Réseaux de Kahn à rafales et horloges entières

Adrien Guatto, Louis Mandel

▶ **To cite this version:**

Adrien Guatto, Louis Mandel. Réseaux de Kahn à rafales et horloges entières. JFLA 2014 - Vingt-cinquièmes Journées Francophones des Langages Applicatifs, Jan 2014, Fréjus, France. hal-00919281

HAL Id: hal-00919281
https://inria.hal.science/hal-00919281v1

Submitted on 16 Dec 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Réseaux de Kahn à rafales et horloges entières

Adrien Guatto[1,3] & Louis Mandel[2,3]

1: École normale supérieure
2: Collège de France
3: INRIA Paris-Rocquencourt

Résumé

Les langages flot de données synchrones à la Lustre proposent un formalisme équationnel de haut niveau dédié à la conception et l'implantation de systèmes temps réel. Ils sont traditionnellement restreints aux systèmes critiques ne nécessitant pas de calcul intensif ; en particulier, le code impératif généré ne contient pas naturellement de boucles.

Lucy-n est une variante récente de Lustre plus adaptée aux traitements multimédias. Dans cet article, nous proposons une extension de la sémantique de Lucy-n où les flots transportent des rafales de valeurs plutôt que de simples scalaires, ainsi qu'un système de types qui caractérise la taille de ces rafales. L'ambition est d'adapter les techniques de génération de code usuelles pour produire des boucles imbriquées.

1. Modèle n-synchrone et horloges entières

Le modèle n-synchrone [5] a été inventé pour la programmation d'applications vidéos. Il est né d'une collaboration entre des industriels spécialistes de ce domaine et des universitaires experts en langages synchrones [1] à partir du constat suivant.

D'une part, les industriels programmaient leurs applications sous la forme de réseaux de Kahn [6]. Ce modèle décrit les applications comme des réseaux de processus communiquant des flots de données par des buffers. Il a l'avantage de concilier concurrence et déterminisme mais peut nécessiter l'utilisation de buffers de taille infinie. Les ingénieurs devaient donc vérifier manuellement que leurs programmes pouvaient s'exécuter en mémoire bornée et calculer les tailles de buffers.

D'autre part, les universitaires avaient montré [3] que les programmes écrits dans les langages synchrones flots de données décrivaient un sous-ensemble des réseaux de Kahn. Un des avantages de ces langages est qu'ils disposent d'une analyse statique, appelée *calcul d'horloge*, qui borne la taille des buffers du réseau sous-jacent. Cependant, cette analyse impose une taille de un pour tous les buffers !

Le modèle n-synchrone relâche cette dernière contrainte, mais conserve le calcul d'horloge qui permet de garantir l'exécution en mémoire bornée. De plus, il fournit un cadre pour calculer automatiquement la taille des buffers. Le langage Lucy-n [9] a été proposé pour programmer en utilisant ce modèle. C'est un langage synchrone flot de données auquel un opérateur `buffer` a été ajouté.

Afin d'assurer l'exécution des programme en mémoire bornée, le modèle synchrone introduit une notion de temps logique global qui est défini comme une succession d'instants discrets. Ainsi, à chaque flot de valeurs, on peut associer une horloge indiquant la présence où l'absence de données à chaque instant. Le calcul d'horloge est un système de types qui assure que lors de l'application d'un opérateur, les arguments sont disponibles et que, lorsque le résultat est produit, le consommateur est prêt à l'utiliser.

En Lucy-n, comme dans les autres langages synchrones, les horloges sont des flots booléens indiquant à chaque instant s'il y a une valeur ou non dans un flot donné. S'il souhaite transporter

simultanément plusieurs valeurs dans le même flot, le programmeur doit utiliser des structures de données composites comme les tableaux. Cette approche lui laisse le choix de la représentation de ses données, mais aussi la responsabilité de les convertir d'un format en un autre. Par exemple, une image peut être vue comme un flot de pixels ou un flot de lignes de pixels. Pour passer d'une représentation à l'autre, le programmeur doit écrire des convertisseurs série/parallèle.

Dans cet article, nous proposons une approche alternative. L'objectif est de laisser le compilateur introduire automatiquement ce code de conversion. Pour cela, plutôt que de manipuler explicitement des tableaux, le langage autorise le présence simultanée de plusieurs valeurs dans chaque flot. Techniquement, comme imaginé dès les débuts du n-synchrone [4], les horloges sont étendues avec des entiers. Ceux-ci correspondent aux nombres de valeurs présentes à chaque instant dans un flot.

La section 2 présente cette idée intuitivement à travers un exemple. Le langage est ensuite défini formellement section 3. Les propriétés algébriques des horloges entières et le système de types sont décrits section 4. Enfin, la section 5 discute d'un prototype de compilateur.

2. Des horloges booléennes aux horloges entières

Le programme Lucy-n suivant est une fonction, ou *nœud*, qui calcule le produit scalaire de deux flots de vecteurs de taille trois arrivant composante par composante. En pratique, il additionne, trois éléments par trois éléments, le produit de ses deux flots d'entrée :

```
let node dotp_3 (x, y) = o where
  rec p = x * y
  and p0 = p when (true false false)
  and p1 = p when (false true false)
  and p2 = p when (false false true)
  and o = buffer p0 + buffer p1 + buffer p2
```

x	3	2	5	4	2	3	...
y	2	4	6	5	3	7	...
p	6	8	30	20	6	21	...
p0	6			20			...
p1		8			6		...
p2			30			21	...
o			44			47	...

La plupart des opérateurs du langage sont déjà présents dans les langages à la ML : définitions locales mutuellement récursives (`where rec`), fonctions, paires. La particularité du langage est qu'il manipule des flots de valeurs. Ainsi, les opérateurs combinatoires comme `*` s'exécutent point à point : ici, la i-ème valeur du flot `p` est égale au produit des i-èmes valeurs de `x` et `y` ($p_i = x_i \times y_i$).

L'opérateur binaire e `when` ce, spécifique aux langages flots de données synchrones, reçoit un flot de valeurs de type quelconque e et un flot de booléens ce, et conserve les valeurs de e uniquement lorsque ce est vraie. En Lucy-n, ce est restreint à un mot binaire ultimement périodique : `(true false false)` représente la répétition infinie du motif entre parenthèses. Dans le nœud `dotp_3`, l'opérateur `when` permet définir trois sous-flots `p0`, `p1` et `p2` constitués respectivement des premières, deuxièmes et troisièmes valeurs modulo trois du flot `p`.

Enfin, l'opérateur `buffer` est spécifique au langage. Il signifie que l'utilisateur souhaite l'introduction d'un buffer borné au point indiqué. Ces buffers préservent l'ordre de messages et sont sans duplication ni perte.

Calcul d'horloge Le caractère synchrone du langage vient de l'hypothèse suivante : lorsqu'un opérateur du langage consomme ou produit des données, il le fait *instantanément*. Par exemple, un opérateur arithmétique comme `*` produit une valeur en sortie pour une valeur sur chacune de ses entrées, et impose donc que ses entrées et sorties soient présentes exactement aux mêmes instants. Ces instants sont caractérisés par les types d'horloges des flots, et chaque programme Lucy-n induit donc un ensemble de contraintes sur les types d'horloges de ses sous-expressions.

Beaucoup de ces contraintes sont des égalités, à l'image de celles issues de l'opérateur `*`. En revanche, la relation entre le type d'horloges d'entrée et le type d'horloges de sortie de `buffer` est une

relation de sous-typage notée <:. Informellement, cette relation garantit que la communication par buffer fini est correcte, c'est-à-dire qu'il n'y a pas de risque de lire dans le buffer alors qu'il est vide ou d'y écrire alors qu'il est plein.

Le nœud `dotp_3` ci-dessus nous permet d'illustrer ce fonctionnement. Appelons respectivement α_x, α_y, α_p et α_o les types d'horloges de `x`, `y`, `p` et `o`. La définition de `p` utilisant un opérateur arithmétique, elle implique que les types d'horloges de `x`, `y` et `p` sont égaux.

