckx}$ l'horloge de $\mathbf{x}$. Puisque $\mathbf{ckx}$ est une sous-horloge de $\mathbf{ck}$, nous avons pour tout k, $(\mathbf{ck}[k] = \text{false}) \implies (\mathbf{ckx}[k] = \text{false})$. La cohérence des horloges (cf. propriété 2.3.2) donne pour k inférieur à la taille du flot $\mathbf{x}$, $(\mathbf{ckx}[k] = \text{false}) \implies (\mathbf{x}[k] = \mathit{abs})$. Ainsi, les valeurs supprimées par $\text{down}^{\mathcal{C}}(\mathbf{ck}, \mathbf{x})$ sont toutes des *abs* et elles sont remises au même endroit par $\text{up}^{\mathcal{C}}\Big(\mathbf{ck}, \text{down}^{\mathcal{C}}(\mathbf{ck}, \mathbf{x})\Big)$.

La sémantique est écrite en figure 2.7. Nous n'activons une équation que quand sont horloge est vraie. Ainsi, nous évaluons son membre droit comme s'il était sur l'horloge de base. Grâce à cela, l'interprétation des expressions est grandement simplifiée. Les constantes n'ont plus besoin d'être intelligentes, le registre synchrone non plus, `merge` ne lit que les entrées nécessaires (les autres sont des w qui peuvent être *abs* ou pas). Le plus surprenant est certainement `when`, qui n'a besoin de réagir que quand le pointeau vaut i, c'est-à-dire la valeur ne filtrant pas l'entrée. En effet, si le pointeau vaut une valeur différente de i alors l'horloge de l'équation du `when` est fausse et l'expression n'est pas évaluée. Ainsi, l'échantillonnage est effectué par le filtrage de l'équation.

Programmes et définitions :

$$\begin{aligned}
[\![ds; eq]\!]^{\mathcal{C}}_{\rho_{in}} &= [\![eq]\!]^{\mathcal{C}}_{H,\rho} \text{ avec } H = [\![ds]\!]^{\mathcal{C}}_{\emptyset} \\
[\![d; ds]\!]^{\mathcal{C}}_{H} &= H' \uplus [\![ds]\!]^{\mathcal{C}}_{H \uplus H'} \text{ avec } H' = [\![d]\!]^{\mathcal{C}}_{H,\rho} \\
[\![f\,\texttt{(}x_1\texttt{, }\cdots\texttt{, }x_n\texttt{) = (}y_1\texttt{, }\cdots\texttt{, }y_m\texttt{) with }eqs]\!]^{\mathcal{C}}_{H} &= [f \mapsto \lambda(\mathbf{x_1}, \cdots, \mathbf{x_n}).(\rho(y_1), \cdots, \rho(y_m))] \\
&\quad \text{avec } \rho = [\![eqs]\!]^{\mathcal{C}}_{H,\rho_{in}} \text{ et } \rho_{in} = [x_1 \mapsto \mathbf{x_1}, \cdots, x_n \mapsto \mathbf{x_n}]
\end{aligned}$$

Équations en posant $\mathbf{ck} = [\![ck]\!]^{\mathbf{true}}_{H,\rho}$:

$$\begin{aligned}
[\![\texttt{rec }eq_1\texttt{ and }\cdots\texttt{ and }eq_n]\!]^{\mathcal{C}}_{H,\rho} &= \text{fix}\,(\lambda\rho'.\, \uplus_{1\le i\le n} [\![eq_i]\!]^{\mathcal{C}}_{H,\rho\uplus\rho'}) \\
\left[\!\!\left[y \texttt{ =}^{ck}\texttt{ } ex\right]\!\!\right]^{\mathcal{C}}_{H,\rho} &= \text{up}^{\mathcal{C}}\Big(\mathbf{ck}, \left[y \mapsto [\![ex]\!]^{\mathcal{C}}_{H,\text{down}^{\mathcal{C}}(\mathbf{ck},\rho)}\right]\Big) \\
\left[\!\!\left[\texttt{(}y_1\texttt{, }\cdots\texttt{, }y_n\texttt{) =}^{ck}\texttt{ } app\right]\!\!\right]^{\mathcal{C}}_{H,\rho} &= \text{up}^{\mathcal{C}}(\mathbf{ck}, \rho_{eq}) \\
\text{avec } \rho' = \text{down}^{\mathcal{C}}(\mathbf{ck}, \rho) \qquad \rho_{eq} &= [y_i \mapsto ([\![app]\!]^{\mathcal{C}}_{H,\rho'})^{(i)} \mid 1 \le i \le n]
\end{aligned}$$

Interprétation des expressions :

$$[\![i]\!]^{\mathcal{C}}_{H,\rho} = i^{\omega} \qquad [\![x]\!]^{\mathcal{C}}_{H,\rho} = \rho(x) \qquad [\![i\ \texttt{fby}\ x]\!]^{\mathcal{C}}_{H,\rho} = i.\rho(x) \qquad [\![op(x_1, x_2)]\!]^{\mathcal{C}}_{H,\rho} = op^{\mathcal{C}}(\rho(x_1), \rho(x_2))$$

$$[\![x_1\ \texttt{when (}i\texttt{ = }p\texttt{)}]\!]^{\mathcal{C}}_{H,\rho} = \text{when}^{\mathcal{C}}_i(\rho(x_1), \rho(p)) \qquad [\![f\texttt{(}x_1\texttt{, }\cdots\texttt{, }x_n\texttt{)}]\!]^{\mathcal{C}}_{H,\rho} = H(f)(\rho(x_1), \cdots, \rho(x_n))$$

$$[\![\texttt{merge }p\texttt{ (0 -> }x_0\texttt{)}\cdots\texttt{(}n\texttt{ -> }x_n\texttt{)}]\!]^{\mathcal{C}}_{H,\rho} = \text{merge}^{\mathcal{C}}(\rho(p), \rho(x_0), \cdots, \rho(x_n))$$

$$\begin{aligned}
op^{\mathcal{C}}(v_1.\mathbf{v_1}, v_2.\mathbf{v_2}) &= v.\, op^{\mathcal{C}}(\mathbf{v_1}, \mathbf{v_2}) \text{ avec } v = op(v_1, v_2) \\
\text{when}^{\mathcal{C}}_i(v.\mathbf{v}, i.\mathbf{p}) &= v.\, \text{when}^{\mathcal{C}}_i(\mathbf{v}, \mathbf{p}) \\
\text{merge}^{\mathcal{C}}(k.\mathbf{p}, w_0.\mathbf{v_0}, \cdots, v.\mathbf{v}_k, \cdots, w_n.\mathbf{v_n}) &= v.\, \text{merge}^{\mathcal{C}}(\mathbf{p}, \mathbf{v_0}, \cdots, \mathbf{v}_k, \cdots, \mathbf{v_n})
\end{aligned}$$

Figure 2.7: Sémantique de Kahn synchrone sans réaction à l'absence

L'interprétation des définitions est inchangée, sauf que l'horloge de base n'est plus une entrée supplémentaire. En effet, à l'appel, l'environnement est échantillonné pour que la fonction se retrouve comme sur l'horloge de base globale.

2.4.4 Théorème Égalité des sémantiques synchrones

$$[\![P]\!]^{\mathcal{C}}_{\rho_{in}} = [\![P]\!]^{\mathcal{S}}_{\rho_{in}}$$

Démonstration Nous donnons uniquement un aperçu de la preuve. Si H est l'environnement des définitions de la sémantique synchrone, nous posons $H' = [f \mapsto H(f)(\mathbf{true})]_{f\in\text{dom}(H)}$, qui servira d'environnement pour la sémantique synchrone implicite. Une propriété intermédiaire à prouver est $[\![eq]\!]^{\mathcal{C}}_{H',\rho} = [\![eq]\!]^{\mathbf{true}}_{H,\rho}$. Celle-ci permet de prouver l'équivalence de comportement de l'équation principale mais aussi, que tous les flots d'un nœud sont calculés par la sémantique implicite, comme si le nœud était appelé sur l'horloge de base globale. Prouvons cette propriété.

En posant $\mathbf{ck} = [\![ck]\!]^{\mathbf{b}}_{H,\rho}$ pour tout ck et $\mathbf{b}$, nous devons prouver la propriété intermédiaire :

$$\mathrm{up}^{\mathcal{C}}\Big(\mathbf{ck}, [\![ex]\!]^{\mathcal{C}}_{H',\mathrm{down}^{\mathcal{C}}(\mathbf{ck},\rho)}\Big) = \mathrm{up}^{\mathcal{C}}\Big(\mathbf{ck}, [\![ex[\bullet/ck]]\!]^{\mathbf{true}}_{H,\mathrm{down}^{\mathcal{C}}(\mathbf{ck},\rho)}\Big) = [\![ex[\bullet/ck]]\!]^{\mathbf{ck}}_{H,\rho} = [\![ex]\!]^{\mathbf{b}}_{H,\rho}$$

La première égalité est immédiate, le seul cas où la substitution importe est la constante pour lequel nous avons bien $[\![i^{ck}]\!]^{\mathcal{C}}_{H',\rho} = [\![i^{\bullet}]\!]^{\mathbf{true}}_{H,\rho}$. La deuxième égalité est conséquence de l'innocuité de la manipulation d'une équation (cf. propriétés 2.4.2 et 2.4.3. La troisième est un simple jeu, changeant l'horloge de base de l'interprétation au lieu de l'horloge de l'expression.

Nous avons la même suite d'égalités en changeant ex par app. Le point délicat est alors la première égalité, $[\![app]\!]^{\mathcal{C}}_{H',\rho} = [\![app]\!]^{\mathbf{true}}_{H,\rho}$ résolu par la définition de H'.

Finalement, nous avons ce que nous cherchions :

2.4.5 *Propriété* **Valeur d'absence** *abs* **implicite**
Aucune des fonctions que nous utilisons dans cette sémantique ne demande de reconnaître *abs* pour réagir. *abs* peut donc être remplacée par n'importe quelle valeur (dépendant du contexte, pour que le tout soit bien typé), sans changer la valeur des flots quand ils sont présents.

2.5 Compilation vers du code séquentiel

Dans cette section, nous parlons brièvement de l'ordonnancement des équations du noyau, étape préliminaire à la compilation vers du code impératif séquentiel. Ensuite nous présentons le langage impératif cible dénommé Obc, qui sert dans Heptagon de langage séquentiel intermédiaire simple avant de compiler vers du Java ou du C. Finalement, nous présentons la compilation du noyau vers Obc. Cette compilation suit de près la compilation dans Heptagon et étend son article fondateur [Bie+08] pour traiter des nœuds avec des entrées et des sorties d'horloges différentes.

Comme nous l'avons esquissé en chapitre 1, le code généré d'un nœud est la combinaison d'un état et d'une fonction de transition. Suivant la sémantique synchrone, la fonction de transition consomme ou produit une valeur pour chaque flot, à chaque activation. Toute les constructions, excepté `fby`, se traduisent en un simple pas scalaire. Ainsi une équation devient une affectation scalaire. L'équation du registre synchrone devient la lecture de son état. La mise à jour de l'état est une deuxième affectation que nous rajouterons en fin d'instant.

2.5.1 Ordonnancement

Nous devons ordonner les équations de telle sorte que la traduction en affectations sera correcte. Il nous faut assurer qu'une valeur est écrite avant d'être lue. Pour cela, nous calculons les dépendances d'une équation eq avec $\mathrm{uses}(eq)$. Notez que l'équation du registre synchrone

ne dépend de rien, car elle correspond à la lecture de la mémoire :

$$\begin{aligned}
\text{defs}\Big(y \texttt{ =}^{ck}\ ex\Big) &= \{y\} & \text{defs}\Big(\texttt{(}y_1\texttt{, }\cdots\texttt{, }y_m\texttt{) =}^{ck}\ _\Big) &= \{y_1, \cdots, y_m\}\\
\text{uses}(\bullet) &= \emptyset & \text{uses}(ck\ \texttt{on (}i\texttt{ = }p\texttt{)}) &= \{p\} \cup \text{uses}(ck)\\
\text{uses}(x) &= \{x\} & \text{uses}\Big(y \texttt{ =}^{ck}\ ex\Big) &= \text{uses}(ck) \cup \text{uses}(ex)\\
\text{uses}(i\ \texttt{fby}\ x) &= \emptyset & \text{uses}(op(x_1, x_2)) &= \{x_1, x_2\}\\
\text{uses}(x\ \texttt{when}\ c) &= \{x,c\} & \text{uses}(\texttt{merge}\ p\ \texttt{(0 -> }x_0\texttt{)}\cdots\texttt{(}n\texttt{ -> }x_n\texttt{)}) &= \{p, x_1, \cdots, x_n\} \cup \text{uses}(ck)\\
\text{uses}(i) &= \emptyset & \text{uses}\Big(\texttt{(}y_1\texttt{, }\cdots\texttt{, }y_m\texttt{) =}^{ck}\ f\texttt{(}x_1\texttt{, }\cdots\texttt{, }x_n\texttt{)}\Big) &= \{x_1, \cdots, x_n\} \cup \text{uses}(ck)
\end{aligned}$$

Pour trouver un ordonnancement, Nous construisons le graphe de dépendances, dont les nœuds sont les équations. Pour toute équation eq, nous ajoutons une dépendance vers toutes les équations eq', dont les variables définies sont utiles à eq, c'est-à-dire qu'un lien est ajouté ssi $\text{defs}(eq') \cap \text{uses}(eq) \neq \emptyset$. Ensuite, tout tri topologique du graphe nous donne un ordonnancement convenable des équations. Si le tri topologique échoue, le réseau est dit non *causal*.

Il est courant que le graphe ainsi défini puisse être linéarisé de plusieurs manières. Le choix effectué par le compilateur peut alors être influencé par des heuristiques pour aider l'optimisation du code généré. Nous le reverrons pour l'optimisation du contrôle et de la mémoire.

2.5.1 *Remarque* **Causalité non modulaire**
La notion de dépendance que nous utilisons est syntaxique, similairement à ce qui a été proposé pour Lustre dans [HRR91]. En particulier, toutes les entrées d'un nœud sont considérées nécessaires pour toutes ses sorties. Ainsi, si un nœud renferme un registre entre une entrée et une sortie, son utilisation peut ne pas être causale, alors que si on l'inline le résultat le sera. Pour remédier à cela, Heptagon permet de forcer l'inlining d'une fonction (cf. section 1.4). Nous discutons de ces problématiques en section 10.1.

2.5.2 *Nota bene* **Résultat de l'ordonnancement**
L'ordonnancement numérote chaque équation avec un indice z, donnant par exemple $y \texttt{ =}^{ck}_{z}\ ex$.

2.5.2 Obc un langage impératif intermédiaire

Le language Obc, présenté en 2.8, est un langage impératif, avec une notion de machines et de processus. C'est une version étendue de la version présentée dans [Bie+08]. Un programme est un ensemble de définitions de machines accompagnées d'un processus principal. Une machine (`machine`) est l'équivalent d'une classe, spécialisée pour représenter un nœud. Elle comporte des mémoires (`memory`) pour les registres synchrones, des instances de nœuds pour chaque appel de nœud, une méthode d'initialisation `reset` et une méthode pour effectuer une transition `step`. Les méthodes sont composées de variables locales et d'instructions. Les instructions sont impératives et organisées sous la forme d'une séquence. Il est à noter que l'affectation permet de définir plusieurs variables d'un coup. Les variables v sont soit des variables déclarées localement, soit des variables d'instances (des mémoires) `this.`x.

2.5.3 Traduction du noyau

La sémantique synchrone implicite (cf section 2.4) nous permet trois choix de compilations interdépendants :

$P ::=$	$d; P$	définitions de machines
	$\mid p;$	processus principal
$p ::=$	`processus {`	définition d'un processus
	`instances` $o : f; \cdots o : f;$	déclarations d'instances
	`vars` $x : t; \cdots x : t;$	declaration de variables locales
	$B;$	instructions
	`}`	
$d ::=$	`machine` f `{`	définition d'une machine
	`memory` $x : t; \cdots x : t;$	déclarations de membres
	`instances` $o : f; \cdots o : f;$	déclarations d'instances
	`reset(){`B`}`	méthode d'initialisation
	`(`$x : t$`, ⋯, `$x : t$`)` `step(`$x : t$`, ⋯, `$x : t$`){`	méthode de pas synchrone
	`vars` $x : t; \cdots x : t; B$	
	`}`	
	`}`	
$B ::=$	$B; B$	séquence d'instructions
	$\mid e \mid$ `(`v`, ⋯, `v`)` `:=` e	expression ou multi affectation
	$\mid$ `switch(`e`){case` $0 : B_0; \cdots$ `case` $n : B_n$`}`	contrôle
	$\mid$ `while(`i `=` x`){`B`}`	boucle
$fun ::=$	$op \mid$ `read` $\mid$ `write` $\mid \ldots$	fonctions
$e ::=$	$i \mid v \mid op$`(`e`,` e`)` $\mid fun$`(`e`, ⋯, `e`)`	expressions
	$\mid o.m$`(`e`, ⋯, `e`)`	appel de la méthode m
$v ::=$	$x \mid$ `this.` x	variable locale ou registre

FIGURE 2.8: Grammaire du langage OBC

— Un flot du domaine $(T + \{abs\})^\infty$ est représenté par un flot de valeurs dans T^∞. Aux instants d'absence du flot, il peut prendre n'importe quelle valeur de T.
— Une équation n'est activée qu'aux instants où son horloge d'activation vaut true.
— Les nœuds sont compilés séparément, une unique fois, comme si leur horloge de base était l'horloge de base globale.

Il est direct de générer le code séquentiel scalaire correspondant à un pas synchrone de chaque équation, voir figure 2.9. Nous notons la fonction de traduction ⟪⟫ et la fonction C chargée de rajouter le contrôle induit par l'horloge d'activation. Chaque variable du noyau est traduite en une variable du même nom. La traduction globale est en figure 2.10.

2.5.3 *Nota bene* Nous ne montrons pas comment le type des variables est récupéré, car nous l'avons omis dans notre noyau. À la place, nous notons t_x le type de x.

Les équations Les équations sont mises en séquence conformément à la numérotation donnée par l'ordonnancement.

L'application d'un nœud se traduit par l'appel à la méthode `step` de l'instance de la classe associée à cette application. Cette instance doit être la même que celle ajoutée dans les objets de

Ajout du contrôle à une instruction :

$$\begin{aligned} C(\bullet, B) &= B \\ C(ck \texttt{ on } (i = c), B) &= C(ck, \texttt{switch}(c)\{\texttt{case}\, i : B\}) \end{aligned}$$

Traduction des équations en affectations :

$$\begin{aligned} (\!|\texttt{rec}\, eq_1 \texttt{ and} \cdots \texttt{and}\, eq_n|\!) &= (\!|eq_1|\!); \cdots ; (\!|eq_n|\!) \quad \text{avec } eq_k = \ldots =^{ck}_k \ldots \\ (\!|y =^{ck}_z i|\!) &= C(ck, y := i) \\ (\!|y =^{ck}_z op(x_1, x_2)|\!) &= C(ck, y := op(x_1, x_2)) \\ (\!|y =^{ck}_z i \texttt{ fby } x|\!) &= C(ck, y := \texttt{this}.\, y) \\ (\!|y =^{ck}_z x \texttt{ when } (i = p)|\!) &= C(ck, y := x) \\ (\!|y =^{ck}_z \texttt{merge } p\, (0 \texttt{ -> } x_0) \cdots (n \texttt{ -> } x_n)|\!) &= C\left(ck, \texttt{switch}(p)\{\begin{array}{cc} \texttt{case}\, 0 : y := x_0; \} \\ \vdots \quad \vdots \\ \texttt{case}\, n : y := x_n \end{array}\right) \\ (\!|(y_1, \cdots, y_m) =^{ck}_z f(x_1, \cdots, x_n)|\!) &= C(ck, (y_1, \cdots, y_m) := o_z.\, \texttt{step}(x_1, \cdots, x_n)) \end{aligned}$$

Traduction d'un programme :

$$\begin{aligned} (\!|ds; eq|\!) &= (\!|ds|\!); \mathrm{main}(eq); \\ (\!|d; ds|\!) &= \mathrm{def}(d); (\!|ds|\!) \end{aligned}$$

Figure 2.9: Traduction des équations du noyau vers Obc après ordonnancement

la classe par la fonction getObjs et que l'initialisation ajoutée par getObjsInit dans la fonction d'initialisation de la classe. C'est le numéro d'équation z qui permet de savoir quel objet correspond à quel appel. En effet un même nœud peut être appelé plusieurs fois, il y a alors plusieurs instances différentes de sa classe dans les objets de la classe de l'appelant.

L'équation de l'échantillonneur $y =^{ck} x$ `when` $(i = p)$ a pour horloge de base celle de y. Quand l'équation est activée, il faut faire l'affectation y `:=` x. C'est le contrôle ajouté par l'horloge d'activation qui effectue l'échantillonnage, comme dans la sémantique synchrone implicite.

Le registre synchrone L'équation du registre synchrone $y =^{ck} i$ `fby` x est spéciale puisqu'il faut pouvoir parler d'une valeur d'un instant passé. Cette valeur est stockée dans la mémoire de l'objet, dans une variable d'instance de même nom. On y accède en utilisant la construction `this.` y. La valeur courante du résultat du registre est ainsi y `:= this.` y.

Il faut de plus une affectation pour mettre à jour la mémoire `this.` y `:=` x. Celle-ci doit se faire impérativement *après* l'affectation de y, sous peine d'avoir dans y la valeur courante de x. La nouvelle valeur de `this.` y n'est pas lue dans l'instant, son affectation peut ainsi être placée à n'importe quel moment dès que x est défini. Par simplicité, toutes les mises à jour des mémoires sont rajoutées à la toute fin avec la fonction getRegsValue.

2.5.4 *Nota bene* **Initialisation des instances**
L'initialisation des registres est ajoutée dans la fonction d'initialisation de la classe avec la fonction getRegsInit. Ceci requiert que la méthode `reset` soit appelée au moins une fois avant de pouvoir utiliser la méthode `step`.

Programme principal Finalement, la fonction principale du programme est traduite en un processus principal qui a pour tâche d'exécuter la boucle de réaction du programme. Il a donc

$$
\begin{aligned}
&\text{main}(\texttt{(}y_1\texttt{,}\cdots\texttt{,}\ y_m\texttt{)} \texttt{=}^{\bullet}\ f\texttt{(}x_1\texttt{,}\cdots\texttt{,}\ x_n\texttt{)}) \ = \\
&\quad \texttt{processus\{} \\
&\qquad \texttt{instances o :}\ f\texttt{;} \\
&\qquad \texttt{vars}\ y_1 : t_{y_1}; \cdots y_m : t_{y_m}; x_1 : t_{x_1}; \cdots x_n : t_{x_n}; \\
&\qquad \texttt{o.reset()} \\
&\qquad \texttt{while(true)\{} \\
&\qquad\quad \texttt{read(}x_1\texttt{,}\cdots\texttt{,}\ x_n\texttt{)} \\
&\qquad\quad \texttt{(}y_1\texttt{,}\cdots\texttt{,}\ y_n\texttt{)} \texttt{ := o.step (}x_1\texttt{,}\cdots\texttt{,}\ x_n\texttt{)} \\
&\qquad\quad \texttt{write(}y_1\texttt{,}\cdots\texttt{,}\ y_n\texttt{)} \\
&\qquad \texttt{\}} \\
&\quad \texttt{\}} \\
&\text{def}(f\texttt{(}x_1\texttt{,}\cdots\texttt{,}\ x_n\texttt{) = (}y_1\texttt{,}\cdots\texttt{,}\ y_m\texttt{) with}\ eqs) \ = \\
&\quad \texttt{machine}\ f\ \texttt{\{} \\
&\qquad \texttt{memory}\ \text{getRegs}(eqs) \\
&\qquad \texttt{instances}\ \text{getObjs}(eqs) \\
&\qquad \texttt{reset()\{} \\
&\qquad\quad \text{getRegsInit}(eqs) \\
&\qquad\quad \text{getObjsInit}(eqs) \\
&\qquad \texttt{\}} \\
&\qquad \texttt{(}y_1\texttt{,}\cdots\texttt{,}\ y_m\texttt{) step(}x_1 : t_{x_1}\texttt{,}\cdots\texttt{,}\ x_n :: t_{x_1}\texttt{)\{} \\
&\qquad\quad \texttt{vars}\ \text{getVars}(eqs) \\
&\qquad\quad (\!|eqs|\!) \\
&\qquad\quad \text{getRegsValue}(eqs) \\
&\qquad \texttt{\}} \\
&\quad \texttt{\}}
\end{aligned}
$$

Fonctions de collectes sur une liste d'équations, pour f égale à getRegs, getRegsInit, getRegsValue, getObjs, getObjsInit ou getVars :

$$
\begin{aligned}
f(eq_1\ \texttt{and}\ eq_2)(acc) &= f(eq_2)(f(eq_1)(acc)) \\
f(\texttt{rec}\ eql) &= f(eql)()
\end{aligned}
$$

Les fonctions de collectes sur une équation :

$$
\begin{aligned}
\text{getRegs}(y \texttt{=}^{ck}_{z}\ i\ \texttt{fby}\ x)(acc) &= y : t_y; acc \\
\text{getRegsInit}(y \texttt{=}^{ck}_{z}\ i\ \texttt{fby}\ x)(acc) &= \texttt{this.}\ y\ \texttt{:=}\ i; acc \\
\text{getRegsValue}(y \texttt{=}^{ck}_{z}\ i\ \texttt{fby}\ x)(acc) &= C(ck, \texttt{this.}\ y\ \texttt{:=}\ x); acc \\
\text{getObjs}(\texttt{(}x\texttt{,}\cdots\texttt{,}\ x\texttt{) =}^{ck}_{z}\ f\texttt{(}x\texttt{,}\cdots\texttt{,}\ x\texttt{)})(acc) &= o_z : f; acc \\
\text{getObjsInit}(\texttt{(}x\texttt{,}\cdots\texttt{,}\ x\texttt{)=}^{ck}_{z} f\texttt{(}x\texttt{,}\cdots\texttt{,}\ x\texttt{)})(acc) &= o_z\texttt{.reset()}; acc \\
\text{getVars}(\texttt{(}y_1\texttt{,}\cdots\texttt{,}\ y_m\texttt{) =}^{ck}_{z}\ \ldots)(acc) &= y_1 : t_{y_1}; \cdots y_m : t_{y_m}; acc
\end{aligned}
$$

Figure 2.10: Traduction du noyau vers Obc après ordonnancement

une instance de la fonction principale qu'il initialise avant d'entrer dans la boucle. La boucle de réaction consiste à lire la valeur courante des entrées du programme, faire un pas synchrone de l'équation principale et écrire les sorties.

2.5.4 Optimisation du contrôle

Une des optimisations traditionnellement effectuées pour les performances est l'optimisation du contrôle. Dans notre cas, cela consiste à agglomérer des instructions gardées par le même contrôle. La séquence

```
switch(p) { case 0 : B_0 ··· case n : B_n };
switch(p) { case 0 : B'_0 ··· case n : B'_n }
```

s'optimise en :

```
switch(p) { case 0 : B_0; B'_0 ··· case n : B_n; B'_n }
```

Cette fusion n'est effectuée que sur des instructions en séquence. Le choix de l'ordre des affectations est donc intimement lié à l'optimisation. L'ordonnanceur a des heuristiques pour maximiser les fusions.

2.5.5 *Remarque* L'analyse flot de données est effectuée dans tous les compilateurs C modernes. Elle est dans le cas général impossible. Dans un code simple sans alias comme celui que l'on génère, on pourrait penser qu'elle est aisée, mais nos expériences montrent que l'ordonnancement que nous choisissons a une influence sur le code et les performances finales. Plus nous demandons au compilateur C d'optimiser, plus les résultats sont « aléatoires », parfois il vaut mieux que nous fusionnons au maximum, parfois nos fusions provoques des chutes de performances. Dans tous les cas, si l'on enlève les optimisation du compilateur C, maximiser la fusion est profitable.

2.5.5 Memalloc

Dans le langage Heptagon complet, il est possible d'utiliser des tableaux. La manipulation des tableaux dans un langage déclaratif équationnel comme Heptagon induit beaucoup de copies superflues si l'on effectue une compilation naïve comme celle présentée dans ce manuscrit. En effet, chaque affectation de tableau est une copie.

Partant de cette remarque, Cédric Pasteur dans son stage de M2 proposa une optimisation appelée Memalloc qui tente de modifier les tableaux en place tant que cela est possible. Comme pour l'optimisation du contrôle, l'ordonnancement choisi est crucial pour les performances de cette optimisation. Malheureusement, ces deux contraintes sur l'ordonnancement ne sont pas orthogonales, ce qui nécessite un arbitrage, chose toujours très délicate.

Pour donner au programmeur un contrôle sur les partages effectués, un système d'annotations semi-linéaires est proposé. Une même annotation sur plusieurs variables exprime le souhait que toutes ces variables soient stockées ensembles. Ce système d'annotation permet en outre de définir et d'importer des fonctions modifiant en place leurs arguments.

Nous avons participé à l'implémentation et à l'article [Ger+12] qui relate ces considérations.

2.6 Réinitialisation

2.6.1 Réinitialisation hyperchrone

La traduction des automates (cf. section 1.3.2) utilise l'équation `reset` eqs `every c` pour réinitialiser les équations eqs à chaque fois que `c` est vrai. Cette construction a été introduite dans Lucid Synchrone [HP00], puis utilisée dans [CPP05] pour compiler les automates. En Heptagon, comme dans ces articles, la condition `c` n'a pas à être de même horloge que les flots définis par eqs. En effet, nous acceptons des flots sur des sous-horloges dans les branches des automates. La réinitialisation de la branche nécessite alors une réinitialisation hyperchrone. C'est-à-dire que les flots réinitialisés sont sur une sous-horloge de l'horloge de la condition.

2.6.1 *Nota bene* Le choix de la réinitialisation hyperchrone est nécessaire si l'on souhaite pouvoir inliner les appels de nœuds (ceux ci peuvent contenir des flots plus lents que leurs entrées et sorties). Dans l'article fondateur d'Heptagon [Bie+08], cela n'était pas possible.

2.6.2 *Remarque* **La réinitialisation hyperchrone est suffisante**
Si les équations eqs ne sont pas sur une sous-horloge de `c`, il est toujours possible de compléter `c` avec des false pour se ramener au cas hyperchrone.

2.6.3 *Exemple*

```
reset p =• 0 fby 1
and x =•on(1=p) 42 when (1=p)
and y =•on(1=p) 7 fby x
every• c
```

c	false	false	false	true	false	true	true	false	false
p	0	1	1	0	1	0	0	1	1
x	.	42	42	.	42	.	.	42	42
y	.	7	42	.	7	.	.	7	42

Pendant les trois premiers instants, `c` vaut false, les équations montrent leur comportement normal. Au quatrième instant, `c` vaut `true`. La réinitialisation du registre synchrone `p` donne sa valeur d'initialisation (ici `0`). La réinitialisation de l'équation de `x` n'a pas d'impact puisqu'elle est sans état, mais `x` est absent car `p` vaut `0`. Similairement, le registre synchrone définissant `y` n'est pas activé. Pourtant il est nécessaire de le réinitialiser. On voit au cinquième instant qu'il fournit sa valeur d'initialisation.

2.6.2 Traduction source à source de la réinitialisation

Seules les expressions à état nécessitent la prise en compte de la réinitialisation. Dans notre noyau, cela nous ramène à deux cas, le registre synchrone et l'appel de fonction. Remarquons de plus que l'on peut considérer chaque équation indépendamment. L'encodage source à source est évoqué dans [HP00], où est aussi fait le constat de sa complexité et lourdeur. Toutefois, comme nous le verrons par la suite, il est nécessaire pour donner une sémantique de Kahn à la réinitialisation hyperchrone.

2.6.2.1 Réinitialisation de l'équation du registre synchrone

Cas synchrone Si le registre et la condition de réinitialisation sont synchrones, les choses sont simples :

`reset y =`^{ck}`i fby x every`^{ck} `c` $\equiv$ `t =`^{ck}`i fby x and y =`^{ck}`if c then i else t`

Ce qui est, in fine, traduit en Obc par :

$C(ck,$
```
  t := this.t;
  if (c) { y := i } else { y := t };
  this.t := x
```
)

Le code généré explicite bien le fait que le registre fonctionne comme auparavant, c'est son résultat qui est pris ou pas en considération suivant la valeur de `c`.

Cas hyperchrone Dans le cas hyperchrone, changer le résultat du registre n'est pas si facile, puisqu'il peut être lu après la réinitialisation. Deux solutions se profilent, avoir une condition de réinitialisation intelligente qui reste vraie tant que la réinitialisation n'a pas été faite, ou bien accélérer le registre pour le réinitialiser effectivement.

Pour simplifier, considérons `c` d'horloge ck et `y` et `x` sur ck `on p` avec `p` un pointeau booléen :

```
reset y =^{ck on p} i fby x every^{ck} c
```

La version qui adapte la condition de réinitialisation est :

```
rec reset y =^{ck on p} i fby x every^{ck on p} adapted_c
and adapted_c =^{ck on p} (c or need_reset) when p
and need_reset =^{ck} merge p false (((false fby need_reset) or c) whenot p)
```

`need_reset` stocke le fait que `c` a été présent sans que `p` le soit, ainsi, la condition de réinitialisation synchrone est donnée par la conjonction de `need_reset` et `c`.

La version qui accélère le registre pour retomber sur une réinitialisation synchrone s'écrit :

```
rec x_fast =^{ck} merge p x (y_fast whenot p)
and reset y_fast =^{ck} i fby x_fast every^{ck} c
and y =^{ck} y_fast when p
```

Dans les deux solutions, un registre est sur l'horloge la plus rapide. Ceci est similaire à l'écriture du nœud `current` (cf section 1.2.6.1). Nous devons garder de l'information entre les instants lents.

La dernière solution est préférable puisqu'elle n'ajoute pas de registre supplémentaire. Son adaptation à une horloge avec plus de pointeaux est immédiate : pour chaque pointeau, un étage de `merge` est nécessaire pour le calcul de `x_fast` et un étage de `when` pour `y_fast`.

2.6.2.2 Compilation source à source de la réinitialisation d'un appel de nœud :

Quand un appel de nœud est réinitialisé, nous devons faire en sorte de réinitialiser ses équations. Pour effectuer la réinitialisation sans avoir besoin d'inliner le corps de chaque nœud, il est nécessaire en compilation source à source de rajouter une entrée implicite à tous les nœuds. Le corps du nœud est réinitialisé par cette entrée `c` :

`f(x1, ..., xn) = (y1, ..., yn) with` eqs

devient :

`f(x1, ..., xn, c) = (y1, ..., yn) with reset` eqs `every•` `c`

En réalité, si nous voulons avoir des horloges bien formées pour Heptagon, il faut rajouter plus de fils que cela. Comme l'horloge de `c` peut être plus rapide que celle des autres entrées, nous devons aussi fournir un (ou plusieurs) pointeau décrivant ceci. L'horloge de base du nœud est modifiée et le code serait de la forme :

`f(x1, ..., xn, p, c) = (y1, ..., yn) with reset` eqs`[• on p/•] every•  c`

Pour résumer, la traduction source à source est très coûteuse, car nous devons propager à *tous* les registres l'information de réinitialisation. Tous les nœuds et registres voient leur horloge d'activation être égale à l'horloge de réinitialisation la plus rapide. Il suffit donc d'une réinitialisation sur l'horloge de base globale pour que tout le code soit actif à tous les instants, ce qui est désastreux pour les performances.

2.6.3 Gestion de la réinitialisation dans le noyau :

Le choix fait en Heptagon est de garder dans le noyau l'opération de réinitialisation hyperchrone. Cependant, à la génération de code, nous souhaitons avoir l'information de réinitialisation là où elle est utile, c'est-à-dire aux registres et aux appels de nœuds. Pour ce faire, nous changeons dans le noyau équationnel l'équation du registre et l'application de fonction, qui prennent une liste de conditions de réinitialisation :

$$\begin{aligned} r &::= [] \mid x^{ck}, r \\ ex &::= i \texttt{ fby } x \texttt{ every } r \mid \ldots \\ app &::= f\texttt{(}x\texttt{, }\cdots\texttt{, }x\texttt{) every } r \end{aligned}$$

Par convention, nous omettons la réinitialisation si elle est de liste vide, ainsi :

$$i \texttt{ fby } x \texttt{ every } [] \equiv i \texttt{ fby } x \qquad\qquad f\texttt{(}x\texttt{, }\cdots\texttt{, }x\texttt{) every } [] \equiv f\texttt{(}x\texttt{, }\cdots\texttt{, }x\texttt{)}$$

La nouvelle classe r représente une liste de conditions de réinitialisation. En effet, nous devons traiter des réinitialisation imbriquées. La fonction de traduction vers ce noyau se contente de prendre les conditions de réinitialisation pour les ajouter à la liste de réinitialisation des registres et fonctions. Tous les cas non évoqués sont l'identité :

$$\begin{aligned} \text{reset}(\texttt{reset } eqs \texttt{ every}^{ck} \ c) &= \text{reset}(c^{ck})(eqs) \\ \text{reset}(c^{ck})(eq \texttt{ and } eqs) &= \text{reset}(c^{ck})(eq) \texttt{ and } \text{reset}(c^{ck})(eqs) \\ \text{reset}(c^{ck})(i \texttt{ fby } x \texttt{ every } r) &= i \texttt{ fby } x \texttt{ every } (c^{ck}, r) \\ \text{reset}(c^{ck})(f(x_1, \cdots, x_n) \texttt{ every } r) &= f(x_1, \cdots, x_n) \texttt{ every } (c^{ck}, r) \end{aligned}$$

Les règles de pointage du noyau sont amendées pour vérifier la cohérence des horloges des flots de réinitialisation. Aucun lien n'est fait entre ces horloges et l'horloge de l'équation :

$$\textsc{Reset}\ \frac{\gamma(x) = ck \qquad \gamma \vdash r}{\gamma \vdash x^{ck}, r} \qquad\qquad \textsc{Fby}\ \frac{\gamma(x) = ck \qquad \gamma \vdash r}{\gamma \vdash i \texttt{ fby } x \texttt{ every } r \ :: ck}$$

$$\textsc{App}\ \frac{\Gamma, \gamma \vdash \texttt{(}y_1\texttt{, }\cdots\texttt{, }y_m\texttt{) =}^{ck} f\texttt{(}x_1\texttt{, }\cdots\texttt{, }x_n\texttt{)} \qquad \gamma \vdash r}{\Gamma, \gamma \vdash \texttt{(}y_1\texttt{, }\cdots\texttt{, }y_m\texttt{) =}^{ck} f\texttt{(}x_1\texttt{, }\cdots\texttt{, }x_n\texttt{) every } r}$$

L'ordonnancement doit prendre en compte les dépendances crées par la réinitialisation :

$$\text{uses}([]) = \emptyset \qquad \text{uses}(i\ \texttt{fby}\ x\ \texttt{every}\ r) = \text{uses}(r)$$
$$\text{uses}\left(x^{ck}{,}r\right) = \{x\} \cup \text{uses}(ck) \cup \text{uses}(r) \qquad \text{uses}(app\ \texttt{every}\ r) = \text{uses}(app) \cup \text{uses}(r)$$

Les changements dans la compilation se réduisent à rajouter la réinitialisation, avant la traduction des équation réinitialisées :

$$(\!|y =_z^{ck'} i\ \texttt{fby}\ x\ \texttt{every}\ c^{ck}, r|\!) = \begin{vmatrix} C(ck, \texttt{switch}(c)\texttt{\{case}\,\text{true}\,\texttt{:}\ \texttt{this.}\ y\ \texttt{:=}\ i\texttt{\}}); \\ (\!|y =_z^{ck'} i\ \texttt{fby}\ x\ \texttt{every}\ r|\!) \end{vmatrix}$$
$$(\!|(y_1, \cdots, y_m) =_z^{ck'} f(x_1, \cdots, x_n)\ \texttt{every}\ c^{ck}{,}r|\!) = \begin{vmatrix} C(ck, \texttt{switch}(c)\texttt{\{case}\,\text{true}\,\texttt{:}\ o_z\texttt{.reset()\}}); \\ (\!|(y_1, \cdots, y_m) =_z^{ck} f(x_1, \cdots, x_n)\ \texttt{every}\ r|\!) \end{vmatrix}$$

2.6.4 *Remarque* **Optimisation du contrôle**
Les instructions que nous générons avec la traduction ne sont pas gardées par la même horloge. Par simplicité, la fonction de traduction les génère collées séquentiellement. Pourtant seul l'ordre « réinitialisation puis activation » doit être respecté. Ce choix diminue les opportunités de l'optimisation du contrôle qui s'effectue de proche en proche. On souhaiterait que ce soit l'ordonnancement qui prenne ce choix en charge. La présentation de ce manuscrit ne permet pas de gérer cela proprement. Il faudrait rajouter des équations dans le noyau effectuant uniquement la réinitialisation et des dépendances invisibles devraient être ajoutées pour s'assurer que l'initialisation est effectuée avant.

2.6.5 *Exemple* Un compteur sur une horloge lente, avec réinitialisation, s'écrit et se compile :

```
natrp(c, p) = (y) with
  rec y =•onp 0 fby x every c•
  and x =•onp y + 1
```

```
machine natrp {
  memory y : int;
  reset() { this.y = 0 }
  (y) step (c : bool; p : bool) {
    vars y : int; x :int;
    switch(c) {case true : this.y := 0}
    switch(p) {
      case true :
        y := this.y;
        x := y + 1;
        this.y := x;
    }
  }
```

2.6.4 Sémantiques de la réinitialisation

Nous ne donnons pas de sémantique de Kahn non synchrone pour des raisons expliquées en remarque 2.6.6. Si l'on désire une sémantique de Kahn non synchrone, il est toujours possible d'effectuer la traduction source à source (cf. section 2.6.2) avant de donner la sémantique. Notez cependant que la traduction nécessite de connaître les horloges des flots.

2.6.4.1 Sémantiques de Kahn synchrone du registre réinitialisé

Tout d'abord, nous devons donner une sémantique à la liste de condition de réinitialisation. Sa valeur est un mot booléen qui indique à chaque instant s'il y doit y avoir réinitialisation ou pas. Ainsi, la sémantique ressemble à une conjonction des conditions mais dont les arguments n'ont pas même horloge :

$$\left[\!\left[c^{ck}, r\right]\!\right]^{\mathbf{b}}_{H,\rho} = \mathrm{resetOr}([\![ck]\!]^{\mathbf{b}}_{H,\rho}, \rho(c), [\![r]\!]^{\mathbf{b}}_{H,\rho}) \qquad [\![[]]\!]^{\mathbf{b}}_{H,\rho} = \mathrm{false}^{\omega}$$
$$\mathrm{resetOr}(\mathrm{true}.\mathbf{ck}, c.\mathbf{c}, v.\mathbf{r}) = (c \vee v).\, \mathrm{resetOr}(\mathbf{ck}, \mathbf{c}, \mathbf{r})$$
$$\mathrm{resetOr}(\mathrm{false}.\mathbf{ck}, \mathit{abs}.\mathbf{c}, v.\mathbf{r}) = v.\, \mathrm{resetOr}(\mathbf{ck}, \mathbf{c}, \mathbf{r})$$

La sémantique synchrone du registre était donnée, dans sa forme générale, avec deux expressions en entrées. Pour insister sur le fait que la réinitialisation touche uniquement le registre et non ses arguments (qui doivent être réinitialisés par eux-mêmes), nous prenons des variables en entrées. Si la réinitialisation indique false, le registre se comporte comme avant, seul le dernier cas est nouveau. La réinitialisation vide la mémoire du registre et le relance :

$$\begin{aligned}
[\![x_1 \texttt{ fby } x_2 \texttt{ every } r]\!]^{\mathbf{b}}_{H,\rho} &= \mathrm{fby}^{\mathcal{S}}_{\mathit{abs}}(\rho(x_1), \rho(x_2))\Big([\![r]\!]^{\mathbf{b}}_{H,\rho}\Big) \\
\mathrm{fby}^{\mathcal{S}}_{\mathit{abs}}(v.\mathbf{v}, u.\mathbf{u})(\mathrm{false}.\mathbf{r}) &= v.\, \mathrm{fby}^{\mathcal{S}}_{u}(\mathbf{v}, \mathbf{u})(\mathbf{r}) \\
\mathrm{fby}^{\mathcal{S}}_{u'}(v.\mathbf{v}, u.\mathbf{u})(\mathrm{false}.\mathbf{r}) &= u'.\, \mathrm{fby}^{\mathcal{S}}_{u}(\mathbf{v}, \mathbf{u})(\mathbf{r}) \\
\mathrm{fby}^{\mathcal{S}}_{w}(\mathit{abs}.\mathbf{v}, \mathit{abs}.\mathbf{u})(\mathrm{false}.\mathbf{r}) &= \mathit{abs}.\, \mathrm{fby}^{\mathcal{S}}_{w}(\mathbf{v}, \mathbf{u})(\mathbf{r}) \\
\mathrm{fby}^{\mathcal{S}}_{u}(\mathbf{v}, \varepsilon)(\mathrm{false}.\mathbf{r}) &= u \\
\mathrm{fby}^{\mathcal{S}}_{\mathit{abs}}(v.\mathbf{v}, \varepsilon)(\mathrm{false}.\mathbf{r}) &= v \\
\mathrm{fby}^{\mathcal{S}}_{w}(\mathbf{v}, \mathbf{u})(\mathrm{true}.\mathbf{r}) &= \mathrm{fby}^{\mathcal{S}}_{\mathit{abs}}(\mathbf{v}, \mathbf{u})(\mathrm{false}.\mathbf{r})
\end{aligned}$$

2.6.6 *Remarque* **Il n'existe pas de sémantique de Kahn non synchrone pour la réinitialisation hyperchrone**
Pour savoir quand réinitialiser, il faut savoir à quels instants le flot de réinitialisation est présent. Ceci empêche de pouvoir donner une sémantique de Kahn. Prenons un exemple, soit $\rho(x_1) = 0^{\omega}$, $\rho(x_2) = 1^{\omega}$ et $\rho(c) = (\mathrm{true}.\mathit{abs})^{\omega}$, nous avons $\left[\!\left[x_1 \texttt{ fby } x_2 \texttt{ every } c^{ck}\right]\!\right]^{\mathbf{true}}_{H,\rho} = (0.1)^{\omega}$, tandis qu'avec $\rho(c') = (\mathrm{true})^{\omega}$ nous avons $\left[\!\left[x_1 \texttt{ fby } x_2 \texttt{ every } c'^{ck'}\right]\!\right]^{\mathbf{true}}_{H,\rho} = 0^{\omega}$. Mais dans la sémantique de Kahn, nous n'avons pas les valeurs d'absences, ni les horloges. L'environnement ρ_k doit être égal à $\mathrm{clean}(\rho)$, ainsi nous aurions $\rho_K(x_1) = 0^{\omega}$, $\rho_K(x_2) = 1^{\omega}$, $\rho_K(c) = \rho_K(c') = \mathrm{true}^{\omega}$. Retrouver que $[\![x_1 \texttt{ fby } x_2 \texttt{ every } c]\!]^{\mathcal{K}}_{\rho_K} = (0.1)^{\omega}$ n'est pas possible car il faut aussi que ce soit égal à $[\![x_1 \texttt{ fby } x_2 \texttt{ every } c']\!]^{\mathcal{K}}_{\rho_K}$ qui vaut 0^{ω}. L'information supplémentaire qu'il faudrait donner correspond aux fils que nous devions rajouter dans la compilation source à source.

2.6.4.2 Sémantiques de Kahn synchrone de l'appel de fonction réinitialisé

Il serait possible de faire comme pour le registre avec l'appel de fonction si nous sortions l'état de la fonction. La réinitialisation consisterait à changer cet état avant de le redonner à la fonction. C'est le choix effectué dans la présentation initiale co-itérative de Lucid Synchrone (cf. [CP98]). Ce n'est pas notre choix de présentation de sémantique car ceci ne correspond pas

à notre compilation. Pour rappel, nous utilisons l'interprétation suivante :

$$[\![app]\!]^{\mathbf{b}}_{H,\rho} = \left[\!\!\left[f^{ck}\texttt{(}x_1\texttt{, }\cdots\texttt{, }x_n\texttt{)} \right]\!\!\right]^{\mathbf{b}}_{H,\rho} = H(f)([\![ck]\!]^{\mathbf{b}}_{H,\rho})(\rho x_1, \cdots, \rho x_n)$$

Les fonctions sont dans l'environnement sous la forme de fonction de flots. La réinitialisation doit se faire par dessus leur appel. Tant qu'il n'y a pas de réinitialisation, le résultat de l'application doit être utilisé tel quel. À partir d'un instant de réinitialisation, le résultat doit être remplacé par un nouveau, correspondant à l'application effectuée *à partir de cet instant*. Pour cela, nous devons garder les entrées pour les refournir à chaque appel.

Prenons le cas concis d'une fonction avec une entrée et une sortie. Nous utilisons la fonction continue $\mathrm{reset}^{\mathcal{S}}_f(\mathbf{y})(\mathbf{ck},\mathbf{x})(\mathbf{r})$, avec $\mathbf{r}$ le flot de réinitialisation, $\mathbf{ck}$ l'horloge d'activation, $\mathbf{x}$ le flot d'entrée courant et $\mathbf{y}$ le flot de sortie courant :

$$\begin{aligned}
\left[\!\!\left[f^{ck}\texttt{(}x\texttt{) every }r \right]\!\!\right]^{\mathbf{b}}_{H,\rho} &= \mathrm{reset}^{\mathcal{S}}_{f^{ck}}\Big(H(f)([\![ck]\!]^{\mathbf{b}}_{H,\rho})(\rho(x))\Big)(\mathbf{b},\rho(x))\Big([\![r]\!]^{\mathbf{b}}_{H,\rho}\Big) \\
\mathrm{reset}^{\mathcal{S}}_{f^{ck}}(w_y.\mathbf{y})(b.\mathbf{b}, w_x.\mathbf{x})(\mathrm{false}.\mathbf{r}) &= w_y.\,\mathrm{reset}^{\mathcal{S}}_{f^{ck}}(\mathbf{y})(\mathbf{b},\mathbf{x})(\mathbf{r}) \\
\mathrm{reset}^{\mathcal{S}}_{f^{ck}}(\mathbf{y})(b.\mathbf{b}, w_x\mathbf{x})(\mathrm{true}.\mathbf{r}) &= w'_y.\,\mathrm{reset}^{\mathcal{S}}_{f^{ck}}(\mathbf{y}')(\mathbf{b},\mathbf{x})(\mathbf{r}) \\
&\qquad \text{avec } w'_y.\mathbf{y}' = H(f)([\![ck]\!]^{b.\mathbf{b}}_{H,\rho})(w_x.\mathbf{x})
\end{aligned}$$

Pour écrire aisément la version complète avec plusieurs entrées et plusieurs sorties, nous allons directement manipuler l'environnement ρ. Nous définissons la concaténation $\rho.\rho' = [\forall x, x \mapsto \rho(x).\rho'(x)]$ et les fonction continues hd et tl pour récupérer la tête et la queue d'un environnement :

$$\mathrm{hd}\,\rho = [\forall x, x \mapsto v \text{ si } v.\mathbf{x} = \rho(x)] \qquad\qquad \mathrm{tl}\,\rho = [\forall x, x \mapsto \mathbf{x} \text{ si } v.\mathbf{x} = \rho(x)]$$

L'écriture suit alors exactement ce qu'il se passait avec une seule entrée et une seule sortie :

$$\begin{aligned}
[\![app \texttt{ every } r]\!]^{\mathbf{b}}_{H,\rho} &= \mathrm{reset}^{\mathcal{S}}_{app}\Big([\![app]\!]^{\mathbf{b}}_{H,\rho}\Big)(\mathbf{b},\rho)\Big([\![r]\!]^{\mathbf{b}}_{H,\rho}\Big) \\
\mathrm{reset}^{\mathcal{S}}_{app}(\rho_{out})(b.\mathbf{b},\rho_{in})(\mathrm{false}.\mathbf{r}) &= (\mathrm{hd}\,\rho_{out}).\,\mathrm{reset}^{\mathcal{S}}_{app}(\mathrm{tl}\,\rho_{out})(\mathbf{b},\mathrm{tl}\,\rho_{in})(\mathbf{r}) \\
\mathrm{reset}^{\mathcal{S}}_{app}(\rho_{out})(b.\mathbf{b},\rho_{in})(\mathrm{false}.\mathbf{r}) &= (\mathrm{hd}\,\rho'_{out}).\,\mathrm{reset}^{\mathcal{S}}_{app}(\mathrm{tl}\,\rho'_{out})(\mathbf{b},\mathrm{tl}\,\rho_{in})(\mathbf{r}) \quad \text{avec } \rho'_{out} = [\![app]\!]^{\mathbf{b}}_{H,\rho_{in}}
\end{aligned}$$

2.6.5 Sémantique synchrone implicite de la réinitialisation d'équations

Maintenant que nous sommes capables de manipuler les environnements, nous pouvons en fait directement donner la sémantique de la réinitialisation d'équations. Faisons le dans le cadre de la sémantique synchrone implicite. L'idée est toujours la même, nous lançons le calcul, tant qu'il n'y a pas de réinitialisation le comportement est l'identité sur l'environnement de sortie et l'environnement d'entrée est consommé petit à petit. Quand il y a réinitialisation, nous réexécutons les équations dans l'environnement tronqué du passé :

$$\begin{aligned}
[\![\texttt{reset }eqs \texttt{ every } r]\!]^{\mathcal{C}}_{H,\rho} &= \mathrm{reset}^{\mathcal{C}}_{eqs}\Big([\![eqs]\!]^{\mathcal{C}}_{H,\rho}\Big)(\rho)\Big([\![r]\!]^{\mathbf{b}}_{\mathbf{true}}\Big) \\
\mathrm{reset}^{\mathcal{C}}_{eq}(\rho_{out})(\rho_{in})(\mathrm{false}.\mathbf{r}) &= (\mathrm{hd}\,\rho_{out}).\,\mathrm{reset}^{\mathcal{C}}_{eq}(\mathrm{tl}\,\rho_{out})(\mathrm{tl}\,\rho_{in})(\mathbf{r}) \\
\mathrm{reset}^{\mathcal{C}}_{eq}(\rho_{out})(\rho_{in})(\mathrm{true}.\mathbf{r}) &= (\mathrm{hd}\,\rho'_{out}).\,\mathrm{reset}^{\mathcal{C}}_{eq}(\mathrm{tl}\,\rho'_{out})(\mathrm{tl}\,\rho_{in})(\mathbf{r}) \quad \text{avec } \rho'_{out} = [\![eqs]\!]^{\mathcal{C}}_{H,\rho_{in}}
\end{aligned}$$

2.7 Conclusion

Nous avons présenté les bases du langage Heptagon. En premier lieu de manière informelle, puis à l'aide de sémantiques dénotationnelles. Nous avons vu que Heptagon a une sémantique de Kahn et que cette sémantique est élégante mais l'utilisation de flots infinis éloigne de la compilation. C'est grâce à la sémantique synchrone que l'on commence à entrevoir comment le calcul peut se faire pas à pas. L'écriture d'une sémantique synchrone, mais dont la valeur représentant l'absence peut être prise quelconque, est importante pour les performances du code généré. Cette dernière requiert l'ajout de constructions sémantiques simulant le contrôle qui est généré dans le code final. Grâce à cette présentation, nous avons insisté sur l'importance de l'horloge de base des nœuds, assurant l'efficacité et la modularité de la compilation. La réinitialisation peut se coder de manière flot de données mais ceci induit un surcoût inacceptable. Ainsi, comme cela est fait dans Heptagon, nous avons ajouté au noyau des constructions de réinitialisation et donné leur sémantique en simulant le contrôle correspondant à leur compilation.

Nous avons donc les bases pour comprendre les problématiques induites par la distribution de code Heptagon que nous regardons en partie II, ainsi qu'un exemple complet de sémantique de Kahn pour attaquer la partie III.

Deuxième partie

Les futures en Heptagon

Nous ajoutons à Heptagon la possibilité de lancer un calcul asynchrone et la notion associée de future [BH77 ; LS88]. Quand le programmeur déclare un calcul asynchrone, il ne récupère pas le résultat immédiatement, mais un future le remplaçant. La fin de la tâche asynchrone associée n'a besoin de terminer qu'au moment où le programmeur explicitement demande la valeur du future.

Le concept de future est adapté aux langages fonctionnels pour lesquels un calcul asynchrone est équivalent à un appel de fonction. En Heptagon, les fonctions manipulent des flots. De cette manière, quand le programmeur déclare un appel de fonction comme asynchrone, ce qu'il récupère est un flot de futures. Nous montrons comment cela permet d'exécuter de manière parallèle le programme `slow_fast` [GNP06] en utilisant les futures dans le code source. Les ajouts peuvent être traités comme de simples annotations. En particulier, si nous les effaçons, nous retrouvons un programme de sémantique dénotationnelle équivalente. En donnant la sémantique de Kahn de ces annotations, nous prouvons cette préservation de la sémantique.

Les futures donnent au programmeur la possibilité d'explicitement indiquer le début et la fin d'un calcul. Ce pouvoir, dans un contexte de programmation synchrone, rend les futures très expressifs. En chapitre 4, nous donnons de multiples squelettes de programmation parallèle utilisant les futures. En particulier, nous soulignons l'importance de la construction de réinitialisation d'une fonction à état. Cette réinitialisation permet d'extraire du parallélisme de donnée de son exécution. Finalement, nous montrons qu'il est possible de sortir du cadre fonctionnel déterministe en ajoutant une primitive pour savoir si un calcul d'un future est fini. Cet oracle permet de programmer des comportement adaptatifs, dépendant de l'exécution.

Les futures sont très expressifs et dynamiques. Ils requièrent une gestion dynamique de la mémoire. Pour cela, nous présentons en détail la génération de code faite par Heptagon. Notre génération Java ne s'en préoccupe pas et utilise le ramasse-miettes du langage. Par contre, notre génération C++ alloue statiquement la mémoire. Pour cela, nous interdisons l'échappement des futures et instrumentons le code généré pour réutiliser en place la mémoire.

Chapitre 3

Les futures en Heptagon

3.1 Généralités sur les futures dans les langages séquentiels

3.1.1 Présentation

Le terme « future » vient de H. Baker et C. Hewitt [BH77]. La terminologie et l'implémentation du concept sont divers, mais l'idée fondamentale reste la même : un future est un calcul dont l'évaluation est lancée de manière concurrente, avant que le résultat ne soit attendu.

L'utilisation la plus répandue des futures est de cacher la latence d'une communication. Considérons le code séquentiel C suivant en :

```
File_content f = get_file_from_server(x);
Graphic_content g = do_some_computations(x);
display(g, f);
```

Si `get_file_from_server()` prend du temps, l'exécution va l'attendre alors que nous pourrions calculer `g` pendant l'attente de la réponse. Pour masquer cette latence, nous allons lancer la requête de manière asynchrone et au lieu d'attendre le résultat, nous récupérons *instantanément* un future `f_f` associé à ce calcul. Ainsi, le calcul de `g` s'exécutera en parallèle de la requête. Ce n'est que quand le résultat de la requête est nécessaire que nous attendons son résultat avec l'appel *bloquant* à `get()` :

```
Future<File_content> f_f = async get_file_from_server();
Graphic_content g = do_some_computations();
display(g, f_f.get());
```

Les futures apportent une abstraction très intéressante de la concurrence. Ils n'exposent pas le moyen de communication utilisé, qui peut être par mémoire partagée, envois de messages, etc. La valeur que porte le future est *réalisée*, une et une seule fois, par la tâche asynchrone associée. Cette réalisation n'est pas gérée par le programmeur. En particulier, il n'est fourni aucun moyen de définir ou de changer la valeur d'un future (rien d'équivalent à `f_f.set(42)` n'existe). *Le seul moyen de créer un future est de lancer un calcul asynchrone.*

Typage `async` prend n'importe quelle fonction et permet de lancer son calcul en parallèle. Avec un typage à la ML, nous avons :

$$\texttt{async} : \forall t.\forall t'.(t \to t') \to t \to (\texttt{future}\, t') \qquad \texttt{get} : \forall t.(\texttt{future}\, t) \to t$$

Ainsi, `future` est un constructeur de type. Les futures forment une monade [PGF96], mais nous n'entrons pas dans les détails ici, car Heptagon est un langage du premier ordre et ces considérations ne nous sont pas utiles. La propriété importante à retenir est l'égalité :

$$\lambda x.\texttt{get}(\texttt{async} f x) = \lambda x. f x$$

Préservation de la sémantique fonctionnelle L'équivalence fonctionnelle entre la version avec et sans les futures est délicate dans un cadre général. Dans notre exemple, si l'appel `do_some_computations()` touche au fichier lu par `get_file_from_server()`, il n'est pas possible d'affirmer que la valeur de `f_f.get()` est égale à celle de `f` dans l'exécution sans future.

Heureusement, en se restreignant à un monde fonctionnel pur, il est le même, que le calcul fasse appel à des futures ou pas.

Dans [FF99], les auteurs donnent trois sémantiques équivalentes à un lambda calcul avec des futures. Pour montrer l'innocuité de l'utilisation des futures, la première traite les constructions `async` et `get` comme des constructeurs. En plus de la β-réduction, elle admet les réductions $\texttt{async}(e) \rightarrow \texttt{async}(e')$ et $\texttt{get}(e) \rightarrow \texttt{get}(e')$ si $e \rightarrow_\beta e'$ et finalement, si v est une valeur finale (terme en forme normale), alors $\texttt{get}(\texttt{async}(v)) \rightarrow v$.

De nombreuses variantes de ces démonstrations existent, dans des cadres plus où moins riches, intégrant par exemple pour AliceML [NSS06] les références à la ML. L'idée a retenir est que, si e est un terme clos, les deux termes ci-dessous confluent grâce aux réductions sus-définies :

$$(\lambda x.e')e \qquad\qquad (\lambda y.e'[\texttt{get}(y)/x])(\texttt{async}(e))$$

3.1.2 Historique

Les premiers [1] à avoir introduit le concept des futures étaient Friedman et Wise [FW76], sous le nom de promesses. Le terme de future apparait dans [BH77], avec l'idée claire qu'un future est un élément de première classe manipulé par le programmeur. Ceci n'est pas le cas en MultiLisp [Hal85], premier langage à effectivement proposer le mécanisme de futures. Le `get` est implicitement effectué, par le runtime de MultiLisp, quand la valeur est nécessaire. Ces futures sont dits implicites et cette gestion par le runtime permet à Halstead de proposer diverses constructions pour calculer en parallèle (y compris les futures).

L'approche avec des futures implicites correspond aux questions de l'époque, concernant les différentes stratégies d'évaluations. En particulier, cela rapproche de l'évaluation paresseuse avec des glaçons (*thunks* de ALGOL-60). Mais au lieu de figer un calcul pour plus tard, la fermeture créée est donnée à un évaluateur concurrent. L'évaluation avec des futures est qualifiée de spéculative [HA87] ou de gloutonne, car même si le programme ne requiert jamais le résultat d'un future, son calcul est lancé et sera effectué. Cette évaluation gloutonne, au lieu de minimiser la quantité de calcul, maximize le parallélisme.

C'est dans le langage Argus que les futures, tels que nous les entendons, sont proposés [LS88]. Ils y sont fortement typés et le `get` des futures est un opérateur explicite. La principale motivation des auteurs est de pouvoir traiter et donner un comportement bien déterminé en présence d'exceptions levées par les calculs des futures.

À peu près à la même époque sont proposées les I-structures [ANP89 ; AN90]. Ce sont des tableaux à une seule écriture, dont l'état de chaque case est géré individuellement. Chaque case est appelée une I-var. La différence entre une I-var et un future est la gestion par le programmeur de la réalisation : l'I-var a une méthode `set`, cependant, le système assure l'unicité de l'appel à

1. Il semble que ce soit le cas, bien que la référence en question ne soit pas trouvable sur internet.

`set` pour chaque I-var (en règle générale avec une erreur à l'exécution). Les M-structures sont des I-structures, dont la lecture `get` vide la case, permettant une nouvelle écriture.

Ainsi, une I-var est la structure sous jacente d'un future, celle nécessaire par le runtime ou la librairie pour proposer l'appel asynchrone renvoyant un future, tandis que le calcul associé réalisera le future avec une unique écriture. Notons que les I-var ont été à la base de l'inspiration pour Concurrent Haskell [PGF96], qui en propose une version Monadic, les `MVar`. Les I-var et les futures se retrouvent en Haskell dans le package `Control.Monad.Par`, dont l'implémentation utilise des `MVar`. Dans la nouvelle norme du C11 et C++11, les I-var sont appelées des promesses, les futures étant la restriction en lecture seule d'une promesse. Nous élaborons ce point ci-après.

De nombreux langages supportent les futures, une partie beaucoup plus restreinte gère ceux implicites. Parmi ceux-ci, AliceML [NSS06] est le plus abouti, avec des futures implicites et explicites, mêlés à l'orde supérieur et à des références à la ML. Il pave la route à l'intégration de futures et autres constructions de parallélisme pour Lucid Synchrone.

3.1.3 Les futures explicites en librairie C++11

Dans cette section, nous donnons une idée de l'implémentation des futures explicites dans une librairie. Notez que les futures existent dans la librairie standard de C++11, mais elle n'était pas encore très efficace, ni robuste au commencement de cette thèse.

Une promesse est en règle générale considérée comme la structure de donnée sous-jacente aux futures. Elle est la coquille qui permet au producteur de lui donner sa valeur et au consommateur de demander de manière bloquante sa valeur. C'est aussi elle qui assure la cohérence des données entre les threads concurrents du producteur et des consommateurs.

Une promesse est une structure avec deux champs. La valeur stockée dans `v` et un drapeau `ready` qui indique si la valeur est prête/réalisée. Le drapeau est le point critique entre le producteur et le consommateur. Nous utilisons un `atomic<bool>` qui permet d'assurer que les lectures et écritures sont atomiques. L'initialisation (ligne 4) est faite à false. La méthode `set` permet au producteur de définir la valeur, ce qui a pour effet de mettre le drapeau `ready` à vrai. La méthode `get` attend que le drapeau `ready` soit à vrai pour renvoyer la valeur stockée.

```
1  template<typename T>
2  class promise {
3    T v;
4    atomic<bool> ready = {false}; // ensure correct initialization
5  public :
6    void set(const T& v) {
7      assert(!is_ready());        // ensure unicity of value
8      this->v = v;
9      this->ready.store(true, memory_order_release);
10   }
11   const T& get() const {
12     while(!is_ready())          // loop until ready
13       this_thread::yield();
14     return v;
15   }
16   bool is_ready() const {
17     return ready.load(memory_order_acquire);
18   }
19 };
```

Notre code repose sur l'hypothèse qu'en tout et pour tout, il y aura au maximum un producteur et un appel à `set`. C'est le modèle mémoire qui assure que si un consommateur utilise `get`, il va lire la valeur de `v` écrite par le producteur. Nous n'entrons pas dans les détails techniques du C++11, mais nous stockons la valeur true dans `ready` avec la sémantique *release*, *après* avoir mis la bonne valeur dans `v`. La lecture de `v` n'est faite par `get` qu'*après* avoir vu `ready` à true avec la sémantique *acquire*, ce qui assure que la valeur de `v` lue est celle écrite par le producteur.

Un future est finalement une référence constante vers une promesse, dont la fonction `set` est rendue inaccessible. Cela peut se programmer ainsi :

```
template<typename T>
class future {
  promise<T> * r;
public :
  future(promise<T> *p) : r(p) { }
  T get() { return r->get(); }
  bool is_ready() { return ready; }
  /* ... */
};
```

Pour assurer qu'une promesse est utilisée correctement et qu'un future est toujours associé à un calcul, la bibliothèque ne devrait exporter que la classe des futures et une fonction `async`. Ci-dessous une version qui prend un pointeur de fonction `f` dont les entrées sont de type `I` et les sorties `O`. `async(f, x)` crée un thread pour calculer `f(x)`, tout en renvoyant immédiatement le future qui portera le résultat :

```
template<typename O, typename I>
void compute(O (*f) (I), I x, promise<O> *p) { p->set(f(x)); };

template<typename O, typename I>
future<O> async(O (*f)(I), I x) {
  promise<O> *p = new promise<O>();        // Who will delete it ?
  new std::thread(compute<O, I>, f, x, p);
  return future<O>(p);
};
```

3.1.1 *Nota bene* Problèmes de gestion mémoire

La promesse est allouée dans `async` pour s'assurer qu'elle est réalisée par le thread créé par `async`. Cette encapsulation pose la question de qui va désallouer la promesse. La solution générale envisageable est d'utiliser des pointeurs intelligents [Sut02] comme les `shared_ptr` à l'intérieur des futures pour que la promesse soit supprimée quand il n'y a plus de futures associés. Mais la suppression de la promesse n'est pas possible tant que le calcul n'est pas fini, puisque le résultat va y être stocké. Il faut donc que le destructeur attende que la promesse soit réalisée. Cependant, dans ce cas, un code aussi simple que `async(f); async(g);` risque de séquentialiser le calcul de `f` et de `g` car le premier future retourné n'est pas utilisé et le compilateur peut choisir d'appeler son destructeur avant l'appel à `async(g)`.

Notez que toutes ces difficultés existent avec les futures de la bibliothèque standard C++, elles sont discutées dans [Sut12].

Deux options s'offrent à nous : utiliser un langage avec un ramasse-miettes ou bien suivre nous même la durée de vie des promesses pour gérer la mémoire. Notre backend Java profite

de la gestion automatique de la mémoire, tandis que notre backend C++ utilise une gestion ad-hoc de la mémoire sans allocation dynamique. Nous présentons cela en section 5.2. Avant cela, regardons comment les futures interagissent avec les fonctions de flots d'Heptagon.

3.2 Motivation des futures en Heptagon

Dans cette section, nous ne nous intéressons pas aux problèmes de gestion de la mémoire. Nous allons donc présenter la génération de code Java.

3.2.1 Parallélisme synchrone

Heptagon n'est pas un langage séquentiel. Les équations que nous écrivons sont mises en parallèle synchrone. Ainsi l'exemple initial (cf. section 3.1.1) que nous avons donné pour l'utilisation des futures n'est plus convaincant. Il se transpose en :

```
node display () returns (graphic_content g, file_content f)
let
  f = get_file_from_server();
  g = do_some_computations();
tel
```

Les deux équations de `display` sont indépendantes et « mises en parallèle synchrone ». Cette concurrence pourrait être directement utilisée par le compilateur. Ici, un calcul de composantes connexes des calculs suffirait à détecter que ces deux calculs sont indépendants et peuvent donc être lancés en parallèle à chaque instant. Cette approche automatique n'est pas intéressante en pratique. Le nombre de composantes concurrentes est rapidement trop important, du fait même du langage source. De plus, les tâches concurrentes sont en règle générale très peu gourmandes en calculs, car elles doivent se finir dans l'instant (hypothèse synchrone). Exploiter automatiquement (et naïvement) toute la concurrence submerge de microscopiques tâches, dont la gestion et la distribution sont désastreuses sur des plateformes conventionnelles. C'est pour cette raison que des plateformes comme KEP [LH12] ou STARPro [Yua+09] sont imaginées (en l'occurrence pour Esterel). Ou bien que des systèmes d'inférences de partitionnement existent [Del08 ; DGP08].

3.2.2 Latence dans les programmes multi-rythmes

Pour obtenir un gain, il faut être en mesure de séparer un morceau qui calcule longtemps. Ceci apparait dans les systèmes muti-rythmes. Nous reprenons un exemple clef des travaux sur la distribution de programmes synchrones multi-rythmes effectués par Caspi et Girault avec l'outil Ocrep [CG95 ; GNP06]. L'exemple est initialement tiré d'un modèle industriel d'une usine nucléaire. Nous le motivons avec un nouveau contexte.

Considérons que le programme doit calculer l'intégrale d'une fonction compliquée. Pour se faire, il a deux intégrateurs, `slow` qui intègre fidèlement mais son efficacité n'est bonne que si on lui demande d'intégrer une longue plage et `fast` qui est approximatif, mais efficace même pour de toute petites avancées. Les deux intégrateurs prennent en argument la taille du pas demandé et l'état.

Nous définissons qu'une unité de temps correspond à un appel à `slow_fast` (un pas du programme). L'entrée `big_step` détermine dans combien de temps une valeur précise sera nécessaire. Ainsi, nous calculons à tous les instants une approximation avec `fast` et lançons `slow` pour `big_step` instants, avec l'aide de `countdown` (cf. page 30) :

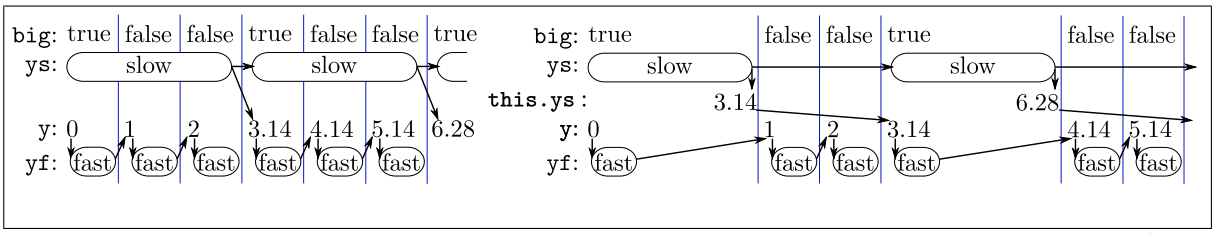


Figure 3.1: Chronogrammes temporels de `slow_fast`. Chaque ovale est un appel à une fonction de transition. Les traits verticaux délimitent les instants synchrones, les flèches sont les dépendances de données. Notez que l'état de `slow` ajoute une dépendance entre les appels à sa fonction de transition.

Les opérateurs postfixés d'un point (ici `+.` et `*.`) sont les opérations arithmétiques sur les flottants. Le code que nous donnons à `fast` et `slow` est volontairement simple, de sorte que les chronogrammes restent clairs. `slow` intègre 1.046666666 tandis que `fast` intègre 1. Ils seraient dans la réalité bien plus complexes.

```
node slow_fast(big_step : int)
returns (y : float)
var ys, yf : float; big : bool;
let
  big = countdown(big_step);
  ys = 0.0 fby slow(big_step when big, ys);
  yf = 0.0 fby fast(1.0, y);
  y = merge big ys (yf when not big);
tel
```

```
fun fast(h,i:float) = (o:float)
let
  o = i +. (h *. 1.0)
tel

node slow(h,i:float) = (o:float)
let
  o = i +. (h *. 1.046666666)
tel
```

Considérons l'appel `slow_fast(3)` :

big	true	false	false	true	false	false	true	false	false	true	...
ys	0			3.14			6.28			9.42	...
yf	0	1	2	3	4.14	5.14	6.14	7.28	8.28	9.28	...
y	0	1	2	3.14	4.14	5.14	6.28	7.28	8.28	9.42	...

En ne considérant que les dépendances de données, le calcul de `slow` devrait pouvoir s'exécuter parallèlement aux trois calculs rapides, comme dessiné dans l'exécution gauche de la figure 3.1.

Cependant, la compilation d'Heptagon requiert le résultat de `slow` à la fin de l'instant, pour le stocker dans le registre. Dans ce cas, la maximum de parallélisme que nous pourrions obtenir est dans l'instant entre `slow` et `fast` (cf exécution droite de la figure). Nous soulignons le point critique dans le code généré :

```
1 class Slow_fast {
2   float ys, float yf;                          // memories
3   Fast fast; Slow slow; Countdown countdown;   // instances
4   public Slow_fast() {
5     fast = new Fast(); slow = new Slow(); countdown = new Countdown();
6   }
7   public void reset() {
8     this.ys = 0.0; this.yf = 0.0;                    // Reinit memories
9     fast.reset(); slow.reset(); countdown.reset(); // Reinit instances
```

```
10   }
11   public float step(int big_step) {
12     float y;
13     big = countdown.step(big_step);
14     if (big) {
15       y = this.ys;
16       this.ys = slow.step(big_step, this.ys);
17     } else {
18       y = this.yf;
19     }
20     this.yf = fast.step(1.f, y);
21     return y;
22   }
23 }
```

Pour ne pas bloquer inutilement, nous souhaitons stocker dans le registre un future du résultat de l'appel à `slow.step`. Le code généré serait :

```
14 if (big) {
15   y = this.ys.get();
16   this.ys = async(slow.step, big_step, this.ys.get());
17 } else {
```

Toutefois, ce code n'est pas correct. La fonction `async` usuelle (et celle que nous avons définie en C++ en section 3.1.3) requiert un terme clos. Or `slow.step` est une fonction membre, elle dépend et modifie en place l'instance `slow`.

3.2.3 L'encapsulation async d'un nœud

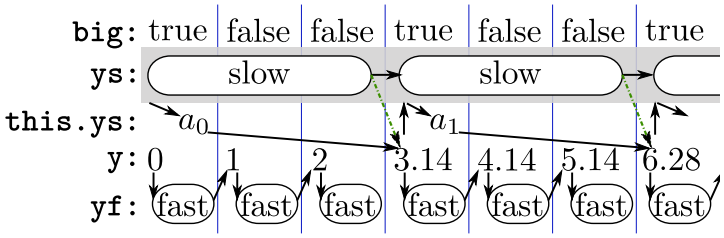
La solution que nous proposons est d'appeler tout le nœud de manière asynchrone et non simplement sa fonction de transition. Pour cela, nous liftons `async` aux fonctions de flots et `get` aux flots de futures. Ils se retrouvent alors dans le code source Heptagon, respectivement avec la syntaxe `async` et l'opérateur point d'exclamation (`!`).

Le code Heptagon de `slow_fast` découplant le calcule de `slow` s'écrit alors :

```
node slow_fast_a(big_step : int) returns (y : float)
var ys : future float; yf : float; big : bool;
let
  big = countdown(big_step);
  ys = (async 0.0) fby async slow(big_step when big, !ys);
  yf = 0.0 fby fast(1.0, y);
  y = merge big (!ys) (yf whenot big);
tel
```

Notez que la variable `ys` change de type et devient un flot de futures de `float`, tout comme le registre qui la stocke. Pour l'initialiser, nous utilisons la syntaxe `async` i qui produit un future à partir de la valeur instantanée i. Finalement, la valeur du future est récupérée uniquement quand elle est nécessaire avec (`!ys`).

L'appel avec `async` lance un thread d'exécution pour l'instance `slow`. Il est représenté par la zone grisée dans le chronogramme suivant. Les futures (a_0 a_1, etc) sont produits immédiatement. Les flèches pointillées correspondent à la réalisation du future et donc à une dépendance de donnée implicite :



3.2.1 *Nota bene* Par la suite, nous omettons la dépendance implicite des futures pour des raisons de lisibilité.

3.2.3.1 Encapsulation `async` d'un nœud

La compilation séparée d'Heptagon génère la classe `Slow` indépendamment de ses contextes d'appels. Pour ne pas changer toute la compilation, `async slow` va produire une nouvelle classe `Async_Slow` encapsulant une instance de `Slow`. Cette machinerie gère un thread pour exécuter une instance de `Slow`. Les entrées fournies à la méthode `step` de `Async_Slow` sont transmises dans l'ordre à celle de l'instance au travers de la queue `q`. Chaque entrée est associée à une promesse qui recevra le résultat du calcul correspondant :

```
class Async_Slow {
  Slow instance;
  BoundedQueue q;
  Async_Slow(int N) {                // Constructor
    instance = new Slow();
    q = new BoundedQueue(N);
    Thread t = new Thread(){         // Create worker thread
      public void run() {            // Worker function
        Promise<Float> p;
        float x;
        instance.reset();            // Initialize instance
        while(true) {                // Simulation loop of the instance
          (p, x) = q.pop();          // Pseudo code with tuples
          p.set(instance.step(x));
    } } };
    T.start();                       // Run worker thread
  }
  Future<Float> step(float x) {
    Promise<Float> p = new Promise<Float>();
    q.push(p, x);                    // Pseudo code with tuples
    return p;
  }
  void reset () { /* ... */ }
}
```

Notez que la file est bornée, de taille `N`, nous reparlerons de cette borne en section 4.1. Nous discuterons plus tard des questions de réinitialisation (cf. section 4.4).

3.2.2 *Nota bene* `Future`, `Promise` ET `StaticFuture`
Dans notre génération de code, nous utilisons les futures fournis par Java depuis la version 5.0 avec l'interface `Future`. Cependant, la notion de promesse n'existe pas. Dans Java 8 arrive

les `CompletableFuture` qui sont des promesses. En attendant, nous utilisons les `FutureTask` fournis avec Java qui rendent le code moins compréhensible.

Finalement, pour faciliter la traduction de `async` i, nous avons la classe `StaticFuture` qui implémente l'interface `Future` et prend i en argument de son constructeur.

3.2.3.2 Code généré pour `slow_fast_a`

La génération de code pour `slow_fast_a` est presque inchangée. Le gros du travail a été la production de l'encapsulation `Async_Slow`. Les futures sont utilisés là où le code source l'indique et l'opérateur (`!`) est traduit par la méthode `get`. Nous soulignons les différences par rapport à `Slow_fast` :

```
class Slow_fast_a {
  Future<Float> ys; float yf;                          // memories
  Fast fast; Async_Slow slow; Countdown countdown;  // instances
  public Slow_fast_a() {
    fast = new Fast(); slow = new Async_Slow(); countdown = new Countdown();
  }
  public void reset() {
    this.ys = new StaticFuture<Float>(0.0); this.yf = 0.0;  // reinit memories
    fast.reset(); slow.reset(); countdown.reset();          // reinit instances
  }
  public float step(int big_step) {
    float y;
    big = countdown.step(big_step);
    if (big) {
      y = this.ys.get();
      this.ys = slow.step(big_step, this.ys.get());
    } else {
      y = this.yf;
    }
    this.yf = fast.step(1.f, y);
    return y;
  }
}
```

3.3 Les futures dans Heptagon

L'ajout des futures explicites dans Heptagon est motivé par deux raisons principales :

— Fournir un moyen de programmer le parallélisme et la répartition dans le *code source* d'un langage synchrone dataflow.

— Changer au minimum la compilation et préserver le sémantique synchrone des programmes.

Les futures explicites remplissent parfaitement ces conditions. Ils peuvent être vus comme des annotations. De plus, le programmeur a le contrôle du lancement des calculs parallèles avec `async`, ainsi que celui de la fin des calculs en demandant le résultat d'un future. Finalement, la compilation est inchangée si un nœud ne contient pas d'appel asynchrone. S'il en contient, les changements correspondent au code source, plus la génération de l'encapsulation asynchrone de chaque nœud appelé avec `async`.

3.3.1 Tuples et typage des nouveaux opérateurs

Trois changements sont apportés au langage Heptagon, le type polymorphe `future` t, l'opérateur `get` noté (`!`) et l'opérateur d'ordre supérieur `async`. Ils ont le type lifté aux flots, de leur homonyme scalaire présenté en section 3.1.1.

Heptagon étant un langage du premier ordre sans polymorphisme, `async` ne peut pas être écrit dans une librairie. Dans tous les cas, nos langages cibles ne le sont pas non plus, ce qui requiert la génération de l'encapsulation `Async_N`, pour chaque nœud `N` appelé avec `async`.

3.3.1 *Remarque* **`async` d'une constante**
Le choix de restreindre `async` à l'application d'un nœud n'enlève pas d'expressivité. Toutefois, il est souvent utile de créer un future à partir d'une constante comme nous l'avons vu avec `slow_fast`. Écrire une fonction pour chaque constante que l'on veut mettre dans un future est quelque chose de lourd et coûteux. Nous acceptons la syntaxe `async` i pour représenter le future constant dont la valeur est la constante i. Ainsi `42` $\equiv$ `!(async 42)`.

Future d'un produit et produit de futures En Heptagon, le type produit n'existe pas, ce qui est compensé par le fait que les fonctions sont n-aires. Un nœud `f() returns (int; bool)` s'appelle de manière asynchrone avec l'équation `(x, y) = async f();` en déclarant les variables avec `x : future int; y : future bool;`

Il y a une conversion implicite d'un flot de futures de tuple en un tuple de flots de futures. Le typage de `async` s'écrit plus correctement avec $t_1, \cdots, t_n$ des types non produits :

$$\texttt{async} : \forall t, t_1, \cdots, t_n.(t \to (t_1 \times \cdots \times t_n)) \to t \to (\texttt{future}\, t_1 \times \cdots \times \texttt{future}\, t_n)$$

En Java, nous compilons les tuples avec des classes nommées `Tuple`n avec n le nombre d'éléments. Le $i$ème élément est le champ `c`i du tuple. Ainsi la méthode step de la classe `F` produit une valeur de type `Tuple2<Integer,Boolean>`. Pour ne pas compliquer la génération de `Async_F`, les tuples ne sont pas gérés spécialement et elle retourne un `Future` de `Tuple2<Integer,Boolean>`. C'est à l'appel que nous utilisons une fonction auxiliaire `at_to_ta2`, pour transformer le résultat en `Tuple2<Future<Integer>, Future<Boolean>>`. Ainsi, l'appel `(x, y) = async f()` produit :

```
Async_F f;
/* ... */
Tuple2 out = Pervasives.at_to_ta2(f.step());
Future<Boolean> y = (Future<Boolean>)(out.c1);
Future<Integer> x = (Future<Integer>)(out.c0);
```

La fonction `at_to_ta2` serait de type ML $\forall t_1, t_2.\, \texttt{future}(t_1 \times t_2) \to \texttt{future}\, t_1 \times \texttt{future}\, t_2$. Son code est simple, bien que verbeux :

```
static Tuple2 at_to_ta2 (Future<Tuple2> at) {
                Future<Object> t0 = new Future<Object>() {
                        public Object get() { return at.get().c0; }
                        /* ... */
                };
                Future<Object> t1 = new Future<Object>() {
                        public Object get() { return at.get().c1; }
                        /* ... */
                };
```

```
            return new Tuple2(t0,t1);
    }
```

3.3.2 *Nota bene* **Tuples en Java**
La gestion des tuples en Java est longue et verbeuse. Dans le code que nous présentons, nous utiliserons du pseudo code pour des raisons de concision et de lisibilité.

3.3.2 Pointage

Pour ne pas changer la sémantique synchrone du programme, un future doit, à tout moment pouvoir être remplacé par sa valeur. Ainsi, les opérateurs `!` et `async` doivent être transparents pour le pointage. Avec une notation à la Lucid Synchrone (ML), nous aurions les signatures :

$$\texttt{async} :: \forall \alpha, \beta.(\alpha \to \beta) \to \alpha \to \beta \qquad \texttt{!} :: \forall \alpha.\alpha \to \alpha$$

Cependant, les horloges d'Heptagon sont des types dépendants. Le cas problématique est celui où β dépend de la valeur d'une des sorties. Reprenons le nœud `positifs` présenté en section 1.2.6.1. Il est de signature $\forall \bullet.\bullet \to (c :: \bullet, \bullet \texttt{ on } c)$. Son appel asynchrone `async positifs(x);` définit deux flots `(ac, ay)`, dont le premier est un pointeau du second. La règle de pointage que nous avons écrite pour `async` va donner l'horloge $\bullet$ `on ac` à `ay`. Mais `ac` est un future, sémantiquement, l'horloge de `ay` devrait être $\bullet$ `on !ac`.

Une solution est de convertir implicitement les pointeaux de type future en pointeaux conventionnels en ajoutant (`!`). Nous l'avons implémenté, mais cela est trompeur pour le programmeur qui peut penser désynchroniser un appel quand en réalité le code généré à besoin de la valeur du pointeau et va donc l'attendre.

3.3.3 *Nota bene* Nous imposons dans la suite, que pour appeler une fonction avec `async`, les pointeaux de ses sorties soient des entrées et non des sorties. Formellement, la signature $\sigma = \forall \bullet, \forall x_1, \cdots, x_n, y_1, \cdots, y_m, (x_1 :: ck_1, \cdots, x_n :: ck_n) \to (y_1 :: ck'_1, \cdots, y_m :: ck'_m)$ peut être utilisée si $\forall 1 \leq j \leq m, \forall 1 \leq i \leq m, y_j \notin FV(ck'_i)$ (cf. figure 2.6 pour les définitions).

3.3.3 Sémantique de référence

La question sémantique admet deux approches. La première opérationnelle, qui tente de rendre compte de l'évaluation concurrente des appels asynchrones. Nous avons proposé une telle sémantique dans l'article [CGP12]. Avec du recul, nous pensons que cela est une figure de style qui n'apporte aucune information utilisable. Cela ne fait que plagier le code de l'encapsulation que nous avons donné en section 3.2.3.1, tout en noyant l'information dans des mathématiques nécessairement complexes pour représenter un système effectuant des transitions de manière concurrente.

Nous donnons ici les sémantiques dénotationnelles. Elles permettent de bien mettre en valeur la simplicité du concept des futures et de prouver la préservation de la sémantique.

3.3.3.1 Sémantique de Kahn

Bien que notre syntaxe empêche l'utilisation de `async` sur autre chose qu'une constante ou un appel de nœud, sa sémantique de Kahn (cf. section 2.1.3) est définie sur n'importe quelle expression e. Chaque élément du flot est encapsulé dans un future par `async` et décapsulé par

(!). Les flots avaient leurs valeurs dans T, ils sont maintenant dans $Tf ::= T \mid \text{future}(Tf)$.

$$[\![\texttt{async}\, e]\!]^{\mathcal{K}}_{H,\rho} = \text{async}^{\mathcal{K}}\left([\![e]\!]^{\mathcal{K}}_{H,\rho}\right) \qquad \text{async}^{\mathcal{K}}(v.\mathbf{v}) = \text{future}(v).\,\text{async}^{\mathcal{K}}(\mathbf{v})$$
$$\text{async}^{\mathcal{K}}((\mathbf{v0}, \cdots, \mathbf{vk})) = (\text{async}^{\mathcal{K}}(\mathbf{v0}), \cdots, \text{async}^{\mathcal{K}}(\mathbf{vk}))$$
$$[\![\texttt{!}\, e]\!]^{\mathcal{K}}_{H,\rho} = \text{bang}^{\mathcal{K}}\left([\![e]\!]^{\mathcal{K}}_{H,\rho}\right) \qquad \text{bang}^{\mathcal{K}}(\text{future}(v).\mathbf{v}) = v.\,\text{bang}^{\mathcal{K}}(\mathbf{v})$$

3.3.3.2 Sémantique de Kahn synchrone

Avec des flots synchrones (cf. section 2.2), les annotations des futures n'ont pas d'influence sur l'absence de valeur. Les filtrages $(*)$ sont nouveaux par rapport à la sémantique de Kahn :

$$[\![\texttt{async}\, e]\!]^{\mathbf{b}}_{H,\rho} = \text{async}^{\mathcal{S}}\left([\![e]\!]^{\mathcal{K}}_{H,\rho}\right) \qquad \text{async}^{\mathcal{S}}(v.\mathbf{v}) = \text{future}(v).\,\text{async}^{\mathcal{S}}(\mathbf{v})$$
$$\text{async}^{\mathcal{S}}(abs.\mathbf{v}) = abs.\,\text{async}^{\mathcal{S}}(\mathbf{v}) \quad (*)$$
$$\text{async}^{\mathcal{S}}((\mathbf{v0}, \cdots, \mathbf{vk})) = (\text{async}^{\mathcal{S}}(\mathbf{v0}), \cdots, \text{async}^{\mathcal{S}}(\mathbf{vk}))$$
$$[\![\texttt{!}\, e]\!]^{\mathbf{b}}_{H,\rho} = \text{bang}^{\mathcal{S}}\left([\![e]\!]^{\mathcal{K}}_{H,\rho}\right) \qquad \text{bang}^{\mathcal{S}}(\text{future}(v).\mathbf{v}) = v.\,\text{bang}^{\mathcal{K}}(\mathbf{v})$$
$$\text{bang}^{\mathcal{S}}(abs.\mathbf{v}) = abs.\,\text{bang}^{\mathcal{S}}(\mathbf{v}) \quad (*)$$

3.3.4 *Nota bene* Les flots avaient leurs valeurs dans $T + abs$, ils sont maintenant dans $Tf + abs$. Il est important de noter que jamais nous n'avons $\text{future}(abs)$.

3.3.3.3 Préservation de la sémantique

Nous allons prouver l'affirmation que les futures peuvent être considérés comme des annotations ne changeant pas la sémantique. Nous définissons la fonction $\text{sync}(P)$ qui supprime les futures du programme P. Elle est l'identité sur tous les éléments de la grammaire présentée en figure 2.1 mais doit supprimer les nouvelles primitives `async` et `!` :

$$\text{sync}(\texttt{async}\, f\texttt{(}e_1\texttt{,}\cdots\texttt{,}\, e_n\texttt{)}) = f\texttt{(}\text{sync}(e_1)\texttt{,}\cdots\texttt{,}\,\text{sync}(e_n)\texttt{)}$$
$$\text{sync}(\texttt{!}e) = \text{sync}(e)$$

3.3.5 *Remarque* Le compilateur propose l'option `-noasync` pour effectuer cette transformation de code. Notez que dans la syntaxe concrète, il faut aussi changer tous les types `future` t en t.

3.3.6 *Propriété*
Pour tout programme P bien typé et bien pointé, le programme $\text{sync}(P)$ l'est aussi.

Démonstration La preuve est immédiate pour le pointage puisque `async` et `!` y sont traités comme des identités. Pour le typage, toute expression qui était de type `future` t devient de type t. Si une expression manipulait un future, soit elle était `!`, soit elle était polymorphe (comme `fby`, `when` ou `merge`). Dans le premier cas `!` devient l'identité de t dans t ce qui est bien typé. Dans le second cas, le typage était de `future` t dans `future` t, il devient de t dans t.

Finalement, pour montrer que les calculs sont les mêmes, nous définissons aussi $\text{sync}(\rho)$ qui décapsule les flots dans l'environnement ρ. En notant wf toute valeur de $Tf + abs$ et w de $T + abs$:

$$\text{sync}(\rho) = [\forall x, x \mapsto \text{sync}(\rho(x))] \qquad \text{sync}(wf.\mathbf{x}) = \text{sync}(wf).\,\text{sync}(\mathbf{x})$$
$$\text{sync}(\text{future}(wf)) = \text{sync}(wf) \qquad \text{sync}(w) = w$$

Lemme 3.3.7. *Pour tout programme P bien typé, nous avons :*

$$\mathrm{sync}([\![P]\!]^{\mathcal{K}}_{\rho_0}) = [\![\mathrm{sync}(P)]\!]^{\mathcal{K}}_{\mathrm{sync}(\rho_0)} \qquad \mathrm{sync}([\![P]\!]^{\mathcal{S}}_{\rho_0}) = [\![\mathrm{sync}(P)]\!]^{\mathcal{S}}_{\mathrm{sync}(\rho_0)}$$

Démonstration Nous allons effectuer une preuve par induction sur le langage. Nous avons besoin de définir $\mathrm{sync}(H) = \left[f \mapsto \lambda(\mathbf{x_1}, \cdots, \mathbf{x_n}).\,\mathrm{sync}\left(H(f)(\mathbf{x_1}, \cdots, \mathbf{x_n})\right)\right]_{f \in \mathrm{dom}(H)}$

Le cœur de la preuve est l'égalité $\mathrm{sync}([\![e]\!]^{\mathcal{K}}_{H,\rho_0}) = [\![\mathrm{sync}(e)]\!]^{\mathcal{K}}_{\mathrm{sync}(H,\rho_0)}$.

Regardons le cas de e valant $\texttt{!}e'$, par définition de sync et induction nous avons $[\![\mathrm{sync}(e)]\!]^{\mathcal{K}}_{\mathrm{sync}(H,\rho_0)} = [\![\mathrm{sync}(e')]\!]^{\mathcal{K}}_{\mathrm{sync}(H,\rho_0)} = \mathrm{sync}([\![e']\!]^{\mathcal{K}}_{H,\rho_0})$, ainsi nous devons prouver que $\mathrm{sync}([\![e]\!]^{\mathcal{K}}_{H,\rho_0}) = \mathrm{sync}([\![e']\!]^{\mathcal{K}}_{H,\rho_0})$, soit pour tout flot $\mathbf{x}$, $\mathrm{sync}(\mathrm{bang}^{\mathcal{K}}(\mathbf{x})) = \mathrm{sync}(\mathbf{x})$, ce qui est le cas si le programme est bien typé (les éléments de $\mathbf{x}$ sont de la forme $\mathrm{future}(wf)$). Similairement, nous prouvons que $\mathrm{sync}([\![\texttt{async}\ f(e_1, \cdots, e_n)]\!]^{\mathcal{K}}_{H,\rho_0}) = \mathrm{sync}([\![f(e_1, \cdots, e_n)]\!]^{\mathcal{K}}_{H,\rho_0})$. Le cas de l'appel « normal » à une fonction est réglé par notre définition de $\mathrm{sync}(H)$. Pour les autres primitives, il n'y a pas de question particulière, appliquer sync aux entrées ou à leur sortie ne change rien, puisque si une entrée est un future, la seule manipulation possible est de l'oublier ou le copier tel quel.

Pour les équations et le point fixe, il suffit de noter que sync est une fonction continue.

Finalement, nous prouvons que $\mathrm{sync}([\![d]\!]^{\mathcal{K}}_{H}) = [\![\mathrm{sync}(d)]\!]^{\mathcal{K}}_{\mathrm{sync}(H)}$, qui se ramène grâce à notre définition de $\mathrm{sync}(H)$ à la propriété sur les équations $\mathrm{sync}([\![eqs]\!]^{\mathcal{K}}_{H,\rho_{in}}) = [\![\mathrm{sync}(eqs)]\!]^{\mathcal{K}}_{\mathrm{sync}(H,\rho_{in})}$ en notant eqs les équations de la fonction f de la définition d.

La version Kahn synchrone est identique, les horloges ne changent rien puisque les valeurs *abs* sont intouchées (cf. note 3.3.4) par toutes nos fonctions et que les horloges sont les mêmes (sync ne les change pas car les pointeaux n'ont pas de futures cf. section 3.3.2).

Le lemme nous permet de conclure avec le théorème de préservation.

3.3.8 Théorème **Préservation de la sémantique**
Pour tout programme P bien typé, dont les entrées et les sorties ne sont pas de type future, le programme calcule les mêmes sorties, avec ou sans calculs asynchrones :

$$[\![P]\!]^{\mathcal{K}}_{\rho_0} = [\![\mathrm{sync}(P)]\!]^{\mathcal{K}}_{\rho_0} \qquad [\![P]\!]^{\mathcal{S}}_{\rho_0} = [\![\mathrm{sync}(P)]\!]^{\mathcal{S}}_{\rho_0}$$

Démonstration Ceci est un corolaire du lemme. Il suffit de remarquer en plus que si un environnement ne contient pas de flots de type future, alors sync est l'identité, ainsi nous avons pour les entrées $\mathrm{sync}(\rho_0) = \rho_0$ et similairement avec les sorties.

3.3.4 Conclusion et travaux liés

L'outil Ocrep [GNP06] est capable de traiter l'exemple `slow_fast` et de générer du code concurrent qui aura l'exécution désirée. Pour cela, il effectue une analyse flot de données sur le code séquentiel OC (Object Code), un langage impératif séquentiel décrivant le programme sous forme d'automate (qui peut être de taille exponentielle). Cela lui permet d'optimiser le contrôle et les communications. Si l'on transpose l'analyse dans le code source, Ocrep arrive sur cet exemple à court-circuiter le registre `ys`. Alors, `slow_fast` ne stocke plus le résultat de `slow`, mais demande directement la précédente valeur quand cela est nécessaire. Ce sont donc les files de communications qui jouent le rôle du registre.

Cette approche a l'avantage d'être automatique. Cependant, c'est aussi sa faiblesse. L'outil est limité par la puissance de l'analyse statique effectuée. Si le programmeur ou l'optimisation ne découplent pas suffisamment les calculs, cela ne pourra que se constater sur le code généré.

L'utilisation de futures explicites permet de marquer, dans le code source, quand un calcul commence (`async`) et quand il finit (`!`). De plus, ces primitives peuvent être considérées comme des annotations et supprimées sans changer la sémantique du programme. La génération de code est inchangée et utilise les futures de la librairie du langage cible (Java dans ce chapitre). Cependant, Java n'est pas polymorphe d'ordre supérieur [2] et nous devons traduire l'opérateur `async` qui l'est. Nous générons ainsi une classe pour chaque fonction `f` appelée avec `async f`. Ce système d'encapsulation a le mérite de permettre l'utilisation de toute fonction déjà compilée ou définie de manière externe.

Nous ne donnons formellement la génération de code qu'en chapitre 5 pour le C++. En effet, celle-ci est plus complexe car elle ajoute la question de la gestion de la mémoire. De plus, avec les exemples d'utilisations du chapitre 4, nous aurons eu l'occasion d'expliquer la gestion de la réinitialisation d'un appel asynchrone.

2. Du moins pour l'instant, car Java 8 va changer les choses

Chapitre 4

Programmer le parallélisme avec des futures en Heptagon

4.1 La file d'entrée

L'encapsulation d'un nœud pour son appel asynchrone utilise une file pour communiquer les entrées à son thread de travail. Nous avons remarqué en section 3.2.3.1 que notre implémentation en Java utilise une file de taille bornée, avec un `push` bloquant si elle est pleine. La première remarque importante est que cette file peut toujours être de taille 1.

