Mémoire d'Habilitation à diriger des recherches

présenté à l'Université Paris 6

par

Marc Pouzet

Lucid Synchrone,

un langage synchrone d'ordre supérieur

Soutenu le 14 novembre 2002 devant la commission d'examen composée de :

MM	Jean-Charles	POMEROL	Président
	Jean-Pierre	BANÂTRE	Rapporteurs
	Gérard	BERRY	
	John	RUSHBY	
	Paul	CASPI	Examinateurs
	Gilles	KAHN	
	Claude	KAISER	
	Jean-Jacques	LÉVY	

à Hélène, Antoine et Simon

Remerciements

C'était une véritable gageure d'arriver à synchroniser pour un jury ces huit personnalités de la communauté informatique, chacune ayant un emploi du temps surchargé. La soutenance de cette habilitation est un vivant exemple des miracles que peuvent accomplir les langages de programmation synchrone.

Jean-Charles Pomerol me fait l'honneur de présider cette habilitation. Je lui en suis extrêmement reconnaissant.

Je tiens à remercier Jean-Pierre Banâtre, Gérard Berry et John Rushby qui ont lu ces pages avec une grande attention. Mon travail de recherche étant à la frontière de plusieurs domaines, Jean-Pierre Banâtre a accepté de le juger avec le regard extérieur du spécialiste des langages de programmation. L'enthousiasme de Gérard Berry au cours de nos rencontres annuelles à "Synchron" et la lecture de ses travaux ont été des stimulants essentiels. John Rushby a accepté de traverser l'Atlantique pour assister à la soutenance et juger ce travail avec le point de vue du spécialiste de la certification de programmes.

Les langages synchrones flot de données doivent beaucoup aux travaux fondateurs de Gilles Kahn sur la sémantique des réseaux de processus. Je n'aurais pu imaginer plus belle récompense que sa participation à ce jury.

Claude Kaiser me fait l'honneur d'examiner ce travail et je l'en remercie.

Jean-Jacques Lévy a dirigé mà thèse au sein du groupe Para à l'INRIA Rocquencourt entre 1990 et 1994 et j'ai eu le plaisir de travailler à nouveau avec lui cette année. Sa grande culture scientifique, sa rigueur et son esprit critique restent pour moi un exemple à suivre.

Le travail décrit dans ce document a débuté lors d'une rencontre fortuite avec Paul Caspi en octobre 94, à la suite de ma thèse. A cette époque, Paul Caspi étudiait les liens entre les langages synchrones et la programmation fonctionnelle typée et nous avons développé ensemble les fondements de Lucid Synchrone. Peu de gens ont eu la chance de travailler avec lui et je mesure la chance que j'ai d'être parmi ceux-là. L'originalité de ses idées, sa curiosité et son immense culture scientifique m'ont profondément influencé.

Je ne pourrais terminer ces remerciements sans un mot spécial pour Thérèse Hardin, responsable du thème SPI au LIP6. Bien que nos sujets de recherche soient différents, nos discussions scientifiques ont été nombreuses et intenses au cours de ces années. Sa grande exigence scientifique a toujours été une source de remise en question essentielle et bénéfique pour moi. Son enthousiasme pour mon travail et ses conseils m'ont été très précieux. Et son effet stimulant m'ont permis d'aller au bout de la réalisation de ce document...

Lucid Synchrone est une oeuvre collective qui s'est enrichie du travail des nombreux étudiants de notre équipe. Je remercie plus particulièrement Pascal Cuoq et Grégoire Hamon pour leur travail de thèse ; Sylvain Boulmé pour son enthousiasme pour le langage et en particulier ses horloges et enfin Jean-Louis Colaço avec lequel j'ai eu beaucoup de plaisir à travailler et qui m'a apporté l'ouverture vers le monde industriel.

Merci aussi aux amis Grenoblois, Jean-Claude Fernandez, Alain Girault, Nicolas Halbwachs, Fabienne Lagnier, Florence Maraninchi et Pascal Raymond.

Je tiens à remercier tous les membres du thème SPI du LIP6 pour l'ambiance chaleureuse qui règne dans l'équipe. Je remercie particulièrement notre ingénieur système Jean-Michel Frenod qui a la lourde tache de faire fonctionner les machines dans un environnement souvent récalcitrant et notre secrétaire David Massot pour son sens de l'organisation.