L'opérateur de filtrage *e* `when` *ce* produit une sortie présente sur l'horloge de *e* filtrée par la condition booléenne *ce*. Ici, `p0`, `p1` et `p2` ont donc pour types d'horloges respectifs α_p `on` (1 0 0), α_p `on` (0 1 0) et α_p `on` (0 0 1), `on` dénotant la composition de deux *types d'horloges* et 1 et 0 la présence et l'absence de valeurs. Intuitivement, l'opérateur *on* compose deux *horloges* en parcourant celle de droite au rythme de celle de gauche. Par exemple, (1 0) *on* (1 0 0) se calcule de la façon suivante :

(1 0)	1	0	1	0	1	0	...
(1 0 0)	1		0		0		...
(1 0) *on* (1 0 0)	1	0	0	0	0	0	...

On peut constater ici que si l'entrée d'un `when` arrive un instant sur deux et que la condition conserve une entrée sur trois, la sortie est présente un instant sur six.

Enfin, les sorties des trois buffers utilisés pour définir `o` sont combinées via `+` : leurs types doivent donc être égaux à celui de `o` et respectivement sous-type de ceux de `p0`, `p1` et `p2`. Le système de contraintes final induit par ce nœud est donc :

$$\left\{\begin{array}{rcl} \alpha_x & = & \alpha_y \\ \alpha_y & = & \alpha_p \\ \alpha_p \texttt{ on } (1\ 0\ 0) & <: & \alpha_o \\ \alpha_p \texttt{ on } (0\ 1\ 0) & <: & \alpha_o \\ \alpha_p \texttt{ on } (0\ 0\ 1) & <: & \alpha_o \end{array}\right\}$$

Horloges binaires Interrogé au sujet de ce programme, le compilateur Lucy-n standard calcule sa signature d'horloge et la taille de chacun des trois buffers :

```
val dotp_3 : (int * int) -> int
val dotp_3 :: forall 'a. ('a * 'a) -> 'a on (0 0 1)
Buffer line 6, characters 10-19: size = 1
Buffer line 6, characters 22-31: size = 1
Buffer line 6, characters 34-43: size = 0
```

La première ligne consiste en un schéma de type de données standard à la ML. La seconde décrit les relations temporelles entre les entrées et sorties. Cette signature correspond à la solution $\alpha_x = \alpha_y = \alpha_p = \alpha$ et $\alpha_o = \alpha$ `on` (0 0 1), avec α une variable d'horloge fraîche [1]. On vérifie la satisfaction du système précédent en y remplaçant les inconnues par leurs valeurs :

$$\left\{\begin{array}{rcl} \alpha & = & \alpha \\ \alpha & = & \alpha \\ \alpha \texttt{ on } (1\ 0\ 0) & <: & \alpha \texttt{ on } (0\ 0\ 1) \\ \alpha \texttt{ on } (0\ 1\ 0) & <: & \alpha \texttt{ on } (0\ 0\ 1) \\ \alpha \texttt{ on } (0\ 0\ 1) & <: & \alpha \texttt{ on } (0\ 0\ 1) \end{array}\right\}$$

Ce système de contraintes est toujours satisfait car quelle que soit la valeur de α, les deux premières égalités sont toujours satisfaites. Par ailleurs, il a été montré dans [10] que $w_1 <: w_2$

1. Le compilateur utilise `'a` comme syntaxe concrète compatible ASCII pour α.

```
type dotp_3_mem =
  { b0: int; b1: int; tick: int; }
let dotp_3_step x y mem =
  let p = x * y in
  if mem.tick mod 3 = 0
  then push mem.b0 p;
  if mem.tick mod 3 = 1
  then push mem.b1 p;
  let o =
    if mem.tick mod 3 = 2
    then pop mem.b0 + pop mem.b1 + p
    else (- 1)
  in
  mem.tick <- mem.tick + 1;
  o
```

(a) Horloges binaires, $\alpha_o = \alpha$ `on` (0 0 1)

```
type dotp_3_mem = unit
let dotp_3_step x y mem =
  let p = Array.create 3 0 in
  for i = 0 to 2 do
    p.(i) <- x.(i) * y.(i)
  end;
  p.(0) + p.(1) + p.(2)
```

(b) Horloges entières, $\alpha_o = \alpha$

FIGURE 1 – Codes générés pour le nœud Lucy-n `dotp_3`

implique w `on` $w_1 <: w$ `on` w_2. Donc pour vérifier la satisfaction des trois contraintes de sous-typage, il suffit de vérifier les trois contraintes suivantes : $(1\ 0\ 0) <: (0\ 0\ 1)$, $(0\ 1\ 0) <: (0\ 0\ 1)$ et $(0\ 0\ 1) <: (0\ 0\ 1)$. Il s'agit de contraintes sur des mots ultimement périodiques, appelées contraintes d'*adaptabilité*. Intuitivement, elles sont satisfaites si les deux mots ont la même proportion de 1 et de 0 et si pour tout indice donné, il y a toujours eu plus d'occurrences de 1 dans le mot de gauche que dans celui de droite. Ces deux conditions sont satisfaites dans notre exemple.

Intéressons nous maintenant à la production de code impératif. En Lucy-n comme en Lustre, le but est d'obtenir une fonction de transition produisant, à partir de l'état courant du programme, la valeur du flot au prochain instant ainsi que le nouvel état. Les flots y sont donc représentés comme des scalaires. La compilation modulaire de tels langages est décrite dans l'article [2].

Le code OCaml de la figure 1(a) pourrait être généré à partir de `dotp_3` et de la solution décrite. Le nœud est compilé vers une fonction de transition recevant ses entrées ainsi que son état courant. L'état est une structure à trois champs mutables : deux correspondent aux buffers de taille non nulle, le troisième est un entier indiquant l'instant courant. La fonction met à jour les buffers et calcule la sortie en fonction de l'horloge de ceux-ci. L'horloge est calculée via le compteur `mem.tick`. Remarquons que la valeur -1 définissant `o` lorsque son horloge vaut zéro (au premier et deuxième instant) ne sera jamais utilisée par un appelant si celui-ci respecte la signature d'horloge de `dotp_3`.

Vers les horloges entières Le nœud `dotp_3` calcule le produit scalaire de flots de vecteurs de taille trois reçus linéairement composante par composante. Avec les types d'horloges calculés précédemment, les deux buffers de taille non nulle permettent, tous les trois instants, de rendre disponible simultanément les produits des deux composantes arrivées aux deux instants précédents.

La sémantique de `when` nous assure que la sortie de `dotp_3` sera produite trois fois plus lentement que l'entrée. En Lucy-n, elle sera absente deux instant sur trois par rapport au rythme d'arrivée des entrées. Le code généré doit donc être activé trois fois pour obtenir un résultat. Une alternative serait de recevoir deux tableaux de taille trois pour produire un scalaire à chaque appel, de manière à obtenir le code OCaml de la figure 1(b).

Si les codes de la figure 1 implémentent la même fonctionnalité, ils ont des caractéristiques différentes en terme d'usage des ressources. Le premier est économe en espace, le second en temps ; il paraît difficile de décréter l'un supérieur à l'autre *a priori*. Le programmeur aimerait guider le compilateur pour générer ces deux implémentations à partir du même source initial et d'éventuelles annotations.

Produire une sortie à chaque instant exige de recevoir trois composantes sur `x` et `y` par instant. Nous appellerons un tel paquet de données une *rafale*. Pour prendre en compte les rafales dans le calcul d'horloge, nous étendons les horloges avec des valeurs entières. Ainsi, on peut proposer la solution suivante au système de contraintes de `dotp_3` : $\alpha_x = \alpha_y = \alpha_p = \alpha$ `on` (3) et $\alpha_o = \alpha$ `on` (1). Ceci produit le système de contraintes suivant :

$$\left\{\begin{array}{rcl} \alpha \texttt{ on } (3) & \texttt{=} & \alpha \texttt{ on } (3) \\ \alpha \texttt{ on } (3) & \texttt{=} & \alpha \texttt{ on } (3) \\ \alpha \texttt{ on } (3) \texttt{ on } (1\ 0\ 0) & \texttt{<:} & \alpha \texttt{ on } (1) \\ \alpha \texttt{ on } (3) \texttt{ on } (0\ 1\ 0) & \texttt{<:} & \alpha \texttt{ on } (1) \\ \alpha \texttt{ on } (3) \texttt{ on } (0\ 0\ 1) & \texttt{<:} & \alpha \texttt{ on } (1) \end{array}\right\}$$

Le calcul du *on* s'étend simplement au cas des mots entiers. Le mot de gauche indique de nombre de valeurs à lire le mot de droite :

(3)	3	3	3	...
(1 0 0)	1 0 0	1 0 0	1 0 0	...
(3) *on* (1 0 0)	1	1	1	...