4.1.1 *Propriété* **Le programme ne peut bloquer**
Quelque soit le choix de la taille des files des encapsulations, l'exécution du programme ne peut bloquer.

Démonstration Le preuve repose sur le fait qu'un programme Heptagon sans future a une fonction de transition qui termine. De plus il n'a pas de cycle de dépendance entre les éléments des flots (cf. section 2.5.1).

Dans un programme Heptagon avec futures, il y a deux actions bloquantes, la réalisation d'un future `!x` et l'appel `async f ()` si la queue d'entrée de l'encapsulation est pleine. L'absence d'interblocage du programme vient de l'impossibilité d'avoir une dépendance récursive d'un future sur lui-même. En effet, il n'est possible de produire un future qu'en donnant *au préalable* les valeurs nécessaires à son calcul. Le graphe des appels asynchrones est un arbre (comme celui de tous les appels d'un programme Heptagon) et chaque appel n'a connaissance que des futures de ses fils. Les feuilles sont des programmes Heptagon sans future, ils ne bloquent donc pas. In fine la file de leur encapsulation est vide. Ainsi l'étage au dessus ne peut bloquer indéfiniment sur un `push` ou un `!` et nous concluons par induction.

Par défaut, si la taille de la file n'est pas spécifiée, elle est égale à 1. Pour la changer, le programmeur peut la donner en premier argument statique de `async`, par exemple `async<<3>> f(x)` créera une encapsulation avec une file de taille 3.

4.1.2 *Remarque* **Taille nulle**
En toute généralité, la queue pourrait être de taille nulle, c'est-à-dire forçant le rendez-vous entre le producteur et le consommateur. Cependant, notre implémentation ne le supporte pas et nous voyons peu d'intérêt comparé au surcoût pour l'implémentation. En effet, une queue à un producteur, un consommateur, de taille strictement positive s'écrit sans verrou.

Tandis que le rendez-vous nécessite un verrou ou bien deux queues à une place pour simuler le rendez-vous avec un accusé de réception.

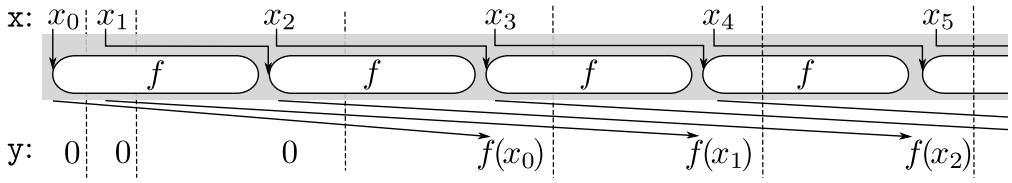
4.1.1 Découplage borné

Pour l'exemple `slow_fast_a`, il n'y a pas besoin d'avoir une taille de file supérieure à 1, car nous demandons le précédent résultat dans le même instant que nous fournissons l'entrée suivante. Si le calcul asynchrone de `slow` prend du retard, le programme principal bloquera pour récupérer la valeur du précédent future.

Pour mettre en évidence le rôle de la file d'entrée, le plus simple est de diminuer la pression sur la sortie en augmentant le nombre de `fby` sur la sortie. Prenons l'équation suivante :

```
y = !(async 0 fby<<3>> async f(x))
```

Nous voyons sur le chronogramme ci-dessous que les deux premiers instants peuvent être très petits puisque rien n'est bloquant, la file d'entrée n'est pas pleine et nous récupérons les valeurs des futures constants. Au troisième instant par contre, la file est pleine. Pour fournir x_2 à l'appel asynchrone de `f`, le programme principal doit attendre. C'est un phénomène de contrôle arrière (*back-pressure*) :

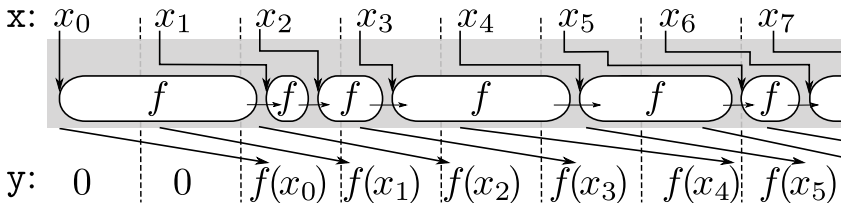


4.1.3 *Nota bene* La taille de la file a deux utilisations : borner statiquement la mémoire nécessaire, mais aussi borner statiquement le découplage.

De manière générique, pour découpler de `n` instants, nous pouvons utiliser le code suivant :

```
node decouple<<node f(int)=(int); n : int | n > 0 >> (x : int) returns (y : int)
let
  y = !(async 0 fby<<n>> async<<n>> f(x));
tel
```

Dans ce cas, `f` peut prendre jusqu'à `n` instants de retard. Ce découplage lisse le comportement d'un nœud. Ci-dessous, les instants sont réguliers, tandis que le calcul de `f` peut varier, tant que la somme de ses k premières activations ne durent pas plus de $k + n$ fois la durée d'un instant. Ci-dessous avec `n` valant 2 :



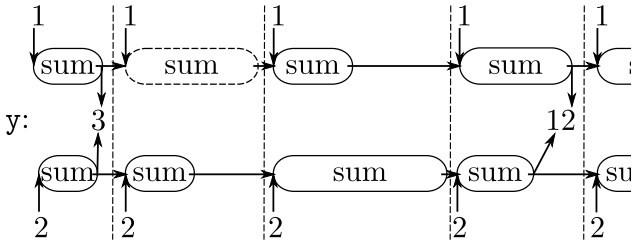
4.1.2 Désynchronisation partielle en échantillonnant les futures

La file d'entrée permet aussi de stocker du travail. Ceci est d'autant plus flagrant quand toutes les sorties ne sont pas utilisées. Considérons l'exemple `partial`, qui calcule deux sommes en parallèle et ne les additionne que si l'entrée `c` est vraie :

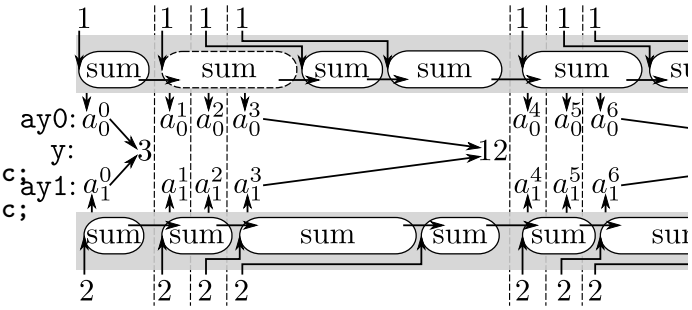
```
node partial (x1, x2 :int; c :bool)
returns (y : int; c : bool)
var y0, y1 : int
let
  y0 = sum(x1) when c;
  y1 = sum(x2) when c;
  y = !ay0 + !ay1;
tel
```

x1	1	1	1	1	1	1	1
x2	2	2	2	2	2	2	2
c	true	false	false	true	false	false	true
sum(x1)	1	2	3	4	5	6	7
sum(x2)	2	4	6	8	10	12	14
y	3	.	.	12	.	.	21

Pour l'exemple, considérons que le temps de calcul des sommes est très irrégulier (ci-contre). Chaque instant de l'exécution synchrone est aussi long que le maximum des calculs de l'instant. Ces synchronisations inutiles disparaissent en utilisant les futures (ci-dessous).




```
node partial_a(x1,x2:int; c:bool)
returns (y : int; c : bool)
var ay0, ay1 : future int
let
  ay0 = (async<<2>> sum(x1)) when c;
  ay1 = (async<<2>> sum(x2)) when c;
  y = !ay0 + !ay1;
tel
```



En échantillonnant les futures et non le résultat des calculs, nous oublions des futures et ne synchronisons tous les éléments que quand `c` est vrai. Si `c` n'est jamais vrai, sans une borne sur la taille des files d'entrées, le programme principal pourrait prendre une avance infinie et remplir les files avec une infinité de valeurs. En spécifiant `async<<2>>`, la désynchronisation ne sera pas influencée par la file d'entrées tant que `c` est vrai au moins une fois sur trois comme dans notre exemple.

4.2 Fork-Join

Un des squelettes usuels de parallélisme est le fork-join. Avec des futures dans un langage séquentiel, la phase de fork consiste en une séquence d'appels asynchrones. Elle est suivie de la phase de join qui revient à récupérer la valeur de tous les futures créés pendant la phase de fork. Pour deux tâches, nous pouvons écrire en Java :

```
Future<int> ay1 = async f(x); // Phase de fork
Future<int> ay2 = async f(x); // Phase de fork
y1 = !ay1; // Phase de join
y2 = !ay2; // Phase de join
```

4.2.1 Ordonnancement des appels asynchrones

Dans un langage déclaratif comme Heptagon, l'ordonnancement risque de faire perdre le parallélisme. Par exemple la fonction `careless` peut être compilée avec l'équation de `y1` placée avant celle de `ay2` puisqu'elle n'en dépend pas, ce qui rend le calcul de `y1` et `y2` séquentiel :

```
fun careless(x1, x2 : int) = (y : int)
var ay1, ay2 : future int; y1, y2 : int;
let
  ay1 = async f(x1);
  ay2 = async f(x2);
  y1 = !ay1;
  y2 = !ay2;
  y = y1 + y2;
tel
```

```
class Careless {
  Async_F f1; Async_F f2;
  /* ... */
  public int step(int x1, x2) {
    Future<int> ay1 = f1.step(x1);
    int y1 = ay1.get();
    Future<int> ay2 = f2.step(x2);
    int y2 = ay2.get();
    return y1 + y2;
  }
  /* ... */
```

L'ordonnanceur a donc l'heuristique de maximiser la durée de vie des futures. Cependant l'ordonnanceur est aussi en charge (cf. section 2.5.5) de minimiser l'utilisation mémoire et le code de contrôle. Il est difficile d'établir une priorité entre ces heuristiques.

En Lucid Synchrone ou Lucy-n, il y a une construction `let` eqs_1 `in` eqs_2 qui permet de forcer l'ordonnancement des équation eqs_1 avant les équations eqs_2. Cette construction n'existe pour l'instant pas en Heptagon.

Nous présentons ci-dessous deux solutions, une première qui utilise des tableaux et une deuxième, beaucoup plus fidèle à l'esprit d'Heptagon, qui utilise les instants logiques pour ordonner les calculs.

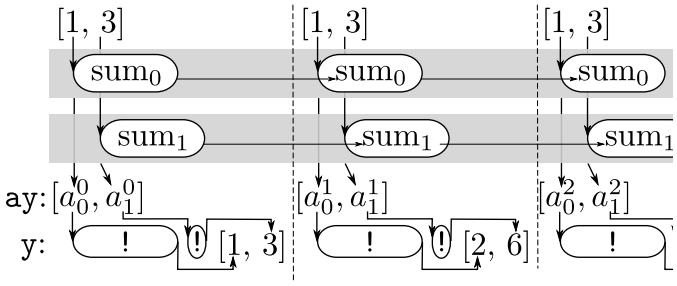
4.2.2 Version instantanée avec des tableaux

Considérons que nous souhaitons calculer `f` sur chaque élément d'un tableau `x` de taille `n`, ce qui s'écrit `map<<n>> f (x)` (cf. section 1.6). Comme chaque élément du tableau a sa propre instance de `f`, il est possible de les calculer en parallèle. La phase de fork est un `map` de `async f`, dont le résultat est un tableau de futures. Une fois ce tableau produit, nous sommes certains d'avoir lancé tous les calculs. Il suffit ensuite d'attendre les résultat avec un `map` de `!` :

```
node parallel_map <<n :int; node f(int)=(int)>> (x :int^n) returns (y :int^n)
var ay :(future int)^n;
let
  ay = map<<n>> (async f) (x);  (* Fork *)
  y = map<<n>> (!) (ay);        (* Join *)
tel
```

Un appel à `parallel_map<<2, sum>>([1,3])` génère le code et le chronogramme suivant :

```
class Parallel_map_2_sum {
  Async_Sum[] sum;
  public int[] step(int[] x) {
    int y = new int[2];
    Future[] ay = new Future[2];
    for(int i = 0; i<2; i++)
      ay[i] = sum[i].step(x[i]);
    for(i = 0; i<2; i++)
      y[i] = ay[i].get();
    return y;
  }
  /* ... */
```



Dans le chronogramme, nous avons marqué l'attente induite par la réalisation des futures en indiquant l'exécution de `!`. Le join est effectué séquentiellement. Ainsi c'est principalement la première récupération du future qui provoque une attente.

4.2.1 *Remarque* Heptagon est un langage strict et les tableaux sont des valeurs de première classe. C'est grâce à ceci que nous sommes certain que la phase de join se fera bien après le lancement de tous les calculs car la phase de join dépend du *tableau* des futures.

Si les tableaux étaient de simple macros comme en Lustre, le code serait expansé et le parallélisme redeviendrait dépendant de l'ordonnancement, comme évoqué précédemment en section 4.2.1.

4.2.3 Version temporelle

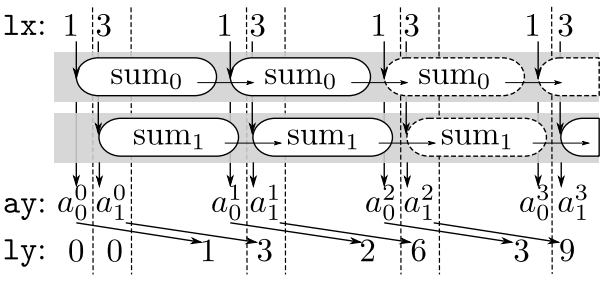
Dans un langage synchrone flot de données, il est courant de représenter un tableau comme le flot de ses éléments. Par exemple si `x` est un flot de tableaux de taille 2, `lx` est le flot qui résulte de l'« aplatissement » de `x` :

`x`	$[x_0, x_1]$		$[x_2, x_3]$		$[x_4, x_5]$		...
`lx`	x_0	x_1	x_2	x_3	x_4	x_5	...

Notez que `lx` est présent deux fois plus souvent que `x`. La partie statique d'Heptagon n'est pas encore assez expressive pour écrire le fork-join temporel de manière paramétrique. Voici la version pour des tableaux de taille 2 :

```
node temporal_fj_2<<node f(int)=(int)>>
(lx : int) returns (ly : int)
var turn : bool;
  ay0, ay1, ay : future int;
let
  turn = true fby (not turn);
  ay0 = async f(lx when turn);
  ay1 = async f(lx whenot turn);
  ay = merge turn (ay0) (ay1);
  ly = !(async 0 fby<<2>> ay);
tel
```

Appel de `temporal_fj_2<<sum>>(lx)` :



L'idée consiste à avoir deux instances de `f`, la première qui reçoit les éléments pairs du flot (les premiers éléments du tableau aplati) et la seconde les impairs. Pour laisser les deux instances calculer en parallèle, nous demandons le résultat avec un tableau de retard (deux instants).

4.2.2 *Remarque* Nous appliquons `merge` pour fusionner deux flots de futures. Dans ce cas précis, cela n'est pas nécessaire, car nous connaissons la taille des tableaux (donc la valeur de `turn`), ce qui permet de faire du retiming [LS91] en poussant les délais avant le `merge`. Le code est alors proche de celui de `slow_fast_a` :

```
node retimed_tfj2<<node f(int) = (int)>> (lx :int) returns (ly :int)
var turn :bool; y0, y1 :int;
let
  turn = true fby (false fby turn);
  y0 = !(async 0 fby async f(lx when turn));
  y1 = !(async 0 fby async f(lx whenot turn));
  ly = merge turn (y0) (y1);
tel
```

Dans un cadre général, nous ne connaissons pas `turn` et ne pouvons pas appliquer cette transformation. En particulier si la taille des tableaux varie, `turn` peut être une entrée du programme.

4.3 Parallélisme de données

La distinction entre le parallélisme de données et les autres parallélismes ne sont pas totalement définies et acceptées. Nous disons qu'il y a parallélisme de données quand un même calcul peut être exécuté indépendamment sur différentes parties d'une donnée. Dans un langage flot de données comme Heptagon, cela recouvre la possibilité de calculer les valeurs de différents indices d'un même flot en parallèle. Pour une équation `y = f(x)`, si `f` est une fonction *sans état*, le parallélisme de données consiste par exemple à calculer $f(x_1)$ et $f(x_2)$ en parallèle.

4.3.1 Fonctions combinatoires

Au niveau du code source, l'absence d'état de `f` permet de dupliquer son code et nous avons donc l'équivalence suivante pour tout `c` :

```
f(x)   ≡   merge c (f(x when c)) (f(x whenot c))
```

Le parallélisme de données s'écrit alors comme le fork-join temporel. Pour avoir trois threads, nous écrirons :

```
data_parallel3_m <<fun f(int)=(int)=>> (x : int) returns (y : int)
var t : int{<2};
let
  t = 0 fby 1 fby 2 fby t;
  ay = merge t (0 -> async f(x when (0=t)))
               (1 -> async f(x when (1=t)))
               (2 -> async f(x when (2=t)));
  y = !(async 0 fby<<3>> ay);
tel
```

Ceci n'est pas pratique et ne peut être écrit de manière paramétrique en Heptagon. Il faudrait pour cela avoir la récursion statique ou bien un `merge` avec un nombre de branches paramétriques. Nous proposons que `async` prenne un deuxième argument statique pour indiquer le nombre de threads qu'il doit utiliser. Ainsi pour avoir des queues de taille 1 et trois threads de calculs, nous écrivons `async<<1, 3>>` :

```
data_parallel3 <<fun f(int)=(int)=>> (x : int) returns (y : int)
let
  y = !(async 0 fby<<3>> async<<1, 3>> f(x));
tel
```

La création de plusieurs threads est déléguée à l'encapsulation.

4.3.2 L'encapsulation `async` pour les fonctions combinatoires

La compilation en Java d'une fonction produit une classe qui a pour seul contenu une fonction de transition statique :

```
class F {
  public static int step(int x) { /* ... */ }
}
```

L'encapsulation n'a donc pas à gérer d'instance. Par contre, si ses paramètres sont `<<N, T>>`, elle gère `T` threads et donc `T` files de taille `N` pour stocker les entrées. La répartition du travail en round-robin est assurée à chaque pas :

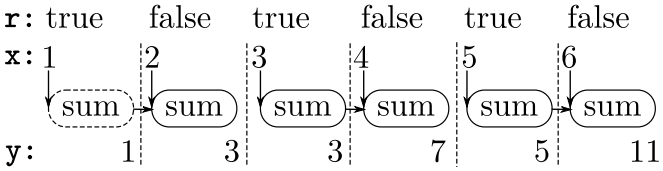
```
class Asyncfun_F {
  BoundedQueue q[];
  int currentQueue, nbQueue;
  Async_Slow(int N, int T) {
    q = new BoundedQueue[T];
    for (int i = 0; i < T; i++) {
      q[i] = new BoundedQueue(N);
      new Thread(){              //Run the ith worker thread
        public void run() {
          while(true) {          //Simulation loop
            (p, x) = q.pop();    //Pseudo code with tuples
            p.set(F.step(x));
      } } }.start();
    }
    nbQueue = T;
    currentQueue = 0;
  }
  Future<Float> step(float x) {
    Promise<Float> p = new Promise<Float>();
    q[currentQueue].push(p, x);        //Pseudo code with tuples
    currentQueue = (currentQueue + 1) % nbQueue;  //Round-robin
    return p;
  }
}
```

4.4 Parallélisme de données et réinitialisation

4.4.1 `async` avec gestion de la réinitialisation

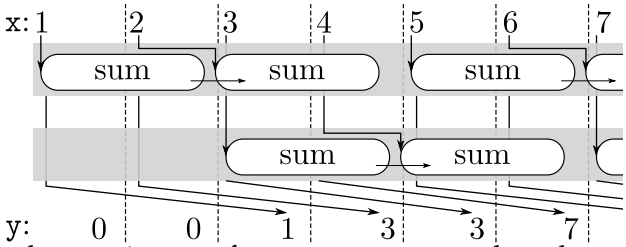
Le parallélisme de données se nourrit de l'indépendance entre des calculs d'un même flot. Cette indépendance existe entre toutes les activations de l'équation `y = f(x)` si `f` est sans état. Si `f` a un état, le $k$ième calcul dépend du $k-1$ième, *sauf* si l'équation est réinitialisée entre temps. L'exemple ci-dessous réinitialise l'appel à `sum` tous les instants impairs :

```
r = true fby (not r);
reset
  y = sum(x);
every r;
```



L'absence de dépendance entre le deuxième et le troisième instant est visible sur le chronogramme. Ainsi, le troisième appel à `sum` pourrait être effectué en parallèle des deux premiers. Tout comme pour le parallélisme de données avec une fonction sans état, il suffit pour en profiter d'ajouter assez de retard sur le résultat et d'indiquer que nous souhaitons utiliser plusieurs threads. Pour notre exemple, si nous voulons deux threads qui travaillent en parallèle, il faut au moins un retard de deux instants pour la sortie, de telle sorte que le deuxième reçoive une entrée sans attendre le premier. Pour la même raison, la file d'entrées de `async` doit pouvoir stocker deux entrées. Notez que le délai est ajouté en dehors de la réinitialisation, sinon le résultat serait toujours 0 :

```
r = true fby (not r);
reset
  ay = async<<1, 2>> sum(x);
every r;
  y = !(async 0 fby<<2>> ay);
```



Les deux premiers instants sont normaux : les entrées sont fournies au premier thread, qui renvoie deux futures. Au troisième instant, comme il y a une réinitialisation, nous passons les entrées à l'autre thread. Quant au cinquième instant, nous revenons au premier thread, qui a été réinitialisé. L'encapsulation se comporte donc similairement à la version sans état, sauf que la file courante n'est pas changée à chaque instant, mais à chaque réinitialisation.

Il reste un problème. Comment réinitialiser les instances encapsulées ? Cela ne peut se faire dans la méthode `reset` de l'encapsulation, car les instances en question n'ont peut être pas fini de calculer les précédents futures. Deux approches existent.

4.4.1.1 Version 1

La version la plus simple en Java est de ne pas se soucier de réinitialiser les précédentes instances. À chaque réinitialisation de l'encapsulation, nous créons une nouvelle instance. Pour que le thread de travail connaisse l'instance à utiliser, celle-ci est ajoutée à la file avec les entrées, comme nous le soulignons dans le code ci-dessous :

```
class Async_v1_Sum {
  Sum currentInstance;
  BoundedQueue q[];
  int currentQueue, nbQueue;
  Async_Slow(int N, int T) {
    currentInstance = new Sum();
```

```
    q = new BoundedQueue[T];
    for (int i = 0; i < T; i++) {
      q[i] = new BoundedQueue(N);
      new Thread(){                  //Run the ith worker thread
        public void run() {
          while(true) {              //Simulation loop
            (f, p, x) = q.pop();     //Pseudo code with tuples
            p.set(f.step(x));
      } } }.start();
    }
    nbQueue = T;
    currentQueue = 0;
  }
  Future<Float> step(float x) {
    Promise<Float> p = new Promise<Float>();
    q[currentQueue].push(currentInstance, p, x);  //Pseudo code with tuples
    return p;
  }
  void reset() {
    currentQueue = (currentQueue + 1) % nbQueue;  //Round-robin
    currentInstance = new Sum();                  //Reset the instance
  }
}
```

Les instances sont nettoyées par le ramasse-miettes une fois qu'elles ne sont plus utiles. Cette version est la plus proche de la sémantique du `reset` (cf. section 2.6.4), mais créer une nouvelle instance à chaque fois n'est pas nécessaire.

4.4.1.2 Version 2, avec réincarnation

Nous souhaitons ne pas créer dynamiquement des instances. Ainsi nous créons une instance par thread à leur création. Nous chargeons le thread de travail de réinitialiser son instance quand cela est nécessaire. Pour qu'il soit au courant, nous rajoutons un drapeau dans la file des entrées, comme souligné dans le code ci-dessous :

```
class Async_v2_Sum {
  bool needReset;
  BoundedQueue q[];
  int currentQueue, nbQueue;
  Async_Slow(int N, int T) {
    q = new BoundedQueue[T];
    for (int i = 0; i < T; i++) {
      q[i] = new BoundedQueue(N);
      new Thread(){                      //Run the ith worker thread
        public void run() {
          Sum sum = new Sum();
          while(true) {                  //Simulation loop
            (r, p, x) = q.pop();     //Pseudo code with tuples
            if (r) sum.reset();
            p.set(sum.step(x));
```

```
      } } }.start();
    }
    nbQueue = T;
    currentQueue = 0;
    needReset = false;
  }
  Future<Float> step(float x) {
    Promise<Float> p = new Promise<Float>();
    q[currentQueue].push(needReset, p, x);    //Pseudo code with tuples
    needReset = false;
    return p;
  }
  void reset() {
    currentQueue = (currentQueue + 1) % nbQueue;  //Round-robin
    needReset = true;                          //Reset the instance
  }
}
```

Cette version supprime les allocations dynamiques (sauf pour les futures) et la réinitialisation peut être effectuée à tout moment sans surcoût de synchronisation.

4.4.1 *Remarque* **Réinitialiser au plus tôt**
Nous réinitialisons au moment où nous revenons dans un thread de calcul. Cela est correct, mais si la réinitialisation prend beaucoup de temps, cela n'est pas optimal. En effet, la réinitialisation peut se faire en partant du thread. Au lieu d'envoyer le drapeau dans `step`, on envoie le drapeau dans `reset` avec une fausse entrée. Le thread de travail, quand il voit le drapeau, doit réinitialiser son instance et jeter l'entrée associée. Nous n'avons donc plus besoin de la variable `needReset`. Le désavantage est que cela augmente le nombre de valeurs communiquées au travers des files. Si le code est réinitialisé à chaque instant, il y aura deux fois plus de communications. Si la réinitialisation est hyperchrone, il peut y avoir plusieurs appels à `reset` entre deux appels à `step` aggravant encore cela.

4.4.1.3 Utilisation canonique du parallélisme de données

Tout comme pour le parallélisme de données avec une fonction sans état, ce sont les tableaux aplatis qui fournissent l'exemple le plus convaincant d'utilisation. Nous pouvons donner la version générique de l'exemple introductif de cette section (cf. section 4.4.1). Considérons que nous avons un flot de tableaux aplatis de taille `n`. Nous souhaitons appliquer le nœud `f` à chaque tableau indépendamment. Le code parallèle avec `m` threads s'écrit (avec `period` défini en section 1.5) :

```
node array_dp<<node f(int) = (int); n, m :int>> (lx :int) = (ly :int)
var new :bool;
let
  r = period<<n>>();
  reset
    ay = async<<n, m>> f(lx);
  every r;
  ly = ! (async 0 fby<<n*(m-1)>> ay);
tel
```

4.4.2 Programmer le parallélisme de données

Le parallélisme de données se programme sans avoir recours au deuxième argument de `async`. Cependant, tout comme dans le cas sans état (cf. section 4.3.1), le code que nous pouvons écrire n'est pas paramétrique et le nombre de threads doit être fixé.

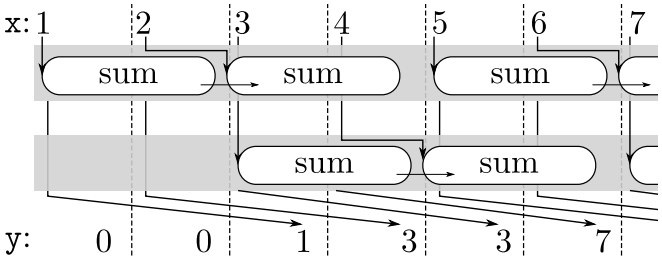
4.4.2.1 Version utilisant un automate

Une manière de programmer ce parallélisme est d'utiliser un automate. Ci-dessous, une version qui utilise deux threads, donc deux occurrences de `async f`.

L'automate commence dans l'état `A`. Il fournit l'entrée `x` à cette instance de `f`, jusqu'à ce que `r` soit vrai. Dans ce cas, il change d'état et c'est l'instance de l'état `B` qui va recevoir les entrées. Les transitions sont fortes. C'est-à-dire qu'elles se font à l'instant où `r` est vrai, et par réinitialisation (les mots clefs `unless` et `then`). En revenant dans l'état `A`, l'instance `f` est réinitialisée. Sur la droite nous donnons le chronogramme avec `r` et `x` définis comme précédemment. Pour permettre le parallélisme, le résultat est fourni avec un délai de deux instants :

```
node data_auto2<<node f(int)=(int)>>
(x : int; r : bool) = (y : int)
var ay : future int;
let
  automaton
    state A do
      ay = async<<2>> f(x)
    unless r then B
    state B do
      ay = async<<2>> f(x)
    unless r then A
  end;
  y = !(async 0 fby<<2>> ay);
tel
```

Un appel à `data_auto2<<sum>>(x, r)` :



4.4.2.2 Version avec les primitives du noyau

Les automates en eux-mêmes ne sont que du sucre syntaxique. Nous donnons une traduction du précédent automate. Pour simplifier la traduction, nous stockons l'état avec un booléen, `A` est représenté par true et `B` par false :

```
node data_autotrad2 <<node f(int)=(int)>> (x : int; r : bool) = (y : int)
var ayA, ayB, ay : future int; s : bool;
let
  s = if r then (false fby (not s)) else (true fby s);
  reset
    ayA = async f(x when s);
    ayB = async f(x whenot s);
  every r;
  ay = merge s ayA ayB;
  y = !(async 0 fby<<2>> ay);
tel
```

Nous retrouvons la structure du parallélisme sans état, mais le round-robin ne change d'état que si `r` est vrai. Ce changement est synchronisé avec la réinitialisation des instances.

4.5 Pipelining

4.5.1 Promise pipelining

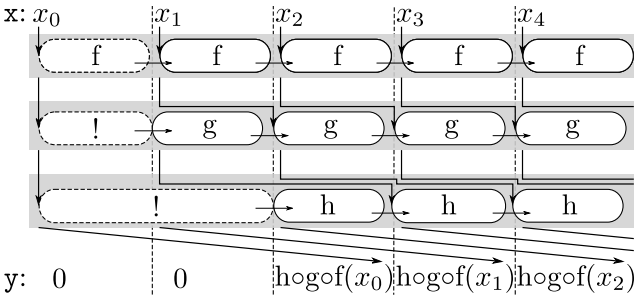
L'utilisation des futures pour le pipelining logiciel a été présenté par Liskov [LS88] sous le nom de *promise pipelining*. L'idée est d'effectuer la composition `g(f(x))` en lançant `f` de manière asynchrone ainsi que `g`, sans que l'appelant ne soit bloqué.

Ceci est une chose naturelle quand on a une vision monadique des futures. En effet, un des foncteurs à exhiber est `fmap` $:(a \to b) \to$ `future` $a \to$ `future` b. Grâce à lui, la composition `g` $\circ$ `f` peut se faire dans le monde des futures `(fmap g)` $\circ$ `(async f)` ce n'est que quand le résultat est strictement nécessaire, que nous récupérerons sa valeur en sortant de la monade avec `!`. Par exemple avant cela, nous pouvons composer avec `h` avec `(fmap h)` $\circ$ `(fmap g)` $\circ$ `(async f)`. Une version de `fmap` (non polymorphique avec l'ordre supérieur statique) se programme ainsi :

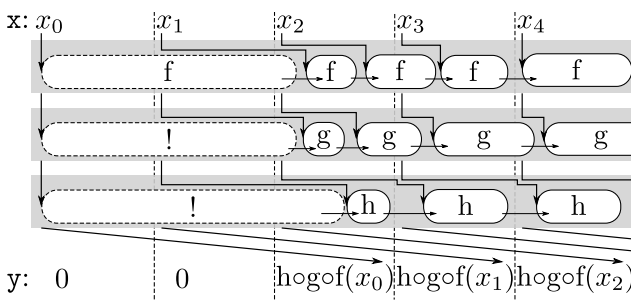
```
node fmap <<int n; node f(int) = (int)>> (i :future int)
returns (o : future int)
let
  o = async<<n>> f (!i);
tel
```

Notez que nous avons ajouté un paramètre statique `n` à `fmap`, pour contrôler la taille de la file d'entrée. Cela est nécessaire pour pouvoir programmer le pipelining en Heptagon. L'exemple ci-dessous effectue la composition `h` $\circ$ `g` $\circ$ `f`. Pour permettre l'exécution parallèle, il est nécessaire de ne pas demander le résultat avant d'avoir rempli les étages, ici au troisième instant. Ainsi, la file d'entrée doit être supérieure à sa profondeur dans le pipeline, de telle sorte à ce que la $k$ième activation de l'étage n puisse se faire à l'instant $k+n$ (les étage commencent à 0) :

```
node hgf(x :int) = (y :int)
var ax0, ax1, ax2 :future int;
let
  ax0 = async f (x);
  ax1 = fmap<<1, g>> (ax0);
  ax2 = fmap<<2, h>> (ax1);
  y = !(async 0 fby<<2>> ax2);
tel
```



4.5.1 *Remarque* **Découplage des étages**
Pour moyenner au maximum les fluctuations des calculs, *toutes* les files d'entrées doivent être de taille égale à la profondeur maximale du pipeline (ici 2). Ainsi nous pouvons avoir un comportement comme ci-contre où `f` prend beaucoup de retard. Nous le montrons au premier instant, mais cela peut arriver à tout moment, s'il avait pris de l'avance auparavant.



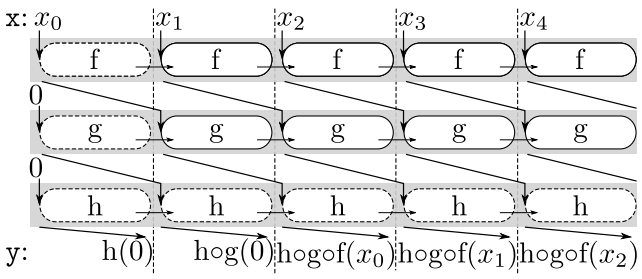
Ce pipelining est le plus souple. Cependant, il requiert une gestion mémoire évoluée, puisque les futures sont alloués par une tâche et doivent être désalloués dans une autre. En Java cela ne pose pas de problème, puisque nous avons choisi d'utiliser son ramasse-miettes.

Pour notre backend C++, nous verrons que l'échappement d'un future (passer un future en argument ou bien renvoyer un future) est problématique. Nous devons donc trouver un autre moyen de programmer le pipeline.

4.5.2 Pipelining synchronisé de fonctions sans état

Quand les fonctions du pipeline n'ont pas d'état, nous pouvons nous passer de l'échappement des futures en synchronisant les étages du pipeline. Nous devons toutefois décaler les calculs en insérant un délai entre chaque étage. Pour insérer ce délai, nous injectons une valeur inutile à tous les étages du pipeline. Ce pipeline synchronisé ne fonctionne bien que si les calculs sont de durées homogènes car chaque instant est synchronisé avec la fin des calculs de *tous* les étages :

```
node sync_fun_hgf(x :int) = (y :int)
var ax0, ax1, ax2 :future int;
let
  ax0 = async f(x);
  ax1 = async g(!(async 0 fby ax0));
  ax2 = async h(!(async 0 fby ax1));
  y = !ax2;
tel
```



4.5.2 *Remarque* Notez que le calcul de `h` pourrait ne pas être asynchrone puisque nous récupérons sa valeur dans le même instant. Cependant, l'écriture générique utilise le programme principal uniquement pour coordonner les calculs.

Nous pouvons construire le pipeline étage par étage avec la fonction `pipe_sw`. L'exemple ci-dessus s'écrit alors `pipe_sw<<h, pipe_sw<<g, f>> >>`.

```
node pipe_sw <<fun g(int)=(int); node f(int)=(int)>> (x :int) returns (y :int)
let  y = g(!(async 0 fby async f(x)));  tel
```

Notez que nous forçons `g` à être sans état (`fun`), mais pas `f` puisque nous lui fournissons l'entrée intacte.

Le besoin d'avoir `g` et `h` sans état est dû au fait que nous ajoutons une valeur d'initialisation sur leurs entrées. Dans le cas général, cette valeur d'initialisation peut changer l'état interne. Alors `y`, tronqué de ses deux premières valeurs, ne sera pas égal à `h(g(f(x)))` comme nous l'attendons. Pour que ce soit le cas, il faut que 0 soit une valeur neutre $[\![\mathtt{g}]\!]^{\mathcal{K}}_{H,\rho}(0.\mathbf{x}) = 0.[\![\mathtt{g}]\!]^{\mathcal{K}}_{H,\rho}(\mathbf{x})$. Ceci est le cas pour le nœud `sum`, car 0 est une entrée qui ne change pas son état interne. Utiliser une valeur neutre, si elle existe, est une bonne solution, mais cela n'est pas toujours possible.

4.5.3 Pipelining synchronisé de fonctions à état

4.5.3.1 Pipelining avec réinitialisation

Pour le pipelining de fonction à état, une solution est de réinitialiser `g` à l'instant correspondant à la première valeur du flot `f(x)`. Cette dernière arrive au deuxième instant puisque nous ajoutons une valeur d'initialisation. Le réinitialisation est donc faite avec un flot vrai uniquement au deuxième instant :

```
node pipe_r <<node f(int)=(int); fun g(int)=(int)>> (x :int) returns (y :int)
var t :int;
let
  t = !(async 0 fby async f(x));
  reset
    y = g(t);
  every (false fby true fby false);
tel
```

L'appel à `pipe_r<<h, pipe_r<<g, f>> >>` calcule comme voulu la composition de nœuds à états `h` ∘ `g` ∘ `f`, avec un délai de deux valeurs nulles.

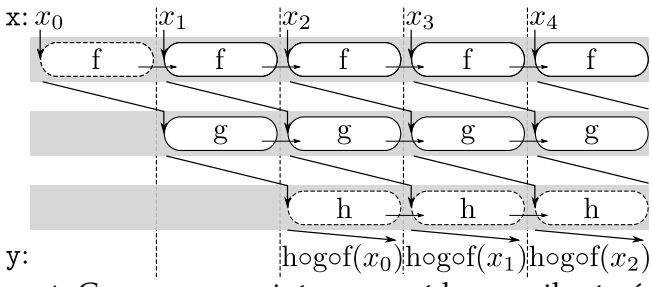
L'utilisation de la réinitialisation est élégante, mais peut être perçue comme contraire au monde flot de données.

4.5.3.2 Pipelining avec horloges

Toujours avec un pipeline dont les étages sont synchronisés, nous pouvons supprimer les calculs parasites en activant les étages les uns après les autres :

```
node hgf_ck(x : int) returns (c0 : bool; c1 : bool on c0; y : int on c0 on c1)
var ax0, ax1, ax2 : future int;
let
  ax0 = async f(x);
  c0 = false fby true;
  ax1 = async g(!((async 0 fby ax0) when c0));
  c1 = false fby true;
  ax2 =  async h(!((async 0 fby ax1) when c1)));
  y = ! ax2;
tel
```

L'entrée de `g` est échantillonnée par `c0`, qui est faux au premier instant. Ainsi, `g` ne reçoit pas la valeur d'initialisation de `ax0`. Similairement, nous retardons la sortie de `g` et supprimons la valeurs d'initialisation en échantillonnant avec `c1`. Notez que `c1` et `ax1` sont sur l'horloge • `on c0`.



Le résultat est sur l'horloge • `on c0 on c1`. Comme ces pointeaux sont locaux, il est nécessaire de les donner en sortie aussi, sinon le compilateur rejette le programme. Mais nous interdisons l'appel asynchrone si une horloge du nœud dépend d'une sortie (cf. section 3.3.2). À cause de cela, nous ne pouvons pas écrire une version générique comme `pipe_r` de la construction du pipeline avec des horloges.

4.6 De Lustre au monde impur

Dans les langages généralistes, l'utilisation des futures est souvent recommandée pour avoir un comportement dynamique, dépendant des temps d'exécution. Par exemple en Java, la méthode `get` peut prendre en argument un temps d'attente maximal, qui une fois dépassé soulève une exception pour sortir proprement de l'appel bloquant. Mais plus simplement, la méthode

`is_ready` des futures est publique et permet de programmer soi même des comportements dépendant des temps d'exécution.

Ajouter la fonction `is_ready` en Heptagon se fait comme pour toute fonction externe. Elle nécessite une ligne d'Heptagon (la déclaration : `val fun is_ready(future int) = (bool)`) et deux lignes pour l'implémentation en C++, de telle sorte que la fonction appelle la méthode `is_ready` du future. Cependant, `is_ready` n'a pas de sémantique dénotationnelle déterministe et cela revient à rajouter un oracle booléen.

Une fois ajoutée, de très nombreuses utilisations sont possibles, donnons deux exemples.

4.6.1 Nœud qui calcule le plus souvent possible

Un exemple représentatif du dynamisme apporté par cet oracle est le nœud `try_best`. Il va exécuter `f` en lui fournissant `x` dès que le dernier calcul a fini. Si le dernier calcul n'est pas fini, la valeur de `x` est jetée. Le résultat `y` correspond ainsi au calcul de `f` auquel on fournit une partie des valeurs de `x`. Pour faciliter la compréhension, la valeur de `x` correspondant à la sortie `y` est aussi renvoyée (`x_y`) :

```
node try_best<<node f(int)=(int)>> (x : int)
returns (c : bool; y : int on c)
var ay, curr_ay : future int; curr_x : int;
let
  ay = async f (x when c);
  c = is_ready(curr_ay);
  curr_ay = async 0 fby (merge c ay (curr_ay whenot c));
  curr_x = 0 fby (if c then x else curr_x);
  y = !(curr_ay when c);
tel
```

Quand `c` est vrai, nous lançons le calcul de `f` avec l'entrée `x` et récupérons le future `ay`. `c` n'est vrai que si le dernier calcul lancé (`curr_ay`) est fini. Le dernier calcul lancé est initialisé avec un faux calcul (`async 0`) qui est prêt au premier instant, puis il stocke la dernière valeur de `ay`. La sortie `y` est le résultat du dernier calcul. Ci-dessous un chronogramme possible quand le calcul de `f` est entre deux et trois fois plus long que le programme principal. Nous notons$<x>$ un future correspondant au calcul x, une fois réalisé, nous le notons ‘x‘. Dans le chronogramme, `f` prend deux instants pour calculer, puis trois, puis deux. Cette donnée n'est pas déterministe et elle correspond à notre choix arbitraire de valeurs pour l'oracle `c=is_ready(curr_ay)` :

`x`	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
`ay`	$<f(x_1)>$		$<f(x_3)>$			$<f(x_6)>$		$<f(x_8)>$
`curr_ay`	‘0‘	$<f(x_1)>$	‘$f(x_1)$‘	$<f(x_3)>$	$<f(x_3)>$	‘$f(x_3)$‘	$<f(x_6)>$	‘$f(x_6)$‘
`c`	true	false	true	false	false	true	false	true
`y`	0		$f(x_1)$			$f(x_3)$		$f(x_6)$

4.6.1 *Remarque* **Une entrée pointée par une sortie**
Au lieu d'oublier des valeurs de `x` quand nous ne sommes pas prêts, il serait souhaitable de ne recevoir une valeur qu'aux instants où `c` est vrai. Ceci n'est pas possible en Heptagon, car l'horloge de l'entrée `x` ne peut pas dépendre d'un pointeau donné en sortie. Cependant, le programmeur sait que `c` est vrai au premier instant et il est défini avec un registre, ainsi la *prochaîne* valeur de `c` est toujours connue et c'est elle qui pourrait être renvoyée et utilisée par l'appelant...

En l'état actuel, c'est à l'appelant de savoir que seules les entrées où `c` est vrai sont retenues.

4.6.2 Le ou-parallèle

Le calcul du « ou » parallèle (aussi dit paresseux) doit être en mesure de renvoyer true si l'un de ses arguments vaut true, même si l'autre n'est pas encore évalué. Le nœud `parallel_ou` que nous avons programmé ci-dessous calcule le ou parallèle du résultat de deux nœuds `f` et `g`. Moralement, nous voulons calculer le flot résultat de l'application point à point du ou sur les flots `f(x)` et `g(x)`. Cependant, l'entrée `x` peut arriver alors que nous n'avons pas fini de calculer. Comme pour `try_best` (cf. remarque 4.6.1) nous ne prenons en considération l'entrée `x` que si `c` est vrai. `c` est faux tant que nous n'avons pas fini de calculer la précédente sortie. À chaque fois que `c` est vrai, nous lançons l'évaluation de `f(x)` et de `g(x)` en parallèle. Nous gardons en mémoire les futures associés tant que leur état ne permet pas de connaître le *ou* de leur réalisation. L'état des futures est encodé avec trois constructeurs, `T` pour indiquer qu'il est réalisé et vaut true, `F` s'il vaut false et `U` s'il n'est pas encore réalisé, donc de valeur inconnue. Le calcul du ou parallèle scalaire est concluant si l'un des deux est à `T`, ou bien les deux sont à `F`.

```
type st = T | F | U

node parallel_ou << n : int; node f(bool)=(bool); node g(bool)=(bool) >>
  (x : bool) returns (c : bool; y : bool)
var a1, a2, curr_a1, curr_a2 : future bool;
    c1, c2 : bool; a1state, a2state : st;
let
  a1 = async<<n>> f(x when c);
  a2 = async<<n>> g(x when c);
  curr_a1 = async false fby (merge c a1 (curr_a1 whenot c));
  curr_a2 = async false fby (merge c a2 (curr_a2 whenot c));
  c1 = is_ready(curr_a1);
  c2 = is_ready(curr_a2);
  a1state = merge c1 (if !(curr_a1 when c1) then T else F) U;
  a2state = merge c2 (if !(curr_a2 when c2) then T else F) U;
  c = (a1state = T) or (a2state = T) or ((a1state = F) & (a2state = F));
  y = (a1state when c = T) or (a2state when c = T);
tel
```

4.6.2 *Remarque* Il est très important d'utiliser `merge c1` pour calculer `a1state`. Si nous écrivions `if c1 then (if !(curr_a1) then T else F) else U`, la branche du `then` serait évaluée à chaque fois et cela serait bloquant quand `c1` n'est pas prêt, alors que nous ne devons pas bloquer.

4.6.3 *Remarque* **Découplage borné**
La file des `async` est bornée par `n`, ce qui fait que notre *ou* n'est parallèle que tant que les deux nœuds sont découplés de moins de `n` calculs. En particulier, si `f` est très rapide et renvoie toujours true, alors que `g` est extrêmement lente, les `n+1` premiers instants vont renvoyer true aussi vite que `f` le peut, puis au `n+2` iéme instant la file de `g` sera pleine et il faudra attendre que `g` ait fini au moins son premier calcul pour pouvoir progresser.

Une exécution possible Par simplicité, nous prenons des fonctions ne dépendant pas de `x` :

$$\texttt{f(x)} = \text{true.false.false.true.}\cdots$$
$$\texttt{g(x)} = \text{false.true.false.false.}\cdots$$
$$\texttt{f(x) ou g(x)} = \text{true.true.false.true.}\cdots$$

Dans le chronogramme ci-dessous, nous notons<x>un future correspondant au calcul x, une fois prêt, nous le notons ‘x‘. Le résultat est donné avec un délai, la première valeur de la sortie est toujours fausse.

Nous considérons que `f` ne prend jamais plus d'un instant pour calculer. Ainsi, `curr_a1` contient toujours un future prêt. Par contre, nous considérons que `g` prend un peu moins de deux instants.

Au deuxième instant, seul le future `curr_a1` est prêt, mais il suffit puisqu'il vaut true. Le future de `curr_a2` (souligné dans le chronogramme) est alors écrasé dans le registre sans que sa valeur n'ait été réalisée. Au troisième instant, le future de `curr_a1` est prêt, mais est faux. Comme le future de `curr_a2` n'est pas prêt, nous ne pouvons décider de la sortie et `c` est faux. Au quatrième instant, `g` a fini de calculer. La sortie vaut true, etc.

`a1`	<true>	<false>		<false>		<true>	$\cdots$
`curr_a1`	‘false‘	‘true‘	‘false‘	‘false‘	‘false‘	‘false‘	‘true‘
`a1state`	F	T	F	F	F	F	T
`a2`	<false>	<true>		<false>		<false>	$\cdots$
`curr_a2`	‘false‘	<u><false></u>	<true>	‘true‘	<false>	‘false‘	<u><false></u>
`a2state`	F	U	U	T	U	F	U
`c`	true	true	false	true	false	true	true
`y`	false	true		true		false	true

4.7 Conclusion

Nous avons montré comment utiliser les registres et la file d'entrée de taille `N` de `async<<N>>` pour découpler un calcul. Quoi qu'il arrive, le découplage reste borné et notre programme est en mémoire bornée. Nous avons vu qu'échantillonner les futures revient à choisir quand et pour quels résultats il y a synchronisation. L'utilisation d'un tableau de futures permet le fork-join dans l'instant. En utilisant les instants synchrones, nous avons donné une version temporelle, où les instants sont utilisés pour séparer les créations de tâches et l'attente des résultats. La réinitialisation d'une fonction à état est mise à profit pour obtenir du parallélisme de données. Cependant, sa programmation requiert d'instancier autant de fois la fonction en question que nous voulons de calculs en parallèle. Pour faciliter son écriture, nous proposons que ce soit l'encapsulation `async<<N, T>>` qui gère `T` instances et threads. Nous étudions ensuite le pipelining. La version la plus souple en terme de programmation, connu sous le nom de *promise pipelining*, requiert de passer un future en argument d'un appel asynchone. Nous verrons dans le prochain chapitre que cela est interdit dans notre backend C++. Il est alors nécessaire d'utiliser une forme plus programmée du pipeline, comme la version synchrone, dont les étages sont synchrones et séparés par un registre. Il faut porter une attention toute particulière au cas où le code des étages a un état. Finalement, nous avons montré qu'en rendant accessible la méthode `is_ready` d'un future, nous pouvons programmer des comportements dynamiques prenant en compte l'exécution réelle. Cela nous a permis de programmer une version à mémoire bornée du ou-parallèle.

Chapitre 5

Implémentation des futures pour l'embarqué

Heptagon a pour vocation de générer du code utilisable dans les systèmes embarqués critiques. Une des restrictions usuelles est de ne pas avoir d'allocation dynamique de mémoire. Celle qui nous concerne vraiment est de pouvoir borner statiquement la quantité de mémoire nécessaire. Nous allons montrer que cela est possible même en présence de futures. Nous en profitons pour expliquer notre génération de code C++ et sa gestion de l'aspect dynamique des futures sans avoir recours à de l'allocation dynamique. Nous finissons par une discussion.

Notre implémentation vise principalement des calculateurs multi-cœurs en mémoire partagée. Nous nous reposons sur le modèle mémoire C++ 2011, ce qui assure une bonne portabilité.

5.1 Mémoire bornée

Sans les futures, la mémoire d'un programme Heptagon se divise en deux parties, les variables temporaires des fonctions de transition et les variables persistantes correspondant aux registres de chaque fonction. Typiquement, les premières seront allouées de manière transitoire sur la pile et les secondes seront allouées au début du programme sur le tas ou tout autre endroit persistant. Que ce soit pour les variables sur la pile ou sur le tas, nous connaissons statiquement les besoins du programme. Quand nous utilisons des futures, les promesses associées ont un comportement dynamique n'entrant dans aucune des deux catégories.

5.1.1 Durée de vie d'une promesse

Un future est une *référence* en lecture seule d'une promesse. Il est alloué comme toute autre variable du code source. Ce sont les promesses associées qui nécessitent une gestion dynamique des allocations, puisque leur durée de vie est dynamique.

Les trois exemples de la figure 5.1 Considérons le code `step_life`. À chaque instant, une promesse est générée par `async f()`. Deux références à cette promesse existent, une dans la tâche associée et l'autre, un future stocké dans `ax`. La durée de vie de cette promesse est incluse dans l'appel à la fonction de transition de `step_life`. En effet, dans le même instant où elle est créée, elle est aussi réalisée à cause de l'appel à `!ax`. La tâche associée étant terminée, la seule référence restante est le future `ax`. Finalement, ce dernier n'est pas stocké dans une mémoire de

```
node step_life()
returns (y : int)
var ax : future int;
let
  ax = async f();
  y = !ax;
tel
```

```
node keep_life(c : bool)
returns (y : int)
var arx : future int;
let
  arx = async f() fby arx;
  y = !(arx when c);
tel
```

```
node full_life(c : bool)
returns (y : int)
var arx : future int;
let
  arx = async 0 fby async f();
  y = merge c !arx 42;
tel
```

Figure 5.1: Des exemples illustrant la durée de vie dynamique d'une promesse

`step_life` et est donc jeté avec la pile d'exécution de la fonction de transition, ce qui supprime la dernière référence.

Considérons le code `keep_life`. Le tout premier future généré par `async f()` est restocké à tous les instants dans le registre `arx`. La promesse associée a donc une référence jusqu'à la fin de l'exécution du programme.

Considérons le code `full_life`. Le registre `arx` est sur l'horloge • `on c`. Si `c` est vrai, la valeur stockée dans le registre `arx` est mise à jour, dans le cas contraire, le registre est intouché. Si `c` est toujours faux, alors `arx` contiendra toujours le future créé initialement par `async 0`, sa durée de vie sera égale à l'exécution du programme. À l'opposé, si `c` est toujours vrai, toutes les promesses générées sont en vie à cheval sur deux instants : celui qui les produit et le suivant où le future associé est réalisé et enlevé du registre.

Finissons les exemples en soulignant qu'il est possible d'avoir plusieurs registres référençant la même promesse. S'il est en plus possible de renvoyer ou de passer un future en argument, alors ces références peuvent être dans des threads et des registres de nœuds différents.

5.1.1 *Nota bene* **Durée de vie dynamique**
En règle générale, la durée de vie des promesses n'est pas connue statiquement. Cela est bien illustré avec `full_life`, puisque la durée de vie dépend de l'entrée `c`.

5.1.2 Mémoire bornée

5.1.2 *Propriété*
Une promesse est morte une fois réalisée et sans future la référençant.

Ceci nous permet d'affirmer que le nombre de promesses en vie simultanément est borné par le nombre de futures plus le nombre de promesses en attente d'être réalisées. Ces deux nombres sont eux-mêmes statiquement bornés. Les futures sont des variables comme les autres, leur nombre sur la pile et sur le tas sont connus. Les promesses en attente n'existent que dans les queues d'entrées des `async` et ces queues sont statiquement bornées.

5.2 Génération de C++ sans allocation dynamique

Maintenant que nous savons statiquement borner la quantité de mémoire nécessaire, nous pouvons nous passer d'allocation dynamique. Nous allons allouer statiquement la mémoire et gérer son utilisation dynamiquement. Puisque les durées de vie sont dynamiques, nous avons besoin de connaître les promesses référencées.

Une solution serait d'utiliser un ramasse-miettes concurrent, comme nous faisons en Java. Cependant, le principe même de ramasse-miettes concurrent est discutable dans un système embarqué. Il travaille sur toute la mémoire alors que des threads non synchronisés y accèdent, ceci engendre un surcoût de synchronisations non négligeable et difficilement prédictible. Des tentatives pour faire en sorte que le ramasse-miettes soit prédictible et acceptable dans un contexte critique existent, comme Schism [Piz+10b] ou Metronome [BCR03].

Notre choix est de restreindre l'utilisation des futures pour avoir une gestion de la mémoire, locale à chaque thread.

5.2.1 Restriction de l'échappement

Pour notre génération C++, nous empêchons les futures d'échapper des nœuds. De cette manière, tous les futures d'une promesse sont dans la mémoire (le tas ou la pile) d'un seul nœud. Notez que les files des encapsulations des appels asynchrones sont aussi dans le nœud appelant. Ainsi, ces futures sont utilisés uniquement par le nœud appelant. Ceci nous soulage de la partie concurrente de la gestion mémoire, puisque le code d'un nœud est exécuté séquentiellement. De plus le code s'exécute de manière synchrone, la fin ou le début des instants sont des moments privilégiés pour effectuer une passe de gestion des promesses du nœud.

5.2.2 Le stock

Pour gérer ses promesses, un nœud va faire appel à ce que nous appelons un stock. Il y a un stock par type de promesse. Un stock contient un nombre fixe de promesses. Il est en charge de suivre les références (les futures) de ses promesses, pour être en mesure de fournir des promesses fraîches et recycler celles qui meurent. Un stock n'est pas un ramasse-miettes parcourant la mémoire pour trouver ce qui est mort. Il implémente un comptage de références, optimisé par la propriété suivante :

5.2.1 *Propriété* **Promesses en vie entre les instants**
Entre deux appels de la fonction transition, seules les promesses qui ont un future stocké dans un registre sont référencées.

Comme un stock est restreint à la mémoire d'un nœud, il n'est pas nécessaire de suivre très précisément l'évolution des futures. Nous nous contentons d'une passe de recyclage entre chaque appel à la fonction de transition. Les seules références à compter sont donc celles mémorisées dans les registres.

5.2.2.1 Interface

L'interface minimaliste d'un stock est composé de quatre méthodes. La méthode `get_free()` permet de récupérer une promesse fraîche pour un appel asynchrone. La méthode `store(f)` marque la promesse associée au future `f` comme étant stockée dans un registre. La méthode `unstore(f)` marque la promesse associée comme étant enlevée d'un registre. Finalement, la méthode `tick` marque la fin/début de l'instant et déclenche la passe de recyclage.

```
template <typename T, int size>
class stock {
public:
  promise<T>* get_free();
  void store(future<T>);
  void unstore(future<T>);
```

```
  void tick();
  /* ... */
}
```

Le bon fonctionnement d'un stock repose sur l'utilisation systématique de ces méthodes. C'est à la génération du nœud que nous instrumenterons le code correctement. Pour obtenir une promesse fraîche nous utiliserons `get_free`, tout future stocké (resp. enlevé) dans un registre sera déclaré comme tel avec `store` (resp. `unstore`) et `tick` devra être appelé entre les appels à la fonction de transition.

5.2.2.2 Les états d'une promesse

L'état d'une promesse dans un stock a trois composantes :
— Son drapeau `ready` indique si elle n'est pas en attente de réalisation.
— Une couleur, blanc si elle n'est pas référencée, noir si elle est référencée par un registre et sinon gris.
— Un compteur de références, utilisé quand elle est noire, pour compter le nombre de registres la référençant.

Ceci nous permet de définir trois états remarquables pour une promesse.

5.2.2 *Propriété* **Invariant**
Une promesse est :
— morte ssi elle est blanche et `is_ready()` est vrai (cf. propriété 5.1.2).
— stockée dans n registres ssi elle est noire et son compteur vaut n.
— sinon, grise.

Nous disons de plus qu'elle est vivante ssi elle n'est pas morte.

5.2.3 *Propriété* **Mise à jour du drapeau `ready`**
Au début du programme, le drapeau `ready` de toutes les promesses doit être vrai. Seule la méthode `get_free` le touche en réinitialisant la promesse avant de la renvoyer, auquel cas `is_ready()` devient faux. Il ne redeviendra vrai qu'une fois la promesse réalisée.

Au début du programme, toutes les promesses du stock sont mortes. L'invariant est alors préservé par les méthodes du stock, qui mettent à jour l'état des promesses :
— `get_free` cherche dans le stock une promesse morte. En la retournant, elle est grisée.
— `store` colorie en noir la promesse associée à son argument et initialise son compteur de références à 1. Si elle était déjà noire, le compteur est simplement incrémenté.
— `unstore` décrémente le compteur de références. S'il atteint 0, la promesse est grisée.
— `tick` blanchit toutes les promesses grises.

5.2.4 *Remarque* **Le code d'un stock n'est pas concurrent**
Techniquement, les promesses d'un stock sont accédées par deux threads concurrents, celui du stock et celui de la tâche associée. Il n'empêche que le code du stock n'est pas concurrent. La seule interaction avec la tâche associée est au travers de la méthode `is_ready` de la promesse, pour savoir si elle est en attente de réalisation. C'est donc la promesse qui est en charge des problèmes de synchronisation. De cette manière, l'interface n'est accédée que par un seul thread et n'a pas besoin de protections spéciales.

5.2.3 Instrumentation du code généré

Pour ceux qui préfèrent immédiatement avoir un exemple de code généré, nous reprenons l'exemple qui avait motivé notre démarche en section 5.2.4.

5.2.3.1 Encapsulation `async`

Nous avons présenté en section 4.4.1.2 l'encapsulation Java d'un nœud `f` dans un nouveau `Async_f`. En C++, nous utilisons la version 2, avec réincarnation des nœuds. La seule allocation dynamique qui restait était celle des promesses. Dans le code Java, à chaque appel, `step` alloue une nouvelle promesse. Ici, la promesse sera fournie en argument de `step`, puisque c'est l'appelant qui gère ses promesses.

5.2.3.2 Traduction vers Obc

Pour présenter la traduction des appels asynchrones, nous reprenons les fonctions de traduction du noyau vers Obc, présentées en section 2.5.3. L'instrumentation va toucher l'initialisation des registres et leur mise à jour. La fonction de traduction est inchangée pour les équations, qu'elles manipulent des futures ou pas. Seuls les nouveaux cas sont à considérer :

$$
\begin{aligned}
(\!|y =_z^{ck} \texttt{!}x|\!) &= C(ck, y \texttt{ := } x\texttt{.get();}) \\
(\!|y =_z^{ck} \texttt{async } i|\!) &= C(ck, y \texttt{ := } stock_{t_y}\texttt{.get\_free(); } y\texttt{.set(}i\texttt{);}) \\
(\!|(y_1, \cdots, y_n) =_z^{ck} \texttt{async} f\texttt{(}x_1, \cdots, x_n\texttt{)}|\!) &= C\Big(ck, y_1 \texttt{ := } stock_{t_{y_1}}\texttt{.get\_free();}\Big) \cdots \\
&\quad C(ck, y_n \texttt{ := } stock_{t_{y_n}}\texttt{.get\_free();}) \\
&\quad C(ck, o_z\texttt{.step(}x_1, \cdots, x_n, y_1, \cdots, y_n\texttt{);})
\end{aligned}
$$

Les stocks utilisés sont membres de la machine. Rappelons et complétons (la partie soulignée) la fonction de traduction de définition d'un nœud :

$$
\begin{aligned}
&\mathrm{def}(f\texttt{(}x_1\texttt{, }\cdots\texttt{, }x_n\texttt{) = (}y_1\texttt{, }\cdots\texttt{, }y_m\texttt{) with } eqs) = \\
&\quad \texttt{machine } f \texttt{ \{} \\
&\qquad \texttt{memory } \mathrm{getRegs}(eqs)\texttt{; instances } \mathrm{getObjs}(eqs)\texttt{; stocks } \underline{\mathrm{getStocks}(eqs)} \\
&\qquad \texttt{reset()\{ } \mathrm{getRegsInit}(eqs)\texttt{; } \mathrm{getObjsInit}(eqs)\texttt{; \}} \\
&\qquad \texttt{(}y_1 : t_{y_1}\texttt{, }\cdots\texttt{, }y_m : t_{y_m}\texttt{) step(}x_1 : t_{x_1}\texttt{, }\cdots\texttt{, }x_n : t_{x_1}\texttt{)\{} \\
&\qquad\quad \texttt{vars } \mathrm{getVars}(eqs) \\
&\qquad\quad \underline{\mathrm{stocksTick}(eqs)} \\
&\qquad\quad (\!|eqs|\!) \\
&\qquad\quad \mathrm{getRegsValue}(eqs) \\
&\qquad \texttt{\}} \\
&\quad \texttt{\}}
\end{aligned}
$$

Un stock ne gère qu'un type de promesse. Nous notons $Ft(eqs)$ l'ensemble des types de futures utilisés par le programme. Au début de l'instant, nous devons appeler la méthode `tick` de tous les stocks :

$$\mathrm{stocksTick}(eqs) \;=\; [stock_t\texttt{.tick();}]_{t \in Ft(eqs)}$$

5.2.5 *Propriété* **Taille d'un stock**
La taille nécessaire d'un stock gérant des promesses de type t est égale à la somme des registres de type future de t, plus le nombre de futures constants (`async` i) et augmenté de $(n+1) \times m \times c + c$ pour chaque appel asynchrone `async<<`n`, `m`>> f` avec c le nombre de sorties de `f` du type t.

Démonstration Les promesses en vie sont soit noires, soit grises, soit blanches avec `is_ready()` faux.

Elles sont noires ssi elles sont stockées dans un registre (cf. propriété 5.2.2).

Au début de l'instant toutes les promesses grises sont mises à blanc par `tick()`. Pour être grise, une promesse doit donc provenir d'un appel à `get_free` de l'instant. Ce nombre est borné par le nombre de futur constants plus la somme des c de chaque appel asynchrone.

Pour être blanche en vie, une promesse doit être passée par `get_free` (cf. propriété 5.2.3). Il n'y a que deux possibilités pour passer par `get_free`. 1) Venir de la traduction d'un future constant `async` i, mais juste après, la méthode `set` de la promesse est appelée, ce qui remet `is_ready()` à vrai. Dans ce laps de temps la promesse était grise. 2) Finalement, une promesse blanche en vie a forcément été donnée en argument à la fonction de transition d'une encapsulation asynchrone. Dans l'encapsulation, la promesse est poussée dans une des m files de taille n. Quand elle sort de la file, elle est encore avec `is_ready()` faux, tant que le calcul du thread de travail n'est pas fini. Nous avons donc au maximum $(n+1) \times m \times c$ promesses blanches en vie en même temps.

Les deux arguments statiques du stock sont, en premier le type de ses promesses, en second sa taille :

$$\begin{aligned}
\text{getStocks}(eqs) &= [\texttt{stock<}t, ssize(eqs)(0)\texttt{>}stock_t\texttt{;}]_{t \in Ft(eqs)} \\
ssize_t(y \texttt{ =}^{ck}_z \texttt{ async } i)(acc) &= acc + 1 \quad \text{si } t \text{ est le type de } y \\
ssize_t(y \texttt{ =}^{ck}_z \texttt{ } i \texttt{ fby } x)(acc) &= acc + 1 \quad \text{si } t \text{ est le type de } y \\
ssize_t(y \texttt{ =}^{ck}_z \texttt{ async«}n\texttt{, } m\texttt{» } f\texttt{(}\cdots\texttt{)})(acc) &= acc + (n+1) \times m + 1 \quad \text{si } t \text{ est le type de } y
\end{aligned}$$

Rappelons aussi les fonctions intermédiaires et ajoutons les cas pour les futures, marqués d'un astérisque. Pour aider l'écriture, nous utilisons le raccourci $\text{storeInReg}(a, r)$ qui écrit le future a dans le registre r :

$$\text{storeInReg}(a, r) = \texttt{unstore(}r\texttt{); } r \texttt{ := } a\texttt{; store(}a\texttt{);}$$

$$\begin{aligned}
\text{getRegs}(y \texttt{ =}^{ck}_z \texttt{ } i \texttt{ fby } x)(acc) &= y : t_y\texttt{; } acc \\
\text{getRegsInit}(y \texttt{ =}^{ck}_z \texttt{ async } i \texttt{ fby } x)(acc) &= \left|\begin{array}{l} \text{storeInReg}(stock_{t_y}\texttt{.get_free(), this.}\ y\texttt{);} (*) \\ \texttt{this.}\ y\texttt{.set(}i\texttt{);} acc \end{array}\right. \\
\text{getRegsInit}(y \texttt{ =}^{ck}_z \texttt{ } i \texttt{ fby } x)(acc) &= \texttt{this.}\ y \texttt{ := } i\texttt{; } acc \\
\text{getRegsValue}(y \texttt{ =}^{ck}_z \texttt{ async } i \texttt{ fby } x)(acc) &= C(ck, \text{storeInReg}(x, \texttt{this.}\ y))\texttt{; } acc \quad (*) \\
\text{getRegsValue}(y \texttt{ =}^{ck}_z \texttt{ } i \texttt{ fby } x)(acc) &= C(ck, \texttt{this.}\ y \texttt{ := } x)\texttt{; } acc \\
\text{getObjs}(\cdots \texttt{ =}^{ck}_z \texttt{ } f\texttt{(}\cdots\texttt{)})(acc) &= o_z : f\texttt{; } acc \\
\text{getObjs}(\cdots \texttt{ =}^{ck}_z \texttt{ async } f\texttt{(}\cdots\texttt{)})(acc) &= o_z : Async_f\texttt{; } acc \quad (*)
\end{aligned}$$

$$\begin{aligned}
\text{getObjsInit}(\cdots \texttt{=}^{ck}_{z}\ f\texttt{(}\cdots\texttt{)})(acc) &= o_z\texttt{.reset(); } acc \\
\text{getObjsInit}(\cdots \texttt{=}^{ck}_{z}\ \texttt{async}\ f\texttt{(}\cdots\texttt{)})(acc) &= o_z\texttt{.reset(); } acc \qquad (*) \\
\text{getVars}(\texttt{(}y_1\texttt{,}\cdots\texttt{,}y_m\texttt{)} \texttt{=}^{ck}_{z}\ \ldots)(acc) &= y_1\ \texttt{:}\ t_{y_1}\texttt{; }\cdots\ y_m\ \texttt{:}\ t_{y_m}\texttt{; } acc
\end{aligned}$$

5.2.4 L'exemple de `slow_fast_a`

Nous présentions en section 3.2.3.2 le code Java de notre exemple `slow_fast_a`. Le code Obc (du pseudo C++) est presque identique, seule la présence du stock et son utilisation sont nouvelles. Les variables intermédiaires `tmp` et `tmp2` viennent de la traduction vers le noyau.

```
machine Slow_fast_a {
  registers yf : float; ys : future float;                (* getRegs *)
  objects countdown : Countdown; fast : Fast; slow : Async_Slow; (* getObjs *)
  stocks stock_float<<2>>;                                (* getStocks *)
  reset() {
    this.yf := 0.0;                                       (* getRegsInit *)
    stock_float.unstore(this.ys);
    this.ys := stock_float.get_free();
    stock_float.store(this.ys);
    this.ys.set(0.0);
    countdown.reset();                                    (* getObjsInit *)
    slow.reset();
    fast.reset();
  }
  (y : float) step (big_step : int) {
  vars yf, tmp2 : float; ys, tmp : future float;               (* getVars *)
    stock_float.tick();                                   (* stocksTick *)
    yf := this.yf;                                        (* T *)
    big := countdown.step(big_step);
    if (big) {
      ys := this.ys;
      y := ys.get();
      tmp := slow.step(big_step, ys.get(), stock_float.get_free());
    } else {
      y := yf;
    }
    tmp2 := fast.step(1.0, y);
    this.yf := tmp2;                                      (* getRegsValue *)
    if (big) {
      unstore(this.ys);
      this.ys := tmp;
      store(this.ys);
    }
  }
}
```

5.2.5 Implémentation du stock

Le principal travail du stock est d'allouer statiquement les promesses et de les garder dans un état cohérent (cf. section 5.2.2.2). Pour cela, les promesses sont mises dans trois chaînes exclusives, une pour les promesses blanches, une pour les grises et une pour les noires. Pour former ces chaînes, une promesse est mise dans une structure (`box`) avec trois autres champs. Un qui va servir de compteur de référence, un la liant à un prédécesseur et un successeur. Les liens entre cases nous permettent de gérer les couleurs comme des listes doublement chainées. Seule la tête des listes est gardée (`whites`, `grays` et `blacks`). La mémoire du stock est le tableau de cases `all`. Le point important est qu'à tout moment, toutes les cases appartiennent à une et une seule liste. Le stock est initialisé avec toutes les cases chainés dans la liste `whites`, les deux autres listes sont vides (pointeur nul).

Toutes les fonctionnalités vont alors reposer sur la méthode `move` qui permet de déplacer un nœud d'une liste à une autre. L'écriture `move(whites, x, grays)` déplace `x` des blancs dans les gris. Les têtes de files sont prises par référence car elles peuvent être changées par l'opération. Une version optimisée `move_all` est fournie pour déplacer toute une liste dans une autre :

```
template <typename T, int size>
class stock {
  struct box {promise<T> v; box * prev; box * next; int ref_count;};
  typedef box* n;
  box all[size];                    // storage
  n whites, grays, blacks;          // lists heads
  void move(n &from, n x, n &dest); // move x from from to dest
  void move_all(n &from, n &dest);  // move all from to end of dest
public:
  stock(); //chain together all nodes in whites
  /* ... */
}
```

5.2.6 *Remarque* La promesse est en tête de `box`, de telle sorte à pouvoir caster `box*`, `promise<T> *` et `future<T>` les uns en les autres, tous trois stockant la même adresse.

La méthode `tick` est alors triviale :

```
void tick() { move_all(grays, whites); }
```

La recherche d'une promesse morte est un simple parcours de la liste blanche pour en trouver une qui n'est pas en attente de réalisation. La taille du stock a été fixée pour qu'à tout moment, il y ait assez de promesses mortes. Si la communication du drapeau `ready` est instantanée entre les threads, nous savons que notre boucle ne fera pas plus qu'un parcours de la liste blanche. C'est pour cela que si nous avons fait au moins un tour de la liste, nous appelons `yield` pour laisser une chance à la communication et ne pas bloquer inutilement les autres threads :

```
promise<T>* get_free() {
  n current = whites;
  int i = 0;
  while(!current->v.is_ready()) {
    current = current->next;
    if (i++ == size) this_thread::yield();
  }
```

```
  move(whites, current, grays);
  current.reset();
  return reinterpret_cast<promise<T>*>(current);
}
```

Notez qu'avant de rendre la promesse, nous la réinitialisons avec sa méthode `reset`. Celle-ci permet de réinitialiser le drapeau à false :

```
void reset() { ready.store(false, memory_order_release); }
```

5.3 Discussion

Nous avons prouvé que l'utilisation des futures ne change pas la garantie d'Heptagon d'avoir une mémoire bornée statiquement. À notre connaissance, il n'y a pas de ramasse-miettes concurrent qui soit utilisé dans l'embarqué. Ils en existent pourtant des spécialisés, comme dans Fiji [Piz+10a], une version de la machine virtuelle Java pour les systèmes embarqués. Des expériences approfondies concernant l'impact d'un ramasse-miettes concurrent pour notre cas précis sont nécessaires pour conclure.

Dans tous les cas, notre expérience est que la gestion avec des stocks en C++ ajoute une charge négligeable. De plus, outre le pipelining de promesses (cf. section 4.5.1), nous n'avons pas rencontré d'exemples nécessitant l'échappement de portée d'un future. En particulier, les cas profitant de la possibilité de retourner un future sont alambiqués. En effet la plupart du temps, au lieu de retourner un future, on peut demander à ce que l'appelant appelle de manière asynchrone la fonction qui voulait retourner un future.

5.3.1 Gains dus à l'allocation dynamique

Ce n'était pas le but originel, mais la gestion dynamique de la mémoire, même locale à un nœud, est une aubaine pour minimiser la mémoire nécessaire et les copies.

En faisant abstraction du drapeau `ready`, un future est une indirection vers de la mémoire allouée : une référence. `get_free` est la méthode d'allocation. Les stocks correspondent à des gestionnaires de mémoire ne contenant qu'un seul type d'objet, similairement aux caches d'allocations SLAB/SLUB du noyau linux [BA01]. L'instrumentation que nous ajoutons pour les futures les transforme en références intelligentes, avec une gestion automatique de la désallocation.

Quand on manipule des tableaux ou toute autre structure de donnée importante, il apparait des copies inutiles coûteuses. Certaines peuvent être supprimées statiquement, ce que nous faisons en Heptagon avec memalloc (cf. section 2.5.5). Mais il y en a aussi que nous ne pouvons supprimer statiquement. Prenons l'exemple minimaliste `gc_one`. Nous omettons la déclaration de la constante entière `n` et de la fonction `f` qui prend un entier et renvoie un tableau de `n` entiers. Sur la droite le code de la méthode `step` de `gc_one` en C++ :

```
node gc_one (x : int) = (o : int)
var y, z : int^n; c : bool;
let
  y = f(x);
  c = (y[0] = 0);
  z = 0^n fby (y when c);
  o = merge c z[0] 42
tel
```

```
void step(int x, int* o) {
  array<int,10> y;
  f.step(x, &y);
  bool c = (y[0]==0);
  if (c) {
    (*o) = this->z[0];
    for (int i = 0; i < 10; i++)
      this->z[i] = y[i];
  }
  else (*o) = 42;
}
```

L'idée de ce code est de ne pas savoir au moment de l'appel à `f` si son résultat va être gardé dans le registre `z` ou pas. Cette décision dépend du pointeau `c` qui lui-même dépend de la sortie de `f`. Quelles que soient les optimisations ou les annotations, comme le registre `z` est une adresse mémoire fixée, chaque fois que `c` est vrai, il faudra copier `y` dedans.

Si par contre le registre `z` ne contient pas un tableau, mais simplement une référence vers un tableau, la copie ne concernera que la référence. Une utilisation détournée des futures permet cela. Seul le future `y` est copié dans le registre et non la promesse qui contient le tableau :

```
node gc_one_a (x : int) = (o : int)
var y, z : future (int^n); c : bool;
let
  y = async<<1,0>> f(x);
  c = (!y)[0] = 0;
  z = async (0^n) fby (y when c);
  o = merge c (!z)[0] 42;
tel
```

```
void step(int x, int* o) {
  this->stock_int_n.tick();
  future<array<int,n>> y;
  y = this->stock_int_n.get_free();
  f.step(x, &y);
  bool c = (y.get()[0]==0);
  if (c) {
    (*o) = this->z.get()[0];
    this->stock_int_n.unstore(this->z);
    this->z = y;
    this->stock_int_n.store(this->z);
  } else {
    (*o) = 42;
  }
}
```

L'appel `async<<1, 0>> f(x)` demande l'utilisation de 0 thread. En effet, nous ne sommes pas intéressé ici par le calcul asynchrone, mais simplement l'utilisation d'un future comme indirection. Le compilateur reconnait cela et spécialise l'encapsulation pour ce cas. Ainsi, l'appel à `f` est fait de manière synchrone. L'encapsulation ne fait que mettre son résultat dans la promesse qui lui est passée.

5.3.2 Performances

Nous donnons dans cette partie des chiffres à prendre avec beaucoup de précautions. Ils sont calculés sur un macbookpro 9,2 i7 deux cœurs à 2,9Ghz, avec le compilateur G++ 4.7.2 et les optimisations O2. Le passage en O3 améliore sensiblement certains cas, mais les résultats sont très difficilement interprétables. Par exemple un algorithme linéaire en un paramètre devient linéaire par morceaux avec des sauts (plus ou moins francs) suivant le paramètre, etc. De nombreux facteurs et détails non évoqués provoquent des changement qui peuvent être

drastiques sans raison apparente. Notez que nous avons effectué de nombreuses petites optimisations dans la génération de code et dans les librairies, sans les détailler ici. Ces optimisations ont pour but de calculer en place le plus possible.

Nous nous contenterons de considérer les ordres de grandeur pour confirmer le bon fonctionnement de nos outils.

5.3.2.1 L'exemple `gc_one`

Nous voulons mesurer le surcoût des stocks, puis celui de l'encapsulation asynchrone. Pour cela, nous codons à la main `f` en C++, de telle sorte qu'elle prenne un temps pratiquement nul, tout en retournant un tableau dont la première case est à 0. Ainsi `c` est toujours vrai.

Sans thread, uniquement le stock Avec l'appel asynchrone `async<<1, 0>> f(x)`, l'appel à `f` est fait de manière synchrone. Ceci nous permet de ne mesurer que l'impact de la gestion du stock et non les communications entre processeurs.

Nous comparons les performances de `gc_one` et `gc_one_a`. Ce deuxième est aussi rapide que le premier à partir de `n` égal à 24 (ce qui correspond à avoir une fonction de transition qui prend 17,5 nanosecondes). Comme attendu, le temps pris par `gc_one_a` ne dépend pas de `n`, tandis que celui de `gc_one` croit proportionnellement.

Un thread plus le stock Si nous mettons un thread (`async<<1,1>> f(x)`), nous mesurons en plus le coût de l'encapsulation et de la communications entre deux cœurs. Dans ce cas, un appel à la fonction de transition prend 324 nanosecondes (pour couvrir ces coûts, nous avons besoin de `n` égal à 530). Ceci est de l'ordre de grandeur attendu puisque la latence entre cœurs est d'environ 100 nanosecondes, donc 200 plus le coût du stock de 18, restent environ 105 nanosecondes de plus qui proviennent de la gestion de la queue par l'encapsulation asynchrone et de l'attente sur les futures.

En étudiant de près l'exécution, nous remarquons que nous passons un temps important dans les appels à la méthode `get` des futures.

Meilleures performances avec une attente active Nous avons présenté l'attente effectuée par `get` comme une boucle `while` dont le corps est un appel à `yield()` (cf. section 3.1.3). Cet appel a un but collaboratif. Si le future n'est pas prêt, nous indiquons à l'ordonnanceur système que le thread peut être mis de côté. Cette stratégie n'est jamais mauvaise, mais rarement la meilleure. Pour `gc_one_a` avec un thread, le code qui donne le meilleur résultat est une boucle vide de plus de 300 tours. Ce qui est certain c'est qu'en dessous de 250 les performances sont significativement moins bonnes :

```
T& get() {
  int i = 0;
  while (is_not_ready()) {
    i++;
    if(i > 300) {
      i = 0;
      this_thread::yield();
    }
  }
}
```

Avec ce code, nous obtenons 269 nanosecondes pour une transition de `gc_one_a` avec un thread. Le surcout imputable aux futures et à l'encapsulation est alors d'environ 50 nanosecondes.

Par contre, si nous mettons plus de threads que de cœurs, cette attente active fait s'effondrer brutalement les performances.

5.3.2.2 L'exemple `slow_fast`

Pour l'exemple `slow_fast`, les paramètres en jeu sont la quantité de calcul de `fast`, celle de `slow`, et la période de `c`. Le but de l'exemple est de se retrouver avec le cas optimal : `c` est vrai une fois sur n et la quantité de calcul de `fast` est n fois moins importante que celle de `slow`. Dans ce cas, en omettant tous les surcoûts, nous devrions avoir un gain d'un facteur 2. Dans la réalité, il faut compter les latences des communications et le surcoût des futures. La granularité est ainsi déterminante. Ce n'est qu'avec une granularité infinie que nous pouvons atteindre 2. Nous observons un gain avec la version asynchrone à partir du moment où `slow` effectue aux alentours de 248 nanosecondes de calculs.

Cette valeur est similaire à celle de `gc_one_a` (269 nanosecondes). L'utilisation ou non de `get` avec une attente active n'a pas d'influence dans cet exemple.

5.3.3 Perspectives

Nous n'avons jamais évoqué les questions de WCET (worst case execution time) qui sont critiques dans les systèmes embarqués. En effet, nous n'avons pas de réponse générale adéquate. Sur les plate-formes à notre disposition, aucune garantie n'est possible concernant les temps de communication entre cœurs. Pire, le modèle mémoire C++ 2011 ne permet pas de prouver que la communication aura lieu, simplement que si elle a lieu, elle sera celle escomptée. Ce modèle est à l'heure actuelle le seul à notre disposition qui ne soit pas dépendant de la plate-forme, tout en étant très efficace puisque l'exécution que nous proposons est sans verrou ni appel système. Nous espérons que de futurs évolutions ou alternatives vont voir le jour.

Le détournement des futures nous a permis d'attirer l'attention sur les possibilités d'optimisations offertes par l'utilisation d'une gestion dynamique de la mémoire. Les conventions d'appels et la structure de la mémoire imposent des restrictions fortes pour l'optimisation mémoire en Heptagon [Ger+12]. Nous imaginons pouvoir les outrepasser avec une gestion dynamique de la mémoire.

Notez que les « références » induites par les futures ne sont pas des références quelconques. En particulier, il y a une et une seule écriture dans chaque référence. Ce qui assure le déterminisme de la sémantique, que nous perdrions si nous proposions au programmeur de vraies références. L'approche monadique et le typage linéaire sont probablement les clefs d'une solution générale à la problématique de la gestion dynamique de la mémoire en gardant une allocation statique. Est ce que cela peut être ajouté sans que la compilation ou la programmation soient beaucoup plus ardues reste une question ouverte.

Troisième partie

Les réseaux de Kahn et les langages synchrones

Nous présentons dans cette partie une approche « avec des dates » des réseaux de Kahn [Kah74] pour tenter d'éclairer les différentes faces d'une même pièce que sont la spécification modulaire d'un programme réactif, sa compilation modulaire, l'approche synchrone et le calcul d'horloge. Une manière de revisiter l'histoire est de dire que les réseaux de Kahn ont défini des fondations sémantiques solides et très générales pour les systèmes flots de données et que les languages synchrones et en particulier la lignée Lustre, Lucid Synchrone, Scade ont apporté une compilation modulaire et efficace, d'une partie de ces réseaux, vers du code séquentiel. Les réseaux acceptés, dits *synchrones*, ont pour principale contrainte de communiquer au travers de files ne pouvant stocker qu'un scalaire et devant être vides entre chaque instant. Le calcul d'horloge assure cette contrainte grâce à une notion d'instant allant de paire avec la compilation vers une fonction de transition, comme montré pour Heptagon en partie I.

La lignée a perduré avec récemment Lucy-n [Pla10] qui a proposé plusieurs nouveaux calculs d'horloges permettant de relâcher la contrainte des communications scalaires en gardant des files de communications statiquement bornées. La génération de code de Lucy-n n'est à ce jour pas totalement achevée. Quoi qu'il en soit, Lucy-n a permis de retrouver une approche et une programmation beaucoup plus fidèle aux réseaux de Kahn et à l'approche flot de données.

Nous tentons dans cette partie une approche collant le plus possible à la sémantique de Kahn et à l'esprit de la programmation flots de données, pour comprendre les choix effectués par les langages synchrones, mais aussi évoquer des possibilités intéressantes qui ont été mises de côté. L'approche sémantique dénotationnelle épure la problématique, mais cache en partie les difficultés de la modularité, de l'ordonnancement et de la compilation. Le problème attaqué est très similaire à celui qui avait cours pour le lambda calcul fin des années 70 [Ber78] : la sémantique de Scott contient des programmes qui ne peuvent s'écrire. Dans ce contexte sont nées les notions de séquentialité [Vui74] et de stabilité [Ber78]. Ces deux notions sont aussi déterminantes pour les réseaux de Kahn.

Notre volonté de comprendre les implications pour la compilation et la génération de code se retrouve dans notre choix de toujours ramener notre analyse à une représentation du réseau sous la forme d'un système de transition représentant une « machine de Mealy ». Pour traiter dans leur ensemble les réseaux de Kahn, sans nous confronter aux problèmes d'équité

de l'ordonnancement, nous plongeons les machines de Mealy dans un cadre général où une transition consomme des mots finis mais produit des mots infinis. Dans ce cadre général, nous pouvons exhiber une forme canonique, parler de granularité et de coefficient de Lipschitz, classer les réseaux en plusieurs catégories, ceux réactifs, ceux interactifs, ceux réagissants initialement sans entrée, etc. Ces considérations et l'introduction de nos notations font l'objet du chapitre 6.

Parmi tous les réseaux, ceux qui nous intéressent sont ceux pour lesquels il existe une datation principale. Une datation principale définit un ordre entre les entrées et les sorties, tel qu'une entrée est consommée avant la production d'une sortie ssi cela est nécessaire. Cette datation principale, si elle existe, est une spécification simple du comportement du réseau et elle permet l'écriture d'un système de transition déterministe. L'idée est d'avoir une compilation modulaire venant avec une spécification, comme cela est fait en Heptagon par exemple.

Nous montrons en chapitre 7, que l'existence d'une datation principale correspond à ce que nous appelons la classe des réseaux ordonnés, une sous-classe des réseaux stables. Nous montrons de plus qu'un réseau est ordonné ssi il s'écrit avec un système de transition dans une forme normale que nous caractérisons comme déterministe, productive et maximale. Cependant, parmi les réseaux ordonnés, certains nécessitent de produire une infinité de valeurs en un temps borné, ce que nous appelons non-réactif. Ils se caractérisent aisément avec la forme normale. Même si le comportement d'un réseau ordonné est déterministe et décrit par une datation principale, certains ne peuvent se compiler dans un langage séquentiel. Si cela est requis, il faut encore restreindre les réseaux considérés et nous tombons sur les réseaux séquentiels réactifs : ceux s'écrivant comme un unique processus dans le langage proposé par Kahn et Plotkin [KM76].

Une datation décrit le comportement du réseau, mais pas celui du code généré. Dans les langages synchrones, la description du comportement repose sur une notion d'instant et une horloge indique pour chaque instant la présence d'une valeur. Ce niveau de description permet de générer du code efficacement, avec des files de communications de taille bornées. Nous avons montré pour Heptagon, que le calcul d'horloge assure que toute valeur produite sera consommée dans le même instant.

Nous montrons dans le chapitre 8 comment le choix des instants permet de passer de la datation aux horloges. Le cadre que nous donnons est général. Les horloges au lieu d'être booléennes et de simplement indiquer la présence, sont entières et indiquent combien il y a de valeur pendant l'instant. Le lien entre instant et fonction de transition est une question de convention. En Heptagon, un instant équivaut à un appel à la fonction de transition. C'est l'approche en boucle simple [HRR91]. Nous développons en détail cette approche, d'autres approches le sont en section 10.1. Nous montrons que les horloges entières ne sont pas nécessaire pour décrire fidèlement un réseau ordonné réactif. Par contre, leur utilisation permet d'avoir une forme normale pour les signatures d'horloges, signatures que nous appelons comblées. Nous montrons aussi que les horloges entières apportent un vrai sur-échantillonage quand elles sont utilisées comme horloges d'activation : un instant peut alors correspondre à plusieurs appels à une même fonction de transition.

Après avoir traité de manière générale le pointage en boucle simple, nous regardons en détail les conséquences de notre nouveau cadre pour Lucy-n et en particulier la notion de signature d'horloge principale. Lucy-n est le cobaye idéal, puisque tout le contrôle du programme est ultimement périodique et statiquement connu. Nous trouvons que l'ordonnancement au plus tard (ALAP *as late as possible*) en est la clef de voûte. Toutefois, son utilisation sur les programmes à plusieurs sorties requiert une attention redoublée, puisque le produit de réseaux ordonnés ne l'est pas forcément. Lucy-n est aussi l'occasion de s'aventurer à donner une définition de la causalité du point fixe d'un réseau de Kahn.

Finalement, nous discutons plusieurs pointages existants. Celui d'Heptagon, en boucle simple, est loin d'être aussi puissant que nous le souhaitions. Sa faiblesse est aussi sa force car il ne nécessite aucune interprétation des flots. À l'opposé il y a le cas de SIGNAL qui n'a pas en général un unique pointage, car sa sémantique n'est pas indépendante de l'ordonnancement. Les objets synchrones avec des polices [Cas+09] tombent dans la même catégorie, même si le contexte est plus simple. Nous étudions, pour ces deux cas, les circonstances où la sémantique est déterministe. Dans ce cas, nous montrons que leur pointage est riche et permet de décrire des réseaux qui ne sont pas ordonnés.

Tout au long de cette partie, nous puisons dans les travaux de J. Dennis [Den74], G. Kahn [Kah74], G. Plotkin [KP78], J. Vuillemin [Vui74], G. Berry [Ber75 ; Ber78], L. Lamport [Lam78], N. Halbwachs [Hal84], P. Caspi [Cas92], plus récemment M. Pouzet [CP98], E. A. Lee [LS98], dernièrement F. Plateau [Pla10] et bien d'autres, ainsi que dans nos discussions et donc dans les points de vue et idées de mes collègues, en particulier A. Guatto et L. Mandel.

Chapitre 6

Les réseaux de Kahn réactifs : rKPN (*Reactive Kahn Process Network*)

Ce premier chapitre traite des réseaux de Kahn et de leurs modélisations avec un système de transition. L'optique est de considérer le réseau comme une machine de Mealy, décrite par une fonction de transition. En effet, cela correspond à la forme compilée du réseau que nous souhaitons obtenir, similairement à ce que nous avons montré pour Heptagon en section 2.5. L'idée est dans un premier temps d'accepter tous les réseaux, pour ne garder in fine que ceux dont la compilation est raisonnable. Nous partons donc d'une fonction représentant le réseau et montrons comment sa continuité permet de la lier à une exécution « pas à pas » (cf. section 6.1.3), qui consomme les entrées et produit les sorties petit à petit. Cette représentation avec un système de transition confluent non déterministe, se heurte à un problème d'équité de l'ordonnancement car les traces considérées sont des mots infinis [Sta89]. La solution classique est de rajouter explicitement la contrainte d'équité de l'ordonnancement [LS89 ; GB03]. Notre solution va être de permettre au système de produire un mot infini en une transition. De cette manière, nous nous ramenons à des systèmes confluents et fortement normalisants pour des entrées finies. Ce type de système de transition permet de minimiser les transitions qui ne consomment pas d'entrées et d'obtenir une forme canonique. Celle-ci permet de séparer les réseaux en deux classes, ceux que nous appelons de Moore qui peuvent produire une sortie sans entrée et ceux que nous appelons Mealy. De plus, cette forme canonique va être utile pour quantifier la granularité de calcul du réseau.

Avant de pouvoir entrer dans le vif du sujet, nous posons les concepts, notations et bases techniques utilisées dans toute cette partie.

6.1 Les rKPN, des réseaux de Kahn associés à une datation globale des entrées et des sorties

Commençons par une définition qui s'apparente à un pléonasme : un rKPN est un réseau de Kahn qui interagit avec son environnement comme un programme réactif. Cette définition informelle requiert de définir ce que nous entendons par programme réactif car ce terme a été et est utilisé dans une multitude de sens. Nous considérons que la notion principale d'un programme réactif est la notion de réaction.

6.1.1 *Définition* **Réaction**
Une réaction est une *période de temps bornée*, que l'on appelera un *instant*, durant lequel le système reçoit un nombre fini d'entrées et produit un nombre fini de sorties. Il n'y a, a priori, pas de contraintes sur ces deux actions, elles peuvent être entremêlées, il peut ne pas y avoir de sortie générée ou bien d'entrée consommée.

6.1.2 *Définition* **Programme réactif**
Un programme est dit réactif si son exécution est une séquence de réactions.

Cette définition est très générale mais souligne l'importance de définir ce que sont les entrées et les sorties du programme. Nous considérons donc les réseaux de Kahn dans lesquels les entrées et les sorties du réseau sont identifiées. Ces ports sont les *seuls* moyens de communications avec le reste du monde.

Il n'y a pas intrinsèquement de notion de temps et a fortiori d'instant dans les réseaux de Kahn. La sémantique dénotationnelle est fonctionnelle sur le domaine des flots possiblement infinis : son exécution s'apparente à un programme transformationnel [HP85], prenant en entrées des flots, *puis* calculant par point fixe les flots des sorties, possiblement en utilisant une mémoire infinie. Bien évidemment, si les réseaux de Kahn n'avaient pas d'autres interprétations, ils seraient peu utiles pour décrire des systèmes réactifs.

Cependant, les réseaux de Kahn ont une propriété fondamentale : la *continuité* sur le domaine des flots avec l'ordre partiel complet d'inclusion préfixe des flots (cf. section 2.1.2). De plus la continuité induit la *monotonie*. Ces deux notions sont présentées par Kahn [Kah74] en parlant des nœuds du réseau, nous citons ses propos ici :

> The restriction of the interpretation of nodes to continuous functions can be understood in concrete terms :
>
> 1. Monotonicity means that receiving more input at a computing station can only provoke it to send more output. Indeed this a crucial property since it allows parallel operation : a machine need not have all of its input to start computing, since future input concerns only future output.
> 2. Furthermore continuity prevents any station from deciding to send some output only after it has received an infinite amount of input.

En propriété 0.2 nous rappelons mathématiquement le sens de ces deux points et nommons le deuxième *attente finie*.

Pour insister sur la relativité de la notion d'instant, nous n'allons pas l'associer dans un premier temps à la présence des valeurs, comme cela est usuellement fait dans les langages synchrones. Nous allons prendre un référentiel absolu (le temps) qui va nous permettre de parler indépendamment des instants (des intervalles de temps) et de la présence des valeurs (des dates). Les dates des valeurs ont pour contrainte de respecter les dépendances de données :

6.1.3 *Hypothèse* **Causalité des dates**
Une valeur, ayant une date précédant la date d'une autre, n'en dépend pas.

Ainsi les dates que nous associerons aux valeurs forment un *ordre total respectant l'ordre partiel des dépendances de données*. Cette approche très classique correspond aux travaux de Lamport [Lam78]. Nous formaliserons les dépendances de données en détail en section 7.1, qui ne sont pas une évidence dans les réseaux de Kahn.

6.1.1 Notations

Dans ce chapitre et la suite du manuscrit, nous utilisons les notations introduites pour la sémantique de Kahn d'Heptagon, cf. chapitre 2. Nous utilisons deux autres notations classiques [KP78], la relation de compatibilité et l'ensemble des approximants.

6.1.4 *Définition* **Compatibilité**
Deux flots $\mathbf{X}$ et $\mathbf{X}'$ sont dits compatibles, noté $\mathbf{X} \uparrow \mathbf{X}'$, ssi ils ont un majorant : $\exists \mathbf{D}, \mathbf{X} \subseteq \mathbf{D} \wedge \mathbf{X}' \subseteq \mathbf{D}$.

6.1.5 *Définition* **Ensemble des approximants**
L'ensemble des approximants d'un flots $\mathbf{X}$ est l'ensemble des flots finis préfixes de $\mathbf{X}$. Cet ensemble est noté $\mathcal{A}(\mathbf{X}) = \{\mathbf{Y} \subseteq \mathbf{X} \mid |\mathbf{Y}| < \infty\}$.

6.1.2 Formalisation

Notre formalisation du comportement temporel d'un système est fortement inspirée des travaux de Halbwachs et Caspi [CH86]. Le comportement d'un système est décrit par la suite des écritures de ses variables et chaque écriture est associée à une date. Cette datation leur permet d'effectuer des preuves sur les systèmes temps réel. Le temps est n'importe quel temps métrique, bien que les exemples soient avec $\overline{\mathbb{R}}$.

Par simplicité, nous nous bornons aux dates positives $\overline{\mathbb{R}}_+$, avec 0 la date spéciale du début de l'exécution. Nous prenons $(\leq)$ l'ordre total usuel de $\overline{\mathbb{R}}_+$, que nous étendons en ordre partiel strict sur les *parties non vides* de $\overline{\mathbb{R}}_+$:

$$\forall X, X' \in \mathfrak{P}(\overline{\mathbb{R}}_+) \setminus \emptyset, \quad X < X' \iff \quad \forall t \in X, \forall t' \in X', t < t'$$

Outre les dates, nous définissons les instants.

6.1.6 *Définition* **Instants**
Un instant est défini comme un ensemble ouvert, non vide et convexe de dates : un intervalle ouvert. L'ensemble des instants d'un réseau est noté R. Il est indexé par $\mathbb{N}^*$. Nous choisissons par commodité de faire commencer les indices à 1 (ce qui permet entre autres d'aisément parler du « premier instant »). Nous notons R^k le $k$ième instant du réseau et imposons que l'indexation respecte l'ordre des instants et qu'ils soient d'intersections vides :

$$\forall k \in \mathbb{N}^*, \quad R^k < R^{k+1}$$

« Toutes les dates du $k$ième instant sont strictement avant celles du $k+1$ième instant. »

6.1.7 *Définition* **Durée**
La durée d'un instant (ou d'un ensemble de dates) est donnée par son diamètre. Le diamètre est défini pour toute partie :

$$\forall X \in \mathfrak{P}(\overline{\mathbb{R}}_+) \setminus \emptyset, \quad \mathrm{diam}(X) = \sup(X) - \inf(X)$$

6.1.8 *Hypothèse* **Temps de réaction uniformément bornés**
Les instants d'un réseau doivent représenter des réactions et donc être de durée uniformément bornée :

$$\exists \Delta \in \mathbb{R}, \forall k, \quad \mathrm{diam}(R^k) \leq \Delta$$

De plus, pour nous prémunir des comportements Zeno, nous demandons à ce qu'il ne puisse y avoir une infinité d'instants dans un temps fini $\lim_k \sup(R^k) = \infty$ ce que nous assurons avec une condition, plus forte, d'uniforme minoration :

$$\exists \Delta_m \in \mathbb{R}_+, \forall k, \quad \Delta_m \leq \mathrm{diam}(R^k)$$

6.1.2.1 Réseau de Kahn réactif

Nous nous bornons aux réseaux de Kahn dont la topologie est fixe. Un tel réseau est défini par le tuple $(N, \twoheadrightarrow, E, S, (E_n, S_n, f_n)_{n\in N})$. N est l'ensemble des nœuds du réseau, chacun étant associé à une fonction continue dans le domaine de Kahn f_n, dont chaque entrée est représentée par un port $p \in E_n$ et chaque sortie un port $p \in S_n$. Finalement, le réseau va se comporter lui aussi comme une fonction continue, dont les entrées sont les ports d'entrée E et de sortie S. Le réseau a d entrées $E = \{e^{(1)}, \cdots, e^{(d)}\}$ et r sorties $S = \{s^{(1)}, \cdots, s^{(r)}\}$. L'ensemble des ports est $P = E + S + (E_n + S_n)_{n\in N}$ avec $+$ l'union disjointe. La relation $\twoheadrightarrow$ identifie les connexions (les files de communications) entre ports. Sa notation est infixe, $\overset{p\ p'}{\longrightarrow\!\!\!\!\!\!\twoheadrightarrow}$ signifie que le port p est connecté au port p'. p peut être une des entrées du réseau ($p \in E$) ou bien être une des sorties d'un élément du réseau ($\exists n \in N, p \in S_n$). Pour expliciter ce deuxième cas, on notera de manière compacte $n \overset{p\ p'}{\longrightarrow\!\!\!\!\!\!\twoheadrightarrow}$. Respectivement, p' peut être une des sorties du réseau ($p' \in S$) ou bien être une des entrées d'un élément du réseau ($\exists n \in N, p' \in E_n$) noté $\overset{p\ p'}{\longrightarrow\!\!\!\!\!\!\twoheadrightarrow} n$. Ainsi $n_1 \overset{p\ p'}{\longrightarrow\!\!\!\!\!\!\twoheadrightarrow} n_2$ indique que la sortie p de n_1 est connectée à l'entrée p' de n_2.