A Antoine et Simon qui donnent un sens à tout ça. A Hélène, enfin, pour le bonheur d'être ensemble.

Avant propos

Ce mémoire est une présentation de mes travaux de recherche effectués entre 1995 et 2001 sur Lucid Synchrone, un langage synchrone dédié à la programmation de systèmes réactifs. Les résultats de ces travaux ont été publiés dans [19, 20, 21, 48, 28, 25, 22, 93]. Les documents relatifs à cette habilitation sont accessibles à l'adresse : `www-spi.lip6.fr/~pouzet/HDR`.

Ayant commencé une activité nouvelle en 1995, ce document n'aborde pas mon activité antérieuse effectuée durant ma thèse [88] sous la direction de Jean-Jacques Lévy. Durant cette période je me suis intéressé aux techniques de compilation et d'ordonnancement cyclique pour les architectures parallèles dites VLIW (les ancêtres des processeurs super-scalaires actuels). Les travaux sur le sujet ont été publiés dans [89, 90, 92, 91].

Ce document présente de manière générale la conception, la sémantique et l'implantation de Lucid Synchrone, une extension du langage synchrone flot de données Lustre avec des traits présents dans les langages fonctionnels typés de la famille ML.

Lucid Synchrone est un langage dédié à la programmation de systèmes réactifs. Il s'agit d'un langage typé qui intègre des traits d'ordre supérieur et qui est basé sur le paradigme des flots synchrones. Comme Lustre, le langage repose sur la notion d'*horloge* permettant de spécifier des rythmes différents dans un système réactif. En Lucid Synchrone, ces horloges sont synthétisées par un calcul, appelé *calcul d'horloges*, décrit sous forme d'un système de types. Des analyses statiques permettent de rejeter les programmes mal définis (e.g. boucles instantanées, erreurs d'initialisation). Enfin, le langage est construit au dessus de Ocaml servant à la fois de langage hôte pour importer des primitives et de langage cible à la compilation.

Le travail sur ce langage a débuté en 1995 en collaboration avec Paul Caspi. A cette époque, Paul Caspi avait jeté les premières bases d'une extension de Lustre avec des traits fonctionnels. Ensemble, nous avons développé les fondements du langage. Le langage s'est ensuite développé à Paris 6. Il a donné lieu à l'encadrement de la thèse de Grégoire Hamon (LIP6) et de Pascal Cuoq (co-encadré par Michel Mauny du projet Cristal à l'INRIA Rocquencourt et financé par une cti CNET). Les soutenances sont prévues pour l'automne 2002.

On ne trouvera pas de justifications techniques des résultats énoncés dans ce document. Ce mémoire est essentiellement un résumé accompagnant les articles donnés en annexe. Certaines parties, cependant, n'ont jamais été publiées. Il ne s'agit pas non plus d'une présentation complète de ce qu'est Lucid Synchrone aujourd'hui. J'ai souhaité profiter de l'occasion de cette rédaction pour donner une vision d'ensemble de mon activité durant cette période, montrer différentes étapes de développement du langage, les choix de conception effectués, les extensions progressives des premiers formalismes et les perspectives de ce travail.

J'ai découvert les langages synchrones et plus particulièrement Lustre en 1995 lors de mon arrivée à VERIMAG. J'ai été séduit par l'élégance et la simplicité de ce langage. Il y a peu de concepts et de constructions en Lustre mais ceux-ci sont simples et puissants : principe de substitution, programmation par invariants et horloges. Cette simplicité est une vertu précieuse pour un langage de programmation, surtout lorsqu'il est dédié à la programmation de systèmes temps-réel critiques. En concevant Lucid Synchrone, j'ai souhaité que les caractéristiques de Lustre soient conservées.