De façon similaire, (3) *on* (0 1 0) = (1) et (3) *on* (0 0 1) = (1). Donc les trois contraintes de sous-typage deviennent α `on` (1) **<:** α `on` (1) et sont donc satisfaites.

Cette solution est découverte par notre prototype de compilateur capable d'inférer des types d'horloges entières lorsqu'on autorise la formation de rafales :

```
val dotp_3
  : (int * int) -> int
  :: ('a on (3) * 'a on (3)) -> 'a
```

À partir de cette signature, il serait possible de générer un code similaire à celui déjà présenté figure 1(b). La fonction OCaml `dotp_3_step` manipule des tableaux et produit une sortie valide par instant. Les types d'horloges à gauche et à droite de chaque contrainte de sous-typage étant les mêmes, les buffers sont de taille nulle. Ainsi, le type `dotp_3_mem` ne contient pas d'information.

Notations Dans ce qui suit, étant donné un ensemble X, nous notons X^* (resp. X^ω) les séquences finies (resp. infinies) d'éléments de X, et ε la séquence finie vide. La longueur d'une séquence finie w est dénotée par $|w|$. Pour une séquence finie de booléens cs, on note $|cs|_{\mathbf{1}}$ (resp. $|cs|_{\mathbf{0}}$) le nombre de booléens vrais (resp. faux) dans cs. On utilise $x[i]$ avec $i \in \mathbb{N}$ pour accéder au i-ème élément de la séquence x, et $x[i,j]$ avec $0 \leq i \leq j$ pour accéder à la sous-séquence de x débutant à l'indice i et terminant à l'indice j. Par abus de notation, $x[i,\dots]$ désigne le flot x privé de ses $i-1$ premiers éléments. On distingue la séquence finie x de $[x]$, cette dernière représentant la séquence (de séquences) de longueur un contenant comme unique élément x

Un élément appartenant à l'ensemble $X^\infty \triangleq X^* \cup X^\omega$ est appelé *flot* (d'éléments de X). On note $x.y$ la concaténation de deux flots, avec $x.y = x$ si x est infini. L'ordre préfixe sur X^∞ est défini par $x \sqsubseteq y$ s'il existe un flot z tel que $x.z = y$. L'ensemble X^∞ ordonné par $\sqsubseteq$ admet ε comme plus petit élément et forme ainsi un ordre partiel complet. Par convention, $u, v, \dots$ désignent des flots finis et $w, w', \dots$ des flots infinis.

Étant donné un ordre partiel complet X et une fonction $f : X \to X$ continue, nous notons $\mathit{fix}_X(f)$ son plus petit point fixe.

Enfin, les opérateurs en gras (**<:**, `on`, etc) sont des opérateurs sur les types et leur version en italique est leur interprétation sur les mots infinis (<:, *on*, etc).

3. Lucy-n et rafales

Si le nœud `dotp_3` précédent admet plusieurs types d'horloges induisant des comportements temporels différents, son action sur les éléments d'un flot est, elle, unique. Pour formaliser cette intuition, nous dotons le langage de deux sémantiques. La première, dite *de Kahn* est atemporelle, indépendante du calcul d'horloge, et manipule des flots de valeurs scalaires. La seconde, dite *synchrone*, manipule des flots de rafales dont la taille est déterminée par ses types d'horloges. Nous montrerons ensuite que ces sémantiques coïncident pour les programmes bien typés.

Nous décrivons les sémantiques d'une variante idéalisée de Lucy-n baptisée Core Lucy-n, essentiellement identique au langage décrit dans la thèse de Plateau [10]. Nous choisissons par souci de simplicité d'évacuer la question orthogonale des types de données. Dans le reste de l'article, les termes *types* et *typage* désigneront respectivement les types d'horloges et le typage d'horloge. On appelle les valeurs de base des *scalaires*. On note S l'ensemble des *scalaires*, celui-ci devant contenir les booléens, les entiers et être fermé par produit.

Le langage On suppose que l'ensemble N_X des noms de variables $x, y, \ldots$, l'ensemble N_F des noms de nœuds $f, g, \ldots$ et l'ensemble N_α des noms de variables de type $\alpha_1, \alpha_2, \ldots$ sont disjoints. Core Lucy-n est un langage à base d'expressions qui sont décrites par la grammaire suivante.

Exp	$\ni$	e	$::=$	c^{st}	constante littérale
			$\mid$	x	variable
			$\mid$	$f^{st}\ e$	application
			$\mid$	(e, e)	couple
			$\mid$	`fst` e $\mid$ `snd` e	projections
			$\mid$	e `where rec` $x = e$	définition locale
			$\mid$	e `when` ce	filtrage
			$\mid$	`merge` $ce\ e\ e$	fusion
			$\mid$	`sync` e	synchronisation
			$\mid$	(`buffer` e)st	mise en mémoire
CExp	$\ni$	ce	$::=$	$b^*(b^+)^{st}$	condition booléenne ultimement périodique

Certaines expressions sont annotées avec un type st. Les types sont formés de variables et de conditions de type. Celles-ci peuvent être entières. La syntaxe des types et conditions de type est :

STy	$\ni$	st	$::=$	α	variable d'horloge
			$\mid$	st `on` stc	horloge composée
STyCond	$\ni$	stc	$::=$	$b^*(b^+) = b$	condition booléenne
			$\mid$	$i^*(i^+)$	mot d'entiers ultimement périodique

Enfin, un programme Core Lucy-n complet est une suite de définitions :

Def	$\ni$	*def*	$::=$	`let node` $f^\alpha\ x$ `=` e	nœud
			$\mid$	*def*; *def*	séquence de définitions

Chaque déclaration `let node` $f^\alpha\ x$ `=` e est annotée avec une variable de type α, quantifiée universellement [2]. Intuitivement, cette variable sera instanciée par l'horloge indiquant le lien avec l'horloge de l'appelant.

2. On se limite à une seule variable uniquement pour simplifier la présentation.

Sémantique de Kahn Les valeurs manipulées par la sémantique de Kahn sont des flots de scalaires. Ces valeurs obéissent à la grammaire suivante :

$$\begin{array}{rcccl} V_K & \ni & v & ::= & S^\infty \quad \text{flot de scalaires} \\ & & & | & (v,v) \quad \text{couple de valeurs de Kahn} \end{array}$$

Pour toute expression e, sa sémantique de Kahn, notée $[\![e]\!]^K$, est une fonction des environnements de Kahn dans les valeurs de V_K. Un environnement de Kahn σ est la donnée d'une fonction $\sigma_v : N_X \to V_K$ et d'une fonction $\sigma_f : N_F \to (V_K \to V_K)$ envoyant les noms de variables et de nœuds dans leurs dénotations respectives.

Étudions la sémantique de chacune des expressions du langage. Les constantes donnent naissance à des flots infinis.

$$\begin{array}{rcl} [\![c^{st}]\!]^K(\sigma) & = & repeat^\#(c) \text{ avec} \\ & & repeat^\#(c) \quad = \quad c.repeat^\#(c) \end{array}$$

Leur sémantique n'utilise pas les annotations de type st car la sémantique de Kahn est atemporelle et ne nécessite donc pas d'horloges.