Pour simplifier notre propos nous considérons que l'ensemble des nœuds N est fini. Permettre un nombre dénombrable de nœuds ne change rien de fondamental. Cependant la complexification induite est inutile, puisque in fine nous souhaitons un réseau fini qui réagit en un temps borné.

Un rKPN est un tel réseau de Kahn auquel associé à des instants R et une datation T.

6.1.2.2 La datation T

La datation spécifie l'*interface de communication* du réseau avec son environnement. Comme toutes les entrées et les sorties passent par les ports du réseau, la fonction de datation se résume à dater le passage des valeurs par ces ports. Le comportement du réseau peut dépendre des valeurs de ses entrées. La datation est donc une fonction des entrées. En prenant $\mathbf{D}$ le vecteur des flots d'entrées, la datation s'écrit $T_{\mathbf{D}}$, mais souvent, nous aurons $\mathbf{D}$ implicite. L'ensemble des dates des valeurs passant par le port $p \in (E \cup S)$ est noté $T_{\mathbf{D}}(p) \in \mathfrak{P}(\overline{\mathbb{R}}_+)$ ou plus simplement $T(p)$. Il y a une date associée à chaque valeur du flot passant au port p, nous utiliserons la notation d'indexation $T(p)[i]$ pour signifier la $i$ème date : $T(p)[i] \overset{\text{def}}{=} t \in T(p)$ tel que $\#\{t' \in T(p) \mid t' \leq t\} = i$. Nous ajoutons en plus deux conventions, $T(p)[0] = 0$ et si i est supérieur au nombre de valeurs passant par le port, alors $T(p)[i] = \infty$.

6.1.9 *Nota bene* **Notation de la date d'un flot et d'un vecteur de flots**
Nous noterons $T(\mathbf{x})$ « la date du flot $\mathbf{x}$ » passant par le port p, c'est-à-dire la date de sa dernière valeur : $T(\mathbf{x}) = T(p)[n]$ avec $n = |\mathbf{x}|$ étendu de manière continue si le flot est de taille infinie. Remarquez que par convention, $T(\varepsilon) = T(p)[0] = 0$. Finalement, « la date d'un vecteur de flots » est le maximum des dates de ses flots :

$$T(\mathbf{X}) = \max_{1\leq i\leq l} T(\mathbf{X}^{(i)}) \quad \text{avec } \mathbf{X} = (\mathbf{X}^{(1)}, \cdots, \mathbf{X}^{(l)})$$

Notez bien que $T(p)$ est une suite de dates, tandis que $T(\mathbf{x})$ et $T(\mathbf{X})$ sont des dates.

Si le flot $\mathbf{x}$ fini passe au port p, nous souhaitons que $T(p)$, muni de l'ordre total de $\overline{\mathbb{R}}_+$, caractérise de manière unique les valeurs du flot $\mathbf{x}$:

6.1.10 *Hypothèse* **Monotonie stricte**

$$\forall p, n, \qquad T(p)[n] < \infty \implies T(p)[n] < T(p)[n+1]$$

De manière générale, la monotonie stricte devient non stricte sur les vecteurs de flots :

$$\forall \mathbf{Z}', \mathbf{Z} \sqsubseteq \mathbf{Z}', \quad T(\mathbf{Z}) \leq T(\mathbf{Z}')$$

Dans sa thèse [Hal84], N. Halbwachs avait aussi pour volonté de décrire le comportement d'un système en associant une date à chaque valeur (ce qu'il appelle événement). Notre contexte est similaire, cependant, nous forçons un certains nombre d'hypothèses simplificatrices comme la monotonie stricte. Cette dernière hypothèse correspond à ce qu'il appelle une datation d'*événements à occurrences simples*.

6.1.11 *Remarque* La monotonie stricte de la datation garantit la bijection entre les dates d'un port et les valeurs du flot passant par ce port.

6.1.2.3 Réactivité

La notion d'instant du réseau a pour fonction de définir les instants de *réaction* du réseau. Ce sont eux qui permettent de parler de réactivité. Un rKPN devra avoir le comportement d'un programme réactif (pour la définition informelle voir 6.1.2). Puisque son comportement est décrit par la datation, c'est elle que nous allons contraindre et un réseau sera dit réactif s'il a une datation réactive.

6.1.12 *Propriété* **Datation réactive**
Une datation est réactive si durant tous les instants, l'ensemble des valeurs lues et écrites est fini. De plus, toutes les dates doivent appartenir à un instant : le système ne lit ni n'écrit de valeurs en dehors des instants :

$$T(E \cup S) \subseteq \bigcup_k R^k \quad \wedge \quad \forall k, \#\Big(T(E \cup S) \cap R^k\Big) < \infty$$

L'ensemble des dates d'une datation réactive est partitionné par les instants et nous pouvons spécifier T pour ne considérer que les dates d'un instant R. Pour tout port p :

$$T_R^k(p) = T(p) \cap R^k \qquad \text{et réciproquement} \qquad T(p) = \bigcup_k T_R^k(p)$$

6.1.13 *Remarque* Toute datation réactive T est telle que si $\mathbf{x}$ est un flot infini, alors $T(\mathbf{x}) = +\infty$ (par uniforme minoration des réactions et réactivité de la datation). Dans sa thèse [Hal84], N. Halbwach impose la réactivité comme une hypothèse des datations. Notre but est en effet de ne garder in fine que des datations réactives. Cependant, il va être plus simple de séparer l'existence d'une datation de sa réactivité.

6.1.14 *Propriété* **Datation uniformément réactive**
Pour être cohérent avec l'hypothèse 6.1.8 qui borne uniformément le temps de réaction, nous nous intéressons particulièrement aux datations ayant un nombre uniformément borné de valeurs par instant :

$$\exists B \in \mathbb{N}, \forall k, \quad \#T_R^k(E \cup S) < B$$

6.1.15 *Remarque* **Relativité des instants**
Au lieu de fixer les instants et de vérifier qu'une datation est uniformément réactive, il est possible de partir d'une datation et de chercher des instants pour qu'elle soit uniformément réactive. La condition nécessaire et suffisante à l'existence d'instants rendant une datation réactive est que *la fonction de datation n'ait pas de point d'accumulation* :

$$\forall p \in P, \quad T(p)[\infty] = \infty$$

Remarquez alors, que s'il est possible de changer les instants, toute datation réactive peut se voir transformée en datation uniformément réactive.

6.1.3 Sémantique d'un réseau de Kahn

Le réseau consiste en un ensemble de nœuds, chaque nœud n est associé à une fonction continue f_n. Ce qui nous intéresse est le comportement global du réseau. Ce comportement global est le résultat du calcul en parallèle de tous les nœuds. La continuité de chaque nœud permet de définir de manière unique le comportement du réseau quelle que soit la manière de calculer le point fixe du système. Ce résultat est souvent appelé le principe de Kahn [LS89].

L'intégralité du réseau se met sous la forme compacte d'une fonction continue f, telle que le premier argument [1] dénote les flots des entrées du système $\mathbf{X}$, le second, les flots des variables internes au réseau $\mathbf{V}$ et le troisième, les flots de sorties du réseau $\mathbf{Y}$. Les entrées sont intouchées par f :

$$\forall \mathbf{X}, \forall \mathbf{V}, \forall \mathbf{Y}, \quad (\mathbf{X}, \mathbf{V}', \mathbf{Y}') = f(\mathbf{X},\mathbf{V},\mathbf{Y})$$

Cette écriture inhabituelle avec l'entrée $\mathbf{X}$ intouchée va nous servir à insister sur l'exécution du réseau et la manière dont il consomme ses entrées *pendant* le calcul de son point fixe.

La description du système sous cette forme est toujours possible. Voici une construction [Kah74] la liant au réseau :

6.1.16 *Propriété* **Construction de f à partir des f_n**
Considérons de manière systématique, que les f_n prennent le vecteur de flots d'entrées $(\mathbf{V}^{(e_n)})_{e_n \in E_n}$, produisent le vecteur de flots de sorties $(\mathbf{V'}^{(s_n)})_{s_n \in S_n}$ et définissent l'équation :

$$\left(\left(\mathbf{V'}^{(s_n)}\right)_{s_n \in S_n}\right) = f_n\left(\left(\mathbf{V}^{(e_n)}\right)_{e_n \in E_n}\right)$$

En notant $\mathbf{X} = ((\mathbf{X}^{(e)})_{e \in E})$, $\mathbf{Y} = ((\mathbf{Y}^{(s)})_{s \in S})$ et $\mathbf{V} = ((\mathbf{V}^{(p)})_{p \in P \setminus (E \cup S)})$, le système $(\mathbf{X}, \mathbf{V}', \mathbf{Y}') = f(\mathbf{X},\mathbf{V},\mathbf{Y})$ est composé des équations des f_n additionnées des interconnexions

1. Nous utilisons dans cette partie les notations introduites pour la sémantique d'Heptagon en section 2.1 et en partie IV

suivantes :

$$\begin{aligned} n \xrightarrow{p\ p'}\!\!\!\!\!\rightarrow n' &\iff \mathbf{V}'^{(p')} = \mathbf{V}^{(p)} \\ \xrightarrow{e\ p}\!\!\!\!\!\rightarrow n &\iff \mathbf{V}'^{(p)} = \mathbf{X}^{(e)} \\ n \xrightarrow{p\ s}\!\!\!\!\!\rightarrow &\iff \mathbf{Y}'^{(s)} = \mathbf{V}^{(p)} \end{aligned}$$

La description du réseau sous la forme d'une fonction f définit de manière globale le calcul effectué par le système. Avec les entrées $\mathbf{X}$, le système calcule les sorties $\mathbf{Y_X}$ par point fixe :

$$(\mathbf{X}, \mathbf{V}, \mathbf{Y}) = \lim_{n \to \infty} f^n(\mathbf{X}, \varepsilon, \varepsilon)$$

Ce point fixe est bien défini car f est continue et $(\mathbf{X}, \varepsilon, \varepsilon) \sqsubseteq f(\mathbf{X}, \varepsilon, \varepsilon)$ donc par monotonie de f, $(f^n(\mathbf{X}, \varepsilon))_{n \in \mathbb{N}}$ est bien une chaîne, ce qui permet d'appliquer le théorème de Kleene.

Dans cette partie, nous nous intéressons particulièrement au réseau vu de l'extérieur. Les variables internes et la formulation technique de l'intérieur du réseau ne nous intéressent pas. Ainsi du réseau, nous n'allons retenir que sa sémantique dénotationnelle globale.

6.1.17 *Nota bene* **Sémantique dénotationnelle du réseau**
Pour représenter le calcul du réseau sous une forme fonctionnelle omettant les variables intermédiaires, nous écrivons :

$$\mathbf{Y} = F(\mathbf{X}) \qquad \text{pour} \qquad (\_, \_, \mathbf{Y}) = \lim_{n \to \infty} f^n(\mathbf{X}, \varepsilon, \varepsilon)$$

6.1.18 *Définition* **Équivalence de réseaux**
Deux représentations d'un réseau f et f' sont dites équivalentes ssi leurs points fixes calculent les mêmes sorties :

$$\forall \mathbf{X}, \qquad F(\mathbf{X}) = F'(\mathbf{X})$$

6.1.19 *Remarque* **Ordonnancement équitable**
Un appel à la fonction f que nous avons construite en propriété 6.1.16, revient à appeler une fois chacune des f_n. Le calcul de point fixe de f effectue donc en parallèle celui des f_n, avec une itération de chacun des points fixes pour une itération de celui de f. Il est possible de définir f pour représenter n'importe quel ordonnancement de ces sous-itérations. Tant que l'ordonnancement est équitable [Kah74], c'est-à-dire qu'il n'y a pas un nœud dont le point fixe attend un temps infini avant d'effectuer une nouvelle itération, la continuité des f_n assurera que le point fixe calculé est le même. C'est cette propriété très forte des réseaux de Kahn qui permet de ne s'intéresser qu'à la fonction globale du système.

On pourrait croire qu'il existe déjà une notion de *réaction* en Kahn, puisque la sémantique repose sur un point fixe, point fixe qui peut se calculer par *itérations*. Ces itérations construisent les flots de manière monotone et constructive, au sens où l'information se propage et augmente à chaque itération de proche en proche dans le réseau. Cependant, vus de manière globale, les flots avancent les uns relativement aux autres de manière « anarchique », ou du moins dans un ordre qui ne correspond pas forcément à ce qui est désiré et qui est, de plus, très dépendant de la formulation technique de la fonction définissant le réseau.

De manière encore plus cruciale, ces itérations du point fixe n'indiquent pas quelles sont les valeurs des flots d'entrées qui sont consommées (plus jamais utiles / utiles au calcul de l'itération / utiles pour les itérations suivantes).

6.1.20 *Exemple* **Double addition**
Prenons le système qui calcule point à point l'équation $\mathbf{y} = \mathbf{X}^{(1)} + \mathbf{X}^{(2)} + \mathbf{X}^{(3)}$. Pour faciliter l'écriture de ce système, nous allons utiliser la fonction continue $plus$ qui additionne deux flots et s'arrête si l'un des deux termine :

$$plus(\varepsilon, \mathbf{x}') = \varepsilon \qquad plus(\mathbf{x}, \varepsilon) = \varepsilon \qquad plus(x.\mathbf{x}, x'.\mathbf{x}') = (x + x').plus(\mathbf{x},\mathbf{x}')$$

— La version la plus « simple » du système est sans variable intermédiaire, en utilisant la fonction d'addition directement :

$$f_{\text{dadd}}(\mathbf{X}, , \mathbf{y}) = \Big(\mathbf{X}, , plus(plus(\mathbf{X}^{(1)}, \mathbf{X}^{(2)}), \mathbf{X}^{(3)})\Big)$$

Le calcul du point fixe nécessaire à F_{dadd} s'effectue en une itération au plus : $f^2_{\text{dadd}} = f_{\text{dadd}}$.

— Il est possible de définir le réseau avec une variable intermédiaire $\mathbf{v}$ pour stocker le résultat de la première addition $\mathbf{X}^{(1)} + \mathbf{X}^{(2)}$:

$$f_{\text{dadd2}}(\mathbf{X}, \mathbf{v}, \mathbf{y}) = \Big(\mathbf{X}, plus(\mathbf{X}^{(1)}, \mathbf{X}^{(2)}), plus(\mathbf{v}, \mathbf{X}^{(3)})\Big)$$

Prenons comme entrées $\mathbf{X}^{(1)} = 1.2$, $\mathbf{X}^{(2)} = 1.2.3$ et $\mathbf{X}^{(3)} = 1.2$, le calcul du point fixe se déroule ainsi :

$$\begin{aligned} f_{\text{dadd2}}((1.2, 1.2.3, 1.2), \varepsilon, \varepsilon) &= (\ldots, 2.4, \varepsilon) \\ f_{\text{dadd2}}(\ldots, 2.4, \varepsilon) &= (\ldots, 2.4, 3.6) \\ f_{\text{dadd2}}(\ldots, 2.4, 3) &= (\ldots, 2.4, 3.6) \end{aligned}$$

Nous pouvons noter que quelque soit $\mathbf{X}$, le système converge en au maximum deux coups : $f^3_{\text{dadd}} = f^2_{\text{dadd2}}$.

— Nous pouvons aussi définir le réseau de manière équivalente mais de telle sorte qu'il ne converge pas en temps fini si l'on considère les variables intermédiaires :

$$f_{\text{daddy}}(\mathbf{X}, \mathbf{v}, \mathbf{y}) = \Big(\mathbf{X}, 0.\mathbf{v}, plus(plus(\mathbf{X}^{(2)}, plus(\mathbf{X}^{(1)},\mathbf{v})), \mathbf{X}^{(3)})\Big)$$

Dans cette version, nous utilisons $\mathbf{v}$ pour construire le flot constant de 0. Sa construction rajoute une valeur à chaque application de f_{daddy} et permet de ne faire qu'une addition de plus à chaque application :

$$\begin{aligned} f_{\text{daddy}}((1.2, 1.2.3, 1.2), \varepsilon, \varepsilon) &= (\ldots, 0, \varepsilon) \\ f_{\text{daddy}}(\ldots, 0, \varepsilon) &= (\ldots, 0.0, 3) \\ f_{\text{daddy}}(\ldots, 0.0, 3) &= (\ldots, 0.0.0, 3.6) \\ f_{\text{daddy}}(\ldots, 0.0.0, 3.6) &= (\ldots, 0.0.0.0, 3.6) \\ &\vdots \end{aligned}$$

Le calcul du point fixe peut néanmoins s'arrêter à partir du moment où le flot des sorties est entièrement calculé : $\forall \mathbf{X}, n \geq \min_i \left|\mathbf{X}^{(i)}\right| + 1, (\_, \_, F_{\text{daddy}}(\mathbf{X})) = f^{\eta}_{\text{daddy}}(\mathbf{X}, \varepsilon, \varepsilon)$.

Loin d'être exhaustives, ces trois versions équivalentes de la double addition mettent en évidence la grande diversité pour exprimer un système. Elles soulignent le fait que pour calculer le même résultat à partir des même entrées, le nombre d'itérations du réseau est très variable suivant l'écriture du réseau.

6.1.4 Maîtrise du calcul du point fixe

Notre présentation a pour but d'initier au calcul d'un réseau de Kahn par une machine réactive : montrer le lien entre point fixe sur des flots infinis et calcul par itérations finies. Au passage, nous distinguons les réseaux pour lesquels les itérations finies sont naturelles (les réseaux interactifs) et les autres.

6.1.4.1 Point fixe incrémental glouton

La première étape est de contrôler la progression des entrées du réseau. Au lieu de faire le calcul de point fixe une seule fois avec les flots d'entrées dans leur intégralité et obtenir comme résultat les flots de sorties dans leur intégralité, nous allons calculer à chaque instant en utilisant uniquement un préfixe du flots des entrées. La continuité du calcul du point fixe permettra d'affirmer que les sorties que l'on récupère à partir de ces calculs intermédiaires sont des préfixes des flots de sorties.

Pour toute chaîne $(\mathbf{X}_k)_{k\in\mathbb{N}}$ qui converge vers $\mathbf{X}$, nous pouvons définir les chaînes $(\mathbf{Y}_k)_{k\in\mathbb{N}}$ et $(\mathbf{V}_k)_{k\in\mathbb{N}}$ des flots de sorties et de variables intermédiaires correspondants avec :

$$\forall k, \quad (\mathbf{X}_k, \mathbf{V}_k, \mathbf{Y}_k) = \lim_{n\to\infty} f^n(\mathbf{X}_k, \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon})$$

Un résultat classique de Scott [Sco69] est la continuité du point fixe par rapport à ses paramètres (les paramètres étant ici les entrées $\mathbf{X}_k$), ce qui implique la continuité de F. Nous pouvons réécrire ce résultat dans notre contexte :

$$\begin{aligned}(\mathbf{X}, \mathbf{V}, \mathbf{Y}) &= \lim_{n\to\infty} f^n(\mathbf{X}, \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}) = \lim_{n\to\infty} f^n(\lim_{k\to\infty} \mathbf{X}_k, \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}) && (6.1)\\ &= \lim_{n\to\infty}\lim_{k\to\infty} f^n(\mathbf{X}_k, \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}) && (6.2)\\ &= \lim_{k\to\infty}\lim_{n\to\infty} f^n(\mathbf{X}_k, \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}) = \lim_{k\to\infty} (\mathbf{X}_k, \mathbf{V}_k, \mathbf{Y}_k) && (6.3)\end{aligned}$$

Le passage de l'équation (6.1) à l'équation (6.2) est assuré par la continuité de f, tandis que la permutation des limites de l'équation (6.2) à l'équation (6.3) est possible car $f^n(\mathbf{X}_k, \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon})$ est croissante pour n et pour k (la preuve d'inversion des limites pouvant alors se ramener comme dans [Sco69] à regarder la chaîne $f^{\max(n,k)}(\mathbf{X}_{\max(n,k)}, \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon})$).

6.1.4.2 Point fixe récursif glouton

Le choix qui détermine toute notre construction est le choix de la chaîne $(\mathbf{X}_k)_k$. Il correspond à la décision de comment et quand le système reçoit des entrées. En effet, nous allons considérer qu'à la $k$ième réaction, le système calcule avec $\mathbf{X}_k$ pour entrées. Mais nous ne souhaitons pas redonner l'entièreté de $\mathbf{X}_k$, seulement les « nouvelles » entrées de l'instant que nous notons $\mathbf{X}_{\delta k}$ définies ainsi :

$$\forall i, 1 \le i \le d, \quad \mathbf{X}_{k-1}.\mathbf{X}_{\delta k} \stackrel{\text{def}}{=} \mathbf{X}_k \quad \text{avec la convention } \mathbf{X}_0 = \boldsymbol{\varepsilon}$$

6.1.21 *Nota bene* Il y a, avec cette définition, correspondance bi-univoque entre une chaîne $(\mathbf{X}_k)_k$ et la suite d'incréments (la suite de « de nouvelles valeurs ») $(\mathbf{X}_{\delta k})_k$. Nous utilisons ce fait de manière intensive dans notre manuscrit, pour passer d'une chaîne à sa suite d'incréments et vice versa. Pour plus de détails, se reporter en partie IV.

6.1.22 *Propriété* **Récursivité du point fixe incrémental glouton**
Il est possible de calculer le point fixe incrémental de manière récursive, en repartant à chaque instant de l'état précédent $(\mathbf{X}_{k-1}, \mathbf{V}_{k-1}, \mathbf{Y}_{k-1})$ avec les nouvelles entrées rajoutées :

$$\begin{aligned} \forall k, \quad (\mathbf{X}_k, \mathbf{V}_k, \mathbf{Y}_k) &= \lim_{n\to\infty} f^n(\mathbf{X}_k, \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}) && \text{(calcul incrémental)} \\ &= \lim_{n\to\infty} f^n(\mathbf{X}_{k-1}.\mathbf{X}_{\delta k}, \mathbf{V}_{k-1}, \mathbf{Y}_{k-1}) && \text{(calcul récursif)} \end{aligned}$$

Démonstration Nous allons montrer l'égalité en montrant la double inclusion :
— Nous avons par définition de $\mathbf{X}_{\delta k}$, $(\mathbf{X}_k, \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}) \sqsubseteq (\mathbf{X}_{k-1}.\mathbf{X}_{\delta k}, \mathbf{V}_{k-1}, \mathbf{Y}_{k-1})$ donc par monotonie de F, $\lim_{n\to\infty} f^n(\mathbf{X}_k, \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}) \sqsubseteq \lim_{n\to\infty} f^n(\mathbf{X}_{k-1}.\mathbf{X}_{\delta k}, \mathbf{V}_{k-1}, \mathbf{Y}_{k-1})$
— Par construction $\mathbf{V}_{k-1} \sqsubseteq \mathbf{V}_k$ et idem pour $\mathbf{Y}_{k-1} \sqsubseteq \mathbf{Y}_k$ donc $(\mathbf{X}_{k-1}.\mathbf{X}_{\delta k}, \mathbf{V}_{k-1}, \mathbf{Y}_{k-1}) \sqsubseteq (\mathbf{X}_k, \mathbf{V}_k, \mathbf{Y}_k)$, soit :
$\lim_{n\to\infty} f^n(\mathbf{X}_{k-1}.\mathbf{X}_{\delta k}, \mathbf{V}_{k-1}, \mathbf{Y}_{k-1}) \sqsubseteq \lim_{n\to\infty} f^n(\mathbf{X}_k, \mathbf{V}_k, \mathbf{Y}_k) = (\mathbf{X}_k, \mathbf{V}_k, \mathbf{Y}_k)$

Nous insisterons sur le calcul récursif et plus particulièrement son implémentation avec un système de transition (co-itératif) en section 6.2.

Dans l'optique de définir des datations réactives, le contrôle des entrées n'est pas suffisant. Nous souhaitons de plus avoir des sorties de tailles finies et calculer uniquement un nombre fini d'itérations par instants.

6.1.4.3 Réseaux interactifs

De manière générale, la suite $(\mathbf{Y}_k)_k$ n'est pas formée uniquement de flots finis. Cette garantie est une caractérisation importante du réseau, nous appelons de tels réseaux des réseaux interactifs.

6.1.23 *Définition* **Réseau interactif**
Un réseau est dit interactif ssi il ne produit pas de flots infinis à partir d'entrées finies.

$$\forall \mathbf{X}, \quad |\mathbf{X}| < \infty \implies |F(\mathbf{X})| < \infty$$

Remarquez que l'exemple 6.1.20 de la double addition est un réseau interactif. Les flots intérieurs ne faisant pas partie de la sémantique, la troisième définition du réseau est interactive, bien qu'elle produise de manière interne un flot infini.

Le pendant uniforme des réseaux interactif est le réseau uniformément interactif. La définition que nous donnons reprend la définition donnée par P. Caspi [Cas92] à propos des réseaux Lipschitz. Nous gérons de plus les réseaux capables de fournir des valeurs en sortie sans en avoir reçu en entrée, ce qui demande d'étendre la définition :

6.1.24 *Définition* **Réseau uniformément interactif ou réseau K-Lipschitz**
Un réseau est dit K-Lipschitz s'il est interactif et si la taille des nouvelles sorties produites est uniformément majoré par K fois la taille des nouvelles entrées fournies. En d'autres termes,

la dérivée de F est bornée. Cependant, le domaine est fermé à gauche, ce qui force à définir spécialement la dérivée en ε :

$$\exists K, \forall \mathbf{X}, \mathbf{X}', \quad |F(\varepsilon)| \leq K \quad \wedge \quad |\mathbf{X}| < \infty \implies |\mathbf{Y}'| \leq K|\mathbf{X}'| \text{ avec } F(\mathbf{X}).\mathbf{Y}' = F(\mathbf{X}.\mathbf{X}')$$

6.1.4.4 Calculs en temps fini

Il n'est pas acceptable de considérer que le système soit réactif si, durant chaque réaction, le réseau calcule un point fixe avec un nombre d'itérations infini. La continuité de F nous assure que si l'on attend une sortie finie du réseau, il la calcule en un temps fini :

6.1.25 *Propriété* **Sortie finie implique calcul fini**
Pour toute entrée $\mathbf{X}$, approximant (préfixe fini) du point fixe pour cette entrée $\mathbf{Y} \in \mathcal{A}(F(\mathbf{X}))$, il existe un rang $\eta \in \mathbb{N}$ à partir duquel les itérations du point fixe ont calculées l'approximant : $(\_, \_, \mathbf{Y}) \sqsubseteq f^{\eta}(\mathbf{X}, \varepsilon, \varepsilon)$.

Démonstration Posons $(\mathbf{X}, \mathbf{V}_k, \mathbf{Y}_k) = f^k(\mathbf{X}, \varepsilon, \varepsilon)$, de limite $(\mathbf{X}, \mathbf{V}, F(\mathbf{X}))$. Comme $(\varepsilon, \varepsilon, \mathbf{Y})$ est un approximant du point fixe de f, la propriété d'attente finie (induite par la continuité de f cf. propriété 0.2) assure l'existence de $\mathbf{X}' \in \mathcal{A}(\mathbf{X})$ et $\mathbf{V}' \in \mathcal{A}(\mathbf{V})$ tels que $(\varepsilon, \varepsilon, \mathbf{Y}) \sqsubseteq f(\mathbf{X}', \mathbf{V}', \mathbf{Y})$. Enfin, il existe $\eta < \infty$, tel que $\mathbf{V}' \sqsubseteq \mathbf{V}_\eta$ et $\mathbf{Y} \sqsubseteq \mathbf{Y}_\eta$, nous concluons avec la monotonie.

Cette propriété prend tout son sens pour les réseaux interactifs :

6.1.26 *Propriété* **Un réseau interactif converge vers son point fixe en un temps fini**
Si un réseau est interactif, alors le calcul du point fixe, comme le calcul récursif du réseau à partir d'une entrée finie va converger en un temps fini :

$$\forall |\mathbf{X}| < \infty, \exists \eta < \infty, \quad (\_, \_, F(\mathbf{X})) = f^{\eta}(\mathbf{X}, \varepsilon, \varepsilon)$$

6.1.27 *Remarque* Un réseau Lipschitz est un réseau interactif, il converge donc en un temps fini pour une entrée finie. Il semblerait naturel de pouvoir borner uniformément le temps de convergence du réseau, mais cela n'est pas possible sous la forme générale. Il faut avoir une représentation canonique plus forte du calcul du réseau pour affirmer une telle chose, cf. propriété 6.3.24.

Finissons avec un résultat générique usuel qui va nous permettre dans la section suivante d'affirmer l'existence d'une datation uniformément réactive pour tout système (cf. lemme 6.1.28). Il est possible de calculer le point fixe du réseau en lui fournissant les entrées petit à petit et en lui requérant les sorties petit à petit en ne le laissant calculer qu'un nombre d'itérations bornées à chaque fois.

Lemme 6.1.28 (Calcul par réactions finies)**.** *Quel que soit le système F et la chaîne $(\mathbf{X}_k)_k$ approximant $\mathbf{D}$, nous pouvons choisir une chaîne $(\mathbf{Y}_k)_k$ approximant $F(\mathbf{D})$, qui est le résultat, à chaque instant k, du calcul récursif du réseau en un nombre uniformément borné d'itérations.*

Démonstration Tout comme pour l'inversion des limites de l'équation 6.3, l'argument principal est la croissante de $f^n(\mathbf{X}_k, \varepsilon, \varepsilon)$ pour n et pour k, qui assure tout chemin croissant à converger vers la même limite. Reprenons l'équation 6.3 avec le changement de variable $n = \eta(k)$, correct tant que η tend vers l'infini $\lim_{k\to\infty} \eta(k) = \infty$:

$$(\mathbf{X}, \mathbf{V}, \mathbf{Y}) = \lim_{k\to\infty} f^{\eta(k)}(\mathbf{X}_k, \varepsilon, \varepsilon) = \lim_{n\to\infty} f^n(\mathbf{X}, \varepsilon, \varepsilon)$$

Notre but est d'exprimer le calcul en effectuant $\Delta(k)$ itérations au $k$ième appel récursif :

$$\forall k, \quad (\mathbf{X}_k, \mathbf{V}_k, \mathbf{Y}_k) = f^{\Delta(k)}(\mathbf{X}_{k-1}.\mathbf{X}_{\delta k}, \mathbf{V}_{k-1}, \mathbf{Y}_{k-1})$$

À condition que $\sum_{k'\leq k} \Delta(k')$ tende vers l'infini, nous posons $\eta(k) = \sum_{k'\leq k} \Delta(k')$ et calculons ce que nous souhaitons :

$$\lim_{k\to\infty} f^{\eta(k)}(\mathbf{X}_k, \varepsilon, \varepsilon) = \lim_{k\to\infty} (\mathbf{X}_k, \mathbf{V}_k, \mathbf{Y}_k)$$

La preuve est proche de celle de la propriété 6.1.22 :
- Par construction $(\mathbf{X}_k, \mathbf{V}_k, \mathbf{Y}_k) \sqsubseteq f^{\eta(k)}(\mathbf{X}_k, \varepsilon, \varepsilon)$.
- Réciproquement, il existe k' tel que $\eta(k') = 2\eta(k)$, ainsi par monotonie, $f^{\eta(k)}(\mathbf{X}_k, \varepsilon, \varepsilon)$ est un préfixe de $f^{\eta(k)}(\mathbf{X}_k, \mathbf{V}_k, \mathbf{Y}_k) \sqsubseteq (\mathbf{X}_{k'}, \mathbf{V}_{k'}, \mathbf{Y}_{k'})$

6.1.29 *Remarque* **Micro-ordonnancement**
Le fait que dans l'instant, nous effectuons un calcul de point fixe (que nous arrêtons arbitrairement tôt avec $\Delta(k)$) représente la volonté d'effectuer plusieurs calculs dans l'instant. Ces calculs ayant des dépendances, il est nécessaire d'effectuer des itérations pour les exécuter, ce qui est couramment appelé micro-ordonnancement [PC06].

Notre présentation récursive met en évidence que l'ordonnancement des instants (le point fixe sur k de l'équation (6.2)) et l'ordonnancement dans l'instant (le point fixe sur n) sont deux notions similaires, séparées par la définition arbitraire du découpage des entrées et de la fonction f.

6.1.5 Datations

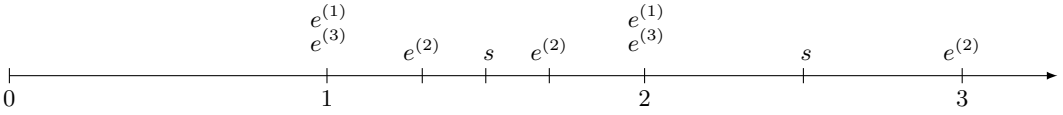
Dans cette section, nous allons nous familiariser avec les datations en donnant deux constructions génériques de datations qui nous seront utiles à plusieurs reprises par la suite.

6.1.5.1 Existence d'une datation (uniformément) réactive

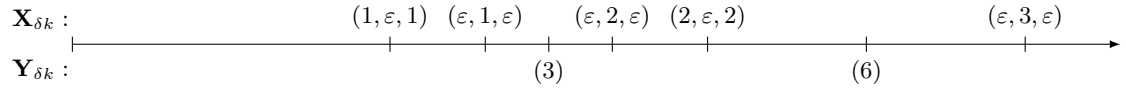
Une datation doit respecter les deux hypothèses :
- La monotonie stricte des dates à chaque port (cf. hypothèse 6.1.10) : une et une seule date par valeur.
- L'ordre des dates est compatible avec l'ordre des dépendances de données (cf. hypothèse 6.1.3) : une sortie est produite strictement après celles qui ont été utiles à son calcul.

6.1.30 *Exemple* **Double addition**
Reprenons le réseau calculant la double addition. Avec les entrées, $\mathbf{X}^{(1)} = 1.3$, $\mathbf{X}^{(2)} = 1.2.3$ et $\mathbf{X}^{(3)} = 1.2$ respectivement aux ports $e^{(1)}$, $e^{(2)}$ et $e^{(3)}$, la sémantique du réseau est $\mathbf{Y} = 3.6$ au port s. La stricte monotonie de $e^{(1)}$ impose que $T(e^{(1)})[1] < T(e^{(1)})[2] < T(e^{(1)})[3] < \infty$. Nous pouvons par exemple prendre $T(e^{(1)})[i] = i$ pour $1 \leq i \leq 3$. Pour la deuxième entrée,

seules deux valeurs passent, nous pouvons par exemple poser $T(e^{(2)})[1] = 1{,}5$, $T(e^{(2)})[2] = 1{,}7$ et $T(e^{(2)})[3] = 3$. Finalement nous décidons que la troisième entrée arrive en même temps que la première $T(e^{(1)})[i] = T(e^{(3)})[i]$ pour tout i. La datation de la sortie doit respecter les dépendances. En l'occurrence, la première valeur sortant nécessite la première valeur des trois entrées (puisque cela en est la somme) : $\forall 1 \leq j \leq 3, T(e^{(j)})[1] < T(s)[1]$, soit $1{,}5 < T(s)[1]$ et $2 < T(s)[2]$, prenons par exemple $T(s)[1] = 1{,}5$ et $T(s)[2] = 2{,}5$. Représentons-la sur un axe :



Ce qui correspond aux entrées et sorties suivantes :



À partir de toute suite $(\mathbf{X}_{\delta k})_k$ dont les éléments sont de taille uniformément bornée, le lemme 6.1.28 permet d'obtenir une suite d'éléments finis $(\mathbf{Y}_{\delta k})_k$ dont le $k$ième élément est calculé à partir de $\mathbf{X}_k$, qui converge vers la sémantique du réseau : $F(\lim_{k\to\infty} \mathbf{X}_k) = \lim_{k'\to\infty} \mathbf{Y}_{k'}$

À partir de deux telles suites, nous construisons une datation réactive, que nous noterons $T_{canon((R^k,\mathbf{X}_k,\mathbf{Y}_k)_k)}$. Nous séparons chaque instant en deux, la première moitié sert à faire entrer les valeurs de $\mathbf{X}_{\delta k}$, l'autre moitié à faire sortir les valeurs de $\mathbf{Y}_{\delta k}$. Ceci assure le respect des hypothèses. Une infinité de datations peut convenir. Les formules ci-dessous sont une manière simple de répartir les valeurs dans les intervalles.

$$T^k_{canon((R^k,\mathbf{X}_k,\mathbf{Y}_k)_k)}(e^{(i)}) \stackrel{\text{def}}{=} \left\{ \inf(R^k) + \frac{u \operatorname{diam}(R^k)}{2|\mathbf{X}^{(i)}_{\delta k}| + 1} \;\middle|\; 1 \leq u \leq |\mathbf{X}^{(i)}_{\delta k}| \right\} \tag{6.4}$$

$$T^k_{canon((R^k,\mathbf{X}_k,\mathbf{Y}_k)_k)}(s^{(j)}) \stackrel{\text{def}}{=} \left\{ \inf(R^k) + \frac{\operatorname{diam}(R^k)}{2} + \frac{u \operatorname{diam}(R^k)}{2|\mathbf{Y}^{(j)}_{\delta k}| + 1} \;\middle|\; 1 \leq u \leq |\mathbf{Y}^{(j)}_{\delta k}| \right\} \tag{6.5}$$

La construction de la datation, avec des valeurs explicites, n'est pas fondamentalement utile, ce qui l'est, par contre, sont les ordonnancement relatifs qu'elle induit. La principale caractéristique de T_{canon}, qui nous sera utile pour le théorème 7.3.10 est la suivante :

6.1.31 *Propriété*
La datation que nous avons construite est telle que les seules sorties avant une entrée sont celles des instants précédant l'entrée :

$$\forall \mathbf{X} \sqsubseteq \mathbf{D}, \forall \mathbf{Y} \sqsubseteq F(\mathbf{D}), \quad T_{canon}(\mathbf{X}) < T_{canon}(\mathbf{Y}) \iff \exists k, \mathbf{X} \sqsubseteq \mathbf{X}_k \wedge \mathbf{Y} \not\sqsubseteq \mathbf{Y}_{k-1}$$

Ceci toujours avec la convention que $\mathbf{Y}_0 = \boldsymbol{\varepsilon}$.

Démonstration Remarquons que par définition, toutes les dates qui ne sont pas explicitement définies sont infinies, par convention $T(\boldsymbol{\varepsilon}) = 0$ et les dates croissent avec k : $T_{canon}(\mathbf{X}) < \infty \iff \exists k_x, \mathbf{X}_{k_x-1} \sqsubset \mathbf{X} \sqsubseteq \mathbf{X}_{k_x}$ soit $T_{canon}(\mathbf{X}) < \infty \iff T_{canon}(\mathbf{X}) \leq T_{canon}(\mathbf{X}_{k_x})$.
Idem pour les sorties $T_{canon}(\mathbf{Y}) < \infty \iff \exists k_y, \mathbf{Y}_{k_y-1} \sqsubset \mathbf{Y} \sqsubseteq \mathbf{Y}_k$ soit $T_{canon}(\mathbf{Y}) < \infty \iff T_{canon}(\mathbf{Y}) \leq T_{canon}(\mathbf{Y}_{k_y})$.
Mais par construction, $T_{canon}(\mathbf{Y}) \leq T_{canon}(\mathbf{Y}_{k_y}) \iff T_{canon}(\mathbf{Y}) \leq T_{canon}(\mathbf{X}_{k_y+1})$. Finalement $T(\mathbf{Y}) \leq T(\mathbf{X}_k) \iff \mathbf{Y} \sqsubseteq \mathbf{Y}_{k-1}$.

6.1.5.2 Liberté des dates

Avancer des entrées ou retarder des sorties À partir de toute datation correcte, il est possible d'avancer les dates de n'importe quelle entrée (en respectant le fait que les dates d'un même flot doivent être monotones), et respectivement retarder les sorties. Ceci provient de la liberté offerte par le lemme 6.1.28 et correspond à choisir $(\mathbf{X}_k)_k$ qui croit plus rapidement (resp. $(\mathbf{Y}_k)_k$ qui croit plus lentement).

Cette liberté a dans les faits un coût en espace. Par exemple, une valeur calculée à l'instant k par le réseau doit, pour sortir plus tardivement, être stockée dans les files du réseau. De manière plus cruciale, cette liberté sur-contraint l'utilisation du réseau, ce que nous verrons en chapitre 7.

Avancer des sorties ou retarder des entrées Dans la datation que nous venons de construire nous avons considéré que toutes les sorties d'un instant dépendent de toutes les entrées de l'instant. Ceci est pourtant loin d'être toujours le cas, surtout si l'on choisit de donner beaucoup de valeurs en entrée pendant un instant, les premières valeurs des sorties ne dépendront probablement pas des dernières valeurs des entrées. Il faudrait, pour connaître les contraintes exactes sur les dates, pouvoir exprimer les dépendances de données. C'est ce que nous étudions dans la section 7.2.

Pour ces deux raisons, laisser libre le choix des sorties n'est pas vraiment intéressant. Nous aurons presque toujours recours à une datation au plus tôt, produisant tout ce qui est possible à chaque instant.

6.1.5.3 Datation au plus tôt : T_{early}

Nous allons construire une datation au plus tôt, notée T_{early}, ayant la propriété suivante :

6.1.32 *Propriété* **Datation au plus tôt**
Une datation au plus tôt date les sorties dans la même réaction que les entrées servant à les générer. Autrement dit, les sorties ne prennent pas de retard et à la fin de chaque instant, le système ne peut produire plus sans recevoir plus d'entrées :

$$T(\mathbf{X}) \in R^k \implies T(F(\mathbf{X})) \in R^k$$

Pour obtenir une datation au plus tôt, nous utilisons la chaîne de sorties définie de manière gloutonne par $(\mathbf{Y}_k)_k = (F(\mathbf{X}_k))_k$. Ensuite, nous souhaitons simplement définir la datation au plus tôt pour la chaîne d'entrées $(\mathbf{X}_k)_k$ par $T_{early((R^k,\mathbf{X}_k)_k)} = T_{canon((R^k,\mathbf{X}_k,\mathbf{Y}_k)_k)}$. Mais cette construction n'est pas toujours correcte puisqu'elle requiert que les $\mathbf{X}_{\delta k}$ *et* les $\mathbf{Y}_{\delta k}$ soient finis. Deux cas sont favorables :

— Si le réseau est Lipschitz et les $\mathbf{X}_{\delta k}$ uniformément bornés, alors les $\mathbf{Y}_{\delta k}$ le sont aussi et T_{early} est uniformément réactive.
— Si le réseau est interactif et les $\mathbf{X}_{\delta k}$ finis, alors les $\mathbf{Y}_k$ le sont aussi et T_{early} est réactive.

Cependant il n'y a pas que les réseaux interactifs qui nous intéressent et pour qui nous pourrons donner une datation intéressante. Notre manuscrit se concentrera dans un premier temps sur les datations sans fixer les instants et nous aurons besoin de construire des datations au plus tôt indépendamment de leur réactivité.

6.1.33 *Remarque* **Construction générale de T_{canon}**
Si le réseau n'est pas interactif, il peut produire un flot infini à partir d'un flot fini, la définition des dates de sorties donnée en équation (6.5) doit être changée par :

$$T^k_{canon((R^k,\mathbf{X}_k,\mathbf{Y}_k)_k)}(s^{(j)}) \stackrel{\text{def}}{=} \left\{ \sup(R^k) - \frac{|R^k|}{2u} \;\middle|\; 1 \leq u \leq |\mathbf{Y}^{(j)}_{\delta k}| \right\}$$

Cette construction générale, bien que possiblement non réactive, préserve la propriété 6.1.31.

6.1.6 Contraintes temps réel et hypothèse synchrone

Quand nous parlons de temps, nous ne parlons pas du temps physique absolu, mais d'un temps abstrait et relatif bien que global. Le lien avec le temps physique est donné par l'hypothèse que nous faisons qui est l'*hypothèse synchrone* usuelle. La voici écrite dans notre contexte :

6.1.34 *Définition* **Hypothèse synchrone**
L'hypothèse synchrone considère que les calculs et communications, nécessaires à l'exécution d'une réaction, prennent moins de temps que la durée de la réaction.

La définition d'une réaction stipule que le temps pris par la réaction est borné, et ce de manière uniforme (cf. hypothèse 6.1.8). Mais l'unité de temps n'est pas définie, ce qui rend cette notion de borne relative et tout est par défaut à une homothétie près.

6.1.35 *Remarque* Si l'uniformité n'est pas nécessaire, tout ce que nous faisons peut être considéré à un automorphisme d'ordre près, c'est-à-dire que l'on peut appliquer toute déformation du temps respectant l'ordre entre les dates.

La vérification de l'hypothèse synchrone s'effectue *a posteriori* une fois le lien entre temps abstrait et temps physique déterminé. Toutefois, il est possible d'affirmer a priori qu'il est impossible de manipuler une infinité de valeurs durant un instant, ce qui est assuré pour les entrées et les sorties par la réactivité de la datation.

6.1.36 *Hypothèse* **Hypothèse synchrone et datation**
Dans tout notre travail, nous supposons que l'hypothèse synchrone pourra être vérifiée a posteriori dès qu'une datation est uniformément réactive.

6.2 Exécution équitable co-itérative

Nous avons souligné que les datations au plus tôt sont celles qui nous intéressent, cependant, même si le réseau considéré est Lipschitz, nous avons vu que nous ne pouvions pas borner uniformément les itérations nécessaires à une réaction. Ceci va de pair avec le constat qu'un calcul de point fixe est mathématiquement commode, mais en pratique inefficace.

La représentation co-récursive du calcul est la plus naturelle et celle couramment utilisée pour donner la sémantique dénotationnelle des langages comme Lucid Synchrone et Heptagon (cf. section 2.5). Nous présentons donc celle-ci, puis partons vers les systèmes de transition étiquetés, pour éclairer ce saut effectué par la compilation vers un « automate de Mealy ». L'idée sous-jacente intuitive est de faire coincider un appel à une fonction de transition avec une réaction du système.

La double addition Rétrospectivement, les trois descriptions de l'exemple 6.1.20 ne sont guère convaincantes. Elles recalculent l'intégralité de leur sortie, même si un préfixe de celle-ci a déjà été calculé et donné en argument à la fonction. Ceci rend le calcul récursif « inutile ». Or notre compilation a pour but de produire une machine à états, calculant les *nouvelles* sorties à partir des *nouvelles* entrées et de son état.

Pour la double addition, comme le calcul est point à point, la machine pourra être minimaliste et sans état, fondamentalement représenté par une fonction de transition scalaire $tF_{dadd}(x1, x2, x3) = x1 + x2 + x3$, qui serait appelée à chaque instant s'il y a des valeurs sur chacune des entrées.

Le registre initialisé L'utilisation d'un état peut rendre le calcul beaucoup plus efficace. L'exemple typique est l'opérateur `0 fby x`. Quand nous avions donné sa sémantique, nous l'avions fait sous une forme très compacte : $fby(\mathbf{x}) = 0.\mathbf{x}$.

Le système pouvant alors s'écrire $f_{\text{fby}}(\mathbf{x}, \_, \mathbf{y}) = (\mathbf{x}, \_, fby(\mathbf{x}))$. À chaque fois que l'entrée grossit, le calcul du point fixe recalcule l'intégralité de la sortie. Alors qu'il serait possible de calculer au fur et à mesure avec l'aide d'une mémoire, similairement à ce que nous avions fait pour la sémantique synchrone d'Heptagon :

$$fby(\mathbf{x}) = rfby(0, \mathbf{x}) \qquad rfby(v, v'.\mathbf{x}) = v.rfby(v', \mathbf{x}) \qquad rfby(v, \varepsilon) = v$$

Le premier argument de $rfby$ est l'*état* de la machine, initialisé à 0. Chaque appel récursif de $rfby$ peut être vu comme une réaction, fournissant une valeur de sortie, consommant une valeur de l'entrée et changeant son état.

Co-récursive Donnons le cadre général de l'écriture co-récursive, puisque c'est elle qui nous semble la plus naturelle comme point de départ :

6.2.1 *Définition* **Description co-récursive**
Le réseau $F : \mathcal{D}_E \to \mathcal{D}_S$ peut se décrire avec une fonction co-récursive $rF : (\mathcal{S}, \mathcal{D}_E) \to (\mathcal{D}_S)$. Son premier argument est l'état du système, l'espace des états $\mathcal{S}$ contient un état distingué $0_{\mathcal{S}} \in \mathcal{S}$ appelé état initial. La sémantique du réseau est alors $F(\mathbf{X}) = rF(0_{\mathcal{S}}, \mathbf{X})$, avec :

$$\forall \mathbf{X} \in \mathcal{D}_E, \forall s \in \mathcal{S}, \quad rF(s, \mathbf{X}) = \mathbf{Y}_\delta.rF(s', \mathbf{X}') \quad \text{avec } s' \in \mathcal{S}, \mathbf{X} = \mathbf{X}_\delta.\mathbf{X}' \text{ et } |\mathbf{X}_\delta| < \infty$$

Cette description n'est correcte que si rF est continue en sont deuxième argument.

6.2.2 *Remarque* **Liberté du choix de l'ensemble des états**
L'état est beaucoup plus libre que l'utilisation de flots intermédiaires comme l'on peut avoir en décrivant un réseau de Kahn. En effet, rien ne requiert que l'ensemble des états soit partiellement ordonné.

6.2.1 Représentation co-itérative

La représentation co-récursive n'est toujours pas calculatoire. Seule la contrainte de continuité empêche la fonction de prendre des décisions à partir d'une quantité infinie de valeurs. La compilation que nous avons donnée pour Heptagon produit une fonction `step` qui ne prend en argument que son état et les entrées qu'elle consomme ($\mathbf{X}_\delta$) et retourne de nouvelles valeurs de sorties et un nouvel état. C'est ce que nous appelons à la suite de Caspi et Pouzet [CP98] la représentation co-itérative du réseau.

La description co-itérative du réseau sépare le comportement du réseau (un système de transition étiquetées) de son exécution qui choisit quelles sont les valeurs des entrées à consommer et active la fonction de transition avec.

6.2.3 *Définition* **Description co-itérative**
La description co-itérative est la donnée d'une fonction de transition tF *partielle*, de type $\mathcal{S} \times \mathcal{A}(\mathcal{D}_E) \to \mathcal{A}(\mathcal{D}_S) \times \mathcal{S}$. La transition $tF(s, \mathbf{X}) = (\mathbf{Y}, s')$ est usuellement représentée $s \xrightarrow{\mathbf{X}/\mathbf{Y}} s'$.

6.2.4 *Remarque* Dans [CP98], la fonction de transition est de type $\mathcal{S} \times X \to Y \times \mathcal{S}$, requérant une valeur pour toutes ses entrées $X = \{\mathbf{X} \mid \mathbf{X} \in \mathcal{D}_E, \forall i, \left|\mathbf{X}^{(i)}\right| = 1\}$ et produisant une valeur pour toutes les sorties $Y = \{\mathbf{Y} \mid \mathbf{Y} \in \mathcal{D}_S, \forall i, \left|\mathbf{Y}^{(i)}\right| = 1\}$. Cependant, cela restreint l'ensemble des fonctions aux fonctions totales qui préservent les longueurs quand nous souhaitons pouvoir représenter toutes les fonctions.

6.2.5 *Définition* **Exécution non déterministe co-itérative**
Un réseau décrit par une fonction de transition tF, s'exécute grâce à $\mathrm{run}(tF)$, en charge de consommer petit à petit les flots d'entrées et de produire ceux des sorties. Cette exécution non déterministe est donnée par :

Si $\nexists \mathbf{X}_\delta \sqsubseteq \mathbf{X}, (s, \mathbf{X}_\delta) \in \mathrm{dom}(tF)$	alors $\mathrm{run}(tF)(s, \mathbf{X}) = \boldsymbol{\varepsilon}$
Sinon, soit $\mathbf{X} = \mathbf{X}_\delta.\mathbf{X}'$ et $tF(s, \mathbf{X}_\delta) = (\mathbf{Y}_\delta, s')$	alors $\mathrm{run}(tF)(s, \mathbf{X}) = \mathbf{Y}_\delta.\mathrm{run}(tF)(s', \mathbf{X}')$

C'est le possible choix de $\mathbf{X}_\delta$ qui rend l'exécution non déterministe.

La particularité de cette exécution tient à l'alphabet des entrées/sorties qui est composé de suites finies (mots). Ceci n'est pas tout à fait habituel pour un système de transition étiquetées, bien que l'on puisse le trouver aussi pour modéliser les réseaux flots de données [Jon89]. À chaque transition, le système consomme un préfixe fini de ses entrées pour produire de nouvelles sorties concaténées aux anciennes.

Le but de cette exécution est de représenter le calcul du réseau, c'est-à-dire de pouvoir poser $F(\mathbf{X}) = \mathrm{run}(tF)(0_{\mathcal{S}}, \mathbf{X})$. Pour cela, nous devons être en mesure d'assurer que l'exécution est une *fonction* continue.

Si les domaines étaient composés uniquement de flots finis, seule la propriété classique de confluence suffirait. Des flots infinis, que ce soit en entrée ou en sortie font apparaitre des problèmes d'équités de l'exécution.

Donnons quelques définitions et rappelons la propriété de confluence avant d'illustrer les problèmes d'équité.

6.2.6 *Définition* **Suites de transition**
Une suite de transition à partir de l'état s_0 avec l'entrée $\mathbf{D}$, est la donnée de la suite $(s_{k-1}, \mathbf{X}_{\delta k}, \mathbf{Y}_{\delta k}, s_k)_k$ avec $tF(s_{k-1}, \mathbf{X}_{\delta k}) = (\mathbf{Y}_{\delta k}, s_k)$ telle que la chaîne des entrées consommées $(\mathbf{X}_k)_k$ est inclue dans $\mathbf{D}$. Cette suite est représentée : $s_0 \xrightarrow{\mathbf{X}_{\delta 1}/\mathbf{Y}_{\delta 1}} s_1 \xrightarrow{\mathbf{X}_{\delta 2}/\mathbf{Y}_{\delta 2}} s_2 \cdots$

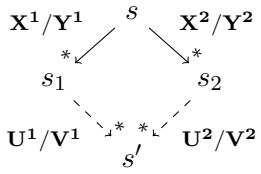
La fermeture réflexive et transitive de cette relation est notée avec une étoile : $s \xrightarrow{\mathbf{X}'/\mathbf{Y}'}{}^* s'$, cela signifie, soit que $s = s'$ avec $\mathbf{X}' = \boldsymbol{\varepsilon}$ et $\mathbf{Y}' = \boldsymbol{\varepsilon}$, soit qu'il existe une suite finie de n transitions $(s_{k-1}, \mathbf{X}_{\delta k}, \mathbf{Y}_{\delta k}, s_k)_k$ avec $tF(s_{k-1}, \mathbf{X}_{\delta k}) = (\mathbf{Y}_{\delta k}, s_k)$, $s_0 = s$ et $\mathbf{X}_n = \mathbf{X}'$, $\mathbf{Y}_n = \mathbf{Y}'$.

Une suite infinie de transition définit deux chaînes $(\mathbf{X}_k)_k$ et $(\mathbf{Y}_k)_k$ de limite $\mathbf{X}$ et $\mathbf{Y}$, nous noterons une telle chaîne $s_0 \xrightarrow{\mathbf{X}/\mathbf{Y}}{}^\omega$

Une suite de transition est dite terminale pour l'entrée $\mathbf{D}$ ssi elle est de longueur infinie ou bien s'il n'est pas possible de l'agrandir : $s \xrightarrow{\mathbf{X/Y}}{}^{*} s'$ est terminale s'il n'existe pas $\mathbf{X}_\delta$ tel que $\mathbf{X}.\mathbf{X}_\delta \sqsubseteq \mathbf{D}$ et $s' \xrightarrow{\mathbf{X}_\delta/\mathbf{Y}_\delta} s''$).

6.2.7 *Définition* **Confluence**
La fonction de transition tF est dite confluente ssi pour tout état s_0 et toute paire de suites de transition de tailles finies $s_0 \xrightarrow{\mathbf{X^i/Y^i}}{}^{*} s_i$ avec $i = 1, 2$, compatibles ($\mathbf{X^1} \uparrow \mathbf{X^2}$), il est possible de les compléter pour arriver dans un même état, en ayant consommé et produit les même flots en un nombre fini de transitions.
C'est-à-dire qu'il existe s' et $s_i \xrightarrow{\mathbf{U^i/V^i}}{}^{*} s'$ telles que :

$$\mathbf{X^1}.\mathbf{U^1} = \mathbf{X^2}.\mathbf{U^2} = \mathbf{X^1} \sqcup \mathbf{X^2} \qquad \mathbf{Y^1}.\mathbf{V^1} = \mathbf{Y^2}.\mathbf{V^2}$$

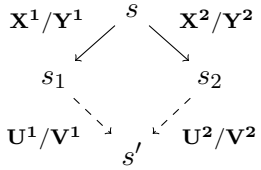


Les flèches en dur sont les hypothèses, celles en tirets sont les conséquences de la propriété.

La confluence d'une fonction de transition peut être complexe à prouver. La version forte usuelle est plus pratique.

6.2.8 *Définition* **Confluence forte**
La confluence forte est la confluence en un pas. Pour tout état s_0 et toute paire de transitions $s_0 \xrightarrow{\mathbf{X^i/Y^i}} s_i$ avec $i = 1, 2$, compatibles ($\mathbf{X^1} \uparrow \mathbf{X^2}$),
il existe s' et $s_i \xrightarrow{\mathbf{U^i/V^i}} s'$ telles que :

$$\mathbf{X^1}.\mathbf{U^1} = \mathbf{X^2}.\mathbf{U^2} = \mathbf{X^1} \sqcup \mathbf{X^2} \qquad \mathbf{Y^1}.\mathbf{V^1} = \mathbf{Y^2}.\mathbf{V^2}$$

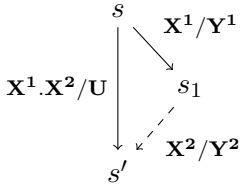


Dans la pratique, nous aurons souvent un domaine de définition clôt par unions compatibles. Dans ce cas, la confluence forte est simplifiée. Il suffit de vérifier que toute transition consommant plus d'entrée produit plus de sorties.

6.2.9 *Définition* **Confluence très forte**
La confluence très forte est la combinaison de la clôture du domaine de définition par union des éléments compatibles :

$$\forall s, \forall \mathbf{X} \uparrow \mathbf{X}', \quad (s, \mathbf{X}) \in \mathrm{dom}(tF) \wedge (s, \mathbf{X}') \in \mathrm{dom}(tF) \implies (s, \mathbf{X} \sqcup \mathbf{X}') \in \mathrm{dom}(tF)$$

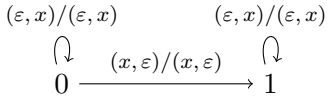
et la confluence en demi-diamant, avec $\mathbf{U} = \mathbf{Y^1}.\mathbf{Y^2}$:



6.2.10 *Exemple* **Problèmes d'équité**
Considérons le réseau F_{eqout}, défini par $F_{eqout}(\varepsilon, \mathbf{x_2}) = (\varepsilon, \mathbf{x_2})$ et $F_{eqout}(x.\mathbf{x_1}, \mathbf{x_2}) = (x, \mathbf{x_2})$, intuitivement, nous aimerions écrire la fonction de transition confluente suivante :

$$tF_{eqout}(0, (x, \varepsilon)) = ((x, \varepsilon), 1) \qquad tF_{eqout}(i, (\varepsilon, x)) = ((\varepsilon, x), i) \text{ avec } i = 0, 1$$

Ce qui donne visuellement :

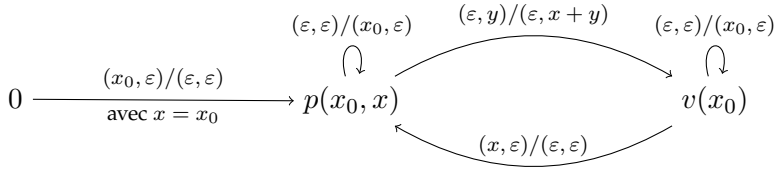


Dans l'état 0, elle n'a pas encore produit la tête de sa première entrée, contrairement à dans l'état 1. Si l'entrée $\mathbf{x_2}$ est infinie, alors la suite de transition $(0, (\varepsilon, \mathbf{x_2}[k]), (\varepsilon, \mathbf{x_2}[k]), 0)_k$ est une exécution possible donnant la sortie $(\varepsilon, \mathbf{x_2})$. Mais toute autre exécution effectuant une fois la transition pour aller dans l'état 1 donne la sortie $(x, \mathbf{x_2})$.

Ce problème est un problème d'équité entre le tirage des transitions. Cela est la manifestation des questions d'ordonnancement équitable des réseaux de Kahn sous-jacent (cf. remarque 6.1.19). Ici, le réseau sous-jacent pourrait être composé de deux nœuds indépendants, le premier calculant la première sortie à partir de la première entrée et le second pour les secondes. ne pas prendre la transition allant de l'état 0 à l'état 1 correspond à ne jamais activer le premier nœud.

Les problèmes d'équité ne concernent pas uniquement des sorties dont les comportements sont « orthogonaux » comme précédemment. Considérons la fonction F_{eqin} telle que la première sortie est la répétition de la première valeur de la première entrée et la seconde sortie est la somme des deux entrées : $F_{eqin}(\varepsilon, \varepsilon) = (\varepsilon, \varepsilon)$ et $F_{eqin}(x.\mathbf{x_1}, \mathbf{x_2}) = (x^\omega, plus(\mathbf{x_1}, \mathbf{x_2}))$. Nous pouvons écrire une fonction de transition fortement confluente à trois états, l'initial 0 qui attend une valeur sur la première entrée sans quoi rien ne peut être fait. L'état plein $p(x, x')$ qui indique que la première valeur de la première entrée était x et que nous avons lue la valeur x' pour l'addition sans l'utiliser, l'état vide $v(x)$ qui n'a pas de valeur prête pour l'addition :

$$
\begin{aligned}
tF_{eqin}(0, (x_0, \varepsilon)) &= ((\varepsilon, \varepsilon), p(x_0, x_0)) \\
tF_{eqin}(p(x_0, x), (\varepsilon, \varepsilon)) &= ((x_0, \varepsilon), p(x_0, x)) \qquad tF_{eqin}(p(x_0, x), (\varepsilon, y)) = ((\varepsilon, x + y), v(x_0)) \\
tF_{eqin}(v(x_0), (\varepsilon, \varepsilon)) &= ((x_0, \varepsilon), v(x_0)) \qquad tF_{eqin}(v(x_0), (x, \varepsilon)) = ((\varepsilon, \varepsilon), p(x, x'))
\end{aligned}
$$



Il est possible de rester dans les états pleins ou vides une infinité de fois sans jamais produire de valeur pour la deuxième sortie. Par exemple $\mathrm{run}(tF_{eqin})(0, (42, 7))$ admet deux résultats, $(42^\omega, \varepsilon)$ et $(42^\omega, 49)$. Les possibilités ne faisant qu'empirer avec des entrées plus grandes. Avec des entrées infinies, il est aussi possible de ne jamais rien produire pour la première sortie.

6.2.1.1 Exécutions équitables

Une solution classique [Pan95 ; GB03] est de restreindre les exécutions possibles avec l'hypothèse que l'ordonnancement est équitable.

6.2.11 *Définition* **Suite de transition équitable**
La suite de transition $(s_{k-1}, \mathbf{X}_{\delta k}, \mathbf{Y}_{\delta k}, s_k)_k$ de tF est dite équitable ssi les composantes des sorties sont finies par obligation et non par choix des transitions. Les transitions ne peuvent pas passer par une infinité d'états avec une transition produisant sur la $i$ème sortie sans jamais

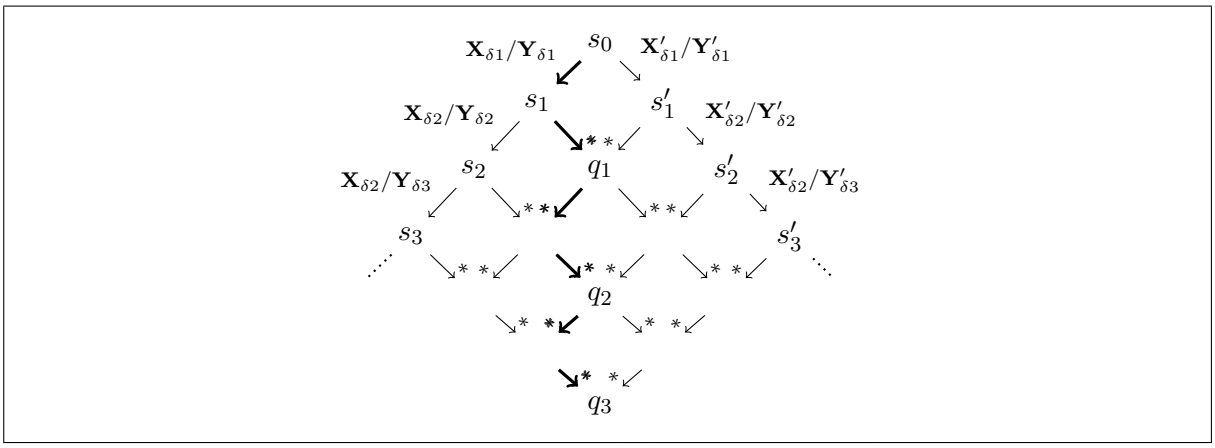


FIGURE 6.1: Construction, en gras, de l'exécution équitable maximale

produire sur cette sortie :

$$\forall i, \exists m \in \mathbb{N}, \quad (\forall k \geq m, \mathbf{Y}_{\delta k}^{(i)} = \varepsilon) \implies \#\Big\{k \;\Big|\; \exists \mathbf{X}, \mathbf{X}_k.\mathbf{X} \sqsubseteq \mathbf{D}, s_k \xrightarrow{\mathbf{X}/\mathbf{Y}}{}^{*} s', \mathbf{Y}^{(i)} \neq \varepsilon\Big\} < \infty$$

6.2.12 *Remarque* Il est d'usage ([Pan95 ; GB03]) de ne pas différencier l'équité de consommation et de production. Celle de consommation ne nous est pas utile, mais s'il fallait l'introduire :

$$\begin{aligned}\forall i, \exists m \in \mathbb{N}, \quad & (\forall k \geq m, \mathbf{X}_{\delta k}^{(i)} = \varepsilon) \\ & \implies \#\Big\{k \mid \exists \mathbf{X}, \mathbf{X}_k.\mathbf{X} \sqsubseteq \mathbf{D}, \mathbf{X}^{(i)} \neq \varepsilon, (s_k, \mathbf{X}) \in \mathrm{dom}(tF)\Big\} < \infty\end{aligned}$$

Se restreindre aux exécutions équitables rend $\mathrm{run}(tF)$ continu si tF conflue. La preuve se retrouve dans de multiples contextes. Le plus proche est probablement celui des automates de ports de [Pan95]. Pour comprendre l'idée, prouvons dans un premier temps que l'exécution devient bien une fonction :

Lemme 6.2.13 (Exécution équitable : $\mathrm{run}(tF)$ est une fonction)**.** *Si tF est confluente, quelque soit la suite terminale et équitable de transition choisie par* run*, le résultat $\mathrm{run}(tF)(s_0, \mathbf{D})$ sera le même.*

Démonstration Soient deux suites de transition terminales et équitables à partir de s_0 pour l'entrée $\mathbf{D}$, trois cas sont à étudier. Dans le premier, les deux suites sont finies, $s_0 \xrightarrow{\mathbf{X}/\mathbf{Y}}{}^{*} s$ et $s_0 \xrightarrow{\mathbf{X}'/\mathbf{Y}'}{}^{*} s'$, par définition nous avons $\mathbf{X} \sqsubseteq \mathbf{D}$ et $\mathbf{X}' \sqsubseteq \mathbf{D}$. Nous tombons dans le cas classique de la confluence qui permet de conclure que $\mathbf{Y} = \mathbf{Y}'$. Dans le deuxième cas, une des deux suites est infinie. Prenons la suite équitable infinie $s_0 \xrightarrow{\mathbf{X}/\mathbf{Y}}{}^{\omega}$ décrite par $(s_{k-1}, \mathbf{X}_{\delta k}, \mathbf{Y}_{\delta k}, s_k)_k$ et la suite $s_0 \xrightarrow{\mathbf{X}'/\mathbf{Y}'}{}^{*} s'$. La confluence assure pour tout k l'existence de $s_k \xrightarrow{\mathbf{U}_k/\mathbf{V}_k}{}^{*} s'$ avec $\mathbf{Y}' = \mathbf{Y}_k.\mathbf{V}_k$.

Ainsi $\mathbf{Y}_k \sqsubseteq \mathbf{Y}'$ et par passage à la limite : $\mathbf{Y} \sqsubseteq \mathbf{Y}'$. De plus, si $\mathbf{Y}$ est strictement inclus dans $\mathbf{Y}'$, les $\mathbf{Y}_k$ le sont aussi : il y a une composante i telle que à partir d'un rang m, $\mathbf{Y}_{\delta k}^{(i)} = \varepsilon$ tandis que $\mathbf{V}_k^{(i)} \neq \varepsilon$, ce qui contredit l'équité. Dans le dernier cas, les deux suites sont infinies. Par la confluence, nous construisons (cf. figure 6.1) grâce à la confluence une suite infinie de transition *passant* par les états q_k telle que $s_0 \xrightarrow{\mathbf{W}_k/\mathbf{Z}_k}{}^{*} q_k$ avec $\mathbf{W}_k = \mathbf{X}_k \sqcup \mathbf{X}'_k$ et $\mathbf{Y}_k \sqsubseteq \mathbf{Z}_k$ et

$\mathbf{Y}'_k \sqsubseteq \mathbf{Z}_k$. Par construction nous avons $\mathbf{Y} \sqsubseteq \mathbf{Z}$. Réciproquement $\mathbf{Z}_k$ est inclus dans $\mathbf{Y}$ par un raisonnement identique au deuxième cas.

De la même manière, nous pouvons conclure la continuité.

6.2.14 Théorème **Continuité de** $\mathrm{run}(tF)$
Sous réserve de se restreindre aux exécutions équitables, la fonction $F(\mathbf{X}) = \mathrm{run}(tF)(0_S, \mathbf{X})$ *est continue, pour une fonction de transition* tF *confluente.*

Démonstration Monotonie et continuité se prouvent en utilisant la construction que nous avons faite pour démontrer le troisième cas dans la preuve du lemme 6.2.13 et le même raisonnement utilisant l'équité pour prouver le sens d'inclusion non trivial.

6.2.1.2 Discussion

Bien que nous arrivions à nos fins, la propriété d'équité de l'exécution n'est pas du tout évidente à assurer en pratique, ni à manipuler mathématiquement et ne nous satisfait pas, bien quelle soit viable. La propriété d'équité a été progressivement délaissée, la dernière apparition est due à Basten [GB03] sa définition est dans la même veine, alliant *maximality* et *fairness* pour obtenir les même garanties que nous avons avec les suites de transition *terminales* et *équitables*.

L'ordonnancement équitable a été au cœur de nombreuses théories concernant les systèmes concurrents, comme le soulignait déjà A. Pnueli [Pnu81], les réseaux de Kahn en étant un cas particulier. La majeure partie des recherches se sont concentrées sur la définition d'un modèle complètement adéquat (fully abstract) pour les réseaux flots de données non déterministes, en particulier avec l'opérateur « fair merge » qui fusionne deux flots de manière non déterministe, avec la contrainte de consommer une infinité de fois chacune des entrées. On y retrouve de nombreux modèles et discussions [Par80 ; Fau82 ; Kok87 ; Jon89], utiles même dans le cadre déterministe. Ce dernier porte souvent le nom de flots de données « pur ». Toutes ces recherches ont besoin, sous une forme où une autre, de manipuler une hypothèse d'ordonnancement équitable similaire à celle que nous avons présentée en définition 6.2.11.

Ce besoin de contraindre l'ordonnancement, par une propriété au-dessus/hors du modèle est la motivation principale de l'invention des CTS (*concurrent transition systems*) par E. Stark [Sta89]. Dans les CTS, la concurrence est partie intégrante de la description du système de transition, elle permet de définir un pré-ordre sur les exécutions. Les exécutions équitables sont remplacées par la classe d'équivalence maximale des exécutions, engendrée par la construction de l'idéal incluant toutes les exécutions finies. Les mathématiques résultantes sont très belles et permettent [PSS90] de faire le lien avec les fonction stables et séquentielles, comme nous allons le faire. Elles font même plus puisque le non-déterminisme y est convenablement traité [Pan95].

Cependant, cette mathématisation perd la proximité qui nous intéresse entre la sémantique opérationnelle du système de transition et la compilation sous forme de fonctions de transition, ce qui est d'autant plus visible que ces travaux ont ouverts la voie aux descriptions catégoriques (cf. [HPW04] une version revisitée de la thèse de Hildebrandt de 1999). Ce même reproche, doit être fait aux systèmes de réécritures orthogonaux [Klo90 ; HL91b ; HL91a], capables de traiter les réseaux de Kahn.

6.3 Exécution et représentation co-itérative étendue

Nous désirons rester dans le cadre simple d'un système de transition, sans avoir recours à des équivalences d'exécutions où à une hypothèse d'équité comme celle que nous avons utilisée tantôt. Dans une vision réactive, la seule chose qui nous importe vraiment sont les exécutions finies. En effet, le rôle de run est de représenter l'environnement qui active le programme avec de nouvelles entrées pour obtenir de nouvelles sorties. Le besoin mathématique de définir convenablement un comportement infini ne devrait pas nous bloquer.

Observons plus en détail les deux exemples qui nous avaient permis d'illustrer les problèmes d'équilibres (cf. exemple 6.2.10) :

— Le réseau F_{eqout} est interactif : pour une entrée finie, il va exécuter un nombre fini de transitions et produire une sortie finie. Ceci nous permet de caractériser le comportement avec des entrées infinies sans avoir à considérer l'équité :
Soit l'entrée $\mathbf{D} = (x, \mathbf{x_2})$ infinie, nous pouvons prendre une chaîne $(\mathbf{X}_k)_k$ approximant $\mathbf{D}$, la continuité de l'exécution équilibrée nous permet d'affirmer que $\mathrm{run}(tF)(0, \mathbf{D}) = \lim_k \mathrm{run}(tF)(0, \mathbf{X}_k)$ or puisque le réseau est réactif, chacun des calculs $\mathrm{run}(tF)(0, \mathbf{X}_k)$ requiert un nombre de transitions fini et nécessite uniquement la confluence de tF et non l'équité de l'exécution.

— À l'opposé, le réseau F_{eqin} n'est pas interactif et, pour toute entrée finie avec au moins une valeur en première composante, le calcul nécessite une infinité de transitions.

Le point d'achoppement, n'est pas de gérer des entrées infinies, mais l'existence de réseaux qui ne sont pas interactifs et qui nous forcent, à partir d'entrées finies, à considérer des exécutions infinies. Remarquez que c'est exactement cette difficulté qui compliquait la recherche d'une datation réactive. Pour autant, les réseaux non interactifs sont importants, ils recouvrent les constantes, mais aussi les réseaux dont l'initialisation n'est pas statiquement connue (comme pour F_{eqin}). Nous ne pouvons les abandonner.

Notre solution consiste à interdire les systèmes de transition ayant des exécutions infinies pour une entrée finie, mais en contrepartie, la fonction de transition est capable de produire des flots infinis. L'idée est assez simple, pour les réseaux interactifs, la fonction de transition correspond à nos attentes. Pour les autres, un travail supplémentaire est à fournir à *l'implémentation*, pour étaler la sortie infinie sur une infinité de transitions.

Nous traiterons ce dernier point par la suite, ce qui nous semble judicieux, car les réseaux non interactifs requièrent une gestion ad-hoc quelle que soit l'implémentation. En refusant de faire ce choix au niveau sémantique, nous rendons notre système de transition homogène et fortement normalisant, donc dans un certain sens beaucoup plus classique.

La simplicité des notations et de la manipulation d'une fonction de transition produisant des flots infinis tient à l'utilisation de la concaténation étendue. La concaténation $\mathbf{X}.\mathbf{Y}$ est naturellement définie si $\mathbf{Y}$ est infini, mais elle l'est tout autant si $\mathbf{X}$ l'est :

6.3.1 *Définition* **Concaténation étendue**
Nous étendons la concaténation avec un flot infini comme membre gauche :

$$\forall \mathbf{X} \in \mathcal{D}, \mathbf{Y} \in \mathcal{D}, i, k, \quad (\mathbf{X}.\mathbf{Y})^{(i)}[k] = \begin{cases} \mathbf{X}^{(i)}[k] & \text{si } k \leq \left|\mathbf{X}^{(i)}\right| \\ \mathbf{Y}^{(i)}\left[k - \left|\mathbf{X}^{(i)}\right|\right] & \text{sinon} \end{cases}$$

La subtilité induite concerne le calcule des résidus qui n'est unique que si le préfixe considéré est fini :

$$\mathbf{X}.\mathbf{Y} = \mathbf{X}.\mathbf{Z} \qquad \Longleftrightarrow \qquad \forall i, \quad \left|\mathbf{X}^{(i)}\right| = \infty \quad \vee \quad \mathbf{Y}^{(i)} = \mathbf{Z}^{(i)}$$

Pour retomber sur nos pieds nous n'allons considérer que les plus petits résidus :

6.3.2 *Propriété* **Plus petit résidu**
Pour tout $\mathbf{X}$ et $\mathbf{Z}$, le plus petit résidu $\mathbf{Y}$ tel que $\mathbf{X}.\mathbf{Y} = \mathbf{Z}$ est celui dont la $i$ème composante vaut ε si la $i$ème de $\mathbf{X}$ est infinie.

6.3.3 *Nota bene* **Concaténation étendue et plus petit résidu**
Nous utilisons la concaténation étendue dans le reste du manuscrit. Si un flot $\mathbf{R}$ est défini par une équation $\mathbf{Y} = \mathbf{X}.\mathbf{R}$, il est implicitement le plus petit résidu de telle sorte que l'utilisation de la concaténation étendue ne pose pas de nouvelles difficultés.

6.3.1 Adapter les définitions

Les définitions des suites de transition, terminales ou pas, sont inchangées, tout comme celles de confluence et de confluence très forte. Le premier changement à apporter à nos définitions concerne le type de la fonction de transition :

6.3.4 *Définition* **Description co-itérative étendue**
La description co-itérative est la donnée d'une fonction de transition *partielle* de type :

$$tF : \mathcal{S} \times \mathcal{A}(\mathcal{D}_E) \to \mathcal{D}_S \times \mathcal{S}$$

Ensuite, au lieu de considérer les exécutions équitables, nous demanderons que le calcul à partir d'une entrée finie s'effectue en un nombre fini de transitions :

6.3.5 *Définition* **Forte normalisation de tF sur $\mathcal{A}(\mathcal{D}_E)$**
Nous dirons que tF est fortement normalisante sur $\mathcal{A}(\mathcal{D}_E)$ ssi, toute suite de transition consommant une entrée finie est de longueur finie.

L'exécution du système pour une entrée finie est alors bien définie :

6.3.6 *Propriété*
Confluence et forte normalisation de tF sur $\mathcal{A}(\mathcal{D})$ permettent d'affirmer que $\lambda\mathbf{X}.\mathrm{run}(tF)(0_\mathcal{S}, \mathbf{X})$ est une fonction monotone sur $\mathcal{A}(\mathcal{D})$.

La monotonie de $\mathrm{run}(tF)$ sur les approximants permet (cf. [SLG94]) de l'étendre de manière continue pour définir $\mathrm{exec}(tF) : \mathcal{D}_E \to \mathcal{D}_S$ avec

$$\mathrm{exec}(tF)(\mathbf{X}) = \begin{cases} \mathrm{run}(tF)(0_\mathcal{S}, \mathbf{X}) & \text{si } |\mathbf{X}| < \infty \\ \bigsqcup_{\mathbf{A} \in \mathcal{A}(\mathbf{X})} \mathrm{run}(tF)(0_\mathcal{S}, \mathbf{A}) & \text{sinon} \end{cases}$$

En particulier, nous avons, pour toute chaîne d'éléments finis $(\mathbf{X}_k)_k$ de limite $\mathbf{X}$:

$$\mathrm{exec}(tF)(\mathbf{X}) = \bigsqcup_{\mathbf{A} \in \mathcal{A}(\mathbf{X})} \mathrm{run}(tF)(0_\mathcal{S}, \mathbf{A}) = \lim_{k \to \infty} \mathrm{run}(tF)(0_\mathcal{S}, \mathbf{X}_k) = \lim_{k \to \infty} \mathrm{exec}(tF)(\mathbf{X}_k)$$

Grâce à ceci, nous n'avons pas besoin de construire explicitement exec. Par contre, nous devons nous assurer que l'environnement fournit les entrées morceaux finis par morceaux finis. Ceci coïncide avec le comportement attendu. Pour l'entrée $\mathbf{X}$, l'environnement la découpe en une suite de morceaux finis $(\mathbf{X}_{\delta k})_k$, dont la chaîne est de limite $\mathbf{X}$. L'exécution réactive doit consommer cette entrée et fournir les résultats au fur et à mesure, ce qu'elle peut faire sans recalculer $\mathrm{run}(tF)(0, \mathbf{X}_k)$ en entier à chaque fois :

6.3.7 *Définition* **Exécution à grands pas**
L'exécution à grands pas, d'une fonction de transition tF fortement normalisante sur $\mathcal{A}(\mathcal{D}_E)$, prend une suite de morceaux d'entrées $(\mathbf{X}_{\delta k})_k$ et produit une suite de morceaux de sorties $(\mathbf{Y}_{\delta k})_k$ définie par :

$$(s_k, \mathbf{T}_k, \mathbf{Y}_{\delta k}) = \mathrm{bigstep}(s_{k-1}, \mathbf{T}_{k-1}.\mathbf{X}_{\delta k}, \boldsymbol{\varepsilon}) \quad \text{avec } s_0 = 0_{\mathcal{S}}, \text{ et } \mathbf{T}_0 = \boldsymbol{\varepsilon}$$

$$\mathrm{bigstep} : \mathcal{S} \times \mathcal{A}(\mathcal{D}_E) \times \mathcal{D}_S \to \mathcal{S} \times \mathcal{A}(\mathcal{D}_E) \times \mathcal{D}_S$$

$$\mathrm{bigstep}(s, \mathbf{X}, \mathbf{Y}) = \begin{cases} \mathrm{bigstep}(s', \mathbf{X}', \mathbf{Y}.\mathbf{Y}_\delta) & \text{si } \mathbf{X} = \mathbf{X}_\delta.\mathbf{X}' \wedge s \xrightarrow{\mathbf{X}_\delta/\mathbf{Y}_\delta} s' \\ (s, \mathbf{X}, \mathbf{Y}) & \text{sinon} \end{cases}$$

Pour un flot $\mathbf{X}$ fini, l'équation $(s', \mathbf{T}, \mathbf{Y}) = \mathrm{bigstep}(s, \mathbf{X}, \boldsymbol{\varepsilon})$ calcule une suite *terminale* de transition partant de s avec l'entrée $\mathbf{X}$. Puisque la fonction de transition est fortement normalisante, cette suite de transition est finie : $s \xrightarrow{\mathbf{X}'/\mathbf{Y}}{}^* s'$ avec $\mathbf{X}'$ les entrées consommées et $\mathbf{T}$ celles non consommées ($\mathbf{X}'.\mathbf{T} = \mathbf{X}$). $\mathbf{T}$ sert donc de file d'attente pour les entrées reçues mais pas consommées.

Il reste à prouver que cette exécution calcule ce que nous attendons :

6.3.8 *Propriété* **L'exécution à grands pas calcule** exec
Pour toute entrée $\mathbf{X}$ approximée par la chaîne $(\mathbf{X}_k)_k$, la chaîne $(\mathbf{Y}_k)_k$ associée par l'exécution à grands pas est telle que $\mathbf{Y}_k = \mathrm{run}(tF)(0_{\mathcal{S}}, \mathbf{X}_k)$.

Démonstration L'exécution à grands pas construit la suite de transition :

$$0_{\mathcal{S}} \xrightarrow{\mathbf{X}'_{\delta 1}/\mathbf{Y}_{\delta 1}}{}^* s_1 \xrightarrow{\mathbf{X}'_{\delta 2}/\mathbf{Y}_{\delta 2}}{}^* s_2 \cdots s_{k-1} \xrightarrow{\mathbf{X}'_{\delta k}/\mathbf{Y}_{\delta k}}{}^* s_k$$

Le point important est de noter que par construction, $\mathbf{X}'_k.\mathbf{T}_k = \mathbf{X}_k$ et que $s_{k-1} \xrightarrow{\mathbf{X}'_{\delta k}/\mathbf{Y}_{\delta k}}{}^* s_k$ est une suite terminale à partir de s_{k-1} pour l'entrée $\mathbf{T}_{k-1}.\mathbf{X}_{\delta k}$.

Par induction, si nous avons que la suite de transition $0_{\mathcal{S}} \xrightarrow{\mathbf{X}'_{\delta 1}/\mathbf{Y}_{\delta 1}}{}^* s_1 \cdots s_{k-1}$ est terminale pour $\mathbf{X}_{k-1}$ en ne consommant que $\mathbf{X}'_{k-1}$, alors $0_{\mathcal{S}} \xrightarrow{\mathbf{X}'_{\delta 1}/\mathbf{Y}_{\delta 1}}{}^* s_1 \cdots s_k$ est terminale pour l'entrée $\mathbf{X}_k = \mathbf{X}'_{k-1}.\mathbf{T}_{k-1}.\mathbf{X}_{\delta k}$, en ne consommant que $\mathbf{X}'_k$.

Le cas $k = 1$ est trivial, puisque par définition $\mathbf{X}_{\delta 1} = \mathbf{X}_1$ et $\mathbf{Y}_{\delta 1} = \mathbf{Y}_1$.

Finalement, puisque $\mathrm{run}(tF)(0_{\mathcal{S}}, \mathbf{X}_k)$ est aussi le résultat d'une suite terminale de transition de $0_{\mathcal{S}}$ pour l'entrée $\mathbf{X}_k$, la confluence permet de conclure.

6.3.9 *Remarque* **L'équité est assurée par un environnement continu**
Bien évidemment, les questions d'équités n'ont pas disparu, mais elles sont maintenant restreintes au découpage de l'entrée $\mathbf{X}$ en morceaux finis $\mathbf{X}_{\delta k}$ tels que la chaîne $(\mathbf{X}_k)_k$ soit de limite $\mathbf{X}$. Cependant nous y voyons un avantage certain, cette contrainte porte sur l'environnement et colle parfaitement à l'intuition et à l'exécution des systèmes réactifs : « l'environnement est continu », il ne peut attendre une infinité de temps avant de fournir une valeur.

6.3.10 *Nota bene* **Fonction de transition confluente et fortement normalisante**
Dans la suite de ce manuscrit, sauf contre-indication, une fonction de transition sera toujours requise d'être confluente et fortement normalisante sur des entrées finies, mais elle pourra avoir des sorties infinies.

6.3.2 Avance rapide dans le manuscrit

Avant d'aller plus loin, nous souhaitons souligner comment l'exécution co-itérative à grands pas illustre l'essentiel des problématiques que nous allons aborder par la suite.

Trois éléments sont à distinguer :

— le rythme d'activation de la fonction tF, qui correspond aux transitions $\rightarrow$.
— le rythme de communication avec le contexte, qui correspond aux grands pas, chaque grand pas renfermant un certain nombre, possiblement nul, mais toujours fini d'activations de tF.
— La variable $\mathbf{T}_k$ qui sert de tampon interne entre la fourniture des entrées par le contexte et la consommation due aux activations de tF.

La distinction entre ces deux rythmes ne tient qu'au choix arbitraire du découpage des entrées : le choix des instants. Ainsi $\mathrm{bigstep}$ illustre parfaitement le lien entre ordonnancement et micro-ordonnancement (entre instants vs. dans l'instant) que nous évoquions en remarque 6.1.29.

L'approche simple et efficace, traditionnellement utilisée en synchrone et plus particulièrement à la suite de Lustre, consiste à n'avoir qu'une activation de tF par grand pas. De plus la communication s'effectue sans tampon interne en obligeant le contexte à fournir exactement ce que l'activation de tF consomme.

Le langage Lucy-n, un descendant de Lustre, est une tentative de relâcher la deuxième contrainte, en acceptant des tampons de tailles bornées. Dans notre discussion sur Lucy-n, nous montrerons comment permettre de multiples activations de tF dans un grand pas et en quoi cela est lié à la taille des tampons nécessaires.

6.3.3 Forme canonique Mealy et Moore

Nous allons montrer que tout réseau de Kahn s'écrit sous forme co-itérative étendue. La forte normalisation de notre construction fera apparaître deux types de réseaux que nous qualifions respectivement de Moore et Mealy.

6.3.11 *Définition* **Réseau Moore et Mealy**
Nous distinguons deux classe de réseaux. Le réseau F est :
— Mealy ssi $F(\varepsilon) = \varepsilon$
— Moore ssi $F(\varepsilon) \neq \varepsilon$

Nous allons construire dans un premier temps une fonction de transition tF_g fortement confluente sans nous soucier de la normalisation forte. La première composante de l'état va stocker au fur et à mesure les entrées consommées quand la deuxième composante mémorisera les sorties déjà produites :

$$\mathcal{S}_g = \{(\mathbf{X}, \mathbf{Y}) \in \mathcal{A}(\mathcal{D}_E) \times \mathcal{D}_S \mid \mathbf{Y} = F(\mathbf{X})\} \cup \{0_{\mathcal{S}}\} \qquad \text{avec} \quad 0_{\mathcal{S}_g} = (\varepsilon_{\mathcal{D}_E}, \varepsilon_{\mathcal{D}_S})$$

Puisque la fonction de transition ne peut consommer que des entrées finies, nous restreignons arbitrairement le domaine de tF_g aux entrées de taille 1 : $\mathrm{dom}(tF_g) = \{(s, \mathbf{X}) \mid |\mathbf{X}| \leq 1\}$. Nous posons alors pour tout état $(\mathbf{X}, \mathbf{Y})$ et entrée $\mathbf{X}_\delta$:

$$tF_g((\mathbf{X}, \mathbf{Y}), \mathbf{X}_\delta) = (\mathbf{Y}_\delta, (\mathbf{X}', \mathbf{Y}')) \quad \text{avec} \quad \mathbf{X}' = \mathbf{X}.\mathbf{X}_\delta \quad \mathbf{Y}' = F(\mathbf{X}') \quad \mathbf{Y}' = \mathbf{Y}.\mathbf{Y}_\delta$$

C'est la monotonie de F qui rend cette définition correcte : comme $\mathbf{X} \sqsubseteq \mathbf{X}'$, nous avons bien $\mathbf{Y} \sqsubseteq \mathbf{Y}'$. Cependant, dans le cas où une des composantes de $\mathbf{Y}$ est infinie, $\mathbf{Y}_\delta$ n'est pas uniquement défini, nous utilisons le plus petit résidu (cf. la concaténation étendue propriété 6.3.2).

La fonction de transition que nous venons de construire est fortement confluente et intuitivement telle que $\mathrm{exec}(tF_g) = F$, mais elle n'est pas fortement normalisante et $\lambda\mathbf{X}.\mathrm{run}(tF_g)(0_\mathcal{S}, \mathbf{X})$ n'est pas une fonction. Le point important à noter est que tF_g ne réagit à ε que de manière très contrainte, car elle correspond à un calcul glouton de la part du réseau :

— Pour tout état $s \neq 0_\mathcal{S}$, $tF_g(s, \varepsilon) = (\varepsilon, s)$
— Pour l'état initial, il y a deux possibilités, suivant si le réseau est :
  — Mealy : $tF_g(0_\mathcal{S}, \varepsilon) = (\varepsilon, 0_\mathcal{S})$
  — Moore : $tF_g(0_\mathcal{S}, \varepsilon) = (\mathbf{Y}_{\delta 1}, (\varepsilon, \mathbf{Y}_{\delta 1}))$ avec $\mathbf{Y}_{\delta 1} \neq \varepsilon$.

Pour utiliser tF_g, nous devons la rendre fortement normalisante, ce que nous allons faire en la mettant sous forme canonique :

6.3.12 *Définition* **Formes canoniques**
Nous définissons deux formes canoniques :

— Une fonction de transition tF en forme canonique Mealy ne réagit pas spontanément :

$$\forall (s, \mathbf{X}) \in \mathrm{dom}(tF), \mathbf{X} \neq \varepsilon$$

— Une fonction de transition tF en forme canonique Moore réagit spontanément une et une seule fois à sa première transition :

$$tF(s, \mathbf{X}) = (\mathbf{Y}, s') \quad \Longrightarrow \quad s' \neq 0_\mathcal{S} \wedge \begin{cases} \mathbf{X} = \varepsilon \wedge \mathbf{Y} \neq \varepsilon & \text{si } s = 0_\mathcal{S} \\ \mathbf{X} \neq \varepsilon & \text{si } s \neq 0_\mathcal{S} \end{cases}$$

6.3.13 *Propriété* **Normalisation forte des formes canoniques**
Une fonction de transition sous forme canonique est fortement normalisante sur $\mathcal{A}(\mathcal{D}_E)$.

Démonstration Si elle est Mealy, c'est immédiat puisque chaque transition consomme un préfixe non vide de l'entrée, une entrée finie ne peut provoquer qu'un nombre fini de transitions.

Si elle est Moore, le raisonnement est le même que pour Mealy, puisqu'une fois la première transition passée, l'état est toujours différent de $0_\mathcal{S}$ et les transitions consomment alors un préfixe non vide.

Nous pouvons maintenant construire, pour tout réseau F une fonction de transition tF_F fortement confluente et fortement normalisante :

6.3.14 *Définition* **La fonction de transition tF_F**
La fonction de transition tF_F est définie comme l'est tF_g, seul son domaine de définition est réduit : $\mathcal{S}_F = \mathcal{S}_g$ et $tF_F(s, \mathbf{X}) = (\mathbf{Y}, s') \implies tF_g(s, \mathbf{X}) = (\mathbf{Y}, s')$.

— Pour un réseau Mealy, le domaine est restreint aux flots de taille 1 :

$$\mathrm{dom}(tF_F) = \{(s, \mathbf{X}) \mid |\mathbf{X}| = 1\}$$

— Pour un réseau Moore, le domaine est :

$$\mathrm{dom}(tF_F) = \{(s, \mathbf{X}) \mid (s = 0_{\mathcal{S}_F} \wedge \mathbf{X} = \varepsilon) \vee (s \neq 0_{\mathcal{S}_F} \wedge |\mathbf{X}| = 1)\}$$

6.3.15 *Propriété*
tF_F est en forme canonique et est fortement confluente.

Démonstration Canonique : Si le réseau est Mealy, c'est trivial. Si le réseau est Moore, il produit, à l'unique première transition, une sortie différente de ε à partir de l'entrée vide. L'état d'arrivée est donc différent de $0_{\mathcal{S}_F}$ et le restera par construction.

Très fortement confluente :

— Si le réseau est Mealy, le domaine est bien clos par unions compatibles, puisque si $|\mathbf{X}| = 1$, $|\mathbf{X}'| = 1$ et $\mathbf{X} \uparrow \mathbf{X}'$ alors $|\mathbf{X} \sqcup \mathbf{X}'| = 1$. Demi diamant : si $(\mathbf{X}, \mathbf{Y}) \xrightarrow{\mathbf{X}_\delta/\mathbf{Y}_\delta} (\mathbf{X}.\mathbf{X}_\delta, \mathbf{Y}.\mathbf{Y}_\delta)$ et $(\mathbf{X}, \mathbf{Y}) \xrightarrow{\mathbf{X}_{\delta 1}/\mathbf{Y}_{\delta 1}} (\mathbf{X}.\mathbf{X}_{\delta 1}, \mathbf{Y}.\mathbf{Y}_{\delta 1})$, avec $\mathbf{X}_{\delta 1}.\mathbf{X}_{\delta 2} = \mathbf{X}_\delta$, alors nous avons que $\mathbf{X}_{\delta 2}$ est aussi de taille 1 et $(\mathbf{X}.\mathbf{X}_{\delta 1}, \mathbf{Y}.\mathbf{Y}_{\delta 1}) \xrightarrow{\mathbf{X}_{\delta 2}/\mathbf{Y}_{\delta 2}} (\mathbf{X}.\mathbf{X}_{\delta 1}.\mathbf{X}_{\delta 2}, \mathbf{Y}.\mathbf{Y}_{\delta 1}.\mathbf{Y}_{\delta 2})$, or $\mathbf{X}.\mathbf{X}_{\delta 1}.\mathbf{X}_{\delta 2} = \mathbf{X}.\mathbf{X}_\delta$, donc par définition des états, $\mathbf{Y}.\mathbf{Y}_{\delta 1}.\mathbf{Y}_{\delta 2} = \mathbf{Y}.\mathbf{Y}_\delta$.

— Si le réseau est Moore. À l'état initial, il n'y a qu'une seule transition, nous avons donc la confluence très forte, pour les autres états, la preuve est celle des réseaux Mealy.

6.3.16 *Propriété*
Pour tout $\mathbf{X}$ fini, $\mathrm{exec}(tF_F)(\mathbf{X}) = F(\mathbf{X})$

Démonstration Pour un réseau Mealy, la fonction de transition est définie pour *tous* les flots de taille 1, une suite terminale ne peut donc s'arrêter qu'une fois l'entièreté des entrées consommées. Il est immédiat par induction sur la taille de l'entrée $\mathbf{X}$ de prouver que la suite de transition termine en étant dans l'état $(\mathbf{X}, F(\mathbf{X})$ après avoir produit $F(\mathbf{X})$. L'induction est bien initialisée puisque, si l'entrée est de taille 0 ($\mathbf{X} = \varepsilon$) la suite terminale ne comprend pas de transition, nous avons donc $\mathrm{exec}(tF_F)(\varepsilon) = \varepsilon$ comme attendu d'un réseau Mealy.

Pour un réseau Moore, la preuve est une variante de celle pour Mealy. Une suite terminale s'arrête aussi après avoir tout consommé et l'induction est identique sauf que, pour la taille 0, la première transition produit la sortie attendue d'un réseau Moore.

Nous pouvons ainsi conclure avec le théorème de représentation suivant :

6.3.17 Théorème **Représentation d'un réseau de Kahn**
Tout réseau de Kahn de sémantique F se décrit sous la forme co-itérative étendue avec la fonction de transition tF_F sous forme canonique fortement confluente.

6.3.18 *Remarque* **Taille de l'état**
tF_F stocke au fur et à mesure l'intégralité de l'histoire de ses entrées et de ses sorties. Il n'est pas possible de s'en passer de manière générale car les fonctions continues peuvent dépendre de l'intégralité de leur histoire. Les languages synchrones apportent traditionnellement des restrictions suffisantes pour que l'état soit de taille *bornée*, ce qui dans un contexte temps réel est primordial.

6.3.19 *Remarque* **Réseau Moore « supérieur » à Mealy**
Remarquez que si la première transition de la forme canonique Moore pouvait produire ε, tout réseau Mealy pourrait s'écrire sous cette forme. Nous ne l'acceptons pas car nous souhaitons distinguer les deux.

6.3.20 *Nota bene* **Lien avec les machines de Moore et de Mealy**
Notre dénomination peut prêter à confusion avec celle des machines Moore et Mealy. Notez bien que, quel que soit le réseau, l'écriture co-itérative est une machine de Mealy généralisée : les sorties de la fonction de transition sont calculées à partir de l'état *et* des entrées.

Toute la subtilité provient de la présence de ε dans notre alphabet, qui permet une première transition sans consommation d'entrée. Ainsi, si nous voulions le mettre sous la forme d'un automate sans ε, tout serait décalé pour que la première sortie soit avec la première entrée et nous aurions un automate Moore dont la sortie ne dépend pas de l'entrée. Cependant, contrairement à l'utilisation traditionnelle des automates, nous avons besoin de pouvoir réagir à ε pour représenter tous les réseaux possibles.

6.3.4 Discussion et exemples

Dans les travaux de Caspi et Pouzet [CP98] la représentation co-itérative avait pour principal but d'exprimer le calcul du réseau sous la forme d'une fonction de transition *scalaire*. Si $tF(s, \mathbf{X}) = (\mathbf{Y}, s)$ alors $|\mathbf{X}| \leq 1$ et $|\mathbf{Y}| \leq 1$.