Lucid Synchrone n'est pas une extension d'un langage fonctionnel (e.g. Ocaml) avec des constructions synchrones venant de Lustre. Ainsi, il n'est pas possible d'écrire directement en Lucid Synchrone des programmes Ocaml. La sémantique statique et dynamique de Lucid Synchrone n'est pas celle de Ocaml et le langage reste dédié à la programmation de systèmes réactifs. Il ne s'agit pas non plus d'une autre version de Lustre basée sur une syntaxe différente et utilisant Ocaml (plutôt que C) comme langage hôte pour importer des primitives ou pour sa génération de code. Lucid Synchrone est un sous-ensemble d'un langage fonctionnel et qui, comme Lustre, est basé sur le paradigme des flots synchrones. Il est fondé sur une sémantique, des analyses statiques et des techniques de compilations dédiées aux constructions synchrones du langage. Lustre étant lui-même fonctionnel, il était naturel d'étudier son extension avec des traits venant de la famille ML. Nous avons donc intégré certains de ces traits (e.g. synthèse des types, polymorphisme, ordre supérieur, types de données) lorqu'ils ne remettaient pas en cause les caractéristiques temps-réel du langage.

Chapitre 1

Introduction

1.1 Systèmes réactifs

De nombreux objets de notre vie courante sont aujourd'hui truffés de processeurs ou *calculateurs* [1]. Ils interviennent dans des secteurs aussi différents que l'automobile (e.g., clef électronique, gestion de l'allumage, assistance au freinage), le transport ferroviaire ou aérien (e.g., gestion des aiguillages de trains, commandes de vol, pilotage automatique) et l'électroménager grand public (e.g., machine à laver, hi-fi, vidéo) et ils sont bien sûr au coeur d'objets nouveaux (e.g., téléphonie mobile, jeux vidéo).

Peu coûteux et surtout programmables, ces calculateurs remplacent désormais des automatismes qui étaient réalisés de manière mécanique, électrique ou électronique. Cette transition s'est effectuée d'abord dans le milieu industriel : passage de systèmes à relais pour la gestion des aiguillages des voies ferrées à des systèmes électroniques puis logiciels, passage d'une gestion hydro-mécanique des gouvernes d'un avion à une gestion électrique, puis analogique, numérique et maintenant logicielle [8]. Elle atteint aujourd'hui les objets les plus anodins : remplacement du *programmateur* mécanique d'une machine à laver par un processeur programmable.

La caractéristique commune de ces systèmes informatiques est leur liaison à un environnement analogique, au moyen de capteurs et d'actionneurs avec lequel ils interagissent de manière permanente. Ces systèmes sont ainsi appelés systèmes *réactifs* ou *temps-réel* (pour reprendre la terminologie introduite dans [50]) car ils ont une interaction continuelle avec leur environnement extérieur et surtout ils doivent réagir au rythme imposé par cet environnement. Par exemple, il va de soi qu'un système logiciel de gestion du freinage d'urgence dans une voiture est soumis aux lois de la physique et qu'il doit *réagir* instantanément lors du blocage d'une roue.

> *Dans un système réactif, le temps est imposé par l'environnement et l'environnement ne peut attendre.*

Les systèmes réactifs se distinguent donc des systèmes que l'on trouve traditionnellement sur un ordinateur (e.g., compilateurs, traitement de texte, logiciel de calcul scientifique) et qui sont appelés systèmes *transformationnels*. De tels systèmes transformationnels ont par nature un comportement séquentiel : ils lisent une entrée, calculent un résultat et s'arrêtent.

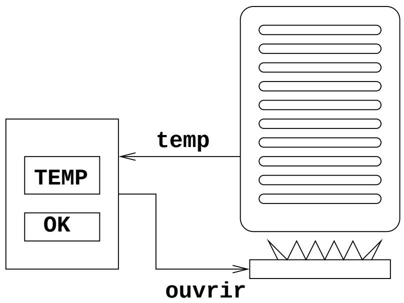
[1] Le terme *calculateur*, traduction fidèle de *computer*, désigne ici indifféremment un composant électronique ou logiciel.

Dans de tels systèmes, le temps n'intervient pas et n'a aucune influence sur le comportement du programme.

Les systèmes réactifs se distinguent également des systèmes *interactifs* (appelés aussi systèmes *événementiels*) comme les systèmes de gestion de fenêtres graphiques ou plus généralement les systèmes d'exploitation. Ces systèmes ont certes une interaction continue avec leur environnement extérieur mais ils ne sont pas soumis à des contraintes fortes de temps de réponse : ils peuvent répondre à leur propre rythme suivant les ressources dont ils disposent. Et surtout, ce temps réel de réaction n'a pas (ou peu) d'influence sur le comportement général du système. Un système réactif, s'il sert à commander un système physique, est soumis à des contraintes de temps de réponse dures — *la physique n'attend pas !* — et le temps (réel) mis à répondre peut avoir des conséquences considérables sur le système physique commandé.