Les variables, applications, paires, projections et définitions locales récursives reçoivent leur sémantique habituelle.

$$\begin{array}{lcl} [\![x]\!]^K(\sigma) & = & \sigma_v(x) \\ [\![f^{st}\ e]\!]^K(\sigma) & = & (\sigma_f(f))\ ([\![e]\!]^K(\sigma)) \\ [\![(e_1,e_2)]\!]^K(\sigma) & = & ([\![e_1]\!]^K(\sigma), [\![e_2]\!]^K(\sigma)) \\ [\![\texttt{fst}\ e]\!]^K(\sigma) & = & v_1 \text{ avec } [\![e]\!]^K(\sigma) = (v_1,v_2) \\ [\![\texttt{snd}\ e]\!]^K(\sigma) & = & v_2 \text{ avec } [\![e]\!]^K(\sigma) = (v_1,v_2) \\ [\![e_1 \texttt{ where rec } x = e_2]\!]^K & = & [\![e_1]\!]^K(\sigma + [x \mapsto v]) \text{ avec} \\ & & \quad v \quad = \quad fix_{V_K}(\lambda v.[\![e_2]\!]^K(\sigma + [x \mapsto v])) \end{array}$$

L'opérateur binaire de filtrage `when` reçoit un flot de scalaires et un flot booléen baptisé *condition*. Il filtre les valeurs du premier en fonction du second :

$$\begin{array}{rcll} [\![e \texttt{ when } ce]\!]^K(\sigma) & = & when^\#([\![e]\!]^K(\sigma), [\![ce]\!]^K) \text{ avec} & \\ & & when^\#(\varepsilon, \_) & = \quad \varepsilon \\ & & when^\#(\_, \varepsilon) & = \quad \varepsilon \\ & & when^\#(x.xs, t.cs) & = \quad x.when^\#(xs, cs) \\ & & when^\#(x.xs, f.cs) & = \quad when^\#(xs, cs) \\ & & when^\#(\_, f^\omega) & = \quad \varepsilon \end{array}$$

L'opérateur `merge` fusionne deux flots en fonction d'une condition booléenne. Si la condition est vraie, il sélectionne le premier flot, sinon le second :

$$\begin{array}{rcll} [\![\texttt{merge } ce\ e_1\ e_2]\!]^K(\sigma) & = & merge^\#([\![ce]\!]^K, [\![e_1]\!]^K(\sigma), [\![e_2]\!]^K(\sigma)) \text{ avec} & \\ & & merge^\#(\varepsilon, \_, \_) & = \quad \varepsilon \\ & & merge^\#(\_, \varepsilon, \_) & = \quad \varepsilon \\ & & merge^\#(\_, \_, \varepsilon) & = \quad \varepsilon \\ & & merge^\#(t.cs, x.xs, ys) & = \quad x.merge^\#(cs, xs, ys) \\ & & merge^\#(f.cs, xs, y.ys) & = \quad y.merge^\#(cs, xs, ys) \end{array}$$

L'opérateur `sync` transforme un couple de flots en flot de couples :

$$\begin{array}{rcll} [\![\texttt{sync } e]\!]^K(\sigma) & = & sync^\#([\![e]\!]^K(\sigma)) \text{ avec} & \\ & & sync^\#(\varepsilon, \_) & = \quad \varepsilon \\ & & sync^\#(\_, \varepsilon) & = \quad \varepsilon \\ & & sync^\#(x.xs, y.ys) & = \quad (x,y).sync^\#(xs, ys) \end{array}$$

L'opérateur `buffer` correspond intuitivement à un décalage temporel fini entre les dates des entrées et des sorties. La sémantique de Kahn étant atemporelle, cet opérateur est l'identité.

$$[\![\texttt{buffer}\ e]\!]^K(\sigma) \quad = \quad [\![e]\!]^K(\sigma)$$

La sémantique des flots conditionnels ultimement périodiques est :

$$[\![u_b(v_b)^{st}]\!]^K \quad = \quad u_b.repeat^{\#}(v_b)$$

Enfin, on interprète un nœud par une fonction, et la sémantique de la composition de définitions est la composition des sémantiques de chacune.

$$\begin{array}{lcl} [\![\texttt{let node}\ f^{\alpha}\ x\ \texttt{=}\ e]\!]^K & = & \sigma + [f \mapsto \lambda v.([\![e]\!]^K(\sigma + [x \to v]))] \\ [\![def_1; def_2]\!]^K(\sigma) & = & [\![def_2]\!]^K([\![def_1]\!]^K(\sigma)) \end{array}$$

Cette sémantique ne manipule que des flots de scalaires. Le temps est absent, et il n'est donc pas possible de parler de simultanéité ou d'absence de valeur. On décrit donc maintenant une sémantique où chaque scalaire est calculé à une date donnée par rapport à un temps logique global.

Sémantique synchrone Les valeurs de base de cette sémantique sont des flots contenant des rafales. Ces rafales sont simplement des séquences finies de scalaires. La séquence vide représente l'absence de valeur :

$$\begin{array}{lcccll} V_S & \ni & v & ::= & (S^*)^{\infty} & \text{flot de rafales} \\ & & & | & (v, v) & \text{couple de valeurs synchrones} \end{array}$$

Étant donné un tel flot de rafales xs, on définit son *horloge* comme le flot d'entiers $clock^{\#}(xs)$ où à tout rang i de xs, l'entier $(clock^{\#}(xs))[i]$ dénote la taille de la rafale $xs[i]$:

$$\begin{array}{lcl} clock^{\#} & : & (S^*)^{\infty} \to \mathbb{N}^{\omega} \\ clock^{\#}(\varepsilon) & = & repeat^{\#}(0) \\ clock^{\#}(x.xs) & = & |x|.clock^{\#}(xs) \end{array}$$

La fonction $pack^{\#}$ permet de construire des flots de rafales à partir d'une horloge et d'un flot de scalaires :

$$\begin{array}{lcl} pack^{\#} & : & \mathbb{N}^{\omega} \to S^{\infty} \to (S^*)^{\infty} \\ pack^{\#}(\_, \varepsilon) & = & \varepsilon \\ pack^{\#}(n.ck, xs) & = & [xs[1, n]].pack^{\#}(ck, xs[n+1, \omega]) \end{array}$$

Symétriquement, la fonction $unpack^{\#}$ aplatit les rafales pour récupérer le flot de scalaires sous-jacents :

$$\begin{array}{lcl} unpack^{\#} & : & (S^*)^{\infty} \to S^{\infty} \\ unpack^{\#}(\varepsilon) & = & \varepsilon \\ unpack^{\#}(x.xs) & = & x.unpack^{\#}(xs) \end{array}$$

Étudions maintenant la sémantique synchrone $[\![e]\!]^S$ des expressions. C'est une fonction des environnements synchrones dans les flots de rafales V_S. Comme pour un environnement de Kahn, un environnement synchrone σ spécifie pour chaque nom de variable x sa valeur $\sigma_v(x)$ et pour chaque nom de nœud g sa sémantique $\sigma_f(g)$. Mais un environnement synchrone spécifie également pour chaque variable de type α sa valeur $\sigma_{st}(\alpha)$ qui est une horloge $w \in \mathbb{N}^{\omega}$.

La sémantique synchrone des constantes, contrairement à la sémantique de Kahn, utilise l'annotation de type pour calculer l'horloge de la constante et crée les rafales avec l'opérateur $pack^{\#}$.

$$[\![c^{st}]\!]^S(\sigma) \quad = \quad pack^{\#}([\![st]\!]^S(\sigma), repeat^{\#}(c))$$

Cette définition nécessite donc l'interprétation des types $[\![st]\!]^S$ et des conditions entières $[\![stc]\!]^S$:

$$\begin{array}{lcl}
[\![\alpha]\!]^S(\sigma) & = & \sigma_{st}(\alpha) \\
[\![st \texttt{ on } stc]\!]^S(\sigma) & = & [\![st]\!]^S \; on \; [\![stc]\!]^S \\
\\
[\![u_i(v_i)]\!]^S & = & u_i.repeat^{\#}(v_i) \\
[\![u_b(v_b) = b]\!]^S & = & bool^{\#}(u_b.repeat^{\#}(v_b), b) \text{ avec} \\
 & & \quad bool^{\#}(x.cs, b) \;=\; (if\ x = b\ then\ 1\ else\ 0).bool^{\#}(cs, t) \\
\\
[\![u_b(v_b)^{st}]\!]^S(\sigma) & = & pack^{\#}([\![st]\!]^S(\sigma), u_b.repeat^{\#}(v_b))
\end{array}$$

L'intuition de l'opérateur *on* de composition d'horloge est la suivante : composer deux horloges w et w' consiste à exécuter w' au rythme de base défini par w. Quand w contient des entiers supérieurs à un, cela implique de fusionner le nombre d'activation correspondant dans w'. Fusionner des activations signifie additionner les entiers correspondants :

$$\begin{array}{rcl}
on & : & \mathbb{N}^\omega \to \mathbb{N}^\omega \to \mathbb{N}^\omega \\
(n.w)\ on\ w' & \triangleq & (\sum_{i=1}^{n} w'_i).(w\ on\ w'[n+1, \ldots])
\end{array}$$

Par exemple, $(2\ 0)\ on\ 0(1) = 1(0\ 2)$:

$(2\ 0)$	2	0	2	0	2	...
$0(1)$	0 1		1 1		1 1	...
$(2\ 0)\ on\ 0(1)$	1	0	2	0	2	...