Ils montrent que la force de cette représentation réside dans l'interaction entre la compilation sous forme de fonction de transition et un « calcul d'horloges ». Ce calcul d'horloges décrit la consommation et la production de valeurs de la fonction de transition. Cette information est cruciale pour permettre de composer ces fonctions de transition sans avoir à gérer de files de communication de taille dynamique. Leur calcul d'horloge est booléen (comme celui que nous avons présenté pour Heptagon), d'où la restriction aux scalaires.

Nous n'avons pas encore discuté d'horloges, cependant nous cherchons une datation, qui comme nous le verrons est à la base de notre présentation des horloges. De manière générale, notre datation, si elle est uniformément réactive permet des tableaux de tailles uniformément bornées.

6.3.21 *Définition* **Granularité de la fonction de transition**
Une fonction de transition $tF : \mathcal{S} \times \mathcal{D}_{tE} \to \mathcal{D}_{tS} \times \mathcal{S}$ est de
- *granularité maximale* n *en entrée* : $\forall \mathbf{X} \in \mathcal{D}_{tE}, |\mathbf{X}| \leq n$
- *granularité maximale* n *en sortie* : $\forall \mathbf{X} \in \mathcal{D}_{tS}, |\mathbf{X}| \leq n$
- *granularité minimale* n *en entrée* : $\forall \mathbf{X} \in \mathcal{D}_{tE}, n \leq |\mathbf{X}|$
- *granularité minimale* n *en sortie* : $\forall \mathbf{X} \in \mathcal{D}_{tS}, n \leq |\mathbf{X}|$

Par défaut, être de granularité n signifie être de granularité *minimale* n *en entrée et en sortie*.

6.3.22 *Nota bene* Dans la suite de cette section, nous expliciterons toujours ε. Si nous utilisons un nom de variable, implicitement celle-ci est différente de ε. Beaucoup de réseaux s'écrivent avec l'aide de seulement deux états, dans ce cas, nous utilisons s_0 pour l'état initial et s_1 pour le second.

Le puits Le réseau dont la sémantique est la fonction qui a toute entrée associe ε est un réseau Mealy. Sa fonction de transition canonique a un domaine de définition vide.

Les constantes Un réseau constant est une fonction qui ne consomme pas d'entrée, mais qui fournit des sorties. La sortie d'une telle fonction est un flot constant. Un flot constant composé toujours de la même constante scalaire est un flot constant stationnaire.

Pour ne pas avoir à traiter spécialement les constantes, nous considérons qu'elles ont une entrée, mais ne la lisent jamais : $(s, \mathbf{x}) \in \mathrm{dom}(tF) \iff \mathbf{x} = \varepsilon$. Toutes les constantes sauf le puits sont des réseaux Moore.

Pour générer le flot $\mathbf{C}$, la forme canonique est de granularité maximale égale à la taille du flot $\mathbf{C}$, donc possiblement infinie, ce qui en fait un réseau non interactif : $tF_{cstc}(s_0, \varepsilon) = (\mathbf{C}, s_1)$.

Le fameux `nat = 0 fby (nat + 1)` est aussi un flot constant puisqu'il ne dépend pas d'entrée. Il peut s'écrire dans une forme *non* canonique, mais fortement confluente de granularité maximale 1, avec pour état initial 0 et tout état entier n : $tF_{nat}(n, \varepsilon) = (n, n+1)$.

Le registre initialisé

La fonction de `0 fby x` est la plus classique des fonctions Moore avec pour forme canonique.

$$tF_{fby}(s_0, \varepsilon) = (0, s_1)$$
$$tF_{fby}(s_1, x) = (x, s_1)$$

L'échantillonneur

La fonction de `x when c` est un réseau de Mealy puisqu'elle ne produit pas de valeur sans en avoir reçue, la fonction canonique usuelle est tF_{when1}, de granularité maximale 1 sans état.

$$tF_{when1}(s_0, (x, \text{true})) = (x, s_0)$$
$$tF_{when1}(s_0, (x, \text{false})) = (\varepsilon, s_0)$$

Quand l'échantillonneur est une constante comme par exemple $(\text{false.true})^\omega$ (le réseau s'écrit `x when (01)` en suivant la syntaxe de Lucy-n), il est possible de l'intégrer directement dans notre fonction $tF_{when1ft}$. L'état sert à connaître la valeur courante du pointeau, initialement à false.

$$tF_{when1ft}(\text{false}, x) = (\varepsilon, \text{true})$$
$$tF_{when1ft}(\text{true}, x) = (x, s_0)$$

Ce même réseau s'écrit sans état avec $tF_{when2ft}$, mais la granularité minimale est de 2 en entrée.

$$tF_{when2ft}(s_0, x_1.x_2) = (x_2, s_0)$$

Attention, si le pointeau vaut $(\text{true.false})^\omega$, la définition $tF_{when1tf}$ est toujours correcte en changeant l'état initial à true, mais la définition ne peut plus être sans état et donne $tF_{when2tf}$.

$$tF_{when2tf}(s_0, x) = (x, s_1)$$
$$tF_{when2tf}(s_1, x_1.x_2) = (x_2, s_1)$$

La complémentation

La fonction de `merge c x y` (avec un pointeau booléen) est de Mealy et de granularité 1 sans état.

$$tF_{merge1}(s_0, (\text{true}, x, \varepsilon)) = (x, s_0)$$
$$tF_{merge1}(s_0, (\text{false}, \varepsilon, y)) = (y, s_0)$$

Si le pointeau est connu et vaut true.false^ω, ce que nous écrivons `merge 1(0) x y`, nous pouvons l'inclure dans la fonction en prenant comment état initial true.

$$tF_{mergetff}(\text{true}, (x, \varepsilon)) = (x, \text{false})$$
$$tF_{mergetff}(\text{false}, (\varepsilon, y)) = (y, \text{false})$$

6.3.5 Granularité et réactivité

Les formes canoniques Mealy et Moore minimisent les possibilités de réactions spontanées, ce qui leur permet de lier le coefficient de Lipschitz avec la granularité de la fonction de transition. En particulier, tout réseau qui n'est pas Lipschitz aura une granularité infinie. De plus, nous allons répondre à la question que nous nous posions en définition 6.1.24 : le nombre d'appels à une fonction de transition canonique, pour une entrée finie, est borné grâce au coefficient K de Lipschitz du réseau.

Avant de prouver ces affirmations, regardons deux exemples.

6.3.23 *Exemple* Pour la fonction qui prend un flot $x_1.x_2.x_3\cdots$ et intercale une constante c entre chaque valeur pour donner $x_1.c.x_2.c.x_3\cdots$. La fonction de transition « naturelle » (à

gauche) est de granularité 1. Mais sous forme canonique (à droite), elle est de granularité 2 en sortie :

$$tF_{interc}(s_0, x) = (x, s_1) \qquad tF_{interc2}(s_0, x) = (x.c, s_0)$$
$$tF_{interc}(s_1, \varepsilon) = (c, s_0)$$

Les fonctions Moore sont plus délicates, car le premier instant est spécial. Pour la fonction qui prend un flot et intercale une constante c dès le premier instant $c.x_1.c.x_2.c.x_3 \cdots$, sous forme canonique, sa granularité est de 2 en sortie *après* la première transition :

$$tF_{interMoore}(s_0, \varepsilon) = (c, s_1) \qquad tF_{interMoore}(s_1, x) = (x.c, s_1)$$

Le point clef, permettant de lier directement la granularité aux nombres d'itérations et au coefficient de Lipschitz, est d'avoir une granularité minimale en entrée non nulle (excepté au premier instant pour les réseau Moore).

Nous donnons dans un premier temps une propriété simple qui éclaire ce lien :

6.3.24 *Propriété* **Granularité minimale en entrée et nombre d'itérations**
Si la fonction de transition tF est de granularité *minimale* d'entrée $g_e > 0$, alors le calcul $\mathrm{exec}(tF)(\mathbf{X})$ s'arrête en au plus $\frac{|\mathbf{X}|}{g_e} \dim(\mathcal{D}_E)$ itérations.

Si elle est Moore et, sauf au premier instant, de granularité *minimale* d'entrée $g_e > 0$, alors le calcul s'arrête en au plus $1 + \frac{|\mathbf{X}|}{g_e} \dim(\mathcal{D}_E)$ itérations.

Démonstration Les pas du calcul de $\mathrm{exec}(tF)(\mathbf{X})$ définissent une suite $(\mathbf{X}_{\delta k})_k$ telle que chaque élément est de taille au moins g_e et la chaîne $(\mathbf{X}_k)_k$ associée est inclue dans $\mathbf{X}$. Cette chaîne ne peut avoir plus de $\frac{|\mathbf{X}|}{g_e} \dim(\mathbf{X})$ éléments et donc de manière générale au plus $\frac{n}{g_e} \dim(\mathcal{D}_E)$ éléments. Pour les réseaux Moore, il faut compter une première transition quelle que soit l'entrée.

La fonction de transition tF_F : Notre fonction tF_F canonique assure, en sus d'avoir une granularité d'entrée minimale et maximale de 1 (sauf au premier instant pour les Moore), d'être définie pour *toutes* les entrées de taille 1. Quelle que soit l'entrée fournie au réseau, elle est entièrement consommée par une exécution terminale :

6.3.25 *Propriété* **Nombre d'itérations de tF_F pour de nouvelles entrées**
Pour toute entrée finie $\mathbf{X}.\mathbf{X}'$, le calcule $\mathrm{exec}(tF_F)(\mathbf{X}.\mathbf{X}')$, le calcule de $\mathrm{exec}(tF_F)(\mathbf{X})$ peut se faire avec exactement $|\mathbf{X}'|$ transitions supplémentaires pour un réseau Mealy et $|\mathbf{X}'| + 1$ pour un réseau Moore.

Ce qui nous permet de conclure sur la granularité de tF_F :

6.3.26 Théorème
La fonction de transition tF_F d'un réseau Mealy est de granularité G_s maximale en sortie, égale à K son coefficient de Lipschitz. Pour un réseau Moore, le premier pas relâche l'égalité en $G_s \leq K \leq 2G_s$.

Cas plus général : Les choses sont plus délicates, puisqu'il peut rester un morceau non consommé de l'entrée qui a servi à parvenir jusqu'à un état, ce morceau pouvant ensuite être utilisé en rajoutant de nouvelles entrées.

6.3.27 *Propriété* **Nombre d'itérations pour de nouvelles entrées**
Si la fonction de transition tF est de granularité en entrée minimale $g_e > 0$ (sauf au premier instant pour Moore) et maximale G_e, alors pour toute entrée $\mathbf{X}.\mathbf{X}'$, le calcul $\mathrm{exec}(tF)(\mathbf{X}.\mathbf{X}')$ effectue au maximum p itérations de plus que le calcul de $\mathrm{exec}(tF)(\mathbf{X})$, avec :

$$p = \frac{(G_e + |\mathbf{X}'|)\dim(\mathcal{D}_E) - 1}{g_e}$$

Auquel il faut rajouter 1 si le réseau est Moore.

Démonstration
— Si le nombre de valeurs non consommées est supérieur ou égal à $G_e \dim(\mathcal{D}_{tE})$, alors il n'y a plus de transition possibles et donc $p = 0$. En effet s'il n'était pas bloqué mais en attente de plus d'entrées alors sa granularité maximale serait supérieure à G_e, puisqu'un flot de taille inférieur à G_e ne peut contenir plus de $G_e \dim(\mathcal{D}_E)$ valeurs.
— Soit ce nombre de valeurs est borné par $G_e \dim(\mathcal{D}_E) - 1$, et alors le nombre d'itérations déclenchées par l'ajout de $\mathbf{X}'$ est borné par celles possibles pour $\mathbf{X}'$ plus le reste de $\mathbf{X}$, soit $(G_e + |\mathbf{X}'|)\dim(\mathcal{D}_E) - 1$ valeurs.

Finalement nous pouvons lier définitivement la granularité des fonctions de transition avec le coefficient de Lipschitz :

6.3.28 Théorème
Une fonction de transition tF est de granularité d'entrée minimale $g_e > 0$ (sauf au premier instant si elle est Moore), maximale G_e et de granularité de sortie minimale g_s et maximale $G_s > 0$ ssi le réseau est K-Lipschitz avec :

$$\frac{g_s}{G_e} \leq K \leq \frac{G_s}{g_e}((G_e + 1)\dim(\mathcal{D}_E) - 1) \qquad (+\ G_s \textit{ si Moore})$$

Démonstration Reprenons les notations de la propriété 6.3.27.
Le nombre minimal d'itérations pour une nouvelle entrée $\mathbf{X}'$ est $\frac{|\mathbf{X}'|}{G_e}$, qu'il faut multiplier par g_s pour obtenir la taille minimale de la sortie. Le coefficient k est alors le rapport de ces deux tailles.
Le nombre maximal d'itérations est p, qu'il faut multiplier par G_s pour obtenir la taille maximale de la sortie : $\frac{G_s}{g_e|\mathbf{X}'|}((G_e + |\mathbf{X}'|)\dim(\mathcal{D}_E) - 1)$. Cette valeur est maximisée pour $|\mathbf{X}'| = 1$. Le cas Moore est identique.

Granularité de la fonction de transition et répartition Pour faire le lien entre la granularité de la fonction de transition (cf. définition 6.3.21) et la granularité dans un sens général, il est nécessaire de considérer que le rapport quantité de calcul sur quantité de communication est constant quoi que l'on fasse si l'algorithme est fixé, tout comme le coefficient de Lipschitz. Si une fonction de transition peut s'exprimer avec une granularité forte, sa quantité de calcul sera plus importante et aura donc une granularité plus grande. La marge de manœuvre est donc restreinte, mais existe, comme nous l'avons vu sur les divers exemples.

6.4 Conclusion

Après avoir introduit les réseaux de Kahn, nous avons montré le passage du calcul du point fixe correspondant à la sémantique à une représentation plus classique avec un système de transition confluent. Cependant, l'exécution du système de transition doit être équitable, ce qui est peu pratique à manipuler. Notre proposition est alors d'accepter que la fonction de transition produise des flots infinis. Grâce à cela, nous pouvons nous borner aux systèmes de transition fortement normalisant, c'est-à-dire effectuant un nombre de pas finis à partir d'une entrée finie. Nous montrons que tout réseau peut s'écrire sous cette forme. La condition d'équité de l'exécution est alors remplacée par une condition de réactivité de l'environment : l'environment (l'utilisateur) doit fournir les entrées morceaux finis par morceaux finis et à chaque fois attendre que le réseau réponde, ce qui est assuré par la forte normalisation. De plus, prouvons que tout réseau peut se mettre sous une forme normale dont les transitions nécessitent toujours au moins une valeur en entrée (sauf la première transition d'un réseau Moore), assurant que le réseau ne fait pas de transition de manière spontanée. Finalement, cela nous permet de lier coefficient de Lipschitz et granularité de la fonction de transition.

Chapitre 7

Réseaux ordonnés et datations

Pour rappel, puisqu'un réseau de Kahn se décrit avec une fonction continue, il peut être utilisé comme nœud d'un autre réseau de Kahn. Cette propriété est une première étape vers une programmation modulaire. La deuxième condition est de connaître le comportement des nœuds que nous utilisons. C'est pour cette raison que nous associons à un réseau de Kahn une fonction de datation. La *datation* abstrait le comportement en donnant un ordre total des communications du système. Elle décide de quand les entrées sont consommées : le système n'est pas passif, il demande à l'environnement/l'utilisateur de se conformer à ses demandes. Cependant, le temps utilisé par le réseau pour spécifier la datation n'est pas absolu et est local au réseau. Il reste donc, à l'environnement du réseau, le choix du temps. C'est exactement ce que nous avons montré dans la première partie, à propos de la compilation modulaire d'Heptagon, où chaque nœud compilé a une signature d'horloge décidant de la présence des entrées et des sorties, mais ce, relativement à une horloge de base qui ne sera fixée qu'à l'utilisation.

La question centrale est de savoir si l'on est capable de donner une datation la plus générale qui ne restreint pas les contextes d'appels du réseau. Une telle datation sera dite *principale*. Cela correspond à avoir un typage principal, à ne pas confondre avec des types principaux [Jim96] [1], puisque la datation décide du type *et* de l'environnement de typage (les dates des entrées). Un tel typage, si nous l'avons, est une garantie de pouvoir effectuer une compilation séparée et une inférence de type incrémentale.

Dans cette recherche, nous avons besoin de deux choses. Comprendre quelles sont les datations correctes d'un réseau et être capable de les comparer pour définir celles qui sont principales. Leur existence nécessite de définir les dépendances de données ce qui requiert de se restreindre aux réseaux *stables* [Ber75]. Mais ceux-ci n'ont pas forcément une datation principale. Nous introduisons la notion de réseau *ordonné* qui correspond à l'existence d'une datation principale. Cette classe n'a pas toutes les bonnes propriétés. En particulier elle n'est pas close par produit. Cependant, tout réseau stable peut se découper en sous morceaux ordonnés, par exemple en prenant sa restriction sur chaque sortie. Nous montrons que tout réseau ordonné se met en forme normale et nous donnons la datation principale correspondant à cette fonction de transition. Cette forme normale est de surcroît de granularité maximale. En règle générale, cette fonction de transition nécessite une lecture non bloquante des entrées. Finalement, les réseaux admettant une fonction de transition séquentielle à lectures bloquantes, sont ceux qui s'écrivent comme un unique processus du langage proposé par Kahn et McQueen [KM76].

Pour mieux comprendre les datations et leur lien avec le calcul du réseau, nous oublions

1. Nous remercions A. Guatto de nous avoir indiqué cette source bibliographique permettant de mettre un nom et une théorie solide derrière nos intuitions.

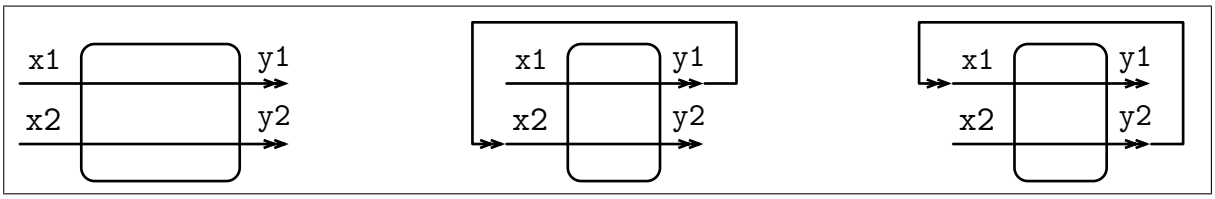


FIGURE 7.1: De gauche à droite, le RKPN de Gonthier, le rebouclage 12 et le rebouclage 21

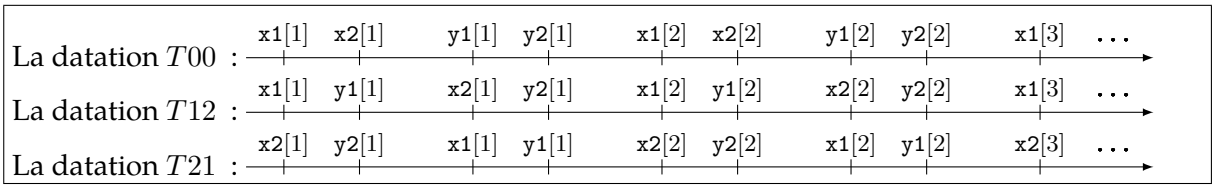


FIGURE 7.2: Des datations du réseau Gonthier.

dans cette section l'existence des instants et les contraintes qu'ils peuvent apporter. Ce choix correspond à l'idée que les instants sont des choix « arbitraires » qui peuvent être effectués a posteriori (cf, remarque 6.1.15), d'autant plus que notre échelle de temps n'est pas absolue. Seule la recherche d'un ordre entre les lectures et les écritures va nous guider.

7.1 Dépendances de données et datations principales

L'exemple suivant est un classique, exposé dans la thèse de G. Gonthier [Gon88] qui met en lumière de manière simple l'impossibilité d'avoir une compilation séparée d'un programme flot de données, vers du code séquentiel, sans perdre des comportements. Nous donnons l'intuition dans cet exemple du lien entre datation et compilation séparée. Ce qui coïncide avec le besoin d'un ordre décrivant totalement les dépendances entre les entrées et les sorties.

7.1.1 La bi-identité de Gonthier

Le réseau de Gonthier a deux entrées `x1` et `x2` et deux sorties `y1` et `y2`. Il est formé du produit de deux fonctions identités : $F_{\text{Gon}}((\mathbf{a},\mathbf{b})) = (\mathbf{a},\mathbf{b})$. Le code Heptagon correspondant est :

```
node gonthier (x1, x2 : int) returns (y1, y2 : int)
let
  y1 = x1;
  y2 = x2;
tel
```

Nous représentons en figure 7.1 le réseau correspondant avec la boîte aux bords ronds, les ports d'entrée sur la gauche et les ports de sortie sur la droite. Ce réseau ne comporte pas de nœuds et les seules files sont $\xrightarrow{\texttt{x1}\ \texttt{y1}}$ et $\xrightarrow{\texttt{x2}\ \texttt{y2}}$. Donnons quelques datations correctes, cf. figure 7.2. Une des premières datations que l'on peut imaginer est $T00$, qui consomme une valeur de `x1`, puis de `x2` avant de produire une valeur pour `y1` puis `y2` et de recommencer ainsi. Formellement, elle peut s'écrire $T00(\texttt{x1})[k] = k$, $T00(\texttt{x2})[k] = k + 0.1$, $T00(\texttt{y1})[k] = k + 0.5$, $T00(\texttt{y2})[k] = k + 0.6$.

Cependant, la datation $T00$ empêche tout rebouclage par le contexte d'une sortie sur une entrée. Nous obtenons ce que nous appelons le rebouclage 12 en rebouclant la première sortie sur la deuxième entrée (cf. rebouclage 12 de figure 7.1) correspondant à l'utilisation `(t, y) = gonthier(x, t)` qui devrait calculer similairement à `y = x`. Ce rebouclage nécessite d'avoir la sortie `y1` produite avant la consommation de l'entrée `x2`, comme le propose la datation $T12$. Mais cette datation empêche le rebouclage 21 qui accepte par contre la datation $T21$.

7.1.1 *Propriété* **Absence de datation principale pour le réseau de Gonthier**
Le réseau de Gonthier n'admet pas de datation qui permette tous les rebouclages possibles.

Démonstration Pour accepter les deux rebouclages que nous avons donnés précédemment, il faudrait avoir la sortie 1 avant l'entrée 2 et la sortie 2 avant l'entrée 1. Mais les dépendances de données du réseau forcent l'entrée 1 avant la sortie 1 et l'entrée 2 avant la sortie 2, ce qui forme un système d'inégalités sans solution.

7.1.2 Datations correctes et principales

Dans un cadre général, l'exécution du calcul du réseau dépend des entrées et les dépendances de données sont relatives aux flots d'entrées. Ceci est visible avec la fonction de transition d'un réseau. Le nouvel état choisi à chaque transition est dépendant des entrées lues, et les futures entrées lues dépendent de l'état.

La datation établit un ordre *total* sur la présence des valeurs aux entrées et sorties du réseau, ordre total qui doit respecter l'ordre *partiel* des dépendances de données. Ce respect est la correction d'une datation. En section 6.1.5.1, pour exhiber la datation T_{canon}, nous nous sommes basés sur le fait que le résultat d'un calcul du réseau n'a pas besoin de plus d'entrées que celles fournies pour ce calcul. Cela ne définit pas les dépendances, mais une sur-approximation que nous appelons suffisances de données.

7.1.2 *Définition* **Données suffisantes**
Les entrées $\mathbf{X} \sqsubseteq \mathbf{D}$ sont suffisantes pour avoir la sortie $\mathbf{Y} \sqsubseteq F(\mathbf{D})$ ssi $\mathbf{Y} \sqsubseteq F(\mathbf{X})$.

Encore une fois, seuls les flots finis nous concernent réellement. Une datation correcte assure la production des sorties finies en un temps fini ainsi que la consommation d'entrées nécessaires strictement avant leurs utilisations.

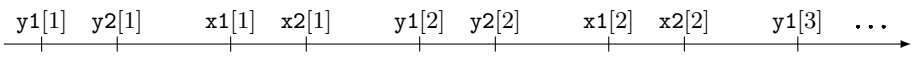
7.1.3 *Définition* **Datation correcte par suffisance**
Le prédicat de correction d'une datation T est donné par :

$$\text{correct}(T) \iff \forall \mathbf{D}, \mathbf{Y} \in \mathcal{A}(F(\mathbf{D})), \exists \mathbf{X} \sqsubseteq \mathbf{D}, \quad \mathbf{Y} \sqsubseteq F(\mathbf{X}) \ \wedge \ T_{\mathbf{D}}(\mathbf{X}) < T_{\mathbf{D}}(\mathbf{Y}) < \infty$$

Rappelons que la datation vaut l'infini pour les valeurs qui ne passent pas par les ports du système. En particulier, $\forall \mathbf{Y} \not\sqsubseteq F(\mathbf{D}), T_{\mathbf{D}}(\mathbf{Y}) = \infty$.

Le prédicat stipule que si le réseau produit $\mathbf{Y}$ (donc $T_{\mathbf{D}}(\mathbf{Y}) < \infty$), alors il consomme une entrée suffisante auparavant.

Pour le réseau de Gonthier, une datation incorrecte met par exemple les $i$ème sorties avant les $i$ème entrées, comme dans la datation suivante :



Pour avoir un typage principal, nous cherchons la datation correcte la plus « générale » si elle existe. Intuitivement, moins une datation est générale, plus elle est restrictive quant-à ses possibles utilisations, c'est-à-dire qu'elle demande des entrées « plus tôt » et donne des sorties « plus tard ».

7.1.4 *Définition* **Relation d'inclusion entre datations**
Une datation T' inclut une autre datation T ssi l'ordre partiel entre les entrées et les sorties qu'elle définit, est compatible avec celui de T. Cette relation est un pré-ordre (réflexif, transitif, mais pas antisymétrique) :

$$T \sqsubseteq T' \iff \forall \mathbf{D}, \mathbf{X} \sqsubseteq \mathbf{D}, \mathbf{Y} \sqsubseteq F(\mathbf{D}), \quad T'_{\mathbf{D}}(\mathbf{X}) < T'_{\mathbf{D}}(\mathbf{Y}) < \infty \implies T_{\mathbf{D}}(\mathbf{X}) < T_{\mathbf{D}}(\mathbf{Y}) < \infty$$

Le terme d'inclusion réfère à l'inclusion des comportements possibles avec des files de stockage en entrées et en sorties. Par exemple, nous avons $T00 \sqsubseteq T12$. En effet, en stockant l'entrée de `x2`, nous pouvons la retarder pour la passer après `y1` et donc adapter la datation $T00$ en la datation $T12$. De même, nous avons $T00 \sqsubseteq T21$. Cette adaptation demande de surcroît de stocker la sortie `y1` pour la fournir après avoir fourni `y2`. Par contre, nous n'avons pas $T12 \sqsubseteq T00$ car $T00$ date la première valeur de `x2` avant la première valeur de `y1`, ce que ne fait pas $T12$. Similairement, nous n'avons pas $T21 \sqsubseteq T00$, ni $T21 \sqsubseteq T12$ ou bien $T12 \sqsubseteq T21$.

Il n'est pas intéressant de définir une relation antisymétrique car la notion de datation est relative à l'échelle de temps que l'on considère et seul l'ordre relatif des dates des entrées et des sorties compte. Ainsi nous pouvons prendre le quotient de la relation d'inclusion pour définir une relation d'équivalence.

7.1.5 *Définition* **Relation d'équivalence entre datations**
Une datation T est équivalente à une autre datation T' ssi les ordres partiels qu'elles définissent entre les entrées et les sorties sont les mêmes :

$$T \backsim T' \qquad \iff \qquad T \sqsubseteq T' \wedge T' \sqsubseteq T$$

L'inclusions de datations permet d'exprimer de manière formelle et concise la remarque que nous faisions en section 6.1.5.2 :

7.1.6 *Propriété* **Avancer des entrées et retarder des sorties**
À partir d'une datation correcte, il est possible d'avancer des entrées et de retarder des sorties pour créer une autre datation correcte :

$$\text{correct}(T) \quad \wedge \quad T' \sqsubseteq T \qquad \implies \qquad \text{correct}(T')$$

Parmi les datations, celles qui nous intéressent sont les plus grandes pour la relation d'inclusion car elles stockent le moins leurs valeurs et permettent le plus de comportements. Une datation principale est, modulo équivalence, un maximum de l'ensemble des datations correctes pour la relation d'inclusion :

7.1.7 *Définition* **Datation principale**
Une datation est dite principale ssi elle est correcte et inclut toutes les datations correctes :

$$\text{principal}(T) \qquad \iff \qquad \text{correct}(T) \quad \wedge \quad \forall \text{correct}(T'), T' \sqsubseteq T$$

L'existence d'une datation principale n'est pas assurée comme l'atteste l'exemple du réseau de Gonthier.

Les datations principales, quand elles existent, définissent un ordre partiel *nécessaire* et suffisant entre les entrées et les sorties. La recherche d'une datation principale amène ainsi à considérer les dépendances de données.

7.2 Dépendances de données et réseaux stables

Dans cette section, nous cherchons les entrées suffisantes et nécessaires à l'obtention d'une sortie et donc une notion de dépendances de données. Intuitivement, la dépendance de donnée est définie par la plus petite entrée suffisante pour générer la sortie voulue. Ce serait donc le plus petit élément des prédécesseurs. Les prédécesseurs de $\mathbf{Y}$ dans l'entrée $\mathbf{D}$ sont $\mathrm{pred}(F, \mathbf{D}, \mathbf{Y})$ définis par :

$$\forall \mathbf{D}, \mathbf{Y} \sqsubseteq F(\mathbf{D}), \quad \mathrm{pred}(F, \mathbf{D}, \mathbf{Y}) = \Big\{ \mathbf{X} \mid \mathbf{X} \sqsubseteq \mathbf{D} \wedge \mathbf{Y} \sqsubseteq F(\mathbf{X}) \Big\}$$

7.2.1 *Définition* **Dépendance de données**
Pour tout flot d'entrées $\mathbf{D}$, la sortie $\mathbf{Y}$ dépend de l'entrée $\mathbf{X}$ et est notée $\mathbf{X} \rightarrow^{F}_{\mathbf{D}} \mathbf{Y}$, avec F généralement omis. Le prédicat de dépendance est :

$$\mathbf{X} \rightarrow_{\mathbf{D}} \mathbf{Y} \qquad \Longleftrightarrow \qquad \mathbf{X} = \bigsqcap \mathrm{pred}(F, \mathbf{D}, \mathbf{Y}) \quad \wedge \quad \mathbf{X} \in \mathrm{pred}(F, \mathbf{D}, \mathbf{Y})$$

Dans le cas général, cette intersection n'est pas dans l'image réciproque, ce qui correspond au fait qu'il y a des sorties dont on ne peut connaître les entrées nécessaires et suffisantes.

La recherche d'un plus petit élément de l'image réciproque d'une fonction continue n'est pas nouveau et remonte (au moins) aux travaux de Gérard Berry concernant le déterminisme du calcul « bottom-up » de programmes récursifs [Ber75]. Dans ses travaux, il introduit la notion de fonction stable, fonction ayant un minimum pour toutes ses images réciproques. La stabilité est réapparue comme une propriété importante dans la recherche du modèle complètement adéquat de PCF [Ber78] et dans de nombreux travaux consécutifs.

7.2.2 *Définition* **Stabilité d'un réseau**
Un réseau F est stable ssi pour toute sortie, il existe une plus petite entrée permettant de générer cette sortie :

$$\forall \mathbf{D}, \mathbf{Y} \sqsubseteq F(\mathbf{D}), \exists \mathbf{X} \sqsubseteq \mathbf{D}, \quad \mathbf{X} \rightarrow_{\mathbf{D}} \mathbf{Y}$$

Le *ou*-parallèle est la fonction classique qui est continue mais pas stable, nous lui consacrons la section 7.2.1.

Les domaines de Kahn ont assez de structure pour permettre une caractérisation plus légère des fonctions stables. Dans un *DI-domaine* [Ber78], un réseau F est stable ssi il est *consistently multiplicative* [Ber78] :

$$\forall \mathbf{X} \uparrow \mathbf{X}', \quad F(\mathbf{X} \sqcap \mathbf{X}') = F(\mathbf{X}) \sqcap F(\mathbf{X}')$$

Or le domaine des suites de Kahn est un *DI-Domain* (cf. annexe IV). Plus précisément, comme F et $\sqcap$ sont des fonctions continues, nous pouvons nous restreindre aux entrées finies :

7.2.3 *Propriété* **Caractérisation des réseaux stables**
Le réseau F est stable ssi :

$$\forall \mathbf{D}, \mathbf{X} \in \mathcal{A}(\mathbf{D}), \mathbf{X}' \in \mathcal{A}(\mathbf{D}), \quad F(\mathbf{X} \sqcap \mathbf{X}') = F(\mathbf{X}) \sqcap F(\mathbf{X}')$$

7.2.4 *Propriété* **La stabilité est close par composition, point fixe et produit : Lustre est stable**
La composition, le point fixe et le produit de deux fonctions stables sont stables (cf. [Ber78]). Comme pour une grande partie des langages fonctionnels, en Lustre, toutes les constructions du langage sont stables et ne permettent de définir que des programmes stables.

7.2.1 Réseaux non-stables et l'exemple du *ou* parallèle

7.2.1.1 Le *ou* parallèle du domaine de Scott

La fonction non stable la plus idiomatique est la fonction calculant le *ou* paresseux, aussi appelé *ou* parallèle. Cette fonction lit ses entrées de manière paresseuse et dans un ordre non déterministe. Dès qu'une des entrées vaut vrai, elle peut fixer sa sortie à vrai. Dans le domaine scalaire de Scott booléen à trois valeurs ($\bot$ signifie l'absence d'information et est inférieur à true et false, ces derniers sont incomparables), la fonction scalaire por est définie par :

$$\begin{array}{ccc} por(\mathrm{true},\bot)=\mathrm{true} & por(\bot,\mathrm{true})=\mathrm{true} & por(\mathrm{false},\mathrm{false})=\mathrm{false} \\ por(\mathrm{true},\mathrm{false})=\mathrm{true} & por(\mathrm{false},\mathrm{true})=\mathrm{true} & por(\bot,\bot)=\bot \end{array}$$

Cette fonction est le cas d'école de la fonction continue qui n'est pas stable. En effet, la sortie true a parmi ses antécédents $(\mathrm{true},\bot)$ et $(\bot,\mathrm{true})$. L'intersection de ces antécédents est $(\bot,\bot)$ et a pour image $\bot$: elle n'est donc pas consistently multiplicative. En d'autres termes, Il n'est pas possible de déterminer de plus petite entrée permettant de produire true.

7.2.1.2 Le *ou* parallèle étendu aux flots

Nous étendons point à point le *ou* parallèle aux flots. Mais de la même manière que por était en mesure de fournir son résultat sans avoir toutes ses entrées, F_{por} donne son résultat dès que possible. Par exemple, il doit donner :

$$F_{por}(\mathrm{true.true},\varepsilon)=F_{por}(\mathrm{true.true},\mathrm{false})=F_{por}(\mathrm{true.false},\mathrm{false.true})=\mathrm{true.true}$$

La définition co-récursive de cette fonction est :

$$\begin{aligned} F_{por}(\mathrm{false}.\mathbf{x},\mathrm{false}.\mathbf{x}') &= \mathrm{false}.F_{por}(\mathbf{x},\mathbf{x}') \\ F_{por}(\mathrm{true}.\mathbf{x},\mathrm{true}.\mathbf{x}')=F_{por}(\mathrm{false}.\mathbf{x},\mathrm{true}.\mathbf{x}')=F_{por}(\mathrm{true}.\mathbf{x},\mathrm{false}.\mathbf{x}') &= \mathrm{true}.F_{por}(\mathbf{x},\mathbf{x}') \\ F_{por}(\varepsilon,\mathrm{true}.\mathbf{x})=F_{por}(\mathrm{true}.\mathbf{x},\varepsilon) &= \mathrm{true}.F_{por}(\mathbf{x},\varepsilon) \\ F_{por}(\varepsilon,\mathrm{false}.\mathbf{x})=F_{por}(\mathrm{false}.\mathbf{x},\varepsilon) &= \varepsilon \end{aligned}$$

Nous pouvons aussi donner sa fonction de transition en forme canonique Mealy. La fonction de transition doit à tout moment être capable de produire true avec uniquement true sur une de ses entrées. Mais, comme par la suite elle peut recevoir une valeur sur l'autre entrée, il faut qu'elle se souvienne de l'avance prise par sa première entrée. Nous prenons pour état un entier, initialisé à 0. S'il est positif, il indique l'avance prise par la première entrée, s'il est négatif son retard (l'avance prise par la deuxième entrée). Avec la variable x valant true ou false mais pas ε, nous avons :

$$\begin{array}{ll} tF_{por}(n>0,(\mathrm{true},\varepsilon))=(\mathrm{true},n+1) & tF_{por}(n>0,(\varepsilon,x))=(\varepsilon,n-1) \\ tF_{por}(n<0,(\varepsilon,\mathrm{true}))=(\mathrm{true},n-1) & tF_{por}(n<0,(x,\varepsilon))=(\mathrm{true},n+1) \\ tF_{por}(0,(\mathrm{false},\mathrm{false}))=(\mathrm{false},0) & \end{array}$$

Pour se convaincre de la confluence de cette fonction de transition, nous pouvons la rendre très fortement confluente et normalisante en ajoutant ces deux cas :

$$tF_{por}(n \geq 0, (\text{true}, \text{false})) = tF_{por}(n \leq 0, (\text{false}, \text{true})) = (\text{true}, n)$$

7.2.1.3 La primitive `WARN`

Comme nous l'avons remarqué en propriété 7.2.4, la plupart des langages de programmation ne permettent pas de définir des fonctions non-stables. Ceci est le cas pour PCF, ce qui est une raison de l'utilisation de cette propriété dans les travaux de Gérard Berry, mais c'est aussi le cas pour Lustre, Lucid Synchrone, Lucy-n, Heptagon, etc.

Le langage rudimentaire proposé par Kahn [Kah74] est aussi stable car il est séquentiel. Nous verrons par la suite qu'un langage séquentiel ne forme que des fonctions stables. Pour palier ceci, il propose, en conclusion de l'article, de rajouter des primitives et en particulier `WARN` :

> The programming language we have introduced can be extended by adding new primitive processes (i.e. that cannot be programmed as processes with wait and send). A typical such process is `WARN (integer in X,Y; logical out Z)` that sends a true value on its output line each time some integer is received on either of its input lines. The only condition to be verified by the new primitive processes, and verified by `WARN`, is that the history of the output line be a continuous function of the histories of the input lines.

La fonction `WARN` a une sémantique plus simple que F_{por}, mais similairement, elle est capable de prendre une décision à partir de *la première des entrées qui arrive*, quelque soit son port d'arrivée : $F_{\texttt{WARN}}(x, \varepsilon) = F_{\texttt{WARN}}(\varepsilon, y) = \text{true}$. Décision qui produit la même sortie, ce qui en fait une fonction non-stable. Sa forme canonique Mealy a pour seul état 0 et :

$$tF_{\texttt{WARN}}(0, (x, \varepsilon)) = (\text{true}, 0) \qquad tF_{\texttt{WARN}}(0, (\varepsilon, y)) = (\text{true}, 0)$$

Pour se convaincre de la confluence, nous pouvons la rendre très fortement confluente en ajoutant la transition $tF_{\texttt{WARN}}(0, (x, y)) = (\text{true.true}, 0)$ de granularité 2 en sortie.

7.2.1.4 Utilisation du *ou* parallèle

Les fonctions non-stables, comme le « ou » parallèle ou bien la fonction `WARN` sont très attrayantes quant-à l'expressivité du réseau et le dynamisme qu'elles apportent à l'exécution. Il nous semble possible d'abstraire le « ou » parallèle par un « ou » strict pour les analyses puis d'utiliser la version parallèle à l'exécution. Cependant, il y a de fortes chances que l'exécution ne puisse pas en tirer profit.

Un résultat dû à Plotkin [Plo77] est que PCF muni du *ou* parallèle contient toutes les fonctions continues du domaine de Scott. De même, ceci est probablement le cas pour les réseaux de Kahn avec le réseau F_{por}. Une option intéressante serait de permettre au programmeur d'utiliser une construction spéciale effectuant le *ou* parallèle, gérée de manière ad-hoc par le compilateur. Un peu à la manière de ce que nous avons proposé pour les *futures* en section 4.6.2.

7.2.2 Fonctions de transition stables

Affirmer qu'une fonction de transition a pour exécution une fonction stable n'est pas immédiat. Une condition évidente à la stabilité du réseau est la « stabilité » de la fonction de transition. Cependant, dans notre cadre, le domaine de définition de la fonction de transition

n'est pas obligatoirement un domaine de calcul, puisque entres autres ε peut en être exclu. Il ne faut donc forcer le domaine de définition que quand cela est nécessaire : quand deux transitions produisent des sorties d'intersection non vide.

7.2.5 *Définition* **Stabilité d'une fonction de transition**
La fonction de transition tF est dite stable ssi

$$\forall s, \forall \mathbf{X_1} \uparrow \mathbf{X_2}, \quad (tF(s, \mathbf{X_1}) = (\mathbf{Y_1}, s_1) \quad \wedge \quad tF(s, \mathbf{X_2}) = (\mathbf{Y_2}, s_2) \quad \wedge \quad \mathbf{Y_1} \sqcap \mathbf{Y_2} \neq \boldsymbol{\varepsilon}) \\ \implies tF(s, \mathbf{X_1} \sqcap \mathbf{X_2}) = (\mathbf{Y_1} \sqcap \mathbf{Y_2}, s')$$

7.2.6 *Propriété*
Soit la fonction de transition tF. Si $\mathrm{exec}(tF)$ est une fonction stable, alors tF est une fonction de transition stable.

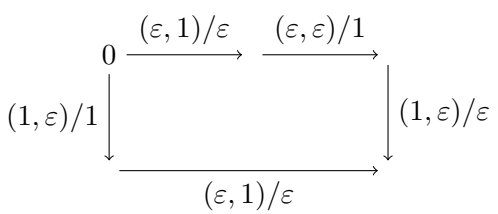
Démonstration Par contraposé, si tF n'est pas stable, il existe un état s, tel que $0 \xrightarrow{\mathbf{X/Y}} s$, $s \xrightarrow{\mathbf{X_1/Y_1}} s_1$ et $s \xrightarrow{\mathbf{X_2/Y_2}} s_2$ avec $\mathbf{X_1} \uparrow \mathbf{X_2}$, donc avec $\mathbf{X}.\mathbf{X_1} \uparrow \mathbf{X}.\mathbf{X_2}$ et $\mathbf{Y_1} \sqcap \mathbf{Y_2} \neq \boldsymbol{\varepsilon}$, de plus,

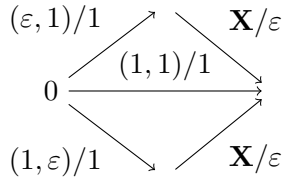
- Soit $\mathbf{X_1} \sqcap \mathbf{X_2}$ n'est pas dans le domaine de définition de la fonction de transition en s.
- Soit $s \xrightarrow{\mathbf{X_1} \sqcap \mathbf{X_2}/\mathbf{Y'}} s'$, avec $\mathbf{Y'} \neq \mathbf{Y_1} \sqcap \mathbf{Y_2}$. Par confluence, $\mathbf{Y'} \sqsubseteq \mathbf{Y_1}$ et $\mathbf{Y'} \sqsubseteq \mathbf{Y_2}$, donc $\mathbf{Y'} \sqsubset \mathbf{Y_1} \sqcap \mathbf{Y_2}$.

Dans les deux cas, il n'existe pas de plus petite entrée $\mathbf{X_m}$ telle que $\mathbf{Y}.(\mathbf{Y_1} \sqcap \mathbf{Y_2}) \sqsubseteq \mathrm{exec}(tF)(\mathbf{X_m})$, le réseau n'est donc pas stable.

Cependant, la stabilité de la fonction de transition n'est pas suffisante, comme le montre l'exemple suivant.

7.2.7 *Exemple* Soit F non stable, sur l'alphabet à une seule lettre 1, telle que $F(\varepsilon, \varepsilon) = \varepsilon$ et sinon $F(\mathbf{x}, \mathbf{y}) = 1$. Ci-dessous sur la gauche, le graphe d'une fonction de transition stable pouvant être utilisée pour représenter ce réseau, sur la droite celui non stable de la version canonique tF_F :





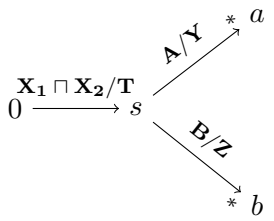
Avec $\mathbf{X} = (\varepsilon, 1)$ ou $(1, \varepsilon)$ ou $(1,1)$.

Remarquez que nous avons réussi à avoir une fonction de transition stable grâce à une transition ne consommant pas son entrée.

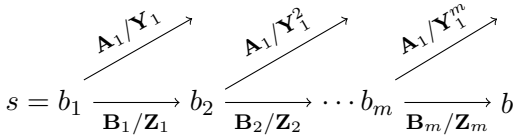
La fonction de transition en forme canonique tF_F a pour principale caractéristique d'être réceptive, c'est-à-dire de pouvoir consommer n'importe quelle entrée à n'importe quel état. Ceci la rapproche beaucoup des automates d'entrées/sorties de Stark et relie informellement leur condition de stabilité [PSS90] avec la nôtre (cf. définition 7.2.5). Ainsi, la stabilité de tF_F implique la stabilité de son exécution :

7.2.8 Théorème **Le réseau F est stable ssi sa fonction de transition canonique tF_F est stable**

Démonstration Soit $\mathbf{X_1}$ et $\mathbf{X_2}$ deux entrées compatibles ($\mathbf{X_1} \uparrow \mathbf{X_2}$). Puisque tF_F est réceptive, il existe une suite terminale de transition consommant exactement $\mathbf{X_1} \sqcap \mathbf{X_2}$: $0 \xrightarrow{\mathbf{X_1} \sqcap \mathbf{X_2}/\mathbf{Y}'}{}^{*} s$. Posons alors $\mathbf{X_1} = (\mathbf{X_1} \sqcap \mathbf{X_2}).\mathbf{A}$ et $\mathbf{X_2} = (\mathbf{X_1} \sqcap \mathbf{X_2}).\mathbf{B}$. Par définition, $\mathbf{A} \sqcap \mathbf{B} = \varepsilon$. Encore une fois, par réceptivité, nous avons :



en détail la branche partant de s :



Notre but est de prouver que $\mathbf{Y} \sqcap \mathbf{Z} = \varepsilon$.

La première chose à faire est de fixer les suites de transition telles que $\mathbf{A} = \mathbf{A}_1.\cdots.\mathbf{A}_n$ et $\mathbf{B} = \mathbf{B}_1.\cdots.\mathbf{B}_m$.

En s, nous avons $\mathbf{A}_1 \sqcap \mathbf{B}_1 = \varepsilon$, ainsi $\mathbf{Y}_1 \sqcap \mathbf{Z}_1 = \varepsilon$, sinon par stabilité il y aurait une transition à partir de s avec l'entrée ε. Or tous les états que nous allons considérer, s y compris, n'ont pas ε dans leurs transitions possibles. C'est par définition si le réseau est Mealy. Si le réseau est Moore, nous sommes certain que s est différent de l'état initial qui n'est alors plus accessible puisque tF_F est canonique.

Similairement, comme $\mathbf{A}_1 \sqcap \mathbf{B}_i = \varepsilon$, il y a pour tous les b_i une transition consommant $\mathbf{A}_1$ et produisant $\mathbf{Y}_1^i$. Par confluence, nous avons que $\mathbf{Y}_1 \sqsubseteq \mathbf{Z}_1.\mathbf{Y}_1^2$ mais $\mathbf{Y}_1 \sqcap \mathbf{Z}_1 = \varepsilon$ donc $\mathbf{Y}_1 \sqsubseteq \mathbf{Y}_1^2$. Par stabilité nous avons que $\mathbf{Y}_1^2 \sqcap \mathbf{Z}_2 = \varepsilon$ donc $\mathbf{Y}_1 \sqcap \mathbf{Z}_2 = \varepsilon$. Par induction et symétrie, nous avons $\mathbf{Y}_1 \sqcap \mathbf{Z}_j = \varepsilon$ et $\mathbf{Z}_1 \sqcap \mathbf{Y}_i = \varepsilon$ avec $i = 1 \ldots n$ et $j = 1 \ldots m$.

Nous pouvons répéter le raisonnement en partant de b_2 pour prouver que $\mathbf{Y}_1^2 \sqcap \mathbf{Z}_j = \varepsilon$, ce qui ne nous avance pas, mais aussi $\mathbf{Z}_2 \sqcap \mathbf{Y}_i^2 = \varepsilon$. Or par confluence, $\mathbf{Y}_1.\cdots.\mathbf{Y}_n \sqsubseteq \mathbf{Y}_1^2.\cdots.\mathbf{Y}_n^2$ donc $\mathbf{Z}_2 \sqcap \mathbf{Y}_i = \varepsilon$. La récurrence est évidente et termine la preuve.

7.2.9 *Remarque* **Une propriété plus générale**
Dans la preuve de cette dernière propriété, la réceptivité de tF_F semble jouer un rôle important, pourtant elle n'aide que pour déterminer une récurrence *explicite*. Avoir, pour seules transitions avec ε comme entrée, des transitions de la forme $s \xrightarrow{\varepsilon/\varepsilon} s$ (sauf une au premier instant) et la normalisation forte, est suffisant. La preuve de cette affirmation est nettement plus technique et demanderait de parler en détail des atomes des flots et de l'interaction entre confluence et « parallélisme » de deux flots ($\mathbf{A} \| \mathbf{B} \iff (\mathbf{A} \sqcap \mathbf{B} = \varepsilon \wedge \mathbf{A} \uparrow \mathbf{B})$ notation en référence à [PSS90]). La preuve repose sur un lemme qui énonce que si $\mathbf{A} \| \mathbf{B}$, $s \xrightarrow{\mathbf{A}/\mathbf{Y}}{}^{*}$ et $s \xrightarrow{\mathbf{B}/\mathbf{Z}}{}^{*}$ avec $\mathbf{Y} \sqcap \mathbf{Z} \neq \varepsilon$, alors la confluence et le fait que toutes les transitions consomment au moins un atome implique l'existence d'un état s' tel que $s' \xrightarrow{\mathbf{A}'/\mathbf{Y}'}$ et $s' \xrightarrow{\mathbf{B}'/\mathbf{Y}'}$ avec $\mathbf{Y}' \sqcap \mathbf{Z}' \neq \varepsilon$, avec $\mathbf{A}' \| \mathbf{B}'$, chacun composé d'au moins un atome de moins. Cependant, nous n'irons pas plus loin dans ce manuscrit.

7.2.3 Les inverses

Nous allons fortement faire appel aux entrées minimales nécessaires à une sortie. Pour les manipuler plus aisément, nous les appelons des inverses. Donnons quelques notations et propriétés utiles.

7.2.10 *Définition* **Inverses**
Pour tout $\mathbf{Y} \sqsubseteq F(\mathbf{D})$, l'entrée $\mathbf{X} \rightarrow^{F}_{\mathbf{D}} \mathbf{Y}$ est unique puisque minimale. Nous appelons cette entrée l'inverse de $\mathbf{Y}$, que nous notons $\mathbf{Y}^{F*\mathbf{D}}$ ou simplement $\mathbf{Y}^{*\mathbf{D}}$.

Les inverses d'un réseau stables sont des outils puissants pour étudier le comportement du réseau. Ils vont de paire avec les bi-inverses.

7.2.11 *Définition* **Bi-inverses**
Le bi-inverse $\mathbf{Y}^{**\mathbf{D}}$ est la sortie maximale produite à partir de l'entrée nécessaire pour produire $\mathbf{Y}$. Il se calcule directement à partir de l'inverse

$$\mathbf{Y}^{**\mathbf{D}} = F(\mathbf{Y}^{*\mathbf{D}})$$

7.2.12 *Remarque* Ce que nous appelons un inverse est noté $m(F, \mathbf{D}, \mathbf{Y})$ par G. Berry [Ber78].

7.2.13 *Propriété* **Les inverses sont indépendant du futur**
Le reflet de la monotonie du réseau est qu'un inverse ne dépend pas des entrées supérieures :

$$\forall \mathbf{D}, \mathbf{D}', \quad \mathbf{Y}^{*\mathbf{D}} \sqsubseteq \mathbf{D}' \implies \mathbf{Y}^{*\mathbf{D}'} = \mathbf{Y}^{*\mathbf{D}}$$

7.2.14 *Propriété* **Propriétés immédiates des bi-inverses**
Par définition, nous avons $\mathbf{Y} \sqsubseteq \mathbf{Y}^{**\mathbf{D}}$, mais aussi $\mathbf{Y}^{*\mathbf{D}} \rightarrow_{\mathbf{D}} \mathbf{Y}^{**\mathbf{D}}$, ainsi l'inverse du bi-inverse est l'inverse tout court : $(\mathbf{Y}^{**\mathbf{D}})^{*\mathbf{D}} = \mathbf{Y}^{*\mathbf{D}}$.

7.2.15 *Propriété* **Monotonie stricte de la dépendance**
Comme attendu, l'entrée minimale pour obtenir une sortie croit avec la sortie :

$$\forall \mathbf{D}, \mathbf{X}, \mathbf{Y}, \quad \mathbf{X} \sqsubset \mathbf{Y}^{*\mathbf{D}} \implies F(\mathbf{X}) \sqsubset \mathbf{Y}$$

Ce qui donne entres autres :

$$\forall \mathbf{D}, \mathbf{Y_1}, \mathbf{Y_2}, \quad \mathbf{Y_1}^{*\mathbf{D}} \sqsubset \mathbf{Y_2}^{*\mathbf{D}} \implies \mathbf{Y_1} \sqsubset \mathbf{Y_2}$$
$$\forall \mathbf{D}, \mathbf{Y_1}, \mathbf{Y_2}, \quad \mathbf{Y_1}^{*\mathbf{D}} \sqsubseteq \mathbf{Y_2}^{*\mathbf{D}} \implies \mathbf{Y_1} \sqsubseteq \mathbf{Y_2}^{**\mathbf{D}}$$

Démonstration La preuve est la conjonction de la monotonie de F et de l'unicité de l'inverse. Les deux conséquences correspondent à l'utilisation de l'inclusion $\mathbf{Y} \sqsubseteq \mathbf{Y}^{**\mathbf{D}} = F(\mathbf{Y}^{*\mathbf{D}})$.

7.2.16 *Propriété* **Monotonie de l'inverse**
Cette propriété est l'essence même des fonctions stables. Elle énonce que plus de sortie ne peut provenir que de plus d'entrée (et non d'une entrée incompatible) :

$$\forall \mathbf{D}, \mathbf{Y_1}, \mathbf{Y_2}, \quad \mathbf{Y_1} \sqsubseteq \mathbf{Y_2} \sqsubseteq F(\mathbf{D}) \implies \mathbf{Y_1}^{*\mathbf{D}} \sqsubseteq \mathbf{Y_2}^{*\mathbf{D}}$$
$$\forall \mathbf{D}, \mathbf{Y_1}, \mathbf{Y_2}, \quad \mathbf{Y_1}^{**\mathbf{D}} \sqsubset \mathbf{Y_2} \sqsubseteq F(\mathbf{D}) \implies \mathbf{Y_1}^{*\mathbf{D}} \sqsubset \mathbf{Y_2}^{*\mathbf{D}}$$

Démonstration Soit $\mathbf{Y_1} = \mathbf{Y_1} \sqcap \mathbf{Y_2} = F(\mathbf{Y_1}^{*\mathbf{D}}) \sqcap F(\mathbf{Y_2}^{*\mathbf{D}})$, en utilisant la stabilité de F, c'est aussi égal à $F(\mathbf{Y_1}^{*\mathbf{D}} \sqcap \mathbf{Y_2}^{*\mathbf{D}})$. Or $\mathbf{Y_1}^{*\mathbf{D}}$ est le plus petit antécédent de $\mathbf{Y_1}$ donc $\mathbf{Y_1}^{*\mathbf{D}} \sqsubseteq \mathbf{Y_1}^{*\mathbf{D}} \sqcap \mathbf{Y_2}^{*\mathbf{D}}$, cqfd.

La précision pour le bi-inverse tient au fait que si $\mathbf{Y_1}^{*\mathbf{D}} = \mathbf{Y_2}^{*\mathbf{D}}$, alors nous avons $\mathbf{Y_1}^{**\mathbf{D}} = \mathbf{Y_2}^{**\mathbf{D}}$ mais par définition $\mathbf{Y_2} \sqsubseteq \mathbf{Y_2}^{**\mathbf{D}}$, ce qui est impossible.

7.2.17 *Propriété* **Inverses finis et sorties finies**
L'inverse fini de toute sortie est aussi inverse d'une sortie finie :

$$\forall \mathbf{D}, \mathbf{Y} \sqsubseteq F(\mathbf{D}), \quad \mathbf{Y}^{*\mathbf{D}} \in \mathcal{A}(\mathbf{D}) \quad \Longrightarrow \quad \exists \mathbf{Y}' \in \mathcal{A}(\mathbf{Y}), \mathbf{Y}^{*\mathbf{D}} = \mathbf{Y}'^{*\mathbf{D}}$$

Démonstration Si $\mathbf{Y}$ est fini, nous pouvons le prendre $\mathbf{Y} = \mathbf{Y}'$. Sinon $\mathbf{Y}$ est infini et nous pouvons prendre une chaîne infinie strictement croissante $(\mathbf{Y}_k)_k$ d'éléments de $\mathcal{A}(\mathbf{Y})$ convergeant vers $\mathbf{Y}$. Maintenant par l'absurde, considérons que tous les $\mathbf{Y}_k$ peuvent être produits par une entrée plus petite : $\forall k, \exists \mathbf{X} \sqsubset \mathbf{Y}^{*\mathbf{D}}, \mathbf{Y}_k \sqsubseteq F(\mathbf{X})$. Puisque $\mathbf{Y}^{*\mathbf{D}}$ est fini, il n'a qu'un nombre fini de minorants : il existe une entrée $\mathbf{X}' \sqsubset \mathbf{Y}^{*\mathbf{D}}$ produisant une infinité des éléments de la suite $(\mathbf{Y}_k)_k$. Nous pouvons donc en extraire une sous-chaîne $(\mathbf{Y}_{\phi k})_k$. Cette sous-chaîne est infinie, elle converge donc aussi vers $\mathbf{Y}$. Pourtant tous les éléments sont inférieurs à $F(\mathbf{X}')$, donc $\mathbf{Y}$ aussi, ce qui est impossible par minimalité de $\mathbf{Y}^{*\mathbf{D}}$.

7.2.18 *Propriété* **Inverse infini et infinité d'inverses finis**
S'il existe un inverse infini $\mathbf{Y}^{*\mathbf{D}}$, alors $\mathbf{Y}$ est infini et il existe une chaîne infinie strictement croissante d'*inverses* finis qui converge vers $\mathbf{Y}^{*\mathbf{D}}$.

Démonstration Premièrement, $\mathbf{Y}$ est infini car sinon par continuité (en particulier l'attente finie propriété 0.2) du réseau impliquerait l'existence d'une entrée finie suffisante pour produire $\mathbf{Y}$, ce qui contredirait la minimalité de $\mathbf{Y}^{*\mathbf{D}}$.

Deuxièmement, soit $\mathbf{X}$ fini inclus dans $\mathbf{Y}^{*\mathbf{D}}$. Nous allons prouver qu'il existe un inverse $\mathbf{Y}'^{*\mathbf{D}}$ fini tel que $\mathbf{X} \sqsubset \mathbf{Y}'^{*\mathbf{D}} \sqsubset \mathbf{Y}^{*\mathbf{D}}$, ce qui permet de construire la chaîne voulue.

Puisque $\mathbf{Y}^{*\mathbf{D}}$ est infini, il existe (cf. partie IV) une chaîne infinie $(\mathbf{X}_k)_k$ strictement croissante d'élément de $\mathcal{A}(\mathbf{Y}^{*\mathbf{D}})$ convergeant vers $\mathbf{Y}^{*\mathbf{D}}$. Il nous suffit alors de prouver qu'il existe un élément $\mathbf{X}_K$ tel que $F(\mathbf{X}) \sqsubset F(\mathbf{X}_K)$. Si ce n'était pas le cas, nous aurions $F(\mathbf{X}) = F(\mathbf{X}_k)$ pour tout k et donc par continuité de F, $F(\mathbf{X}) = F(\mathbf{Y}^{*\mathbf{D}})$ ce qui est impossible par minimalité de $\mathbf{Y}^{*\mathbf{D}}$.

7.3 Datations principales et réseaux ordonnés

La stabilité et particulièrement les inverses d'un réseau, apportent une grande expressivité. Avec leur aide, nous pouvons définir simplement les conditions nécessaires et suffisantes à l'existence d'une datation principale. Dans un premier temps, reformulons la correction :

7.3.1 *Propriété* **Datation correcte**
Pour tout réseau stable, le prédicat de correction s'écrit :

$$\mathrm{correct}(T) \qquad \Longleftrightarrow \qquad \forall \mathbf{D}, \mathbf{Y} \in \mathcal{A}(F(\mathbf{D})), \quad T_{\mathbf{D}}(\mathbf{Y}^{*\mathbf{D}}) < T_{\mathbf{D}}(\mathbf{Y}) < \infty$$

Grâce à la stabilité, nous proposons une autre manière de comprendre ce qu'est une datation principale avec la notion de datation représentative :

7.3.2 *Définition* **Datation représentative**
Pour tout flot d'entrées $\mathbf{D}$, une datation représentative d'un réseau stable ne date avant une sortie *que* les entrées nécessaires pour calculer cette sortie :

$$\mathrm{repr}(T) \iff \begin{array}{l} \mathrm{correct}(T) \quad \wedge \\ \forall \mathbf{D}, \mathbf{X} \sqsubseteq \mathbf{D}, \mathbf{Y} \sqsubseteq F(\mathbf{D}), \quad T_{\mathbf{D}}(\mathbf{X}) < T_{\mathbf{D}}(\mathbf{Y}) \iff \mathbf{X} \sqsubseteq \mathbf{Y}^{*\mathbf{D}} \wedge |\mathbf{X}| < \infty \end{array}$$

7.3.3 *Propriété* **Une datation représentative est principale**
Pour un réseau stable, si une datation est représentative, alors elle est équivalente à toutes les datations principales. La réciproque est aussi vraie (cf. propriété 7.3.13).

Démonstration Il suffit de prouver qu'une datation T représentative est principale. Prenons T' une datation correcte, si $T(\mathbf{X}) < T(\mathbf{Y})$ alors $\mathbf{X} \sqsubseteq \mathbf{Y}^{*\mathbf{D}}$ et $\mathbf{X}$ est fini.

Soit $\mathbf{Y}^{*\mathbf{D}}$ est fini, alors par propriété 7.2.17, nous avons l'existence de $\mathbf{Y}' \in \mathcal{A}(\mathbf{Y})$ tel que $\mathbf{Y}^{*\mathbf{D}} = \mathbf{Y}'^{*\mathbf{D}}$. Par monotonie et correction de T', nous avons $T'(\mathbf{X}) \leq T'(\mathbf{Y}^{*\mathbf{D}}) = T'(\mathbf{Y}'^{*\mathbf{D}}) < T'(\mathbf{Y}') \leq T'(\mathbf{Y})$ ce qui permet de conclure.

Soit $\mathbf{Y}^{*\mathbf{D}}$ est infini, mais alors par la propriété 7.2.18, nous avons une infinité d'inverses finis inclus dans $\mathbf{Y}^{*\mathbf{D}}$, il en existe au moins un supérieur à $\mathbf{X}$ (puisque $\mathbf{X}$ est fini inférieur à $\mathbf{Y}^{*\mathbf{D}}$), ce qui permet de retomber dans le cas où $\mathbf{Y}^{*\mathbf{D}}$ est fini.

L'existence d'une datation représentative est très intéressante puisque d'une certaine manière, la datation relative des entrées et des sorties encode exactement les dépendances de données. Nous le verrons dans le théorème 7.3.10, l'existence d'une datation représentative est équivalente au fait que le réseau soit *ordonné*.

7.3.4 *Définition* **Réseau ordonné**
La fonction F est ordonnée pour $\mathbf{D}$ ssi elle est stable et l'ensemble de ses inverses $\mathrm{Inv}(F, \mathbf{D})$ est totalement ordonné, avec :

$$\mathrm{Inv}(F, \mathbf{D}) \stackrel{\text{def}}{=} \{\mathbf{Y}^{*\mathbf{D}} \mid \mathbf{X} \in \mathcal{A}(\mathbf{D}), \mathbf{Y} \sqsubseteq F(\mathbf{X})\}$$

Un réseau est dit ordonné ssi sa fonction F l'est pour tout flot d'entrées $\mathbf{D}$.

Les travaux de Gérard Berry [Ber75] définissent la notion proche de fonction *deterministic* dans le cadre de l'étude des fonctions récursives. Nous n'avons pas souhaiter utiliser cette dénomination car nous la trouvons surchargée et les liens avec les définissions de G. Berry ne nous sont pas limpides.

7.3.5 *Remarque* **La trace au sens de Girard [Gir86]**
L'ensemble des inverses est une projection ne retenant que les éléments liés à $\mathbf{D}$ de ce qui est appelée la trace de F, définie par $Tr(F) = \{(\mathbf{X}, \mathbf{Y}) \mid |\mathbf{X}| < \infty, \mathbf{Y} \sqsubseteq F(\mathbf{X}), \forall \mathbf{X}' \sqsubseteq \mathbf{X}, \mathbf{Y} \not\sqsubseteq F(\mathbf{X}')\}$. Cette notion de trace, introduite par Girard [Gir86], est une manière de représenter la fonction F. L'ensemble des inverses est donc une représentation fidèle de F avec $\mathbf{D}$ comme entrée. Dire que le réseau est ordonné correspond donc à dire que sa trace est telle que pour tout éléments $(\mathbf{X}, \mathbf{Y})$ et $(\mathbf{X}', \mathbf{Y}')$ de la trace, $\mathbf{X} \uparrow \mathbf{X}'$ implique $\mathbf{X} \sqsubseteq \mathbf{X}'$ ou $\mathbf{X}' \sqsubseteq \mathbf{X}$.

Nous avons vu que le réseau de Gonthier n'avait pas de datation principale. Nous devons donc trouver qu'il n'est pas ordonné. La caractéristique d'un réseau ordonné est qu'à tout

moment, il y a un *unique* plus petit préfixe de l'entrée qui doit être consommé pour produire des sorties, ce qui assure que nous pouvons ordonner la consommation des entrées.

7.3.6 *Exemple* **Réseau stable et non ordonné : la fonction de Gonthier**
Reprenons l'exemple du réseau de Gonthier (cf. section 7.1.1). Ce réseau est trivialement stable par la propriété 7.2.3, mais n'est pas ordonné. En effet, l'ensemble

$$\mathrm{Inv}(F_{\mathrm{Gon}}, (\mathbf{a}, \mathbf{b})) = \{(\mathbf{x_a}, \mathbf{x_b}) \mid \mathbf{x_a} \sqsubseteq \mathbf{a}, \mathbf{x_b} \sqsubseteq \mathbf{b}\}$$

n'est pas totalement ordonné sauf si $\mathbf{a} = \varepsilon$ ou $\mathbf{b} = \varepsilon$. Avec $\mathbf{a} = a.\cdots$ et $\mathbf{b} = b.\cdots$, il a entre autres deux inverses (a, ε) et (ε, b) qui ne sont pas comparables.

Ces deux inverses correspondent aux deux datations que nous avons données, $T12$ requérant qu'il y ait en premier lieu une valeur sur la première entrée et symétriquement pour $T21$. Ces deux datations permettent deux usages qui ne sont pas interchangeables comme nous l'avons précédemment indiqué. Nous retrouvons ces deux stratégies en exprimant la fonction de transition de F_{Gon} sous une forme canonique Mealy [2] :

$$tF_{Gon}(s_0, (a, \varepsilon)) = ((a, \varepsilon), s_0) \qquad tF_{Gon}(s_0, (\varepsilon, b)) = ((\varepsilon, b), s_0)$$

Intuitivement, le réseau n'est pas ordonné du fait que $\mathrm{exec}(tF_{Gon})(a, b)$ a un choix à effectuer entre la transition utilisant (a, ε) et celle utilisant (ε, b), alors que toutes deux produisent une sortie. Ce choix non déterministe produit des datations qui correspondent à des usages différents.

Finalement, il est important de noter que le produit de deux fonctions ordonnées n'est pas forcément ordonné. En effet, l'identité est une fonction ordonnée mais la fonction de Gonthier, qui est le produit de l'identité avec l'identité, n'est pas ordonnée.

La principale caractéristique et utilité des réseaux ordonnés tient à l'existence de la chaîne des inverses et la chaîne des bi-inverses, qui décrivent intégralement le réseau :

7.3.7 *Propriété* **Chaines strictement croissantes des inverses et bi-inverses**
Un réseau ordonné admet une chaîne $(\mathbf{Y}_k^{*\mathbf{D}})_k$ contenant tous les inverses finis du réseau. La chaîne des inverses est strictement croissante, son premier élément est $\mathbf{Y}_1^{*\mathbf{D}} = \boldsymbol{\varepsilon}$, elle converge vers $(F(\mathbf{D}))^{*\mathbf{D}}$ et permet de tout calculer :

$$F(\mathbf{D}) = \lim_k F(\mathbf{Y}_k^{*\mathbf{D}}) = \lim_k \mathbf{Y}_k^{**\mathbf{D}}$$

Par monotonie, la chaîne $(\mathbf{Y}_k^{**\mathbf{D}})_k$ est strictement croissante. Elle contient $\boldsymbol{\varepsilon}$ ssi le réseau est Mealy.

Démonstration Puisque tous les inverses sont ordonnés, en les prenant dans l'ordre, nous construisons la chaîne $(\mathbf{Y}_k^{*\mathbf{D}})_k$, dont les éléments sont distincts deux à deux et produisent des sorties strictement croissantes par construction. Prouvons $F(\mathbf{D}) = \lim_k F(\mathbf{Y}_k^{*\mathbf{D}})$, ce qui donnera par continuité de F que $\lim_k \mathbf{Y}_k^{*\mathbf{D}} = (F(\mathbf{D}))^{*\mathbf{D}}$. Si le cardinal de $\mathrm{Inv}(F, \mathbf{D})$ est :

- fini égal à n, alors $F(\mathbf{D}) = F(\mathbf{Y}_n^{*\mathbf{D}})$ et $\left|\mathbf{Y}_n^{*\mathbf{D}}\right| < \infty$. Si ce n'était pas le cas, il faudrait que l'inverse de $F(\mathbf{D})$ ne soit pas fini, mais alors il y aurait une infinité d'inverses par la propriété 7.2.18.
- infini. Par monotonie des inverses, l'inverse de $F(\mathbf{D})$ ne peut pas être fini, sinon il y en aurait un nombre fini. Il est donc infini. Nous pouvons conclure par la propriété 7.2.18.

2. La confluence très forte est évidente en rendant le domaine clos par union avec $tF_{Gon}(s_0, (a, b)) = ((a, b), s_0)$.

Avant le théorème principal, considérons les conséquences de l'ordre entre les inverses.

7.3.8 *Propriété* **Caractérisation de $\mathbf{Y}_k^{*\mathbf{D}}$, inverse de rang k de $\mathbf{D}$**
L'indice k d'un inverse $\mathbf{Y}_k^{*\mathbf{D}}$ n'est pas caractéristique de $\mathbf{D}$, mais de l'inverse en lui-même :

$$\forall \mathbf{D}, k, \quad \#\operatorname{Inv}(F, \mathbf{Y}_k^{*\mathbf{D}}) = k \qquad \text{et} \qquad \forall \mathbf{D}', \quad \mathbf{Y}_k^{*\mathbf{D}} \sqsubseteq \mathbf{D}' \quad \Longrightarrow \quad \mathbf{Y}_k^{*\mathbf{D}'} = \mathbf{Y}_k^{*\mathbf{D}}$$

Nous pouvons donc parler du rang d'un inverse indépendamment de $\mathbf{D}$: $\mathbf{X}$ est un inverse de rang k ssi $\mathbf{X} = (\mathbf{F}(\mathbf{X}))^{*\mathbf{X}} = \mathbf{Y}_k^{*\mathbf{X}}$ et il sera le $k$ième inverse de toute entrée $\mathbf{D}$ l'incluant.

Démonstration Premièrement, par définition, nous avons $\operatorname{Inv}(F, \mathbf{Y}_k^{*\mathbf{D}}) \subseteq \operatorname{Inv}(F, \mathbf{D})$ qui sont tous deux totalement ordonnés, par monotonie des inverses, les éléments de $\operatorname{Inv}(F, \mathbf{Y}_k^{*\mathbf{D}})$ sont les éléments de $\operatorname{Inv}(F, \mathbf{D})$ inférieurs à $\mathbf{Y}_k^{*\mathbf{D}}$ qui sont au nombre de k.

Deuxièmement, si $\mathbf{Y}_k^{*\mathbf{D}} \sqsubseteq \mathbf{D}'$ alors nous avons par définition $\operatorname{Inv}(F, \mathbf{Y}_k^{*\mathbf{D}}) \sqsubseteq \operatorname{Inv}(F, \mathbf{D}')$ qui sont tous deux totalement ordonnés et les k premiers inverses de $\mathbf{D}'$ sont $\operatorname{Inv}(F, \mathbf{Y}_k^{*\mathbf{D}})$ donc $\mathbf{Y}_k^{*\mathbf{D}} = \mathbf{Y}_k^{*\mathbf{D}'}$.

7.3.9 *Propriété* **Les inverses de même rang sont incompatibles**
Deux inverses de même rang sont soit identiques soit incompatibles.

Démonstration Soit deux inverses compatibles de même rang $\mathbf{Y}_k^{*\mathbf{D_1}}$ et $\mathbf{Y}_k^{*\mathbf{D_2}}$. Puisqu'ils sont compatibles, nous pouvons poser $\mathbf{X} = \mathbf{Y}_k^{*\mathbf{D_1}} \sqcup \mathbf{Y}_k^{*\mathbf{D_2}}$. Par la propriété 7.3.8, $\mathbf{Y}_k^{*\mathbf{D_1}} = \mathbf{Y}_k^{*\mathbf{X}} = \mathbf{Y}_k^{*\mathbf{D_2}}$.

7.3.1 Datation principale

7.3.10 Théorème **Principalité, représentativité**
Un réseau de Kahn de sémantique F est ordonné ssi il admet une datation principale.

Démonstration Nous allons prouver ce théorème en trois étapes. Le lemme 7.3.12 montre que l'existence d'une datation principale implique que F soit ordonnée, puis le lemme 7.3.11 va nous permettre de construire à partir de F ordonnée une datation représentative qui par la propriété 7.3.3 est principale.

Lemme 7.3.11 (Construction de T_{repr})**.** *Pour tout réseau ordonné et tout flot d'entrées $\mathbf{D}$, il existe une datation représentative du réseau que l'on note T_{repr}*

Démonstration Puisque le réseau est ordonné, la propriété 7.3.7 donne la chaîne $(\mathbf{Y}_k^{*\mathbf{D}})_k$ qui contient *tous* les inverses finis de $\operatorname{Inv}(F, \mathbf{D})$. Nous utilisons cette chaîne et les instants $R^k =]k-1; k[$ (par exemple) pour construire la datation au plus tôt $T = T_{early((R^k, \mathbf{X}_k)_k)}$ (que le réseau soit réactif ou pas cf. remarque 6.1.33).

Prouvons que cette datation est représentative. Par la propriété 6.1.31 nous avons :

$$\forall \mathbf{X} \sqsubseteq \mathbf{D}, \forall \mathbf{Y} \sqsubseteq F(\mathbf{D}), \quad T(\mathbf{X}) < T(\mathbf{Y}) \iff \exists k, \mathbf{X} \sqsubseteq \mathbf{Y}_k^{*\mathbf{D}} \wedge \mathbf{Y} \not\sqsubseteq \mathbf{Y}_{k-1}^{**\mathbf{D}}$$

Prouver que la datation est représentative revient à prouver que $\exists k, \mathbf{X} \sqsubseteq \mathbf{Y}_k^{*\mathbf{D}} \wedge \mathbf{Y} \not\sqsubseteq \mathbf{Y}_{k-1}^{**\mathbf{D}}$ équivaut à la finitude de $\mathbf{X}$ et à avoir $\mathbf{X} \sqsubseteq \mathbf{Y}^{*\mathbf{D}}$:

Implication : les $\mathbf{Y}_k^{*\mathbf{D}}$ sont finis, donc $\mathbf{X}$ l'est. Or nous avons $\mathbf{Y} \not\sqsubseteq \mathbf{Y}_{k-1}^{**\mathbf{D}}$ donc (cf. propriété 7.2.15) $\mathbf{Y}^{*\mathbf{D}} \not\sqsubseteq (\mathbf{Y}_{\mathbf{k-1}}^{**\mathbf{D}})^{*\mathbf{D}} = \mathbf{Y}_{k-1}^{*\mathbf{D}}$, mais les inverses sont ordonnés donc $\mathbf{Y}_k^{*\mathbf{D}} \sqsubseteq \mathbf{Y}^{*\mathbf{D}}$.

Réciproque : Puisque $\mathbf{X}$ est fini et inférieur à un inverse, il est inférieur à un inverse fini (cf. la preuve de la propriété 7.3.3 utilisant la propriété 7.2.18 si $\mathbf{Y}^{*\mathbf{D}}$ est infini), donc il existe k tel que $\mathbf{X} \sqsubseteq \mathbf{Y}_k^{*\mathbf{D}} \sqsubseteq \mathbf{Y}^{*\mathbf{D}}$, puisque les inverses sont strictement croissants, $\mathbf{Y}^{*\mathbf{D}} \not\sqsubseteq \mathbf{Y}_{k-1}^{*\mathbf{D}}$.

Lemme 7.3.12 (Nécessité d'avoir un réseau ordonné)**.** *Pour tout réseau représenté par sa fonction F, l'existence d'une datation principale implique que F soit ordonnée.*

Démonstration **Dans un premier temps**, montrons que le réseau doit être stable. Si ce n'est pas le cas (cf. propriété 7.2.3), il existe deux entrées $\mathbf{X_a}$ et $\mathbf{X_b}$ finies et compatibles mais non comparables . Nous posons $\mathbf{D} = \mathbf{X_a} \sqcup \mathbf{X_b}$. Il existe de plus une sortie $\mathbf{Y}$ telle que $\mathbf{Y} \sqsubseteq F(\mathbf{X_a})$ et $\mathbf{Y} \sqsubseteq F(\mathbf{X_b})$ mais $\mathbf{Y} \not\sqsubseteq F(\mathbf{X_a} \sqcap \mathbf{X_b})$.

Nous construisons deux datations T_a et T_b symétriquement. Soit $T_a = T_{canon((R^k, \mathbf{X_a}, \mathbf{Y_a})_k)}$ avec la chaîne $(\mathbf{X_a}k)_k$ ne contenant que deux éléments, $\mathbf{X_a}1 = \mathbf{X_a}$ et $\mathbf{X_a}2 = \mathbf{D}$ et $\mathbf{Y_a}1 = \mathbf{Y}$ et $\mathbf{Y_a}2 = F(\mathbf{D})$. Ainsi, la datation est bien correcte avec $T_a(\mathbf{X_a}) < T_a(\mathbf{Y}) < T_a(\mathbf{D}) < T_a(F(\mathbf{D})) < \infty$ et de manière plus importante, puisque $\mathbf{X_b} \not\sqsubseteq \mathbf{X_a}$, la propriété 6.1.31 donne $T_a(\mathbf{Y}) \leq T_a(\mathbf{X_b})$.

S'il existait une datation principale T, elle serait telle que $T_a \sqsubsetneq T$. Ainsi par contraposé, $T(\mathbf{Y}) \leq T(\mathbf{X_b})$. Symétriquement, nous construisons T_b correcte telle que $T_b(\mathbf{Y}) \leq T_b(\mathbf{X_a})$ et donc $T(\mathbf{Y}) \leq T(\mathbf{X_a})$. Or T devrait être correcte et il existerait $\mathbf{X}$ tel que $T(\mathbf{X}) < T(\mathbf{Y})$ avec $\mathbf{Y} \sqsubseteq F(\mathbf{X})$. Cependant, $\mathbf{X}$ ne peut être inclus dans $\mathbf{X_a}$ *et* $\mathbf{X_b}$ par hypothèse (sinon $\mathbf{Y} \sqsubseteq F(\mathbf{X_a} \sqcap \mathbf{X_b})$).

Finalement, supposons que $\mathbf{X} \not\sqsubseteq \mathbf{X_a}$, la propriété 6.1.31 donne $T_a(\mathbf{Y}) \leq T_a(\mathbf{X})$ et donc $T(\mathbf{Y}) \leq T(\mathbf{X})$, ce qui est absurde. Ainsi T ne peut être principale et correcte, absurde.

Dans un second temps, montrons que le réseau doit être ordonné en plus d'être stable. La preuve est similaire bien que plus simple. À nouveau par l'absurde, si le réseau n'est pas ordonné, il existe $\mathbf{Y_a}^{*\mathbf{D}}$ et $\mathbf{Y_b}^{*\mathbf{D}}$, non comparables mais compatibles, donc strictement inclus dans $(\mathbf{Y_a} \sqcup \mathbf{Y_b})^{*\mathbf{D}}$. La propriété 7.2.18 nous permet de considérer qu'ils sont tous deux finis.

Nous créons à nouveau deux datations correctes $T_a = T_{early((R^k, \mathbf{X_a})_k)}$ avec la chaîne $(\mathbf{X_a}k)_k$ telle que $\mathbf{X_a}1 = \mathbf{Y_a}^{*\mathbf{D}}$ et $\mathbf{X_a}2 = (\mathbf{Y_a} \sqcup \mathbf{Y_b})^{*\mathbf{D}}$. Nous pouvons appliquer la propriété 6.1.31, pour affirmer que $T_a(\mathbf{Y_a}^{**\mathbf{D}}) \leq T_a(\mathbf{Y_b}^{*\mathbf{D}})$. Symétriquement pour T_b.

La datation principale T est donc telle que $T(\mathbf{Y_a}^{**\mathbf{D}}) \leq T(\mathbf{Y_b}^{*\mathbf{D}})$ et $T(\mathbf{Y_a}^{**\mathbf{D}}) \leq T(\mathbf{Y_b}^{*\mathbf{D}})$. Mais par minimalité des inverses et correction de T, nous avons aussi que $T(\mathbf{Y_a}^{*\mathbf{D}}) < T(_{\mathbf{a}}{}^{**\mathbf{D}})$ et $T(\mathbf{Y_b}^{*\mathbf{D}}) < T(\mathbf{Y_b}^{**\mathbf{D}})$, ce qui forme une boucle non satisfaisable, donc T n'existe pas.

Ce dernier lemme, en plus de finir la preuve du théorème, assure que toutes les datations principales sont représentatives.

7.3.13 *Propriété* **Les datations principales sont représentatives**
Toute datation principale est représentative et vice-versa.

Démonstration S'il existe une datation principale T alors le réseau est ordonné, il existe donc une datation représentative équivalente à T (cf. propriété 7.3.3) et T est représentative. La réciproque est due à la propriété 7.3.3.

7.3.2 Datation principale réactive

Dans ce chapitre, c'est l'existence d'une datation principale qui nous importe et donc, comme nous venons de le prouver, le fait que le réseau soit ordonné. Cependant, il ne faut pas oublier que notre finalité est de trouver des datations *réactives* [3].

Quand nous discutions des problèmes d'équité de l'ordonnancement du système de transition (cf. exemple 6.2.10), nous avions présenté le réseau F_{eqin} qui demandait, pour homogénéiser la représentation co-itérative des réseaux d'avoir des fonctions de transition produisant des flots infinis.

Nous pouvons maintenant aller un peu plus loin dans la compréhension de ce réseau et en particulier montrer qu'il est ordonné donc admet une datation principale, mais qu'il n'existe pas de datation principale réactive pour ce réseau.

7.3.2.1 Non réactivité de F_{eqin}

Rappelons sa définition :

$$F_{eqin}(\varepsilon,\varepsilon) = (\varepsilon,\varepsilon) \qquad F_{eqin}(x.\mathbf{x_1},\mathbf{x_2}) = (x^\omega, plus(\mathbf{x_1},\mathbf{x_2}))$$

Ainsi, les inverses finis pour l'entrée $\mathbf{D} = (x.\mathbf{x_1},\mathbf{x_2})$ sont $\mathbf{Y}_1^{*\mathbf{D}} = \boldsymbol{\varepsilon}$, $\mathbf{Y}_2^{*\mathbf{D}} = (x,\varepsilon)$ qui produit $\mathbf{Y}_1^{**\mathbf{D}} = (x^\omega,\varepsilon)$ et les $\mathbf{Y}_k^{*\mathbf{D}} = (x.\mathbf{a},\mathbf{b})$ avec $|\mathbf{a}| = |\mathbf{b}| = k-2$ et $\mathbf{a} \sqsubseteq \mathbf{x_1}$ et $\mathbf{b} \sqsubseteq \mathbf{x_2}$. Si $\mathbf{D} = (\varepsilon,\mathbf{x_2})$ alors le seul inverse est $\boldsymbol{\varepsilon}$. Tous ses inverses sont ordonnés donc le réseau l'est.

Par contre, le deuxième bi-inverse est de taille infinie. En prenant $\mathbf{D} = (x.x',y')$, le réseau a trois inverses, la construction de T_{repr} est telle que :

$$T_{\text{repr}}(\mathbf{Y}_2^{*\mathbf{D}}) < T_{\text{repr}}(\mathbf{Y}_2^{**\mathbf{D}}) < T_{\text{repr}}(\mathbf{Y}_3^{*\mathbf{D}}) < T_{\text{repr}}(\mathbf{Y}_3^{**\mathbf{D}}) < \cdots$$

Ce qui force $T_{\text{repr}}(\mathbf{Y}_2^{**\mathbf{D}})$ à être fini et donc empêche T_{repr} d'être réactive. La réactivité revient à ce que seuls des flots finis ont une date finie (cf. remarque 6.1.15).

De manière générale, ce réseau n'a pas de datation principale réactive. En effet, il faudrait $T(\mathbf{Y}_2^{**\mathbf{D}}) = \infty$, mais pour être principale T doit aussi être équivalente à T_{repr}, donc avoir $T(\mathbf{Y}_2^{**\mathbf{D}}) \leq T(\mathbf{Y}_3^{*\mathbf{D}}) < T(\mathbf{Y}_3^{**\mathbf{D}})$, ce qui est impossible.

7.3.2.2 Des réseaux réactif mais pas interactifs

Les réseau non intéractifs, mais ayant une datation réactive sont un peu spéciaux, mais très importants, puisque c'est le cas des constantes infinies.

Un réseau constant ne consomme pas d'entrée pour produire ses sorties. Prenons le plus simple : $F_{zero}() = 0^\omega$. Ce réseau a pour seul inverse $\boldsymbol{\varepsilon}$ et bi-inverse 0^ω. Il est clair que nous pouvons donner une datation principale réactive à ce réseau : s le port de sortie, $T(s)[k] = k$ est une datation réactive (que nous rajoutions un port d'entrée e ou pas, puisque nous aurions $T(e)[0] = 0$ et $T(e)[k] = \infty$).

Un réseau plus complexe et proche de F_{eqin} est F_{oneo}, défini par :

$$F_{oneo}(\varepsilon) = \varepsilon \qquad F_{oneo}(x.\mathbf{x}) = x^\omega$$

Ce réseau a deux inverses finis pour toute entrée $\mathbf{D} = x.\mathbf{x}$, égaux à ε et x dont le bi-inverse est $\mathbf{Y}_2^{**\mathbf{D}} = x^\omega$. La datation T_{repr} est telle que $T_{\text{repr}}(\mathbf{Y}_2^{*\mathbf{D}}) < T_{\text{repr}}(\mathbf{Y}_2^{**\mathbf{D}})$. Hormis la construction de T_{repr}, il n'y a pas de borne supérieure imposée à $T_{\text{repr}}(\mathbf{Y}_2^{**\mathbf{D}})$, elle pourrait donc être infinie. La correction de la datation oblige uniquement les *approximants* de $\mathbf{Y}_2^{**\mathbf{D}}$ à avoir une date finie. Par exemple la datation T définie par $T(e)[1] = 1$, $T(e)[k>1] = \infty$ et $T(s)[k] = k+1$ (avec e le port d'entrée et s celui de sortie) est principale et réactive.

3. Rappelons qu'une datation réactive est une datation qui n'a pas de point d'accumulation, cf. propriété 6.1.12.

7.3.2.3 Réseaux ordonnés réactifs

Cela doit maintenant être clair, pour avoir une datation principale réactive, un réseau doit être ordonné et tous ses bi-inverses, sauf le dernier s'il existe, doivent être finis. Un tel réseau sera dit ordonné réactif.

7.3.14 *Définition* **Réseau ordonné réactif**
Un réseau sera dit ordonné réactif ssi il est ordonné et ses bi-inverses sont finis, excepté pour le dernier s'il existe. Formellement, la chaîne des inverses $(\mathbf{Y}_k^{*\mathbf{D}})_k$ du réseau est de longueur q (possiblement égal à ∞) et telle que :

$$\forall k < q, \quad \left|\mathbf{Y}_k^{**\mathbf{D}}\right| < \infty$$

7.3.15 Théorème **Datation principale réactive : réseau ordonné réactif**
Un réseau admet une datation principale réactive ssi il est ordonné réactif.

Démonstration Nous avons déjà par le théorème 7.3.10 l'équivalence ordonné/réactif. Nous pouvons donc raisonner avec la chaîne des inverses finis du réseau $(\mathbf{Y}_k^{*\mathbf{D}})_k$. Les inverses consommés par cette datation sont finis et ne posent donc pas de problème. Par contre, par construction de T_{repr}, nous avons que toute datation principale est telle que $T(\mathbf{Y}_{k-1}^{**\mathbf{D}}) \leq T(\mathbf{Y}_k^{*\mathbf{D}})$: tout bi-inverse ayant un inverse de rang supérieur doit être fini pour avoir une datation réactive. Réciproquement, si c'est le cas et que le nombre d'inverses est infini, alors T_{repr} est réactive. Si le nombre d'inverses est fini égal à q, le dernier bi-inverse peut être infini, car ses dates ne sont pas majorées. La datation T_{repr} peut être adaptée pour étaler les dates du dernier bi-inverses : nous prenons une chaîne approximant ce bi-inverse et donnons pour le $i$ième élément la date $T(\mathbf{Y}_q^{*\mathbf{D}}) + i$. La datation est alors principale et réactive.

7.3.16 *Remarque* **Réseau ordonné réactif non interactif**
Tous les réseaux ordonnés interactifs (cf. définition 6.1.23) sont ordonnés réactifs. Les seuls réseaux ordonnés réactifs non interactifs sont ceux ayant un nombre fini d'inverses avec le dernier bi-inverse infini.

7.3.3 Fonction de transition d'un réseau ordonné

Nous utiliserons par la suite la condition suffisante suivante pour prouver qu'un réseau est ordonné :

7.3.17 *Propriété* **Clôture par intersection compatible du domaine de tF**
Toute fonction de transition confluente et fortement normalisante, définit une réseau de Kahn ordonné si son domaine de définition est clos par intersections compatibles :

$$\forall s, \mathbf{X_1} \uparrow \mathbf{X_2}, (s, \mathbf{X_1}) \in \text{dom}(tF) \wedge (s, \mathbf{X_2}) \in \text{dom}(tF) \implies (s, \mathbf{X_1} \sqcap \mathbf{X_2}) \in \text{dom}(tF)$$

En particulier, s'il existe un préfixe de l'entrée $\mathbf{X}$ qui appartient au domaine de définition de tF, alors il en existe un plus petit.

Démonstration Pour tout $\mathbf{D}$, considérons deux inverses finis $\mathbf{Y_1}^{*\mathbf{D}}$ et $\mathbf{Y_2}^{*\mathbf{D}}$ et montrons qu'ils sont ordonnés. Si l'un des deux est ε, nous avons fini. Considérons qu'ils ne sont pas

ordonnés. Par définition ils sont compatibles, nous posons $\mathbf{U} = {\mathbf{Y_a}}^{*\mathbf{D}} \sqcup {\mathbf{Y_b}}^{*\mathbf{D}}$. Considérons la suite de transition $(\mathbf{U}_{\delta k}, s_k, \mathbf{V}_{\delta k})_k$ terminale, de longueur n, à partir de $0_{\mathcal{S}}$, pour l'entrée $\mathbf{U}$, telle que $\mathbf{U}_{\delta k}$ est toujours le plus petit possible. Une telle suite existe par hypothèses.

Nous allons montrer qu'il existe k_i tel que $\mathbf{U}_{k_i} = {\mathbf{Y_i}}^{*\mathbf{D}}$ pour $i = 1, 2$, ce qui permettra de conclure puisque $(\mathbf{U}_k)_k$ est une chaîne.

Soit $m_i = \max\{k \mid \mathbf{U}_k \sqsubset {\mathbf{Y_i}}^{*\mathbf{D}}\}$, ce maximum existe puisque $\boldsymbol{\varepsilon} \sqsubset {\mathbf{Y_i}}^{*\mathbf{D}} \sqsubseteq \mathbf{U}_n$ et nous posons comme toujours $\mathbf{U}_0 = \boldsymbol{\varepsilon}$.

Soit $0_{\mathcal{S}} \xrightarrow{\mathbf{U}_{m_i}/\mathbf{V}_{m_i}}{}^* s_i$, la confluence permet d'affirmer qu'il existe une transition $s_i \xrightarrow{\mathbf{X}/\mathbf{Y}}$ telle que $\mathbf{U}_{m_i}.\mathbf{X} \sqsubseteq {\mathbf{Y_i}}^{*\mathbf{D}}$. Mais ${\mathbf{Y_i}}^{*\mathbf{D}} \sqsubseteq \mathbf{U}$, ainsi par construction, nous avons $\mathbf{U}_{m_i+1} \sqsubseteq \mathbf{X}$ (puisque nous prenons toujours la plus petite transition possible). Ce qui donne $\mathbf{U}_{m_i+1} \sqsubseteq \mathbf{Y_i}$ et donc par maximalité de m_i, nous obtenons $\mathbf{X}_{m_i+1} = {\mathbf{Y_i}}^{*\mathbf{D}}$.

7.3.4 Constructions de réseaux ordonnés

La première remarque importante à faire concerne les réseaux ayant une seule entrée ou bien une seule sortie. Ces réseaux recouvrent une bonne partie des réseaux qui nous intéressent et nous avons la garantie qu'ils sont ordonnés.

7.3.18 *Propriété* **Réseau à une seule sortie ou une seule entrée**
Si un réseau F n'a qu'un seul port d'entrée, il est ordonné, idem avec un seul port de sortie.

Démonstration Si $\dim(\mathbf{D}) = 1$, l'ensemble des $\mathbf{X} \sqsubseteq \mathbf{D}$ est totalement ordonné, donc l'ensemble $\mathrm{Inv}(F, \mathbf{D})$ est totalement ordonnée. Si un réseau F n'a qu'une seule sortie, $\dim(\mathbf{F}(\mathbf{D})) = 1$, l'ensemble des $\mathbf{Y} \sqsubseteq F(D)$ est totalement ordonné. Par monotonie de l'inverse (cf. propriété 7.2.16), l'ensemble $\mathrm{Inv}(F, \mathbf{D})$ est totalement ordonnée.

7.3.4.1 Composition et point fixe de réseaux ordonnés

Bien que le produit de deux fonctions ordonnées ne soit pas ordonné, leur composition et point fixe sont ordonnés.

7.3.19 *Propriété* **Composition**
La composition $G \circ F$ de deux fonctions stables est ordonnée si au moins l'une des deux l'est. De plus, ses inverses sont inclus dans ceux de F et ses bi-inverses dans ceux de G.

Démonstration Nous cherchons à prouver que l'ensemble des inverses de $G \circ F$ est totalement ordonné :

$$\forall \mathbf{D}, \mathrm{Inv}(G \circ F, \mathbf{D}) = \left\{ \mathbf{X} \;\middle|\; \mathbf{X} \xrightarrow{G\circ F}_{\mathbf{D}} \mathbf{Z} \right\} \tag{7.1}$$

$$= \left\{ \mathbf{X} \;\middle|\; \mathbf{X} \xrightarrow{F}_{\mathbf{D}} \mathbf{Y} \xrightarrow{G}_{F(\mathbf{D})} \mathbf{Z} \right\} \tag{7.2}$$

$$= \left\{ \mathbf{Y}^{*\mathbf{F}} \;\middle|\; \mathbf{Y} \in \mathrm{Inv}(G, F(\mathbf{D})) \right\} \tag{7.3}$$

Le passage à l'équation (7.2) est donné par la monotonie de l'inverse (cf. propriété 7.2.16) : pour trouver le plus petit $\mathbf{X}$ tel que $\mathbf{Z} \sqsubseteq G \circ F(\mathbf{X})$, il suffit de chercher le plus petit $\mathbf{X}$ tel que $\mathbf{Y} \sqsubseteq F(\mathbf{X})$ pour $\mathbf{Y}$ le plus petit tel que $\mathbf{Z} \sqsubseteq G(\mathbf{Y})$.

L'équation (7.3) est une réécriture qui permet d'aisément conclure :

— Comme $\mathrm{Inv}(G \circ F, \mathbf{D}) \subseteq \mathrm{Inv}(F, \mathbf{D})$, si F est ordonnée, la composition aussi.

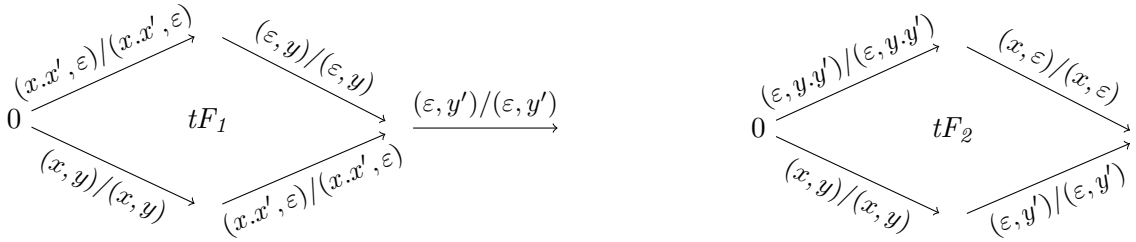
— Si G est ordonnée, $\mathrm{Inv}(G, F(\mathbf{D}))$ est totalement ordonné, donc par monotonie de l'inverse, $\mathrm{Inv}(G \circ F, \mathbf{D})$ l'est aussi.

Ce résultat est très intéressant car il va nous permettre d'utiliser les fonctions de manipulations des flots qui sont des fonctions stables (croisement de ports, identités multiples comme le réseau de Gonthier, etc). Nous en discutons plus en détail en section 7.3.4.3.

Cependant, la réciproque n'est pas vraie, ce qui complique donc grandement les choses :

7.3.20 *Remarque* **Réseau ordonné formé de fonctions non ordonnées**
Il existe des réseaux ordonnés qui sont formés par la composition de réseaux qui ne sont pas ordonnés. La mise en parallèle en offre de multiples exemples (cf. remarque 7.3.26).

Un exemple minimal nécessite deux fonctions de deux entrées et deux sorties chacune :



Ces deux fonctions de transition sont bien confluentes, fortement normalisantes et stables. De plus elles calculent l'identité(partiellement définie), donc le bi-inverse est égal à l'inverse. Les inverses de F_1 sont $\{(x.x', \varepsilon); (x, y); (x.x', y); (x.x', y.y')\}$ et ne sont pas ordonnés à cause des deux premiers. Les inverses de F_2 sont $\{(x, y); (\varepsilon, y.y'); (x, y.y')\}$ et non ordonnés.

La composition est telle que l'inverse de (x, y) par F_2 est lui-même puis par F_1 encore lui-même. Par contre, l'inverse de $(\varepsilon, y.y')$ est lui-même par F_2, mais est $(x.x', y.y')$ par F_2. Finalement l'inverse de $(x, y.y')$ est aussi $(x.x', y.y')$. L'ensemble des inverses de $F_2 \circ F_1$ est $\{(x, y); (x.x', y.y')\}$, ce qui rend cette composition ordonnée.

7.3.21 *Propriété* **Réactivité**
Si G est ordonnée réactive, alors pour toute fonction stable F, la composition $G \circ F$ est ordonnée réactive.

Démonstration La propriété 7.3.19 assure que la composition est ordonnée. De plus, les bi-inverses de la composition $G \circ F$ sont inclus dans ceux de G.

7.3.22 *Propriété* **Point fixe**
Le point fixe d'une fonction ordonnée est ordonné.