> *En pratique, il est nécessaire de connaître à priori le temps de calcul du système pour traiter des événements venant de l'environnement dans le cas le plus défavorable.*

Les systèmes réactifs sont par essence des systèmes *concurrents*. Ils travaillent en parallèle avec leur environnement qui peut être physique ou un autre calculateur, ils traitent leurs données au fur et à mesure qu'elles arrivent, ils retournent les résultats au fur et à mesure qu'ils les produisent et exécutent de fait, une boucle sans fin de lecture/traitement/émission. Par exemple, dans un système de contrôle/commande d'un chauffe-eau, le régulateur de température et la chaudière fonctionnent en parallèle et dépendent l'un de l'autre.



Le contrôleur reçoit une directive de l'usager lui demandant de maintenir la température de la chaudière dans un intervalle proche de `TEMP` et reçoit en permanence la température courante `temp` de la chaudière. Suivant la valeur de `temp`, il doit actionner (avec `ouvrir`) la commande de chauffage. Il indique que la température demandée est atteinte en allumant un voyant `OK`. Le régulateur et la chaudière forment un système en boucle : le calculateur surveille et, en même temps, commande la chaudière. L'ouverture du chauffage aura pour effet d'augmenter la température de la chaudière et la conséquence de cette augmentation sera d'inhiber les commandes de chauffage à venir.

> *Les systèmes réactifs sont des systèmes concurrents qui fonctionnent en parallèle avec leur environnement auquel ils sont fortement couplés.*

Les systèmes réactifs sont souvent des systèmes *critiques* dans lesquels un défaut (réponse erronée ou pas de réponse) peut avoir des conséquences tragiques (pour s'en convaincre, on peut remplacer la chaudière par un processus chimique explosif !). C'est pourquoi les systèmes réactifs requièrent des méthodes de conception rigoureuses, des langages ayant une sémantique précise dont les propriétés sont bien étudiées et des outils de vérification formelle.

```
initialise memory ;                initialise memory ;
foreach input event do             foreach clock tick do
        read other inputs ;                read (inputs) ;
        compute ;                          compute ;
        emit (outputs)                     emit (outputs)
done                               done
```

FIG. 1.1 – Fonctionnement événementiel versus périodique d'un système réactif

> *Les systèmes réactifs sont souvent des systèmes critiques dans lesquels la sûreté de fonctionnement est particulièrement importante.*

Finalement, du point de vue opérationnel, un système réactif se ramène à un modèle calculatoire très rudimentaire consistant à exécuter une boucle infinie d'acquisition des entrées/calcul élémentaire/émission d'un résultat. Ce fonctionnement prend typiquement deux formes possibles comme le montre la figure 1.1.

Dans un fonctionnement événementiel, le programme attend la présence d'un événement d'entrée (e.g., appui sur un bouton) puis effectue une itération. Chaque itération définit la *réaction* du système : lecture des autres entrées, calcul puis émission d'un résultat. Ce qui est essentiel est que le temps de calcul de cette réaction et la mémoire utilisée soient prévisibles statiquement et inférieurs au délai minimal entre deux occurrences d'un événement d'entrée [2]. En pratique, le fonctionnement d'un système réactif est souvent encore plus simple. Les entrées/sorties sont *échantillonnées*, c'est à dire lues/produites périodiquement. `tick` est donné par une horloge réelle périodique (e.g., la milliseconde) et bien sûr, le pas d'échantillonnage devra être inférieur au temps maximal de réaction du programme.

> *Du point de vue calculatoire, un système réactif est un automate d'état fini ayant un comportement cyclique : acquisition des entrées/calcul/émission des résultats.*

1.2 Programmation des systèmes réactifs

Dans cette section, nous recensons très brièvement les principales solutions adoptées pour la conception de systèmes réactifs. Nous les jugeons au regard des caractéristiques principales des systèmes réactifs : nécessité de pouvoir décrire précisément le *temps*; *concurrence* du système et besoin de *méthodes formelles*.