Les opérateurs `when`, `merge` et `sync` sont appliqués point à point aux rafales :

$$\begin{array}{lcl}
[\![e \texttt{ when } ce]\!]^S(\sigma) & = & map_2^{\#}(when^{\#}, ([\![e]\!]^S(\sigma), [\![ce]\!]^S(\sigma))) \\
[\![\texttt{merge } ce\ e_1\ e_2]\!]^S(\sigma) & = & map_3^{\#}(merge^{\#}, ([\![ce]\!]^S(\sigma), [\![e_1]\!]^S(\sigma), [\![e_2]\!]^S(\sigma))) \\
[\![\texttt{sync } e]\!]^S(\sigma) & = & map_2^{\#}(sync^{\#}, [\![e]\!]^S(\sigma))
\end{array}$$

Les fonctions de la famille $map_k^{\#}$ utilisées ci-dessus appliquent une fonction d'arité k à k flots :

$$\begin{array}{lcl}
map_k^{\#} & : & ((S^*)^k \to S^*) \to ((S^*)^\infty)^k \to (S^*)^\infty \\
map_k^{\#}(f, (\ldots, \varepsilon, \ldots)) & = & \varepsilon \\
map_k^{\#}(f, (x_1.xs_1, \ldots, x_k.xs_k)) & = & f(x_1,\ \ldots\ x_k).map_k^{\#}(f, (xs_1, \ldots, xs_k))
\end{array}$$

La sémantique synchrone des buffers change l'horloge d'un flot. Pour cela, elle transforme le flot de rafales d'entrée en un flot de scalaires avec l'opérateur $unpack^{\#}$. Puis, comme pour les constantes, elle utilise l'annotation de type pour reconstruire un flot de rafales sur une nouvelle horloge.

$$\begin{array}{lcl}
[\![(\texttt{buffer } e)^{st}]\!]^S & = & buffer^{\#}([\![st]\!]^S(\sigma), [\![e]\!]^S(\sigma)) \\
 & & \text{avec } buffer^{\#}(w, xs) \;=\; pack^{\#}(w, unpack^{\#}(xs))
\end{array}$$

Pour les déclarations de nœuds, l'environnement des nœuds est enrichi avec les nouvelles définitions. Par rapport à la sémantique de Kahn, les fonctions prennent un argument supplémentaire qui est l'horloge de base du nœud. Celle-ci permet de calculer l'interprétation des types qui sont utilisés dans le corps du nœud.

$$\begin{array}{lcl}
[\![\texttt{let node } f^\alpha\ x \texttt{ = } e]\!]^S(\sigma) & = & \sigma + [f \mapsto \lambda w.v.([\![e]\!]^S(\sigma + [\alpha \mapsto w, x \mapsto v]))] \\
[\![def_1; def_2]\!]^S(\sigma) & = & [\![def_2]\!]^S([\![def_1]\!]^S(\sigma))
\end{array}$$

Lors de l'appel d'un nœud, il faut donc instancier cet argument avec l'interprétation de l'horloge de l'appel donnée par l'annotation de type.

$$[\![f^{st}\ e]\!]^S(\sigma) \;=\; (\sigma_f(f))\ ([\![st]\!]^S(\sigma))\ ([\![e]\!]^S(\sigma))$$

Enfin, la sémantique des variables, paires, projections et définitions récursives est identique au cas de la sémantique de Kahn.

4. Horloges entières

4.1. Système de types

Nous présentons d'abord quelques définitions et propriétés des horloges entières comme objets mathématiques [10] [3]. Nous définissions ensuite le système de types fondé sur cette algèbre d'horloge. Les propriétés seront implicitement utilisées par le système de types et justifieront sa correction.

Algèbre des horloges L'opérateur de composition *on* défini précédemment est associatif et admet un élément neutre et un absorbant :

Propriété 1. *Pour toute horloge w, w' et w'' :*
- w *on* $(1) = (1)$ *on* $w = w$
- w *on* $(0) = (0)$ *on* $w = (0)$
- $(w$ *on* $w')$ *on* $w'' = w$ *on* $(w'$ *on* $w'')$

La sémantique de l'opérateur `buffer` n'impose *a priori* aucune relation entre l'horloge de son entrée et l'horloge de sa sortie. Nous souhaitons toutefois rejeter certains types de comportements :
- on ne lit jamais dans un buffer vide ;
- le buffer est de capacité finie (et on n'écrit jamais dans un buffer plein).

Ces conditions peuvent être formalisées via la fonction de cumul d'une horloge w : $\mathcal{O}_w(i)$. Cette fonction fournit la quantité totale de données transportée par w du premier au i-ème instant. Elle est définie par :

$$\begin{array}{lcl} \mathcal{O} & : & \mathbb{N}^\omega \to \mathbb{N} \to \mathbb{N} \\ \mathcal{O}_w(0) & \triangleq & 0 \\ \mathcal{O}_{n.w}(1+i) & \triangleq & n + \mathcal{O}_w(i) \end{array}$$

La quantité de données contenue dans un buffer dont l'entrée (resp. la sortie) est horloge w (resp. w') à la fin de l'instant i peut maintenant être décrite comme $\mathcal{O}_w(i) - \mathcal{O}_{w'}(i)$. La première propriété de ne jamais lire dans un buffer vide peut être caractérisée par la relation de *précédence*, notée $\preceq$. La seconde propriété qui garantit que le nombre d'écritures et de lectures dans le buffer ne va pas diverger peut être caractérisée par la relation de *synchronisabilité*, notée $\bowtie$. La conjonction des deux relations forme la relation d'*adaptabilité* notée $<:$ et évoquée en section 2 :

$$\begin{array}{lcl} w \preceq w' & \triangleq & \forall i \in \mathbb{N}.\ \mathcal{O}_w(i) \geq \mathcal{O}_{w'}(i) \\ w \bowtie w' & \triangleq & \exists b \in \mathbb{N}.\ \forall i \in \mathbb{N}.\ -b \leq \mathcal{O}_w(i) - \mathcal{O}_{w'}(i) \leq b \\ w <: w' & \triangleq & w \preceq w' \wedge w \bowtie w' \end{array}$$

Une propriété importante de l'adaptabilité est sa préservation par changement d'horloge de base. C'est en effet pour cette raison que l'on peut instancier l'horloge de base d'un nœud par une valeur arbitraire.

Lemme 1. *Pour toutes horloges w et w' et à chaque instant i, on a $\mathcal{O}_{w \texttt{ on } w'}(i) = \mathcal{O}_{w'}(\mathcal{O}_w(i))$.*

Propriété 2. *Pour toutes horloges w, w' et w'', si $w <: w'$ alors w'' `on` $w <: w''$ `on` w'.*

Démonstration. Prouvons la précédence :

$$\begin{array}{lcll} w'' \texttt{ on } w \preceq w'' \texttt{ on } w' & \Leftrightarrow & \forall i.\ \mathcal{O}_{w'' \texttt{ on } w}(i) \geq \mathcal{O}_{w'' \texttt{ on } w'}(i) & \{ \text{ par définition de } \preceq \ \} \\ & \Leftrightarrow & \forall i.\ \mathcal{O}_w(\mathcal{O}_{w''}(i)) \geq \mathcal{O}_{w'}(\mathcal{O}_{w''}(i)) & \{ \text{ par le lemme 1 } \} \\ & \Leftrightarrow & \mathit{true} & \{ \text{ par } w \preceq w' \ \} \end{array}$$

Le raisonnement est identique pour la synchronisabilité. □

3. Une formalisation Coq de ces propriétés par Florence Plateau et Louis Mandel est disponible à l'adresse `http://lucy-n.org/coq/inw_prop.html`.