Démonstration Le point fixe est une conséquence immédiate de la composition car le point fixe préserve la stabilité et le point fixe est égal au point fixe composé avec la fonction initiale qui est ordonnée, donc le point fixe est ordonné.

7.3.4.2 Restrictions sur le produit et la mise en parallèle de réseaux ordonnés

L'exemple du nœud de Gonthier démontre que dans le cas général le produit de deux réseaux ordonnés n'est pas ordonné. Nous donnons ici deux propriétés simples qui peuvent aider à vérifier si le produit ou la mise en parallèle de deux réseaux ordonnés est ordonné.

7.3.23 *Propriété* **Produit**
Soit deux réseaux $F_1 : \mathcal{D}_{E1} \to \mathcal{D}_{S1}$ et $F_2 : \mathcal{D}_{E2} \to \mathcal{D}_{S2}$, leur produit $F(\mathbf{X_1}, \mathbf{X_2})$ égal à

$(F_1(\mathbf{X_1}), F_2(\mathbf{X_2}))$ de type $(\mathcal{D}_{E1} \times \mathcal{D}_{E2}) \to (\mathcal{D}_{S1} \times \mathcal{D}_{S2})$ a pour ensemble d'inverses le produit des inverses de ses composantes :

$$\forall \mathbf{D}, \quad \mathrm{Inv}(F, \mathbf{D}) = \mathrm{Inv}(F_1, \mathbf{D}) \times \mathrm{Inv}(F_2, \mathbf{D})$$

7.3.24 *Propriété* **Restriction sur le produit**
Le produit de deux fonctions ordonnées est ordonné ssi au moins l'une d'elles est une constante.

Démonstration L'ensemble des inverses du produit inclut les ensembles $\{(\mathbf{X}, \varepsilon) \mid \mathbf{X} \in \mathrm{Inv}(F_1, \mathbf{D})\}$ et l'ensemble $\{(\varepsilon, \mathbf{X}) \mid \mathbf{X} \in \mathrm{Inv}(F_2, \mathbf{D})\}$. Les éléments de ces ensembles ne peuvent être ordonnés que si l'un des deux est réduit au singleton $\{\varepsilon\}$.

Remarquez, qu'il est immédiat d'affirmer que le réseau de Gonthier n'est pas ordonné. Contrairement au produit, la mise en parallèle partage les entrées.

7.3.25 *Propriété* **Mise en parallèle**
Soit deux fonction $F_1 : \mathcal{D}_E \to \mathcal{D}_{S1}$ et $F_2 : \mathcal{D}_E \to \mathcal{D}_{S2}$, alors leur mise en parallèle $F_1 \| F_2(\mathbf{X}) = (F_1(\mathbf{X}), F_2(\mathbf{X})) : \mathcal{D}_E \to (\mathcal{D}_{S1} \times \mathcal{D}_{S2})$ est telle que :

$$\forall \mathbf{D}, \quad \mathrm{Inv}(F_1 \| F_2, \mathbf{D}) = \mathrm{Inv}(F_1, \mathbf{D}) \cup \mathrm{Inv}(F_2, \mathbf{D})$$

Vérifier que le produit de deux fonction ordonnées dont on a pour chacune l'ensemble ordonné des inverses revient à ce que la fusion triée des deux ensembles succède.

7.3.26 *Remarque* La mise en parallèle est un cas particulier de la composition du réseau non ordonné dupliquant les entrées (avec deux entrées : $F_{dup}(\mathbf{x_1}, \mathbf{x_2}) = (\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_1}, \mathbf{x_2})$) et du produit des deux fonctions. Ni le produit des deux fonctions, ni le réseau de duplication ne sont ordonnés. Pourtant $F \| F$ est toujours ordonné.

Puisque l'ensemble des inverses est l'union des deux, la mise en parallèle de deux réseaux ordonnés réactifs, même ordonnée, n'est pas forcément réactive. Ce que nous avions d'ailleurs déjà remarqué en section 7.3.2 avec le réseau ordonné F_{eqin} non réactif, provenant de la mise en parallèle de deux réseaux réactifs ordonnés.

7.3.27 *Propriété* **Réactivité**
Pour être ordonnée réactive, la mise en parallèle $F_1 \| F_2$ doit être ordonnée et avoir au pire un bi-inverse infini en dernière position (cf. définition 7.3.14). Soit le flot d'entrées $\mathbf{D}$, $i = 1, 2$ et k_i^∞ le cardinal de l'ensemble des inverses finis de F_i. Si un des deux derniers bi-inverses est infini, alors l'autre est inclus dedans. Par exemple $k_1^\infty < \infty$ et $\left|\mathbf{Y}_{k_1^\infty}^{F_1 * * \mathbf{D}}\right| = \infty$ implique $\mathbf{Y}_{k_2^\infty}^{F_2 * \mathbf{D}} \sqsubseteq \mathbf{Y}_{k_1^\infty}^{F_1 * \mathbf{D}}$ et donc $k_2^\infty < \infty$.
En particulier, ceci permet d'affirmer que si la mise en parallèle de deux réseaux interactifs est ordonnée, elle est aussi interactive.

Démonstration Les bi-inverses sont donnés par l'ensemble $\{(F_1(\mathbf{X}), F_2(\mathbf{X})) \mid \mathbf{X} \in \mathcal{A}(\mathbf{D})\}$ qui est ordonné par hypothèse. Chacun de ces éléments est le produit d'un bi-inverse de F_1 avec un de F_2 : $\forall \mathbf{X} \in \mathcal{A}(\mathbf{D}), \exists k_1, k_2, \quad (F_1(\mathbf{X}), F_2(\mathbf{X})) = (\mathbf{Y}_{k_1}^{F_1 * * \mathbf{D}}, \mathbf{Y}_{k_2}^{F_2 * * \mathbf{D}})$. Les k_i sont croissants avec le bi-inverse et si k_i^∞ est fini alors il est atteint, ce qui permet de conclure. En

effet, pour être réactif, il faut que seul le dernier bi-inverse, s'il existe, soit de taille infinie. Trois cas de figures sont à vérifier. Le bi-inverse est composé de bi-inverses qui, respectivement,

— ne sont pas les derniers pour F_1 et F_2. Dans ce cas ils sont finis, leur produit aussi.
— un des deux est dernier, il ne doit alors pas être infini puisqu'il y a des bi-inverses du réseau plus grands et seul le plus grand peut l'être.
— les deux sont derniers, pas de contraintes.

7.3.4.3 Palliatifs

Bien que le produit de deux fonctions ordonnées ne le soient pas, le produit de deux fonctions stables est stable. Ainsi, la composition $G \circ (F_1 \times \cdots \times F_n)$, d'une fonction ordonnée (resp. réactive) G avec un produit de fonctions stables F_1, …, F_n est ordonnée (resp. réactive), par les propriétés 7.3.19 et 7.3.21.

7.3.28 *Nota bene* De nombreuses compositions et manipulations de fonctions ordonnées sont possibles puisqu'entres autres, l'identité, le produit et la projection sont des fonctions stables.

Parmi les conséquences de cette remarque, donnons deux propriétés élémentaires utiles :

7.3.29 *Propriété* **Composition partielle**
La composition partielle $G \circ (F \times I)$, d'une fonction ordonnée $G : \mathcal{D}_1^{ge} \times \cdots \times \mathcal{D}_d^{ge} \to \mathcal{D}_1^{gs} \times \cdots \times \mathcal{D}_r^{gs}$ avec la fonction stable $F : \mathcal{D}_1^{fe} \times \cdots \times \mathcal{D}_{d'}^{fe} \to \mathcal{D}_1^{ge} \times \cdots \times \mathcal{D}_{r'}^{ge}$ avec $r' \leq d$ et la fonction identité $I : \mathcal{D}_{r'}^{ge} \times \cdots \times \mathcal{D}_d^{ge} \to \mathcal{D}_{r'}^{ge} \times \cdots \times \mathcal{D}_d^{ge}$. Si G est réactive la composition aussi.

7.3.30 *Propriété* **Point fixe partiel**
Similairement, le point fixe $Y_{1\cdots r}(F) : \mathcal{D}_{r+1} \times \cdots \mathcal{D}_d$ de $F : \mathcal{D}_1 \times \cdots \times \mathcal{D}_d \to \mathcal{D}_1 \times \cdots \times \mathcal{D}_d$, effectué avec seulement les $r < d$ première sorties sur les r premières entrées est ordonné si F l'est. Il est de plus réactif si F l'est.

7.4 Forme normale, fonction de transition et séquentialité

Tous les réseaux ne sont pas ordonnés, cependant, ils sont les seuls à admettre une datation principale et donc à avoir une interface de communication simple et déterministe qui ne restreint pas la sémantique du réseau.

Dans l'optique d'un langage de programmation, la première chose à noter est que les réseaux à une seule sortie ou une seule entrée sont ordonnés (cf. propriété 7.3.18). Dans les faits cela recouvre une grande partie des programmes. Deuxièmement, un réseau non ordonné peut toujours se découper en sous-réseaux à une seule sortie donc ordonnés, au prix d'une possible duplication de code. Inversement, il peut être à la charge du programmeur de sur-spécifier un réseau avec des annotations pour le rendre ordonné, donc au compilateur de rejeter le programme tant que ce n'est pas le cas.

Ce compromis entre modularité et taille de code est toujours délicat, mais nous sommes confiants que des solutions intelligentes au niveau langage existent. Nous nous concentrons donc sur les réseaux ordonnés qui admettent une forme normale « optimale » pour la compilation en une fonction de transition.

7.4.1 Fonction de transition déterministe et forme normale tnF

Nous allons voir qu'un réseau ordonné admet une fonction de transition sous forme normale ayant deux avantages, le déterminisme d'exécution et la maximalité de la granularité.

7.4.1 *Définition* **Fonction de transition déterministe**
Une fonction de transition tF fortement normalisante est dite déterministe ssi pour tout état, les éléments de son domaine sont incompatibles :

$$\forall s, \forall \mathbf{X} \neq \mathbf{X}', \quad (s, \mathbf{X}) \in \mathrm{dom}(tF) \wedge (s, \mathbf{X}') \in \mathrm{dom}(tF) \quad \Longrightarrow \quad \nexists \mathbf{D}, \mathbf{X} \sqsubseteq \mathbf{D} \wedge \mathbf{X}' \sqsubseteq \mathbf{D}$$

7.4.2 *Propriété* tF **déterministe équivaut à l'unicité de la suite de transition**
La fonction de transition tF est déterministe ssi pour toute entrée $\mathbf{D}$, le calcul $\mathrm{exec}(tF)(\mathbf{D})$ admet une et une seule suite terminale de transition.

Il faut noter qu'une fonction de transition déterministe est automatiquement confluente et donc décrit un réseau. De plus ce réseau est ordonné :

7.4.3 *Propriété* tF **déterministe implique réseau ordonné**
L'unicité, pour toute entrée, de la suite de transition effectuée par le système permet de construire dans l'ordre l'ensemble des inverses pour cette entrée. Ainsi le réseau est ordonné.

La réciproque n'est pas vraie dans le cas général, en particulier s'il y a des transitions avec ε. Cependant, pour tout réseau ordonné, nous sommes capable de construire une fonction de transition déterministe qui va de plus être une forme normale.

7.4.4 *Définition* **Forme normale déterministe**
Une fonction de transition tF sera dite en forme normale déterministe, ssi elle est déterministe et produit toujours des sorties maximales et non nulles :

— Productivité : Les transitions produisent

$$tF(s, \mathbf{X}) = (\mathbf{Y}, s') \implies \mathbf{Y} \neq \varepsilon$$

— Maximalité : Les transitions passées ont produit le maximum de sortie

$$\forall (s, \mathbf{X}) \in \mathrm{dom}(tF), \quad \mathbf{X} = \varepsilon \implies s = 0_{\mathcal{S}}$$

Nous retrouvons à nouveau la distinction entre les réseaux Mealy et les réseaux Moore, ces derniers nécessitant au premier instant d'effectuer une transition sans recevoir d'entrée.

Notre notion de forme normale ne tient pas compte de l'espace d'état et des états utilisés. La raison est que les états correspondent au choix de l'implémentation et que nous ne souhaitons pas en dépendre et encore moins inhiber des optimisations. Par contre, cette forme définit de manière unique les étapes du calcul effectuées par le réseau :

7.4.5 *Propriété* **Unicité du calcul**
Deux fonctions de transition tF_1 et tF_2, en forme normale déterministe d'un même réseau, définissent deux suites de transition qui ne diffèrent que par les états utilisés. Pour toute entrée $\mathbf{D}$ et $i = 1, 2$, soit $(s^i_{k-1}, \mathbf{X^i}_{\delta k}, \mathbf{Y^i}_{\delta k}, s^i_k)_k$ les suites terminales de transition de longueur n^i avec $s^i_0 = 0_{\mathcal{S}}$, alors $n^1 = n^2$et pour tout k, $\mathbf{X^1}_{\delta k} = \mathbf{X^2}_{\delta k}$ et $\mathbf{Y^1}_k = \mathbf{Y^2}_k$.

Démonstration Pour toute entrée $\mathbf{D}$, nous prouvons par induction.

Au tout début des calculs, les deux suites sont également vides. Considérons que les deux suites sont arrivées dans les états s^1 resp. s^2 en ayant toutes deux consommé $\mathbf{X} \sqsubseteq \mathbf{D}$ et produit $\mathbf{Y}$. Alors :

— Soit $F(\mathbf{X}) = F(\mathbf{D})$ et alors par la productivité, il n'existe plus de transition pour tF_i et l'induction termine.

— Soit $F(\mathbf{X}) \sqsubset F(\mathbf{D})$ et alors par déterminisme, il existe un unique $\mathbf{X^i}_\delta$ dans le domaine de tF_i tel que $\mathbf{X}.\mathbf{X^i}_\delta \sqsubseteq \mathbf{D}$. De plus par productivité, $tF_i(s^i, \mathbf{X^i}_\delta) = (\mathbf{Y^i}_\delta, q^i)$ avec $\mathbf{Y^i}_\delta \neq \varepsilon$. Mais alors $\mathbf{X^1}_\delta = \mathbf{X^2}_\delta$ puisque sinon, par unicité de $\mathbf{X^1}_\delta$, il n'y a pas de transition pour tF_1 consommant $\mathbf{X^2}_\delta$: $\mathrm{exec}(tF_1)(\mathbf{X}.\mathbf{X^2}_\delta) = \mathbf{Y}$ qui est strictement inférieur à $\mathrm{exec}(tF_2)(\mathbf{X}.\mathbf{X^2}_\delta) = \mathbf{Y}.\mathbf{Y^2}_\delta.\cdots$, impossible car elles représentent le même réseau. Finalement, $\mathrm{exec}(tF_1)(\mathbf{X}.\mathbf{X^2}_\delta) = \mathbf{Y}.\mathbf{Y^1}_\delta = \mathrm{exec}(tF_2)(\mathbf{X}.\mathbf{X^2}_\delta) = \mathbf{Y}.\mathbf{Y^2}_\delta$.

Lemme 7.4.6 (Construction de la fonction de transition tnF)**.** *Tout réseau ordonné admet une fonction de transition tnF en forme normale déterministe. Son état est réduit à un entier, de plus, pour tout flot d'entrées $\mathbf{D}$, si $(\mathbf{Y}^{*\mathbf{D}}_k)_k$ est sa chaîne d'inverses, alors le calcul du réseau est la suite de transition $\left(k-1, \mathbf{Y}^{*\mathbf{D}}_{\delta z}, \mathbf{Y}^{**\mathbf{D}}_{\delta z}, k\right)_{k>0}$ avec $z = k$ pour un réseau Moore et $z = k+1$ pour un réseau Mealy.*

Démonstration Premièrement, rappelons que $\mathbf{Y}^{*\mathbf{D}}_1 = \varepsilon$ et que $\mathbf{Y}^{**\mathbf{D}}_1$ est égal à ε pour un réseau Mealy et différent pour un Moore. C'est pour cela que nous supprimons pour les réseaux Mealy cette première étape qui ne fait rien. Nous prenons l'espace d'état $(\mathcal{S}, 0_\mathcal{S}) = (\mathbb{N}, z)$ et pour toute entrée $\mathbf{D}$ et tout k compris entre 1 et $\#\,\mathrm{Inv}(F, \mathbf{D})$ moins 1 si Mealy,

$$tnF(\mathbf{Y}^{*\mathbf{D}}_{\delta z}, k-1) = (\mathbf{Y}^{**\mathbf{D}}_{\delta z}, k)$$

Vérifions qu'elle est en forme normale déterministe :

— Maximalité : Puisque pour tout $\mathbf{D}$, la chaîne $(\mathbf{Y}^{*\mathbf{D}})_k$ est strictement croissante, $\mathbf{Y}^{*\mathbf{D}}_{\delta k}$ est différent de ε pour tout $k > 1$, elle n'est égale à ε qu'au premier pas d'un réseau Moore.

— Productivité : La chaîne $(\mathbf{Y}^{**\mathbf{D}})_k$ est strictement croissante, si elle vaut ε en $k = 1$ alors le réseau est Mealy et ce bi-inverse n'est pas utilisé grâce à $z = k + 1$.

— Déterminisme : Pour tout état $s = k - 1$, les transitions consomment des entrées incompatibles. En effet, prenons deux transitions $s \xrightarrow{\mathbf{X}_\delta/\mathbf{Y}_\delta}$ et $s \xrightarrow{\mathbf{X'}_\delta/\mathbf{Y'}_\delta}$ avec $\mathbf{X}_\delta \uparrow \mathbf{X'}_\delta$. Puisque $\mathbf{X}_\delta$ est dans le domaine de définition, il existe $\mathbf{X}$ inverse de rang $k-1$, tel que $\mathbf{X}.\mathbf{X}_\delta$ est un inverse de rang k. Par la propriété 7.3.8, c'est aussi l'inverse de rang k de $\mathbf{X}.(\mathbf{X}_\delta \sqcup \mathbf{X'}_\delta)$, mais par productivité, $\mathbf{X}.\mathbf{X'}_\delta$ produit une sortie plus grande que $\mathbf{X}$, donc $\mathbf{X}.\mathbf{X'}_\delta \sqsubseteq \mathbf{X}.\mathbf{X}_\delta$, et comme $\mathbf{X}$ est fini $\mathbf{X'}_\delta \sqsubseteq \mathbf{X}_\delta$. Symétriquement, nous trouvons que $\mathbf{X}_\delta = \mathbf{X'}_\delta$.

7.4.7 Théorème **Forme normale de la fonction de transition des réseaux ordonnés**
Tout réseau ordonné peut s'écrire avec une fonction de transition en forme normale déterministe. Réciproquement, toute fonction de transition déterministe définit un réseau ordonné.

Avant de regarder plus en détail cette forme normale, notons que sa construction et son existence correspondent exactement à celles de la datation principale T_{repr}. Elle est de plus optimale en terme de granularité :

7.4.8 *Propriété* tnF **maximise la granularité**
Toute fonction de transition d'un réseau est de granularité inférieur ou égale à celle de tnF.

Vue d'une autre manière, la forme normale des réseaux ordonnés calcule en communiquant un nombre minimum de fois.

Démonstration Considérons un état s faisant partie de la suite de transition terminale de tnF avec $\mathbf{D}$ en entrée : $0_{\mathcal{S}} \xrightarrow{\mathbf{A/B}}{}^{*} s \xrightarrow{\mathbf{X/Y}} \cdots$. Par définition de tnF, le réseau représenté est tel que $F(\mathbf{A}) = \mathbf{B}$) et $F(\mathbf{A}.\mathbf{X}) = \mathbf{B}.\mathbf{Y}$. Toute autre fonction de transition doit être telle que les parcours $0_{\mathcal{S}} \xrightarrow{\mathbf{A/B}}{}^{*} s' \xrightarrow{\mathbf{X/Y}}{}^{*}$ existent. Il existe donc une transition à partir de $s' \xrightarrow{\mathbf{X'/Y'}}$ avec $\mathbf{X} \sqsubseteq \mathbf{X'}$ et $\mathbf{Y} \sqsubseteq \mathbf{Y'}$. Si s est un état tel que la granularité minimale en entrée ou en sortie de tnF était atteinte, alors l'autre fonction de transition a obligatoirement une granularité minimale en entrée/sortie inférieure.

Taux et coefficient de Lipschitz Pour un réseau ordonné dont la chaîne d'inverses est $(\mathbf{Y}_k^{*\mathbf{D}})_k$. En posant $z = k$ si le réseau est Moore et $k+1$ sinon, sa fonction de transition en forme normale déterministe est de granularité minimale (resp. maximale) :

$$\text{en entrée : } \min_{k>0,\mathbf{D}} \left|\mathbf{Y}_{\delta z}^{*\mathbf{D}}\right| \text{ (resp. } \sup_{k>0,\mathbf{D}} \left|\mathbf{Y}_{\delta z}^{*\mathbf{D}}\right|) \qquad \text{en sortie : } \min_{k>0,\mathbf{D}} \left|\mathbf{Y}_{\delta z}^{**\mathbf{D}}\right| \text{ (resp. } \sup_{k>0,\mathbf{D}} \left|\mathbf{Y}_{\delta z}^{**\mathbf{D}}\right|)$$

Une fois les inverses et bi-inverses du réseaux déterminés, même le taux de tnF se calcule exactement, simplement en effectuant la division de la production sur la consommation :

— taux instantané minimal (resp. maximal) à la transition de l'état k

$$\min_{\mathbf{D}} \frac{|\mathbf{Y}_{\delta z}^{**\mathbf{D}}|}{\max(1, |\mathbf{Y}_{\delta z}^{*\mathbf{D}}|)} \quad \text{resp.} \quad \max_{\mathbf{D}} \frac{|\mathbf{Y}_{\delta z}^{**\mathbf{D}}|}{\max(1, |\mathbf{Y}_{\delta z}^{*\mathbf{D}}|)}$$

— coefficient de Lipschitz, conforme à la définition 6.1.24 et bien plus précis que l'approximation faite en théorème 6.3.28 :

$$K = \sup_{k>0,\mathbf{D}} \frac{|\mathbf{Y}_{\delta z}^{**\mathbf{D}}|}{\max(1, |\mathbf{Y}_{\delta z}^{*\mathbf{D}}|)}$$

Remarquez que par définition, $|\mathbf{Y}_{\delta z}^{*\mathbf{D}}| \geq 1$, sauf pour le premier instant d'un réseau Moore. Dans ce cas le maximum que nous effectuons au dénominateur permet d'obtenir la dérivé en ε.

7.4.2 La fonction de Gustave et dépendance en D

Bien que notre forme normale décrive une fonction de transition déterministe, elle n'est pas « séquentielle ». Nous allons illustrer nos propos avec le réseau de Gustave qui est ordonné mais pas séquentiel.

Il est la transposition directe de la fonction de Gustave. Cette fonction de Gérard Berry (initialement appelée b [Ber75] puis h [Ber78]) est devenue au fil des années un classique. L'histoire de cette fonction se trouve dans la présentation de Gérard Huet [Hue11]. Elle est le prototype de la fonction qui est stable mais qui n'est pas séquentielle (au sens de Vuillemin [Vui74]). Dans le domaine de Scott, elle se définit comme la plus petite fonction (donnant $\perp$ dans tous les cas non définis) satisfaisant :

$$GG(\perp, \text{true}, \text{false}) = \text{true} \qquad GG(\text{true}, \text{false}, \perp) = \text{true} \qquad GG(\text{false}, \perp, \text{true}) = \text{true}$$

La séquentialité est l'idée qu'il est possible d'évaluer les arguments dans un ordre prédéfini pour effectuer le calcul de la fonction. GG n'est pas séquentielle car pour tout indice, choisir d'évaluer l'argument correspondant peut bloquer le calcul (s'il n'y a pas de valeur, représenté ici par $\perp$), alors qu'en lisant les autres entrées il aurait pu s'achever.

Définissons un réseau sans état effectuant à chaque instant la fonction de Gustave :

$$tF_{GG}((\varepsilon, \text{true}, \text{false}), s_0) = (\text{true}, s_0)$$
$$tF_{GG}((\text{true}, \text{false}, \varepsilon), s_0) = (\text{true}, s_0)$$
$$tF_{GG}((\text{false}, \varepsilon, \text{true}), s_0) = (\text{true}, s_0)$$

Cette fonction de transition est en forme normale déterministe, elle décrit donc un réseau ordonné. Ainsi, ce réseau a une datation principale pour toute entrée $\mathbf{D}$. Observons avec $\mathbf{D} = (\text{true.true}, \text{true.false}, \text{false.true})$, la sortie du réseau est true.true et nous avons $(\text{true})^{*\mathbf{D}} = (\varepsilon, \text{true}, \text{false})$ et $(\text{true.true})^{*\mathbf{D}} = (\text{true}, \text{true.false}, \text{false})$, puis le réseau bloque.

Nous n'avons pas besoin d'aller au-delà de la première réaction pour comprendre ce qu'il se passe pour la datation. Notons les ports des entrées a, b, c et celui de sortie y. Une datation principale pour $\mathbf{D_1} = (\varepsilon, \text{true}, \text{false})$ est telle que les entrées nécessaires et suffisantes sont avant la sortie : $T_{\mathbf{D_1}}(b)[1] < T_{\mathbf{D_1}}(y)[1] \leq T_{\mathbf{D_1}}(a)[1]$ et $T_{\mathbf{D_1}}(c)[1] < T_{\mathbf{D_1}}(y)[1] \leq T_{\mathbf{D_1}}(a)[1]$. Nous faisons identiquement pour les entrées $\mathbf{D_2} = (\text{true}, \text{false}, \varepsilon)$ et $\mathbf{D_3} = (\text{false}, \varepsilon, \text{true})$.

Comme attendu, nous nous retrouvons avec trois datations, $T_{\mathbf{D_1}}$, $T_{\mathbf{D_2}}$ et $T_{\mathbf{D_3}}$ qui ne s'accordent pas sur quels ports lire avant de donner la première sortie. La datation pour $\mathbf{D_1}$ ne lit pas sur a, celle pour $\mathbf{D_2}$ sur b et pour $\mathbf{D_3}$ sur c. Ces trois choix ne se font pas suivant ce qui est déjà lu puisque nous sommes avant la première lecture. Si nous cherchons une datation qui les met toutes d'accord, alors nous n'aurons aucune lecture avant la sortie de la première valeur, ce qui est impossible.

Pour définir quelles entrées doivent être lues, la datation nécessite la valeur de certaines des entrées en question. C'est pourquoi *la date d'une valeur peut dépendre de sa valeur*, ce que nous voyons avec le réseau de Gustave. La datation est utile pour l'utilisateur du réseau (l'appelant), or lui connait les valeurs, cela ne pose donc pas de problème.

Par contre, la génération de code pour le réseau est plus complexe. Le déterminisme de tF_{GG} permet d'utiliser une primitive comme la primitive système `select`, tout en restant déterministe. Pour écrire la fonction de transition `step_GG`, nous utilisons du pseudo-code avec trois primitives :

— l'abstraction de `select` est `await_ready`. Elle renvoie l'indice d'une file non vide parmi celles données en argument et bloque jusqu'à ce qu'il en existe une.
— `peek` lit sans la consommer la première valeur disponible d'une file (bloquante).
— `pop` supprime la première valeur disponible d'une file (bloquante).
— `push` qui met une valeur en queue d'une file sans bloquer.

Le code pour la fonction de transition du réseau de Gustave est donné en liste 7.1. Quelques remarques concernant ce code :

— La primitive `await_ready` doit au pire (comme dans notre cas) permettre d'attendre sur un nombre aussi important d'entrées que l'arité de la fonction de transition. Cette primitive est la clef permettant de générer du code. Si l'on accepte une attente active, une primitive `is_ready` prenant une unique file est suffisante, exactement comme nous avons fait avec les futures (cf. section 4.6.2). Dans tous les cas, il nous *faut* pouvoir vérifier la présence d'une valeur dans une file.
— La ligne 13 montre bien que notre code n'a pas besoin de toutes les entrées pour réagir. Si les deux premières files qui sont prêtes sont `a` et `b` et qu'elles contiennent respectivement `true` et `false`, la réaction peut se faire.

```
1  void step_GG(fifo a, b, c, y) {
2    switch (await_ready([a, b, c]))
3      case 1 :                                          \\a ready
4      if (peek(a))                                      \\(t, ., .)
5        switch(await_ready([b, c])) {
6        case 1 :                                         \\a, b ready
7          if (peek(b)) {                                 \\(t, t, .)
8            if (peek(c)) return;                          \\OK (t, t, t)
9            else {                                        \\OK (t, t, f)
10             pop(b); pop(c); push(y, true); return;
11           }
12         }
13         else{                                          \\OK (t, f, .)
14           pop(a); pop(b); push(y, true); return;
15         }
16       case 2 :                                         \\a, c ready
17         if(peek(c)){                                   \\(t, ., t)
18           if(peek(b)) return;                           \\OK (t, t, t)
19           else{                                         \\OK (t, f, t)
20             pop(a); pop(b); push(y, true); return;
21           }
22         }
23         else {                                         \\(t, ., f)
24           if (peek(b)) {                                \\OK (t, t, f)
25             pop(b); pop(c); push(y, true); return;
26           }
27           else {                                        \\OK (t, f, f)
28             pop(a); pop(b); push(y, true); return;
29           }
30         }
31       }
32     else {                                             \\(f, ., .)
33 ...
```

Listing 7.1 : Début du code de la fonction `step` du réseau Gustave

— À la ligne 8, nous n'utilisons pas `await_ready` avant d'essayer de lire la valeur de la troisième file car à partir de ce qui est su (que la file `a` et la file `b` ont en tête la valeur `true`), la fonction de transition ne peut pas décider sans connaître la valeur de la dernière file. Nous aurions pu écrire `await_ready(c); if (peek(c)) ...`, mais comme `peek` est bloquant cela n'est pas nécessaire.

— La primitive `peek` n'est pas nécessaire mais elle permet de laisser la gestion du stockage des entrées hors de la fonction de transition. En son absence, il serait nécessaire d'utiliser `pop` et si la valeur contenue n'est pas utile pour la réaction courante, il faudrait la stocker pour les prochaînes fois.
Cette dernière remarque est bien visible à la ligne 10 puisque nous n'appelons `pop` que sur deux files bien que la décision ait été prise en lisant une valeur des trois files.

— Toujours à la ligne 8, la fonction de transition finit, sans avoir enlevé de valeur de ses files d'entrées, ni avoir rajouté une valeur sur `y`. Cela correspond au fait que $\mathrm{exec}(tF_{GG})$

s'arrête car il n'y a plus de préfixe de l'entrée qui corresponde à une transition acceptable : quelles que soient les entrées supplémentaires qui seront données, le réseau ne fournira plus de sortie.
Ce comportement est une conséquence du fait que le réseau est défini partiellement. Il aurait été possible de changer l'exemple en rajoutant les deux transitions :

$$tF((\text{false}, \text{false}, \text{false}), k) = (\text{false}, k+1) \tag{7.4}$$
$$tF((\text{true}, \text{true}, \text{true}), k) = (\text{false}, k+1) \tag{7.5}$$

Ce qui le rend totalement défini sans pour autant changer le fait qu'il est ordonné mais pas séquentiel. Dans le code généré, la seule différente aurait été d'avoir les `pop` des trois entrées et un `push(y, false)` juste avant `return` aux lignes 8 et 18.
Pour autant le fait que le réseau ne soit pas totalement défini n'est pas étrange ou inhabituel. C'est monnaie courante en Signal et même en Heptagon. La sémantique synchrone de `merge` force un certain nombre de ses arguments à être égaux à *abs* (la valeur représentant l'absence). C'est à l'utilisateur du réseau de l'utiliser comme sa spécification le demande, éventuellement aidé par un système de type vérifiant que la datation du réseau est respectée.

7.4.2.1 Discussion

Bien qu'il existe une forme normale pour la fonction de transition et que le code généré soit déterministe, ce code repose sur des primitives de communication qui permettent beaucoup plus. En effet, avec `await_ready` il est possible d'écrire le code de la fonction de transition du *ou* parallèle, et donc l'intégralité des réseaux de Kahn. Notez que c'est exactement le même genre d'information que nous avons ajouté avec `is_ready` pour les futures, information qui nous a permis de programmer le ou-parallèle en section 4.6.2.

Ainsi, avec `await_ready` nous pourrions générer du code pour tous les réseaux de Kahn, par exemple en utilisant la forme canonique de la fonction de transition que nous donnions en section 6.3.3. Cependant, nous avons cherché à nous restreindre à ceux ayant une datation principale et nous avons trouvé des réseaux dont la fonction de transition utilise certes `await_ready`, mais est déterministe. Ce résultat est tout à fait cohérent. L'existence d'une datation principale est la possibilité de définir une typage temporel permettant à *l'utilisateur* du réseau de savoir quand sont requises les entrées et quand sont produites les sorties, or cet utilisateur connait les valeurs qu'il fournit.

La gestion d'une telle datation n'est pas triviale. Les travaux sur les automates d'interfaces[LX04] et la théorie des jeux pourraient permettre d'apporter des réponses que nous n'avons pas explorées. Des travaux récents concernant leurs utilisations dans le monde synchrone sont discutées [Tri+11]. Le langage Signal est relationnel, il est ainsi capable d'accepter un programme tel que `GG`. Pour le traiter, il va lui ajouter automatiquement des fils cachés, comme ce que nous allons présenter ci-après. Nous discutons en détail du cas de Signal en section 10.3.

De plus, nous parlons de code généré, alors que nous n'avons pas vraiment de langage source. Nous ne connaissons pas de primitives qui permette cette dépendance entre présence et valeur d'un flot, tout en étant moins puissant que `await_ready`. Moins puissant signifiant ne permettant pas de programmer le *ou* parallèle.

Finalement, nous considérons qu'un code comme celui de Gustave n'est pas utile en pratique. La spécification de l'interface de ce genre de réseau aura de fortes chances d'être aussi complexe que le réseau en lui-même alors que dans le même temps le code généré avec `await_ready`

est factoriellement grand (toutes les permutations d'arrivée des entrées doivent être prises en compte).

7.4.2.2 Fils cachés

Puisque le réseau Gustave a une datation, son appelant est en mesure de connaître quelles sont les entrées nécessaires. Dans un schéma de compilation proche de celui d'Heptagon (ou simplement comme l'utilisation initiale de la co-itération [CP98]), la gestion des entrées et des sorties n'est pas effectuée à l'aide de files. Les entrées sont passées comme des arguments de la fonction de transition. Ce point est très important pour avoir de bonne performances. De plus, le nombre d'arguments de la fonction de transition ne varie pas. La signature de la fonction de transition serait alors `bool step_GG(bool a, b, c)`.

Cependant, ces valeurs doivent être lues pour décider de la réaction comme nous venons de le voir. *Conséquemment, sans autre information, il n'est plus possible d'utiliser n'importe quelle valeur pour encoder l'absence*. Plusieurs options sont envisageables. Pour mieux les illustrer, nous définissons GGG, une version de Gustave non constante :

$$tF_{GGG}((\varepsilon, \text{true}, \text{false}), s_0) = (1, s_0)$$
$$tF_{GGG}((\text{true}, \text{false}, \varepsilon), s_0) = (2, s_0)$$
$$tF_{GGG}((\text{false}, \varepsilon, \text{true}), s_0) = (3, s_0)$$

— Nous pouvons décider d'ajouter une valeur qui code explicitement pour l'absence. Ce qui requiert que le producteur d'une des entrées, même s'il n'a pas de valeur réelle à fournir, doit être au courant de l'action des autres et fournir la valeur codant pour l'absence. Cette solution revient à synchroniser tout le monde.

— Une version sans valeur spéciale, mais avec des pointeaux est possible. Nous pouvons rajouter un fil portant une valeur booléenne pour chaque entrée servant à indiquer sa présence :

```
int step_GGG(bool present_a, a, present_b, b, present_c, c) {
  if (present_a) {
    if (present_b) {
      assert (!present_c && a && !b);
      return 2;
    } else {
      assert (present_c && !present_b && !a && c);
      return 3;
    }
  } else {
    assert (present_b && present_c && b && !c);
    return 1;
  }
}
```

Si l'appelant est correct vis-à-vis de la datation tous les `assert` que nous avons mis ne servent à rien.

— En considérant que l'appelant est correct vis-à-vis de la datation, rajouter un fil de présence pour chaque entrée est inutile, dans le cas de GGG un unique argument supplémentaire est nécessaire. Nous choisissons arbitrairement `present_a`, ce qui donne :

```
int step_GGG(bool present_a, a, b, c) {
  if (present_a) {
    if (a) return 2;
    else return 3;
  }
  else return 1;
}
```

Ce fil de présence supplémentaire est ce que nous appelons un « fil caché ». Il représente une information nécessaire si l'on souhaite se passer de l'utilisation de files de communications avec une primitive du type `await_ready`. Remarquez qu'il n'est pas utilisé pour réagir à l'absence d'une valeur, mais bien à la présence. Nous sommes dans les réseaux de Kahn, l'absence de valeur n'a pas de signification. Par contre il est utilisé pour définir la présence (l'existence) d'une valeur. L'ajout de ces fils est appelé au choix *endochronization* ou bien *Kahnification* dans le monde du langage Signal [Ben07]. La question de savoir combien de fils et lesquels sont nécessaires a toujours été au cœur des problématiques de compilation de Signal (cf. sections 10.3.2 et 10.3.2.2).

7.4.9 *Remarque* **La fonction de transition pour la fonction de Gustave**
Si l'appelant est correct vis-à-vis de la datation, le code de la fonction de transition du réseau Gustave, sans les `assert`, est déroutant :

```
bool step_GG(bool a, present_a, b, present_b, c, present_c) {
  return true;
}
```

Il pourrait alors se simplifier simplement en une fonction sans arguments.

7.4.3 Réseaux séquentiels

L'approche de Lustre et de ses descendants, les langages synchrones les plus proches des réseaux de Kahn, ne permettent pas une dépendance complexe entre les dates des valeurs et les valeurs. Cette dépendance est celle que nous avons décrite pour Heptagon (cf. section 2.3.2.1) c'est-à-dire un arbre de dépendances. Nous étudions dans cette section le lien que cela a avec la possibilité de générer du code déterministe *séquentiel*, y compris la lecture des entrées.

7.4.3.1 Séquentialité

La séquentialité d'une fonction continue remonte à 1974 avec la définition de J. Vuillemin. Il la décrit ainsi :

> Intuitively, g is sequential if, at any given moment, the value of (at least) one of its arguments is crucially needed in order to increase the value of the result.

Nous donnons sa définition avec nos notations :

7.4.10 *Définition* **V-séquentialité [Vui74]**
Un réseau F est V-séquentiel ssi à tout moment, il y a une des entrées qui doit croître pour que le résultat croisse :

$$\forall \mathbf{X}, \exists i, \forall \mathbf{Y}, \quad \mathbf{X} \sqsubseteq \mathbf{Y} \wedge \mathbf{X}^{(i)} = \mathbf{Y}^{(i)} \quad \Longrightarrow \quad F(\mathbf{X}) = F(\mathbf{Y})$$

L'indice i est appelé indice critique pour $\mathbf{X}$.

7.4.11 *Propriété* **Composition et point fixe [Vui74]**
Comme pour les fonctions ordonnées, la V-séquentialité d'une fonction est préservée par composition et point fixe, mais ne l'est pas par produit.

7.4.12 *Propriété* **Chaine d'entrées critiques**
Une Fonction V-séquentielle admet, pour tout flot d'entrées $\mathbf{D}$, au moins une chaîne $(\mathbf{X}_k)_k$ dite critique, telle que chaque élément est égal au précédent plus une valeur à l'indice critique. Nous prenons i_k un indice critique de la fonction avec $\mathbf{X}_k$ en entrée et définissons la chaîne récursivement :

$$\mathbf{X}_1 = \varepsilon \qquad \forall k > 1, \quad \mathbf{X}_K^{(j)} = \begin{cases} \mathbf{X}_{k-1}^{(j)}.d & \text{si } j = i_k \text{ avec } d \text{ scalaire, tel que } \mathbf{X}_{k-1}^{(j)}.d \sqsubseteq \mathbf{D}^{(j)} \\ \mathbf{X}_{k-1}^{(j)} & \text{sinon} \end{cases}$$

En ajoutant uniquement un scalaire sur une des composantes, $\mathbf{X}_k$ *couvre* $\mathbf{X}_{k-1}$ (cf. [KP78] et partie IV), c'est-à-dire qu'il n'y a pas d'intermédiaire : $\mathbf{X}_{k-1} \sqsubseteq \mathbf{Z} \sqsubseteq \mathbf{X}_k \implies (\mathbf{X}_{k-1} = \mathbf{Z} \wedge \mathbf{X}_k = \mathbf{Z})$.

7.4.13 *Propriété* **Une fonction V-séquentielle est ordonnée**
Un réseau représenté par la fonction F V-séquentielle est ordonné. La réciproque est fausse, le réseau Gustave de la section 7.4.2 en est un exemple.

Démonstration Si la fonction est V-séquentielle, elle est stable (cf. [Ber78]). Ensuite nous devons prouver que les inverses sont ordonnés. Pour toute entrée $\mathbf{D}$, nous prenons une chaîne d'entrées critiques $(\mathbf{X_k})_k$. Prouvons que cette chaîne contient tous les inverses.

Pour tout inverse fini $\mathbf{Y}^{*\mathbf{D}}$, soit $\mathbf{Y}^{*\mathbf{D}} = \varepsilon$ et alors il est égal à $\mathbf{X}_1$, soit il est strictement plus grand et nous pouvons prendre le k maximum tel que $\mathbf{X}_k \sqsubset \mathbf{Y}^{*\mathbf{D}}$. Il existe car $\mathbf{Y}^{*\mathbf{D}}$ est fini. $\mathbf{Y}^{*\mathbf{D}}$ est un inverse, il génère une sortie plus grande que $\mathbf{X}_k$. Pour cela, la V-séquentialité impose qu'il contienne des valeurs aux indices critiques, donc au moins $\mathbf{X}_{k+1}$, soit $\mathbf{Y}^{*\mathbf{D}} = \mathbf{X}_{k+1}$

La V-séquentialité n'est pas directement transposable à la fonction de transition. En effet, la V-séquentialité tire sa force du fait que le domaine de définition d'une fonction continue est un ordre partiel *complet*. Ainsi un réseau F V-séquentiel produit quelle que soit son entrée au moins $F(\varepsilon)$ et comme ε est inférieur à tout argument, il y a un indice critique commun à *toutes* les entrées possibles.

Le domaine de définition des fonctions de transition n'est pas forcément un ordre partiel *complet* et ne contient pas forcément ε. Toutefois, une entrée qui n'est pas dans le domaine de la fonction de transition, correspond à une entrée qui ne « produit » rien puisque $\mathrm{exec}(tF)$ bloquera.

Nous nous inspirons d'une autre définition de la séquentialité, donnée par R. Milner qui coincide avec la définition de J. Vuillemin dans les domaines de Scott plats (cf. [Ber78]) :

7.4.14 *Définition* **M-séquentialité [Mil77]**
Une fonction f est M-séquentielle (dans le domaine de Scott) si elle est constante ou bien s'il existe un indice i tel que si $x^{(i)} = \bot$ alors f ne produit pas quelles que soient les autres entrées ($f(\ldots, x^{(i)}, \ldots) = \bot$) et la fonction des autres arguments en fixant $x^{(i)}$ est aussi M-séquentielle.

Cette définition s'adapte directement aux formes canoniques des fonctions de transition puisque leur domaine de définition ne comprend que des flots de taille inférieure à 1, domaine

qui permet une bijection entre le domaine de Scott et le domaine de Kahn, en remplaçant $\perp$ par ε.

Nous souhaitons toutefois définir la séquentialité de n'importe quelle fonction de transition, en gardant l'essence de la M-séquentialité : une fonction qui, quelles que soient les entrées qu'elle a déjà lues, est capable de décider quelle nouvelle entrée lire.

7.4.15 *Définition* **KM-séquentialité : M-séquentialité dans Kahn**
Une fonction f d'arité d est KM-séquentielle ssi son domaine de définition est réduit à ε ou bien s'il existe un indice $1 \leq i \leq d$ tel que pour tout $\mathbf{X}$ dans le domaine de F, la $i$ème composante est strictement différente de ε, donc $\mathbf{X}^{(i)} = x.\mathbf{r}$ et la fonction suivante est KM-séquentielle :

$$\mathbf{X} \mapsto f(\mathbf{X}^{(1)}, \cdots, x.\mathbf{X}^{(i)}, \cdots, \mathbf{X}^{(d)})$$

7.4.16 *Exemple* Une fonction de transition avec, pour tout état, le domaine $\{(0.1, \varepsilon), (0, 1)\}$ n'est pas séquentielle : le premier indice critique est 1 puisque les deux paires ont une valeur en cette composante. Nous devons ensuite regarder la fonction une fois la valeur 0 lue à l'indice critique. Le domaine de la fonction restante est $\{(1, \varepsilon), (\varepsilon, 1)\}$ qui n'a pas d'indice critique.

Tandis que si le domaine avait été $\{(1.1, \varepsilon), (0, 1)\}$, le premier indice critique est 1 puisque les deux paires ont une valeur en cette composante, ensuite en fixant cette première valeur à 0, le domaine de la fonction restante est $\{(\varepsilon, 1)\}$ qui a pour indice critique 2, puis qui est constante. En fixant la valeur du premier indice critique à 1 au lieu de 0, le domaine de la fonction restante est $\{(1, \varepsilon)\}$ qui a pour indice critique 1 puis qui est constante. Cette fonction est donc bien séquentielle.

7.4.17 *Propriété* **La KM-séquentialité de tF rend le réseau ordonné**

Démonstration La KM-séquentialité de tF implique que son domaine soit clos par intersection compatible donc que le réseau qu'elle décrit est ordonné (par la propriété 7.3.17).

La réciproque n'est pas vraie comme nous pouvons le voir dans l'exemple suivant :

7.4.18 *Exemple* Le réseau F_{seqno} qui renvoie true à chaque fois qu'il a reçu une entrée sur chacune de ses entrées est ordonné (l'ensemble des inverses $\mathrm{Inv}(F_{seqno}, \mathbf{D}) = \{(\mathbf{x_1}, \mathbf{x_2}) \mid (\mathbf{x_1}, \mathbf{x_2}) \sqsubseteq \mathbf{D} \wedge |\mathbf{x_1}| = |\mathbf{x_2}|\}$ est bien totalement ordonné). Pourtant nous pouvons décrire ce réseau avec une fonction de transition qui n'est pas KM-séquentielle :

$$tF_{seqno}(s_0, (x_1, \varepsilon)) = (\varepsilon, \top_1) \qquad tF_{seqno}(s_0, (\varepsilon, x_2)) = (\varepsilon, \top_2)$$
$$tF_{seqno}(\top_1, (\varepsilon, x_2)) = (\mathrm{true}, s_0) \qquad tF_{seqno}(\top_2, (x_1, \varepsilon)) = (\mathrm{true}, s_0)$$

Cependant, ce réseau est V-séquentiel puisqu'il n'a que deux entrées (cf. propriété 7.4.20). La subtilité tient à l'existence de deux indices critiques à l'état s_0, et au lieu de les lire tous les deux, la fonction de transition expose le choix de lire l'un avant l'autre. Cependant, le choix pris est indistinguable puisque ne produisant pas de sortie avant que les deux indices aient été comblés.

La condition de productivité de la forme normale empêche cette situation et la rend KM-séquentielle :

$$tnF_{seqno}(s_0, (x, y)) = (\mathrm{true}, s_0)$$

7.4.19 Théorème **Les réseaux V-séquentiels**
Tout réseau de Kahn V-séquentiel admet une fonction de transition en forme normale déterministe KM-séquentielle. Inversement, tout réseau de Kahn est V-séquentiel s'il s'exprime avec une fonction de transition KM-séquentielle.

Démonstration Nous avons vu (cf. propriété 7.4.13) qu'un réseau V-séquentiel est ordonné. Il admet donc une fonction de transition en forme normale tnF.

Par productivité de tnF, à partir de tout état, les transitions produisent une valeur différente de ε. Or par V-sequentialité, pour produire plus, toute transition doit consommer une valeur à l'indice critique. Si cette valeur n'est pas suffisante pour une transition, le raisonnement est le même une fois cette valeur fixée.

Réciproquement, une fonction de transition KM-séquentielle implique que le réseau soit V-séquentiel. En effet, pour tout $\mathbf{X}$,

— soit $\mathrm{Inv}(F, \mathbf{X})$ est de cardinal fini égal à n. Ainsi $0 \xrightarrow{\mathbf{A/B}}{}^{*} s$ est une suite terminale avec $\mathbf{X} = \mathbf{A}.\mathbf{X}_\delta$. Deux sous-cas sont possibles, soit $\mathbf{X}_\delta$ est incompatible avec le domaine de tnF, auquel cas quelles que soit les entrées rajoutées, le réseau ne produira jamais plus. Deuxième cas, la fonction $\lambda\mathbf{R}.tnF(s, \mathbf{X}_\delta.\mathbf{R})$ est KM-sequentielle, ainsi pour produire plus, le réseau devra fournir une entrée à l'indice critique de cette fonction.

— soit $\mathrm{Inv}(F, \mathbf{X})$ est de cardinal infini, alors pour toute entrée $\mathbf{D}$ incluant $\mathbf{X}$, nous avons pour tout k $\mathbf{Y}_k^{*\mathbf{X}} = \mathbf{Y}_k^{*\mathbf{D}}$ (cf. propriété 7.3.8), ainsi $F(\mathbf{X}) = \lim_k F(\mathbf{Y}_k^{*\mathbf{X}}) = F(\mathbf{D})$ (cf. propriété 7.3.7).

7.4.20 *Propriété*
Une fonction ordonnée est V-séquentielle si elle a au plus deux entrées.

Démonstration La V-séquentialité equivaut à tnF KM-séquentiel. Or le domaine de tnF est composé d'éléments incompatibles. Mais incompatibles pour des paires de flots est très restrictif. Inductivement, avec la propriété ils sont incompatibles, mais avec un préfixe commun. Au début, ils sont incompatibles et ont ε comme préfixe commun. Soit il n'y a qu'une paire alors. Soit il y en a plusieurs, incompatibles, avec un préfixe commun : par l'absurde, s'il n'y a pas d'indice où ils ont tous une valeur en plus du préfixe, cela signifie qu'il y en a une qui n'a pas de valeur en plus sur sa première composante et qu'il y en a une autre qui n'en a pas sur sa deuxième composante, mais alors ces deux éléments ne sont pas incompatibles. Ils ont donc tous une valeur sur une des composantes. Nous la fixons et la rajoutons au préfixe pour relancer l'induction. L'induction finit car le domaine de tF est formé de flots de tailles finies.

7.4.21 *Remarque* Cette dernière propriété est aussi vraie pour la stabilité, ce qui explique pourquoi la fonction Gustave est un des exemples les plus petits de fonction ordonnée non séquentielle.

7.4.3.2 Datation séquentielle

La causalité d'une datation n'est pas une propriété simple à exprimer formellement, d'autant plus que nous n'avons pas spécifié comment cette datation est calculée et encore moins le formalisme utilisé pour la décrire.

Il est toutefois possible de considérer des datations qui ne peuvent pas ne pas être causales. C'est le cas des datations que nous appellons séquentielles. Une datation séquentielle est telle qu'à tout moment nous connaissons le prochain port d'entrée lu :

7.4.22 *Définition* **Datation séquentielle**
Pour tout flot d'entrées $\mathbf{X} = (\mathbf{X}^{(1)}, \ldots, \mathbf{X}^{(d)})$ respectivement de taille $n_1, \ldots, n_d$ et donné aux ports d'entrées $e^{(1)}, \ldots, e^{(d)}$, il existe un indice critique i tel que si le réseau est capable de consommer, alors il consommera en premier sur le port i :

$$\forall \mathbf{X}, \exists i, \forall j \neq i, \mathbf{X} \sqsubset \mathbf{D}, \quad T_{\mathbf{D}}(e^{(j)})[n_j + 1] < \infty \implies T_{\mathbf{D}}(e^{(i)})[n_i + 1] < T_{\mathbf{D}}(e^{(j)})[n_j + 1]$$

7.4.23 *Propriété* **Réseau séquentiel et datation séquentielle**
Tout réseau V-séquentiel admet une datation principale séquentielle. Réciproquement, toute datation principale séquentielle d'un réseau l'oblige à être séquentiel.

Démonstration La réciproque est le sens le plus délicat. Si le réseau admet une datation principale séquentielle, il est lui-même séquentiel. Par l'absurde, s'il ne l'est pas alors $\exists \mathbf{X}, \forall i, \exists \mathbf{D_i}, \mathbf{X} \sqsubset \mathbf{D_i}, \mathbf{X}^{(i)} = \mathbf{D_i}^{(i)} \wedge F(\mathbf{X}) \sqsubset F(\mathbf{D_i})$. Nous ajoutons par continuité du réseau que $\mathbf{D_i}$ est fini. La datation séquentielle affirme : $\exists j, \forall k \neq j, \mathbf{X} \sqsubset \mathbf{D}, T_{\mathbf{D}}(e^k)[n_k + 1] < \infty \implies T_{\mathbf{D}}(e^j)[n_j + 1] < T_{\mathbf{D}}(e^k)[n_k + 1]$. Nous nous intéressons donc à $\mathbf{D_j}$ qui est tel que $\mathbf{X}^{(j)} = \mathbf{D_j}^{(j)}$, ainsi tout flot $\mathbf{A}$ inclus dans $\mathbf{D_j}$ est tel que $\mathbf{A}^{(j)} \sqsubseteq \mathbf{X}^{(j)}$.

Ce qui est donc le cas de $(\mathbf{F}(\mathbf{D_j}))^{*\mathbf{D_j}}$ que nous notons $\mathbf{Z}$. Cependant, $\mathbf{Z}$ produit strictement plus que $\mathbf{X}$ donc il existe k tel que $\mathbf{X}^{(k)} \sqsubset \mathbf{Z}^{(k)}$. Par ailleurs, la datation est correcte donc $T_{\mathbf{D_j}}(\mathbf{Z})$ est finie puisque $\mathbf{D_j}$ est fini, donc produit une sortie finie. Ainsi, $T_{\mathbf{D_j}}(e^k)[n_k + 1] \leq T_{\mathbf{D_j}}(\mathbf{Z}) < T_{\mathbf{D_j}}(F(\mathbf{D_j})) < \infty$ donc $T_{\mathbf{D_j}}(e^j)[n_j + 1] < T_{\mathbf{D_j}}(F(\mathbf{D_j}))$. Ceci est finalement absurde car par représentativité cela signifierait que $(\mathbf{F}(\mathbf{D_j}))^{*\mathbf{D_j}}$ a une $n_j + 1$ième valeur sur l'entrée e^j, ce qui n'est pas le cas.

L'autre sens. La datation principale séquentielle est construite en fournissant T_{early} avec une chaîne $(\mathbf{X}_k)_k$ d'entrées critiques. La spécificité de cette chaîne est qu'elle ne met qu'un scalaire de plus pour chaque élément, ainsi $\mathbf{X}_{\delta k}$ est atomique (contient un et un seul scalaire) : la datation résultante date une seule valeur en entrée par instant impliquant que $T(\mathbf{X}_{k-1}) < T(\mathbf{X}_k)$. La séquentialité est donc assurée. La principalité découle immédiatement du fait que la chaîne des entrées critiques est une chaîne et contient tous les inverses du réseau (cf. la preuve de propriété 7.4.13).

La datation séquentielle en souhaitant tout refléter à l'aide d'un unique ordre total, décrit de manière très/trop fine le comportement du réseau et requiert quelquefois des choix qui n'ont pas d'importance. Regardons un exemple Heptagon :

7.4.24 *Exemple* **Arbre d'horloge d'Heptagon**

```
node tree(bool c; int a; int b) = (y : int)
let
  y = merge c (a+b) 0
tel
```

Ce programme est V-séquentiel car une de ses fonctions de transition (celle induite par la sémantique de Kahn présentée en section 2.1) est KM-séquentielle :

$$tnF_{tree}(s_0, (\text{false}, \varepsilon, \varepsilon)) = (0, s_0) \qquad tnF_{tree}(s_0, (\text{true}, a, b)) = (a + b, s_0)$$

Cette fonction de transition a 1 pour premier indice critique. Ensuite, si la valeur lue est false la fonction restante est constante, tandis que si la valeur lue est true les deux indices critiques

restants sont 2 et 3 *pris dans n'importe quel ordre* avant que la fonction restante ne devienne constante.

La datation représentative de la forme normale est donnée par :

$$T_{\text{repr}}(c)[k] = k + 1/3 \qquad T_{\text{repr}}(y)[k] = k + 2/3$$

$$T_{\text{repr}}(a)[k] = T_{\text{repr}}(b)[k] = k' + 1/3 \qquad \text{avec} \qquad k = \sum_{j=1}^{k'} (c[j] \ ? \ 1 \ : \ 0)$$

Pour la rendre causale, il est nécessaire de déplacer la consommation de a et b postérieurement à c, dont la valeur décide de leurs lectures. Cependant pour être séquentielle, elle devra en outre ordonner la lecture de a relativement à celle de b, par exemple avec :

$$T(c)[k] = k + 1/4 T(b)[k] = 1/8 + T(a)[k] = k' + 1/2$$

Si nous revenons à Heptagon, la signature d'horloge donnée par le compilateur est plus claire `tree(c :: .; a :: . on true(c); b :: . on true(c)) = (y :: .)`. Cette signature se lit : le flot `c` est nécessaire à tous les instants, le flot `a` ne l'est que quand `c` vaut true, idem pour `b`, tandis que `y` est présent à tous les instants. Ainsi, elle ne sur-contraint pas l'arrivée de `a` relativement à `b`.

En Heptagon, comme nous venons de le voir, l'utilisation des instants, associés à une règle, ici syntaxique, pour ordonner (partiellement) les lectures et les écritures de l'instant, soulage grandement la compréhension. Nous le verrons dans le chapitre suivant, cette simplification est féconde.

7.5 Conclusion

Nous avons dès le début souhaité traiter la datation comme une boîte noire, donnant un ordre total entre les occurrences de chacune des valeurs quelles que soient les entrées ou les sorties. Nous avons cherché à ce que la datation soit un pointage principal, c'est-à-dire décidant des dates des sorties *et* des entrées, indépendamment du contexte. Tous les réseaux n'admettent pas une telle datation sans perdre des comportements possibles. L'idée est d'être capable de dire « une valeur n'est consommée avant la production d'une sortie *que* si elle est nécessaire.

Pour comprendre quels sont les critères nécessaires à l'existence d'une datation principale, nous avons réexploré les réseaux de Kahn à la recherche de l'expression des dépendances de données. Dans le cas général, celles-ci n'existent pas, comme le montre le réseau calculant le *ou*-parallèle. L'existence de dépendances de données bien définies équivaut à la stabilité de la sémantique du réseau, notion à créditer à l'un des fondateurs des langages synchrones, G. Berry, bien que ceci soit antérieur à ses aventures dans le synchrone. Peu de travaux se penchent sur la stabilité dans les réseaux de Kahn. La stabilité exclut le *ou*-parallèle, mais n'exclut pas le réseau de Gonthier. Ce réseau est stable, mais l'ordre de consommation des entrées n'est pas unique et il n'est pas possible de choisir sans connaître l'ordre d'utilisation des sorties. La recherche d'une datation principale et donc d'un ordre total entre les entrées et les sorties nous a poussé à définir les réseaux ordonnés. Pour ceux-ci, l'ordre entre les consommations et les productions est entièrement déterminé par les valeurs des entrées. Un réseau est ordonné ssi il a une datation principale.

Connaissant l'ordre entre les entrées et les sorties, il est possible de « compiler » un réseau ordonné vers une fonction de transition. Nous donnons une forme normale tnF associée à la datation principale T_{repr}. Cette forme normale maximise la granularité de communication.

Avant de pouvoir parler d'une véritable génération de code, nous devons exclure les réseaux non réactifs. En effet, ceux-ci nécessitent une fonction de transition capable de produire une infinité de valeurs en une transition.

Une fois restreints aux réseaux ordonnés réactifs, nous sommes en mesure de produire une fonction de transition (bien qu'avec une mémoire infinie, mais là n'est pas encore la question). Bien que déterministe, cette fonction de transition requiert, en général, une primitive permettant d'attendre et de lire des valeurs sur plusieurs entrées en parallèle. Cette primitive s'apparente à la primitive système `select`. Nous illustrons ce besoin avec le réseau de Gustave, illustrant une fonction stable (de plus ordonnée) mais non séquentielle.

Pour finalement arriver à une fonction de transition dont le code peut être un simple programme séquentiel, il faut en plus se restreindre aux réseaux séquentiels. Dans ce cas, la datation principale ordonne totalement les entrées entres elles, au lieu de se borner à ordonner les entrées par rapport aux sorties : à tout moment, on est en mesure de connaître le prochain port qui doit être lu.

7.5.1 Travaux connexes et parallèles

Comme nous pouvions nous y attendre, un réseau séquentiel correspond à un processus du langage défini par Kahn et MacQueen [KM76]. Essentiellement, ce langage permet de définir des processus séquentiels communicant au travers de files infinies avec la primitive `get` *bloquante* et la primitive `put`.

Les travaux de Berry sur les DI-domaines que nous avons cités pour parler de la stabilité avaient l'idée originale d'associer deux fonctions stables ensemble. L'une définissant ce qui est calculé et l'autre la stratégie d'évaluation. Le parallèle entre la datation et cette fonction de stratégie d'évaluation demanderait certainement une analyse plus approfondie. Cependant, notre approche consistait à considérer les réseaux pour lesquels il existe une *unique* meilleure stratégie d'évaluation (les réseaux ordonnés). Ainsi dans notre cas, la spécifier n'est pas nécessaire puisqu'elle est intrinsèquement définie par le calcul.

Dans la veine de Kahn, MacQueen et Poltkin [KP78] introduisant les structures de données concrètes, de très nombreux travaux ont essayé de définir des notions de séquentialité/causalité des calculs, dans des cadres plus ou moins généraux. Notons la définition du langage CDS par Berry et Curien [BC83], mais aussi les structures d'événements de Winskel [Win86], avec dans les deux cas, une notion centrale de condition d'activation.

En plus de tous les travaux qui nous ont inspirés, nous devons indiquer que nos résultats peuvent en grande partie être vus comme un sous-cas des résultats sur les système de réécriture orthogonaux de Lévy et Huet [HL91a ; HL91b]. En effet, les réseaux de Kahn font partie de ces systèmes. L'on y retrouve en particulier, l'existence d'une forme normale pour les réseaux stables ainsi que la distinction entre les réseaux qui ont besoin d'une évaluation parallèle de leurs entrées et les réseaux séquentiels (fortement séquentiels dans leur vocable). Les liens entre ces travaux sont brièvement expliqués dans les transparents de Lévy [Lev12]. Cependant, le contexte des systèmes de réécriture est strictement plus riche que les réseaux de Kahn. L'existence d'une datation principale correspond à la possibilité d'avoir un « calcul par nécessité », calcul qui soulève de nombreuses questions dans les systèmes de réécriture comme l'atteste [Dur05].

Finalement, nous pouvons noter que les réseaux non déterministes avec une primitive `merge` non déterministe ont généré de très nombreux travaux pour tenter de leur donner sémantiques et modèles. La stabilité et la séquentialité y sont définies de manière plus générique, mais jouent toujours un très grand rôle comme l'atteste [HPW04]. Pour notre part, nous nous contenterons des réseaux de Kahn déterministes.

Pour conclure partiellement notre escapade dans la sémantique dénotationnelle, argumentons que, bien que tout ce que nous avons présenté a de toute évidence été dans la tête des pères des langages synchrones, nous considérons qu'exprimer ces notions, de manière ad-hoc pour les réseaux de Kahn est intéressant en soi. De plus, notre sentiment est que beaucoup des belles théories développées dans ce domaine sont mal transmises et donc en partie oubliées. Nous-mêmes ne prétendons pas en avoir fait le tour, loin de là.

7.5.2 S'il n'y avait qu'une seule chose à retenir

Dans la littérature, d'autant plus que les travaux sont récents, nous pouvons lire des choses comme « Dans un réseau de Kahn, il est impossible de choisir de manière non-déterministe quelle entrée lire. » ou bien « Les lectures sont bloquantes dans les réseaux de Kahn sinon le réseau ne serait pas déterministe. ». Ces affirmations sont au mieux des raccourcis, mais correspondent plus souvent à des préjugés inexacts.

7.5.1 *Nota bene* La seule propriété demandée à la fonction d'un réseau de Kahn et à celles de ses nœuds est d'être *continue*. Avec le réseau sous la forme d'un système de transition, la propriété correspondante est la *confluence* et non pas la séquentialité des lectures qui restreint aux réseaux séquentiels.

Chapitre 8

Cadre général des pointages et signatures

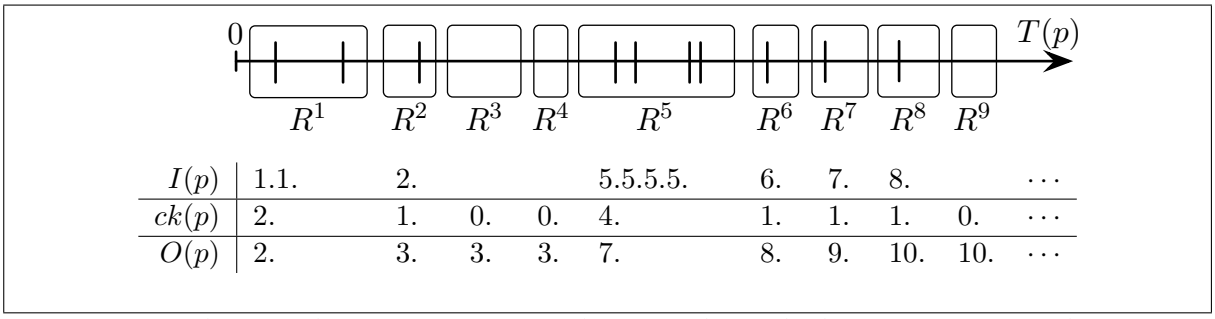
Dans ce chapitre, nous donnons un cadre général à l'utilisation des instants d'un rKPN. Leur utilité vient principalement de l'hypothèse synchrone qui permet de considérer chaque instant comme un point sans durée. Grâce à ceci, les dates s'abstraient avec un simple numéro d'instant et nous retrouvons la notion centrale d'*horloge d'un flot*, comme il y a en Lustre, Heptagon, Lucy-*n*, Lucid Synchrone, Signal, etc. L'horloge d'un flot abstrait la datation pour n'en retenir que le nombre de valeurs appartenant à chaque instant.

La finitude des horloges équivaut à la réactivité de la datation, condition que nous posons à partir de maintenant comme hypothèse. Sous cette hypothèse, les datations n'ont pas de point d'accumulation et il est possible de choisir les instants pour qu'ils contiennent une et une seule date. Alors, les horloges et les dates sont isomorphes. L'attrait des horloges au lieu des datations vient de la modélisation. Manipuler des entiers est plus aisé. Mais surtout, l'hypothèse synchrone rend presque systématique la concomitance de plusieurs flots alors qu'il y a des dépendances entre les valeurs de ces flot. Les horloges, contrairement aux datations, décrivent un ordre partiel qui n'intègre pas toutes les contraintes de dépendance.

Les langages synchrones font appel à des informations complémentaires pour contraindre l'ordre des valeurs d'un même instant. L'association d'horloges et des informations complémentaires est ce que nous appelons le pointage. Nous étudions de manière détaillée le pointage en boucle simple, qui demande que dans un instant, les dates des entrées soient avant les dates des sorties. Essentiellement, cette contrainte permet de lier un pour un les instants et les appels à une fonction de transition. Ce pointage n'est plus seulement une description globale du réseau, il exprime en plus le comportement d'une fonction de transition du réseau. C'est le pointage des nœuds d'Heptagon et sa représentation est une signature d'horloge.

Le pointage est relatif aux choix des instants, tout comme la datation était relative à l'échelle du temps. Ce choix des instants correspond à la notion d'horloge d'activation. Elle est une des clefs de voûtes de la compilation séparée des langages synchrones. Nous l'avons abondamment utilisée pour la compilation d'Heptagon et particulièrement pour la sémantique implicite (cf. section 2.4).

Bien que nous retrouvons des choses connues, nous les retrouvons dans un cadre plus général où en particulier les horloges prennent des valeurs entières et non booléennes. Ainsi, durant chacun de ses instants, l'horloge d'activation a le choix d'activer plusieurs fois une fonction de transition quand son équivalent booléen ne pouvait que choisir d'activer ou pas. Nous montrons comment ceci permet d'utiliser pleinement la notion d'horloge d'activation



$I(p)$	1.1.	2.			5.5.5.5.	6.	7.	8.		$\cdots$
$ck(p)$	2.	1.	0.	0.	4.	1.	1.	1.	0.	$\cdots$
$O(p)$	2.	3.	3.	3.	7.	8.	9.	10.	10.	$\cdots$

Figure 8.1: La datation d'un port p sur un axe horizontal, avec les boîtes représentant les instants choisis. Dessous, les différentes représentations du pointage de cette datation. Il y a autant de valuations pour $I(p)$ qu'il y a de valeurs sur p, tandis qu'il y a une valuation par instant pour O et ck. Le chronogramme représente les mots de chacune des représentations du pointage avec les valeurs alignées par instant.

pour ralentir, mais aussi accélérer une signature.

8.0.2 *Nota bene* Dans tout ce chapitre, nous ne considérons que des datations réactives, dont la définition a été donnée en propriété 6.1.12.

8.1 Abstraction synchrone des datations : le pointage

8.1.1 Horloge, indexation, compteur d'occurrence

En définissant la datation des entrées sorties d'un rKPN, nous avons insisté sur l'existence d'instants. Ces instants sont des segments de $\overline{\mathbb{R}}_+$ non vides, disjoints et indexés dans l'ordre : $R^k < R^{k+1}$. L'abstraction associée à l'hypothèse synchrone est de considérer chaque instant comme un point discret : son indice. Par réactivité d'une datation T (cf. propriété 6.1.12), les instants R contiennent toutes les dates en question. Chaque valeur entrante ou sortante est associée au numéro de l'instant contenant sa date.

Cette abstraction est notée $\alpha(T,R)$. Elle peut être représentée de manière tri-univoque par, une fonction d'*indexation* I conceptuellement équivalente à la datation, une fonction d'*horloges* ck qui est celle historiquement utilisée dans les langages synchrones et une fonction d'*occurrence* O qui est souvent la plus commode mathématiquement et utilisée dans les systèmes d'événements. La figure 8.1 donne un exemple de ces représentations.

8.1.1 *Définition* **Fonction d'indexation (I)**
Pour toute entrée $\mathbf{D}$, $I_{\mathbf{D}}(p)[i]$ donne l'indice k de l'instant contenant la $i$ème valeur passant par le port p. S'il n'existe pas, l'indexation est complétée comme la datation par ∞ :

$$I_{\mathbf{D}}(p)[i] = k \iff T_{\mathbf{D}}(p)[i] \in R^k \qquad I_{\mathbf{D}}(p)[i] = \infty \iff T_{\mathbf{D}}(p)[i] = \infty$$

Traditionnellement, les langages synchrones utilisent des horloges booléennes comme ce que nous avons présenté pour Heptagon. L'horloge booléenne d'un flot est une fonction qui indique pour chaque instant, avec true la présence d'une valeur dans le flot et avec false l'absence de valeur dans le flot. Si l'on note $ck_{bool\mathbf{D}}(p)$ l'horloge booléenne de p est :

$$ck_{bool\mathbf{D}}(p)[k] = \mathrm{true} \iff \exists i, T_{\mathbf{D}}(p)[i] \in R^k$$

Le mot booléen ainsi défini ne représente pas toujours fidèlement le flot qu'il pointe : quand il y a plusieurs valeurs dans un même instant, l'horloge booléenne perd cette information. Nous définissons l'horloge dans un cadre plus général permettant d'avoir plusieurs valeurs par instant.

8.1.2 *Définition* **Horloge (ck)**
Une horloge indique pour chaque instant le nombre de valeurs du flot qu'elle pointe :

$$ck_{\mathbf{D}}(p)[k] = \#\{i \mid T_{\mathbf{D}}(p)[i] \in R^k\}$$

Si la datation est réactive, l'horloge d'un flot $ck_{\mathbf{D}}(p)$ est un mot infini d'entiers. Si la datation est uniformément réactive, ces entiers sont uniformément bornés (cf. propriétés 6.1.12 et 6.1.14).

Déjà dans sa thèse, N. Halbwachs [Hal84] a eu recours à une datation des valeurs d'un réseau flot de données. Bien que sa datation soit plus générale que la nôtre, et qu'il n'utilise pas explicitement l'abstraction synchrone, il définit les compteurs d'occurrence qu'il note μ^+. Ce sont principalement ces compteurs qui lui permettent d'avoir un cadre mathématique puissant.

8.1.3 *Définition* **Compteur d'occurrence (O) aussi dit fonction d'accumulation**
Le compteur d'occurrence donne le nombre des valeurs passées au port p, depuis le premier instant, jusqu'à l'instant k compris :

$$O_{\mathbf{D}}(p)[k] = \#\{i \mid T_{\mathbf{D}}(p)[i] < \sup R^k\}$$

À entrées $\mathbf{D}$ fixées, le compteur d'occurrence d'un flot $O_{\mathbf{D}}(p)$ se représente avec un mot infini d'entiers croissants.

Les liens entres ces trois représentations sont amplement étudiés dans la thèse de F. Plateau [Pla10]. Pour faciliter la manipulation des horloges et le passage d'une représentation à l'autre, elle définit deux fonctions que voici :

8.1.4 *Définition* **Fonctions auxiliaires sur les mots d'entiers $\mathcal{O}$ et $\mathcal{I}$**
La fonction $\mathcal{I}$ permet de passer du mot de l'horloge au mot de la fonction d'indexation, et la fonction $\mathcal{O}$ donne le mot du compteur d'occurrence à partir du mot $\mathbf{u}$ de l'horloge :

$$\forall \mathbf{u} \in \mathbb{N}^\infty, k \in \overline{\mathbb{N}}, \quad \mathcal{O}_{\mathbf{u}}[k] = \sum_{k' \leq k} \mathbf{u}[k'] \qquad \forall \mathbf{u} \in \mathbb{N}^\infty, i \in \mathbb{N}, \quad \mathcal{I}_{\mathbf{u}}[i] = \inf\{k \mid i \leq \mathcal{O}_{\mathbf{u}}[k]\}$$

Les conventions habituelles font que $\mathcal{O}_{\mathbf{u}}[0] = 0 = \mathcal{I}_{\mathbf{u}}[0]$. De plus, nous retrouvons bien que $\mathcal{I}_{\mathbf{u}}[i]$ est à valeurs dans $\overline{\mathbb{N}}$ pour donner un sens à $\inf \emptyset$, tandis que $\mathcal{O}_{\mathbf{u}}[k]$ reste dans $\mathbb{N}$ si k l'est.

Des définitions inductives correspondant à celles de F. Plateau sont peut être plus intuitives :

$$\mathcal{I}_{n.\mathbf{u}}[i] = \begin{cases} 1 & \text{si } i \leq n \\ 1 + \mathcal{I}_{\mathbf{u}}[i-n] & \text{si } i > n \end{cases} \qquad \mathcal{O}_{n.\mathbf{u}}[k] = \begin{cases} n & \text{si } k = 1 \\ n + \mathcal{O}_{\mathbf{u}}[k-1] & \text{si } k \leq n \end{cases}$$

8.1.5 *Remarque* Dans sa thèse, F. Plateau utilise la formule $\mathcal{I}_{\mathbf{u}}[i] = \inf\{k \mid i = \mathcal{O}_{\mathbf{u}}[k]\}$. Celle-ci ne tient pas avec les horloges entières, puisque $\mathcal{O}$ peut croître de plus d'une valeur.

8.1.6 *Propriété* ck, I, O **sont tri-univoques**
Pour toutes entrées $\mathbf{D}$ et port p, nous avons par convention $O_{\mathbf{D}}(p)[0] = 0$ et :

$$\forall i \in \mathbb{N}^*, I_{\mathbf{D}}(p)[i] = k < \infty \iff O_{\mathbf{D}}(p)[k-1] < i \leq O_{\mathbf{D}}(p)[k]$$
$$I_{\mathbf{D}}(p) = \mathcal{I}_{ck_{\mathbf{D}}(p)} \qquad \forall k \in \mathbb{N}^*, ck_{\mathbf{D}}(p)[k] = \#\{i \mid I_{\mathbf{D}}(p)[i] = k\}$$
$$O_{\mathbf{D}}(p) = \mathcal{O}_{ck_{\mathbf{D}}(p)} \qquad \forall k \in \mathbb{N}^*, ck_{\mathbf{D}}(p)[k] = O_{\mathbf{D}}(p)[k] - O_{\mathbf{D}}(p)[k-1]$$

Démonstration Toutes ces formules reposent sur les faits suivants :
- Les instants sont ouverts, disjoints et ordonnés suivant leur indice.
- La fonction de datation est strictement croissante pour tout $\mathbf{D}$ et port p.
- Toutes les valeurs de la fonction de datation appartiennent à un des instants.
- La fonction d'indexation et le compteur d'occurrence sont croissants en k.

8.1.7 *Nota bene* Ces trois représentations sont équivalentes. Nous nous permettrons, suivant les propriétés ou définitions, d'utiliser l'une ou l'autre de ces trois représentations.

8.1.2 Choix des instants et raffinement : le pointage

L'opération inverse de l'abstraction est la concrétisation γ qui associe un ensemble de datations à une abstraction. En interprétation abstraite de laquelle nous tirons les notations et les termes bien que nous n'utilisons aucun des résultats, la définition canonique de la concrétisation $\gamma(I) = \{T \mid \exists R, \alpha(T, R) = I\}$ est peu utilisée car elle n'est pas constructive.

Dans notre cas, la concrétisation décrit l'ensemble des exécutions « réelles » possibles du réseau. Les construire explicitement n'est pas notre but. Par contre, nous souhaitons que ces exécutions soient correctes, ce qui nécessite des précautions.

8.1.8 *Exemple* Prenons le réseau qui calcule l'identité `y = x`. Une datation correcte de ce réseau doit être telle que $\forall i, T(\mathtt{x})[i] < T(\mathtt{y})[i]$. Soit la datation correcte $T_{one}(\mathtt{x})[i] = i + 0{,}1$ et $T_{one}(\mathtt{y})[i] = i + 0{,}6$. Avec les instants $R^k =]k; k+1[$, elle s'abstrait en $\alpha(T_{one}, R) = I_{one}$ avec pour tout i, $I_{one}(\mathtt{x})[i] = I_{one}(\mathtt{y})[i] = i$ ou de manière équivalente $ck_{one}(\mathtt{x}) = ck_{one}(\mathtt{y}) = 1^\omega$.

Pourtant, la datation $T_{inv}(\mathtt{x})[i] = i + 0{,}6$ et $T_{inv}(\mathtt{y})[i] = i + 0{,}1$ n'est pas correcte, mais elle appartient à la concrétisation du même pointage puisque $\alpha(T_{inv}, R) = I_{one}$.

Sans contrainte sur la concrétisation, l'exemple précédent illustre la perte d'information critique, qui arrive avec une sortie et une entrée dans le même instant. Cependant, si le code généré est une fonction de transition `step` appelée une fois par instant (comme en Heptagon), alors nous savons que les entrées seront lues strictement avant les sorties. Dans ce cas, l'exemple n'est pas problématique puisque la concrétisation ne devrait pas contenir (T_{inv}, R). Pour refléter les propriétés du code généré, nous allons raffiner la concrétisation avec une contrainte $c(T, R)$ qui filtre les cas impossibles.

8.1.9 *Définition* **Concrétisation raffinée**
La concrétisation raffinée est l'ensemble des exécutions admises par l'abstraction, raffiné par la contrainte c :

$$\gamma_c(I) \stackrel{\text{def}}{=} \{T \mid \exists R, c(T, R) \wedge \alpha(T, R) = I\}$$

Symétriquement, le choix des instants en Heptagon est restreint aux instants dont toutes les entrées sont strictement avant les sorties. Cette convention est une règle de bonne formation des horloges, cohérente avec la génération de code sous la forme d'une fonction `step`.

8.1.10 *Définition* **Abstraction raffinée : le pointage**
Les horloges calculées par le pointage des langages synchrones, résultent d'un choix d'instants compatibles avec une contrainte c. L'abstraction raffinée α_c définit l'ensemble des pointages possibles :

$$\alpha_c(T) \stackrel{\text{def}}{=} \{\alpha(T,R) \mid c(T,R)\}$$

8.1.11 *Remarque* Nous avons la propriété essentielle $\forall T, I \in \alpha_c(T), T \in \gamma_c(I)$.

8.1.12 *Nota bene* **Adéquation entre les choix des instants et le code généré**
Les libertés dans le choix des instants sont spécifiées par les contraintes qu'impose le code généré. Réciproquement, en permettant certains choix d'instants, nous imposons au code généré de pouvoir s'exécuter en les respectant.

Cette adéquation est essentielle et sera assurée par l'utilisation d'une abstraction et d'une concrétisation partageant toujours la même contrainte de raffinement.

Nous discuterons de trois approches différentes. Celle en « boucle simple » de Scade, Lucy-n, Lucid Synchrone, Heptagon, etc. L'utilisation d'une spécification ad-hoc pour les objets [Cas+09] appelée « politique » et finalement du cas de Signal qui permet de décrire ces contraintes avec des formules booléennes [ABL95].

D'autres conditions subalternes existent. Par exemple en Heptagon, il est impossible d'avoir plus d'une valeur par port et par instant puisque les horloges sont booléennes. Ces contraintes peuvent changer suivant les versions du compilateur ou même les options de compilation. Quoi qu'elles soient, nous les agrégeons en une unique contrainte c.

8.1.13 *Nota bene* Le choix des instants est effectué [1] par le compilateur : *il choisit* $I \in \alpha_c(T)$.

Suivant les contraintes c, il sera possible ou pas de trouver des instants permettant d'abstraire une datation. S'il ne trouve pas d'abstraction à une datation, le compilateur devra rejeter cette datation (et donc le programme qui va avec), ce qui restreint les datations que nous allons considérer aux datations pointables :

8.1.14 *Définition* **Datation c-pointable**
Une datation T est dite c-pointable ou simplement pointable ssi elle est réactive et il est possible de l'abstraire : $\exists R, c(T,R)$

8.1.3 Pointage correct, principal et fidèle

Pour les datations, deux propriétés sont apparues importantes, la correction et la principalité. Puisque le pointage a vocation à remplacer les datations, nous transposons ces propriétés dans le monde abstrait synchrone. La correction est une propriété *nécessaire à toute datation*, un pointage ne doit être considéré correct que si toutes ses concrétisations le sont :

8.1.15 *Définition* **Correction d'un pointage**
Un pointage I est γ_c-correct ssi sa concrétisation ne contient que des datations correctes [2] :

$$\gamma_c\text{-correct}(I) \iff \gamma_c(I) \neq \emptyset \wedge \forall T \in \gamma_c(I), \text{correct}(T)$$

1. Nous pensons malheureusement que ce choix est souvent un choix de modélisation et le programmeur va devoir écrire les choses sous une certaine forme pour obtenir du compilateur un certain choix d'instants. Donner au programmeur plus de pouvoir, et donc de libertés, est une des principales motivations de notre recherche dans les langages synchrones.

La principalité d'un pointage est une propriété d'expressivité qui assure que le comportement du réseau est entièrement capturé :

8.1.16 *Définition* **Principalité d'un pointage**
Un pointage I est γ_c-principal ssi il est correct et pour toute datation correcte, sa concrétisation en contient une qui l'inclut.

$$\gamma_c\text{-principal}(I) \iff \gamma_c\text{-correct}(I) \quad \wedge \quad \forall T', \exists T \in \gamma_c(I), \quad \text{correct}(T') \implies T' \sqsubseteq T$$

Comme toutes les datations ne sont pas forcément c-pointables, il est possible qu'un réseau n'ait pas de pointage principal alors qu'il a une datation principale [3]. Ce cas de figure est bien évidemment préjudiciable et nous chercherons au maximum à ce qu'un c-pointage soit fidèle :

8.1.17 *Définition* **Fidélité du pointage avec les contraintes c**
Le pointage avec contraintes c est dit fidèle ssi tout réseau qui admet une datation principale admet un pointage γ_c -principal.

Inversement, la principalité d'un pointage n'implique pas l'existence d'une datation correcte, ce qui assure une plus grande expressivité. Nous verrons que cela arrive avec les objets en section 10.2.3.1.

Dans la suite de ce chapitre, nous étudions le pointage en boucle simple, qui est fidèle.

8.2 Pointage en boucle simple (cadre général)

Comme nous l'avons souligné, le choix des contraintes de raffinement du pointage est majoritairement dicté par le code que l'on souhaite générer. La compilation la plus simple et la plus utilisée est celle en boucle simple. Cette compilation génère un code exécutant une boucle infinie constituée de trois phases, la lecture des entrées, puis l'appel à la fonction de transition et finalement l'écriture des sorties.

Le pointage en boucle simple assimile une itération de la boucle de simulation à un instant et donc à un appel de la fonction de transition. Dans notre formalisme, un pointage en boucle simple est donné par la condition c_{bs}, qui requiert que dans l'instant, les entrées arrivent avant les sorties :

$$c_{bs}(T, R) \iff \forall k, e \in E, s \in S, t_e \in T_R^k(e), t_s \in T_R^k(s), t_e < t_s$$

Cette contrainte est tellement naturelle que nous l'avons respectée dans la construction de toutes nos datations : T_{canon}, T_{early} et T_{repr} (cf. sections 6.1.5.1 et 6.1.5.3 et lemme 7.3.11).

8.2.1 *Définition*
Pour faciliter les notations, nous définissons la relation $\lessdot$ entre parties de $\overline{\mathbb{R}}_+$ qui permet de comparer des ensembles de dates en traitant de manière transparente l'ensemble vide.

$$\forall X, X' \in \mathfrak{P}(\overline{\mathbb{R}}_+), \quad X \lessdot X' \iff X = \emptyset \vee X' = \emptyset \vee X < X'$$

Cette relation n'est pas irréflexive mais presque : $X \lessdot X \implies X = \emptyset$ Elle n'est pas non plus transitive, mais il est tout de même possible d'écrire des chaînes avec la convention que : $X_1 \lessdot X_2 \lessdot \cdots \lessdot X_n \overset{\text{def}}{\iff} \forall i,j, 1 \leq i < j \leq n, \quad X_i \lessdot X_j$

2. Pour la correction d'une datation voir définition 7.1.3 et propriété 7.3.1
3. Un exemple immédiat est d'avoir une contrainte c insatisfaisable.

Grâce à cette notation, nous pouvons écrire simplement :

$$c_{bs}(T,R) \quad \iff \quad \forall k, T_R^k(E) \lessdot T_R^k(S)$$

8.2.1 Définitions et propriétés de base

Le cœur du pointage en boucle simple est donné par la propriété suivante :

8.2.2 *Propriété* **Préservation de l'ordre relatif entre les entrées et les sorties**
Le pointage en boucle simple $\alpha_{c_{bs}}$ préserve l'ordre relatif des entrées et des sorties. Pour toute datation T réactive et c_{bs} pointable, fonction d'indexation $I \in \alpha_{c_{bs}}(T)$, flot d'entrées $\mathbf{D}$, port d'entrée $e \in E$ et port de sortie $s \in S$:

$$\forall i, \qquad T_{\mathbf{D}}(e)[i] < T_{\mathbf{D}}(s)[i'] \quad \iff \quad I_{\mathbf{D}}(e)[i] \leq I_{\mathbf{D}}(s)[i'] \ \wedge \ I_{\mathbf{D}}(e)[i] < \infty$$

Démonstration ($\Rightarrow$) Puisque T est pointable, il existe une fonction d'indexation $I \in \alpha_{c_{bs}}(T)$. Ensuite, $I = \alpha(T,R)$ avec $c_{bs}(T,R)$. Nous avons $T_{\mathbf{D}}(e)[i]$ fini donc appartenant à un instant R^k : $I_{\mathbf{D}}(e)[i] = k$. Pour être supérieure, $T_{\mathbf{D}}(s)[i']$ ne peut pas être dans un instant inférieur à R^k : soit elle est infinie et $I_{\mathbf{D}}(s)[i']$ l'est aussi, soit elle est dans un instant $R^{k'}$ avec $k \leq k'$.
($\Leftarrow$) Puisque $I_{\mathbf{D}}(e)[i] = k$ est fini, nous avons $T_{\mathbf{D}}(e)[i] \in R^k$. Nous avons $k \leq I_{\mathbf{D}}(s)[i']$, donc soit $T_{\mathbf{D}}(s)[i']$ est infinie, soit elle est dans un instant supérieur ou égal à k. Le cas limite k est réglé par $c_{bs}(T,R)$.

Cette propriété est très forte, elle assure un ordre relatif entre toutes les entrées et toutes les sorties. Presque tous nos résultats sur les datations réactives se basaient sur cela. Le pointage en boucle simple va pouvoir prendre la place des datations réactives.

8.2.3 *Définition* **Relation d'inclusion entre pointages en boucle simple**
Tout comme pour les datations (cf. définition 7.1.4), un pointage en boucle simple I précède un autre I' ssi l'ordre qu'il définit entre les entrées et les sorties est compatible avec celui de I'. Cette relation est un pré-ordre :

$$I \sqsubsetneq I' \quad \iff \quad \forall \mathbf{D}, k_e, k_s, e \in E, s \in S, \ I'_{\mathbf{D}}(e)[k_e] \leq I'_{\mathbf{D}}(s)[k_s] \implies I_{\mathbf{D}}(e)[k_e] \leq I_{\mathbf{D}}(s)[k_s]$$

8.2.4 *Définition* **Relation d'équivalence entre pointages en boucle simple**
Comme pour les datations (cf. définition 7.1.5), un pointage en boucle simple I est équivalent à un autre I' ssi l'ordre qu'il définit entre les entrées et les sorties est le même que celui de I'.

$$I \backsim I' \qquad \iff \qquad I \sqsubsetneq I' \wedge I' \sqsubsetneq I$$

8.2.5 *Propriété* **Inclusion et équivalences de pointages en boucles simples**
Pour toutes datations pointables T et T', T est incluse dans T' ssi son pointage en boucle simple est inclus dans celui de T' :

$$\forall T, T', I \in \alpha_{c_{bs}}(T), I' \in \alpha_{c_{bs}}(T'), \qquad T \sqsubsetneq T' \qquad \iff \qquad I \sqsubsetneq I'$$

La conséquence immédiate est que nos classes d'équivalence sont alors identiques :

$$\forall T, T', I \in \alpha_{c_{bs}}(T), I' \in \alpha_{c_{bs}}(T'), \qquad T \backsim T' \qquad \iff \qquad I \backsim I'$$

Démonstration La propriété 8.2.2 a besoin d'être étendue par une étude du cas où le pointage est infini. Si $I_{\mathbf{D}}(e)[k_e]$ est infini et $I_{\mathbf{D}}(e)[k_e] \leq I_{\mathbf{D}}(s)[k_s]$, alors $I_{\mathbf{D}}(s)[k_s]$ aussi est infini. L'argument est alors que cette sortie n'est pas produite, mais le pointage contient une datation correcte, elle-même mettant à l'infini cette sortie, donc cette sortie ne *peut pas* être produite par le réseau et vaut l'infini pour toute datation correcte, donc pour tout pointage en boucle simple correct.

8.2.6 *Remarque* Nous choisissons dans cette section d'utiliser les notations $I_{\mathbf{D}}(p)[k]$ au lieu d'utiliser des flots $I_{\mathbf{D}}(\mathbf{X})$ comme nous faisions avec les datations. Ceci est un choix esthétique, qui permet de coller à la représentation de $O_{\mathbf{D}}(p)$ avec un mot d'entiers. C'est ce choix qui rend la preuve précédente non immédiate.

8.2.2 Fidélité

La propriété 8.2.5 permet d'exprimer la principalité d'un pointage en boucle simple sous une forme ne faisant pas intervenir les datations :

8.2.7 *Propriété* **Principalité d'un pointage en boucle simple**
Le pointage I est principal ssi il est correct et inclut tout autre pointage correct :

$$\gamma_{c_{bs}}\text{-principal}(I) \qquad \Longleftrightarrow \qquad \forall I', \gamma_{c_{bs}}\text{-correct}(I'), \quad I' \sqsubseteq\!\!\!\backsim I$$

Nous ne pouvons donner dans un premier temps qu'un sens de la preuve :

Démonstration ($\Rightarrow$) Les pointages corrects ne sont pas vides. Nous pouvons donc prendre une datation T dont l'abstraction est I ainsi que T' pour I'. Puisque I est principal, nous avons $T' \sqsubseteq\!\!\!\backsim T$, donc par la propriété 8.2.5, $I' \sqsubseteq\!\!\!\backsim I$. ($\Leftarrow$) Nous fixons une datation T de I. Pour toute datation T' correcte et *pointable*, nous prenons une de ses abstractions I', elle est inclue dans I par hypothèse, donc $T' \sqsubseteq\!\!\!\backsim T$. Que faire des datations qui ne sont pas pointables ? En utilisant la propriété 8.2.9, nous pouvons conclure, car toute datation T'' admet une datation T' pointable et équivalente, alors finalement $T'' \backsim T' \sqsubseteq\!\!\!\backsim T$.

Pour finir la preuve, il nous manque la possibilité de prendre en compte les datations non pointables.

Lemme 8.2.8 (Non concomitance)**.** *Toute datation T est équivalente à une datation T' telle que deux valeurs n'ont jamais la même date :*

$$\forall p,p',k,k', \quad T'(p)[k] \neq T'(p')[k']$$

Démonstration Par définition, une datation n'a pas de point d'accumulation. Donc pour toute date, il existe un voisinage tel qu'il n'y a pas d'autres dates. En particulier, pour toute date qui correspond à une entrée et une sortie $T(e^{(i)})[k] = T(s^{(j)})[k']$, il existe pour tout $\epsilon > 0$ un voisinage de diamètre $3\epsilon > 0$ qui ne contient aucune autre date. Nous définissons une datation T' égale à T partout sauf à ces dates problématiques que nous déplaçons ainsi : $T'(s^{(j)})[k'] = T(s^{(j)})[k'] - \epsilon$. Elle préserve l'ordre *strict* entre les dates des entrées et des sorties, donc elle est bien équivalente à T.

8.2.9 *Propriété* **Pointage total modulo equivalence**
Pour toute datation T, il existe une datation $T' \backsim T$, telle que T' est pointable en boucle simple (il existe des instants R tels que $c_{bs}(T', R)$).

Démonstration Le lemme 8.2.8, permet de définir une datation équivalente à T dont les entrées et les sorties n'ont jamais la même date. Comme par définition une datation n'a pas de point d'accumulation, il est possible de définir des instants n'incluant que des entrées ou bien que des sorties.

Outre la possibilité de terminer la preuve de la propriété 8.2.7, cette dernière propriété rend le pointage en boucle simple fidèle :

8.2.10 Théorème
Un réseau est ordonné réactif ssi il admet un pointage en boucle simple principal.

Démonstration Un réseau ordonné réactif admet une datation principale réactive (cf. théorème 7.3.15) qui est équivalente à une datation pointable T'(cf. propriété 8.2.9). Ainsi T' est principale, tout comme les pointages appartenant à son abstraction.
S'il existe un pointage principal, alors nous choisissons T une de ses datations et prouvons qu'elle est principale. À nouveau, pour toute datation T'', il existe une datation T' équivalente et pointable. Comme le pointage de T est principal, il inclut celui de T' donc $T'' \backsim T' \sqsubseteq T$.

La preuve du théorème et en particulier de la propriété 8.2.9 reste valide si nous raffinons encore jusqu'à avoir au maximum une valeur par instant.

8.2.2.1 Variante booléenne

Les horloges booléennes ne dépassent jamais une valeur par port et par instant :

$$c_{boolbs}(T, R) \quad \Longleftrightarrow \quad c_{bs}(T, R) \wedge \forall \mathbf{D}, k, p, ck_{\mathbf{D}}(p)[k] \leq 1$$

8.2.11 Théorème
Un réseau est ordonné réactif ssi il admet un pointage booléen, en boucle simple et principal.

8.2.2.2 Variante exclusive [4]

Les horloges exclusives sont booléennes et ne peuvent être vraies simultanément :

$$c_{exc}(T, R) \quad \Longleftrightarrow \quad \forall \mathbf{D}, k, \exists p_0, \quad ck_{\mathbf{D}}(p_0)[k] = 1 \quad \wedge \quad \forall p \neq p_0, ck_{\mathbf{D}}(p)[k] = 0$$

8.2.12 Théorème
Un réseau est ordonné réactif ssi il admet un pointage exclusif, en boucle simple et principal.

4. L'idée originale de ce pointage est d'Adrien Guatto. C'est entre autre cette vision des horloges qui a inspiré nos travaux sur les datations.

8.2.3 Fonction de transition et signatures principales

Le pointage en boucle simple décrit dans l'instant un comportement fonctionnel strict en forçant les entrées à arriver avant les sorties. Pour tout $\mathbf{D}$, à chaque instant k, la fonction de transition associée au pointage consomme $ck_{\mathbf{D}}(e^{(i)})[k]$ valeurs de la $i$ème entrée et produit $ck_{\mathbf{D}}(s^{(j)})[k]$ valeurs pour la sortie j. Pour ressembler au typage usuel de fonction, nous pouvons écrire la signature d'horloge correspondante [5] :

$$\forall \mathbf{D}, \left(ck_{\mathbf{D}}(e^{(1)}), \cdots, ck_{\mathbf{D}}(e^{(d)})\right) \rightarrow \left(ck_{\mathbf{D}}(s^{(1)}), \cdots, ck_{\mathbf{D}}(s^{(r)})\right)$$

La fonction de transition correspondante est déterministe (cf. définition 7.4.1) car elle n'admet pour l'entrée $\mathbf{D}$ que la suite de transition $(s_{k-1}, \mathbf{X}_{\delta k}, \mathbf{Y}_{\delta k}, s_k)$, avec :

$$ck_{\mathbf{D}}(e^{(i)})[k] = \left|\mathbf{X}^{(i)}_{\delta k}\right| \qquad \text{et} \qquad ck_{\mathbf{D}}(s^{(j)})[k] = \left|\mathbf{Y}^{(j)}_{\delta k}\right|$$

Remarquez que de manière équivalente, si nous regardons l'intégralité de ce qui a été consommé (resp. produit), le pointage se définit avec les compteurs d'occurrence :

$$O_{\mathbf{D}}(e^{(i)})[k] = \left|\mathbf{X}^{(i)}_{k}\right| \qquad \text{et} \qquad O_{\mathbf{D}}(s^{(j)})[k] = \left|\mathbf{Y}^{(j)}_{k}\right|$$

Il y a de nombreuses signatures équivalentes, mais nous avons un nouveau critère de comparaison que nous n'avions pas avec les datations. Comparer la valeur des dates ne faisait pas sens puisque l'unité de temps n'est pas fixée et que le temps est dense. Par contre, les pointages sont accrochés aux instants qui sont indexés avec des entiers positifs.

8.2.13 *Définition* **Rapidité d'un pointage**
Un pointage en boucle simple O est plus rapide que O', ce que nous notons $O' \preceq O$ ssi il consomme et produit les valeurs plus tôt :

$$O' \preceq O \qquad \Longleftrightarrow \qquad \forall \mathbf{D}, \forall p, \forall k, \quad O'_{\mathbf{D}}(p)[k] \leq O_{\mathbf{D}}(p)[k]$$

La relation de rapidité est un ordre partiel.

8.2.14 *Exemple* **Choix des instants et fonction de transition**
Prenons l'exemple du réseau F_{whenft} correspondant à l'équation `y = x when (01)`. Si l'entrée est le flots $\mathbf{x} = x_1.x_2.x_3.x_4 \cdots$ alors le réseau supprime une valeur sur deux : $F_{whenft}(\mathbf{x}) = x_2.x_4.x_6.\cdots$. Le contrôle ne dépend pas de $\mathbf{x}$, donc le pointage non plus. Considérons quelques signatures d'horloges et la fonction de transition correspondante :

— La signature $2^\omega \rightarrow 1^\omega$ indique qu'à chaque instant, la fonction de transition attend deux valeurs pour en fournir une. La fonction de transition associée est la forme normale du réseau :

$$tnF_{whenft2}(s_0, x.x') = (x', s_0)$$

— La signature booléenne $1^\omega \rightarrow (0.1)^\omega$ indique qu'à chaque instant, la fonction de transition attend une valeur, mais n'en fournit une que tous les instants pairs. Une fonction de transition correspondante est :

$$tF_{whenftb}(s_0, x) = (\varepsilon, s_1) \qquad\qquad tF_{whenftb}(s_1, x) = (x, s_0)$$

5. Rappelons que les $e^{(i)}$ sont les ports d'entrées et les $s^{(i)}$ les ports de sorties.

— La signature exclusive $(1.1.0)^\omega \to (0.0.1)^\omega$ peut être associée à la fonction de transition suivante :

$$tF_{whenftexc}(s_0, x) = (\varepsilon, s_1) \quad tF_{whenftexc}(s_1, x) = (\varepsilon, s_2(x)) \quad tF_{whenftexc}(s_2(x), \varepsilon) = (x, s_0)$$

Ces trois pointages sont principaux et contiennent les mêmes datations mais avec des choix différents d'instants. Leur principalité se retrouve dans le fait que les fonctions de transition associées calculent la même fonction : F_{whenft}. Bien que toutes équivalentes, elles ne sont pas toutes aussi rapides et nous avons :

$$((1.1.0)^\omega \to (0.0.1)^\omega) \prec (1^\omega \to (0.1)^\omega) \prec (2^\omega \to 1^\omega)$$

Il est toujours possible de ralentir le pointage en insérant des instants qui ne consomment et ne produisent rien. Par contre, il n'est pas possible d'aller plus vite que la datation $(2^\omega \to 1^\omega)$ sans changer la sémantique du réseau, ce que nous prouvons avec le théorème suivant.