1.2.1 Les approches classiques

Les systèmes réactifs ont longtemps été implantés de manière matérielle, d'abord mécanique, électrique (e.g., systèmes à relais, systèmes analogiques) puis électronique (e.g., circuits spécialisés). Ces implantations étaient naturellement parallèles et donnaient une information précise sur les temps de réaction (donnés par la longueur du chemin critique). Dès l'arrivée des premiers processeurs programmables, les systèmes réactifs ont alors été programmés directement en assembleur, essentiellement pour son efficacité et le contrôle fin des ressources (temps et mémoire). Il s'agissait donc, grosso-modo, d'implanter la boucle de réaction donnée

[2] La quantité de mémoire doit, elle aussi, pouvoir être bornée pour éviter un arrêt du système dû à un dépassement de la quantité de mémoire disponible.

plus haut et donc d'une programmation de très bas niveau. Contrairement à une implantation électrique, l'ordonnancement des tâches à effectuer devait être prévu par le programmeur. Naturellement, cette programmation directe s'est ensuite effectuée dans des langages de plus haut niveau donnant cependant des informations fines sur les ressources utilisées (e.g., C).

Les systèmes réactifs peuvent être décrits dans des modèles de plus haut niveau tels que les automates finis (en fait, des machines de Mealy ou de Moore) [53] ou les réseaux de Petri. D'une part ces modèles offrent une précision sur les temps de calcul et une efficacité très grandes : une transition d'un automate décrit la réaction et le temps de réaction est donné simplement par le maximum des temps de calcul de chacune des transitions. D'autre part, il existe de nombreuses méthodes de vérification automatique de propriétés temporelles [95, 23, 32].

Cependant, il s'agit essentiellement de modèles de description à plat : il n'y a aucun dispositif de composition de programmes (hiérarchique ou parallèle) d'où leur difficulté à décrire des systèmes complexes (e.g., des systèmes réalistes peuvent contenir des millions d'états). Ces modèles se sont révélés être de bons modèles de bas niveau des systèmes réactifs mais certainement pas de bon langages pour programmer directement des systèmes réactifs.

1.2.2 L'approche système et l'approche langage

Les systèmes réactifs étant parallèles par nature, il était tentant d'utiliser les mécanismes liés à la concurrence dans les systèmes d'exploitation (e.g., processus concurrents, communication et synchronisation, interruptions et non-déterminisme). Ces traits, largement étudiés dans les calculs de processus asynchrones [71, 72], sont utilisables directement dans la plupart des langages de programmation généralistes (e.g., C, Ada, Java, Ocaml). Ici, le parallélisme permet de décrire plus simplement et modulairement les composants du système réactif tout en profitant des structures de données et de contrôle du langage hôte.

Cette approche n'est cependant pas entièrement satisfaisante pour décrire des systèmes temps-réel. En effet, le non-déterminisme sous-jacent de la programmation asynchrone rend la conception et la mise au point des programmes délicate. La vérification formelle s'avère difficile en pratique à cause principalement de l'explosion combinatoire du système, due à la sémantique du parallélisme par entrelacement [3]. La concurrence, même lorsqu'elle est mise en oeuvre dans un système dédié, nécessite un mécanisme d'exécution coûteux dans lequel il est difficile de prédire le temps de réaction. Et plus ennuyeux, c'est un modèle dans lequel il n'y a pas de sémantique du temps. Ainsi, un programme contenant la séquence `wait 2 second; B.alarm` signifie seulement que `B` recevra `alarm` dans au moins 2 secondes s'il *écoute*. De même, `wait 2 second; wait 3 second` ne prend pas nécessairement le même temps que `wait 5 second`. Il n'y a pas de communication par diffusion (*broadcast*). Ainsi, dans `B.alarm; C.alarm`, les processus `B` et `C` ne reçoivent pas `alarm` au même instant d'où la difficulté de donner une vue commune de l'état global simultanément à tous les processus [4].

Finalement, il y a peu de chance que les programmes aient le même comportement d'une machine à une autre car il dépend fortement des ressources et de l'ordonnanceur de la machine.

[3] La composition parallèle $p||q$ de deux processus décrit trois comportements : p puis q, q puis p et p et q simultanément d'où la croissance exponentielle du nombre d'états possibles du système par rapport à la taille des processus mis en parallèle. Les méthodes de vérification de modèles (*Model-checking*) basées sur une énumération des états du système se révèlent donc vite impuissantes.