$$\textsc{Const}\ \frac{}{\Gamma \vdash c^{st} :: st} \qquad \textsc{Var}\ \frac{\Gamma(x) = st}{\Gamma \vdash x :: st} \qquad \textsc{App}\ \frac{\Gamma(f) = ty_{sch} \qquad Inst(ty_{sch},\ st'') = st, st' \qquad \Gamma \vdash e :: st}{\Gamma \vdash f^{st''}\ e :: st'}$$

$$\textsc{Pair}\ \frac{\Gamma \vdash e_1 :: st_1 \qquad \Gamma \vdash e_2 :: st_2}{\Gamma \vdash (e_1, e_2) :: st_1 \times st_2} \qquad \textsc{Fst}\ \frac{\Gamma \vdash e :: st_1 \times st_2}{\Gamma \vdash \texttt{fst}\ e :: st_1} \qquad \textsc{Snd}\ \frac{\Gamma \vdash e :: st_1 \times st_2}{\Gamma \vdash \texttt{snd}\ e :: st_2}$$

$$\textsc{When}\ \frac{\Gamma \vdash e :: st}{\Gamma \vdash e\ \texttt{when}\ p^{st} :: st\ \texttt{on}\ (p = \texttt{true})} \qquad \textsc{Merge}\ \frac{\Gamma \vdash e_1 :: st\ \texttt{on}\ (p = \texttt{true}) \qquad \Gamma \vdash e_2 :: st\ \texttt{on}\ (p = \texttt{false})}{\Gamma \vdash \texttt{merge}\ p^{st}\ e_1\ e_2 :: st}$$

$$\textsc{Sync}\ \frac{\Gamma \vdash e :: st \times st}{\Gamma \vdash \texttt{sync}\ e :: st} \qquad \textsc{Buffer}\ \frac{\Gamma \vdash e :: st \qquad st <: st'}{\Gamma \vdash (\texttt{buffer}\ e)^{st'} :: st'}$$

$$\textsc{Where}\ \frac{\Gamma + [x \mapsto st] \vdash e_2 :: st \qquad \Gamma + [x \mapsto st] \vdash e_1 :: st'}{\Gamma \vdash e_1\ \texttt{where rec}\ x = e_2 :: st'}$$

FIGURE 2 – Calcul d'horloge pour Core Lucy-n.

Enfin, remarquons que les horloges ultimement périodiques sont des objets manipulables algorithmiquement.

Propriété 3. *La composition de deux horloges ultimement périodiques est calculable et ultimement périodique. L'adaptabilité de deux horloges ultimement périodiques est décidable.*

Munis de ces éléments mathématiques, on décrit maintenant les propriétés des types d'horloges et du calcul d'horloge, reliant horloges, types et programmes.

Types composés et calcul d'horloge Les types d'horloges simples dénotés par st caractérisent des flots de scalaires ou des flots de couples de scalaires. Le langage comprenant également des couples de flots dont les deux composantes peuvent être d'horloges différentes, nous introduisons les types d'horloges composés ty.

ty	::=		type d'horloges composé
	|	st	type d'horloges simple
	|	$ty \times ty$	type produit

Un nœud Core Lucy-n est polymorphe en ses horloges de base décrits par des schémas de types ty_{sch}. Ceux-ci ne lient ici qu'une seule variable, par simplicité.

$$ty_{sch} \quad ::= \quad \forall \alpha.\ ty \ \rightarrow\ ty \quad \text{schéma de type d'horloges composé}$$

Le jugement de typage $\Gamma \vdash e :: ty$ indique que dans le contexte de typage Γ, l'expression e a le type composé ty. Un environnement Γ associe un nom de variable à un type composé et un nom de nœud à un schéma de type.

La figure 2 présente les règles du calcul d'horloge pour les expressions. On retrouve pour les variables, paires, projections, déclarations et définitions composées les règles classiques de ML. Une

constante est typée selon son annotation (règle CONST). Le type d'horloges de e `when` ce est comme attendu celui de e composé avec ce (règle WHEN). La fusion de deux flots étant la contrepartie binaire du filtrage, la règle MERGE est symétrique. L'opérateur `sync` reçoit un couple de flots dont les rafales sont de longueur égale et produit un flot de couples par rafales de même longueur (règle SYNC).

Un buffer (règle BUFFER) est bien typé si le type de son entrée est sous-type du type de sa sortie. Nous lions la notion de sous-typage à la notion d'adaptabilité sur les horloges en remarquant qu'un type st est toujours de la forme α `on` stc_1 `on` $\dots$ `on` stc_n. On appelle α sa racine et $stc_1, \dots stc_n$ ses conditions. Les conditions de Core Lucy-n sont des mots ultimement périodiques connues. Leur composition peut donc être calculée. On aboutit à la définition suivante : deux types st et st' sont sous-types (st **<:** st') s'ils ont la même racine et que la composée des conditions de st est adaptable à la composée des conditions de st'. Les définitions ci-dessous expriment formellement cette relation à l'aide de la fonction $Factor(st)$ qui scinde un type d'horloges st en deux : sa variable de type d'une part, la composée de ses conditions d'autre part.

$$
\begin{array}{lcl}
Factor(\alpha) & \triangleq & (\alpha, (1)) \\
Factor(st \texttt{ on } stc) & \triangleq & (\alpha, w \ on \ w') \text{ avec } \left\{ \begin{array}{lcl} (\alpha, w) & = & Factor(st) \\ w' & = & unpack^{\#}([\![stc]\!]^S) \end{array} \right.
\end{array}
$$

$$
st \mathbf{<:} st' \quad \triangleq \quad \alpha = \alpha' \wedge w <: w' \text{ avec } \left\{ \begin{array}{lcl} (\alpha, w) & = & Factor(st) \\ (\alpha', w') & = & Factor(st') \end{array} \right.
$$

La propriété 3 justifie la décidabilité de cette relation. Par exemple, si $st \triangleq \alpha$ `on` (2) `on` $(1\ 0)$ et $st' \triangleq \alpha$ `on` $(0\ 2)$, on a st **<:** st' car $Factor(st) = (\alpha, (1))$, $Factor(st') = (\alpha, (0\ 2))$ et $(1) <: (0\ 2)$.

Le jugement de typage $\Gamma \vdash def :: \Gamma'$ indique que la définition def typée dans l'environnement Γ renvoie un environnement étendu Γ'. Les règles de typage des définitions sont :

$$
\text{NODE} \quad \frac{\Gamma + [x \mapsto st] \vdash e :: st' \qquad FV(\Gamma) = \emptyset}{\Gamma \vdash \texttt{let node } f^{\alpha}\ x \texttt{ = } e :: \Gamma + [f \mapsto Gen(st \to st', \alpha)]}
\qquad
\text{DEFS} \quad \frac{\Gamma \vdash def_1 :: \Gamma' \qquad \Gamma' \vdash def_2 :: \Gamma''}{\Gamma \vdash def_1; def_2 :: \Gamma''}
$$

Les applications et définitions de nœud fonctionnent comme dans un système à la ML. Les fonctions Gen et $Inst$ permettent de passer d'un type flèche à un schéma de type (règle NODE) et vice-versa (règle APP). Ces fonctions sont définies ci-dessous. La gestion des variables libres et liées étant sans surprises ici, on ne définit pas formellement l'ensemble $FV(ty)$ des variables libres d'un type ty ni l'opération de substitution d'un type ty à une variable de type α dans un type ty' ($ty[ty'/\alpha]$).

$$
\begin{array}{lcl}
Gen(st \to st', \alpha) & \triangleq & \forall \alpha.\ st \ \to \ st' \text{ si } \{\alpha\} = FV(st) \cup FV(st') \\
Inst(\forall \alpha.\ st \ \to \ st',\ st'') & \triangleq & st[st''/\alpha], st'[st''/\alpha]
\end{array}
$$

Notons une particularité : dans un lambda-calcul polymorphe, on doit prendre garde à ne pas généraliser de variable de types libre dans l'environnement de typage. Ici, comme l'environnement ne contient que des types clos après la généralisation, toute variable peut toujours être généralisée.

4.2. Sûreté du typage

Le système de types présenté assure intuitivement la cohérence de la gestion des rafales vis-à-vis de la sémantique de Kahn originale. Plus précisément, pour un programme bien typé, les sémantiques de Kahn et synchrones "coïncident". Nous allons nous concentrer sur les seules règles spécifiques du système de types, c'est à dire WHEN, MERGE, SYNC et BUFFER. Les autres sont essentiellement identiques à celles de ML. Prouver la correction du calcul d'horloge exige d'étudier le comportement des fonctions correspondantes lorsqu'on les applique à des flots finis.