Un réseau, qui ne produit qu'une sortie finie s'arrête de produire à partir d'un certain rang q. Toutes les entrées qu'il consomme par la suite sont inutiles, une signature principale peut ou pas les pointer. Un exemple basique est le réseau de l'équation `y = x when 1(0)` qui ne garde que la première valeur de `x` pour produire un flot composé d'une seule valeur. Ce réseau admet comme signature principale $1.0^\omega \to 1.0^\omega$, mais aussi des signatures principales consommant plus de valeurs, comme $1.1^\omega \to 1.0^\omega$ ou $1.n^\omega \to 1.0^\omega$ quel que soit n. Notez que $1.0^\omega \to 1.0^\omega \prec 1.1^\omega \to 1.0^\omega \prec 1.2^\omega \to 1.0^\omega$. Ainsi, en consommant des entrées inutiles, une signature est plus rapide. Ce qui n'est pas très intuitif, puisqu'une signature plus rapide est une signature qui devrait « finir plus tôt ». Outre le fait de consommer des valeurs inutiles, ces signatures sont sources de nombreux cas particuliers à gérer. Nous les évitons en nous restreignant aux signatures nettoyées :

8.2.15 *Définition* **Signature nettoyée**
Une signature O est nettoyée ssi le réseau ne consomme que s'il va encore produire :

$$\forall \mathbf{D}, n, (\forall k \geq n, \forall s \in S, O(s)[k] = O(s)[n]) \implies (\forall k \geq n, \forall e \in E, O(e)[k] = O(e)[n])$$

Si la signature est donnée par la fonction d'indexation I :

$$\forall i, e, I(e)[i] < \infty \implies \exists j, s, I(e)[i] \leq I(s)[j] < \infty$$

L'ensemble des signatures principales d'un réseau est partiellement ordonné par la rapidité ($\preceq$). La restriction à celles qui sont nettoyées rend possible l'existence d'une signature la plus rapide. Nous la nommons comblée et son existence correspond à l'interactivité du réseau :

8.2.16 *Définition* **Signature comblée**
Une signature est dite comblée, ssi elle est la signature principale nettoyée la plus rapide.

Un certain nombre d'exemples sont donnés en sections 9.2.4 et 9.2.5.
Cette signature est celle de la forme normale de la fonction de transition :

8.2.17 Théorème **Signature comblée**
Tout réseau interactif ordonné admet une signature comblée O correspondant à sa fonction de transition

en forme normale tnF. Soit la chaîne des inverses du réseau $(\mathbf{Y}_k^{*\mathbf{D}})_k$ *avec* q *(possiblement infini) éléments. En posant* $z = k$ *pour les réseaux Moore et* $z = k + 1$ *pour les réseaux Mealy, pour tout* $k > 0$ *:*

$$O_{\mathbf{D}}(e^{(i)})[k] = \left|\mathbf{X}_k^{(i)}\right| \text{ avec } \mathbf{X}_k = \mathbf{Y}^{*\mathbf{D}}_{\min(z,q)} \qquad O_{\mathbf{D}}(s^{(j)})[k] = \left|\mathbf{Y}_k^{(j)}\right| \text{ avec } \mathbf{Y}_k = \mathbf{Y}^{**\mathbf{D}}_{\min(z,q)}$$

Une signature comblée correspond à la fonction de transition en forme normale, qui est normale grâce à deux propriétés, la productivité et la maximalité (cf. définition 7.4.4). Ce sont ces deux-mêmes propriétés qui permettent de reconnaître le pointage comblé :

8.2.18 *Propriété* **Caractérisation d'une signature comblée**
Productivité : si les horloges de toutes les sorties sont nulles à un instant, alors elles le resteront pour tous les instants suivants, ainsi que les entrées. *Maximalité* : si les horloges des entrées sont nulles à un instant, soit c'est le premier instant d'un réseau Moore, soit elles le resteront pour tous les instants suivants et les horloges de sorties seront aussi nulles.

Le pointage ck est comblé ssi il est principal et pour tout $\mathbf{D}$:

$$\begin{aligned}
&\forall n \geq 1, \quad (\forall s \in S, ck_{\mathbf{D}}(s)[n] = 0) \implies (\forall k \geq n, \forall e \in E, ck_{\mathbf{D}}(s)[k] = ck_{\mathbf{D}}(e)[k] = 0)\\
&\forall n > 1, \quad (\forall e \in E, ck_{\mathbf{D}}(e)[n] = 0) \implies (\forall k \geq n, \forall s \in S, ck_{\mathbf{D}}(e)[k] = ck_{\mathbf{D}}(s)[k] = 0)
\end{aligned}$$

8.2.3.1 Réseaux réactifs et signature quasi-comblée

Le théorème 8.2.17 ne considère que des réseaux ordonnés interactifs alors qu'il existe une signature principale pour les réseaux ordonnés réactifs (cf. théorème 8.2.10). Un réseau ordonné réactif non-interactif (cf. remarque 7.3.16) a un dernier bi-inverse infini. Sa signature comblée demanderait d'avoir une sortie infinie durant un instant, ce qui n'est pas acceptable avec la vision un instant pour un appel à la fonction de transition.

8.2.19 *Exemple* Le réseau donné par l'équation `y = merge 11(0) x 42` ne consomme que deux valeurs de `x` avant de répéter infiniment 42. Ce réseau a seulement trois inverses, ε, le préfixe de taille 1 de `x`, qui permet de produire la première valeur de sortie et le préfixe de taille 2 de `x`, qui permet de produire toutes les sorties.

Une signature principale binaire est $1.1.0^\omega \to 1.1.1^\omega$ mais la signature $1.1.0^\omega \to 1.1.4^\omega$ est tout aussi principale et plus rapide. Ce réseau n'a pas de signature principale la plus rapide étant donné qu'une fois les entrées consommées, nous pouvons accélérer la sortie jusqu'à l'infini. Une possibilité pour lui donner une signature principale la plus rapide serait de permettre des mots finis pour les horloges et d'accepter d'écrire $1.1.0 \to 1.1.\infty$. Cependant, la gestion de telles signatures complexifie la présentation et nous perdons l'équivalence, un instant / un appel à la fonction de transition.

8.2.20 *Définition* **Signature quasi-comblée**
Soit un réseau ordonné réactif et sa chaîne de longueur n d'inverses $(\mathbf{Y}_k^{*\mathbf{D}})_{0<k}$. Nous posons $z = k$ et $q = n$ pour les réseaux Moore et $z = k + 1$ et $q = n - 1$ pour les Mealy. Une signature quasi-comblée O est la plus rapide des signatures principales, excepté pour ses sorties à partir du $q$ième instant :

$$\forall 0 < k, O_{\mathbf{D}}(e^{(i)})[k] = \left|\left(\mathbf{Y}^{*\mathbf{D}}_{\min(z,n)}\right)^{(i)}\right| \qquad \forall 0 < k < q, O_{\mathbf{D}}(s^{(j)})[k] = \left|\left(\mathbf{Y}_z^{**\mathbf{D}}\right)^{(j)}\right|$$

Remarquez que si $n = \infty$, le réseau est interactif et une signature quasi-comblée est comblée.

8.2.21 *Remarque* Sans contrainte de granularité, il sera pratique de produire le dernier bi-inverse une valeur par instant : avec une horloge de 1^ω. Ainsi pour `y = merge 11(0) x 42`, nous privilégierons la signature quasi-comblée $1.1.0^\omega \to 1.1.1^\omega$, à la place de signatures principales plus rapides comme $1.1.0^\omega \to 1.1.4^\omega$ ou plus lentes comme $1.1.0^\omega \to 1.1.(01)^\omega$.

8.2.22 *Propriété* **Quasi-comblée est nettoyée**
Une signature quasi-comblée est nettoyée.

Conclusion Tous les réseaux ordonnés admettent une datation principale nettoyée. Pour les réseau interactifs, il en existe une qui est la plus rapide et dite comblée. Pour les autres réseau, il existe les signatures quasi-comblées.

8.2.4 Ensemble d'occurrence des inverses

La correspondance très étroite, établie par le théorème 8.2.17, entre les inverses du réseau et le compteur d'occurrence d'une signature quasi-comblée, invite une autre représentation de l'ensemble des inverses. Au lieu de manipuler les flots des inverses, nous allons manipuler uniquement leurs dimensions.

8.2.23 *Définition* **Ensemble d'occurrence des inverses**
L'ensemble d'occurrence des inverses ou simplement ensemble d'occurrence, $\mathrm{Occ}(F, \mathbf{D})$ ne garde que les dimensions des inverses :

$$\mathrm{Occ}(F, \mathbf{D}) = \left\{ (n_1, \cdots, n_d) \;\middle|\; \forall \mathbf{X} \in \mathrm{Inv}(F, \mathbf{D}), n_i = \left|\mathbf{X}^{(i)}\right| \right\}$$

Notre utilisation de l'ensemble des inverses se réduit essentiellement à vérifier qu'il est bien totalement ordonné. Pour porter ce test sur l'ensemble des occurrences, nous avons besoin de définir un ordre partiel $\preceq$ sur les tuples d'entiers de telle sorte que $\mathrm{Occ}(F, \mathbf{D})$ est totalement ordonné pour $\preceq$ ssi $\mathrm{Inv}(F, \mathbf{D})$ l'est pour $\sqsubseteq$. L'ordre partiel unanime rempli cette tâche.

8.2.24 *Définition* **Ordre partiel unanime sur les tuples d'entiers**
Un tuple est inférieur, pour l'ordre partiel unanime $\preceq$, à un autre tuple si chacune de ses composantes est inférieure :

$$(n_1, \cdots, n_r) \preceq (n'_1, \cdots, n'_r) \quad \iff \quad n_1 \leq n'_1 \wedge \cdots \wedge n_r \leq n'_r$$

8.2.25 *Propriété* **Ordre partiel des flots et ordre partiel unanime de leurs tailles**
Deux flots $\mathbf{X}$ et $\mathbf{Y}$ compatibles, sont tels que l'un inclut l'autre ssi leurs dimensions sont ordonnées pour l'ordre partiel unanime :

$$\forall \mathbf{X} \uparrow \mathbf{Y}, \qquad \mathbf{X} \sqsubseteq \mathbf{Y} \quad \iff \quad \left(\left|\mathbf{X}^{(1)}\right|, \cdots, \left|\mathbf{X}^{(r)}\right|\right) \preceq \left(\left|\mathbf{Y}^{(1)}\right|, \cdots, \left|\mathbf{Y}^{(r)}\right|\right)$$

Ce qui permet de conclure la propriété que nous cherchons :

8.2.26 *Propriété* **Réseau ordonné et ensemble d'occurrence**
Un réseau F stable est ordonné ssi pour tout flot d'entrées $\mathbf{D}$, son ensemble d'occurrence $\mathrm{Occ}(F, \mathbf{D})$ est totalement ordonné pour l'ordre partiel unanime. De plus, pour tout pointage quasi-comblé O d'un réseau Moore, nous avons :

$$\mathrm{Occ}(F, \mathbf{D}) = \left\{ (O_{\mathbf{D}}(e^{(1)})[k], \cdots, O_{\mathbf{D}}(e^{(d)})[k]) \;\middle|\; \forall k \right\}$$

S'il est Mealy, il faut en plus rajouter à l'ensemble des occurrences $(0, 0)$.

8.2.27 *Remarque* Nous utilisons la même notation $\preceq$ pour l'ordre partiel unanime des tuples d'entiers que pour la rapidité d'un pointage (cf. définition 8.2.13). Ceci ne pose pas de problème car leurs définitions coincident si l'on considère un pointage comme un tuple infini formé de la concaténation des tuples pour tous les instants. Nous pouvons même redéfinir la rapidité d'un pointage :

$$O \preceq O' \quad \Longleftrightarrow \quad \forall \mathbf{D}, \forall k \quad \left(O_{\mathbf{D}}(e^{(1)})[k], \cdots, O_{\mathbf{D}}(s^{(r)})[k]\right) \preceq \left(O'_{\mathbf{D}}(e^{(1)})[k], \cdots, O'_{\mathbf{D}}(s^{(r)})[k]\right)$$

8.2.28 *Remarque* **L'ordre lexicographique ne convient pas**
Le fait que l'ordre lexicographique soit un ordre total est déjà rédhibitoire. L'exemple de Gonthier le montre clairement. Son ensemble d'occurrence est formé des couples de deux entiers puisqu'il peut produire quelle que soit son entrée : $\mathrm{Occ}(F_{Gon}, (\mathbf{a}, \mathbf{b})) = \{(a, b) \mid 0 \leq a \leq |\mathbf{a}| \wedge 0 \leq b \leq |\mathbf{b}|\}$. Cet ensemble est totalement ordonné pour l'ordre lexicographique, mais par pour l'ordre unanime puisqu'il peut contenir $(0, 1)$ et $(1, 0)$. Ce qui prouve une nouvelle fois que Gonthier n'est pas ordonné.

8.2.5 Accélérer une signature

Les signatures comblées sont importantes, autant en pratique qu'en théorie, puisqu'elles sont les plus rapides. Nous étudions dans cette section le lien qu'elles ont avec les autres signatures. En propriété 8.2.18, nous expliquons qu'une signature comblée est sans « trou ». Pour accélérer une signature, il faut combler ces trous.

8.2.29 *Propriété* **Accélérer une signature par fusion**
Pour tout pointage en boucle simple O (associé à ck), s'il n'est pas le plus rapide, il existe un instant, tel que les horloges de toutes les entrées (resp. toutes les sorties) sont nulles. Cet instant peut être fusionné avec l'instant précédent (resp. suivant) pour former un pointage en boucle simple *équivalent* mais strictement plus rapide.

$$\exists \mathbf{D}, \exists n, \quad (\forall e \in E, ck_{\mathbf{D}}(e)[n+1] = 0) \vee (\forall s \in S, ck_{\mathbf{D}}(s)[n] = 0) \implies \exists O' \backsim O \wedge O' \prec O$$

$$\text{Avec} \qquad \forall p, \forall k, \quad O'_{\mathbf{D}}(p)[k] = \begin{cases} O_{\mathbf{D}}(p)[k] & \text{si } k < n \\ O_{\mathbf{D}}(p)[n+1] & \text{si } k = n \\ O_{\mathbf{D}}(p)[k+1] & \text{si } k > n \end{cases}$$

Démonstration Nous allégeons les notations en considérant que tout est annoté avec $\mathbf{D}$. Faisons la preuve du cas où ce sont les sorties qui sont toutes d'horloges nulles.

Montrons que toute datation T de la concrétisation de O est aussi dans O'. Nous allons montrer que cette datation est toujours correcte en boucle simple quand elle est soumise aux instants R' égaux aux instants R dont on a fusionné les instants R^n et R^{n+1}, avec $R'^n =]\inf(R^n); \sup(R^{n+1})[$ et $R'^k = R^k$ si $k < n$ et $R'^k = R^{k+1}$ si $k > n$.

Il suffit alors de vérifier que $T_{R'}$ est bien en boucle simple et de pointage O', ce qui est assuré puisque nous avions $T^n_R(E) \lessdot T^n_R(S) \lessdot T^{n+1}_R(E) \lessdot T^{n+1}_R(S)$, mais comme toutes les entrées sont d'horloges nulles, $T^n_R(S) = \emptyset$ et donc $T^n_R(E) \lessdot T^{n+1}_R(E) \lessdot T^n_R(S) \lessdot T^{n+1}_R(S)$, ce qui permet d'écrire $(T^n_R(E) \cup T^{n+1}_R(E)) \lessdot (T^n_R(S) \cup T^{n+1}_R(S))$, ce qui est finalement égal à $T^n_{R'}(E) \lessdot T^n_{R'}(S)$. Pour les autres instants, rien n'est changé.

Finalement, en suivant le cheminement dans l'autre sens, nous prouvons que toute datation de O' est aussi une datation de O.

La fusion associée à l'accélération d'une signature peut être vue comme un changement des instants utilisés et donc de la fonction de transition associée. Mais elle peut aussi être vue comme l'accélération de l'échelle du temps de base et donc l'appel à deux reprises de l'ancienne fonction de transition dans le même instant. Illustrons notre propos :

8.2.30 *Exemple* **Suite de l'exemple 8.2.14, étude de** F_{whenft}

Accélération de $(1.1)^\omega \to (0.1)^\omega$ **à** $2^\omega \to 1^\omega$ **:** la signature booléenne[6] $(1.1)^\omega \to (0.1)^\omega$ est associée à la fonction de transition $tF_{whenftb}$. Tous les instants impairs, les sorties sont d'horloges nulles. Ils peuvent donc être fusionnés avec les instants suivants (pairs). Ces fusions se résument avec le mot $\text{base}_{b2} = 2^\omega$ qui a pour signification qu'à chaque instant k, la fonction de transition $tF_{whenftb}$ est appelée $\text{base}_{b2}[k] = 2$ fois. Effectuer deux appels concaténés de $tF_{whenftb}$ correspond à effectuer deux pas en un seul. En utilisant les notations plus visuelle de transition étiquetées :

$$\text{false} \boxed{\xrightarrow{x/\varepsilon} \text{true} \xrightarrow{x'/x'}} \text{false} \qquad \text{devient} \qquad \text{false} \xrightarrow{x.x'/x'} \text{false}$$

Ainsi, le résultat de la fusion est la fonction de transition en forme normale du réseau que nous avions définie : $tnF_{whenft2}(x.x', \text{false}) = (x', \text{false})$ de signature $2^\omega \to 1^\omega$.

Accélération de $(1.1.0)^\omega \to (0.0.1)^\omega$ **à** $(1.1)^\omega \to (0.1)^\omega$ **:** cette accélération résulte de la fusion avec le précédent de tous les instants divisibles par trois. La fonction de transition $tF_{whenftexc}$:

$$0 \xrightarrow{x/\varepsilon} 1 \boxed{\xrightarrow{x'/\varepsilon} (2, x') \xrightarrow{\varepsilon/x'}} 0 \qquad \text{devient} \qquad 0 \xrightarrow{x/\varepsilon} 1 \xrightarrow{x'/x'} 0$$

Nous pouvons résumer cette fusion avec le mot $\text{base}_{excb} = (1.2)^\omega$ indiquant bien que tous les instants pairs du résultat contiennent deux appels à la fonction de transition exclusive.

Accélération de $(1.1.0)^\omega \to (0.0.1)^\omega$ **à** $2^\omega \to 1^\omega$ **:** En effectuant trois pas d'un coup, nous serions retombés sur la fonction de transition $tnF_{whenft2}$ et la signature comblée. Le mot résumant cette fusion est $\text{base}_{exc2} = 3^\omega$.

8.2.5.1 Changement d'horloge d'activation : l'opérateur dr

8.2.31 *Définition* **Horloge d'activation**

Le mot résumant les fusions à effectuer est une *horloge d'activation*. Sa valeur à l'instant k décide du nombre d'activations à effectuer.

6. Nous avons déroulé le mot 1^ω de l'entrée en $(1.1)^\omega$, pour avoir la même période que celui de la sortie.

L'opérateur dr travaille sur une paire de mots d'entiers ($\mathbf{u}$, $\mathbf{v}$) et adopte une notation infixe $\mathbf{u}\ \mathrm{dr}\ \mathbf{v}$. Il permet de calculer l'horloge d'un flot en appliquant une horloge d'activation $\mathbf{u}$ à son ancienne horloge $\mathbf{v}$.

8.2.32 *Définition* **L'opérateur associatif** dr
La définition la plus naturelle est co-récursive et applique la fusion au fur et à mesure, pour tout n entier, $c_1, \ldots, c_n$ entiers et mots d'entier $\mathbf{u}$ et $\mathbf{v}$:

$$(n.\mathbf{u})\ \mathrm{dr}\ (c_1.\cdots.c_n.\mathbf{v}) = \left(\sum_{i=1}^{n} c_i\right).(\mathbf{u}\ \mathrm{dr}\ \mathbf{v}) \qquad \varepsilon\ \mathrm{dr}\ \mathbf{v} = \varepsilon \qquad (n.\mathbf{u})\ \mathrm{dr}\ \mathbf{v} = \varepsilon \text{ si } |\mathbf{v}| < n$$

Nous suivons la convention usuelle pour la somme, donc $(0.\mathbf{u})\ \mathrm{dr}\ \mathbf{v} = 0.(\mathbf{u}\ \mathrm{dr}\ \mathbf{v})$. Si l'on souhaite que les deux flots soient totalement consommés, il faut avoir $\sum_{i=1}^{|\mathbf{u}|} \mathbf{u}[i] = |\mathbf{v}|$.

8.2.33 *Propriété* **Neutre à gauche**
Le mot 1^ω est le neutre à gauche : $1^\omega\ \mathrm{dr}\ \mathbf{v} = \mathbf{v}$, ce qui est bien attendu d'une horloge d'activation activant une fois à chaque instant.

8.2.34 *Propriété* **Associativité**
L'associativité suit trivialement celle de la somme : $\mathbf{u1}\ \mathrm{dr}\ (\mathbf{u2}\ \mathrm{dr}\ \mathbf{v}) = (\mathbf{u1}\ \mathrm{dr}\ \mathbf{u2})\ \mathrm{dr}\ \mathbf{v}$

L'horloge d'activation est particulièrement utilisée pour l'activation d'un nœud et donc d'une fonction de transition. Dans ce cas, l'horloge d'activation s'applique à toutes les horloges de la signature puisqu'elle partagent la même horloge de base.

8.2.35 *Définition* **Extension aux tuples et aux signatures**
Nous étendons, point à point, l'opérateur dr aux tuples puis aux signatures :

$$\begin{aligned} \mathbf{u}\ \mathrm{dr}\ (\mathbf{v1}, \cdots, \mathbf{vn}) &= (\mathbf{u}\ \mathrm{dr}\ \mathbf{v1}, , \mathbf{u}\ \mathrm{dr}\ \mathbf{vn}) \\ \mathbf{u}\ \mathrm{dr}\ \Big((\mathbf{V}^{(1)}, \cdots, \mathbf{V}^{(d)}) \to (\mathbf{W}^{(1)}, \cdots, \mathbf{W}^{(s)})\Big) &= \\ (\mathbf{u}\ \mathrm{dr}\ \mathbf{V}^{(1)}, \cdots, \mathbf{u}\ \mathrm{dr}\ \mathbf{V}^{(d)}) &\to (\mathbf{u}\ \mathrm{dr}\ \mathbf{W}^{(1)}, \cdots, \mathbf{u}\ \mathrm{dr}\ \mathbf{W}^{(s)}) \end{aligned}$$

8.2.36 *Exemple* **Suite de l'exemple 8.2.30**
Nous pouvons vérifier, par le calcul, les horloges d'activations que nous avions trouvées dans notre précédent exemple :

$$\begin{aligned} &\mathrm{base}_{b2}\ \mathrm{dr}\ (1^\omega \to (0.1)^\omega) = 2^\omega\ \mathrm{dr}\ (1^\omega \to (0.1)^\omega) = (2^\omega\ \mathrm{dr}\ 1^\omega) \to (2^\omega\ \mathrm{dr}\ (0.1)^\omega) = 2^\omega \to 1^\omega \\ &\mathrm{base}_{excb}\ \mathrm{dr}\ ((1.1.0)^\omega \to (0.0.1)^\omega) = ((1.2)^\omega\ \mathrm{dr}\ (1.1.0)^\omega) \to ((1.2)^\omega\ \mathrm{dr}\ (0.0.1)^\omega) = 1^\omega \to (0.1)^\omega \end{aligned}$$

L'associativité de dr permet de définir l'horloge d'activation qui passe directement de la plus lente à la plus rapide de nos signatures :

$$\begin{aligned} 2^\omega \to 1^\omega &= \mathrm{base}_{b2}\ \mathrm{dr}\ (\mathrm{base}_{excb}\ \mathrm{dr}\ ((1.1.0)^\omega \to (0.0.1)^\omega)) \\ &= (\mathrm{base}_{b2}\ \mathrm{dr}\ \mathrm{base}_{excb})\ \mathrm{dr}\ ((1.1.0)^\omega \to (0.0.1)^\omega) = 3^\omega\ \mathrm{dr}\ ((1.1.0)^\omega \to (0.0.1)^\omega) \end{aligned}$$

Finalement, nous trouvons bien que $\mathrm{base}_{exc2} = 3^\omega = \mathrm{base}_{b2}\ \mathrm{dr}\ \mathrm{base}_{excb}$.

8.2.5.2 Horloges d'activations quelconques et subtilitées

Que se passe-t-il si l'horloge d'activation fusionne trop d'instants ? Par exemple, activons deux fois à chaque instant la fonction de transition comblée. La signature obtenue est $2^\omega \text{ dr } (2^\omega \to 1^\omega) = 4^\omega \to 2^\omega$. La fonction de transition obtenue est $tF_{whenft4}(s_0, x_1.x_2.x_3.x_4) = (x_2.x_4, s_0)$. Cette fonction de transition est de granularité supérieure à la granularité de la fonction de transition en forme normale, ce qui est impossible par la propriété 7.4.8.

La sémantique du réseau est altérée par ces fusions : $\text{exec}(tF_{whenft4})(x_1.x_2) = \varepsilon$ alors que $\text{exec}(tF_{whenft2})(x_1.x_2) = x_2$. La raison est que ces fusions sont appliquées alors que ni les entrées ni les sorties ne sont d'horloges nulles.

8.2.37 *Nota bene* Remarquez que la sémantique est altérée bien qu'elle coincide si l'entrée $\mathbf{x}$ est infinie : $\text{exec}(tF_{whenft2})(\mathbf{x}) = \text{exec}(tF_{whenft4})(\mathbf{x}) = \mathbf{y}$ avec $\forall k, \mathbf{y}[k] = \mathbf{x}[2k]$. Le fait que les sémantiques soient égales avec une entrée infinie mais différentes pour des flots finis est typique d'une compilation qui a « changé la causalité » avec un choix non modulaire, en l'occurrence trop augmenter la granularité. Finalement, la signature n'est plus principale et il existe des contextes d'appels qui mettront à jour ces différences.

Peut-on prendre n'importe quelle horloge d'activation si elle est « plus lente » que la « plus rapide » ? Il est très tentant de parler de rapidité pour l'horloge d'activation, alors que ceci n'a pas de sens bien défini. La rapidité est définie sur un pointage : une signature d'horloge. La grande différence est que la signature d'horloge encode les dépendances de données d'un réseau ordonné réactif et lie la rapidité des entrées et des sorties.

Dans notre exemple, nous avons vu qu'il est possible d'accélérer la signature $(1.1.0)^\omega \to (0.0.1)^\omega$ avec l'horloge d'activation $(1.2)^\omega$ et l'horloge d'activation 3^ω en restant équivalent et donc sans changer la sémantique du réseau décrit. Par contre, l'accélérer avec l'horloge d'activation 2^ω change la sémantique du réseau.

Nous avons $2^\omega \text{ dr } ((1.1.0)^\omega \to (0.0.1)^\omega) = (2.1.1)^\omega \to (0.1.1)^\omega$. Nous groupons par deux les transitions et la période est de trois. Pour calculer le système de transition $tF_{whenft22}$ résultant, nous déplions une fois la période :

$$0 \boxed{\xrightarrow{x/\varepsilon} 1 \xrightarrow{x'/\varepsilon}} (2, x') \boxed{\xrightarrow{\varepsilon/x'} 0 \xrightarrow{x/\varepsilon}} 1 \boxed{\xrightarrow{x'/\varepsilon} (2, x') \xrightarrow{\varepsilon/x'}} 0$$

qui devient :

$$0 \xrightarrow{x.x'/\varepsilon} (2, x') \xrightarrow{x/x'} 1 \xrightarrow{x'/x'} 0$$

Encore une fois, si l'entrée est infinie, la sémantique est la même tandis qu'elles diffèrent sur des flots finis : $\text{exec}(tF_{whenft22})(x_1.x_2) = \varepsilon$ mais $\text{exec}(tF_{whenft2})(x_1.x_2) = x_2$.

Ce comportement est dû à l'horloge d'activation 2^ω qui fusionne le troisième et le quatrième instant de la signature $(1.1.0)^\omega \to (0.0.1)^\omega$. D'après la propriété 8.2.29, pour pouvoir les fusionner, il aurait fallu soit qu'au troisième instant les sorties soient d'horloges nulles, soit qu'au quatrième instant les entrées soient d'horloges nulles. Cette fusion force l'entrée à avoir trois valeurs avant que la sortie puisse produire une valeur, comme l'atteste le fait que $\text{exec}(tF_{whenft22})(x_1.x_2) = \varepsilon$ mais $\text{exec}(tF_{whenft22})(x_1.x_2.x_3) = x_2$.

8.2.6 Ralentir une signature

S'il y a des 0 dans l'horloge d'activation, alors à ces instants, il n'y aura pas d'activation et les horloges des entrées *et* des sorties seront nulles. Par exemple $(0.1)^\omega \text{ dr } (2.3)^\omega = (0.2.0.3)^\omega$.

8.2.38 *Propriété* **Horloge d'activation booléenne**
Si l'horloge d'activation ne contient que des 0 et des 1, alors la signature résultante reste *équivalente* à la signature initiale, mais elle est strictement plus lente à partir du moment où il y a au moins un 0.

Démonstration L'ordre entre les entrées et les sorties n'est pas changé par l'ajout d'instants qui ne produisent et ne consomment rien.

Cependant, nous ne pouvons pas avec cette opération passer de la signature $2^\omega \to 1^\omega$ à $(1.1.0)^\omega \to (0.0.1)^\omega$, puisqu'elle n'a pas d'instant dont toutes les horloges sont nulles. Nous devons la ralentir pour ne produire une valeur que tous les trois instants. Le plus proche que nous pouvons faire est de l'activer sur $(0.1.0)^\omega$, qui donne $(0.1.0)^\omega \text{ dr } (2^\omega \to 1^\omega) = (0.2.0)^\omega \to (0.1.0)^\omega$, or nous cherchons à obtenir la signature $(1.1.0)^\omega \to (0.0.1)^\omega$.

En regardant ce que nous essayons de faire sur les fonctions de transition, cette impossibilité est claire. Nous tentons de découper un appel à la fonction de transition $tF_{whenft2}$ en deux sous-appels, ce qui n'est pas possible sans en connaître l'intérieur.

Par contre, nous pouvons stocker les entrées jusqu'à en avoir assez, appeller la fonction de transition $tF_{whenft2}$, stocker ses sorties et ne les fournir que plus tard. Illustrons avec la fonction de transition $tF_{whenft2buff}$. Son état, initialisé à $(0, 0)$, stocke temporairement les entrées et les sorties ainsi que l'état de $tF_{whenft2}$:

$$(0, s) \xrightarrow{x/\varepsilon} (1, x, s) \underset{\text{avec } (y,s')=tF_{whenft2}(s,x.x')}{\xrightarrow{x'/\varepsilon}} (2, y, s') \xrightarrow{\varepsilon/y} (0, s')$$

La signature de cette fonction de transition est celle que nous attendons : $(1.1.0)^\omega \to (0.0.1)^\omega$ et l'utilisation de $tF_{whenft2}$ est effectuée sur $(0.1.0)^\omega$.

Pour autant, nous ne souhaitons pas programmer à la main le stockage des entrées et des sorties. Mieux, nous ne souhaitons pas stocker explicitement ces valeurs, mais les laisser dans les files de communication du réseau. Le ralentissement d'une signature ne donnera donc jamais toutes les signatures possibles et les adaptations devront être faites par les files de communication.

8.2.6.1 Avancer des entrées et retarder des sorties à mémoire bornée

À la fin, les files de communication sont de taille bornée. Notre formalisme, très proche de celui de F. Plateau [Pla10], permet d'utiliser ses résultats et en particulier la définition de la relation d'adaptabilité ($<:$). Celle-ci indique l'existence d'une file bornée pour communiquer le flot de p vers p' en comparant les quantités produites et consommées avec $O(p) <: O(p')$:

$$w <: w' \qquad \iff \qquad \exists B, \forall k, \quad 0 \leq w[k] - w'[k] \leq B$$

Pour paraphraser la formule, à chaque instant la quantité produite est supérieure ou égale à celle consommée et la différence n'excède pas B. Dans [Pla10], seules des horloges booléennes sont considérées. Pour entièrement assurer une taille bornée à la file, il faut en plus assurer que $ck_{\mathbf{D}}(p)[k]$ est uniformément bornée. Sinon, dans l'instant, il peut y avoir transfert et donc stockage d'une quantité non bornée de valeurs. Nous discuterons plus en détail de la taille des files en section 9.3.2.

La notion d'adaptabilité s'étend aisément aux signatures :

8.2.39 *Définition* **Adaptabilité des signatures**
Un pointage O est adaptable en O', ssi le passage de l'un à l'autre correspond à rajouter des files de tailles bornées sur les entrées et les sorties :

$$O <: O' \qquad \Longleftrightarrow \qquad \forall \mathbf{D}, \forall e \in E, s \in S, \quad O_{\mathbf{D}}(e) <: O'_{\mathbf{D}}(e) \wedge O'_{\mathbf{D}}(s) <: O_{\mathbf{D}}(s)$$

Remarquez que la relation est inversée pour les sorties.

Dans l'exemple précédent ($tF_{whenft2}$), nous souhaitions stocker les entrées et les sorties pour adapter la signature $(0.2.0)^\omega \to (0.1.0)^\omega$ en $(1.1.0)^\omega \to (0.0.1)^\omega$. C'est effectivement possible puisque $\mathcal{O}_{(1.1.0)^\omega} <: \mathcal{O}_{(0.2.0)^\omega}$ et $\mathcal{O}_{(0.1.0)^\omega} <: \mathcal{O}_{(0.0.1)^\omega}$.

8.2.40 *Propriété*
L'adaptabilité est l'inclusion à files bornées.

$$O <: O' \implies O' \sqsubseteq_{\backsim} O$$

Le changement de sens entre ces relations est malheureux. La tension vient du fait que l'inclusion $\sqsubseteq_{\backsim}$ est une approche de typage : le type le plus général est le plus « grand » tandis que l'adaptabilité est une approche temporelle : le plus « grand » est le plus tardif.

Ceci, couplé à la fidélité et à la propriété 7.1.6 assure l'évidence : ajouter des files sur les entrées et les sorties ne rend pas le comportement incorrect :

8.2.41 *Propriété*
Si O' est correct et $O <: O'$ alors O est correct.

8.2.42 *Remarque* **Avancer des entrées / retarder des sorties sans perdre la principalité**
En règle générale, adapter une signature O en O' ($O' <: O$) ne préserve pas l'ordre entre les entrées et les sorties ($\not\Rightarrow (O' \backsim O)$). Par exemple puisque l'horloge 1^ω est adaptable à l'horloge 0.1^ω, nous avons la signature $(2^\omega \to 1^\omega)$ adaptable en $(2^\omega \to 0.1^\omega)$. Or la seconde requiert 4 entrées avant de fournir sa première sortie quand la deuxième en requiert 2 : elles ne sont pas équivalentes.

Pour avancer une entrée, tout en restant équivalent, il faut que l'instant la précédant n'ait pas de sortie. Pour retarder une sortie, c'est l'instant suivant qui doit ne pas avoir d'entrée. Bref, il ne faut pas changer l'ordre relatif entre entrées et sorties. Ces conditions ne sont toutefois pas utiles en pratique bien qu'elles fournissent de l'intuition.

8.2.6.2 Discussion sur les opérateurs dr et on

Nous avions déjà discuté du changement de l'horloge de base d'un nœud en Heptagon pour donner la sémantique de Kahn synchrone en section 2.3. Cela concernait uniquement des horloges d'activations binaires pour ne pas activer le nœud à tous les instants. L'opérateur que nous utilisions est le `on`. Contrairement à dr, il a pour deuxième argument un échantillonneur : un flot booléen *du programme*.

La seule sémantique que nous avons fournie à `on` (cf. figure 2.4) était une sémantique de Kahn synchrone ($\mathrm{on}_i^{\mathcal{S}}(\mathbf{ck}, \mathbf{p})$), avec **ck** l'horloge d'activation et **p** le flot synchrone du pointeau (avec les absences explicites). L'horloge résultante était vraie uniquement si l'horloge d'activation était vraie et si le pointeau valait la valeur attendue (i).

En oubliant comment est calculé l'échantillonneur, il est possible de voir l'opérateur on comme un opérateur binaire prenant une horloge et un flot de booléen. Ci-dessous, la sémantique de Kahn synchrone à gauche et de Kahn sans absence à droite. La sémantique de Kahn est celle que l'on peut trouver par exemple dans [Pla10] :

$$\mathrm{on}^{\mathcal{S}}(\mathrm{true}.\mathbf{ck}, \mathrm{true}.\mathbf{c}) = \mathrm{true}.\,\mathrm{on}^{\mathcal{S}}(\mathbf{ck}, \mathbf{c}) \qquad \mathrm{on}^{\mathcal{K}}(\mathrm{true}.\mathbf{ck}, \mathrm{true}.\mathbf{c}) = \mathrm{true}.\,\mathrm{on}^{\mathcal{K}}(\mathbf{ck}, \mathbf{c})$$
$$\mathrm{on}^{\mathcal{S}}(\mathrm{true}.\mathbf{ck}, \mathrm{false}.\mathbf{c}) = \mathrm{false}.\,\mathrm{on}^{\mathcal{S}}(\mathbf{ck}, \mathbf{c}) \qquad \mathrm{on}^{\mathcal{K}}(\mathrm{true}.\mathbf{ck}, \mathrm{false}.\mathbf{c}) = \mathrm{false}.\,\mathrm{on}^{\mathcal{K}}(\mathbf{ck}, \mathbf{c})$$
$$\mathrm{on}^{\mathcal{S}}(\mathrm{false}.\mathbf{ck}, \mathit{abs}.\mathbf{c}) = \mathrm{false}.\,\mathrm{on}^{\mathcal{S}}(\mathbf{ck}, \mathbf{c}) \qquad \mathrm{on}^{\mathcal{K}}(\mathrm{false}.\mathbf{ck}, \mathbf{c}) = \mathrm{false}.\,\mathrm{on}^{\mathcal{K}}(\mathbf{ck}, \mathbf{c})$$

Finalement, l'opérateur on est un sous-cas de dr, avec les booléens encodés par 0 et 1. Ce dernier peut ainsi servir à donner la sémantique de Kahn (non synchrone) de l'opérateur `on`.

8.2.7 Compilation modulaire séparée

Une compilation modulaire séparée consiste à pouvoir découper un réseau pour le traiter en morceaux. Plus particulièrement, la compilation séparée est la possibilité de générer le code d'un sous-réseau indépendamment de ses contextes d'appels. En l'occurrence, le code que nous souhaitons générer est une fonction de transition. La modularité est la possibilité d'utiliser un de ces sous-réseaux en ne connaissant qu'une information partielle : une signature d'horloge principale.

Toutes les pièces du puzzle sont maintenant en place. Nous avons montré, pour les réseaux ordonnés réactifs, comment une signature d'horloge principale décrit la consommation et la production d'une fonction de transition à chacune de ses activations. De plus le théorème 8.2.10 affirme qu'un réseau est ordonné ssi il a une signature principale.

8.2.43 *Nota bene* La compilation modulaire séparée requiert de découper le réseau en sous-réseaux *ordonnés réactifs*.

8.2.8 Conclusion

Le cadre général des horloges que nous avons présenté, utilise des horloges entières et non booléennes. Cette expressivité est nécessaire pour la signature comblée qui décrit la forme normale de la fonction de transition d'un réseau ordonné réactif. Outre le fait de maximiser la granularité, cette signature a deux avantages algorithmiques, elle est la plus dense en information et décrit précisément l'ensemble des inverses et bi-inverses du réseau.

Cependant, s'il existe une signature principale avec des entiers, il en existe une booléenne et même une exclusive avec un seul port actif par instant. Quand la granularité est un désavantage ou pour simplifier le code généré, il est *toujours* possible de se restreindre à ces signatures.

Nous avons montré comment passer d'une signature à une autre, laissant au compilateur ou au programmeur la chance de choisir une signature. Une fois ce choix pris, le réseau est compilé en une fonction de transition correspondante. À l'utilisation, ce choix est réversible en jouant sur l'horloge d'activation et en rajoutant des buffers sur les entrées et les sorties.

Une horloge d'activation avec des valeurs supérieures à 1 va accélérer la signature en fusionnant dans un même instant plusieurs appels à la fonction de transition. Nous avons donné les conditions nécessaires et suffisantes pour que ces fusions ne fassent pas perdre la principalité.

Une horloge d'activation avec des valeurs inférieures à 1 ralentit la signature puisque les 0 insèrent des instants d'inactivité. Pour exprimer toutes les signatures à partir de la signature la plus rapide, il est nécessaire d'ajouter des files bornées sur les entrées et les sorties : le fameux « avancer les entrées et retarder les sorties » de nos discussions sur les datations.

Un point reste en suspens, quel langage utiliser pour exprimer les pointages. Les mots infinis dépendant des entrées que nous avons manipulées dans cette partie ne sont pas calculables statiquement, ce qui nous empêche par exemple de vérifier que les files de communication sont de tailles bornées.

Les langages de la lignée Lustre, Lucid Synchrone, Scade, Lucy-n et Heptagon utilisent une signature d'horloge associée à chaque programme. Cependant, la notion de signature principale n'est pas évoquée telle quelle et nous tâchons d'y remédier dans les chapitres suivants.

Chapitre 9

Pointage en boucle simple ultimement périodique à la Lucy-n

Le langage Lucy-n est initialement le fruit d'une collaboration [Coh+06], mais le principal de son développement a été fait pendant la thèse de F. Plateau [Pla10] avec l'aide de L. Mandel. Un article long résume la partie qui nous intéresse ici, incluant les avancés récentes [MP12]. Cependant, la génération de code n'est pas finalisée à l'heure actuelle.

Lucy-n est un parfait cobaye pour illustrer nos propos sur les RKPN et le pointage en boucle simple. Il est déclaratif flot de données avec une sémantique de Kahn stable. Ses opérateurs sont familiers puisqu'identiques à ceux d'Heptagon (`merge`, `when` et `fby`). Sans le dire, nous avons déjà discuté de Lucy-n au travers de différents exemples, en particulier `y = x when (01)`. La syntaxe utilisée est proche de Lucid Synchrone et donc des langages fonctionnels à la ML, bien qu'il ne soit pas d'ordre supérieur. Il s'en distingue par deux points très importants :

— Les pointeaux sont réduits aux flots constants ultimement périodiques. Leur valeur est interprétée contrairement à ceux d'Heptagon qui sont uniquement définis par leurs noms.

— L'opérateur `buffer` dénote une file de communication non synchrone de taille bornée.

La volonté principale des concepteurs de Lucy-n est de relâcher la synchronie des communications grâce à `buffer`, tout en pouvant déterminer statiquement la taille des files de communication. Les réseaux nécessitant des files infinies sont rejetés. Pour assurer statiquement ces contraintes, le flot de contrôle du programme est entièrement statique : les pointeaux sont réduits à des constantes ultimement périodiques. Le pointage et les contraintes de files bornées sont posées sous la forme d'un problème d'optimisation linéaire en nombres entiers (cf. [Pla10 ; MP12]), les contraintes linéaires étant dues à l'adaptation (cf. section 8.2.6.1) des horloges de l'entrée et de la sortie des `buffer`. L'optimisation peut minimiser la surface totale des `buffer` ou bien maximiser le taux du réseau. Ce système est résolu à l'aide de GLPK.

Dans sa thèse [Pla10], F. Plateau étudie deux pointages, l'un exact en boucle simple et l'autre qui abstrait plus encore le pointage en ne gardant qu'une enveloppe affine du compteur d'occurrence. Nous n'étudions ici que le pointage exact en boucle simple.

Comme le flot de contrôle est ultimement périodique, un programme Lucy-n s'apparente très fortement à un modèle CSDF [EBL94]. Ainsi, une approche globale d'ordonnancement du programme est envisageable.

Cependant, notre but est de trouver un ordonnancement qui produise une signature *principale* de façon à avoir un ordonnancement modulaire et une compilation séparée. La minimisation de la surface des files ou bien la maximisation du débit du réseau ne sont que secondaires

dans notre approche. Nous étudions un langage de programmation et il nous semble que la modularité et la compilation séparée sont primordiales. Sans elles, l'inlining et/ou l'outlining ne sont pas corrects, ce qui est pour le moins déroutant en tant que programmeur et limitant pour les transformations automatiques de code.

Par définition, les signatures principales autorisent une compilation séparée sans perte de comportement, chose actuellement ignorée en LUCY-n. Selon nous, ceci est la source actuelle de la fragilité du système de type, qui peut nécessiter beaucoup d'annotations pour assurer la modularité et la causalité. Nous montrerons aussi comment profiter pleinement des mots ultimement périodiques. En effet, LUCY-n propose un langage unique pour décrire le régime transitoire et périodique, ce qui est une grande avancée par rapport aux modèles comme les CSDF. Cette promesse n'est actuellement pas entièrement tenue avec beaucoup de programmes rejetés. Avec les signatures principales nettoyées, nous proposons une solution ne polluant pas le code généré du régime périodique. Au passage, nous discutons de la notion de causalité dans le cadre de LUCY-n, nous amenant à proposer une notion de causalité dans le cadre général des réseaux de Kahn. Finalement nous montrerons les difficultés que peut rencontrer notre approche avec la question de la découpe du programme en réseaux ordonnés réactifs.

9.1 Introduction

9.1.1 *Nota bene* **UNIQUEMENT DES SIGNATURES PRINCIPALES**
Sauf explicitement indiqué, toutes les signatures présentées dans ce chapitre sont *principales* et implicitement quand nous parlons de signature, nous parlons de signature *principale*.

Avant de montrer les conséquences pratiques de notre recherche de signatures principales, donnons la propriété qui assure que ce que nous cherchons est calculable. Cette propriété se retrouve sous diverses formes et en particulier dans les problèmes d'ordonnancements.

9.1.2 *Propriété* **ULTIMEMENT PÉRIODIQUE**
Tout réseau LUCY-n est ordonné réactif ssi il admet une signature quasi-comblée, dont toutes les horloges sont ultimement périodiques.

Démonstration **MÉTA-PREUVE**
L'intuition est simple et repose sur la notion de configuration du réseau :

9.1.3 *Définition* **CONFIGURATION D'UN RÉSEAU LUCY-n**
Une configuration est l'état du contrôle et la quantité de valeurs stockées dans chacune des files de communications.

9.1.4 *Propriété* **FINITUDE DES CONFIGURATIONS**
En LUCY-n, les réseaux acceptés sont à mémoire bornée. De plus, le contrôle est décidé par les pointeaux qui sont eux-mêmes des constantes ultimement périodiques. Le nombre total de configurations possibles est fini.

Le comportement du réseau et donc les horloges des entrées et sorties dépendent uniquement de la configuration du réseau. Si le réseau est ordonné réactif, il admet une signature principale quasi-comblée. Si elle est comblée, cette signature est la plus rapide à tous les instants, donc obligatoirement ultimement périodique puisque le nombre de configurations est fini. Si elle n'est que quasi-comblée, elle a un nombre fini d'inverses et au moins une sortie infinie. Comme elle ne consomme que ce qui est nécessaire et en quantité finie, les horloges

des entrées sont nulles dans la partie périodique. À partir de ce moment, les sorties peuvent être générées comme bon nous semble, y compris périodiquement.

Cette pseudo-preuve requiert de prouver, avant l'ordonnancement, que le réseau peut s'exécuter en mémoire bornée. Des critères nécessaires et suffisants seront donnés au fur et à mesure. Bien que nous soyons dans un cadre plus expressif, ils dérivent tous des classiques équations d'équilibre des SDF/CSDF

9.2 Briques de base

9.2.1 Mots ultimement périodiques

Nous avons déjà largement utilisé les mots ultimement périodiques a la Lucy-n. De manière générale, un mot peut être utilisé pour dénoter une expression définissant un *flot constant du programme* et admet donc une syntaxe concrète [1]. Les mots considérés dans la plupart du travail de F. Plateau [Pla10] sont binaires, mais ceci ne change rien de fondamental. Le mot $0.1.2.3.(4.5)^\omega$ est dénoté par `0123(45)` en syntaxe concrète. La sémantique de Kahn de cette syntaxe est le mot correspondant. Si u et v sont des mots finis d'entiers, alors

$$[\![u\texttt{(}v\texttt{)}]\!]^{\mathcal{K}}_{H,\rho} = u.v^\omega$$

9.2.1 *Nota bene* **Écriture d'échantillonneur avec un mot de `0` et de `1`**
Nous utiliserons de manière transparente l'écriture d'un mot booléen et d'un mot binaire. Ainsi, la syntaxe `0(01)` a, suivant le contexte, la sémantique $\mathrm{false}.(\mathrm{false}.\mathrm{true})^\omega$ ou bien $0.(0.1)^\omega$. Alors, les échantillonneurs eux-mêmes peuvent s'écrire avec cette syntaxe et il y a équivalence entre `x when (2 = 1(12))` et `x when 0(01)`, ce qui permet de retomber sur la syntaxe booléenne utilisée en Lucy-n.

Avant de traiter de manière générale la signature d'horloge d'une constante, nous souhaitons insister sur un point qui peut être source de confusions :

9.2.2 *Remarque* **Pointeaux et horloges**
Bien que le pointage d'un programme Lucy-n est ultimement périodique et donc avec des horloges représentées par des mots ultimement périodiques, il est important de distinguer ces mots des constantes ultimement périodiques utilisées comme pointeaux. Les premiers sont des objets mathématiques et les seconds des flots du programme calculés à l'exécution, à un rythme déterminé par leur horloge.

9.2.3 *Définition* **Taille et somme d'un mot d'entiers**
Un mot d'entiers $\mathbf{u}$ est comme un flot, nous notons $|\mathbf{u}|$ l'entier possiblement infini donnant sa taille. La somme de ses éléments est :

$$\|\mathbf{u}\| = \sum_{k=1}^{|\mathbf{u}|} \mathbf{u}[k]$$

1. Ces expressions ne peuvent être utilisées en Lucy-n que pour l'expression des pointeaux. Cependant, puisqu'elles définissent un flot, il n'y a pas de raison de les refuser. Par exemple l'équation `y = x + 0(2)` additionne la première valeur du flot `x` avec `0` puis les suivantes avec `2`.

9.2.2 Les constantes

Les constantes sont caractérisées par le fait de ne pas avoir d'entrées (cf. section 6.3.4). Que ce soit la constante stationnaire `0` de sémantique 0^ω ou bien un mot ultimement périodique $u(v)$ de sémantique $u.v^\omega$ et plus généralement un réseau sans entrée `f()`.

Comme tout réseau, une constante peut se compiler avec une fonction de transition. Cette fonction de transition n'a pas d'entrée, mais ne pas avoir d'entrée est comme en avoir une qui n'est jamais consommée et donc d'horloge 0^ω. Ainsi toute signature est quasi-comblée et nous avons entière liberté pour choisir l'horloge de la sortie, tant que la somme de ses éléments est égale à la longueur de la constante.

Dans le cas d'un nœud sans entrée `f()`, le code est déjà compilé en une fonction de transition et associé à une signature d'horloge de la forme $0^\omega \to \mathbf{w}$.

En règle générale, une constante n'a pas vocation à être compilée séparément. Dans ce cas, la signature la plus pratique est la quasi-comblée $0^\omega \to 1^n.0^\omega$, avec n la taille de la constante. Si la constante est infinie, ce sera :

$$0^\omega \to 1^\omega$$

L'avantage de ces signatures est de fournir les valeurs à n'importe quel rythme $\mathbf{w}$, tant qu'il ne requiert pas plus de valeurs que la constante ne peut fournir ($\|\mathbf{w}\| \leq n$) et ce simplement en activant la constante avec ce rythme :

$$0^\omega \to \mathbf{w} = \mathbf{w} \text{ dr } (0^\omega \to 1^\omega) = \mathbf{w} \text{ dr } (0^\omega \to 1^n.0^\omega) \quad \text{si } \|\mathbf{w}\| \leq n \text{ et } n < \infty$$

Ainsi, une constante est utilisable à l'envie.

9.2.3 Opérateurs binaires scalaires

Les opérateurs binaires op (par exemple $+$, $-$, etc.) sont appliqués point à point dans la sémantique de Kahn (cf. figure 2.2). Le réseau formé est interactif et donné par l'équation `y = x1` op `x2`, de signature comblée :

$$(1^\omega, 1^\omega) \to 1^\omega$$

9.2.4 Complémentation : `merge`

L'opérateur de complémentation peut être vu comme un réseau à deux entrées et une sortie, décrit par l'équation `y = merge` u`(`v`) x1 x2`. Pour simplifier, le pointeau booléen n'est pas considéré comme une entrée puisque c'est une constante. À chaque activation, sa sortie est égale à l'entrée consommée. L'entrée consommée est choisie suivant la valeur du pointeau : s'il vaut true (1) alors la première entrée est consommée, sinon c'est la seconde. C'est un réseau interactif de signature comblée $(u.(v)^\omega = 1, u.(v)^\omega = 0) \to 1^\omega$. Comme le pointeau est un mot binaire, cette signature peut se réécrire avec la négation not point à point du mot :

$$(u.(v)^\omega, \mathrm{not}(u.(v)^\omega)) \to 1^\omega$$

Quelques exemples que nous utiliserons :

`y = merge 1(0) x1 x2`	:	$(1.0^\omega, 0.1^\omega) \to 1^\omega$
`y = merge (01) x1 x2`	:	$((0.1)^\omega, (1.0)^\omega) \to 1^\omega$
`y = merge (10) x1 x2`	:	$((1.0)^\omega, (0.1)^\omega) \to 1^\omega$
`y = merge 0(01) x1 x2`	:	$(0.(0.1)^\omega, 1.(1.0)^\omega) \to 1^\omega$

9.2.5 Échantillonnage : `when`

L'opérateur d'échantillonnage peut être vu comme un réseau à une entrée et une sortie donné par l'équation `y = x when` $u(v)$, avec $u.v^\omega$ un mot binaire. La signature d'horloge booléenne, donnée par Lucy-n, est $1^\omega \to u.v^\omega$. Elle est principale car à chaque activation, elle consomme une entrée et fournit en sortie *l'entrée courante* quand elle n'est pas filtrée [2].

Cependant, elle n'est pas comblée alors que le réseau est interactif. Quelques exemples, avec la signature principale binaire usuelle et la comblée à droite :

`y = x when (01)`	:	$1^\omega \to (0.1)^\omega$	ou :	$2^\omega \to 1^\omega$	
`y = x when (10)`	:	$1^\omega \to (1.0)^\omega$	ou :	$1.2^\omega \to 1^\omega$	
`y = x when (1001)`	:	$1^\omega \to (1.0.0.1)^\omega$	ou :	$(1.3)^\omega \to 1^\omega$	
`y = x when (0110)`	:	$1^\omega \to (0.1.1.0)^\omega$	ou :	$2.(1.3)^\omega \to 1^\omega$	
`y = x when 010(1)`	:	$1^\omega \to 0.1.0.1^\omega$	ou :	$2.2.1^\omega \to 1^\omega$	
`y = x when (0)`	:	$1^\omega \to 0^\omega$	ou :	$0^\omega \to 0^\omega$	(*)
`y = x when 101(0)`	:	$1^\omega \to 1.0.1.0^\omega$	ou :	$1.2.0^\omega \to 1.1.0^\omega$	(*)

Taux non nul Si l'échantillonneur a au moins un 1 dans sa partie périodique, la signature comblée va pouvoir renvoyer une valeur à chaque instant et pour cela doit consommer des entrées jusqu'à obtenir une nouvelle valeur, la signature est donnée par $ck(\mathtt{y}) = 1^\omega$ et

$$\forall k, \quad ck(\mathtt{x})[k] = \mathcal{I}_{u.(v)^\omega}[k] - \mathcal{I}_{u.(v)^\omega}[k-1]$$

La signature résultante est ultimement périodique, de période plus petite que v. Le préfixe change de taille (il grossit pour l'échantillonneur `(10)` mais diminue pour `010(1)`).

Taux nul Si l'échantillonneur est ultimement égal à 0^ω (cf. les cas marqués avec (*)), il va ultimement filtrer l'intégralité de ses entrées. La signature usuelle requiert tout de même qu'il reçoive des entrées alors qu'il ne fournira plus de sortie. La signature comblée est nettoyée et donc ne consomme pas de valeur inutile.

L'utilité d'avoir une signature nettoyée est discutée en section 9.3.5. Le calcul général de la signature comblée est :

$$\forall k, \begin{cases} ck(\mathtt{x})[k] = \mathcal{I}_{u.(v)^\omega}[k] - \mathcal{I}_{u.(v)^\omega}[k-1] \quad \text{et} \quad ck(\mathtt{y})[k] = 1 & \text{si } \mathcal{I}_{u.(v)^\omega}[k] \neq \infty \\ ck(\mathtt{x})[k] = 0 \quad \text{et} \quad ck(\mathtt{y})[k] = 0 & \text{sinon} \end{cases}$$

9.2.6 Le registre : `fby`

En Lucy-n, le registre est initialisé avec un flot. Il peut être vu comme un réseau réactif à deux entrées et une sortie donné par l'équation `y = x1 fby x2`. Sa sémantique est la même que celle du réseau `y = merge 1(0) x1 x2`. Sa signature comblée est donc la même :

$$(1.0^\omega, 0.1^\omega) \to 1^\omega$$

9.2.4 *Nota bene* **La signature usuelle n'est pas principale**
La signature usuelle pour `fby`, aussi donnée par Lucy-n est $(1^\omega, 1^\omega) \to 1^\omega$. Elle n'est pas principale car au premier instant, seule la valeur de la première entrée est utile pour la sortie. Cette signature s'explique quand on considère le traitement spécial classiquement réservé à `fby`, comme nous le verrons en section 10.1.2.1.

2. Ce passage immédiat de l'expression du pointeau à l'horloge de la sortie fait qu'il est tentant de considérer ces deux éléments comme étant égaux, bien qu'ils soient de nature différente comme nous l'avons souligné en remarque 9.2.2.

9.2.7 Un nœud

La programmation en LUCY-n, tout comme en Heptagon, est l'écriture de fonctions de flots, appelées des nœuds, ayant plusieurs entrées et sorties. Tout comme en Heptagon, le corps du nœud est un système d'équations récursives. Les expressions peuvent être une des briques de base ou bien l'appel à un nœud déjà déclaré.

Par exemple, le réseau Gonthier qui nous a suivi tout du long s'écrit :

```
let node gonthier (x1, x2) = (y1, y2) where
  rec y1 = x1
  and y2 = x2
```

Mais Gonthier n'admet pas de signature principale. Nous ne savons donc pas compiler ce nœud en une fonction de transition. Pour cette raison, nous ne souhaitons pas l'accepter. En section 9.6.3, nous discuterons plus en détail de la découpe d'un `node` écrit par le programmeur en « vrais » nœuds, c'est-à-dire en réseaux ordonnés réactifs.

9.2.5 *Nota bene* **SEULS LES RÉSEAUX ORDONNÉS RÉACTIFS SONT DITS « NŒUDS »**
Nous réservons la notion de nœud aux réseaux ordonnés réactifs. Une fois un nœud défini, le compilateur a pour tâche de lui trouver une signature principale et de le compiler en accord. Plus tard, quand le nœud est appelé, seule sa signature est disponible pour l'analyse et sa fonction de transition pour la génération de code.

9.2.6 *Exemple* `my_fby`
Un nœud valable est par exemple le nœud `my_fby` qui a la même sémantique que la primitive `fby` et qui a pour signature comblée $(1.0^\omega, 0.1^\omega) \to 1^\omega$, s'écrit :

```
let node my_fby (x, y) = merge 1(0) x y
```

9.2.8 Miscs

Notre analyse se concentre sur les signatures et donc sur la notion de fonction. Cependant, un réseau LUCY-n est donné sous forme équationnelle, ce qui offre une grande liberté. En particulier, un nom de variable peut être utilisé dans différentes expressions et un nom de variable peut être défini mais pas utilisé.

Pour se ramener à un cadre fonctionnel et flot de données, nous avons besoin de pouvoir dupliquer un flot. Ceci se retrouve dans les réseaux flots de données originaux de J. Dennis [Den74] qui utilisent les même nœuds que ceux que nous avons présentés plus le nœud de duplication.

9.2.8.1 Duplication : `dup`

En LUCY-n, nous allons dupliquer les flots de façon pédantesque avec le nœud `dup`, de signature comblée $1^\omega \to (1^\omega, 1^\omega)$:

```
let node dup x = (y1, y2) where
  rec y1 = x
  and y2 = x
```

9.2.8.2 Projection : `proj1`

Nous aurons aussi besoin de projections. Par exemple pour oublier le premier flot d'un couple, nous écrivons le nœud Lucy-n `proj1`, de signature comblée $(1^\omega, 0^\omega) \to 1^\omega$

```
let node proj1 (x1, x2) = x1
```

Comme toutes ses signatures principales, l'horloge du deuxième argument de `proj1` est toujours nulle puisqu'il n'est jamais nécessaire. Ceci peut donner l'impression de ne pas remplir le rôle souhaité de jeter les valeurs du deuxième flot. En réalité, la gestion que nous allons faire des « taux nuls » (0^ω) rendra cette signature effective, ce que nous discutons en section 9.3.5.6.

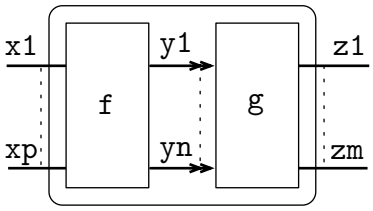
9.2.7 *Remarque* Notez que ce nœud n'est pas nécessaire puisque finalement il a la même sémantique et signature que `merge (1) x1 x2`.

9.3 Composition de réseaux ordonnés

Nous nous intéressons dans cette section à la composition de deux nœuds Lucy-n. Nous avons vus, que la composition de deux nœuds est un nœud (cf. propriété 7.3.21). Nous allons mettre en évidence le lien entre l'ordonnancement des activations des deux sous-nœuds et les signatures principales de leur composition.

Notre approche est focalisée sur la sémantique de Kahn des programmes, la composition de deux réseaux passe donc par des files de communications. En Lucy-n, il n'est pas obligatoire d'utiliser `buffer` pour composer deux nœuds. Nous discuterons, en fin de section, de cette simplification due à l'approche synchrone. Pour l'instant, nous considérons que la composition de deux nœuds `f` et `g` utilise `buffer` pour chaque communication. Le nœud `f_g` résultant de la composition peut s'écrire en Lucy-n avec la syntaxe suivante :

```
let node f_g (x1,···,xp) = (z1,···,zm) where
  rec (y1,···,yn) = f(x1,···,xp)
  and (z1,···,zm) = g(buffer y1,···,buffer yn)
```



Notre but est double, calculer une signature principale pour le réseau `f_g` et le « compiler » en une fonction de transition. Or, le comportement d'un tel réseau est très simple. À tout moment, pour produire des valeurs, il faut activer `g`, pour activer `g` il faut qu'il y ait assez de valeurs dans les `buffer` et pour avoir assez de valeurs dans les `buffer`, il faut activer `f`, qui consomme des entrées.

La question est donc de décider, pour chaque transition de `f_g`, des activations de `f` et de `g`, ainsi que leur ordre. Les ordonnancements possibles ne sont pas uniques et nous pourrions les comparer suivant des critères comme la latence, la quantité de mémoire nécessaire, le débit, etc. La littérature est florissante sur ces sujets. *Notre critère est la principalité de la signature résultante* et donc la fidélité de la compilation séparée à la sémantique de Kahn.

Bien que notre but ne soit pas usuel, l'ordonnancement utile pour l'assurer est classique :

9.3.1 *Définition* **Ordonnancement au plus tard (ALAP) de la composition**
L'ordonnancement au plus tard de `f_g`, active `g` dès que cela est possible et active `f` uniquement quand les files `y1`, …, `yp` ne contiennent pas assez de valeurs pour activer `g`.

Fondamentalement, l'ordonnancement au plus tard est une *simulation du réseau* ayant pour contrainte de consommer des entrées que quand cela est strictement nécessaire. Cette simulation définit un ordre unique entre les activations de `f` et de `g`, duquel nous obtenons une signature principale pour la composition.

9.3.1 Un exemple complet à une seule file de communication

Commençons par le cas très courant d'une composition avec une seule file de communication (`g(buffer f(x))`). Considérons que la signature de `f` est $f : 0.2.1^\omega \to 1.0.3^\omega$ et celle de `g` est $g : 0.(1.2)^\omega \to 1.(0.2)^\omega$.

9.3.1.1 Union ALAP des listes d'occurrence

Pour mieux visualiser le comportement d'un nœud, nous avons recours à sa « liste d'occurrence » (introduite en section 8.2.4). Celle de `f` est :

$$O_f = [O_f(\mathtt{x})[k] \to O_f(\mathtt{y})[k] \mid k = 1, \ldots] = [0 \to 1;\ 2 \to 1;\ 3 \to 4;\ 4 \to 7;\ \cdots]$$

Celle de `g` commence par $[0 \to 1;\ 1 \to 1;\ 3 \to 3;\ 4 \to 3;\ 6 \to 5;\ \cdots]$. L'avantage de l'utilisation des compteurs d'occurrence à la place des horloges est que nous n'avons pas besoin de compter les valeurs stockées dans la file. Pour activer `g`, il suffit que son compteur d'occurrence d'entrées soit inférieur ou égal au compteur d'occurrence des sorties d'une précédente activation de `f`.

La première activation de `g` ne requiert pas de valeurs, tandis que la seconde en requiert une, il faut alors activer `f` une fois, ce qui fournit exactement 1. Par contre, il faut activer `f` deux fois de plus avant d'avoir assez pour réactiver `g`. Mais cette troisième activation de `f` fournit 4, ce qui permet d'activer deux fois `g`, etc. L'ordonnancement s'écrit avec la liste d'occurrence :

$$O_{f\_g} = [\underline{0} \xrightarrow{g} 1;\ 0 \xrightarrow{f} \underline{1};\ \underline{1} \xrightarrow{g} 1;\ 2 \xrightarrow{f} \underline{1};\ 3 \xrightarrow{f} \underline{4};\ \underline{3} \xrightarrow{g} 3;\ \underline{4} \xrightarrow{g} 3;\ 4 \xrightarrow{f} \underline{7};\ \underline{6} \xrightarrow{g} 5;\ \cdots]$$

Nous avons souligné les occurrences des entrées de `g` qui doivent être supérieures aux dernières occurrences des sorties de `f`, elles aussi soulignées.

9.3.1.2 Période, configuration et taille maximale de la file

Comment retrouver une description ultimement périodique ? Pour trouver la période, il nous faut retrouver deux fois une même configuration. Comme nous l'avons expliqué en définition 9.1.3, une configuration est l'état du contrôle plus l'état des files. Pour le contrôle, nous notons g^i la $i$ème activation de la période de `g`. Les activations du préfixe ne sont pas à distinguer. L'état de la file est altérée à chaque activation. Nous notons le nombre de valeurs contenues à la fin d'un activation en dessous du point-virgule. Dans notre cas, la configuration qui débute la période est une file vide, `f` en attente pour f^1 et `g` en attente pour g^2 :

$$O_{f\_g} = \left[\underset{0}{\underline{0} \xrightarrow{g} 1};\ \underset{1}{0 \xrightarrow{f} \underline{1}};\ \underset{0}{\underline{1} \xrightarrow{g^1} 1};\ \underset{0}{2 \xrightarrow{f} \underline{1}};\ \underset{1}{\overbrace{3 \xrightarrow{f^1} \underline{4};\ \underline{3} \xrightarrow{g^2} 3}^{\underset{3}{}}};\ \underset{0}{\underline{4} \xrightarrow{g^1} 3};\ \underset{1}{\overbrace{4 \xrightarrow{f^1} \underline{7};\ \underline{6} \xrightarrow{g^2} 5}^{\underset{3}{}}};\ \cdots \right]$$

Notez que la file ne contient jamais plus de 3 valeurs. Nous écrivons de manière compacte la liste d'occurrence en mettant entre parenthèses la première exécution de la période :

$$O_{f\_g} = \left[\underset{0}{\underline{0} \xrightarrow{g} 1};\ \underset{1}{0 \xrightarrow{f} \underline{1}};\ \underset{0}{\underline{1} \xrightarrow{g^1} 1};\ \underset{0}{2 \xrightarrow{f} \underline{1}};\ \left(\underset{3}{3 \xrightarrow{f^1} \underline{4}};\ \underset{1}{\underline{3} \xrightarrow{g^2} 3};\ \underset{0}{\underline{4} \xrightarrow{g^1} 3}; \right) \right]$$

9.3.1.3 Choix des instants et signatures principales

L'ordonnancement ALAP fixe de manière unique l'exécution du réseau. Pour décider d'une signature, nous devons choisir comment les instants découpent cette exécution. La seule contrainte est d'obtenir une signature principale, pour rappel, cela signifie que dans l'instant, ne sont consommées que des entrées nécessaires aux sorties. Cela se transpose en la règle :

9.3.2 *Propriété* **Instants d'une composition principale en boucle simple**
Dans un instant, toutes les activations de `f`, consommant une valeur ou plus, doivent être avant les activations de `g` produisant une valeur ou plus.

9.3.3 *Remarque* **Instants vides**
Les instants ne contenant pas d'activation sont légaux. La signature résultante sera d'horloge nulle en entrée et en sortie et la fonction de transition de la composition n'aura rien à faire.

Signature avec des horloges d'activations booléennes (v1) Nous représentons le choix des instants avec des boîtes :

$$O_{f\_gv1} = \left[\boxed{\underline{0} \xrightarrow{g} 1} ; \boxed{0 \xrightarrow{f} \underline{1};\ \underline{1} \xrightarrow{g^1} 1} ; \boxed{2 \xrightarrow{f} \underline{1}} ; \left(\boxed{3 \xrightarrow{f^1} \underline{4} ;\ \underline{3} \xrightarrow{g^2} 3} ; \boxed{\underline{4} \xrightarrow{g^1} 3} ; \right) \right]$$

Il y a maintenant deux informations distinctes, le pointage de `f_g` qui cache ce qu'il y a dans les boîtes en ne gardant que les occurrences des entrées sorties et l'ordonnancement des activations de `f` et `g`, interne à la fonction de transition de `f_g` :

$$O_{f\_gv1} = \left[0 \xrightarrow{\boxed{g}} 1;\ 0 \xrightarrow{\boxed{fg^1}} 1;\ 2 \xrightarrow{\boxed{f}} 1;\ \left(3 \xrightarrow{\boxed{f^1 g^2}} 3;\ 3 \xrightarrow{\boxed{g^1}} 3; \right) \right]$$

La signature de `f_g` est alors $f\_gv1 : 0.0.2.(1.0)^\omega \to 1.0.0.(2.0)^\omega$ et les activations de `f` et `g` peuvent se résumer à deux horloges d'activations, pour `f` : $0.1.1.(1.0)^\omega$ et pour `g` : $1.1.0.(1.1)^\omega$.

Nous pouvons vérifier la cohérence de ces deux informations en calculant la signature de `f` et de `g` soumises à leurs horloges d'activations :

pour `f` : $0.1.1.(1.0)^\omega \text{ dr } (0.2.1^\omega \to 1.0.3^\omega) = 0.0.2.(1.0)^\omega \to 0.1.0.(3.0)^\omega$

pour `g` : $1.1.0.1^\omega \text{ dr } (0.(1.2)^\omega \to 1.(0.2)^\omega) = 0.1.0.(2.1)^\omega \to 1.0.0.(2.0)^\omega$

L'entrée est bien consommée sur $0.0.2.(1.0)^\omega$ et la sortie produite sur $1.0.0.(2.0)^\omega$.

Signature (v2) Nous le voyons dans la signature v1, il y a beaucoup d'instants qui ne produisent rien et/ou qui ne consomment rien. Pour accélérer cette signature, nous pouvons suivre les critères précédemment donnés (cf. propriété 8.2.29), mais la présentation visuelle des occurrences donne une meilleure intuition.

La première activation de `g` peut être mise avec la première activation de `f`, car cette dernière *ne consomme rien* : $\boxed{\underline{0} \xrightarrow{g} 1;\ 0 \xrightarrow{f} \underline{1}}$, mais comme cette dernière pouvait se mettre avec la deuxième activation de `g`, nous pouvons avoir : $\boxed{\underline{0} \xrightarrow{g} 1;\ 0 \xrightarrow{f} \underline{1};\ \underline{1} \xrightarrow{g^1} 1}$. Dans la période, nous associons les deux activations successives de `g`, pour obtenir la signature :

$$O_{f\_gv2} = \left[\boxed{\underline{0} \xrightarrow{g} 1;\ 0 \xrightarrow{f} \underline{1};\ \underline{1} \xrightarrow{g^1} 1} ; \boxed{2 \xrightarrow{f} \underline{1}} ; \left(\boxed{3 \xrightarrow{f^1} \underline{4};\ \underline{3} \xrightarrow{g^2} 3;\ \underline{4} \xrightarrow{g^1} 3} ; \right) \right]$$

Soit $O_{f\_gv2} = \left[0 \xrightarrow{\boxed{gfg^1}} 1;\ 2 \xrightarrow{\boxed{f}} 1;\ \left(3 \xrightarrow{\boxed{f^1g^2g^1}} 3; \right) \right]$, cette deuxième version de la signature de `f_g` est $0.2.1^\omega \to 1.0.2^\omega$, l'horloge d'activation de `f` est 1^ω, celle de `g` est $2.0.2^\omega$.

$$\begin{aligned} &\text{pour } \texttt{f} : && 1^\omega \text{ dr } (0.2.1^\omega \to 1.0.3^\omega) = 0.2.1^\omega \to 1.0.3^\omega \\ &\text{pour } \texttt{g} : && 2.0.2^\omega \text{ dr } (0.(1.2)^\omega \to 1.(0.2)^\omega) = 1.0.3^\omega \to 1.0.2^\omega \end{aligned}$$

Signature comblée (v3) La signature comblée, nécessite de dérouler une fois la période, pour combiner la deuxième et la troisième activation de `f` :

$$O_{f\_gv3} = \left[0 \xrightarrow{\boxed{gfg^1}} 1;\ 3 \xrightarrow{\boxed{ff^1g^2g^1}} 3;\ \left(4 \xrightarrow{\boxed{f^1g^2g^1}} 5; \right) \right]$$

La signature comblée de `f_g` est $0.3.1^\omega \to 1.2^\omega$, avec pour horloge d'activation de `f` : $1.2.1^\omega$ et pour `g` : 2^ω.

$$\begin{aligned} &\text{pour } \texttt{f} : && 1.2.1^\omega \text{ dr } (0.2.1^\omega \to 1.0.3^\omega) = 0.3.1^\omega \to 1.3.3^\omega \\ &\text{pour } \texttt{g} : && 2^\omega \text{ dr } (0.(1.2)^\omega \to 1.(0.2)^\omega) = 1.3^\omega \to 1.2^\omega \end{aligned}$$

9.3.2 Composition synchrone et taille synchrone d'une file

9.3.2.1 Taille synchrone : tas (état) vs pile

Précédemment, nous avons considéré le nombre maximal de valeurs stockées après chaque activation du producteur `f` ou du consommateur `g`. Si la composition était effectuée à l'aide d'une file logicielle, cette taille serait la mémoire minimale à allouer.

Cependant, si le réseau est compilé en une fonction de transition, les communications dans l'instant ou entre instant sont de natures différentes. Celles dans l'instant peuvent être, en interne, gérées spécifiquement par la fonction de transition. Pour stocker les valeurs d'un instant à l'autre, la fonction de transition de la composition doit stocker les valeurs non consommées dans son état, tandis que dans l'instant, ces valeurs peuvent simplement transiter par des variables temporaires.

Reprenons notre exemple avec le choix des instants de v1, le nombre de valeurs dans la file est indiqué sous le point-virgule :

$$O_{f\_gv1} = \left[\boxed{\underline{0} \xrightarrow{g} 1} \underset{0}{;} \boxed{0 \xrightarrow{f} \underline{1};\ \underline{1} \xrightarrow[1]{g^1} 1} \underset{0}{;} \boxed{2 \xrightarrow{f} \underline{1}} \underset{0}{;} \left(\boxed{3 \xrightarrow[3]{f^1} \underline{4};\ \underline{3} \xrightarrow{g^2} 3} \underset{1}{;} \boxed{\underline{4} \xrightarrow{g^1} 3} \underset{0}{;} \right) \right]$$

Ce choix d'instants internalise presque toutes les communications entre `f` et `g`. Il n'y a qu'à la fin du premier instant de la période qu'une valeur doit être stockée dans l'état. Nous dirons que la taille synchrone de la file utilisée est 1.

9.3.4 *Définition* **TAILLE SYNCHRONE D'UNE FILE**
La taille synchrone d'une file de communication `y' = buffer y`, adaptant $O(\texttt{y})$ en $O(\texttt{y'})$ est donnée par $\text{syn\_size}(O(\texttt{y}),O(\texttt{y'}))$ avec :

$$\text{syn\_size}(w, w') = \sup_k (w[k] - w'[k])$$

Cette définition est étendue pour la gestion des taux nuls en définition 9.3.25.

Dans notre exemple, `y` est la sortie de `f` et `y'` l'entrée de `g`. En appliquant les horloges d'activations à `f` et `g`, nous avions trouvé les horloges de ces deux flots : $ck(\texttt{y}) = 0.1.0.(3.0)^\omega$ et $ck(\texttt{y'}) = 0.1.0.(2.1)^\omega$. Nous retrouvons bien que la taille de cette file de communication est 1 :

$$\text{syn\_size}(0.1.0.(3.0)^\omega, 0.1.0.(2.1)^\omega) = \max\{(0-0);(1-1);(1-1);(4-3);(4-4)\} = 1$$

9.3.2.2 Ordre dans l'instant et sur-approximation de la taille d'une file

La taille synchrone d'une file se calcule directement à partir du compteur d'occurrence de l'entrée de la file et celui de sa sortie. Pour connaître la taille absolue de la file, il faut de plus connaître ce qui se passe dans l'instant.

Avec uniquement les horloges des flots O (`y`) et O (`y'`), nous pouvons donner une approximation de la taille de la file de `y' = buffer y`. Une borne B de cette taille est donnée par la borne supérieure de la différence entre le nombre de valeur produite jusqu'à l'instant k moins les valeurs consommées jusqu'à l'instant $k-1$, que nous relions à la taille synchrone ainsi :

$$\begin{aligned} B &= \sup_k(O(\texttt{y})[k] - O(\texttt{y'})[k-1]) \\ &= \sup_k(ck(\texttt{y})[k] + O(\texttt{y})[k-1] - O(\texttt{y'})[k-1]) \\ &\leq \sup_k ck(\texttt{y})[k] + \text{syn\_size}(O(\texttt{y}),O(\texttt{y'})) \end{aligned}$$

En règle générale, B est une sur-approximation de la taille nécessaire. En prenant `f` de signature $1^\omega \to 1^\omega$, et `g` de signature $1^\omega \to 0.1^\omega$, leur composition accepte la signature $2.1^\omega \to 1^\omega$ avec les instants suivants :

$$\left[\boxed{1 \xrightarrow[1]{f} \underline{1};\ \underline{1} \xrightarrow[0]{g} 0;\ 2 \xrightarrow[1]{f} \underline{2};\ \underline{2} \xrightarrow{g} 1}\ ;_0 \left(\boxed{3 \xrightarrow[1]{f} \underline{3};\ \underline{3} \xrightarrow{g} 2}\ ;_0\right)\right]$$

Dans ce cas, B vaut 2 car au premier instant, $ck(\texttt{y})[1] + O(\texttt{y})[0] - O(\texttt{y'})[0] = 2 + 0 - 0$. Or comme nous l'indiquons sous les points virgules, la file ne porte jamais plus d'une valeur.

Cette sur-approximation provient de la perte de l'information de l'ordonnancement interne à l'instant quand nous ne retenons que les compteurs d'occurrence. Sans information, c'est l'ordonnancement le plus général qui est assumé : la consommation des entrées avant la production des sorties :

$$\left[\boxed{1 \xrightarrow[1]{f} \underline{1};\ 2 \xrightarrow[2]{f} \underline{2};\ \underline{1} \xrightarrow[1]{g} 0;\ \underline{2} \xrightarrow{g} 1}\ ;_0 \left(\boxed{3 \xrightarrow[1]{f} \underline{3};\ \underline{3} \xrightarrow{g} 2}\ ;_0\right)\right]$$

9.3.5 *Propriété* **Ordre dans l'instant**
Dans l'instant, l'ordre des activations de g et f peut être oublié. Si g pouvait s'activer avant f alors il peut aussi s'activer après. En ne retenant que les horloges d'activations, l'ordre implicite est que dans l'instant les activations de g se font après celles de f. La seule conséquence est une possible augmentation de la taille de la file de communication.

9.3.6 *Remarque* **Complexité induite par les signatures non comblées**
Ce mouvement relatif est la conséquence de l'utilisation de signatures qui ne sont pas comblées, ici celle de `g`. En effet, si les signatures sont comblées, le choix des instants ne peut pas entrelacer les appels de `f` et `g` dans un même instant (cf. propriété 9.3.2).

9.3.2.3 Composition synchrone

9.3.7 *Définition* **COMPOSITION SYNCHRONE**
Une composition est synchrone ssi les files de communications utilisées pour la composition sont vides à la fin de tous les instants : sont de taille synchrone nulle.

Une composition n'est synchrone que relativement au choix des instants. Plus les instants regroupent d'activations, plus la taille synchrone de la file de communication a des chances de diminuer. En effet, fusionner deux instants en un seul ne fait que diminuer le nombre de valeurs à prendre en compte pour le calcul de la taille synchrone.

L'ordonnancement $v1$ n'est pas une composition synchrone, mais la signature v2 un peu plus rapide l'est :

$$O_{f\_gv2} = \left[\boxed{\underline{0} \xrightarrow[0]{g} 1 \;;\; 0 \xrightarrow[1]{f} \underline{1};\; \underline{1} \xrightarrow{g^1} 1} \;;_0\; \boxed{2 \xrightarrow[0]{f} \underline{1}} \;;_0\; \left(\boxed{3 \xrightarrow[3]{f^1} \underline{4};\; \underline{3} \xrightarrow[1]{g^2} 3 \;;\; \underline{4} \xrightarrow{g^1} 3} \;;_0 \right) \right]$$

Une simple heuristique gloutonne permet de s'assurer que nous minimisons la taille synchrone : en partant du début, fermer l'instant dès que la file est vide. Il convient de vérifier après coup si ces instants sont corrects (cf. propriété 9.3.2). Si ce n'est pas le cas, il n'y aura pas de composition synchrone possible.

9.3.8 *Propriété* **MINIMISATION DE LA TAILLE SYNCHRONE AVEC UNE SIGNATURE COMBLÉE**
S'il existe des instants tels que la composition est synchrone, alors il est possible de les fusionner afin que la composition reste synchrone et que la signature de la composition soit comblée.

Démonstration Une fois les instants choisis pour avoir une composition synchrone, si la signature n'est pas déjà comblée, cela signifie qu'il est possible de l'accélérer et donc de fusionner deux instants (cf. propriété 8.2.29). Or cette fusion ne peut que diminuer la taille synchrone des files.

9.3.9 *Remarque* **COMPLEXITÉ INDUITE PAR LES SIGNATURES NON COMBLÉES (BIS)**
Nous sommes tentés de penser qu'il suffit de choisir les instants les plus gros possibles pour que la composition soit synchrone si elle le peut. Mais encore une fois l'utilisation de signatures non comblées pour `f` ou `g` n'assure pas cela.

L'ordonnancement $v3$ donne une composition synchrone et une signature comblée. Cependant, l'activation g^1 au premier instant peut être mise dans le deuxième instant (car elle ne produit rien) :

$$O_{f\_gv3} = \left[0 \xrightarrow[1]{\boxed{gf}} 1;\; 3 \xrightarrow[0]{\boxed{g^1 f f^1 g^2 g^1}} 3;\; \left(4 \xrightarrow[0]{\boxed{f^1 g^2 g^1}} 5; \right) \right]$$

La signature résultante est la même (comblée) mais la composition n'est plus synchrone.

9.3.2.4 Composition sans `buffer`

En LUCY-n, comme dans tous les programmes synchrones, la composition synchrone est courante et s'écrit sans file `g(f(x))`. Notre vision axée réseaux de Kahn nous force à considérer la composition comme passant par une file quoi qu'il arrive. Notre approche consiste donc à traiter cette composition comme les autres avec une file implicite et de trouver des instants tels

que cette composition est synchrone. S'il n'en existe pas, alors le compilateur devra rejeter le programme ou bien émettre un avertissement que cette composition ne sera pas effectuée de manière synchrone.

Heureusement, le cas de composition le plus courant est assuré de se faire sans `buffer` :

9.3.10 *Propriété*
Si `f` a une seule sortie et est de granularité maximale de sortie inférieure à 1, alors la composition sans buffer intermédiaire (de code source `g(f(x))`) est toujours possible.

Démonstration Avec une granularité maximale de 1 en sortie, chaque activation de `f` produit au plus une valeur. L'ordonnancement ALAP n'activera donc `f` que pour produire exactement ce qui est nécessaire à l'activation de `g`. Les instants commençant et finissant après chaque activation de `g` sont toujours légaux et dans notre cas définissent une composition synchrone.

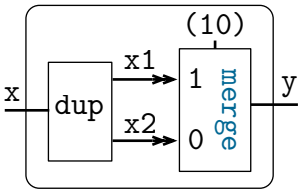
9.3.3 Équilibre et composition de fonctions avec n files

La composition avec plusieurs files de communication ne change pas le recours à l'ordonnancement ALAP. La différence fondamentale tient au fait qu'il n'est plus certain d'obtenir une composition nécessitant des files de taille finie. Nous regardons deux exemples, le premier correct et important dans la compréhension des horloges entières, le second en est une variante incorrecte.

9.3.3.1 Le nœud de bégaiement `stutter`

Le nœud `stutter` bégaye une fois chaque valeur de son entrée. Avec l'entrée $x_1.x_2.\cdots$, la sortie est $x_1.x_1.x_2.x_2.\cdots$. Nous l'écrivons comme la composition de `dup` et de `merge (10)` :

```
let node stutter x = y where
  rec (x1, x2) = dup x
  and y = merge (10) (buffer x1) (buffer x2)
```



La signature utilisée pour `dup` est $d : 1^\omega \to (1^\omega, 1^\omega)$, tandis que celle de `merge (10)` est $m : ((1.0)^\omega, (0.1)^\omega) \to 1^\omega$. L'ordonnancement ALAP est périodique :

$$\left[\overbrace{1 \xrightarrow{d^1} \underbrace{(1,1)}_{(1,1)}}^{\text{;}} \overbrace{\underbrace{(1,0)}_{(1,1)} \xrightarrow{m^1} 1}^{\text{;}}\ \underbrace{(1,1)}_{(0,1)} \xrightarrow{m^2} 2\ \underbrace{;}_{(0,0)}\ \overbrace{2 \xrightarrow{d^1} \underbrace{(2,2)}_{(1,1)}}^{\text{;}} \overbrace{\underbrace{(2,1)}_{(1,1)} \xrightarrow{m^1} 3}^{\text{;}}\ \underbrace{;}_{(0,1)}\ \cdots\right]$$

Avec deux choix d'instants intéressants :

— La composition synchrone avec la signature comblée : $1^\omega \to 2^\omega$

$$\left[\left(\boxed{1 \xrightarrow{d^1} \underbrace{(1,1)}_{(1,1)}\ ;\ \underbrace{(1,0)}_{(1,1)} \xrightarrow{m^1} 1\ \underbrace{;}_{(0,1)}\ \underbrace{(1,1)}_{(0,1)} \xrightarrow{m^2} 2}\ \underbrace{;}_{(0,0)}\right)\right]$$

— Une composition non synchrone mais de signature d'horloges binaires : $(1.0)^\omega \to 1^\omega$

$$\left[\left(\boxed{1 \xrightarrow{d^1} \underbrace{(1,1)}_{(1,1)}\ ;\ \underbrace{(1,0)}_{(1,1)} \xrightarrow{m^1} 1}\ \underbrace{;}_{(0,1)}\ \boxed{\underbrace{(1,1)}_{(0,1)} \xrightarrow{m^2} 2}\ \underbrace{;}_{(0,0)}\right)\right]$$

9.3.11 *Remarque* **STUTTER SANS `buffer`**
Puisqu'il existe une composition synchrone, le code sans `buffer` devrait être accepté :

```
let node stutter x = merge (10) x x
```

La contrainte de synchronisme de la composition force l'utilisation d'horloges entières et la signature comblée $1^\omega \to 2^\omega$. Pour cette raison, le code ci-dessus est actuellement rejeté en LUCY-n.

9.3.3.2 Tension entre la taille des files internes et externes

Pour résumer grossièrement, il est possible de minimiser la taille des files d'une composition si les nœuds utilisés ont des signatures de granularité faible (cf. propriété 9.3.10). Mais pour obtenir des compositions synchrones, il est souvent nécessaire de grossir les instants et donc de produire une signature à forte granularité (cf. `stutter`). L'optimisation de la taille totale des files est donc un problème très complexe et global. Cette optimisation est à comparer avec la volonté d'avoir du code à forte granularité, que ce soit pour distribuer le code ou bien avoir un code généré plus efficace.

9.3.3.3 Problème d'équilibre

Regardons un exemple déséquilibré, variante de `stutter` avec un pointeau différent :

```
let node wrong_eq x = y where
  rec (x1, x2) = dup x
  and y = merge (110) (buffer x1) (buffer x2)
```

Soit $m : ((1.1.0)^\omega, (0.0.1)^\omega) \to 1^\omega$ la signature de `merge (110)`. Cette fois l'ordonnancement ALAP n'est pas périodique, il n'est même pas ultimement périodique car la deuxième file de communication va stocker de plus en plus de valeurs :

$$\Big[\ 1 \xrightarrow[(1,1)]{d^1} (1,1)\ ;\ (1,0) \xrightarrow[(0,1)]{m^1} 1\ ;\ 2 \xrightarrow[(1,2)]{d^1} (2,2)\ ;\ (2,0) \xrightarrow[(0,2)]{m^2} 2\ ;\ (2,1) \xrightarrow[(0,1)]{m^3} 3\ ;$$
$$3 \xrightarrow[(1,2)]{d^1} (3,3)\ ;\ (3,1) \xrightarrow[(0,2)]{m^1} 4\ ;\ 4 \xrightarrow[(1,3)]{d^1} (4,4)\ ;\ (4,1) \xrightarrow[(0,3)]{m^2} 5\ ;\ (4,2) \xrightarrow[(0,2)]{m^3} 6\ ;\ \cdots\ \Big]$$

La première file force l'activation de `dup` deux fois par période de `merge (110)`, tandis que la deuxième file n'en nécessite qu'une. C'est ce déséquilibre qui requiert une file de taille infinie, chose que nous devons refuser. La notion sous-jacente intéressante est le taux de production (resp. de consommation) des fonctions sur chacune des composantes.

9.3.12 *Définition* **TAUX DE CONSOMMATION ET DE PRODUCTION**
Soit la fonction `f` de signature O_f. Son taux de consommation de l'entrée i est la limite du rapport entre le nombre de valeurs consommées et le nombre d'activations. À droite, la version équivalente simplifiée si la signature est ultimement périodique $O_f(e^{(i)}) = \mathbf{u_i}.\mathbf{w_i}^\omega$:

$$\mathrm{tx}_f(e^{(i)}) = \lim_{k\to\infty} \frac{O_f(e^{(i)})[k]}{k} \qquad\qquad \mathrm{tx}_f(e^{(i)}) = \frac{\|\mathbf{w_i}\|}{|\mathbf{w_i}|}$$

Similairement pour le taux de production d'une sortie.

9.3.13 *Définition* **Composition équilibrée**
La composition `(y1,···,yn) = f(···)` avec `(···) = g(y1', ···, yn')` et `yi' = buffer yi` est dite équilibrée ssi, pour toute file de communication, le rapport entre le taux de production et de consommation est le même :

$$\exists K \in \mathbb{Q}, \forall i, \quad \frac{\mathrm{tx}_f(\mathtt{y}i)}{\mathrm{tx}_g(\mathtt{y}i\mathtt{'})} = K$$

Cette définition est étendue pour gérer les taux nuls en définition 9.3.22.

Intuitivement, si les taux sont différents, la composition ne pourra pas se faire à files bornées. C'est le cas pour `wrong_eq`, puisque les taux de `dup` sont de 1 pour chacune des sorties et le taux de la première entrée de `merge (110)` est de $\frac{2}{3}$, alors que pour la seconde il est de $\frac{1}{3}$.

Dans le cas des signatures ultimement périodiques, une composition équilibrée s'effectue à files bornées. Il est important que les signatures soient ultimement périodiques sinon l'analyse en moyenne qu'apporte les taux ne permet pas de conclure.

9.3.14 *Propriété* **Composition équilibrée : un critère de la composition à files bornées**
La composition ALAP de deux nœuds `f` et `g` de signatures ultimement périodiques est à files bornées ssi elle est équilibrée.

Démonstration Tout d'abord, le rapport entre le nombre total d'appels de `f` et de `g` doit tendre vers K, sinon il y aura soit pénurie soit les files qui grossissent à l'infini. La deuxième partie de la preuve consiste à montrer, grâce à l'ultime périodicité des signatures, que les files sont de tailles bornées.

Soit les signatures écrites avec un mot pour la partie transitoire et un autre pour la partie périodique, $f : \cdots \to (u_1.w_1^\omega, \cdots, u_n.w_n^\omega)$ et $g : (u'_1.{w'_1}^\omega, \cdots, u'_n.{w'_n}^\omega) \to \cdots$.

Nous avons $\forall i,\ K = \frac{\|w_i\|\|w'_i\|}{\|w'_i\|\|w_i\|}$ et posons $a = \mathrm{ppcm}(|w_1|, \cdots, |w_n|, |w'_1|, \cdots, |w'_n|)$.

Une fois les périodes transitoires passées, l'activation de `g` q_g fois avec $q_g = a|w'_1|\|w_1\|$, requiert exactement $r_g = q_g\|w'_i\|/|w'_i|$ (qui est un entier grâce à a) valeurs de la file i. Or l'activation de `f` q_f fois avec $q_f = a|w_1|\|w'_1\|$ produit exactement $r_f = q_f\|w_i\|/|w_i|$ valeurs sur la file i. Or $\frac{r_f}{r_g} = \frac{|w_1|\|w'_1\|\|w_i\|}{|w_i|}\frac{|w'_i|}{|w'_1|\|w_1\|\|w'_i\|} = \frac{K}{K} = 1$

Ainsi durant toute période où `g` est activée q_g fois, il n'y a pas besoin d'activer plus de q_f fois `f`, ce que ne fera donc pas l'ordonnancement ALAP. La quantité de valeurs dans les buffers sera la même au début et à la fin d'une telle période. De plus, durant cette période, la quantité de valeurs produites dans les buffers est bornée par r_f, cqfd.

9.3.4 Composition n pour 1 et partielles

L'impossibilité de faire le produit de réseaux ordonnés force l'utilisation de compositions de fonctions plus expressives. L'idée principale présentée en section 7.3.4.3 est d'avoir toujours au moins une des deux parties de la composition ordonnée. Grâce à cela, l'ordonnancement relatif des entrées et des sorties restera unique.

9.3.15 *Définition* **ALAP pour une composition n pour 1**
Soit les n nœuds `fi` dont l'ensemble des sorties sont connectées à l'ensemble des entrées de `g`. Dans l'esprit, la composition formée est `g` $\circ$ (`f1` $\times \cdots \times$ `fn`) bien que l'ordre des connections ne sont pas ainsi figées. L'ordonnancement ALAP de cette composition active `g` tant qu'il y a assez de valeurs dans ses files d'entrées. Quand ce n'est plus le cas, les `fi` sont activés individuellement de manière à ne produire que ce qui est nécessaire à une activation de `g`.

–

9.3.16 *Remarque* La différence avec la composition simple réside dans le fait que l'ordre total déterminé par l'ordonnancement ALAP n'est pas unique, puisqu'il n'y a pas d'ordre déterminé entre l'activation des `f`i nécessaires. Cependant, puisqu'uniquement les `f`i nécessaires sont activées, seules les entrées nécessaires sont consommées avant la production des sorties l'ordre entre ces consommations n'a donc pas d'importance.

Le point critique est que `g` est un goulot d'étranglement, c'est parce qu'il est ordonné que nous connaissons ses besoins et donc quels `f`i activer. Considérons `mw_01`, qui est la composition du produit ($\texttt{when (01)} \times \mathrm{id}$) avec `merge (01)` :

```
let node mw_01 (x1, x2) = y where
  rec y = merge (01) (buffer (x1 when (01))) (buffer x2)
```

Nous prenons les signatures booléennes, $w : 1^\omega \to (0.1)^\omega$ pour `when (01)`, $i : 1^\omega \to 1^\omega$ pour l'identité et $m : ((0.1)^\omega, (1.0)^\omega) \to 1^\omega$ pour `merge (01)`. L'ordonnancement ALAP est :

$$\left[\begin{array}{ccccccccc}
\underset{0}{;} & & \underset{0}{;} & 1\xrightarrow{w^1}\underline{0}\ \underset{0}{;} & 2\xrightarrow{w^2}\underline{1}\ \underset{1}{;} & & \underset{0}{;} & \underset{0}{;} & \underset{0}{;} \\
 & \underline{(0,1)}\xrightarrow{m^1}1 & & & & \underline{(1,1)}\xrightarrow{m^2}2 & & \underline{(1,2)}\xrightarrow{m^1}3 & \\
1\xrightarrow{i^1}\underline{1}\ \underset{1}{;} & & \underset{0}{;} & & \underset{0}{;} & & \underset{0}{;} & 1\xrightarrow{i^1}\underline{1}\ \underset{1}{;} & \underset{0}{;}
\end{array}\right]$$

Nous avons une composition synchrone de signature booléenne $((0.1.1)^\omega, (1.0.0)^\omega) \to (1.0.1)^\omega$:

$$\left[\left(\boxed{1\xrightarrow{i^1}\underline{1}\ \underset{(0,1)}{;}\ \underline{(0,1)}\xrightarrow{m^1}1}\ \underset{(0,0)}{;}\ \boxed{1\xrightarrow{w^1}\underline{0}}\ \underset{(0,0)}{;}\ \boxed{2\xrightarrow{w^2}\underline{1}\ \underset{(1,0)}{;}\ \underline{(1,1)}\xrightarrow{m^2}2}\ \underset{(0,0)}{;}\right)\right]$$

Cette signature s'accélère en $((0.2)^\omega, (1.0)^\omega) \to 1^\omega$, soit en utilisant initialement la signature comblée $2^\omega \to 1^\omega$ pour `when (01)`, soit en choisissant maintenant les instants :

$$\left[\left(\boxed{1\xrightarrow{i^1}\underline{1}\ \underset{(0,1)}{;}\ \underline{(0,1)}\xrightarrow{m^1}1}\ \underset{(0,0)}{;}\ \boxed{1\xrightarrow{w^1}\underline{0}\ \underset{(0,0)}{;}\ 2\xrightarrow{w^2}\underline{1}\ \underset{(1,0)}{;}\ \underline{(1,1)}\xrightarrow{m^2}2}\ \underset{(0,0)}{;}\right)\right]$$

9.3.17 *Nota bene* La signature $(1^\omega, (1.0)^\omega) \to 1^\omega$ que donne actuellement LUCY-n à ce nœud n'est pas principale puisqu'une valeur de sa première entrée est nécessaire avant la première sortie. Ce qui posera problème quand nous voudrons effectuer un point fixe avec ce nœud (cf. section 9.4.1.4).

9.3.18 *Remarque* **COMPOSITION** 1 **POUR** n
La composition ($\texttt{g1} \times \cdots \times \texttt{gn}$) $\circ$ `f` pourrait aussi être considérée. Cependant, ce serait un ordonnancement ASAP qu'il faudrait et la réactivité du résultat n'est pas assurée et demande des vérifications supplémentaires. En effet, il est possible d'avoir `g1` qui produit une infinité de sorties à partir d'une quantité finie d'entrées (signature quasi-comblée), tandis que `g2` nécessite une quantité infinie d'entrées.