[4] Le temps entre l'émission de `B` et `C` peut durer arbitrairement longtemps, suivant les ressources de la machine au moment de l'exécution du programme.

1.2.3 L'approche synchrone

Dans les années 80, une approche nouvelle appelée *approche synchrone* a été proposée pour programmer les systèmes réactifs [10, 41]. Elle est basée sur un modèle idéal où les communications et les calculs sont supposés prendre un temps nul. Dans ce modèle, la sortie d'un calcul est effectuée dans le même temps que la lecture des entrées courantes. Le système évolue sur une échelle de temps discrète définie par la suite des réactions du système. Pour cette raison, on dira que la machine est *synchrone* avec son environnement. Suivant cette approche, un système réactif est décrit en :

- définissant d'abord les relations temporelles entre les évènements et les diverses réactions à ces événements. A ce niveau, le temps est vu de manière *logique*, comme la succession des réactions du système aux évènements d'entrée.
- en vérifiant, *a posteriori*, la correspondance entre le temps logique et le temps physique. Autrement dit, la machine est-elle suffisamment rapide pour tenir compte des contraintes temps-réel de l'environnement et justifier que l'on puisse négliger les temps de calcul lors de la modélisation ?

Cette approche a plusieurs conséquences : d'abord, elle permet de s'affranchir des contraintes de temps réel d'une architecture donnée, d'écrire donc des programmes portables et de faciliter le raisonnement sur ceux-ci. Par exemple, une expression `x + x` pourra être remplacée par `2*x` qui lui est fonctionnellement équivalente et qui n'a pourtant pas le même temps d'exécution réel. Cette fois, `wait 2 second;B.alarm` signifiera que `B` reçoit le signal `alarm` *au même instant* que la deuxième occurrence du signal `second`. Ensuite, un système pourra être décrit de manière modulaire comme la composition parallèle de plusieurs sous-systèmes sans introduire pour autant du non déterminisme ou de temps : la mise en parallèle de $p||q$ signifie seulement que p et q s'exécutent simultanément [5] et ces processus pourront communiquer librement. Le non déterminisme dû au temps partagé disparaît puisque la machine est supposée disposer d'un nombre illimité de ressources. Ce parallélisme pourra alors être *compilé* et les programmes traduits en des programmes séquentiels très simples sur le modèle de la figure 1.1. Pour cette raison, on a souvent dit que les langages synchrones permettaient de réconcilier *parallélisme* et *déterminisme*.

L'apparente naïveté de cette approche — l'hypothèse de temps nul est-elle bien raisonnable ? — s'est révélée extraordinairement féconde dans le domaine de la programmation de systèmes réactifs. Bien entendu, cette approche n'interdit pas de modéliser les contraintes temps-réel d'un environnement donné [6].

Plusieurs langages de programmation sont fondés sur l'approche synchrone. ESTEREL [11] est basé sur un calcul de processus concurrents avec communications par diffusion et a donc, un parfum impératif ; les STATECHARTS [49], ARGOS [68] et SYNCCHARTS [4] sont basés sur un paradigme graphique de composition d'automates de Mealy. SIGNAL [9] et LUSTRE [43] trouvent leur origine dans l'automatique (*control theory*) et la programmation *flot de données* ou *data-flow* [5, 6, 59] et manipulent donc des *flots* ou *suites infinies* comme valeurs de base.

M'étant essentiellement intéressé aux langages synchrones basés sur le modèle flot de données, je termine cette introduction par une présentation de ce modèle et son plongement dans les langages fonctionnels généralistes. L'ouvrage de N. Halbwachs [41] donne une présentation générale des principaux langages synchrones. [42] donne une bibliographie plus récente des travaux menés autour des langages synchrones.

[5] Autrement dit, le processus $p||q$ effectue une transition si et seulement si p et q effectuent une transition.

[6] De même que le fait de négliger les frottements dans la mécanique classique n'empêche pas de les modéliser.

1.3 Le flot de données

Le modèle flot de données trouve son origine principalement dans les travaux de Gilles Kahn et ceux menés autour du langage LUCID.