Propriété des combinateurs

Propriété 4. *Appliquée à deux flots finis de même longueur, la fonction when$^\#$ produit un flot fini dont la taille correspond au nombre de booléens à vrai dans sont flot droit :*

$$\forall xs \in S^*, cs \in \mathbb{B}^*, |xs| = |cs| \Rightarrow |when^\#(xs, cs)| = |cs|_{\mathbf{1}}$$

De plus, la fonction when$^\#$ commute avec la concaténation sous les mêmes conditions :

$$\begin{array}{l}\forall xs, xs' \in S^*, cs, cs' \in \mathbb{B}^*, |xs| = |cs| \wedge |xs'| = |cs'| \Rightarrow \\ \quad when^\#(xs.xs', cs.cs') = when^\#(xs, cs).when^\#(xs', cs')\end{array}$$

Propriété 5. *Appliquée à trois flots finis dont le second (resp. troisième) est d'une longueur égale au nombre de booléens à vrai (resp. faux) dans le premier, la fonction merge$^\#$ produit un flot fini de longueur égale à celle de son premier argument :*

$$\forall cs \in \mathbb{B}^*, xs, ys \in S^*, |xs| = |cs|_{\mathbf{1}} \wedge |ys| = |cs|_{\mathbf{0}} \Rightarrow |merge^\#(cs, xs, ys)| = |cs|$$

De plus, elle commute avec la concaténation sous les mêmes conditions :

$$\begin{array}{l}\forall xs, xs', ys, ys' \in S^*, cs, cs' \in \mathbb{B}^*, |xs| = |cs|_{\mathbf{1}} \wedge |ys| = |cs|_{\mathbf{0}} \wedge |xs'| = |cs'|_{\mathbf{1}} \wedge |ys'| = |cs'|_{\mathbf{0}} \Rightarrow \\ \quad merge^\#(cs.cs', xs.xs', ys.ys') = merge^\#(cs, xs, ys).merge^\#(cs', xs', ys')\end{array}$$

Propriété 6. *Appliquée à deux flots finis de même longueur, la fonction sync$^\#$ produit un flot fini de longueur identique :*

$$\forall xs, ys \in S^*, |xs| = |ys| \Rightarrow |sync^\#(xs, ys)| = |xs|$$

De plus, elle commute avec la concaténation sous les mêmes conditions :

$$\begin{array}{l}\forall xs, xs' \in S^*, ys, ys' \in \mathbb{B}^*, |xs| = |ys| \wedge |xs'| = |ys'| \Rightarrow \\ \quad sync^\#(xs.xs', ys.ys') = sync^\#(xs, ys).sync^\#(xs', ys')\end{array}$$

Propriété 7. *Si l'horloge de l'entrée d'un buffer est adaptable à son horloge de production, cette dernière est identique à l'horloge de sa sortie.*

$$\forall xs \in S^*, w \in \mathbb{N}^\omega, clock^\#(xs) <: w \Rightarrow clock^\#(buffer^\#(w, xs)) = w$$

De plus, la sortie d'un buffer est toujours un préfixe de son entrée :

$$\forall xs \in S^*, w \in \mathbb{N}^\omega, unpack^\#(buffer^\#(w, xs)) \sqsubseteq unpack^\#(xs)$$

Propriétés du calcul d'horloge Un système de types ordinaire approxime des ensembles de valeurs (objets sémantiques) par leurs types (objets syntaxiques). Le calcul d'horloge présenté approxime les horloges par des types ; il est naturel de se demander si cette approximation est correcte. Une première idée est de vérifier que l'horloge de la sémantique de toute expression bien typée est égale à la sémantique de son type d'horloges : informellement, si e est de type st dans Γ alors pour tout environnement σ "raisonnable" (respectant Γ), la relation $clock^\#([\![e]\!]^S(\sigma)) = [\![st]\!]^S(\sigma)$ est vérifiée. Cependant, la présence de points fixes dans le langage affaiblit cette égalité. Par exemple, soit l'expression close $e =$ `x where rec x = x` et un environnement σ tel que $\sigma_{st}(\alpha) = (1)$. Selon le calcul d'horloge, e admet n'importe quel type d'horloges, et en particulier $\vdash e :: \alpha$, pourtant $clock^\#([\![e]\!]^S(\sigma)) = clock^\#(fix_{V_S}(\lambda x.x)) = clock^\#(\varepsilon) = (0) \neq [\![\alpha]\!]^S(\sigma) = \sigma_{st}(\alpha) = (1)$.

Le type d'horloges d'une expression est donc une surapproximation. Étant données deux horloges w et w', on dit que w approxime w' (noté $w \ \leq_{ck} \ w'$) si w et w' sont égales pour toujours ou bien jusqu'à ce que w contienne une infinité de zéros :

$$\begin{array}{lll}(0) & \leq_{ck} & w \\ n.w & \leq_{ck} & n.w' \text{ pour tout } n, w, w' \text{ tels que } w \ \leq_{ck} \ w'\end{array}$$

On peut maintenant définir la relation $\sigma \models_\tau x$ qui signifie informellement que dans l'environnement synchrone σ, l'horloge de l'objet (valeur synchrone ou fonction) x est approximée par le type composé ou schéma de type τ. Cette relation est définie inductivement sur les types composés :

$$\begin{array}{lcl}
\sigma \models_{st} r & \triangleq & clock^{\#}(r) \leq_{ck} [\![st]\!]^S(\sigma) \\
\sigma \models_{ty \times ty} (r, r') & \triangleq & \sigma \models_{ty} r \wedge \sigma \models_{ty'} r' \\
\sigma \models_{\forall \alpha.\ ty \to ty'} f & \triangleq & \forall w, r.\ (\sigma' \models_{ty} r) \Rightarrow (\sigma' \models_{ty'} f\ w\ r) \\
 & & \text{avec } \sigma' = \sigma + [\alpha \mapsto w]
\end{array}$$

Un environnement de typage Γ et un environnement synchrone σ sont dits *cohérents* (noté $\Gamma \perp \sigma$) si les horloges des valeurs et fonctions contenues dans σ correspondent à la sémantique des types et schémas de types des valeurs et nœuds dans Γ. En d'autres termes :

$$\Gamma \perp \sigma \quad \triangleq \quad \forall (x, ty) \in \Gamma,\ \sigma \models_{ty} \sigma(x) \ \wedge\ \forall (f, ty_{sch}) \in \Gamma,\ \sigma \models_{ty_{sch}} \sigma(f)$$

Ces définitions permettent de définir la première propriété du système de types.

Théorème 1 (Cohérence)**.** *L'horloge d'une expression bien typée est décrite par son type.*

$$\forall e, ty, \Gamma, \sigma,\ (\Gamma \perp \sigma \wedge \Gamma \vdash e :: ty) \Rightarrow \sigma \models_{ty} [\![e]\!]^S(\sigma)$$

Enfin, on définit formellement la notion d'équivalence entre les deux sémantiques. Intuitivement, un flot de rafales $s \in (S^*)^\infty$ et un flot de scalaires $s \in S^\infty$ sont équivalents si en aplatissant les rafales du premier on obtient un préfixe du second. La relation $\lhd$ étend cette relation aux types composés et schémas de types :

$$\begin{array}{lcl}
r \lhd_{st} s & \triangleq & unpack^{\#}(r) \sqsubseteq s \\
(r, s) \lhd_{(ty,ty')} (r', s') & \triangleq & r \lhd_{ty} s \wedge r' \lhd_{ty'} s' \\
f \lhd_{\forall \alpha_1, \ldots, \alpha_n.\ ty \to ty'} g & \triangleq & \forall w_1, \ldots, w_n, x_S, x_K,\ (x_S \lhd_{ty} x_K \Rightarrow f\ w_1\ \ldots\ w_n\ x_S \lhd_{ty'} g\ x_K)
\end{array}$$

Nous voulons maintenant énoncer formellement la correction du calcul d'horloge. La relation $\lhd$ doit pour cela être étendue aux environnements. Un environnements de Kahn et un environnement synchrone définissant les mêmes variables sont cohérents (pour $\lhd$) lorsqu'ils s'entendent sur les valeurs des variables par rapport à un environnement de typage Γ donné :

$$\sigma_S \approx_\Gamma \sigma_K \quad \triangleq \quad \forall (x, ty) \in \Gamma,\ \sigma_S(x) \lhd_{ty} \sigma_K(x)$$

Nous pouvons maintenant formaliser la définition de la correction du typage donnée plus haut.