9.3.4.1 Composition partielle

La composition partielle est le sous-cas des compositions n pour 1 dans lequel tous les `f`i sont l'identité sauf un (cf. propriété 7.3.29). L'intérêt des compositions partielles est d'ordre pédagogique : les activations des identités ne sont pas intéressantes et gênent la lisibilité de

notre propos comme nous venons de le voir. Nous pouvons les omettre, surtout qu'elles n'ont d'existence que pour des raisons mathématiques et ne seront pas générées.

Reprenons le nœud `mw_01`, mais comme une composition partielle, il n'y a plus qu'une seule file de communication qui nous importe :

```
let node mw_01 (x1, x2) = merge (01) (buffer (x1 when (01))) x2
```

Notez que nous ne soulignons que l'entrée de m concernée par la composition :

$$\left[\overbrace{(\underline{0},1)\xrightarrow[0]{m^1}1;\ 1\xrightarrow[0]{w^1}\underline{0}}\ ;\ 2\xrightarrow[1]{w^2}\underline{1};\ (\underline{1},1)\xrightarrow[0]{m^2}2;\ \overbrace{(\underline{1},2)\xrightarrow[0]{m^1}3;\ 3\xrightarrow[0]{w^1}\underline{1}}\ ;\right]$$

Le seul point un peu délicat est dans le choix des instants. Par exemple, il n'est pas possible de mettre dans le même instant $(\underline{1},1)\xrightarrow{m^2}2$ et $(\underline{1},2)\xrightarrow{m^1}1$ car cette dernière activation *lit* une entrée, tandis que la première écrit une sortie.

9.3.5 Gestion des taux nuls

Une gestion complète des taux nuls est délicate mais très utile car cela permet une programmation homogène de la phase transitoire du réseau et de la phase périodique. C'est une des grandes avancées que propose le *langage* Lucy-n par rapport aux *modèles* que sont les SDF, CSDF, etc. Cependant, cette promesse ne fonctionne pour l'instant que très modestement. L'approche ALAP avec des signatures principales permet de les traiter entièrement.

9.3.5.1 Exemple à une seule file

Le nœud le plus naturel pour examiner la phase transitoire est `my_fby` (cf. exemple 9.2.6). Seule la première valeur du flot d'initialisation est utilisée. Quelle que soit la signature principale, le taux de cette entrée sera nul. Sa signature comblée est $b : (1.0^\omega, 0.1^\omega) \to 1^\omega$.

L'exemple `special_fby`, initialise `my_fby` avec la sortie d'un nœud `filter_even` :

```
let node filter_even x = x when (01)

let node special_fby (x, y) = my_fby(buffer (filter_even(x)), y)
```

Le nœud `filter_even` est générique et peut produire une infinité de valeur (il filtre une valeur sur deux de son entrée). Nous choisissons de lui donner la signature booléenne $f : 1^\omega \to (0.1)^\omega$:

$$\left[\boxed{1\xrightarrow[0]{f^1}\underline{0}}\ ;\ \boxed{2\xrightarrow[1]{f^2}\underline{1};\ (\underline{1},0)\xrightarrow{b}1}\ ;\ \left(\boxed{(\underline{1},1)\xrightarrow[0]{b^1}2}\ ;\right)\right]$$

L'ordonnancement ALAP n'active que deux fois f, les instants choisis lui donnent l'horloge d'activation $1.1.0^\omega$. Le résultat est la signature booléenne $s : (1.1.0^\omega, 0.0.1^\omega) \to 0.1^\omega$, elle indique bien que la première valeur de `x` est filtrée, la seconde initialise `my_fby` puis les autres ne sont pas consommées.

9.3.19 *Nota bene* Nous avons choisi des horloges booléennes pour montrer que ce n'est pas spécifique aux horloges entières. Pourtant, actuellement en Lucy-n, un tel code est rejeté. *L'inférence d'une horloge d'activation de taux nul* avec une communication au travers de `buffer` est une chose délicate. La difficulté de la résolution actuelle de Lucy-n est d'être globale et mettre toutes les horloges à o^ω est solution des contraintes, ce qui requiert de demander un ordonnancement le « plus grand ». Ce qui n'est pas aisé.

Avec notre approche locale et l'ordonnancement ALAP, nous sommes certains de trouver un ordonnancement qui produit au maximum, tout en étant capable d'inférer des horloges d'activations de taux nul comme dans l'exemple `special_fby`. L'ordonnancement ALAP gère tout aussi correctement le cas où c'est la source qui est de taux nulle. Un exemple très simple refusé actuellement par LUCY-n est le suivant :

```
let node plusone x = x+1
let node filteradd x = plusone(buffer (x when 11(0)))
```

Le nœud `plusone` est de signature $p : 1^\omega \to 1^\omega$, nous prenons la signature comblée $w : 1.1.0^\omega \to 1.1.0^\omega$ pour `when 11(0)` :

$$\left[\boxed{1 \xrightarrow[1]{w} \underline{1};\ \underline{2} \xrightarrow{p^1} 1}_{0};\ \boxed{2 \xrightarrow[1]{w} \underline{2};\ \underline{2} \xrightarrow{p^1} 2}_{0};\ \left(\boxed{2 \xrightarrow{w^1} \underline{2}}_{0};\right)\right]$$

Le résultat de l'ordonnancement ALAP est la signature comblée $1.1.0^\omega \to 1.1.0^\omega$. L'horloge d'activation de `plusone` est de taux nul : $1.1.0^\omega$.

9.3.20 *Propriété* **OPTIMISATION DU CODE PÉRIODIQUE ET SIGNATURES NETTOYÉES**
La partie périodique active w, mais ne consomme rien et ne produit rien. Une simple optimisation consiste alors à ne pas ordonnancer inutilement w et définir pour les même signatures l'ordonnancement :

$$\left[\boxed{1 \xrightarrow[1]{w} \underline{1};\ \underline{2} \xrightarrow{p^1} 1}_{0};\ \boxed{2 \xrightarrow[1]{w} \underline{2};\ \underline{2} \xrightarrow{p^1} 2}\left(\boxed{\ }_{0};\right)\right]$$

C'est cette optimisation qui donne sa force aux signatures nettoyées. Le code qui ne produit et ne consomme que transitoirement n'a plus besoin d'exister une fois le système dans la phase périodique.

9.3.5.2 Multiples files mélangeant taux nuls et non nuls de consommation : puits

Considérons l'exemple suivant :

```
let node stutter_first x = y where
  rec (x1, x2) = dup x
  and y = my_fby (x1, x2)
```

Ce nœud a pour sémantique de bégayer la première valeur de son entrée : pour le flots $x_1.x_2.x_3.\cdots$, il produit $x_1.x_1.x_2.x_3.\cdots$. Rappelons les signature $d : 1^\omega \to (1^\omega, 1^\omega)$ pour `dup` et $b : (1.0^\omega, 0.1^\omega) \to 1^\omega$ pour `my_fby`, qui donnent l'ordonnancement ALAP suivant :

$$\left[1 \xrightarrow{d^1} \underline{(1,1)}_{(1,1)};\ \underline{(1,0)} \xrightarrow{b} 1_{(0,1)};\ \underline{(1,1)} \xrightarrow{b^1} 2_{(0,0)};\ 2 \xrightarrow{d^1} \underline{(2,2)}_{(1,1)};\ \underline{(1,2)} \xrightarrow{b^1} 3_{(1,0)};\ 3 \xrightarrow{d^1} \underline{(3,3)}_{(2,1)};\ \cdots\right]$$

Cet ordonnancement n'est, à première vue, pas périodique. Mais à partir du moment où `my_fby` est dans sa phase périodique (à partir du premier appel b^1), plus aucune valeur n'est consommée de la première file pendant qu'elle reçoit une valeur à chaque activation de `dup`. La file peut jeter toutes ces valeurs, puisqu'elle n'a plus de consommateur. Elle y est même contrainte si elle ne veut pas exploser.

Nous dirons qu'elle devient un puits. À partir de ce moment, nous supprimons son état de l'état du système et l'ordonnancement ALAP devient ultimement périodique :

$$\left[1 \xrightarrow{d^1} \underline{(1,1)}_{(1,1)};\ \underline{(1,0)} \xrightarrow{b} 1_{(0,1)};\ \overbrace{\underline{(1,1)} \xrightarrow{b^1} 2_{(\_,0)};\ 2 \xrightarrow{d^1} \underline{(2,2)}}^{}_{(\_,1)};\ \overbrace{\underline{(1,2)} \xrightarrow{b^1} 3_{(\_,0)};\ 3 \xrightarrow{d^1} \underline{(3,3)}}^{}_{(\_,1)};\right]$$

La composition est synchrone avec la signature comblée $1^\omega \to 2.1^\omega$:

$$\left[\boxed{1 \xrightarrow{d^1} \underbrace{(1,1)}_{(1,1)} \;;\; \underbrace{(1,0)}_{(0,1)} \xrightarrow{b} 1 \;;\; \underbrace{(1,1)}_{(\_,0)} \xrightarrow{b^1} 2} \;;\; \left(\boxed{2 \xrightarrow{d^1} \underbrace{(2,2)}_{(\_,1)} \;;\; \underbrace{(1,2)}_{(\_,0)} \xrightarrow{b^1} 3} \;;\;\right)\right]$$

La taille synchrone de la première file est 0. Plus précisément, les seules valeurs la traversant appartiennent au premier instant. À la fin de celui-ci, elle devient un puits et son existence n'est plus nécessaire.

Expliciter les puits Toute personne habituée aux langages comme Lucy-n aurait tendance à dire que pour jeter les valeurs inutiles de `x1` il suffit d'utiliser `when 1(0)` et écrirait :

```
let node stutter_first_when x = y where
  rec (x1, x2) = dup x
  and y = my_fby (x1 when 1(0), x2)
```

Premièrement, nous avions indiqué que `when 1(0)` pouvait avoir la signature nettoyée $1.0^\omega \to 1.0^\omega$ ce qui ne changerait donc rien au problème.

Deuxièmement, même en utilisant la signature principale $ww : 1^\omega \to 1.0^\omega$ pour `when 1(0)`, le calcul d'une signature principale pour la composition partielle `my_fby (x1 when 1(0), x2)` donne obligatoirement une première composante de taux nul puisque ww n'est activée qu'une seule fois par l'ordonnancement ALAP :

$$\left[1 \xrightarrow[1]{ww} \underline{1}; \; (\underline{1},0) \xrightarrow[0]{b} 1; \; \overbrace{(\underline{1},1) \xrightarrow[0]{b^1} 2}; \; \overbrace{(\underline{1},2) \xrightarrow[0]{b^1} 3}; \cdots\right]$$

Comme attendu, la signature $(1.0^\omega, 0.1^\omega) \to 1^\omega$ de `my_fby` est une solution de cet ordonnancement. En effet, la sémantique de Kahn de `my_fby(x1 when 1(0), x2)` est identique à celle de `my_fby(x1,x2)`. L'ajout de `when 0(1)` n'aura donc rien changé, ni à la sémantique, ni au besoin d'un taux nul pour la première entrée.

Cependant, en terme de langage, il est certainement souhaitable de demander au programmeur d'expliciter les endroits où il y a des puits. La solution la moins déroutante serait de vérifier que le puits a pour destination un `when` avec pointeau de taux nul. Le problème de cette solution est qu'elle n'est pas modulaire. Que faire si `my_fby(x1 when 1(0), x2)` est mis dans un nœud et donc compilé séparément ? Des solutions alternatives sont à explorer, warning, annotations, etc.

9.3.21 *Remarque* Le nœud `stutter_first` n'est évidemment pas accepté par Lucy-n actuellement puisqu'il nécessite une file puits. Par contre, écrit sous la forme `stutter_first_when`, il est accepté mais avec une signature non principale $1^\omega \to 1^\omega$, quand une signature booléenne principale serait par exemple $1.0.1^\omega \to 1^\omega$. Cette signature non principale empêche de faire convenablement un point fixe (cf. section 9.4.1.4) et réduit donc l'utilité de ce nœud.

Implémentation des files puits Aussi étrange que cela puisse paraître, l'implémentation de files ayant aussi le rôle de puits est très peu contraignante, même souvent gratuite.

Dans le cadre de la génération d'un code distribué et donc d'une file asynchrone, il est nécessaire, *à l'exécution*, que le consommateur prévienne la file quand il se déconnecte pour ne plus rien consommer. L'enregistrement des consommateurs auprès de la file de communication est une pratique répandue et ne posera pas de difficulté majeure.

Encore plus aisé : dans le cadre de la génération de code séquentiel, les files seront implémentés par des tampons cycliques de taille fixe. L'ordonnancement assurera alors la correction des valeurs lues et écrites sans nécessiter de vérification spéciale de la part du consommateur ou du producteur, chacun parcourant les tampons à son rythme. Il n'y a donc rien à faire de spécial pour que cette file devienne un puits : quoi qu'il arrive l'écrivain continuera à écrire dans le tampon de manière circulaire en écrasant ses précédentes écritures, qu'elles aient été lues ou pas.

9.3.5.3 Taux nuls de consommation et de production

Quand les taux sont nuls, l'équilibre est plus qu'une vérification du rapport de production et de consommation. La nouvelle contrainte est de produire en quantité suffisante ce qui n'est plus assuré. Considérons l'exemple suivant qui doit être rejeté :

```
let node dup_when x = (y1, y2) where
  rec y1 = x when (0)
  and y2 = x
let node stutter_first_wrong x = y where
  rec (x1, x2) = dup_when x
  and y = my_fby (buffer x1, buffer x2)
```

Le nœud `dup_when` prend la signature $dw : 1^\omega \to (0^\omega, 1^\omega)$, l'ordonnancement ALAP de la composition avec `my_fby` fait exploser la deuxième file puisque `my_fby` ne pourra jamais être activé par faute de valeur dans la première file :

$$\left[1 \xrightarrow{dw} \underbrace{(0,1)}_{(0,1)} \;;\; 2 \xrightarrow{dw} \underbrace{(0,2)}_{(0,2)} \;;\; 3 \xrightarrow{dw} \underbrace{(0,3)}_{(0,3)} \;;\; 4 \xrightarrow{dw} \underbrace{(0,4)}_{(0,4)} \;;\; 5 \xrightarrow{dw} \underbrace{(0,5)}_{(0,5)} \;;\; \cdots\right]$$

La condition d'équilibre doit rejeter les compositions dont le producteur a une sortie de taille infinie et une de taille (finie) inférieure à la demande du consommateur. En effet, l'ordonnancement ALAP va activer le producteur une infinité de fois pour tenter d'obtenir les valeurs nécessaires sur la deuxième file, sans pouvoir activer le consommateur : la première file va se remplir à l'infini.

Nous avons maintenant les conditions *nécessaires et suffisantes* pour déterminer si une composition peut se faire avec des files de tailles finies en prenant en compte le comportement de puits.

9.3.5.4 Équilibre avec puits

L'ordonnancement ALAP gère correctement l'ordonnancement des nœuds ayant des horloges de taux nuls. Cependant, le test d'équilibre des communications a besoin d'être adapté. Nous redéfinissons la notion de composition équilibrée (cf. définition 9.3.13) :

9.3.22 *Définition* **COMPOSITION ÉQUILIBRÉE (BIS)**
La composition `(y1,···,yn) = f(···)` de signature f, avec `(···) = g(y1', ···, yn')` de signature g raccordées par `y`i`' = buffer y`i est dite équilibrée ssi :

$$\exists K \in \mathbb{Q}, \forall i, \quad \mathrm{tx}_g(\mathtt{y}i\mathtt{'}) \neq 0 \implies \frac{\mathrm{tx}_f(\mathtt{y}i)}{\mathrm{tx}_g(\mathtt{y}i\mathtt{'})} = K \tag{9.1}$$

$$\forall i_0, \forall i, \quad \mathrm{tx}_f(\mathtt{y}i_0) \neq 0 \implies O_f(\mathtt{y}i)[\infty] \geq O_g(\mathtt{y}i)[\infty] \tag{9.2}$$

L'équation (9.1) : les files avec un taux de consommation non nul doivent toutes avoir le même ratio K, comme pour la définition ne prenant pas en compte les taux nuls. Les files avec un taux de consommation nul ne sont pas prises en comptes, car cela correspond au comportement de puits des files.

Si tous les taux de production sont nuls, alors $K = 0$ et quelque soit l'ordonnancement, la quantité totale de valeurs produites est bornée. Nous pouvons laisser faire l'ordonnancement ALAP. C'est pourquoi la condition équation (9.2) est inactive et gardée par l'existence d'une file avec un producteur de taux non nul.

Finalement, nous devons nous prémunir contre l'exemple de la section 9.3.5.3. S'il n'est pas possible de produire assez de valeurs alors il ne faut pouvoir activer une infinité de fois `f` sans faire exploser les files, ce qui est la contrainte équation (9.2).

9.3.23 *Remarque* Nous avons connecté la $i$ième sortie avec la $i$ième entrée. Gérer un ordre quelconque de connections ne change rien à part complexifier les notations. Les compositions partielles, ainsi que les duplications n'ont pas besoin d'un traitement différent et peuvent être cachées dans les connections. On fera attention, au choix des instants pour les compositions partielles cf. section 9.3.4.1.

9.3.5.5 Taille synchrone et adaptabilité avec des taux nuls

La transformation des files en puits nécessite d'adapter (cf. définition 9.3.4) le calcul de la taille des files. L'instant où la file devient un puits est caractérisé par une horloge de consommation qui reste nulle. Ceci est équivalent à avoir le compteur d'occurrence du consommateur qui atteint sa limite :

9.3.24 *Définition* **Calcul du dernier instant interactif avec** stagne
Pour tout mot w, nous notons $\mathrm{stagne}(w)$ le premier indice à partir duquel le mot est stationnaire

$$\mathrm{stagne}(w) = \inf\{n \mid \forall k \geq n, w[k] = w[n]\}$$

En particulier, si $n = \mathrm{stagne}(O_{\mathbf{D}}(p))$ alors pour tout k supérieur ou égal à n, l'horloge de p est nulle : $\forall k \geq n, ck_{\mathbf{D}}(p)[k] = 0$.

9.3.25 *Définition* **Taille synchrone d'une file (bis)**
La taille synchrone d'une file de communication `y' = buffer y`, adaptant $O(\mathtt{y})$ en $O(\mathtt{y'})$ est donnée par $\mathrm{syn\_size}(O(\mathtt{y}),O(\mathtt{y'}))$ avec :

$$\mathrm{syn\_size}(w, w') = \sup_{k \leq q}(w[k] - w'[k]) \qquad \text{avec} \quad q = \mathrm{stagne}(w')$$

Puisque les buffers servent de puits, il faut aussi retoucher la notion d'adaptabilité (cf. section 9.3.2.2) :

9.3.26 *Définition* **Adaptabilité (bis)**
La communication de `y` à `y'` (`y' = buffer y`) se contente d'une file de taille bornée ssi $O(\mathtt{y})$ est adaptable ($<:$) à $O(\mathtt{y'})$. Nous redéfinissons l'adaptabilité ainsi :

$$w <: w' \qquad \Longleftrightarrow \qquad \exists B, \forall k \leq q, \quad 0 \leq w[k] - w'[k] \leq B \qquad \text{avec} \quad q = \mathrm{stagne}(w')$$

Notez que la définition est inchangée pour les mots de taux non nuls.

Notez que $B = \text{syn\_size}(O(\texttt{y}),O(\texttt{y'}))+\sup_k ck(\texttt{y})[k]$ est toujours une approximation correcte pour la nouvelle définition d'adaptabilité si nous utilisons la nouvelle définition de la taille synchrone.

9.3.5.6 Projection

Grâce à cette gestion des taux nuls, une des choses élémentaires qui semblait compromise est idéalement réalisée :

```
let node id x = y where
  rec (x1, x2) = dup x
  and y = proj1(x1, x2)
```

En section 9.2.8.2, nous avions expliqués que *la seule signature principale* de `proj1` était $(1^\omega, 0^\omega) \to 1^\omega$ qui ne consomme donc pas sa deuxième entrée. Or `dup` produit également sur ses deux sorties. Cette composition, aussi simple soit-elle, nécessite la gestion des taux nuls. Ici, la deuxième file de communication n'a jamais de consommateur, dès le début elle devra fonctionner comme un puits. Ce n'est donc pas `proj1` qui jette les valeurs, mais sa signature force la file à le faire.

Réinsistons sur le fait que si nous avons la composition partielle `proj1(x1, f(x2))`, `f` n'est jamais activée par l'ordonnancement ALAP. N'importe quel nœud `f` sera donc acceptable et accepté sans que ceci ne change la signature : `proj1(x1, f(x2))` est de seule signature principale $(1^\omega, 0^\omega) \to 1^\omega$.

Incidemment, nous avons là un exemple qui montre l'intérêt et la robustesse de notre approche. Regardons ce qu'il se passe si nous tentons d'utiliser `when` pour jeter les valeurs inutiles. Nous donnons aussi les versions isolant l'application de `when (0)` avec `proj1` ou `dup` :

```
let node id2 x = y where
  rec (x1, x2) = dup x
  and x22 = x2 when (0)
  and y = proj1(x1, x22)
```

```
let node partial (x1, x2)
= y where
  rec x22 = x2 when (0)
  and y = proj1(x1, x22)

let node idp x = y where
  rec (x1, x2) = dup x
  and y = partial (x1, x2)
```

```
let node partial2 x
= (x1, x22) where
  rec (x1, x2) = dup x
  and x22 = x2 when (0)

let node idp2 x = y where
  rec (x1, x22) = partial2(x)
  and y = proj1(x1, x22)
```

Le nœud `partial` est de signature principale $(1^\omega, 0^\omega) \to 1^\omega$ et quelle qu'elle soit, elle sera de taux nul sur son deuxième argument. `idp` nécessite des files faisant office de puits.

Par contre, `partial2` a pour signature comblée $1^\omega \to (1^\omega, 0^\omega)$, ainsi `idp2` ne nécessite pas de puits. Par contre dans `partial2`, le besoin de puits existe si nous utilisons la signature nettoyée $0^\omega \to 0^\omega$ pour `when (0)`, mais n'existe pas si nous utilisons $1^\omega \to 0^\omega$.

Concluons ces histoires de puits en remarquant que le code généré avec `when (0)` pour jeter des valeurs est moins efficace que l'utilisation d'un puits sans `when (0)`. En effet, le `when (0)` rajoute un test à chaque instant. Ce test pollue le code du fonctionnement périodique du réseau. Étant toujours faux, il pourrait être optimisé, mais ceci est gratuit avec les puits.

9.4 Point fixe et causalité

Nous avons motivé l'utilisation de signatures principales, en remarquant que quel que soit son contexte d'appel, une signature principale ne restreint pas le comportement du réseau : il calcule ce qui est prévu par la sémantique de Kahn.

C'est particulièrement en rebouclant des sorties sur des entrées que nous avions montré que le réseau de Gonthier n'avait pas de signature principale, le choix d'une signature pour qu'un des point fixe fonctionne prohibait d'autres points fixes.

9.4.1 En Lucy-n

Comme nous l'avons démontré, le point fixe d'un réseau `f` ordonné réactif est un réseau ordonné réactif (cf. propriété 7.3.22), donc le point fixe d'un nœud est un nœud. Regardons le point fixe qui permet de calculer le flot `nat` des entiers naturels. Nous initialisons le flot avec 0 puis effectuons un point fixe avec une addition. Il s'écrit traditionnellement `y = 0 fby (y + 1)`. Nous le décomposons pour bien faire apparaître le nœud utilisé pour le point fixe :

```
let node gen_nat x = my_fby(0, x + 1)

let node nat () = y where
  rec (y, y') = dup (gen_nat (buffer y'))
```

Prenons la signature comblée $gn : 0.1^\omega \to 1^\omega$ de `gen_nat`. La composition avec `dup` ne fait que dupliquer la sortie : $dgn : 0.1^\omega \to (1^\omega, 1^\omega)$.

Comme toute composition, le point fixe s'effectue au travers d'une file. Cependant il n'y pas d'ordonnancement spécifique à effectuer, seul `gen_nat` est à activer, il va être le producteur et le consommateur de la file. Écrivons, comme auparavant, les compteurs d'occurrence de chaque activation et indiquons la quantité de valeurs, stockées dans la file, sous le point virgule. À nouveau, nous cherchons le comportement périodique en repérant la répétition de la configuration du contrôle et de l'état de la file :

$$\left[\underline{0}\xrightarrow{dgn}(\underline{1},1)\overbrace{\underset{1}{;}\ \underline{1}\xrightarrow{dgn^1}(\underline{2},2)\underset{1}{;}\ \underline{2}\xrightarrow{dgn^1}(\underline{3},3)}\underset{1}{;}\ \cdots\right] \qquad \text{soit} \qquad \left[\underline{0}\xrightarrow{dgn}(\underline{1},1)\underset{1}{;}\ \left(\underline{1}\xrightarrow{dgn^1}(\underline{2},2)\underset{1}{;}\ \right)\right]$$

Les contraintes pour le choix des instants sont identiques à la composition : dans un même instant, pas de sortie après une entrée. Ici, il n'y a pas d'entrée, le réseau est une constante, donc le choix des instants est non contraint. Le choix le plus raisonnable et pratique est de produire une valeur par instant, pour obtenir la signature $0^\omega \to 1^\omega$:

$$\left[\boxed{\underline{0}\xrightarrow{gn}(\underline{1},1)}\underset{1}{;}\ \left(\boxed{\underline{1}\xrightarrow{gn^1}(\underline{2},2)}\underset{1}{;}\ \right)\right]$$

|| 9.4.1 *Nota bene* L'ajout explicite de `dup` est inutile à l'analyse. Par la suite, nous l'omettons.

9.4.1.1 Blocages

Besoin de principalité Si nous utilisons des signatures non principales, le point fixe risque de bloquer et sera dit non-causal. Par exemple si à `x + 1` est donné la signature non principale $1^\omega \to 0.1^\omega$, alors `gen_nat` a pour signature [3] $gnw : 0.1^\omega \to 1.0.1^\omega$. La liste d'occurrence du point fixe ne peut contenir que deux appels. Le troisième requiert 2 valeurs quand seulement une a été produite :

$$\left[\underline{0}\xrightarrow{gnw}\underset{1}{\underline{1}};\ \underline{1}\xrightarrow{gnw}\underset{0}{\underline{1}};\ \text{! Blocage ! pour appeler } \underline{2}\xrightarrow{gnw^1}\underline{1}\right]$$

3. au premier instant `my_fby` produit sans entrée, puis au second instant `plusone` en consomme une sans produire, donc `my_fby` n'est pas activée, au troisième instant, tout rentre dans l'ordre.

Similairement, en utilisant la signature $badfby : (1^\omega, 1^\omega) \to 1^\omega$ traditionnelle de `fby`, `gen_nat` a pour signature $gnww : 1^\omega \to 1^\omega$ et nous avons un blocage dès le premier instant :

$$\left[\text{! Blocage ! pour appeler } \underline{1} \xrightarrow{gnww^1} \underline{1}\right]$$

Bien évidemment, il existe aussi des nœuds dont le point fixe bloque même en utilisant une signature principale comme par exemple `y = 1 + (buffer y)` qui bloquera dès le premier instant (point fixe de la signature $1^\omega \to 1^\omega$).

Production finie Par ailleurs, le blocage ne correspond pas au fait d'avoir des horloges de taux nulles. Il est possible d'effectuer un point fixe qui ne bloque pas sans pour autant produire une infinité de valeurs comme le point fixe de `bw11_0` :

```
let node bw11_0 x = y where
  rec y = my_fby(0, x) when 11(0)
```

Sans entrée ce réseau Moore produit la sortie 0. À partir de l'entrée $x_1.x_2.x_3\cdots$, le réseau produit la sortie finie $0.x_1$. Sa signature comblée est $bwc : 0.1.0^\omega \to 1.1.0^\omega$. Son point fixe ne bloque pas :

$$\left[\underline{0} \xrightarrow[1]{bwc} \underline{1};\ \underline{1} \xrightarrow[1]{bwc} \underline{2};\ \underline{1} \xrightarrow[1]{bwc^1} \underline{2};\ \underline{1} \xrightarrow[1]{bwc^1} \underline{2};\ \cdots\right] \quad \text{soit} \quad \left[\boxed{\underline{0} \xrightarrow[1]{bwc} \underline{1}};\ \boxed{\underline{1} \xrightarrow[1]{bwc} \underline{2}};\ \left(\boxed{\underline{1} \xrightarrow[1]{bwc^1} \underline{2}};\ \right)\right]$$

Ce réseau n'a alors plus d'entrée, en considérant la sortie dupliquée, nous avons la signature $0^\omega \to 1.1.0^\omega$, qui correspond bien à sa sémantique (la constante $0.x_1$).

Signature nettoyée Le nœud `bw11_0` a aussi une signature principale qui n'est pas nettoyée $bww : 0.1.1^\omega \to 1.1.0^\omega$. Si nous utilisons cette signature, son point fixe bloque :

$$\left[\underline{0} \xrightarrow[1]{bww} \underline{1};\ \underline{1} \xrightarrow[1]{bww} \underline{2};\ \underline{2} \xrightarrow[0]{bww^1} \underline{2};\ \text{! blocage ! pour appeler } \underline{3} \xrightarrow{bww^1} \underline{2}\right]$$

Cependant, ce blocage n'arrive qu'une fois toutes les valeurs des sorties produites. La sémantique de Kahn du point fixe est bien respectée, comme nous l'avons affirmé pour les signatures principales, qu'elles soient nettoyées ou pas.

Un traitement ad-hoc pourrait être fait, symétrique à la gestion des taux nuls de la composition. En effet, la file de communication n'a plus de producteur à partir du troisième appel (taux nul de production). La signature utilisée étant principale, ce nœud peut bloquer ou abandonner le calcul, nous savons qu'il a produit tout ce qu'il devrait .

Cependant, la causalité n'est pas une chose aisée pour le programmeur et nous souhaitons pouvoir l'aider.

9.4.1.2 Causalité d'un réseau ordonné

La définition de la causalité repose sur la notion de signature principale nettoyée. Tous les nœuds (réseau ordonné réactif) en ont une, puisque les signatures comblées et quasi-comblées le sont. Leur intérêt est de ne consommer que ce qui est strictement nécessaire. S'il y a blocage par manque d'une entrée nous sommes certains que cette entrée *doit* être consommée et permettrait de produire de nouvelles sorties.

9.4.2 *Définition* **Point fixe causal**
Le point fixe d'un réseau de signature principale nettoyée O est dit causal, ssi pour toute sortie s qu'il reboucle sur l'entrée e, il consomme à un instant k moins que ce qu'il a déjà produit :

$$\forall k \geq 1, \quad O(e)[k] \leq O(s)[k-1]$$

Cette définition n'est pas relative à la signature utilisée, comme le suggèrent les exemples que nous avons regardés. Avant de le prouver, nous avons besoin de considérer cette définition sous l'angle des fonctions d'indexation, ce qui nous rapprochera de la définition d'équivalence entre signatures :

9.4.3 *Propriété* **Point fixe causal**
Si la signature du réseau est donnée avec la fonction d'indexation I, la causalité du rebouclage de la sortie s sur l'entrée e est donnée par :

$$\forall i, e \in E, s \in S, \quad I(s)[i] < I(e)[i] \quad \vee \quad I(e)[i] = \infty$$

Démonstration Prouvons que cette propriété est suffisante. Par l'absurde, nous avons k' fini, tel que $O(s)[k'-1] < O(e)[k'] = i'$. Par propriété 8.1.6, $I(e)[i'] = I(e)[O(e)[k']] \leq k'$ donc par notre propriété $I(s)[i'] < I(e)[i']$. Mais $O(s)[k'-1] < i'$ donc par propriété 8.1.6, $I(s)[i'] > k'-1$ et donc $I(s)[i'] \geq I(e)[i']$, absurde.

Prouvons qu'elle est nécessaire. Par contraposition, nous avons i' tel que $I(e)[i'] \leq I(s)[i']$ et $I(e)[i'] < \infty$ et nous voulons prouver qu'il existe k' tel que $O(e)[k'] > O(s)[k'-1]$. Posons $k' = I(e)[i']$ qui est bien fini par hypothèse. La propriété 8.1.6 donne $O(e)[k'-1] < i' \leq O(e)[k']$. Or par hypothèse, il y a deux cas possibles. Soit $k' = I(s)[i']$, mais alors encore par propriété 8.1.6, $O(s)[k'-1] < i' \leq O(s)[k']$ et donc $O(s)[k'-1] < O(e)[k']$. Soit $k' < I(s)[i']$, mais alors $O(s)[k'] < i'$ et par monotonie $O(s)[k'-1] \leq O(s)[k'] < O(e)[k']$.

9.4.4 Théorème **La causalité du point fixe d'un réseau ordonné est intrinsèque**
La causalité du point fixe d'un réseau ordonné ne dépend pas de la signature principale nettoyée utilisée.

Démonstration Prenons une signature I telle que le point fixe est causal et une autre équivalente et nettoyée I'. Par l'absurde, nous aurions une entrée e et une sortie s telles que $\forall i, I(s)[i] < I(e)[i] \wedge I(e)[i] = \infty$, ainsi qu'un indice i' tel que $I'(e)[i'] \leq I'(s)[i']$ avec $k' = I'(e)[i'] < \infty$.

Par équivalence des signatures, $I(e)[i'] \leq I(s)[i']$ donc par hypothèse $I(e)[i'] = \infty$.

Mais I' est nettoyée (cf. définition 8.2.15) donc il existe une sortie s' et un indice j tels que $I'(e)[i'] \leq I'(s')[j] < \infty$. Par équivalence, $I(e)[i'] \leq I(s')[j]$. Par correction des signatures, comme $I'(s')[j] < \infty$ cette sortie est aussi produite par I : $I(s')[j] < \infty$, ainsi $I(e)[i'] < \infty$.

9.4.5 *Remarque* **Signatures principales non nettoyées**
Comme il est possible de nettoyer toute signature principale, étendre la causalité à toutes les signatures est une formalité. On pourra dire que le point fixe d'un nœud est causal ssi il l'est en nettoyant sa signature.

9.4.1.3 Débordement de file et puits

Tout comme pour la composition, nous devons nous restreindre à des files finies pour assurer le triptyque ordonnancement ultimement périodique, implémentation en mémoire bornée et signature ultimement périodique.

9.4.6 *Exemple* Un exemple de débordement est le point fixe du nœud `init_stutter` qui initialise `stutter` (cf. section 9.3.3.1) avec une première valeur 0 :

```
let node init_stutter x = y where
  rec y = my_fby(0, stutter(x))
```

Ce nœud est de signature comblée $is : 0.1^\omega \to 1.2^\omega$, son point fixe ne bloque pas, mais par contre la file n'est pas bornée :

$$\left[\underset{}{\underline{0}} \xrightarrow{is} \underset{1}{\underline{1}};\ \underset{}{\underline{1}} \xrightarrow{is^1} \underset{2}{\underline{3}};\ \underset{}{\underline{2}} \xrightarrow{is^1} \underset{3}{\underline{5}};\ \underset{}{\underline{3}} \xrightarrow{is^1} \underset{4}{\underline{7}};\ \underset{}{\underline{4}} \xrightarrow{is^1} \underset{5}{\underline{8}};\ \cdots\right]$$

En première approximation, nous pouvons dire que le point fixe d'un réseau de signature O nécessite, pour chaque rebouclage de s sur e, une file de taille $\sup_k(O(s)[k] - O(e)[k])$. Mais il faut prendre en compte les taux nuls. La problématique est beaucoup plus simple que pour la composition puisque l'horloge d'activation est la même pour la production et la consommation : pas besoin de regarder l'équilibre des divers taux. Le seul point délicat est de permettre aux files de devenir des puits, une fois sans consommateur (cf. section 9.3.5.4) :

9.4.7 *Propriété* **Taille des files (avec puits)**
Le point fixe d'un réseau de signature O, se suffit d'une file de taille B pour reboucler la sortie s sur l'entrée e avec :

$$\forall k < \text{stagne}(O(e)), \quad O(s)[k] - O(e)[k] \leq B$$

Nous pouvons combiner la causalité et cette condition en une seule propriété :

9.4.8 *Propriété* **Point fixe causal en mémoire bornée (avec puits)**
Le point fixe d'un réseau de signature O nettoyée est causal en mémoire bornée, ssi pour toute sortie s, rebouclée sur l'entrée e :

$$\exists B, \forall 0 \leq k < \text{stagne}(O(e)), \quad O(e)[k+1] \leq O(s)[k] \leq O(e)[k] + B$$

Démonstration La causalité est $\forall 0 \leq k, O(e)[k+1] \leq O(s)[k]$, pour tout k supérieur ou égal à $\text{stagne}(O(e))$, $O(e)[k] = O(e)[k+1]$, tandis que $O(s)$ est croissant. La causalité revient à prouver $\forall 0 \leq k < \text{stagne}(O(e)), O(e)[k+1] \leq O(s)[k]$. L'autre côté est trivial.

9.4.1.4 Causalité dans les horloges et point fixe *partiel* de nœuds Mealy

Une condition nécessaire de la causalité est, pour les entrées participant au point fixe, de ne pas nécessiter de valeur au premier instant ($O(e)[1] = 0$). Mais si le point fixe est partiel, ne rebouclant que certaines des entrées, alors les autres entrées ne sont pas contraintes. Il est donc possible, avec la causalité dans les horloges, d'utiliser des cycles dont tous les nœuds sont Mealy : sans registre initialisé, ni valeur d'initialisation dans les files.

Pas besoin de chercher loin pour trouver un tel cas de figure. Le nœud `my_fby` convient. Rappelons qu'il ne contient pas de registre, il est simplement composé d'un `merge`. Reprenons le code de `gen_nat`, mais au lieu d'énumérer les entiers à partir de 0, commençons avec une valeur donnée en argument :

```
let node gen_nati (i, x) = y where
  rec y = my_fby(i, plusone x)

let node nati i = y where
  rec y = gen_nati(i, buffer y)
```

La signature comblée de `gen_nati` est $gni : (1.0^\omega, 0.1^\omega) \to 1^\omega$, son point fixe avec `y` relié à `x` est exactement comme celui de `gen_nat`, sauf que nous faisons apparaître la première entrée :

$$\left[(1,\underline{0}) \xrightarrow{gni} \underbrace{\underline{1}}_{1}; \overbrace{(1,\underline{1}) \xrightarrow{gni^1} \underline{2}}^{1}; \overbrace{(1,\underline{2}) \xrightarrow{gni^1} \underline{3}}^{1}; \cdots\right] \qquad \text{soit} \qquad \left[(1,\underline{0}) \xrightarrow{gni} \underbrace{\underline{1}}_{1}; \left((1,\underline{1}) \xrightarrow{gni^1} \underbrace{\underline{2}}_{1}; \right)\right]$$

`nati` a alors la signature quasi-comblée $1.0^\omega \to 1^\omega$. Nous retrouvons bien que seule la première valeur de `i` est utilisée pour initialiser le nœud.

9.4.9 *Remarque* Bien évidemment, si le point fixe reboucle toutes les entrées, il ne peut être causal que si le réseau est Moore ou si c'est le réseau inerte de signature $0^\omega \to 0^\omega$.

Fragilité de la causalité actuelle en Lucy-n Soit la composition de `mw_01` et `stutter_first` :

```
let node stmw x = y where
  rec y = stutter_first(mw_01(x, 42))
```

Rappelons que `mw_01` est de signature comblée $((0.2)^\omega, (1.0)^\omega) \to 1^\omega$. Ainsi `mw_01(x, 42)` est de signature comblée $mwc : 0.2^\omega \to 1.2^\omega$. Pour `stutter_first`, la signature comblée est $st : 1^\omega \to 2.1^\omega$.

$$\left[\boxed{0 \xrightarrow{mwc^1} \underbrace{\underline{1}}_{1}; \underbrace{\underline{1}}_{1} \xrightarrow{st} 2}_{0}; \left(\boxed{2 \xrightarrow{mwc^2} \underbrace{\underline{3}}_{2}; \underbrace{\underline{2}}_{2} \xrightarrow{st^1} 3; \underbrace{\underline{3}}_{1} \xrightarrow{st^1} 4}_{0}; \right)\right]$$

Leur composition est synchrone et de signature comblée $0.2^\omega \to 2^\omega$. Ainsi, le point fixe `y = stwm(buffer y)` est bien causal et produit une infinité de valeurs.

Nous avions noté pour `mw_01` (cf. note 9.3.17) et `stutter_first_when` (la version acceptée par Lucy-n de `stutter_first` cf. remarque 9.3.21) que les signatures données par le compilateur actuel ne sont pas principales. Ceci est très problématique car ce point fixe est par défaut rejeté comme non-causal par le compilateur Lucy-n.

Pour `mw_01`, la signature non principale $(1^\omega, (1.0)^\omega) \to 1^\omega$ est donnée à la place d'une signature principale booléenne comme $((0.1.1)^\omega, (1.0.0)^\omega) \to (1.0.1)^\omega$. Pour `stutter_first_when`, la signature non principale $1^\omega \to 1^\omega$ est donnée à la place par exemple de $1.0.1^\omega \to 1^\omega$.

De manière critique, corriger l'une ou l'autre des signatures ne suffit pas [4]. Il est nécessaire d'annoter à la main les deux nœuds avec des signatures principales. Dans ce cas et seulement dans ce cas, Lucy-n les reconnait correctes et accepte le point fixe de `stmw` en le reconnaissant causal.

PS : inliner entièrement le code ne résout pas non plus le problème avec la version actuelle de Lucy-n.

4. En effet, le point fixe est causal grâce aux deux signatures. Il est nécessaire que `mw_01` fournisse une première valeur sans lire sa première entrée (ce qui n'est pas le cas avec la signature inférée par Lucy-n). Mais il est tout aussi important que `stutter_first` n'ait besoin de lire qu'une seule valeur pour en fournir deux (ce qui n'est pas non plus le cas de la signature inférée par Lucy-n), sans quoi le point fixe bloquerait après une itération.

9.4.2 Discussion de la notion de causalité

Nous avons vu qu'une signature principale est nécessaire pour assurer que le point fixe du nœud produise tout ce qu'il peut. Cette production « maximale » est le respect de la sémantique de Kahn. Pour autant, dans le cas général, l'absence de blocage n'est pas nécessaire à la production maximale. Ce point un peu contre-intuitif provient des réseaux avec des sorties de taux nuls : ils produisent des sorties finies quel que soit le contexte d'appel. Pour avoir une notion de causalité solide, nous avons alors besoin de nous restreindre aux signatures nettoyées. Celles-ci nous permettent de savoir si un nœud arrête pour toujours de consommer une entrée.

L'utilisation de signatures nettoyées devrait rendre le lecteur suspicieux. Pourquoi en avons nous besoin ? La principalité n'est-elle pas le respect de la sémantique ? La réponse courte est qu'il n'existe pas de réseau non-causal. Si les signatures utilisées sont principales, les blocages ne sont pas problématiques, ils font même partie de la sémantique.

Pourquoi alors parler de causalité ? Il y a plusieurs visions qui s'affrontent, celle de la sémantique, celle de l'implémentation et finalement la plus importante, celle du programmeur.

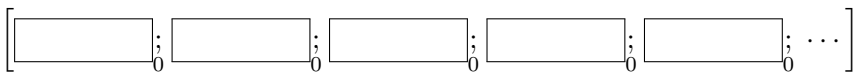
9.4.2.1 Point fixe de sémantique ε

Le point fixe de la duplication est un bon premier exemple, dans notre syntaxe il s'écrit :

```
let node loop_dup () = y where
  rec (y, x) = dup(buffer x)
```

Du point de vue de la sémantique de Kahn, ce réseau est parfaitement défini puisqu'il est composé de fonctions continues, il en est aussi une. Sa sémantique est la constante ε (fonction sans argument qui produit le flot vide). Si nous revenons aux bases, cette sémantique est calculée par itérations de la fonction $F_{ld}((), \mathbf{x}, \mathbf{y}) = ((), \mathbf{x}, \mathbf{x})$ à partir du plus petit élément de son domaine $((), \varepsilon, \varepsilon)$, le résultat est bien $F_{ld}() = \varepsilon$.

Du point de vue de l'implémentation, la question est plus délicate. Le flot vide n'a pas de représentation concrète, tout appel à `loop_dup` est donc du code mort qui peut être supprimé. Dans tous les cas il ne doit pas influencer le reste du code. La génération de code d'heptagon repose entièrement sur la notion d'horloge. En particulier, un flot vide doit avoir une horloge toujours fausse. Le code généré est alors assuré de ne jamais utiliser la variable représentant ce flot et elle peut donc prendre n'importe quelle valeur (cf. section 2.4). L'ordonnancement du point fixe ne s'y trompe pas : comme la signature de `dup` est $d : 1^\omega \to (1^\omega, 1^\omega)$ il ne va jamais pouvoir l'activer et tous les instants seront vides :



La signature résultante pour `loop_dup` est $0^\omega \to 0^\omega$, avec l'horloge d'activation de `dup` égale à 0^ω. Les horloges sont donc correctes et correspondent à la sémantique de Kahn. Ainsi, la question de la causalité, au moins dans notre cadre n'est pas un problème fondamental.

Productivité La causalité est un choix de langage pour le programmeur. En tant que programmeur [5], le point fixe est le seul outil à disposition pour écrire des comportement infinis non triviaux. Son utilisation se résume principalement à un raisonnement par récurrence : nous définissons l'état du système à tout moment en fonction des états précédents qui sont accessibles grâce au point fixe.

5. Cette vision n'est certainement pas partagée par tous, mais un concepteur de langage a la réputation de le concevoir en premier lieu pour lui-même.

Si le compilateur voit des équations récurrentes qui ne produisent rien, il a toutes les raisons de penser que c'est une erreur de la part du programmeur. Le programmeur utiliserait un nœud capable de produire une infinité de valeur (`dup`) pour écrire une équation qui ne produira rien ? Non. Traiter ceci comme une « erreur de causalité » est une aide précieuse pour le programmeur.

Dans la littérature, si une notion de causalité est utilisée, elle est presque toujours informellement équivalente à la productivité (la production d'une infinité de valeurs). Si ce n'est pas le cas, le calcul sera dit non-causal.

9.4.2.2 Point fixe de sémantique finie

Les choses sont beaucoup plus nuancées dans notre approche qui permet au programmeur d'écrire des flots de taille finie. Reprenons l'exemple `bw11_0` (cf. section 9.4.1.1) et la variante :

```
let node bw110_1 x = y where
  rec y = my_fby(0, x) when 110(1)
```

La signature comblée de `bw110_1` est $bww : 0.1.2.1^\omega \to 1^\omega$, l'ordonnancement ALAP de son point fixe bloque :

$$\left[0 \xrightarrow[1]{bww} 1;\ 1 \xrightarrow[1]{bww} 2;\ \text{! blocage ! pour appeler } 3 \xrightarrow{bww} 3\right]$$

Notez que la sémantique du point fixe de `bw110_1` est la constante produisant le flot fini 0.0, exactement comme celle du point fixe de `bw11_0`. Dans ce cas, pourquoi rejetons-nous le point fixe de `bw110_1` en le qualifiant de non-causal tandis que nous acceptons celui de `bw11_0` ?

La réponse est encore dans la tête du programmeur et la compréhension du point fixe en tant que récursion. Quand le programmeur effectue une récursion avec le nœud `bw11_0`, le fait que le résultat est un flot fini est déjà dans ses hypothèses puisque `bw11_0` ne peut quoi qu'il arrive pas produire plus qu'un flot de deux valeurs. Ceci se retrouve dans sa signature $bwc : 0.1.0^\omega \to 1.1.0^\omega$. Elle-même est la conséquence de l'utilisation par le programmeur de `when 11(0)`, qui *explicitement* produit un flot d'au maximum deux valeurs.

Dans le cas de la récursion avec le nœud `bw110_1`, le programmeur manipule un objet capable de produire une infinité de valeurs. L'utilisation de `when 110(1)` dans le corps du nœud était certainement là pour supprimer la troisième valeur du flot. Le fait que cela bloque le point fixe du nœud n'est pas trivial à prévoir et est possiblement une erreur de la part du programmeur : un manque de valeurs d'initialisation.

En dupliquant la première valeur de sortie de `bw110_1`, le programmeur pourra résoudre son erreur : le point fixe de `y = stutter_first(bw110_1 x)` (de signature comblée $0.1.2.1^\omega \to 2.1^\omega$) est causal et produira une infinité de valeurs. S'il souhaite réellement ne produire que deux valeurs en utilisant `bw110_1`, il doit l'expliciter, par exemple, en effectuant le point fixe de `y = (bw110_1 x) when 11(0)`, dont la sémantique et la signature sont celles de `bw11_0`.

9.4.2.3 Une notion générale de causalité dans les réseaux de Kahn

Notre approche et les choix que nous avons faits dans cette section ne sont pas restreints à Lucy-n. Ils nous permettent de proposer une notion de causalité d'un point fixe, dans le cadre général des réseaux de Kahn. Il faut toutefois garder à l'esprit que cette notion n'a pas pour but de prévenir des comportements dont la sémantique est incorrecte ou même dont l'implémentation est difficile.

9.4.10 *Définition* **CAUSALITÉ DU POINT FIXE D'UN RÉSEAU DE KAHN**
Nous dirons que le point fixe du réseau F est causal s'il s'arrête non par manque d'entrées, mais par choix. Mathématiquement, le point fixe est dit causal ssi la suite $(\mathbf{X}_k)_k$ définie par $\mathbf{X}_k = F(\mathbf{X}_{k-1})$ avec $\mathbf{X}_0 = \varepsilon$ est telle que :

$$\forall k, \quad \mathbf{X}_{k+1} = \mathbf{X}_k \quad \Longrightarrow \quad \forall \mathbf{X}_k \sqsubseteq \mathbf{X}, F(\mathbf{X}) = \mathbf{X}_k$$

9.5 Mise en parallèle

Outre la composition et le point fixe, nous devons traiter la mise en parallèle qui est au cœur des langages synchrones. La mise en parallèle est à la fois un gain d'expressivité très important et une source de complexité très grande. Nous avons déjà noté que le produit (mise en parallèle de réseaux ne partageant pas leurs entrées) de deux réseaux ordonnés n'est pas ordonné, cf. propriété 7.3.24. De manière générale, nous ne pouvons traîter sans contexte l'équation `(y1, y2) = (f1(x1), f2(x2))`, le réseau Gonthier en est l'exemple de base. Nous restreignons notre analyse à la mise en parallèle de deux nœuds `f1` et `f2` ayant les même entrées :

```
let node para_f1_f2 (x1, ···, xd) = (y1, ···, yp, y'1, ···, y'p') where
  rec (y1, ···, yp) = f1(buffer x1, ···, buffer xd)
  and (y'1, ···, y'p') = f2(buffer x1, ···, buffer xd)
```

9.5.1 *Nota bene* **DUPLICATIONS IMPLICITES**
Avec une seule entrée, la mise en parallèle est la composition 1 pour 2 de `dup` avec `(f1, f2)`, que nous pourrions traiter avec un ordonnancement ASAP (cf. remarque 9.3.18) .

```
let node para_f1_f2 x = (y1, ···, yp, y'1, ···, y'p') where
rec (x', x'') = dup(x)
and (y1, ···, yp) = f1(x')
and (y'1, ···, y'p') = f2(x'')
```

Cependant, dans le cas général, c'est une composition d pour 2 que nous ne pouvons pas traiter avec l'ordonnancement ALAP ou ASAP.

En propriété 7.3.25, nous avons prouvé que l'ensemble des inverses d'une mise en parallèle est l'union des inverses de chacune des composantes, le résultat étant ordonné ssi cette union est totalement ordonnée. En utilisant une signature quasi-comblée, l'ensemble des inverses est fidèlement respecté par les compteurs d'occurrence des entrées (cf. section 8.2.4) avec l'ordre partiel unanime $\preceq$ (cf. propriété 8.2.25). Nous devons donc vérifier que l'union de l'ensemble des occurrences des signatures quasi-comblées de `f1` et de `f2` est ordonnée.

Finalement, nous avons besoin de prouver la réactivité de la mise en parallèle, ce qui n'est pas assuré.

9.5.1 Réseaux interactifs

Pour simplifier la présentation, nous commençons avec des réseaux interactif (avec une signature comblée), car leur mise en parallèle l'est aussi (cf. propriété 7.3.27).

Nous calculons l'union des ensembles ordonnés avec la fusion gloutonne (du tri fusion). Cet algorithme donne le résultat s'il existe et échoue sinon. Regardons avec différents exemples les cas de figures.

9.5.1.1 Exemple non ordonné : `bimerge`

Le nœud `bimerge` est un des exemples simples où la mise en parallèle n'est pas ordonnée :

```
let bimerge (x1, x2) = (y1, y2) where
  rec y1 = merge (10) (buffer x1) (buffer x2)
  and y2 = merge (01) (buffer x1) (buffer x2)
```

La signature comblée de `merge (10)` est $m10 : ((1.0)^\omega, (0.1)^\omega) \to 1^\omega$, tandis que celle de `merge (01)` est $m01 : ((0.1)^\omega, (1.0)^\omega) \to 1^\omega$. Écrivons $O_{m10} \cup O_{m01}$:

$$\left[(1,0) \xrightarrow{m10^1} 1;\ (1,1) \xrightarrow{m10^2} 2;\ \cdots\right] \cup \left[(0,1) \xrightarrow{m01^1} 1;\ (1,1) \xrightarrow{m01^2} 2;\ \cdots\right]$$

La fusion de ces deux ensembles ordonnés échoue dès le début. En effet, $(1,0)$ n'est pas comparable à $(0,1)$ pour l'ordre unanime. La fusion n'a donc pas de plus petit élément. Conclusion, `bimerge` n'est pas ordonné et n'admet pas de signature principale.

9.5.2 *Remarque* L'absence d'une datation principale pour `bimerge` se comprend comme pour Gonthier. Le rebouclage `(y, y') = bimerge(x, buffer y when (10))` nécessite une signature incompatible avec `(y, y') = bimerge(buffer y' when (10), x)`. Le premier est causal si `bimerge` produit sa première sortie avant de consommer sa deuxième entrée, par exemple avec la signature $((1.0)^\omega, (0.1)^\omega) \to (1^\omega, (0.2)^\omega)$. Tandis que pour le second, c'est l'inverse, et il faudrait la signature $((0.1)^\omega, (1.0)^\omega) \to ((0.2)^\omega, 1^\omega)$.

Actuellement le compilateur Lucy-n donne sans message particulier la signature $((1.0)^\omega, (1.0)^\omega) \to (1^\omega, 1^\omega)$, signature empêchant les deux rebouclages ci-dessus.

9.5.1.2 Exemple ordonné : `addmerge`

Le nœud `addmerge` est la mise en parallèle de `merge (10)` et de l'addition, respectivement de signature comblées $m : ((1.0)^\omega, (0.1)^\omega) \to 1^\omega$ et $a : (1^\omega, 1^\omega) \to 1^\omega$:

```
let node addmerge (x1, x2) = (y1, y2) where
  rec y1 = merge (10) (buffer x1) (buffer x2)
  and y2 = (buffer x1) + (buffer x2)
```

L'union des ensembles d'occurrence $O_m \cup O_a$ est bien ordonné :

$$\left[(1,0) \xrightarrow{m^1} 1;\ (1,1) \xrightarrow{m^2} 2;\ (2,1) \xrightarrow{m^1} 3;\ (2,2) \xrightarrow{m^2} 4;\ \cdots\right] \cup \left[(1,1) \xrightarrow{a^1} 1;\ (2,2) \xrightarrow{a^1} 2;\ \cdots\right]$$

$$= \left[\overbrace{\begin{matrix}(1,0) \xrightarrow{m^1} 1; & (1,1) \xrightarrow{m^2} 2; \\ & (1,1) \xrightarrow{a^1} 1;\end{matrix}}\ \overbrace{\begin{matrix}(2,1) \xrightarrow{m^1} 3; & (2,2) \xrightarrow{m^2} 4; \\ & (2,2) \xrightarrow{a^1} 2;\end{matrix}}\ \begin{matrix}\cdots \\ \cdots\end{matrix}\right] = \left[\left(\begin{matrix}(1,0) \xrightarrow{m^1} 1; & (1,1) \xrightarrow{m^2} 2; \\ & (1,1) \xrightarrow{a^1} 1;\end{matrix}\right)\right]$$

Le comportement périodique est trouvé par reconnaissance de l'enchaînement de l'appel de l'un avec celui de l'autre, ici l'enchaînement $m^1; a^1$. Nous avons la signature de la mise en parallèle $((1.0)^\omega, (0.1)^\omega) \to (1^\omega, (0.1)^\omega)$, avec les compteurs d'occurrence :

$$\left[\left((1,0) \xrightarrow{m^1} (1,0);\ (1,1) \xrightarrow{m^2 \| a^1} (2,1);\right)\right]$$

Remarquez que nous n'avons pas de choix d'instants. Nous avons pris les signatures comblées (donc les plus rapides) pour n'avoir que des inverses dans les ensembles d'occurrence, l'union est elle-aussi uniquement composée d'inverses et décrit donc la signature comblée de la mise en parallèle.

C'est cette dernière remarque qui nous permet de revenir sur la reconnaissance du comportement périodique. Le comportement périodique est normalement reconnu avec l'état du système qui inclut l'état des files. Les files en question relient implicitement la duplication des entrées de `addmerge` à celles de `f1` et `f2`. La signature comblée obtenue consomme les entrées strictement nécessaires pour les activations de `f1` et `f2`. Comme ces activations consomment de manière ordonnée, à la fin de chaque instant les files d'entrées du nœud activé sont vides. Mettons la taille de ces files sous le point virgule :

$$\left[\overbrace{(1,0)\xrightarrow[(0,0)]{m^1}1\ ;\ (1,1)\xrightarrow[(0,0)]{m^2}2\ ;}\ \overbrace{(2,1)\xrightarrow[(0,0)]{m^1}3\ ;\ (2,2)\xrightarrow[(0,0)]{m^2}4\ ;}\ \cdots \right.$$
$$\left. ;\ (1,1)\xrightarrow[(1,0)]{a^1}1\ ;\qquad ;\ (2,2)\xrightarrow[(1,0)]{a^1}2\ ;\ \cdots \right]$$

Trouver la répétition d'un enchaînement revient bien à trouver la répétition d'un état du système tout entier.

9.5.3 *Propriété* **TERMINAISON DE LA FUSION**
Si les ensembles d'occurrence des signatures comblées de `f1` et de `f2` sont infinis (les entrées sont de taux non nuls), alors la fusion échoue ou bien trouve un comportement périodique.

Démonstration Si la fusion n'échoue pas et ne trouve pas de comportement périodique, c'est qu'au moins un des deux nœuds est activé un nombre fini de fois. Considérons que c'est le cas pour `f1`. Prenons x le plus petit tuple d'entiers des compteurs d'occurrence de `f1` qui n'est pas fusionné. La fusion n'échoue pas, donc x est supérieur à une infinité d'éléments (les compteurs d'occurrence de `f2`). Or ceci est impossible, x est un tuple d'entiers, il n'y a qu'un nombre fini de tuples inférieurs.

9.5.4 *Remarque* **ACTUELLEMENT EN LUCY-n**
Le compilateur donne, à `addmerge`, la signature non principale $((1.0)^\omega, (1.0)^\omega) \to (1^\omega, (1.0)^\omega)$. À cause de cette signature, le compilateur rejette comme non-causal le point fixe `(t, y) = addmerge(x, buffer (t when (10)))`, alors qu'il est accepté si nous annotons `addmerge` avec une signature principale.

9.5.5 *Remarque* **UTILISATION DE SIGNATURES PLUS LENTES**
L'utilisation des signatures comblées nous a permis de manière concise de présenter l'algorithme de fusion. Cependant, il est possible de l'adapter pour traiter des signatures principales non comblées. La fusion doit ordonner les inverses, tous les autres appels des fonctions de transition (ne produisant pas de nouvelles sorties) peuvent être intercalés arbitrairement.

9.5.1.3 Gestion des entrées de taux nuls

Encore une fois, les taux nuls font apparaître des cas particuliers à traiter. En sortie, il n'ont pas d'incidence et la preuve de la terminaison de la fusion (cf. propriété 9.5.3) reste valable. En entrée, il n'y a pas de différence à partir du moment où l'on prend en compte les files qui deviennent des puits. Comme pour la composition, une file ne doit plus être comptée dans l'état du système à partir du moment où elle n'a plus de consommateur.

Il n'y a pas besoin d'entrer dans plus de détails et nous nous contenterons d'un exemple minimaliste, la mise en parallèle de `plus_one` et de `filteradd` (cf. section 9.3.5.1). Ces nœuds

sont respectivement de signatures comblées $p : 1^\omega \to 1^\omega$ et $a : 1.1.0^\omega \to 1.1.0^\omega$. La fusion donne :

$$\begin{bmatrix} 1\xrightarrow[0]{p}1;\ 2\xrightarrow[0]{p}2;\ 3\xrightarrow[0]{p}3;\ 3\xrightarrow[0]{p}3;\ \cdots \\ 1\xrightarrow[0]{a}1;\ 2\xrightarrow[0]{a}2;\ \ ;\ \ ;\ \cdots \end{bmatrix}_{\substack{\ \\ 1\ 2}} = \begin{bmatrix} 1\xrightarrow[0]{p}1;\ 2\xrightarrow[0]{p}2;\ \overbrace{3\xrightarrow[0]{p}3;}\ \overbrace{3\xrightarrow[0]{p}3;}\ \cdots \\ 1\xrightarrow[0]{a}1;\ 2\xrightarrow[0]{a}2;\ \ ;\ \ ;\ \cdots \end{bmatrix}_{\substack{\ \\ \_\ \_}}$$

En oubliant la file d'entrée de `filter_add` à partir du moment où il n'est plus consommateur, nous trouvons le comportement périodique :

$$\left[1\xrightarrow{p\|a}(1,1);\ 2\xrightarrow{p\|a}(2,2);\ (3\xrightarrow{p}(3,2);)\right]$$

9.5.6 *Remarque* La fusion de cet exemple ne peut pas échouer car il n'y a qu'un seul argument, donc l'ordre des inverses est total. Ceci est cohérent avec le fait qu'un réseau stable à une seule entrée est ordonné (cf. propriété 7.3.18). Ceci est aussi cohérent avec le fait que cette mise en parallèle est en fait une composition avec `dup` qui est ordonné (cf. note 9.5.1). Toutefois, la simplicité de notre exemple n'a pas d'incidence sur notre propos.

9.5.7 *Propriété* **Terminaison de la fusion**
La fusion, de la mise en parallèle de deux nœuds avec des signatures comblées, échoue ou termine.

Démonstration La preuve suit celle de la propriété 9.5.3. La différence est que si un des taux de sortie est nul, alors les files correspondantes n'entrent plus dans l'état du système.

9.5.2 Signatures quasi-comblées

Si les nœuds ne sont pas interactifs, ils ont une signature quasi-comblée dont les entrées sont de taux nuls et au moins une sortie est de taux non nul (cf. remarque 7.3.16). Nous avons énoncé la condition de réactivité de la mise en parallèle en propriété 7.3.27. En termes simples, il est suffisant et nécessaire de rejeter la mise en parallèle de deux nœuds, ssi il y a un moment où l'un peut produire une infinité sans consommer tandis que l'autre a encore besoin de consommer.

Illustrons en reprenant `nati`, de signature quasi-comblée $1.0^\omega \to 1^\omega$. Nous définissons `natiw(i) = nati(i when 0(1))`, de signature quasi-comblée $2.0^\omega \to 1^\omega$. La mise en parallèle `noreac(i) = (nati(buffer i), natiw(buffer i))` n'a pas de signature principale puisqu'elle n'est pas réactive. En effet, une « signature principale non réactive » serait $1.1^\omega \to (\infty, 0.1^\omega)$, puisqu'avant d'activer `natiw`, il faut activer `nati` une infinité de fois pour produire une infinité de valeurs [6].

Adaptons, au contexte de Lucy-n, la propriété 7.3.27, pour déterminer la réactivité d'une mise en parallèle de deux nœuds.

9.5.8 *Propriété* **Mise en parallèle réactive**
Soient les deux signatures quasi-comblées O et O' de `f` et `f'`. La mise en parallèle de `f` et `f'` est réactive ssi $\forall e \in E, s \in S, \quad O(e)[\infty] < O'(e)[\infty] \implies O(s)[\infty] < \infty$ et symétriquement pour O' et O.

6. Cette nécessité est illustrée par le point fixe `(y,z) = noreac(my_fby(0, buffer y when 0^n(1)))` qui demande $n+1$ valeurs de `y` avant de fournir une deuxième valeur à `noreac`.

9.5.9 *Propriété* **MISE EN PARALLÈLE RÉACTIVE(BIS)**
Les seuls cas où la mise en parallèle de `f` et `f'` est réactive sont :

— soit `f` et `f'` sont interactifs
— sinon, les signatures quasi-comblées respectivement O et O' sont telles que $\forall e \in E, O(e)[\infty] = O'(e)[\infty]$ et au moins une des sorties d'une des signatures est infinie (sinon on est dans le premier cas).

Démonstration Une signature comblée est telle que $O(e)[\infty] < \infty \implies O(s)[\infty] < \infty$ (cf. propriété 8.2.18). Si une des sorties est infinie, la propriété 9.5.8 oblige $\forall e, O(e)[\infty] = O'(e)[\infty]$.

L'exemple le plus simple est la mise en parallèle de `nati` avec lui-même. Notons $n1$ et $n2$ les signatures quasi comblées $1.0^\omega \to 1^\omega$ des deux instances. Une fois ordonnées les activations consommant les inverses, il reste des activations ne consommant rien. Ces dernières activations peuvent être ordonnancées comme bon nous semble (tant que c'est équitable) et regroupées dans des instants quelconques. À gauche une version produisant les sorties une par une au même rythme, à droite une version avec un rythme deux fois plus lents pour la deuxième :

$$\left[1 \xrightarrow{n1\|n2} (1,1); (1 \xrightarrow{n1\|n2} (2,2))\right] \qquad \left[1 \xrightarrow{n1\|n2} (1,1); (\boxed{1 \xrightarrow{n1} (2,1);\ 1 \xrightarrow{n2} (2,2)};\ \boxed{1 \xrightarrow{n2} (2,3)};)\right]$$

Les signatures quasi comblées respectives sont $1.0^\omega \to (1^\omega, 1^\omega)$ et $1.0^\omega \to ((1.0)^\omega, 1^\omega)$. Les deux sont principales, laquelle choisir ? Il n'est en fait pas possible de choisir indépendamment du contexte d'appel si l'on souhaite que les files utilisées dans le contexte soient de tailles bornées. Nous allons donc restreindre à ce que nous pouvons faire totalement modulairement :

9.5.10 *Propriété* **MISE EN PARALLÈLE RÉACTIVE ET PRINCIPALE À MÉMOIRE BORNÉE**
La mise en parallèle des nœuds `f` et `f'` pour être ordonnée réactive sans ambiguité sur le rythme des sorties, requiert que l'un ait une signature comblée O et l'autre une signature quasi-comblée O'. De plus, O finit de consommer avant O' :

$$\forall e \in E, \quad O(e)[\infty] \leq O'(e)[\infty]$$

Un exemple aceptable est donc la mise en parallèle de `filteradd` de signature comblée $f : 1.1.0^\omega \to 1.1.0^\omega$ et `natiw` de signature quasi-comblée $nw : 2.0^\omega \to 1^\omega$. L'ordonnancement de leurs compteurs d'occurrence est :

$$\left[1 \xrightarrow{f} (1,0); 2 \xrightarrow{f\|nw} (2,1); (2 \xrightarrow{nw^1} (2,2);)\right]$$

Notez que f peut ou pas être appelée pendant la période, cela ne change rien, puisque ses horloges sont toutes nulles. Ne pas l'appeler correspond à l'optimisation que nous faisions pour la composition (cf. propriété 9.3.20).

9.6 Programmes LUCY-n

Pour résumer ce que nous avons présenté, l'ordonnancement ALAP correspond à la recherche de signatures principales et traite complètement la composition n pour 1 et le point fixe d'un nœud. La gestion des taux nuls rajoute des cas particuliers, mais rien d'insurmontable en utilisant des files qui deviennent des puits. Finalement, la mise en parallèle n'est pas toujours ordonnée. Dans tous les cas, pour vérifier si elle l'est, il faut avoir la signature de chacune des

composantes. Au niveau du nœud, cette problématique de la mise en parallèle est résolue par l'approche générique suivante.

Nous devons, pour chaque nœud `f` du code source, vérifier qu'il est bien ordonné réactif et lui associer une signature principale, correspondant à la fonction de transition générée. L'approche générique que nous proposons procède en quatre étapes.

— Les composantes `fi` sont définies par

```
let node fi (x1, ···, xd) = yi where
  rec yi = proji(f(x1, ···, xd))
```

— Chacune des composantes est ordonnée puisqu'elle n'a qu'une seule sortie. Nous les analysons donc et obtenons une signature principale pour chacune.

— Nous vérifions que leur mise en parallèle est ordonnée. Nous obtenons alors une signature principale pour `f`.

— Nous reprenons le code source original de `f`, annotons les sorties et effectuons l'ordonnancement et la compilation de `f`. Remarquez que la seule contrainte pour l'ordonnancement est de respecter la signature précédemment définie. Cet ordonnancement peut dans ces limites optimiser l'utilisation mémoire, etc.

Nous devons discuter trois points. Comment annotons-nous les sorties ? Comment passer de nos opérations locales de composition, point fixe, mise en parallèle à l'analyse d'un `fi` en entier ? Finalement, que faire si la mise en parallèle n'est pas ordonnée ?

9.6.1 Annotations

L'annotation des sorties de `f` a pour but de permettre l'ordonnancement ALAP. Elle doit forcer l'ordonnancement à produire les sorties dans un certain ordre. L'idée très simple est d'utiliser un nœud factice consommant les sorties dans l'ordre voulu. Si la mise en parallèle est de signature O_p, que ses sorties sont s^1 à s^r, alors nous créons le nœud `annot` de signature $(O_p(s^1), \cdots, O_p(s^r)) \to (O_p(s^1), \cdots, O_p(s^r))$. Le nœud `f` est inliné dans `f_annot` pour rajouter `annot` sur les sorties avec une composition n pour 1 :

```
let node f_annot (x1, ···, xd) = (y1', ···, yr') where
  rec (y1, ···, yr) = inline f(x1, ···, xd)
  and (y1', ···, yr') = annot(y1, ···, yr)
```

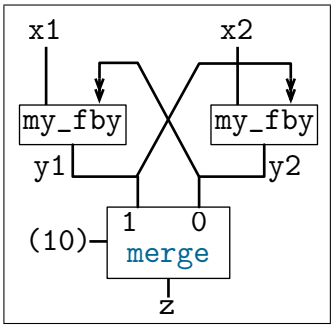
9.6.1 *Remarque* **Annoter les entrées**
Annoter les entrées relève de la même idée mais nécessite la composition 1 pour n (cf. remarque 9.3.18) que nous n'avons pas détaillée puisque nous ne l'utilisons pas.

9.6.2 ALAP global

La possibilité d'ordonnancer localement (une composition, un point fixe, etc) est très puissante et a permis de montrer l'intérêt de notre approche et de discuter de choix locaux. Il serait tentant de conclure en expliquant que l'ordonnancement d'un nœud à une seule sortie peut alors se faire de manière gloutonne. Ceci est effectivement une possibilité pour réduire le nombre de nœuds dans le graphe mais malheureusement en Lucy-n il est possible de faire le point fixe d'un produit, ce qui ne rentre pas dans les constructions que nous avons proposées puisque c'est le point fixe d'un réseau non ordonné.

9.6.2.1 L'exemple de `cross_merge`

Regardons l'exemple minimaliste du point fixe du produit de deux registres en croix :




```
let node cross_merge (x1, x2) = z where
  rec y1 = my_fby(x1, buffer y2)
  and y2 = my_fby(x2, buffer y1)
  and z = merge (10) y1 y2
```

Ce nœud a une unique sortie, il est donc ordonné. La première valeur de la sortie se contente d'une valeur de `y1`, donc d'une valeur de `x1`. La seconde demande une valeur de `y2`, donc une de `x2`. Ensuite, les points fixes ne requièrent plus d'entrées et les `buffer` ont assez de valeurs pour que le point fixe produise une infinité de valeurs. L'ordonnancement ALAP global donne donc une signature quasi-comblée $(1.0^\omega, 0.1.0^\omega) \to 1^\omega$.

Si nous avions essayé de calculer de manière locale, nous aurions dû essayer de donne une signature à `fby_cross`, le point fixe du produit :

```
let node fby_cross (x1, x2) = (y1, y2) where
  rec y1 = my_fby(x1, buffer y2)
  and y2 = my_fby(x2, buffer y1)
```

La projection `fby_cross1` calcule `y1 = my_fby(x1, buffer my_fby(x2, buffer y1))`. Elle est donc le point fixe partiel de `my_fby(x1, buffer my_fby(x2, buffer x3))` qui est de signature comblée $(1.0^\omega, 0.1.0^\omega, 0.0.1^\omega) \to 1^\omega$. Le point fixe est bien causal avec une file de communication de deux places et `fby_cross1` a pour signature comblée $(1.0^\omega, 0.1.0^\omega) \to 1^\omega$.

Symétriquement, l'autre projection est de signature comblée $(0.1.0^\omega, 1.0^\omega) \to 1^\omega$. Leur mise en parallèle n'est pas ordonnée puisque leurs premiers inverses ne sont pas comparables (de taille $(1, 0)$ et $(0, 1)$).

Il en résulte que `fby_cross` n'a pas de signature principale. En l'occurrence, il en a deux plus petites incomparables, $(1.0^\omega, 0.1.0^\omega) \to (1^\omega, 0.1^\omega)$ et $(0.1.0^\omega, 1.0^\omega) \to (0.1^\omega, 1^\omega)$. C'est la première qui est utile pour `cross_merge` mais ce choix ne peut être fait localement.

9.6.2 *Remarque* Il existe des points fixes de produits qui sont ordonnés, contrairement à `fby_cross`. Un exemple est donné en section 9.6.4 avec `share_cross`.

9.6.2.2 Remarques sur l'algorithme ALAP

Il est important que l'ordonnancement ALAP soit appliqué sur un graphe ayant *un unique dernier* nœud. S'il y en avait plusieurs, l'ordonnancement ALAP ne peut pas savoir quelle sortie est nécessaire en premier. Traditionnellement, cet algorithme ordonnance tous les derniers nœuds ensembles ce qui reviendrait par exemple pour `fby_cross`, à demander une valeur pour `y1` et `y2` ensembles ce qui conduirait à une signature non principale. C'est pourquoi nous proposons de traiter un programme LUCY-n en analysant au préalable chacune des composantes du nœud, pour vérifier que leur mise en parallèle est ordonnée et ainsi finalement pouvoir ordonner globalement le nœud, grâce à l'ajout en dernière position d'un nœud d'annotations.

L'algorithme ALAP en lui-même n'a rien de bien nouveau. Par contre, il n'est pas très courant d'accepter d'avoir plusieurs files entre deux nœuds, les conditions d'équilibre que nous avons données définition 9.3.22 entre ces files parallèles doivent être vérifiées pour assurer que l'ordonnancement finisse. Les cycles doivent être traités comme il est de coutume dans les CSDF, en vérifiant que le rapport des taux de production et de consommation est égal à 1 pour assurer des files de tailles finies [EBL94]. L'existence de taux nuls ajoute le cas acceptable

d'un rapport nul (ou infini si le 0 est en consommation et donc faisant appel à des files puits). La causalité sera décrétée si l'algorithme est causal : pour fournir de nouvelles entrées à un nœud, il ne nécessite pas de nouvelles sorties de ce nœud. Cette notion de nécessité comme nous l'avons vu requiert l'utilisation de signatures nettoyées. Toutes ces conditions sont la généralisation de la propriété 9.4.8[7].

9.6.3 Si le nœud écrit par le programmeur n'est pas ordonné

Nous avons vu que tout réseau peut s'exprimer comme la mise en parallèle de nœuds avec une seule sortie (ses composantes). Si la mise en parallèle n'est pas ordonnée, il est tout de même possible de générer le code pour chacune des composantes indépendamment. Cette solution est brutale et elle force à dupliquer tout code initialement partagé entre les composantes.

La découpe optimale de programmes synchrones en sous-programmes modulaires est une question complexe et dépendante du langage concerné. De nombreux travaux s'y sont intéressés et ce dès le début de Lustre [Ray88]. Parmi les travaux récents, nous pouvons citer [LST09 ; PR09] qui montrent tous deux que le problème est NP-complet bien qu'avec de bonnes heuristiques, il soit de complexité faible en pratique [PR09]. Notez qu'ils ne considèrent en fait qu'un sous-problème du nôtre, puisqu'ils ne considèrent pas les horloges et toutes les communications sont synchrones avec une notion de causalité restreinte au strict minimum : un `fby` dans chaque cycle. En particulier, s'il y a deux `fby` de suite, cela aura exactement la même influence qu'un unique `fby`, ce qui n'est pas le cas dans notre contexte puisque : `y = my_fby(7, my_fby(7, buffer y when 0(1)))` est causal tandis que `y = my_fby(7, buffer y when 0(1))` ne l'est pas.

Trouver un algorithme, de complexité acceptable en pratique, en présence d'horloges ultimement périodiques n'est pas sans espoir. Cependant, nous pensons que le programmeur a une idée de ce qu'il souhaite. Le prévenir quand le nœud qu'il a écrit n'a pas de signature principale est déjà une grande avancée. Dans notre contexte, il peut alors corriger ou forcer le compilateur avec des annotations d'horloges.

9.6.4 Un dernier exemple, le partage de ressources

Prenons les nœuds `f` et `g` combinatoires, de signature $1^\omega \to 1^\omega$. Le but est d'avoir `z` égal à `g(f(x1))` aux instants impairs et à `f(g(x2))` aux instants pairs. Voici le code Lucy-n que nous allons considérer `sharefg` (son réseau en figure 9.1) et son code :

```
let node sharefg (x1, x2) = (z) where
  rec y1 = f(merge (01) (buffer y2 when (01)) x1)
  and y2 = g(merge (10) (buffer y1 when (10)) x2)
  and z = merge (01) (y1 when (01)) (y2 when (10))
```

Nous nous attendons à trouver `sharefg` de signature comblée $((1.0)^\omega, (0.1)^\omega) \to 1^\omega$.

Pour simplifier, nous utilisons `mw_01 (x, y)` à la place de `merge (01) (x when (01)) y` (cf. section 9.3.4.1). Ainsi, nous pouvons réécrire `y1 = f(mw_01(buffer y2, x1))` et affirmer que cette équation est de signature comblée $((0.2)^\omega, (1.0)^\omega) \to 1^\omega$. Symétriquement, nous avons `y2 = g(mw_10(buffer y1, x2))` de signature comblée $(1.(0.2)^\omega, (0.1)^\omega) \to 1^\omega$.

Intéressons-nous uniquement au point fixe du produit calculant `(y1, y2)` :

```
let node share_cross (x1, x2) = (y1, y2) where
  rec y1 = mw_01(buffer y2, x1)
  and y2 = mw_10(buffer y1, x2)
```

7. Preuve non incluse.

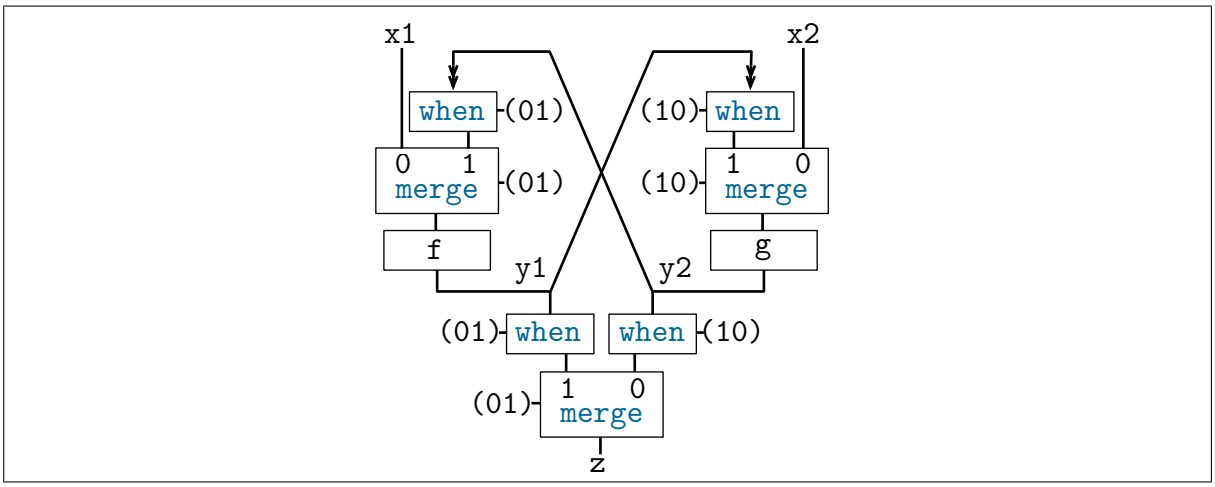


FIGURE 9.1: Réseau de `sharefg`

Ce nœud est de signature comblée $((1.0)^\omega, (0.1)^\omega) \to (1^\omega, 1^\omega)$. Pour nous rendre compte nous appliquons notre algorithme en calculant une signature pour chacune des composantes et vérifions la mise en parallèle. La première projection se réduit à :

```
let node share_cross1 (x1, x2) = y1 where
  rec y1 = mw_01(buffer mw_10(buffer y1, x2), x1)
```

Le cœur est la composition `mw_01(mw_10(t1, t2), t3)` qui est de signature comblée : $(0.1.(0.2)^\omega, (0.1)^\omega, (1.0)^\omega) \to 1^\omega$. Le point fixe qui reboucle la sortie sur la première entrée est bien causal. Nous obtenons alors la signature comblée $((1.0)^\omega, (0.1)^\omega) \to 1^\omega$ pour `share_cross1`.

Le calcul se fait symétriquement pour la deuxième composante et nous obtenons la *même* signature comblée. Leur mise en parallèle est donc ordonnée et `share_cross` est un nœud de signature comblée $((1.0)^\omega, (0.1)^\omega) \to (1^\omega, 1^\omega)$.

La deuxième partie de `sharefg` est `merge (01) (y1 when (01)) (y2 when (10))`, de signature comblée $(1^\omega, 1^\omega) \to 1^\omega$. La composition avec `share_cross` est immédiate et nous avons bien `sharefg` de signature $((1.0)^\omega, (0.1)^\omega) \to 1^\omega$.

Historique et discussion Le partage optimal de ressources, dans les circuits synchrones, nécessite des boucles qui ne traversent pas de registre [Kau70]. Un exemple clair est dû à [Sto92]. Il est repris dans [Mal93] comme motif de base de partage de ressources. Ce motif a pour but de calculer `z = if (c) then f(g(x)) else g(f(x))` avec *une seule* instance de `f` et *une seule* de `g`. Une analyse basée sur une logique tri-valuée permet de distinguer les circuits synchrones causaux des autres. C'est cette approche qui est à l'origine de la causalité constructive en ESTEREL [SBT96 ; Ber99]. Cette causalité est très puissante puisqu'elle propage l'information de présence et d'absence d'une valeur quand nous nous contentons de la présence. Mais pour ce faire, la vision synchrone (circuit synchrone : un fil porte une valeur par instant) est nécessaire, sans quoi l'absence d'une valeur n'a pas de sens.

En LUCY-n, le contrôle est ultimement périodique. Le motif de partage de ressources que nous avons présenté est statique. Grâce à cela, le pointage peut entièrement analyser le flot de données et accepter ce réseau. Nous n'avons pas explicité l'ordonnancement interne généré par ALAP mais il est à noter qu'il change suivant la parité de l'instant.

9.7 Conclusions

En Lucy-n, le contrôle est entièrement connu à la compilation. L'utilisation d'horloges entières rapproche alors un programme Lucy-n des SDF et de leur contre-partie périodique les CSDF [PPL95]. Cependant, Lucy-n est un langage de programmation, à ce titre la compilation séparée et la programmation de la phase d'initialisation sont très importantes alors qu'elles sont délaissés dans les modèles comme les CSDF. Ces deux points correspondent respectivement à la recherche de signatures principales et à la gestion des taux nuls.

La gestion des taux nuls rajoute des cas particuliers, mais rien d'insurmontable à partir du moment où l'on génère du code avec des files pouvant faire office de puits : détruisant les valeurs reçues si elle n'a plus de consommateur. Dans les KRG [Bou+08 ; Coa10], une mise en forme normale du graphe permet d'isoler les comportements de taux nuls, de telle sorte que l'ordonnancement se fait ensuite sur un graphe dénué de taux nuls. Malheureusement, l'élégance de cette proposition empêche la modularité, rajoute du contrôle et conduit à dupliquer le code qui est utilisé dans les deux phases.

Nous avons montré comment l'ordonnancement ALAP et le choix des instants permettent de trouver une signature principale à un réseau ayant une seule sortie. Bien que l'ordonnancement ALAP consomme le plus tard possible ses entrées, il n'assure pas une minimisation de la taille des files de communication. Nous allons donc à contre-courant de la majorité des travaux autour des SDF et CSDF dont la finalité est la minimisation des files et la maximisation du débit [SGB07 ; Ben+10 ; BTV12]. Il faut noter que [BTV12] a l'avantage de proposer un algorithme polynomial en restreignant le type d'ordonnancement recherché. La recherche d'un ordonnancement ALAP est une simulation du réseau et donc de complexité exponentielle.

Toutefois, sauf point fixe complexe, la recherche de signature principale peut s'effectuer localement et de proche en proche. Cette approche locale nous semble aussi très intéressante pour être en mesure de fournir des messages d'erreurs informatifs et compréhensibles. Essayer de diminuer la taille des graphes à ordonnancer en regroupant par cluster a déjà été abordé dans le cadre des SDF [Hsu+07] et est une technique courante en pratique. Les signatures principales fournissent un cadre beaucoup plus général que ce qui est usuellement utilisé, mais la comparaison est peu fondée, les uns essayant de maximiser le débit quand les signatures principales « maximisent la réutilisation du code ».

Nous avons rappelé que dans le cadre d'une compilation vers une fonction de transition, il est important de distinguer les valeurs qui seront stockées sur le tas et sur la pile. La taille synchrone ne considère que les premières. Le choix des instants rentre alors en compte et l'utilisation d'horloges entières apporte de nouvelles marges de manœuvres.

Pour finir, nous avons vu que les signatures principales (nettoyées) fournissent une notion de causalité des points fixes. Pour faire court, nous aimons à dire que nous avons mis la causalité dans les horloges. Cette approche est beaucoup plus fine que les travaux ne considérant que les connections comme dans [ZL06] ou bien qu'une vision synchrone ne regardant que l'existence de registre sur les cycles [CP01]. Notez que ce dernier article traite Lucid Synchrone et donc l'ordre supérieur, chose actuellement hors de portée de nos analyses.

Chapitre 10

Autres pointages

10.1 Heptagon

Nous avons déjà présenté le pointage d'Heptagon en section 2.3. Pour résumer, ce pointage est en boucle simple avec des horloges booléennes. Sa grande simplicité vient de la non interprétation des pointeaux et de la structure en arbre des horloges (cf. section 2.3.2.1). Les nœuds de l'arbre ne font intervenir que des pointeaux strictement au-dessus d'eux dans l'arbre. Pour cette raison, tout nœud est séquentiel et en plus Mealy car il y a toujours une entrée qui est sur l'horloge de base.

Toutes ces restrictions permettent une analyse et une génération de code légères et bien maîtrisées comme nous les avons présentées en chapitre 2. Avec notre recul, nous allons discuter de ces restrictions et, en particulier, de la difficulté d'avoir un pointage principal et même simplement la possibilité de faire des points fixes.

10.1.1 Discussion concernant la causalité

La causalité en Heptagon est déléguée à une analyse spécialisée qui ne prend pas en compte le pointage (cf. section suivante). Qu'aurions nous pu faire avec le pointage d'Heptagon ?

Dans notre discussion sur la causalité de Lucy-n, nous indiquons (cf. section 9.4.1.4) que la restriction aux nœuds Mealy, oblige l'utilisation de points fixes partiels avec des nœuds non stricts. Intuitivement, il faut reboucler des sorties sur des entrées qui ne sont pas tout le temps nécessaires, ce au moins au premier instant. En Lucy-n, `my_fby` est le prototype de nœud Mealy non strict que l'on souhaite utiliser pour un point fixe car il requiert sa deuxième entrée uniquement à partir du deuxième instant. En Heptagon, les pointeaux sont des flots arbitraires qui dépendent possiblement des entrées du programme. Sans aucune information, il est impossible d'assurer qu'une des entrées n'est pas nécessaire au premier instant donc, encore moins, d'assurer qu'un point fixe est causal.

Quoi qu'il arrive, il faut enfreindre une de ces deux hypothèses : il n'y a pas de nœud Moore, les pointeaux ne sont pas interprétés. En interprétant le pointeau `c` en un mot booléen c, la signature comblée de `merge c` est $(c, \neg c) \to 1^\omega$. La causalité (cf. propriété 9.4.8) du point fixe `y = merge c (buffer y) 0` requiert $\forall k \geq 0, \mathcal{O}_c(k+1) \leq \mathcal{O}_{1^\omega}(k) = k \leq \mathcal{O}_c(k) + B$ avec B la taille de la file. Ceci revient à dire que c vaut 0 au premier instant et est égal à 0 moins de B fois. L'idée de n'avoir qu'une information partielle sur les pointeaux est au cœur de la notion d'enveloppe proposée initialement dans la deuxième partie de la thèse sur Lucy-n de Florence Plateau [Pla10].

Il y a de nombreux compromis possibles pour interpréter partiellement les pointeaux et accepter certains points fixes. Une des pistes est l'utilisation d'expressions spéciales pour les pointeaux. Nous pourrions rajouter `fby` dans les pointeaux, alors `merge (false fby c)` est de signature comblée $(\text{false}.c, \text{true}.(\neg c)) \to 1^\omega$, ce qui est suffisant pour traiter le premier instant convenablement. Le point fixe `y = merge (false fby c) (buffer y when (true fby c)) 0` devrait être accepté avec une file de communication de taille 1. Notez que l'horloge $\text{false}.c$ est égale à 0.1^ω `on` c (resp. $\text{true}.(\neg c)$ et 1.0^ω `on` $\neg c$). Cette approche est en fait un hybride entre les pointeaux ultimement périodiques de Lucy-n et ceux non interprétés.

10.1.2 La causalité en Heptagon

Au lieu de vouloir interpréter les pointeaux pour vérifier la bonne utilisation de `buffer`, nous pouvons introduire des constructions Moore pour tenter de rester dans un monde où tout est synchrone. C'est en fait l'approche d'Heptagon.

10.1.2.1 Point fixe synchrone et `fby`

Pour effectuer un point fixe, il est nécessaire de stocker les sorties pour les fournir en entrée à l'instant d'après. Mais en Heptagon, tout est synchrone, ce qui force toutes les files de communications à être vides à la fin de l'instant.

Cette apparente contradiction a été résolue en restreignant les points fixes à des cycles contenant un registre synchrone. Ceci explique le traitement très spécial de `fby` à la compilation (cf. section 2.5.3). Le code séquentiel généré à partir de `y = f(0 fby y)` est en trois morceaux. Le premier morceau est spécial, c'est du code exécuté avant le programme, pour initialiser statiquement une adresse mémoire $memId$ avec 0. Dans le corps de la fonction de transition, la lecture et l'écriture dans le registre sont séparées et ordonnées de telle sorte à être respectivement au début et à la fin du code du cycle (ici l'appel à `f`). Dans l'ordre cela donne `x = fby_read(`$memId$`); y = f(x); fby_write(`$memId$`, y)`.

> 10.1.1 *Nota bene* **Le registre synchrone : une file de communication spéciale**
> Par son découpage en une partie pour lire la valeur stockée et une pour écrire, le registre joue le rôle de file de communication. Par construction, à chaque instant, soit il y a lecture *et* écriture du registre, soit il est intouché. Ainsi le registre remplace une file de communication à une place initialisée qui est toujours pleine entre les instants. Son comportement est celui d'un nœud Moore mais dont l'entrée et la sortie sont sur la même horloge : c'est un nœud en boucle simple inversée avec les sorties avant les entrées.

Cette solution pour `fby` est habile mais délicate car il y a un état partagé entre `fby_read` et `fby_write`, matérialisé par l'adresse mémoire $memId$.

10.1.2.2 Nœuds Moore et découpage

Avec la gestion ad-hoc des registres, un nœud, tel que `sum_synchrone` ci-dessous, devrait être capable de fournir dans le même instant une valeur en sortie avant d'obtenir son entrée.

```
node sum_synchrone (x : int) = (y : int)
let y = 0 fby (x + y) tel
```

Toutefois, ceci est contraire à la compilation et au pointage en boucle simple que nous avons présentés qui forcent `sum_synchrone` à être Mealy. C'est pour cette raison que Lustre permet de mettre à plat le code avant l'analyse de causalité.

Dans [CP01], Pouzet propose une approximation modulaire de la causalité en Lucid Synchrone qui repose sur un typage avec des rangées et traite l'odre supérieur. L'approche repose sur la séparation de la fonction de transition en deux, une première qui fournit les sorties et une autre qui change l'état. Ce qui permet de traiter `sum_synchrone` et d'accepter le point fixe `y = sum_synchrone(y)`. Cependant, la génération séparée de code n'est pas toujours possible sans avoir un langage paresseux car l'ordre d'évaluation n'est pas connu indépendamment du contexte.

Une solution, proposée par Pascal Raymond [Ray88] pendant les débuts de Lustre, est la découpe des nœuds en sous-nœuds en boucle simple, fournis avec un code de glue en Lustre qui est à inliner par l'appelant. L'avantage est de laisser l'ordonnancement du code glue à l'appelant, en exposant au minimum [1] l'intérieur du nœud. Pour `sum_synchrone`, nous obtiendrions deux morceaux :

```
let node sum_synchrone_read (s) = y where
  rec y = fby_read(s.memId)

let node sum_synchrone_write (s, x, y) = () where
  rec fby_write(s.memId, x + y)

let node sum_synchrone_glue x = y where
  rec y = sum_synchrone_read (s)
  and () = sum_synchrone_write(s, x, y)
```

Nous explicitons le partage de l'état s en l'ajoutant comme argument. Le premier morceau `sum_synchrone_read` ne prend que l'état et renvoie la sortie, tandis que le deuxième met à jour l'état. Le code à inliner `sum_synchrone_glue` permet de transférer les calculs effectués dans un morceau et nécessaires dans un autre, comme ici `y`.

À l'origine, la proposition de P. Raymond ne partageait pas d'état entre les morceaux comme nous le faisons ici. En effet, cela est délicat et n'est pas source à source puisque s n'est pas un flot. Pour contourner cela, il devait sortir les registres utilisés par plusieurs morceaux. Dans notre cas, il n'y aurait en fait qu'un seul morceau et le registre serait dans le code de glue.

Sortir l'état, comme nous l'avons fait, permet de traiter le registre comme n'importe quel nœud découpé en deux morceaux. La question qui reste est celle du partage de l'état en source à source. Une réponse que nous étudions dans la section suivante est l'utilisation d'objets avec politiques.

10.2 Pointage avec politiques

Les « objets » avec leurs politiques synchrones étaient en court de développement pour Lucid Synchrone V4 au début de ma thèse, que nous présentons dans [Cas+09]. La motivation de l'introduction de ce concept dans un langage synchrone est de partager un état entre différents sous-systèmes, tout en assurant la cohérence de l'ensemble. Ce besoin se faisait sentir pour la compilation séparée, comme nous venons de le souligner en section 10.1.2.2. Plus conceptuellement, le but était de pouvoir extraire d'un automate la partie qui n'est pas du contrôle : un état partagé entre plusieurs modes de fonctionnement. La métaphore avec les objets est pour souligner que les sous-systèmes partagent un état : les variables partagées sont les variables d'instance et les sous-systèmes sont les méthodes de l'objet. Finalement, chaque sous-système est en boucle simple et le comportement global est régi par une règle de

1. En section 9.6.3, nous discutons de la recherche d'un découpage minimal.

bon ordonnancement des sous-systèmes appelées *scheduling policy*, que nous traduisons par *politique*.

La syntaxe est celle de Lucid Synchrone, inspirée de Ocaml pour les objets. Prenons un objet simulant l'opérateur `0 fby x` :

```
let obj_fby0 = object
  last mem = 0
  when get () returns (y) where
  do
    y = last mem
  done
  when set (x) returns () where
  do
    mem = x
  done
  with (get < set) # {}
end
```

Cet exemple déclare un objet `obj_fby0` qui a une variable partagée `mem`, initialisée à `0`. Trois opérations sont possibles sur les variables partagées, l'écriture et la lecture de la valeur de l'instant comme pour toute variable, ainsi que la lecture de la valeur de l'instant d'avant avec `last mem`. Ainsi, la méthode `get` récupère la valeur précédente du registre tandis que `set` donne la valeur de l'instant courant au registre. Pour être certain que le registre ainsi programmé donne une et une fois les valeurs qu'il stocke, il faut qu'il y ait alternance entre les appels à `get` et à `set`. La solution classique est de faire en sorte qu'à chaque instant, soit le registre synchrone n'est pas activé, soit il donne la valeur qu'il a stockée puis stocke une nouvelle valeur. C'est ce comportement qui est décrit avec la politique `(get < set) # {}` avec l'opérateur `<` de séquence et `#` opérateur de choix.

Le nœud `sum_synchrone` (cf. section 10.1.2.2) que nous souhaitions découper en deux morceaux en partageant un registre s'écrit ainsi :

```
let obj_sum_synchrone = object
  new r = obj_fby0
  last v
  when read () returns (y) where
  do
    v = r.get();
    y = v
  done
  when write (x) returns () where
  do
    r.set(v+x)
  done
  with (read < write) # {}
end
```

La syntaxe `new r = obj_fby0` déclare une instance `r` de notre objet registre. L'appel aux méthodes de `r` se fait avec la notation pointée. Malgré tout, nous faisons appel à une variable partagée `v` pour avoir accès à `y` dans la méthode `write`.

10.2.1 Les politiques en Lucid Synchrone V4

Une politique, associée à un objet, définit les ordonnancements *dans l'instant* des méthodes $\hat{m}$ de l'objet avec la grammaire suivante [2] :

$$\hat{P} ::= \hat{P} \texttt{ || } \hat{P} \mid \hat{P} \texttt{ \# } \hat{P} \mid \hat{P} \texttt{ < } \hat{P} \mid \hat{m} \mid \texttt{\{\}}$$

La politique vide `{}` dénote l'absence d'activation de méthode, $\hat{m}$ est l'appel de la méthode de nom `m`, $\hat{P}$ `<` $\hat{P}'$ est la mise en séquence de deux politiques, $\hat{P}$ est exécutée avant $\hat{P}'$, $\hat{P}$ `#` $\hat{P}'$ est l'exclusion mutuelle de deux politiques, $\hat{P}$ est exécutée sinon $\hat{P}'$ est exécutée et, finalement, $\hat{P}$ `||` $\hat{P}'$ est la mise en parallèle permettant tous les entrelacements entre les exécutions de $\hat{P}$ et $\hat{P}'$.

10.2.1 *Nota bene* **Pas de mémoire**
Le parti pris des politiques est de ne pas avoir de mémoire. Une politique est une condition qui ne dépend pas de l'état du système, elle est identique quel que soit l'instant considéré.

10.2.1.1 Bonne construction d'une politique

La politique a pour but de prévenir les courses critiques *dans l'instant*, ce en vérifiant que toute méthode qui modifie une variable partagée ($\downarrow x$) le fait avant une méthode qui l'utilise ($\uparrow x$). La politique doit en plus assurer qu'il n'y a pas deux écritures d'une variable partagée dans un même instant. En somme, dans l'instant, sont interdits $\uparrow x$ `<` $\downarrow x$ et $\downarrow x$ `<` $\downarrow x$. Par ailleurs, pour ne pas avoir à garder l'ancienne valeur du registre (`last mem`), il a été décidé d'interdire aussi $\downarrow x$ `<` $\uparrow$ `last` x. De cette manière `last mem` sera simplement compilé comme la lecture de `mem` qui contiendra la dernière valeur.

Les politiques s'apparentent à un système de types à effets. On calcule les effets de chaque méthode sur les variables partagées, puis on remplace chaque appel de méthode dans la politique pour obtenir les effets des utilisations possibles de l'objet.

Pour `obj_fby0`, la méthode `get` a pour effet de lire la mémoire $\uparrow mem$ et la méthode `set` a pour effet d'écrire la mémoire $\downarrow mem$. La substitution dans la politique donne ($\uparrow mem$`<`$\downarrow mem$) `#` `{}` qui ne contient pas de comportement interdit. Cette politique est donc bien formée.

Pour `obj_sum_synchrone`, il faut en plus vérifier que l'instance `r` de `obj_fby0` est utilisée conformément à sa politique. Notez que `read` appelle seulement la méthode `get`, tandis que `write` appelle uniquement la méthode `set`. En appliquant ces deux substitutions dans la politique de `obj_sum_synchrone` nous obtenons `(get < set) # {}` ce qui est bien inclus dans les comportements acceptés de `r`. La vérification de l'inclusion des politiques fait appel à la mise en forme normale disjonctive.