Théorème 2 (Correction)**.** *La sémantique synchrone d'une expression bien typée approxime sa sémantique de Kahn :*

$$\forall e, ty, \Gamma, \sigma_S, \sigma_K,\ (\Gamma \vdash e :: ty \wedge \sigma_S \approx_\Gamma \sigma_K) \Rightarrow [\![e]\!]^S(\sigma_S) \lhd_{ty} [\![e]\!]^K(\sigma_K)$$

Démonstration. La preuve se fait par induction sur les dérivations du calcul d'horloge. On esquisse quelques cas intéressants :

— Les sémantiques synchrones des opérateurs `when`, `merge` et `sync` ont la même forme. Par hypothèse d'induction, on doit prouver, pour $f \in \{when^{\#}, merge^{\#}, sync^{\#}\}$,

$$unpack^{\#}(map^{\#}_n(f, pack^{\#}(w_1, s_1), \ldots, pack^{\#}(w_n, s_n))) \sqsubseteq f\ s_1\ \ldots\ s_n$$

ce que le théorème 1 et les deuxièmes parties des propriétés 4, 5 et 6 justifient.

— La deuxième partie de la propriété 7 permet prouver la correction des buffers.

— La généralisation des variables de type est correcte grâce à la propriété 2 : pour prouver α `on` p `<:` α `on` p', vérifier $p <: p'$ suffit.

□

5. Perspectives d'implantation

Nous sommes en train de réaliser un prototype de compilateur pour un langage proche de Core Lucy-n. On discute ici de quelques aspects non discutés dans le reste de l'article mais nécessaires à la conception d'un compilateur complet.

Inférence d'horloge Le calcul d'horloge présenté en section 4 vérifie la cohérence du programme. Comme indiqué en section 2, notre prototype est capable d'inférer des horloges entières. Il utilise actuellement un algorithme développé pour les horloges binaires dans l'article [8], qui s'étend naturellement au cadre entier. Nous étudions actuellement des raffinements plus efficaces inspirés des techniques utilisées sur les *Synchronous Data-Flow* [7]. Par ailleurs, le prototype accepte des conditions arbitraires à la Lucid Synchrone et ne se restreint donc pas aux horloges ultimement périodiques.

Causalité et génération de code Les compilateurs pour langages synchrones emploient une analyse dite de *causalité* pour rejeter les programmes comportant des interblocages. Dans certains cas, cette analyse assure également que les équations du source peuvent être ordonnées statiquement sans introduire d'interblocage. Cela permet de générer du code impératif séquentiel.

Dans les langages synchrones fonctionnels, la causalité est une analyse de *productivité* : on cherche à s'assurer que chaque expression, définitions récursives comprises, produit des flots infinis. En pratique, on assure que la définition d'une variable ne dépend d'elle-même qu'à travers un délai. L'opérateur de délai est spécifique à chaque langage.

Dans notre cadre, cette notion est délicate. On peut, comme en Lucy-n [10], décider qu'un délai est un buffer n'ayant jamais besoin de son entrée à l'instant courant pour produire sa sortie. On appelle *adaptabilité stricte* ($\ll:$) la relation suivante entre les horloges d'entrée et sortie d'un buffer :

$$w \ll: w' \triangleq w <: w' \wedge \forall i \in \mathbb{N}, \mathcal{O}_w(i) \geq \mathcal{O}_{w'}(i+1)$$

Néanmoins, cette notion ne se comporte *a priori* pas bien vis-à-vis de la généralisation des variables de type. En effet, on souhaite avoir une propriété semblable à la propriété 2 pour l'adaptabilité stricte : si $w \ll: w'$ alors w'' `on` $w \ll: w''$ `on` w'. Malheureusement, cette propriété est fausse. Par exemple, on a $(1) \ll: 0(1)$ mais pas (2) `on` $(1) \ll: (2)$ `on` $0(1)$ car $(2) \not\ll: 1(2)$.

Pour vérifier nœud par nœud la causalité des programmes, plusieurs solutions sont envisageables. La première est de complexifier le système de types pour introduire de la quantification bornée. Tous les jugements α `on` *stc* $\ll:$ α `on` *stc*$'$ vérifiés localement ajoutent au type du nœud une contrainte sur α. Cette contrainte doit être telle que tout type la respectant ne falsifie pas le jugement initial. Cette contrainte dépendante de *stc* et *stc*$'$ reste à définir.

Une autre approche explorée par le prototype est d'exploiter un artefact du processus de génération de code. Pour présenter celle-ci, nous expliquons grossièrement le processus de génération de code.

Comme illustré en section 2, les compilateurs pour Lucid Synchrone et SCADE 6 génèrent pour chaque nœud une unique fonction de transition dans un langage en appel par valeur. Ce choix est un compromis : il permet une génération de code et analyse de causalité plus simples, mais rejette des programmes valides (comme `x where rec x = f x`, avec `f` un nœud dont la sortie ne dépend pas instantanément de l'entrée).

Nous avons choisi de suivre cette approche, et de fixer la taille des rafales en entrée et sortie d'une transition. Chaque rafale est représentée par un tableau. Par exemple, la fonction de transition générée depuis un nœud `f` de type $\forall\alpha.\ \alpha$ `on` $(3) \rightarrow \alpha$ `on` (2) reçoit et produit des tableaux de tailles respectives trois et deux. Cela implique qu'à chaque application de `f`, le compilateur génère du code de contrôle et convertisse les tableaux en entrée et sortie.

Par exemple, imaginons le code généré par un appel `y = f x`, avec `x` de type α' `on` (6) et `y` de type α' `on` (4). Cela correspond à instancier α avec α' `on` (2) dans le type de `f`. Cet appel est traduit

en une boucle de deux itérations autour d'un appel à la fonction de transition `f_step` de `f`, ainsi qu'en du code convertissant `x` et `y` en des tableaux de tailles attendues par `f_step`. Ici, il faut découper le tableau de taille six correspondant à `x` en deux tableaux de taille trois, et concaténer les deux sorties de taille deux en un tableau de taille quatre correspondant à `y`.

Cette méthode de génération de code esquive le problème de causalité évoqué précédemment. Intuitivement, le fait que w'' `on` $w \ll:$ w'' `on` w' ne dépende pas uniquement de w et w' provient du fait que w'' peut être "trop grand" ; générer le code de contrôle et conversion autour de la fonction de transition revient à ne jamais instancier l'horloge de base du buffer qu'avec (1) : on n'y produit ni ne consomme jamais de rafale d'une taille différente de celles induites par w ou w'. Notre compilateur se contente donc de l'adaptabilité stricte à la Lucy-n. Nous n'avons pas encore d'explication sémantique convaincante de ce phénomène.

Conclusion Nous avons proposé un langage qui étend les modèles synchrones et n-synchrones et brièvement décrit un embryon de compilateur. Formaliser la génération de code devrait aider à clarifier ses subtilités.

Remerciements Nous remercions Guillaume Baudart, Albert Cohen, Nhat Minh Lê, Marc Pouzet et Gabriel Scherer pour leur relecture.

Références

[1] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[2] D. Biernacki, J.-L. Colaço, G. Hamon, and M. Pouzet. Clock-directed modular code generation for synchronous data-flow languages. In *Proceedings of the conference on Languages, compilers, and tools for embedded systems*, 2008.

[3] P. Caspi and M. Pouzet. Réseaux de Kahn Synchrones. In *Journées Francophones des Langages Applicatifs*, 1996.

[4] A. Cohen, P. Dumont, M. Duranton, and M. Pouzet. Horloges entières. Draft, 2007.

[5] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. *N*-synchronous Kahn networks: a relaxed model of synchrony for real-time systems. In *ACM International Conference on Principles of Programming Languages*, 2006.

[6] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475. North Holland, Amsterdam, 1974.

[7] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, Jan. 1987.

[8] L. Mandel and F. Plateau. Typage des horloges périodiques en Lucy-n. In *Vingt deuxièmes Journées Francophones des Langages Applicatifs*, 2011.

[9] L. Mandel, F. Plateau, and M. Pouzet. Lucy-n : une extension n-synchrone de Lustre. In *Vingt et unièmes Journées Francophones des Langages Applicatifs*, 2010.

[10] F. Plateau. *Modèle n-synchrone pour la programmation de réseaux de Kahn à mémoire bornée*. PhD thesis, Université Paris-Sud 11, 2010.