10.2.1.2 Mise en forme normale disjonctive

L'opérateur de mise en parallèle peut être considéré comme l'entrelacement de ses composantes. Dans ce cas, une politique se met sous une forme normale disjonctive (cf. [Cas+09]), énumérant les séquences possibles :

$$\mathrm{Norm}(\hat{P}) ::= \#_i \hat{S}_i \qquad\qquad \hat{S} ::= \hat{S} \texttt{ < } \hat{S} \mid \hat{m} \mid \texttt{\{\}}$$

La politique (`get` `<` `set`) `#` `{}` est déjà en forme normale, mais si l'on prend l'exemple de l'objet de Gonthier :

2. Les symbols et lettres sont pris de l'article, mais avec un chapeau en plus pour les différencier des symboles mathématiques que nous utilisons ailleurs dans le manuscrit.

```
let gonthier = object
  when a(x) returns (y) where do y = x done
  when b(x) returns (y) where do y = x done
  with (a # {}) || (b # {})
end
```

sa politique se lit « la méthode `a` peut être activée ou pas, en parallèle de la méthode `b` qui peut être activée ou pas ». Sa forme normale en traduisant le parallèle est `(a < b) # (b < a) # a # b # {}` « `a` avant `b` ou `b` avant `a` ou `a` ou `b` ou rien ».

10.2.1.3 Interprétation d'une politique

La politique fait partie de la spécification de l'objet. Nous ne pouvons parler de sémantique de l'objet sans y référer. L'exemple de gonthier met en évidence qu'une politique mélange en une formule deux notions différentes. Pour chaque instant, il y a un ordre relatif entre les appels des méthodes et il y a le choix d'activer ou pas chaque méthode. Transcrivons ces informations en contraintes sur la datation du système.

Pour une méthode `m`, les entrées et les sorties ont la contrainte d'être en boucle simple : toutes les entrées de l'instant sont avant les sorties de l'instant. Avec $\hat{m}.in$ l'ensemble des ports d'entrée associés à la méthode `m` (resp. $\hat{m}.out$ pour les sorties), cette contrainte s'écrit, pour tout instant k, $T_R^k(\hat{m}.in) \lessdot T_R^k(\hat{m}.out)$. La contrainte d'ordre induite par une séquence se calcule en substituant les méthodes par leurs interprétations, ce qui définit le prédicat suivant :

$$\text{Ordre}(T,R,\hat{S}) \iff \forall k, \hat{S}\Big[\texttt{<}\Big/\lessdot\Big]\Big[\hat{m}\Big/T_R^k(\hat{m}.in) \lessdot T_R^k(\hat{m}.out)\Big]\Big[\texttt{\{\}}\Big/\emptyset\Big]$$

La contrainte de présence des flots, associée à une séquence $\hat{S}$ est d'avoir, pour chaque instant, une valeur pour les entrées et sorties des méthodes exécutées et zéro pour les autres. En notant P_m l'ensemble des ports utilisés dans la séquence $P_m = \hat{S}[\texttt{<}/\cup][\hat{m}/\hat{m}.in \cup \hat{m}.out][\texttt{\{\}}/\emptyset]$ et toujours P l'ensemble de tous les ports, nous pouvons définir le prédicat :

$$\text{Presence}(T,R,\hat{S}) \iff \forall k, \quad \forall p \in P_m, ck(p)[k] = 1 \quad \wedge \quad \forall p \in P \setminus P_m, ck(p)[k] = 0$$

Un système avec $\hat{P}$ sous forme normale disjonctive a pour contrainte de pointage $c_{pol(\hat{P})}$:

$$c_{pol(\hat{P})}(T,R) \iff \hat{P}\Big[\texttt{\#}\Big/\vee\Big]\Big[\hat{S}\Big/\text{Ordre}(T,R,\hat{S}) \wedge \text{Presence}(T,R,\hat{S})\Big]$$

10.2.1.4 Manque de concurrence

La politique de l'object `gonthier` donne la contrainte suivante, en notant a_x (resp. a_y) le port d'entrée (resp. de sortie) de a et avec l'aide des macros $B_a^k = T_R^k(a_x) \lessdot T_R^k(a_y)$ et $C_a^k = ck(a_x)[k] = ck(a_y)[k]$:

$$\begin{aligned} &c_{pol(\texttt{(a<b)\#(b<a)\#a\#b\#\{\}})}(T,R) \\ &\quad\iff \forall k, \quad (B_a^k \lessdot B_b^k \wedge C_a^k = C_b^k = 1) \vee (B_b^k \lessdot B_a^k \wedge C_a^k = C_b^k = 1) \\ &\qquad\qquad \vee (B_a^k \wedge C_a^k = 1 \wedge C_b^k = 0) \vee (B_b^k \wedge C_a^k = 0 \wedge C_b^k = 1) \vee (C_a^k = C_b^k = 0) \end{aligned}$$

La principale remarque que nous devons faire est que le remplacement de `||` par l'entrelacement des *appels* des méthodes *supprime* la concurrence effective entre les méthodes. Par exemple, $B_a^k \lessdot B_b^k$ oblige les sorties de a avant les entrées de b.

10.2.2 *Nota bene* **Concurrence**
L'équivalence entre l'opérateur parallèle et l'entrelacement de ses composantes ne tient que si les méthodes sont considérées atomiques. Cette vision est très restrictive surtout quand on voudra générer du code distribué puisqu'il faudra séquentialiser des actions que l'on voudrait parallèles.

En traitant directement l'opérateur parallèle comme une conjonction de contraintes pour le calcul de l'ordre et comme une union dans le calcul des présences, nous obtiendrions après simplifications [3] une condition beaucoup plus proche de l'intuition :

$$c_{pol((\texttt{a||b})\texttt{\#a\#b\#\{\}})}(T,R) \iff \forall k, B_a^k \wedge B_b^k \wedge C_a^k \leq 1 \wedge C_b^k \leq 1$$

Elle décrit le comportement en boucle simple de chaque méthode avec $B_a^k \wedge B_b^k . C_a^k \leq 1 \wedge C_b^k \leq 1$. Cette contrainte implique qu'une méthode lit et écrit ses entrées dans le même instant et qu'elle est activée au plus une fois par instant.

Ces deux derniers points sont très contraignants. Le seul parallélisme possible met *un* appel de `a` en parallèle à *un* appel de `b`. Tout autre chevauchement est interdit, comme par exemple un appel à `a` qui commence pendant un appel à `b` et finit pendant un autre, ou bien deux appels à `a` en même temps qu'un seul appel à `b`, etc. De telles restrictions sont intrinsèques à l'expression des contraintes en utilisant des instants globaux comme blocs de base.

10.2.1.5 Fils cachés et sémantique en présence d'exclusions

Certes, la sémantique de l'objet dépend de la politique, mais est-on capable de donner une sémantique de Kahn d'un objet muni de sa politique ? La réponse courte est non. Grossièrement, la cause est le choix offert par `#` qui devrait venir avec une propriété d'équité et de confluence.

Manque d'équité Si nous reprenons l'objet `gonthier` et lui fournissons un flot infini `a` sur l'entrée de la méthode `a` et `b` sur l'entrée de `b`. La politique permet d'activer uniquement `a` et, ce, une infinité de fois puisque la politique n'a pas de mémoire (cf. note 10.2.1). Le flot de sortie de `a` est alors infini, tandis que celui de `b` est vide. Ce raisonnement tient en inversant `a` et `b`, ainsi avec le même flot d'entrées, nous avons deux résultats différents qui soulignent le problème d'équité de l'application des politiques. La question d'équité n'est pas la plus traumatisante ; elle pourrait très bien être résolue avec une hypothèse d'équité qui serait probablement cohérente avec l'implémentation.

Non confluence Le problème fondamental est que l'on peut aisément écrire des systèmes non confluents. Regardons l'exemple suivant :

```
let current = object
  last mem = 0
  when set(x) returns () where do
    mem = x
  done
  when get() returns (y) where do
    y = mem
  done
  with (set # {}) < get
end
```

3. Sachant que si $C_a^k = 0$, les ensembles de dates $T_R^k(a_y)$ sont vides et donc B_a^k est vrai.

Le but de l'objet `current` est de récupérer avec `get` la dernière valeur qui a été écrite avec `set`, même si cette dernière l'a été dans l'instant. Il est clair que la sémantique de Kahn est indéterminée tant que ne sont choisis les instants d'écritures. Ainsi, la sémantique est dépendante du pointage. Mais ce dernier n'est pas contraint. En effet dans la politique, rien n'oblige `set` à être appelée. La politique donne après simplification la contrainte de pointage :

$$c_{pol((\texttt{set\#\{\}})<\texttt{get})}(T,R) \iff \forall k, T_R^k(\texttt{x}) \lessdot T_R^k(\texttt{y}) \wedge ck(\texttt{x})[k] \leq 1 \wedge ck(\texttt{y}) = 1$$

Cette contrainte ne réduit pas le domaine d'abstraction puisque, pour toute datation, nous pouvons trouver des instants qui la rende pointable.

Choix des instants et règles de bonne formation Cette liberté dans le choix du pointage rend bancales les règles de bonne formation d'une politique (cf. section 10.2.1.1). Suivant le choix des instants, un même enchaînement (produisant les mêmes sorties pour les mêmes entrées), sera accepté ou refusé. Par exemple la suite avec les instants indiqués par les boîtes `get`; `set; get` est acceptée, quand `get; set`; `get` ne l'est pas, puisque $\downarrow mem <\uparrow mem$ est correct quand $\uparrow mem <\downarrow mem$ est rejeté.

Avec un peu de recul, ces règles avaient pour but de distinguer *dans l'instant* si la lecture d'une mémoire fournissait la valeur de l'instant d'avant ou la valeur de cet instant. Ceci n'a évidemment de sens que si les instants font partie de la sémantique, ce qui était le cas dans [Cas+09], mais ne l'est pas dans notre recherche. Rappelons qu'une sémantique qui dépend des instants ne peut être distribuée efficacement. Dans l'exemple de `current`, le processus qui appelera `get` devra se synchroniser avec celui appelant `set` sans cette synchronisation, le réseau n'a pas de sémantique.

La question de la principalité du pointage est encore loin puisque simplement choisir un pointage serait choisir la sémantique. Il nous faut lever ces ambiguïtés.

10.2.1.6 Proposition pour expliciter les fils cachés [4]

Il est possible de réintégrer l'information de synchronisation dans le système. Cette information est ce que nous appelons un fil caché car nous voulons l'expliciter avec des fils (des entrées supplémentaires, comme des oracles).

Pour `current`, le fil caché est un fil indiquant à chaque instant si `set` est activée. La difficulté vient alors du fait que la politique dépend des *valeurs* de ce fil. Illustrons ceci avec une syntaxe inspirée du `merge` d'Heptagon :

```
let current_kahn = object
  last mem = 0
  when ck_set(b) returns ()
  when set(x) returns () where do
    mem = x
  done
  when get() returns (y) where do
    y = mem
  done
  with ck_set(b) < (#b (true -> set) (false -> {})) < get
end
```

4. Notez que ces fils cachés ne sont pas exactement du même type que ceux dont nous parlions en section 7.4.2.2, nécessaires à la génération de code séquentiel pour les fonctions stables. Nous donnerons plus de détails en parlant de Signal, section 10.3.2.

Le fil caché est l'entrée de la nouvelle méthode `ck_set` qui a pour vocation d'indiquer l'horloge d'appel de la méthode `set`. La méthode en elle même n'a rien à faire. C'est la politique qui va utiliser la valeur de son entrée. Au début de l'instant, `ck_set(b)` obtient une valeur pour `b`. Ensuite il y a un branchement dépendant de `b`, s'il vaut `true` alors `set` sera appelée sinon rien. L'instant se finit avec la methode `get`.

La dépendance en une entrée lue devrait rappeler le pointage d'Heptagon. Mais en réalité, il n'y a pas de raison de se limiter aux entrées, la politique pourrait tout aussi bien dépendre d'une sortie comme c'est le cas en SIGNAL. Dans les deux cas, comme la politique n'a pas d'état, cette valeur doit apparaître avant son utilisation et dans le même instant.

Si tous les choix (`#`) sont conditionnés par un flot, il n'y a plus d'ambiguïté sur la présence des flots et nous retrouvons un pointage. Ce pointage raffiné par les dépendances conditionnées de la politique est un sous-cas du pointage de SIGNAL (cf. section 10.3.4). Pour cette raison nous n'allons pas plus en parler ici.

10.2.2 Proposition de pointage avec politiques

Pour résumer les problèmes des politiques de LUCID SYNCHRONE V4, nous avons un faux parallélisme et une absence de sémantique de Kahn qui interdit le choix modulaire d'un pointage. Pour la sémantique de Kahn, nous avons vu qu'en explicitant les flots utilisés par le contrôle (fils cachés), nous pourrions nous en sortir. Mais ceci rend les choses plus complexes et nous rapproche beaucoup du langage SIGNAL. Dans un premier temps au moins, nous allons mettre ceci de côté.

Nous proposons simplement de bannir l'opérateur de choix `#`. D'autant plus que cela simplifie le traitement et la compréhension du parallélisme. Les questions d'équité disparaissent et l'opérateur de parallélisme a sa propre sémantique qui ne peut se réduire à des entrelacements.

Dernier point qui était fâcheux, le parallélisme restreint à un appel de méthode est décevant. Permettre un nombre non borné d'appels ramène le problème de l'équité. Permettre un nombre borné demande de pouvoir compter, cependant les politiques sont sans mémoire. Une solution est d'accepter plusieurs appels à une même méthode dans un instant. La première conséquence est que les horloges ne sont plus booléennes mais entières. Cette proposition n'entre donc dans aucun cadre usuel des langages synchrones mais est en ligne avec notre proposition pour LUCY-n.

Ci-dessous la grammaire que nous proposons. Elle ne contient pas la politique vide `{}`, car celle-ci n'a pas d'utilité sans l'opérateur de choix.

$$\hat{P} ::= \hat{P} \texttt{ || } \hat{P} \mid \hat{P} \texttt{ < } \hat{P} \mid \hat{m}$$

Pour assurer une sémantique de Kahn, des règles de bonne formation restent nécessaires. Les écritures vraiment concurrentes doivent être interdites $: \downarrow x \texttt{ || } \downarrow x$ et la lecture concurrente d'une écriture aussi $: \uparrow x \texttt{ || } \downarrow x$ (et le symétrique venant de la commutativité de `||`).

10.2.3 *Remarque* Dans la version LUCID SYNCHRONE V4, il serait aussi possible d'avoir plusieurs appels à une méthode dans l'instant. Cependant, l'obstacle principal était la contrainte de bonne formation qui empêche dans un même instant d'avoir des méthodes qui modifient une même variable. Cette contrainte était nécessaire car le parallélisme était équivalent à l'entrelacement.

Il reste une dernière condition. Jusqu'ici, nous n'avons pas considéré de méthode ayant un état propre (non partagé), ce qui arrive s'il y a par exemple un `fby` dans son corps. Dans ce cas, il nous faut pouvoir ordonner les activations d'une méthode. Si $\hat{m}$ est une méthode à état, il faut interdire $\hat{m} \texttt{ || } \hat{m}$.

10.2.3 Exemples

L'object `obj_fby0` (cf. début de la section 10.2) est accepté avec la politique `get < set`. Notez que nous avons supprimé l'ordonnancement vide, ceci n'a pas d'importance car implicitement, si l'objet n'est pas activé, il est de politique vide.

Comme désiré, il est maintenant impossible d'écrire un object comme `current` sans ajouter une nouvelle méthode, pour avoir plus d'information dans la politique.

10.2.3.1 Découplage borné

Considérons `gonthier`, mais avec la politique `(a < a) || (b < b)` que signifie-t-elle ? Dans un instant, deux appels à `a` sont effectués en parallèle de deux appels à `b`. Par exemple les deux premières sorties de `a` peuvent être produites avant la lecture des entrées de `b`. À la fin de l'instant il y a synchronisation. Le résultat est un découplage au maximum de deux appels, en particulier il est impossible d'avoir trois appels à `a` d'avance sur `b`.

De manière générale, l'objet `gonthier` peut être associé à $\texttt{(a <} \overset{i}{\cdots} \texttt{< a) ||(b <} \overset{j}{\cdots} \texttt{< b)}$, qui active i fois `a` en parallèle à j activations de `b`. La fonction de transition correspondante, d'état initial $(0,0)$ est :

$$tF_{g_{ij}}((n_a < i, n_b), (a, \varepsilon)) = ((a, \varepsilon), (n_a + 1, n_b)) \qquad tF_{g_{ij}}((i, j), (\varepsilon, \varepsilon)) = ((\varepsilon, \varepsilon), (0, 0))$$
$$tF_{g_{ij}}((n_a, n_b < j), (\varepsilon, b)) = ((\varepsilon, b), (n_a, n_b + 1))$$

Il est immédiat de voir que cette fonction de transition est confluente et normalisante. Pour tout i et j non nuls, le réseau résultant n'est pas ordonné car il a au moins les deux inverses (a, ε) et (ε, b). Il ne peut donc avoir de datation principale. Cependant son pointage avec la politique est principal. Ceci est un résultat intéressant d'expressivité :

10.2.4 *Propriété* **Plus d'expressivité que le pointage en boucle simple**
Le pointage avec politique que nous proposons peut être principal sans qu'il existe de datation principale, en particulier pour les réseaux exhibant un découplage borné.

10.2.3.2 Pointage périodique

Le fait de compter de manière bornée rend en outre possible la simulation de tout nœud Lucy-n de signature périodique (sans phase d'initialisation puisque nous n'avons pas de mémoire). De manière systématique, nous créons un objet avec une méthode `in` pour chaque entrée du nœud et une méthode `out` pour chaque sortie. Un instant de la politique représentera l'intégralité d'une période de la signature. La politique s'écrit avec q la longueur de la période :

$$\left(\underset{i}{||} (\overbrace{\texttt{in\_}i \texttt{ < } \cdots \texttt{ < in\_}i}^{ck(e^{(i)})[1]}) \right) \texttt{ < } \left(\underset{j}{||} (\overbrace{\texttt{out\_}j \texttt{ < } \cdots \texttt{ < out\_}j}^{ck(e^{(j)})[1]}) \right)$$
$$\texttt{ < } \cdots \texttt{ < } \left(\underset{i}{||} (\overbrace{\texttt{in\_}i \texttt{ < } \cdots \texttt{ < in\_}i}^{ck(e^{(i)})[q]}) \right) \texttt{ < } \left(\underset{j}{||} (\overbrace{\texttt{out\_}j \texttt{ < } \cdots \texttt{ < out\_}j}^{ck(e^{(j)})[q]}) \right)$$

Nous ne faisons que suivre l'ordre induit par la signature et quand il n'y a pas d'ordre, nous mettons en parallèle. Un exemple peut être plus parlant, soit la signature $((1.0)^\omega, 2^\omega) \to 1^\omega$, la politique correspondante serait : `(in_1 || (in_2 < in_2)) < out_1 < in_2 < in_2 < out_1`.

10.2.4 Interprétation des politiques

L'interprétation d'une politique en contrainte sur les datations suit celle que nous avons présentée pour les politiques de LUCID SYNCHRONE V4. La grande différence est que nous avons troqué l'opérateur `#` contre `||`. L'interprétation se basait sur une mise en forme normale disjonctive que nous ne pouvons plus faire. L'opérateur `||` est intuitivement une conjonction de contraintes. La politique $(\hat{m1}$ `||` $\hat{m2})$ `<` $\hat{m3}$ doit produire la contrainte $(\hat{m1}$ `<` $\hat{m3}) \wedge (\hat{m2}$ `<` $\hat{m3})$, cependant nous ne pouvons distribuer innocemment la séquence sur le parallèle car cela demande de répéter le nom d'une méthode. Pour indiquer que les deux occurrences de $\hat{m3}$ correspondent au même appel, nous indexons les appels de méthode. Par exemple, si la politique est $(\hat{m1}$ `||` $\hat{m2})$`<`$\hat{m3}$`<`$\hat{m1}$, nous annotons les occurrences : $(\hat{m1}^1$ `||` $\hat{m2}^1)$`<`$\hat{m3}^1$`<`$\hat{m1}^2$ et nous pouvons alors la mettre sous une forme normale conjonctive : $(\hat{m1}^1$ `<` $\hat{m3}^1$ `<` $\hat{m1}^2)$ `||` $(\hat{m2}^1$ `<` $\hat{m3}^1$ `<` $\hat{m1}^2)$ et traduire l'appel de méthode comme auparavant, `<` par $\ll$ et `||` par $\wedge$.

10.2.5 Parallélisme restreint au fork-join

L'interprétation de l'opérateur de parallélisme comme une conjonction est révélateur. Une politique est la description d'un ordre partiel à l'aide de contraintes. Pourquoi ne pas alors simplement fournir un langage de contraintes ? Pour le faire, il faudrait une possibilité de désigner plusieurs fois un même appel de méthode or nous ne fournissons pas de syntaxe pour ce faire.

Ceci rend la lecture d'une politique assez intuitive car proche de la lecture d'un ordonnancement. Cependant cette contrainte réduit les ordres partiels qui peuvent être décrits à l'aide d'une politique. Seul du parallélisme de tâches imbriquées (*nested task parallelism* ou *fork-join*) est exprimable. Ce modèle de parallélisme est surtout réputé comme étant celui de Cilk[FLR98]. En particulier, il est impossible de décrire l'ordre partiel $(\hat{a}^1$ `<` $\hat{c}) \wedge (\hat{b}$ `<` $\hat{d^1}) \wedge (\hat{a}^1$ `<` $\hat{d^1})$ sans dupliquer `a` ou `d` dans la politique.

10.2.6 Conclusion

Les objets initialement proposés avec LUCID SYNCHRONE V4 sont peu pertinents quand il s'agit de distribuer le code. L'absence de sémantique de Kahn d'une partie de ces objets est problématique. De plus, la simulation du parallélisme par une sémantique d'entrelacement se heurte à la vision atomique de l'appel des méthodes. Nous avons proposé différentes pistes pour combler ces lacunes. La première qui nous rapproche de SIGNAL, est de faire en sorte que les politiques soient des types dépendants. Ainsi l'opérateur de choix des politiques peut être dépendant d'un flot et devenir déterministe. La seconde est de supprimer l'opérateur de choix et de consacrer l'opérateur de mise en parallèle. La possibilité d'écrire plusieurs fois une méthode donne alors assez d'expressivité pour décrire du découplage borné ainsi que toute signature d'horloge périodique (à la LUCY-n). Reste que les politiques imposent une structure fork-join au parallélisme. En Cilk, cette restriction simplifie grandement la gestion des tâches, des communications et de la mémoire. Dans notre cas, les gains ne sont pas si évidents car quoi qu'il arrive nous avons les limites de l'instant qui bornent le parallélisme. Des investigations plus poussées sur la pertinence des politiques nous semblent nécessaires.

10.3 Pointage à la SIGNAL

Contrairement à LUSTRE, SIGNAL a dès le début cherché à avoir un pointage très riche et une gestion fine des dépendances de données. De nombreuses problématiques que nous avons

évoquées se retrouvent en SIGNAL, en particulier les fils cachés, la stabilité des réseaux de Kahn et la représentation du pointage avec des politiques. La force de SIGNAL est aussi son point délicat, le pointage et la compilation reposent sur la résolution de contraintes booléennes possiblement complexes et non modulaires. Ce choix est motivé par une volonté de modélisation puissante, permettant par exemple d'exprimer des contraintes dépendantes de l'exécution, d'effectuer des raffinements, d'importer des modèles AADL, etc. Cependant la compilation devient très complexe et la génération de code séparée n'est pas toujours possible.

10.3.1 Contraintes d'horloges

En SIGNAL, les horloges sont booléennes et n'ont pas de structure imposée. Ceci contraste avec LUSTRE ou Heptagon dont les horloges sont en arbre grâce à des contraintes presque exclusivement syntaxiques (cf. section 2.3.2.1). De plus, en Heptagon, toute horloge provient *directement* de la valeur d'un pointeau (flot du programme) et a donc une existence dans le programme.

En SIGNAL, chaque flot `x` est associé à une inconnue $\hat{x}$ son horloge. Les horloges sont des inconnues car elles sont par défaut implicites. L'utilisation des flots par le programme ajoute des contraintes sur ces inconnues par exemple l'addition de deux flots `x + y` requiert l'égalité de l'horloge de `x` et de `y`. De surcroît, le programmeur peut directement en exprimer. Par exemple pour `x` et `y`, leur synchronisme ($\hat{x} = \hat{y}$) peut être assuré avec une égalité d'horloge de syntaxe `x ^= y`. Cette contrainte d'égalité d'horloges peut avoir comme arguments des expressions d'horloges formées principalement avec les opérateurs d'union d'horloges `^+`, d'intersection `^*` et de différence `^-`. Par exemple `x ^= y ^* z` signifie que `x` est présent ssi `y` et `z` le sont. Il est aussi possible de créer une horloge à partir d'une expression booléenne `c` avec `when c`. La contrainte `x ^= when c` indique que `x` est présent ssi `c` est vraie. Finalement bien qu'implicites, les horloges sont accessibles au programmeur pour définir des flots du programme avec l'expression booléenne `^x` vraie si `x` est présent et fausse sinon.

Les contraintes d'horloges récoltées doivent être satisfaisables sinon les horloges ne sont pas cohérentes. C'est pourquoi le compilateur SIGNAL intègre diverses technologies de preuve. Il a pour objectif de minimiser le nombre de variables du système en le triangularisant (cf. [ABL95 ; Ouy+08]). Ce problème est NP-difficile en général. Le compilateur SIGNAL cherche à hiérarchiser les horloges dans un nombre minimal d'arbres. Chaque nœud correspond à une expression booléenne donnant une horloge, ses nœuds fils sont des sous-horloges, vraies uniquement si leur mère l'est.

Si le compilateur obtient un unique arbre, toutes les variables sont définies à partir de la racine et elle-même est non contrainte. C'est le cas *endochrone*. Nous retombons alors dans le cas simple de LUSTRE et Heptagon avec une horloge de base : la racine de l'arbre.

S'il y a plusieurs arbres, les choses sont plus complexes et diverses stratégies de génération de code existent dans le compilateur. La plus basique est l'endochronisation qui ramène au cas endochrone en ajoutant des fils cachés. Le plus avancé, encore au stade de prototype, est l'utilisation de l'endochronie faible pour la génération de code distribué.

10.3.2 Endochronisation

L'idée de base de l'endochronisation ([Bes+10], aussi appelée *Kahnification* [Ben07]), est très simple. Les horloges ont été hiérarchisées en plusieurs arbres dont les racines sont des variables. Le but est d'avoir un seul arbre d'horloge et donc avoir une unique variable (horloge de base). Une nouvelle variable est créée à cet effet et tous les arbres sont alors ajoutés comme fils de cette nouvelle racine. Pour ce faire, les variables racines des anciens arbres doivent être remplacées

par une condition booléenne. L'approche basique consiste à remplacer chaque variable par une entrée booléenne (un fil caché). Avec ces nouvelles entrées, le programme est entièrement spécifié et a une sémantique de Kahn.

Nous allons distinguer deux type de fils cachés, ceux de type 1 qui sont nécessaires à la sémantique de Kahn et ceux inutiles, de type 2. À notre connaissance, cette distinction n'existe pas dans les travaux à propos de SIGNAL. Notre catégorisation est en effet assez arbitraire, puisqu'en présence de plusieurs fils cachés, le type de chacun peut dépendre de celui des autres, le type d'un fil n'est donc pas une propriété intrinsèque de chaque fil. Toutefois, cette séparation apporte une compréhension plus intuitive de ces fils cachés, ceux de types 2 ne seront utiles que pour générer du code séquentiel, tandis que les autres devront être considérés comme des oracles rendant déterministe une sémantique initialement non déterministe.

10.3.2.1 Fils cachés de type 2

Les fils cachés de type 2 ne participent pas à la sémantique de Kahn. Ils sont du même genre que ceux que nous présentions en section 7.4.2.2 pour compiler les réseaux stables en code séquentiel. Reprenons notre exemple favori pour les illustrer.

Gonthier Ci-dessous, le programme de Gonthier en syntaxe SIGNAL. `?` indique les entrées, `!` les sorties, tandis que les équations `p:= e` sont entre `(|` et `|)` et mises en parallèles avec `|` :

```
process gonthier = (? integer A, B; ! integer X, Y;)
  (| X:= A | Y:= B |)
```

Ce nœud a deux arbres d'horloges (cf. figure 10.1 à gauche), le premier avec un seul nœud contenant `^A` et `^X`, le deuxième avec un seul nœud contenant `^Y` et `^B` (un nœud peut contenir plusieurs expressions booléennes équivalentes). Il y a donc deux variables libres dans ce système. Pour le rendre endochrone, le compilateur le transforme en un code très proche de[5] :

```
process gonthier_SCH_TRA =
  (? integer A, B; boolean C_A, C_B; ! integer X, Y;)
  (| C_A ^= C_B ^= when true
   | when C_A ^= A ^= X
   | X:= A
   | when C_B ^= B ^= Y
   | Y:= B
   |)
```

La première remarque est l'ajout de deux entrées booléennes `C_A` et `C_B`. La première ligne indique que ces deux flots sont présents à tous les instants. La deuxième ligne définit `C_A` comme vrai ssi `A` est présent. Ainsi le flot `C_A` porte la valeur de la variable qui était à la racine du premier arbre d'horloge. L'arbre d'horloge (cf. figure 10.1 à droite) a une racine avec `^C_A` et `^C_B` ainsi que deux feuilles, l'une avec `when C_A`, `^A` et `^X` et l'autre avec `when C_B`, `^B` et `^Y`.

Contraintes partielles La séparation en arbres n'implique pas leur indépendance. Deux arbres peuvent partager des conditions booléennes, sans pour autant qu'un des arbres inclut l'autre. Dans ce cas, les variables d'horloges (les racines) ne sont pas entièrement libres. Les fils rajoutés par l'endochronisation ne pourront pas prendre n'importe quelle combinaison de valeurs et ce sera à l'appelant de s'y conformer.

5. Nous avons nettoyé la sortie du compilateur car il y a beaucoup de choses qui ne servent pas notre présentation.

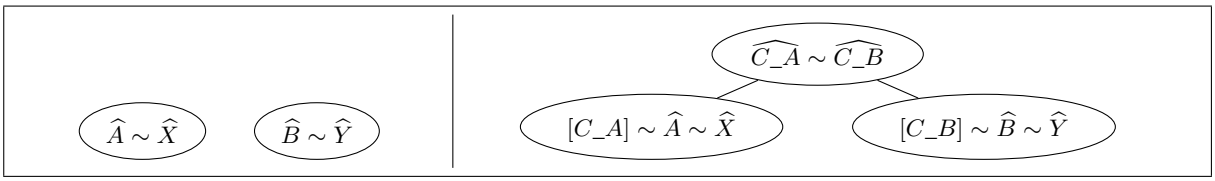
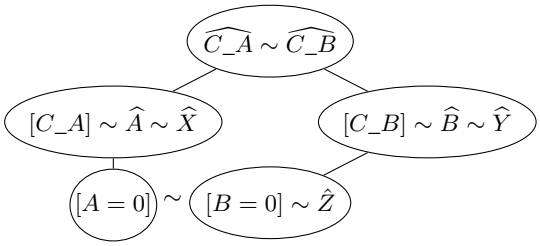


FIGURE 10.1: À gauche la forêt d'horloges de `gonthier` et à droite l'arbre d'horloge de sa version endochronisée. Avec les notation usuelles de SIGNAL, $\widehat{X}$ représente l'horloge du flot `X`, $[X]$ représente les instants où le flot booléen `X` vaut true et $\sim$ indique l'équivalence entre conditions.

Illustrons cela en ajoutant au programme la contrainte `when (A=0) ^= when (B=0)` qui demande que les instants où `A` vaut 0 soient les mêmes que ceux où `B` vaut 0. Nous faisons apparaître cette contrainte avec un calcul :

```
process gonthier_c =
  ( ? integer A, B;
    ! integer X, Y, Z; )
  (| X:= A | Y:= B
   | Z:= (4 when A=0) + (3 when B=0)
   |)
```



L'endochronisation est possible, mais les flots `C_A` et `C_B` ajoutés sont partiellement contraints. Cela se traduit ici, par l'équivalence entre deux feuilles de l'arbre d'horloge.

Quand il reste des contraintes, le pointage peut être incomplet et la génération séparée (sans connaître le contexte d'appel) de code séquentiel se verra refusée par le compilateur avec un message indiquant « clock constraints ».

10.3.2.2 Fils cachés de type 1

SIGNAL est principalement un langage de modélisation. Il permet d'écrire des spécifications qui peuvent réagir à l'absence de leurs entrées. Dans le cadre de la sémantique de Kahn (sans valeur pour signifier l'absence), une telle spécification n'est pas déterministe, puisqu'elle est dépendante de l'ordonnancement de l'exécution. Une fois endochronisée, elle aura une sémantique de Kahn déterministe, donc au moins un des fils ajoutés est nécessaire à la sémantique. Ces fils de type 1 sont des oracles rendant déterministe une sémantique non déterministe.

Il n'est pas besoin d'aller loin pour trouver une spécification non déterministe en SIGNAL. Les opérateurs de base `when` et `default` cachent chacun un oracle.

L'opérateur `when` Le `when` d'Heptagon ou de LUSTRE a autant d'arguments que celui de SIGNAL. Il cache pourtant un fil car il ne contraint pas de la même manière la présence de ses entrées. Comme en Heptagon, `x when b` laisse passer la valeur de `x` si `b` vaut true et la jette si il vaut false. Par contre, si `x` n'est pas présent, `b` est *consommée* sans rien en faire. En SIGNAL, la contrainte sur la présence est une contrainte d'inclusion : `b` doit être présent au moins quand `x` l'est, ce qui s'écrit `x ^< b`. En Heptagon, `x` et `b` doivent avoir les même instants de présence (`x ^= b`). Pour cette raison, si nous écrivons la fonction de transition sans état du `when` de SIGNAL, elle n'est pas confluente :

$$tF_{\texttt{when}}(x, \mathrm{true}) = x \qquad tF_{\texttt{when}}(x, \mathrm{false}) = \varepsilon \qquad tF_{\texttt{when}}(\varepsilon, b) = \varepsilon$$

Pour l'entrée $(\mathbf{x},\mathbf{b}) = (x_1.x_2, \mathrm{false}.\mathrm{true})$, le résultat de $\mathrm{exec}(tF_{\texttt{when}})$ peut valoir soit ε, soit x_1, soit x_2.

L'endochronisation avec le moins de fils possibles est l'ajout de l'entrée booléenne `h`, présente quand `b` l'est (`h ^= b`) qui indique si `x` l'est (`x ^= when h`). En rajoutant ce fil `h` en première entrée de la fonction de transition, elle devient séquentielle (donc confluente et endochrone) :

$$tF_{\texttt{whenh}}(\text{true}, x, \text{true}) = x \qquad tF_{\texttt{whenh}}(\text{true}, x, \text{false}) = \varepsilon \qquad tF_{\texttt{whenh}}(\text{false}, \varepsilon, b)) = \varepsilon$$

L'opérateur `default` Le `default` de SIGNAL joue le rôle du `merge` d'Heptagon (de manière un peu plus distante que celui du `current` de LUSTRE). À nouveaux, les contraintes qu'il impose sur la présence de ses entrées sont moindres. `x default y` vaut la valeur de `x` quand elle est présente et sinon vaut la valeur de `y` quand elle est présente. Il n'y a donc pas de contraintes entre les horloges de `x` et de `y`. La fonction de transition est loin d'être confluente :

$$tF_{\texttt{default}}(x, \varepsilon) = x \qquad tF_{\texttt{default}}(x, y) = x \qquad tF_{\texttt{default}}(\varepsilon, y) = y$$

Pour l'entrée (`x`,`y`) $= (x, y)$, le résultat de $\text{exec}(tF_{\texttt{default}})$ vaut soit x, soit $y.x$, soit $x.y$.

L'endochronisation la plus efficace est l'ajout de l'entrée booléenne `h`, présente si l'une des deux autres l'est (`h ^= x ^+ y`) et valant true ssi `x` est présente (`when h ^= x`). En rajoutant ce fil `h` en première entrée de la fonction de transition, elle devient séquentielle :

$$tF_{\texttt{defaulth}}(\text{true}, x, \varepsilon) = x \qquad tF_{\texttt{defaulth}}(\text{true}, x, y) = x \qquad tF_{\texttt{defaulth}}(\text{false}, \varepsilon, y) = y$$

10.3.1 *Remarque* La sémantique de base de `when` et `default` cache un oracle, mais cela n'implique en rien qu'un programme écrit avec ces opérateurs sera non déterministe. Ce n'est qu'avec la totalité du programme que l'on a toutes les contraintes. Inversement, si tous les opérateurs avaient une sémantique de Kahn (comme en Heptagon), il ne serait pas possible d'avoir un programme non-déterministe, contrevenant à la volonté de modélisation du langage SIGNAL.

10.3.3 Génération de code distribué et endochronie faible

Considérons un programme n'ayant que des fils cachés de type 2, c'est-à-dire dont la sémantique de Kahn est entièrement définie par les fils explicites. Nous pouvons alors imaginer générer du code sans expliciter les fils cachés, en implémentant un réseau de Kahn. Nous retrouvons cette remarque dans [PSS07 ; PSS08], formulée dans l'autre sens : un programme SIGNAL peut se compiler comme un réseau de Kahn sans ajout de fils ssi il décrit un système de transition confluent. Ceci corrobore le résultat classique que nous avons décrit en théorème 6.2.14. Pour autant, ce résultat reste théorique. Vérifier qu'un programme SIGNAL décrit un système de transition confluent semble hors de portée en général.

Les premiers travaux concernant la distribution de code SIGNAL, donc de l'existence d'une sémantique de Kahn sans absence, menaient à la notion d'endochronie, associée à l'isochronie pour la composition [BCL99]. Résumons grossièrement l'isochronie : à la mise en parallèle de réseaux endochrones, s'ils partagent une entrée, ils doivent aussi partager les flots donnant son horloge, sans quoi l'endochronie du résultat n'est pas assurée. Cette approche est intuitive, mais ne compose pas. L'endochronie est une propriété globale du système, comme souligné dans l'erratum [CP03a]. Les complications viennent du parallélisme interne à un nœud endochrone, comme il s'en trouve dans `gonthier`.

Il faut pouvoir décrire un système avec du parallélisme interne, typiquement avec un système de transition confluent. C'est cette recherche qui a poussé la définition de l'endochronie faible (associée à l'isochronie faible), introduite dans [PCB04] (avec la version longue [PCB06]).

10.3.3.1 Endochronie faible

L'endochronie faible est une condition suffisante pour assurer qu'un réseau n'a pas de fils cachés de type 1. Sa définition est donnée dans un contexte *synchrone et relationnel*. Relationnel, c'est-à-dire que le système de transition accepte d'avoir deux transitions $s \xrightarrow{\mathbf{X}/\mathbf{Y}} s'$ et $s \xrightarrow{\mathbf{X}/\mathbf{Y}'} s''$ telles que $\mathbf{Y} \neq \mathbf{Y}'$. Synchrone, au sens où les transitions se font dans un domaine ayant la valeur d'absence $\perp$ et produisent et consomment une et une seule valeur sur tous les ports. Dans ce contexte, les transitions sont écrites $s \xrightarrow{r} s'$ avec r le tuple des valeurs des entrées et des sorties. L'ordre partiel entre les tuples avec absence est en bijection avec l'ordre partiel d'inclusion sur les suites de taille maximale 1. L'extension aux tuples est la même et pour toute valeur x, nous avons $\perp \sqsubseteq x$, similairement au fait que pour tout flot $\mathbf{x} = x$ de taille 1, $\varepsilon \sqsubseteq \mathbf{x}$.

Les trois hypothèses originales de l'endochronie faible sont :

— (W1) Déterminisme : $s \xrightarrow{r} s_1 \wedge s \xrightarrow{r} s_2 \implies s_1 = s_2$

— (W2) Les entrées parallèles sont indépendantes : si r_1 et r_2 sont de supports disjoints ($r_1 \uparrow r_2$ et $r_1 \cap r_2 = \perp$) alors $s \xrightarrow{r_1} s_1$ et $s \xrightarrow{r_2} s_2$ implique que $s \xrightarrow{r_1 \sqcup r_2} s'$, ainsi que $s \xrightarrow{r_1} s_1 \xrightarrow{r_2} s_2$ implique $s \xrightarrow{r_2} s'$.

— (W3) Cloture par intersections compatibles : si r_1 et r_2 sont compatibles et $s \xrightarrow{r_1} s_1$ et $s \xrightarrow{r_2} s_2$ alors il existe $s \xrightarrow{r_1 \sqcap r_2} s'$, $s' \xrightarrow{r_1 \setminus r_2} s_1$ et $s' \xrightarrow{r_2 \setminus r_1} s_2$.

Le contexte relationel simplifie la présentation mathématique en oubliant la distinction entre entrée et sortie. Toutefois, comme souligné dans les articles cherchant à produire du code distribué pour les programmes faiblement endochrones [Pap+11 ; Pot+11], il est nécessaire de se restreindre au cas fonctionnel, pour produire un réseau déterministe.

Stabilité du réseau Dans le cas fonctionnel, l'endochronie faible implique[6] la confluence très forte de la fonction de transition et la fermeture de son domaine par résidus et intersections compatibles. Nous avions noté en propriété 7.3.17 qu'une fonction de transition dont le domaine de définition est clos par intersections compatibles définit un réseau ordonné si elle est de surcroît fortement *normalisante*. Cependant les fonctions de transition considérées pour l'endochronie faible ne sont pas fortement normalisantes. En particulier parce qu'elles sont *stuttering-invariant* c'est-à-dire avec la transition $s \xrightarrow{(\perp,\cdots,\perp)/(\perp,\cdots,\perp)} s$ pour tout état s.

Toutefois, l'endochronie faible implique la stabilité de la fonction de transition (cf. définition 7.2.5) et nous tombons dans le cas de la remarque 7.2.9 qui affirme que le réseau décrit est stable. La stabilité transparait plus clairement avec l'autre caractérisation [Pot+11] couramment donnée d'un programme faiblement endochrone : toutes les transitions sont générées à partir d'un ensemble de transitions atomiques. Ces dernières sont des transitions minimales et *deux à deux indépendantes ou incompatibles*. Dans le cas d'un réseau ordonné nous avons vu qu'à tout moment une transition admet *un* plus petit atome (cf. propriété 7.3.17). N'étant pas ordonné mais simplement stable, il peut y avoir un choix entre différents atomes qui commutent (W2).

Exemples Typiquement `gonthier` est faiblement endochrone car ses transitions se ramènent aux deux atomes indépendants : $0 \xrightarrow{(x,\perp)/(x,\perp)} 0$ et $0 \xrightarrow{(\perp,y)/(\perp,y)} 0$ avec 0 le seul état du système.

De même le réseau de Gustave `GG` (cf. section 7.4.2, avec le code SIGNAL en figure 10.2) est faiblement endochrone et se décompose avec les trois atomes incompatibles $0 \xrightarrow{(\perp,\text{true},\text{false})/\text{true}} 0$, $0 \xrightarrow{(\text{true},\text{false},\perp)/\text{true}} 0$ et $0 \xrightarrow{(\text{false},\perp,\text{true})/\text{true}} 0$ avec 0 le seul état du système. Nous donnons en

6. Voir [Pap+11] pour les preuves. Grossièrement, la confluence très forte (cf. définition 6.2.9) est l'extension de (W2) à toutes les transitions en utilisant (W3) et les fermetures du domaine viennent de (W3) étendue avec (W2).

```
process GG =  ( ? boolean a, b, c; ! event o; )
  (| c1 ^= when a | c1 ^= when (not b)
   | c2 ^= when c | c2 ^= when (not a)
   | c3 ^= when b | c3 ^= when (not c)
   | o ^= c1 default c2 default c3
   |)
  where event c1, c2, c3;
end
```

FIGURE 10.2: Un codage de la fonction de Gustave, un `event` est un type à une seule valeur, ce qui explique que nous n'avons pas d'équation attribuant de valeur à `c1`, `c2`, `c3` et `o`, il n'y a besoin que de spécifier leurs présences avec des contraintes d'horloges.

partie IV deux exemples de programmes SIGNAL non faiblement endochrones qui ont une sémantique de Kahn fonctionnelle non stable.

Code généré Le code généré suit cette vision. Chaque atome est compilé en une fonction de transition. Ensuite une multitudes d'approches sont possibles [Bes+10]. Parlons des deux grandes classes :

- Sans ordonnanceur [Pap+11] : chacune est mise dans son propre thread réagissant quand les bonnes entrées sont présentes, il est ajouté des synchronisations entre ces threads pour que les sorties soient générées dans le bon ordre et les entrées consommées une et une seule fois.
- Avec ordonnanceur [Bes+10] : un seul thread travaille, avec une primitive proche de `select` pour choisir quelle transition prendre. C'est l'approche que nous présentions pour les réseau ordonnés en section 7.4.2. La différence ici étant qu'il y a plusieurs atomes concurrents, l'équité de l'ordonnancement redevient un point à ne pas négliger.

Finalement, notons que la gestion de l'état est un point délicat en pratique. Pour déterminer si un programme SIGNAL est faiblement endochrone [Pot+11], le programme est abstrait en une version sans état en considérant toute variable d'état comme une entrée. Cette abstraction est bien fondée, c'est-à-dire que si l'abstraction est faiblement endochrone alors le programme initial l'était aussi.

10.3.2 *Remarque* **LES RÉSEAUX STABLES, DES PROGRAMMES SIGNAL FAIBLEMENT ENDOCHRONES ?** Remarquez que stutter invariant associé à la restriction aux fonctions implique que le réseau considéré n'est pas Moore et ne fait pas de sur-échantillonage (produit plus de valeur qu'il n'en lit). C'est un point qui a été laissé de côté dans les articles restreignant l'endochronie faible aux fonctions. Sans ces difficultés, il serait probablement aisé de conclure que tout réseau de Kahn stable réactif peut s'écrire comme un programme SIGNAL faiblement endochrone avec des entiers non bornés.

En effet, la fonction de transition tF_F fortement confluente de granularité d'entrée 1 (cf. définition 6.3.14) existe et est stable ssi le réseau décrit est stable (cf. théorème 7.2.8). Les seuls points délicats proviennent du fait que son état est infini et son domaine est clos par intersection et résidu, exception faite de ε, présent pour le premier état des réseaux Moore.

Notez finalement que réactif assure une granularité de sortie de tF_F bornée mais pas inférieure à 1. Il faudra permettre du sur-échantillonage à la SIGNAL : transitions sans consommation d'entrée pour étaler les sorties.

10.3.3.2 Polyendochronie et synthèse de contrôleur

Une approche possible dans certains cas est de générer le code séquentiel endochronisé et de l'envelopper avec un contrôleur dédié. Le contrôleur joue le même rôle que exec jouait (cf. définition 6.2.5) : il décide du préfixe des entrées à consommer, sous la contrainte d'être dans le domaine de définition de la transition. Il a pour tâche supplémentaire de produire les valeurs des fils cachés du code endochrone. La synthèse du contrôleur [Mar+00] peut être déléguée à l'outil Sigali, distribué avec le compilateur SIGNAL dans Polychrony, comme présenté dans [Ouy+08]. Cette solution a pour vocation de traiter les programmes polyendochrone [Ouy08], un cas intermédiaire entre l'endochronie et l'endochronie faible, qui a l'avantage de pouvoir se vérifier en temps polynomial.

Prenons l'exemple `gonthier_c`. Le contrôleur doit assurer que si une des entrées vaut zéro, l'autre aussi. Dès qu'une des entrées est nulle, il devra consommer uniquement l'autre, tant qu'il n'a pas deux entrées nulles.

10.3.4 Graphe de dépendances conditionnées

Outre les horloges, le pointage de SIGNAL calcule les dépendances *dans l'instant*. Ces dépendances sont représentées par un graphe dont les dépendances sont conditionnées à la validité d'une expression booléenne. Suivant les formalismes et le but des articles, ce graphe prend différentes formes, *Synchronous-Flow Dependence Graph* [ML94], *Conditional dependency graph* [Le +91 ; ABL95], *Conditioned Precedence Graph* [Bes+10], etc. Nous n'entrerons pas dans les détails, notre but est de souligner la parenté de SIGNAL sur les politiques.

En SIGNAL, le graphe des dépendances a une syntaxe concrète et peut être explicité dans le source (principalement pour la compilation source à source). Pour indiquer que la valeur de `x` est nécessaire pour calculer la valeur de `y`, la syntaxe est `x --> y`. Implicitement, cette dépendance n'existe qu'aux instants où `y` est présent, ce qui se dénote explicitement avec `{x --> y} when ^y`. Le cas le plus singulier est l'opérateur de complétion `z:= x default y`, qui produit deux dépendances, la première `{x --> z} when ^x` puisque si `x` est présent, `z` prend sa valeur, la deuxième `{y --> z} when (^y ^- ^x)` indique de `y` est nécessaire s'il est présent sans que `x` le soit. Finalement un flot dépend se son horloge ce qui ajoute `^x --> x` pour tout `x`.

Le graphe de dépendances de SIGNAL peut avoir des cycles, contrairement aux politiques qui l'empêchent syntaxiquement (pas de répétition d'un élément comme nous le discutons en section 10.2.5). Un cycle dans ce graphe correspond à un cycle de causalité ssi il est possible de satisfaire la conjonction des conditions d'activations de tous les arcs du cycle. Tout comme le calcul des horloges, la causalité du graphe de dépendances est un problème NP de satisfiabilité booléenne. Les deux problèmes sont en réalité très proches et la hierarchisation des horloges apporte beaucoup d'information pour traiter le problème des dépendances. Dans les versions récentes de SIGNAL, ces deux informations sont regroupées dans une même structure, le graph de contrôles de données (*Data Control Graph* [Bes+10]). Le DCG est une forêt d'horloges avec un graphe de dépendances pour chaque nœud des arbres. C'est cette structure qui contient toutes les informations du pointage et dont les politiques[7] sont un sous-cas.

10.3.5 Un exemple spécifique du pointage de SIGNAL

Contrairement à LUCY-*n* ou Heptagon, le pointage de SIGNAL peut dépendre de conditions booléennes qui ne sont pas portées par des entrées ou des sorties. En particulier, cela permet

7. Les politiques de LUCID SYNCHRONE V4 (sans répétition).

au pointage de dépendre de l'état du nœud, comme le montre l'exemple suivant :

```
process countdown =
  ( ? integer x; ! integer y; )
  (| y:= x default (prev_y - 1)
   | prev_y:= y $ init 0
   | x ^= when (prev_y <= 0)
   |)
   where integer prev_y; end
```

x	2			1		3				2	$\cdots$
prev_y	0	2	1	0	1	0	3	2	1	0	$\cdots$
y	2	1	0	1	0	3	2	1	0	2	$\cdots$

L'horloge de `x` dépend de `prev_y` qui est l'état du nœud, similairement pour les dépendances, puisque nous avons `{x --> y} when prev_y<=0`. Cependant, `prev_y` n'est pas utile à la sémantique de Kahn. Ce programme est endochrone (cf. l'arbre d'horloges en figure 10.3), le compilateur Signal accepte donc de générer séparément son code séquentiel.

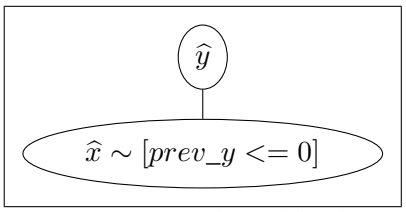


Figure 10.3: Arbre d'horloge de `countdown`

Quoi qu'il en soit, à l'utilisation, il est nécessaire de connaître son pointage (au minimum pour calculer la taille des files de communication, si ce n'est pour être synchrone). Or cette connaissance repose sur l'état, qui lui-même requiert de connaître le code de `countdown`. Le pointage est alors l'intégralité du code source.

10.4 Conclusion

Pour gagner en simplicité ou en expressivité ou en efficacité du code généré, les différents pointages développés par les langages synchrones se sont éloignés des réseaux de Kahn. Nous avons montré dans cette partie comment la compilation en machines de Mealy synchrones de Heptagon fait apparaitre la question de causalité. Nous avons évoqué les différents travaux se penchant sur cette question et proposé d'ajouter à la compilation la possibilité de produire des machines de Moore. Nous sommes aussi revenus sur les politiques synchrones. Leurs but n'a jamais été d'être fidèle aux réseaux de Kahn, mais pour la compilation parallèle cela nous semble primordial. Ainsi nous avons montré les incompatibilités de ce type de pointage et proposé des variantes compatibles avec une sémantique de Kahn. La dernière variante proposée se rapproche fortement du pointage de Signal. Cependant ce dernier est encore plus expressif. Nous avons finalement essayé d'apporter un nouveau regard sur ce pointage en liant ses différentes propriétés avec celle des réseaux de Kahn, en particulier le lien entre l'endochronie et les réseaux ordonnés, ainsi qu'entre la faible endochronie et les réseaux stables.

Quatrième partie

Conclusion

Cette thèse a été pour nous l'occasion de présenter deux facettes importantes de notre travail en lien avec l'exécution parallèle des programmes synchrones. La première est technique avec le développement du compilateur Heptagon et l'introduction des futures. La deuxième est la réflexion théorique concernant le lien entre programmation synchrone et réseaux de Kahn. Par la première nous espérons avoir montré que les futures sont adaptés aux langages synchrones et que leur ajout apporte avec simplicité une grande expressivité. Finalement nous espérons avoir souligné le lien entre la compilation des réseaux de Kahn et les langages synchrones. En particulier, nous avons caractérisé le besoin pour avoir une modularité compatible avec le synchrone, de se limiter aux réseaux ordonnés. En utilisant ce résultat sur le langage Lucy-n, nous avons proposé une nouvelle manière de calculer les horloges, tout en proposant un concept de causalité qui en soit indépendant.

Annexes

Domaines de la sémantique des réseaux de Kahn

Domaine de calcul

Soit un ensemble D muni d'un ordre partiel $\sqsubseteq$ (une relation réflexive, antisymétrique et transitive). Nous donnons ici des définitions génériques concernant les ordres partiels, pour de plus amples détails, se référer par exemple à [KP78].

Borne supérieure et inférieure Pour tout sous-ensemble H de D, $y \in D$ est
- un majorant de H ssi $\forall x \in H, x \sqsubseteq y$, respectivement un minorant de H ssi $\forall x \in H, y \sqsubseteq x$
- une borne supérieure de H notée $\bigsqcup H$ ssi il majore H et minore les majorants de H
- une borne inférieure de H notée $\bigsqcap H$ ssi il minore H et majore les minorants de H

La borne supérieure (resp. inférieure) de deux éléments est notée $x \sqcup x'$ (resp. $x \sqcap x'$).

Ordre partiel complet Un ensemble H est dit dirigé ssi il est non vide et toute pair d'éléments de H est majorée par un élément de H. L'ordre partiel $(D, \sqsubseteq)$ est dit complet ssi il admet un plus petit élément et tout sous-ensemble dirigé admet une borne supérieure. Notez que l'existence de la borne inférieure de tout sous-ensemble existe.

Fonction continue Le principal outil dans les ordres partiels complets est la continuité. Une fonction f d'un ordre partiel $(D, \sqsubseteq)$ dans un autre $(D', \sqsubseteq')$, est continue ssi pour tout ensemble dirigé H de D, $f(\bigsqcup H) = \bigsqcup' f(H)$.

Entres autres, la fonction $\sqcup$, de $D \times D$ dans D est continue.

Ordre partiel cohérent Deux éléments sont dits compatibles (noté $x \uparrow x'$) ssi ils ont une borne supérieure. D est dit cohérent ssi tout ensemble consistant (dont les élément sont deux à deux compatibles) admet une borne supérieure. Remarquez que s'il est cohérent il est aussi complet.

Approximants Un élément x de D est dit isolé ssi de tout ensemble dirigé H dont la borne superieure existe et majore x, on peut extraire un element y qui majore x, c'est-à-dire : $x \sqsubseteq \bigsqcup H \implies \exists y \in H, x \sqsubseteq y$. L'ensemble des éléments isolés majorés par x est appelé l'ensemble de approximants de x et est noté $\mathcal{A}(x)$.

Ordre partiel séparable L'ordre partiel D est dit algébrique ssi tout élément est la borne supérieure de ses approximants ($x = \bigsqcup \mathcal{A}(x)$). Il est séparable ssi il est algébrique et de surcroît l'ensemble de tous les approximants $\mathcal{A}(D)$ est dénombrable.

Domaine de calcul L'ordre partiel D est un domaine de calcul ssi il est cohérent et séparable. Un tel ordre partiel admet de nombreuses propriétés intéressantes, donnons les plus usuelles.
- Un élément est majoré par un autre ssi l'ensemble de ses approximants est inclus dans ceux de l'autre : $x \sqsubseteq y \iff \mathcal{A}(x) \subseteq \mathcal{A}(y)$.
- Entre un élément et un de ses approximants stricts, il y a toujours un autre approximant : $\forall x \in \mathcal{A}(y), x \sqsubset y \implies \exists z \in \mathcal{A}(y), x \sqsubset z \sqsubseteq y$.
- Le produit cartésien d'un nombre dénombrable de domaines de calculs $(D_i, \sqsubseteq_i)_{i \in F}$ est un domaine de calcul. Prenons $(D, \sqsubseteq)$ le produit cartésien de la famille $(D_i, \sqsubseteq_i)_{i \in F}$, alors $X \sqsubseteq Y$ équivaut à pour tout i dans F , $x_i \sqsubseteq_i y_i$, avec $X = (x_1, \cdots, x_i, \cdots)$.
- L'ensemble des fonctions continues d'un domaine de calcul dans un autre est aussi un domaine de calcul, avec l'ordre défini par extensionnalité : f est inférieure à g ssi pour tout x, $f(x) \sqsubseteq g(x)$.

Soulignons deux résultats qui nous sont très utiles :

0.1 Théorème KNASTER-TARSKI
Toute fonction continue $fonction$, du domaine de calcul D dans lui-même, admet un plus petit point fixe $Y(f)$ qui vérifie $Y(f) = \bigsqcup_{n\geq 0} f^n(\bot)$, avec $\bot$ le plus petit élément de D.

0.2 *Propriété* CARACTÉRISATION D'UNE FONCTION CONTINUE DANS UN DOMAINE DE CALCUL
Toute fonction f, d'un domaine de calcul dans un autre est continue ssi elle est [8] :
— *monotone :* $\forall x, x', \quad x \sqsubseteq x' \implies f(x) \sqsubseteq f(x')$
— *à attente finie :* $\forall x, y \in \mathcal{A}(f(x)), \quad \exists x' \in \mathcal{A}(x), y \sqsubseteq f(x')$

Le domaine de Kahn

Les flots

Flots scalaire et tuples de flots Soit T un ensemble dénombrable de valeurs. Nous appellerons scalaire tout élément de T. Un flot de valeurs dans T est une séquence finie ou infinie de scalaires. Nous notons T^* l'ensemble des suites finies, T^ω l'ensemble des suites infinies et $T^\infty = T^* \cup T^\omega$ l'ensemble des flots sur T. Nous parlons de tuple de n flots scalaires pour toute valeur dans $T_1^\infty \times \cdots \times T_n^\infty$. Sans précision, un flot est un tuple de flots scalaires.

Conventions de notations des flots Par conventions, les scalaires sont des lettres latines minuscules en italiques (x), les flots scalaires des lettres minuscules en gras ($\mathbf{x}$) et les tuples de flots des lettres majuscules en gras ($\mathbf{X}$). Les seules exceptions sont le flot vide noté ε et le tuple de flots vides noté $\boldsymbol{\varepsilon}$ (nous omettons la taille du tuple avec cette notation). Sauf contre indication, les composantes d'un tuple de flots $\mathbf{X}$ sont des flots notés avec la même lettre en minuscule, associée à un exposant indiquant sa position dans le tuple, par exemple $\mathbf{X} = (\mathbf{X}^{(1)}, \cdots, \mathbf{X}^{(n)})$.

Concaténation Nous utilisons le point comme opérateur de concaténation, $v.\mathbf{x}$ est la suite de T^∞, dont la première valeur est le scalaire $v \in T$, suivi de la suite $\mathbf{x} \in T^\infty$. Plusieurs sucres syntaxiques et conventions sont utiles. Un scalaire est aussi un flot scalaire à une seule valeur. La concaténation est étendue aux flots vides, $\varepsilon.\mathbf{x} = \mathbf{x}$ et $x.\varepsilon = x$, plus généralement, la concaténation est étendue pour prendre un flot fini $\mathbf{x} \in T^*$ et un flot $\mathbf{y} \in T^\infty$, telle que $(x.\mathbf{x}).\mathbf{y} = x.(\mathbf{x}.\mathbf{y})$. La concaténation est aussi étendue, composante par composante aux tuples de flots : $\mathbf{X}.\mathbf{Y} = (\mathbf{X}^{(1)}.\mathbf{Y}^{(1)}, \cdots, \mathbf{X}^{(n)}.\mathbf{Y}^{(n)})$, avec $\mathbf{X} \in T_1^* \times \cdots \times T_n^*$.

Taille La taille d'un flot $\mathbf{x}$ est notée $|\mathbf{x}|$, elle prend ses valeurs dans $\overline{\mathbb{R}}$, telle que $|\varepsilon| = 0$, $|x.\mathbf{x}| = 1 + |\mathbf{x}|$ et elle est continue, donc si $\mathbf{x}$ est un flot infini (dans T^ω) alors $|\mathbf{x}| = \infty$. La taille est étendue sur les tuples de flots en prenant le maximum de la taille des composantes : $|\mathbf{X}| = \max_{1\leq i\leq n} \left|\mathbf{X}^{(i)}\right|$.

Indexation Le $i$ème élément d'un flot $\mathbf{x}$ est noté $\mathbf{x}[i]$, il n'est défini que si i est inférieur à la taille de $\mathbf{x}$. Nous comptons à partir de 1 : $(x.\mathbf{y})[1] = x$ et pour i supérieur à 1, $(x.\mathbf{y})[i] = \mathbf{y}[i-1]$.

Ordre préfixe du domaine de Kahn

Un flot $\mathbf{x}$ est un préfixe de $\mathbf{x}'$ noté $\mathbf{x} \sqsubseteq \mathbf{x}'$, si $\mathbf{x}'$ commence avec $\mathbf{x}$. La relation de préfixe est un ordre partiel, dont le plus petit élément est ε.

$$\mathbf{v} \sqsubseteq \mathbf{v}' \iff \begin{cases} \exists \mathbf{q} \in T^\infty \ \mathbf{v}.\mathbf{q} = \mathbf{v}' & \text{si } \mathbf{v} \in T^* \\ \mathbf{v} = \mathbf{v}' & \text{sinon} \end{cases}$$

Cet ordre partiel s'étend composante par composante aux tuples de flots, $\mathbf{X} \sqsubseteq \mathbf{Y} \iff \forall 1 \leq i \leq n, \mathbf{X}^{(i)} \sqsubseteq \mathbf{Y}^{(i)}$, son plus petit élément est alors $\boldsymbol{\varepsilon} = (\varepsilon, \cdots, \varepsilon)$.

L'ensemble des tuples de flots, équipé de la relation de préfixe $(T_1^\infty \times \cdots \times T_n^\infty, \sqsubseteq)$, est un domaine de calcul que nous appelons domaine de Kahn. Pour être tout à fait général, nous devrions considérer des tuples dénombrables, nous n'en avons toutefois pas l'utilité dans ce manuscrit.

Les éléments isolés du domaine de Kahn sont exactement les flots finis.

Les chaînes

La presque totalité de nos manipulations dans la sémantique des réseaux de Kahn fait appel à la notion de chaîne. Une chaîne est un ensemble dirigée dont tous les éléments sont comparables. Nous écrivons une chaîne comme une suite de flots $(\mathbf{X}_i)_{1 \leq i \leq p}$ avec p le nombre possiblement infini de flots dans la chaîne, telle que $\mathbf{X}_{i-1} \sqsubseteq \mathbf{X}_i$. Notez que les indices d'une chaîne commencent à 1 et par convention $\mathbf{X}_0 = \boldsymbol{\varepsilon}$. Nous notons la borne supérieure d'une chaîne avec une notation inspirée des suites : $\bigsqcup(\mathbf{X}_i)_i = \lim_{i \to \infty} \mathbf{X}_i$.

Résidus et suite d'incréments Pour tout flots $\mathbf{X}$ et $\mathbf{Y}$ compatibles, le résidu $\mathbf{R} = \mathbf{Y} \setminus \mathbf{X}$ est l'unique flot tel que $(\mathbf{X} \sqcap \mathbf{Y}).\mathbf{R} = \mathbf{Y}$. Si $\mathbf{X}$ est préfixe de $\mathbf{Y}$, nous avons $\mathbf{X}.(\mathbf{Y} \setminus \mathbf{X}) = \mathbf{Y}$.

Pour toute chaîne $(\mathbf{X}_k)_{1 \leq k \leq n}$ dont les éléments sont finis (n possiblement infini), nous définissons la suite d'incréments $(\mathbf{X}_{\delta k})_{1 \leq k \leq n}$ telle que $\mathbf{X}_{k-1}.\mathbf{X}_{\delta k} \stackrel{\text{def}}{=} \mathbf{X}_k$, toujours avec la convention que $\mathbf{X}_0 = \boldsymbol{\varepsilon}$. Notez que $\mathbf{X}_{\delta 1} = \mathbf{X}_1$.

Atomes et chaînes maximales Un flot $\mathbf{Y}$ couvre un flot $\mathbf{X}$, noté $\mathbf{X} \prec \mathbf{Y}$, ssi $\mathbf{X} \sqsubset \mathbf{Y}$ et pour tout flot $\mathbf{Z}$, $\mathbf{X} \sqsubseteq \mathbf{Z} \sqsubseteq \mathbf{Y}$ implique que $\mathbf{Z}$ soit égal à $\mathbf{X}$ ou $\mathbf{Y}$.

Un atome est un flot qui couvre $\boldsymbol{\varepsilon}$. Il n'a donc qu'un seul scalaire : $\mathbf{X}$ est un atome ssi il existe i tel que $\mathbf{X}^{(i)} = x$ et pour tout j différent de i, $\mathbf{X}^{(j)} = \varepsilon$.

La structure des flots permet en outre de bien comprendre la notion de couverture. Si $\mathbf{Y}$ couvre $\mathbf{X}$, alors il existe un atome $\mathbf{A}$ tel que $\mathbf{Y} = \mathbf{X}.\mathbf{A}$.

Une chaîne est dite maximale ssi tous ses incréments $\mathbf{X}_{\delta k}$ sont des atomes. Une propriété intéressante est que tous les chaînes maximales de même borne supérieure $\mathbf{X}$ sont de même longueur. Cette longueur est appelée la hauteur de $\mathbf{X}$. De manière intuitive, la hauteur est le nombre de scalaires qui composent les flots de $\mathbf{X}$, par exemple $(1.1, 1.1.1)$ est de hauteur 5 mais de taille 3.

Chaine approximante Nous dirons qu'une chaîne approxime un flot $\mathbf{X}$ ssi tous ses éléments sont isolés (des flots finis) et sa limite est $\mathbf{X}$. Si l'élément approximé n'est pas isolé, tout chaîne l'approximant est infinie. Pour tout élément il existe une chaîne $(\mathbf{X}_k)_k$ qui l'approxime. Mieux, cette chaîne peut être choisie pour être une chaîne maximale.

Quelques propriétés spécifiques

Propriétés usuelles Le domaine de Kahn n'est pas un domaine concret [KP78], car il ne vérifie pas la propriété Q. Par contre de nombreux résultats se suffisent des propriétés I et C.

0.3 *Propriété* **I**
Entre deux éléments finis comparables, toute chaîne est finie et composée d'éléments finis.

0.4 *Propriété* **C**
Si $\mathbf{X}$ et $\mathbf{Y}$ sont finis et compatibles, alors $(\mathbf{X} \sqcap \mathbf{Y}) \prec \mathbf{X} \implies \mathbf{Y} \prec (\mathbf{X} \sqcup \mathbf{Y})$

Propriétés du domaine avec un seul flot Un flot a une structure très forte. La conséquence la plus remarquable est la borne supérieure de deux flots compatibles : $\mathbf{x} \sqcup \mathbf{y} = \begin{cases} \mathbf{y} & \text{si } \mathbf{x} \sqsubseteq \mathbf{y} \\ \mathbf{x} & \text{sinon} \end{cases}$
C'est pourquoi si deux flots infinis sont compatibles alors ils sont égaux, ce qui n'est pas le cas pour les vecteurs de flots (par exemple $(\varepsilon, 1^\omega)$ et $(1^\omega, \varepsilon)$ sont compatibles, infinis).

Toutes les propriétés qui passent au produit peuvent être prouvées en considérant le domaine avec un seul flot, ce qui donne des conditions très favorables. C'est en particulier le cas pour prouver que le domaine de Kahn est un DI-domaine.

DI-domaine Le domaine de calcul de Kahn est un DI-domaine [Ber78], puisqu'en plus de la propriété I, il a la propriété de distributivité.

0.5 *Propriété* **Distributivité**
$\forall \mathbf{X}, \mathbf{Y}, \mathbf{Z}, \mathbf{Y} \uparrow \mathbf{Z} \implies \mathbf{X} \sqcap (\mathbf{Y} \sqcup \mathbf{Z}) = (\mathbf{X} \sqcap \mathbf{Y}) \sqcup (\mathbf{X} \sqcap \mathbf{Z})$

Programmes Signal non stables

Pour aller plus loin dans la compréhension de ce que l'on peut faire avec Signal, il est intéressant de programmer des fonctions non faiblement endochrones, ayant pour sémantique de Kahn (sans absence) une fonction non stable.

`warn`, réseau non stable sur-échantillonneur

Le nœud `WARN` de l'article de Kahn [Kah74] a une sortie dont la taille est égale à la somme de la taille de ses entrées. Nous avions donné en section 7.2.1 sa représentation sous la forme d'une fonction de transition sans état. Voici une modélisation synchrone possible en Signal :

```
process warn =  ( ? integer x, y; ! event z )
  (| z ^= x ^+ y
   | ^x ^# ^y
   |)
```

La première équation indique que `z` est présent quand une des entrées l'est, la seconde empêche les deux entrées d'être présentes en même temps (`^#` est l'exclusion d'horloges, ce qui est équivalent à `^0 ^= ^x ^* ^y` : l'intersection est égale à l'horloge nulle `^0`, qui est toujours fausse). On est ainsi assuré que `z` est présent le même nombre de fois que la somme des présences des entrées et donc que `warn` a la sémantique de Kahn fonctionnelle recherchée.

La contrainte d'exclusion entre les horloges des entrées doit être assurée (dans le monde synchrone) par l'appelant, ce qui rend ce programme Signal difficile d'utilisation. Or ces

contraintes ne sont pas fondamentales, elles proviennent de l'utilisation d'horloges booléennes, quand il serait souhaitable de produire deux valeurs pour `z` aux instants où les deux entrées sont présentes. Le sur-échantillonnage existe en SIGNAL, mais il repose sur le ralentissement des entrées et fait donc appel à des contraintes empêchant les entrées d'arriver quand elles le souhaitent.

En ajoutant un état, nous sommes en mesure de compter le nombre de fois où `z` peut être activé sans l'arrivée de nouvelles entrées :

```
process warnbl = ( ? integer x, y; ! event z; )
(| bl := next_bl $ init 0
 | together ^= x ^* y
 | onlyone ^= (x ^+ y) ^- together
 | next_bl := (bl + 1) when together
     default bl when onlyone default (bl-1) when z default bl
 | z ^= x ^+ y ^+ when bl>0
 | bl ^> x ^+ y
 |)
where integer bl, next_bl; event together, onlyone; end
```

L'état [9] est `bl`, il stocke la différence entre la somme des présences des entrées et celles de `z`. la sortie est activée si une des entrées est présente ou bien si `bl` est actif et vaut plus que 0. Ainsi, `bl` est incrémenté quand les deux entrées sont présentes, intouché s'il y en a une seule, s'il n'y en a pas et `z` est actif alors il est décrémenté et finalement intouché si tous sont absents.

Pour spécifier correctement le système, il faut au moins que `bl` soit actif quand une des entrées est présente (`bl ^> x ^+ y`). Sinon, on risque de ne pas bien compter ce qu'il se passe, puisque la primitive `default` ne contraint pas `next_bl` à être calculé.

Pour forcer le sur-échantillonage et assurer que `bl` ne dépassera jamais 1, il suffit de rajouter la contrainte `(x ^+ y) ^= when bl=0` qui indique que les entrées arrivent ssi `bl` est nul.

`both`, réseau non stable sans contrainte sur les entrées

Le *ou* parallèle était en section 7.2.1 notre exemple phare de réseau de Kahn non stable. Sa modélisation synchrone en SIGNAL est un peu fastidieuse et comme `warn`, elle impose naturellement des contraintes sur ses entrées [10]. Nous présentons donc `both`, dont la sémantique est la suivante : la longueur de la sortie est égale au maximum des longueurs des entrées. Contrairement à `warn`, il n'y a pas besoin de sur-échantillonage.

Une fonction de transition Mealy dans la sémantique de Kahn serait :

$$tF_{\texttt{both}}(n >= 0, (x, \varepsilon)) = (\text{true}, n+1) \qquad tF_{\texttt{both}}(n <= 0, (\varepsilon, y)) = (\text{true}, n-1)$$
$$tF_{\texttt{both}}(n > 0, (\varepsilon, y)) = (\varepsilon, n-1) \qquad tF_{\texttt{both}}(n < 0, (x, \varepsilon)) = (\varepsilon, n+1)$$

L'état compte la différence entre la longueur de la première entrée et la deuxième. Pour ne pas avoir à contraindre les entrées, nous allons aussi considérer la transition suivante, qui ne change pas la sémantique :

$$tF_{\texttt{both}}(n, (x, y)) = (\text{true}, n)$$

9. Bien évidemment ceci est une approximation puisque les entiers de SIGNAL sont bornés et donc `bl` l'est.

10. En effet, quand une des entrées vaut false, soit on consomme uniquement l'autre entrée en attendant de trouver un autre false, soit on continue de pouvoir consommer les deux entrées, mais alors l'état du système est complexe avec possiblement une infinité de dimensions non bornées, l'utilisation d'une structure de donnée comme la liste est indispensable.

L'état n devrait être un entier non borné. Dans l'implémentation synchrone ci-dessous, nous prenons à la place, un entier de SIGNAL qui est borné :

```
process both = ( ? integer x, y; ! event z; )
(| n:= next_n $ init 0
 | nup ^= x ^- y
 | ndown ^= y ^- x
 | next_n:= (n+1) when nup default (n-1) when ndown default n
 | z ^= (x ^* when n >= 0) ^+ (y ^* when n <= 0) ^+ (x ^* y)
 | n ^= x ^+ y
 |)
where integer n, next_n; event nup, ndown, together; end
```

Bibliographie

[ABL95] Pascalin Amagbégnon, Loïc Besnard et Paul Le Guernic. « Implementation of the data-flow synchronous language SIGNAL ». In : *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*. New York, NY, USA : ACM, 1995, p. 163–173.

[AN90] Arvind et Rishiyur S Nikhil. « Executing a program on the MIT tagged-token dataflow architecture ». In : *IEEE Transactions on Computers* 39.3 (mar. 1990), p. 300–318.

[ANP89] Arvind, Rishiyur S Nikhil et Keshav K Pingali. « I-structures : data structures for parallel computing ». In : *ACM Trans. Program. Lang. Syst.* 11.4 (oct. 1989).

[Aug13] Cédric Auger. « Compilation certifiée de SCADE/LUSTRE ». Thèse de doct. Fév. 2013.

[BA01] Jeff Bonwick et Jonathan Adams. « Magazines and Vmem : Extending the Slab Allocator to Many CPUs and Arbitrary Resources ». In : *2002 USENIX Annual Technical Conference*. USENIX Association, juin 2001, p. 15–33.

[BC83] Gérard Berry et P L Curien. « Theory and practice of sequential algorithms : the kernel of the applicative language CDS ». In : (1983).

[BC84] Gérard Berry et L Cosserat. « The Synchronous Programming Language ESTEREL and its Mathematical Semantics ». In : *Seminar on Concurrency* (1984).

[BCL99] Albert Benveniste, Benoit Caillaud et Paul Le Guernic. « From synchrony to asynchrony ». In : *CONCUR'99 Concurrency Theory* (1999), p. 776–776.

[BCR03] David F Bacon, Perry Cheng et V T Rajan. « The Metronome : A Simpler Approach to Garbage Collection in Real-Time Systems ». In : *On The Move to Meaningful Internet Systems 2003 : OTM 2003 Workshops*. Berlin, Heidelberg : On The Move to Meaningful Internet Systems 2003 : OTM 2003 Workshops, 2003, p. 466–478.

[Ben+10] Mohamed Benazouz, Olivier Marchetti, Alix Munier-Kordon et T Embedded Systems for Real-Time Multimedia ESTIMedia 2010 8th IEEE Workshop on Michel. « A new method for minimizing buffer sizes for Cyclo-Static Dataflow graphs ». In : *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2010 8th IEEE Workshop on* (2010).

[Ben07] Albert Benveniste. « Embedded system design with the Polychronous paradigm ». In : *swste*. Oct. 2007, p. 1–160.

[Ber75] Gérard Berry. *Bottom-up Computation of Recursive Programs*. 1975.

[Ber78] Gérard Berry. « Stable Models of Typed lambda-Calculi ». In : *Proceedings of the Fifth Colloquium on Automata, Languages and Programming*. London, UK, UK : Springer-Verlag, juil. 1978, p. 72–89.

[Ber86] J L Bergerand. « LUSTRE : un langage déclaratif pour le temps réel ». Thèse de doct. 1986.

[Ber99] Gérard Berry. *The Constructive Semantics of Pure Esterel*. Draft Version 3. Juil. 1999.

[Bes+10] Loïc Besnard, Thierry Gautier, Paul Le Guernic et Jean-Pierre Talpin. « Compilation of Polychronous Data Flow Equations ». In : *Synthesis of Embedded ...* Boston, MA : Springer US, juin 2010, p. 1–40.

[BH77] Henry C Baker Jr et Carl Hewitt. « The incremental garbage collection of processes ». In : *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*. New York, NY, USA : ACM, 1977, p. 55–59.

[Bie+08] Darek Biernacki, Jean-Louis Colaço, Grégoire Hamon et Marc Pouzet. « Clock-directed Modular Code Generation for Synchronous Data-flow Languages ». In : *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. Tucson, Arizona, mar. 2008, p. 1–10.

[Bou+08] Julien Boucaron, Anthony Coadou, Benoît Ferrero, Jean-Vivien Millo et Robert de Simone. *Kahn-extended Event Graphs*. Rapp. tech. 2008.

[BTV12] Adnan Bouakaz, Jean-Pierre Talpin et Jan Vitek. « Affine Data-Flow Graphs for the Synthesis of Hard Real-Time Applications ». In : *International Conference on Application of Concurrency to System Design*. IEEE, 2012, p. 183–192.

[Cas+09] Paul Caspi, Jean-Louis Colaço, Léonard Gerard, Marc Pouzet et Pascal Raymond. « Synchronous objects with scheduling policies : introducing safe shared memory in lustre ». In : *LCTES '09 : Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*. ACM Request Permissions, juin 2009.

[Cas+87] Paul Caspi, D Pilaud, Nicolas Halbwachs et J A Plaice. « LUSTRE : a declarative language for real-time programming ». In : *POPL '87 : Proceedings of the 14th ACM SIGACT- SIGPLAN symposium on Principles of programming languages*. New York, 1987, p. 178–188.

[Cas92] Paul Caspi. « Clocks in dataflow languages ». In : *Theoretical Computer Science* 94.1 (1992), p. 125–140.

[CG95] Paul Caspi et Alain Girault. « Execution of distributed reactive systems - Springer ». In : *EURO-PAR'95 Parallel Processing* (1995).

[CGP12] Albert Cohen, Léonard Gerard et Marc Pouzet. « Programming parallelism with futures in lustre ». In : *EMSOFT '12 : Proceedings of the tenth ACM international conference on Embedded software*. ACM Request Permissions, oct. 2012.

[CH86] Paul Caspi et Nicolas Halbwachs. « A functional model for describing and reasoning about time behaviour of computing systems ». In : *Acta Informatica* 22.6 (1986), p. 595–627.

[Coa10] Anthony Coadou. « Réseaux de processus flots de données avec routage pour la modélisation de systèmes embarqués ». Thèse de doct. AOSTE - INRIA Rocquencourt / INRIA Sophia Antipolis / Laboratoire I3S : Université de Nice Sophia-Antipolis, 2010.

[Coh+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)*. Charleston, South Carolina, USA, jan. 2006.

[CP01] Pascal Cuoq et Marc Pouzet. « Modular Causality in a Synchronous Stream Language. » In : *ESOP*. Sous la dir. de David Sands. Springer, 2001, p. 237–251.

[CP03a] Benoit Caillaud et Dumitru Potop-Butucaru. *Erratum to A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in Dataflow Synchronous languages, Specification and Distributed Code Generation.* Rapp. tech. 2003.

[CP03b] Jean-Louis Colaço et Marc Pouzet. « Clocks as First Class Abstract Types ». In : *Third International Conference on Embedded Software (EMSOFT'03)*. Philadelphia, Pennsylvania, USA, mar. 2003, p. 1–17.

[CP96] Paul Caspi et Marc Pouzet. « Synchronous Kahn networks ». In : *ICFP '96 : Proceedings of the first ACM SIGPLAN international conference on Functional programming*. New York, NY, USA : ACM, sept. 1996, p. 226–238.

[CP98] Paul Caspi et Marc Pouzet. « A Co-iterative Characterization of Synchronous Stream Functions ». In : *CMCS '98, First Workshop on Coalgebraic Methods in Computer Science*. 1998, p. 1–40.

[CPP05] Jean-Louis Colaço, Bruno Pagano et Marc Pouzet. « A Conservative Extension of Synchronous Data-flow with State Machines ». In : *ACM International Conference on Embedded Software (EMSOFT'05)* (juil. 2005), p. 1–10.

[Del08] Gwenaël Delaval. « Répartition modulaire de programmes synchrones ». Thèse de doct. Institut National Polytechnique de Grenoble, 2008.

[Den74] J B Dennis. « First version of a data flow procedure language ». In : *Programming Symposium, Proceedings Colloque sur la Programmation*. Springer-Verlag, avr. 1974.

[DGP08] Gwenaël Delaval, Alain Girault et Marc Pouzet. « A Type System for the Automatic Distribution of Higher-order Synchronous Dataflow Programs ». In : *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)* (sept. 2008), p. 1–10.

[DMR10] Gwenaël Delaval, Hervé Marchand et Eric Rutten. « Contracts for Modular Discrete Controller Synthesis ». In : *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2010)*. Stockholm, Sweden, 2010.

[Dur05] Irène Durand. « Call by need computations in orthogonal term rewriting systems ». Thèse de doct. tel.archives-ouvertes.fr, juil. 2005.

[EBL94] M Engels, G Bilson et R Lauwereins. « Cyclo-Static Dataflow : Model and Implementation ». In : *Conference Record of the Twenty-Eighth Asilomar Conference on Signals, Systems and Computers* (1994).

[Fau82] A A Faustini. « An Operational Semantics for Pure Dataflow ». In : *Proceedings of the 9th Colloquium on Automata, Languages and Programming*. London, UK, UK : Springer-Verlag, 1982, p. 212–224.

[FF99] C Flanagan et M Felleisen. « The semantics of future and an application ». In : *Journal of Functional Programming* (1999).

[FLR98] M Frigo, Charles Leiserson et K H Randall. « The Implementation of the Cilk-5 Multithreaded Language ». In : *ACM Symp. on Programming Language Design and Implementation (PLDI'98)*. 1998, p. 212–223.

[FW76] Daniel P Friedman et David S Wise. « The Impact of Applicative Programming on Multiprocessing ». In : *International Conference on Parallel Processing*. Juin 1976, p. 263–272.

[GB03] Marc Geilen et Twan Basten. « Requirements on the execution of Kahn process networks ». In : *Programming Languages and Systems* (2003), p. 319–334.

[Ger+12] Léonard Gerard, Adrien Guatto, Cédric Pasteur et Marc Pouzet. « A modular memory optimization for synchronous data-flow languages : application to arrays in a lustre compiler ». In : *LCTES '12 : Proceedings of the 13th International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*. ACM Request Permissions, juin 2012.

[Ger08] Léonard Gerard. *Des horloges entières pour la répartition de programmes synchrones flot de données*. Rapp. tech. Sept. 2008.

[Ger11] Léonard Gerard. « Oversampling in the Dataflow Synchronous Language Heptagon ». In : *International Open Workshop on Synchronous Programming (SYNCHRON'11)*. Nov. 2011.

[Gir86] Jean-Yves Girard. « The system F of variable types, fifteen years later ». In : *Theoretical Computer Science* 45 (jan. 1986), p. 159–192.

[Gir94] Alain Girault. « Sur la Répartition de Programmes Synchrones ». Thèse de doct. Grenoble, France : INPG, jan. 1994.

[GN03] Alain Girault et Xavier Nicollin. « Clock-Driven Automatic Distribution of Lustre Programs ». In : *Third International Conference, EMSOFT 2003, Philadelphia*. 2003.

[GNP06] Alain Girault, Xavier Nicollin et Marc Pouzet. « Automatic rate desynchronization of embedded reactive programs ». In : *ACM Transactions on Embedded Computing Systems (TECS)* 5.3 (2006), p. 687–717.

[Gon88] Georges Gonthier. « Sémantiques et modèles d'exécution des langages réactifs synchrones : application à ESTEREL ». Thèse de doct. ANRT, 1988.

[GP10] Adrien Guatto et Marc Pouzet. *SCADE/Lustre to VHDL*. Rapp. tech. Août 2010.

[HA87] P Hudak et S Anderson. « Pomset interpretations of parallel functional programs - Springer ». In : *Functional Programming Languages and ...* (1987).

[Hal84] Nicolas Halbwachs. « Modélisation et analyse du comportement des systèmes informatiques temporisés ». Thèse de doct. Université Scientifique et Médicale de Grenoble, juin 1984.

[Hal85] Robert H Halstead Jr. « MULTILISP : a language for concurrent symbolic computation ». In : *ACM Trans. Program. Lang. Syst.* 7 (1985), p. 501–538.

[Ham02] Grégoire Hamon. « Calcul d'horloges et structures de contrôle dans Lucid Synchrone, un langage de flots synchrones à la ML ». Thèse de doct. 2002.

[HL91a] Gérard Huet et J J Levy. « Computations in orthogonal rewriting systems, I ». In : *Computational Logic* (1991).

[HL91b] Gérard Huet et J J Levy. « Computations in orthogonal rewriting systems, II ». In : *Computational Logic* (1991).

[HP00] Grégoire Hamon et Marc Pouzet. « Modular resetting of synchronous data-flow programs ». In : *PPDP '00 : Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming*. ACM Request Permissions, sept. 2000.

[HP85] D Harel et A Pnueli. « On the development of reactive systems ». In : (1985).

[HPW04] T T Hildebrandt, Prakash Panangaden et G Winskel. « A relational model of non deterministic dataflow ». In : *Mathematical Structures in Computer Science* 14.05 (2004), p. 613–649.

[HRR91] Nicolas Halbwachs, Pascal Raymond et Christophe Ratel. « Generating efficient code from data-flow programs ». In : *Programming Language Implementation and Logic Programming*. 1991, p. 207–218.

[Hsu+07] Chia-Jui Hsu, Ming-Yung Ko, Shuvra S Bhattacharyya, Suren Ramasubbu et José Luis Pino. « Efficient simulation of critical synchronous dataflow graphs ». In : *Transactions on Design Automation of Electronic Systems (TODAES* 12.3 (août 2007).

[Hue11] Gérard Huet. « Gustave, ses pompes et ses œuvres ». In : *GGJJ Conference*. Gérardmer, jan. 2011.

[Jim96] Trevor Jim. « What are principal typings and what are they good for ? » In : *POPL '96 : Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Request Permissions, jan. 1996.

[Jon89] B Jonsson. « A fully abstract trace model for dataflow networks ». In : *POPL '89 : Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Request Permissions, jan. 1989.

[Kah74] Gilles Kahn. « The Semantics of a Simple Language for Parallel Programming ». In : *In Information Processing '74 : Proceedings of the IFIP Congress* 74 (1974), p. 471–475.

[Kau70] W H Kautz. « The Necessity of Closed Circuit Loops in Minimal Combinational Circuits ». In : *IEEE Transactions on Computers* C-19.2 (1970), p. 162–164.

[Klo90] Jan Willem Klop. « Term Rewriting Systems : From Church-Rosser to Knuth-Bendix and Beyond ». In : *Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming*. New York, NY, USA : Springer-Verlag New York, Inc., 1990, p. 350–369.

[KM76] Gilles Kahn et David B MacQueen. *Coroutines and Networks of Parallel Processes*. 1976.

[Kok87] J N Kok. « A fully abstract semantics for data flow nets ». In : *PARLE Parallel Architectures and Languages Europe* (1987).

[KP78] Gilles Kahn et Gordon D Plotkin. *Domaines concrets*. Rapp. tech. 336. Déc. 1978.

[KS08] O Kermia et Yves Sorel. « Schedulability Analysis for Non-Preemptive Tasks under Strict Periodicity Constraints ». In : *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA '08. 14th IEEE International Conference on*. 2008, p. 25–32.

[Lam78] Leslie Lamport. « Time, clocks, and the ordering of events in a distributed system ». In : *Communications of the ACM* 21.7 (juil. 1978), p. 558–565.

[Le +91] Paul Le Guernic, Thierry Gautier, M Le Borgne et C Lemaire. « Programming real-time applications with SIGNAL ». In : *Proceedings of the IEEE* 79.9 (1991), p. 1321–1336.

[Lev12] J J Levy. « Sequentiality in Kahn-Macqueen networks and lambda-calculus ». In : *Macqueen Fest*. Mai 2012.

[LH12] Xin Li et R von Hanxleden. « Multithreaded Reactive Programming—the Kiel Esterel Processor ». In : *Computers, IEEE Transactions on* 61.3 (2012), p. 337–349.

[LM87] Edward A Lee et D G Messerschmitt. « Synchronous data flow ». In : *Proceedings of the IEEE*. 1987, p. 1235–1245.

[LS88] B Liskov et L Shrira. « Promises : linguistic support for efficient asynchronous procedure calls in distributed systems ». In : *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*. New York, NY, USA : ACM, 1988, p. 260–267.

[LS89] Nancy Lynch et Eugene W Stark. « A Proof of the Kahn Principle for Input/Output Automata ». In : *Information and Computation* 82 (1989), p. 81–92.

[LS91] Charles Leiserson et James B Saxe. « Retiming synchronous circuitry ». In : *Algorithmica* 6 (1991), p. 5–35.

[LS98] Edward A Lee et Alberto Sangiovanni-Vincentelli. « A framework for comparing models of computation ». In : *IEEE Transactions on computer-aided design of integrated circuits and systems* 17.12 (1998), p. 1217–1229.

[LST09] Roberto Lublinerman, Christian Szegedy et Stavros Tripakis. « Modular code generation from synchronous block diagrams : modularity vs. code size ». In : *POPL '09 : Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Request Permissions, jan. 2009.

[LX04] Edward A Lee et Yuhong Xiong. « A behavioral type system and its application in Ptolemy II ». In : *Formal Aspects of Computing* 16.3 (mai 2004).

[Ma10] Yue Ma. « Compositional Modeling of Globally Asynchronous Locally Synchronous (GALS) Architectures in a Polychronous Model of Computation ». Thèse de doct. 2010.

[Mal93] Sharad Malik. « Analysis of cyclic combinational circuits ». In : *ICCAD '93 : Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*. IEEE Computer Society Press, nov. 1993.

[Mar+00] Hervé Marchand, Patricia Bournai, Michel Le Borgne et Paul Le Guernic. « Synthesis of Discrete-Event Controllers Based on the Signal Environment ». In : *Discrete Event Dynamic Systems* 10.4 (2000), p. 325–346.

[Mil77] Robin Milner. « Fully abstract models of typed λ-calculi ». In : *Theoretical Computer Science* 4.1 (fév. 1977), p. 1–22.

[ML94] Olivier Maffeïs et Paul Le Guernic. « Distributed implementation of SIGNAL : Scheduling & graph clustering ». In : *FTRTFT*. Sous la dir. d'Hans Langmaack, Willem P de Roever et Jan Vytopil. Springer, 1994, p. 547–566.

[MP12] Louis Mandel et Florence Plateau. « Scheduling and Buffer Sizing of n-Synchronous Systems : Typing of Ultimately Periodic Clocks in Lucy-n ». In : *Eleventh International Conference on Mathematics of Program Construction (MPC'12)*. Madrid, Spain, 2012.

[MPP11] Louis Mandel, Florence Plateau et Marc Pouzet. « Static scheduling of latency insensitive designs with Lucy-n ». In : *FMCAD '11 : Proceedings of the International Conference on Formal Methods in Computer-Aided Design*. FMCAD Inc, 2011, p. 171–175.

[MR98] Florence Maraninchi et Yann Rémond. « Mode-Automata : About Modes and States for Reactive Systems ». In : *European Symposium On Programming*. Lisbon (Portugal) : Springer verlag, 1998.

[MRR00] Florence Maraninchi, Yann Rémond et Yannick Raoul. « MATOU : An Implementation of Mode-Automata ». In : *Compiler Construction*. Berlin, Heidelberg : Springer Berlin Heidelberg, mar. 2000, p. 249–263.

[NSS06] Joachim Niehren, Jan Schwinghammer et Gert Smolka. « A Concurrent Lambda Calculus with Futures ». In : *Theoretical Computer Science* 364.3 (2006), p. 338–356.

[Ouy+08] Julien Ouy, Jean-Pierre Talpin, Loïc Besnard et Paul Le Guernic. « Separate compilation of polychronous specifications ». In : *Electronic Notes in Theoretical Computer Science* 200.1 (2008), p. 51–70.

[Ouy08] Julien Ouy. « Génération de code asynchrone dans un environnement polychrone pour la production de systèmes GALS ». Thèse de doct. Rennes : Rennes 1, 2008.

[Pan95] Prakash Panangaden. « The expressive power of indeterminate primitives in asynchronous computation ». In : *Foundations of Software Technology and Theoretical Computer Science* (1995), p. 124–150.

[Pap+11] V Papailiopoulou, Dumitru Potop-Butucaru, Yves Sorel, Robert de Simone, Loïc Besnard et Jean-Pierre Talpin. « From design-time concurrency to effective implementation parallelism : The multi-clock reactive case ». In : *Electronic System Level Synthesis Conference* (2011), p. 1–6.

[Par80] David Park. « On the semantics of fair parallelism ». In : *Abstract Software Specifications* (1980).

[PC06] Dumitru Potop-Butucaru et Benoit Caillaud. « Correct-by-construction asynchronous implementation of modular synchronous specifications ». In : *FUNDAMENTA INFORMATICAE* (2006).

[PCB04] Dumitru Potop-Butucaru, Benoit Caillaud et Albert Benveniste. « Concurrency in synchronous systems ». In : *Application of Concurrency to System Design, 2004. ACSD 2004. Proceedings. Fourth International Conference on*. 2004, p. 67–76.

[PCB06] Dumitru Potop-Butucaru, Benoit Caillaud et Albert Benveniste. « Concurrency in Synchronous Systems ». In : *Formal Methods In System Design* 28.2 (mai 2006), p. 111–130.

[PGF96] Simon Peyton Jones, Andrew Gordon et Sigbjorn Finne. « Concurrent Haskell ». In : *the 23rd ACM SIGPLAN-SIGACT symposium*. New York, New York, USA : ACM Press, 1996, p. 295–308.

[Piz+10a] Filip Pizlo, Lukasz Ziarek, Ethan Blanton, Petr Maj et Jan Vitek. « High-level programming of embedded hard real-time devices ». In : *EuroSys '10 : Proceedings of the 5th European conference on Computer systems*. ACM Request Permissions, avr. 2010.

[Piz+10b] Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L Hosking, Ethan Blanton et Jan Vitek. « Schism : fragmentation-tolerant real-time garbage collection. » In : *PLDI* (2010), p. 146–159.

[Pla10] Florence Plateau. « Modèle n-synchrone pour la programmation de réseaux de Kahn à mémoire bornée ». Thèse de doct. Université Paris-Sud 11, jan. 2010.

[Plo77] Gordon D Plotkin. « LCF considered as a programming language ». In : *Theoretical Computer Science* 5.3 (déc. 1977), p. 223–255.

[Pnu81] A Pnueli. « The temporal semantics of concurrent programs ». In : *Theoretical Computer Science* (1981).

[Pot+11] Dumitru Potop-Butucaru, Yves Sorel, Robert de Simone et Jean-Pierre Talpin. « From Concurrent Multi-clock Programs to Deterministic Asynchronous Implementations ». In : *FUNDAMENTA INFORMATICAE* 108.1-2, SI (2011), 91–118.

[Pou02] Marc Pouzet. « Lucid Synchrone : un langage synchrone d'ordre supérieur ». Habilitation 'a Diriger des Recherches. Paris, France : Université Paris 6 - Pierre et Marie Curie, déc. 2002.

[PPL95] Thomas M Parks, José Luis Pino et Edward A Lee. « A comparison of synchronous and cycle-static dataflow ». In : 1 (1995), p. 204–210.

[PR09] Marc Pouzet et Pascal Raymond. « Modular static scheduling of synchronous dataflow networks : an efficient symbolic representation ». In : *EMSOFT '09 : Proceedings of the seventh ACM international conference on Embedded software*. ACM Request Permissions, oct. 2009.

[PSS07] Dumitru Potop-Butucaru, Robert de Simone et Yves Sorel. « Necessary and sufficient conditions for deterministic desynchronization ». In : *EMSOFT '07 : Proceedings of the 7th ACM & IEEE international conference on Embedded software*. ACM Request Permissions, sept. 2007.

[PSS08] Dumitru Potop-Butucaru, Robert de Simone et Yves Sorel. « Deterministic execution of synchronous programs in an asynchronous environment. A compositional necessary and sufficient condition ». In : *hal.inria.fr* (2008).

[PSS90] Prakash Panangaden, Vasant Shanbhogue et Eugene W Stark. « Stability and Sequentiality in Dataflow Networks ». In : *ICALP*. Sous la dir. de Michael S Paterson. Berlin/Heidelberg : Springer-Verlag, 1990, p. 308–321.

[Ray88] Pascal Raymond. « Compilation séparée de programmes LUSTRE ». Thèse de doct. IMAG, juil. 1988.

[SBT96] Thomas R Shiple, Gérard Berry et H Touati. « Constructive analysis of cyclic circuits ». In : *European Design and Test Conference, 1996. ED&TC 96. Proceedings*. 1996, p. 328–333.

[Sco69] Dana S Scott. « A type-theoretical alternative to ISWIM, CUCH, OWHY ». In : *Theoretical Computer Science* 121.1-2 (déc. 1969), p. 411–440.

[SGB07] Sander Stuijk, Marc Geilen et Twan Basten. « Throughput-Buffering Trade-Off Exploration for Cyclo-Static and Synchronous Dataflow Graphs ». In : *IEEE Trans. Computers* 57.10 (nov. 2007), p. 1331–1345.

[SLG94] V Stoltenberg-Hansen, I Lindström et E R Griffor. *Mathematical Theory of Domains*. Cambridge Tracts in Theoretical Computer Science Series. University Press, 1994.

[Sta87] Eugene W Stark. « Concurrent transition system semantics of process networks ». In : (1987), p. 199–210.

[Sta89] Eugene W Stark. « Concurrent Transition Systems ». In : *Theor. Comput. Sci.* 64.3 (1989), p. 221–269.

[Sto92] L Stok. « False loops through resource sharing ». In : *Computer-Aided Design, 1992. ICCAD-92. Digest of Technical Papers., 1992 IEEE/ACM International Conference on*. 1992, p. 345–348.

[Sut02] Herb Sutter. *The New C++ : Smart(er) Pointers*. Août 2002. url : `http://www.drdobbs.com`.

[Sut12] Herb Sutter. *async and future*. Sept. 2012. url : `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3451.pdf`.

[Tec13] Esterel Technologies. *Scade Suite*. 2013. url : `http://www.esterel-technologies.com/products/scade-system/`.

[Tri+11] Stavros Tripakis, Ben Lickly, Thomas A Henzinger et Edward A Lee. « A Theory of Synchronous Relational Interfaces ». In : *Transactions on Programming Languages and Systems (TOPLAS* 33.4 (juil. 2011).

[Vui74] Jean Etienne Vuillemin. « Correct and optimal implementations of recursion in a simple programming language ». In : *Theoretical Computer Science* 9.3 (déc. 1974), p. 332–354.

[Win86] G Winskel. « Event structures ». In : *Advances in Petri Nets*. 1986, p. 325–392.

[Yua+09] S Yuan, L H Yoong, S Andalam et P S Roop. « A new multithreaded architecture supporting direct execution of Esterel ». In : *EURASIP Journal on …* (2009).

[ZL06] Ye Zhou et Edward A Lee. « A causality interface for deadlock analysis in dataflow ». In : (2006), p. 44–52.