1.3.1 Lucid et le modèle flot de données

Dans les années 70, Ashcroft et Wadge proposent le langage LUCID [7, 5] comme alternative aux langages de programmation existants. Leur objectif est de proposer un langage d'usage général basé sur le paradigme de la programmation par itération sur des flots. A la même époque, Gilles Kahn [59] s'intéresse aux réseaux de processus déterministes asynchrones communiquant par des files d'attente non bornées. Il montre que leur sémantique est donnée par la solution d'un système d'équations sur des flots. Cette correspondance entre processus déterministes asynchrones et systèmes d'équations sur des flots est connue sous le nom de *principe de Kahn*.

Ces travaux donnant naissance au modèle *flot de données* ou *data-flow* ont eu une influence importante dans plusieurs domaines de l'informatique : en programmation des machines parallèles [36], en architecture [52], en sémantique du parallélisme, en compilation [75] et ont largement inspiré les langages synchrones flot de données. Nous présentons donc rapidement ce modèle.

Le modèle classique de la programmation séquentielle est le modèle de Von-Neuman. Dans ce modèle, un programme est une suite d'actions modifiant un état global qui représente les valeurs courantes des données manipulées par le programme. L'opération de composition de programmes est fondamentalement la mise en séquence. Elle requiert la terminaison des programmes : `P;Q` n'a d'intérêt que si `P` s'arrête. Le modèle de Von-Neuman est donc bien adapté à la réalisation de systèmes transformationnels.

Dans le modèle flot de données, chaque variable est représentée par la suite des valeurs qu'elle est amenée à prendre pendant l'exécution, c'est à dire par son *histoire*. Sa valeur au cours du temps est définie par une équation pouvant faire intervenir les valeurs d'autres données. Un programme est donc constitué d'un système d'équations sur des suites. L'opération de composition se résume ici à la mise en parallèle d'équations et il est bien clair que l'ordre entre les équations est sans importance. Ainsi, le programme LUCID suivant :

```
t = x + 1
x = y + z
nat = (0 fby nat) + 1
```

définit les trois suites récurrentes $(t_i)_{i\in\mathbb{N}}$, $(x_i)_{i\in\mathbb{N}}$ et $(nat_i)_{i\in\mathbb{N}}$ telles que $t_i = x_i + 1$, $x_i = y_i + z_i$, $nat_i = nat_{i-1} + 1$ et $nat_0 = 0$. Les opérations classiques (e.g., la somme `+`) sont appliquées point à point. Certains opérateurs permettent de décrire un flot de manière itérative : x `fby` y qui se lit "x followed by y" est le flot dont le premier élément est le premier élément de x et le n-ième est le $(n-1)$-ième de y. Les équations de LUCID peuvent définir indifféremment des flots finis ou infinis. Ainsi, la suite `fact` définie dans l'exemple ci-dessous est une suite finie.

```
init counter = n
next counter = counter - 1
init fact = 1
next fact = counter * fact as soon as counter = 0
```

Le modèle flot de données est un modèle séduisant pour plusieurs raisons. C'est d'abord un modèle simple, proche de mathématiques élémentaires (suites récurrences) dans lequel seules les véritables dépendances — les dépendances entre les calculs — ont besoin d'être exprimées [7].

Les règles d'évolution du système sont locales : chaque équation définit l'évolution d'une cellule mémoire au cours du temps en fonction, et en fonction seulement, des cellules dont elle dépend. Dans un modèle séquentiel, au contraire, la mémoire évolue de manière globale et il est assez difficile d'identifier l'évolution d'une seule cellule au cours du temps. Les inconvénients de cette approche globale sont bien illustrés par la difficulté à retrouver un invariant (de boucle) dans un programme impératif. Dans le modèle flot de données, le programmeur écrit directement des invariants.

Un système d'équations peut à la fois être vu comme un programme et comme la spécification d'une propriété. La distinction entre programme et propriété est donc moins franche que dans les langages séquentiels habituels. On peut donc mélanger équations définissant des calculs et équations établissant des propriétés de ces calculs. Cette idée a été introduite par les concepteurs de LUCID et associée à des principes de preuve de programme [6]. Elle a été largement développée dans LUSTRE avec le principe des *observateurs synchrones* [44]. Enfin, cette description peut se faire de manière incrémentale en ajoutant/enlevant une équation.

Ce modèle flot de données est essentiellement celui de LUCID. La sémantique adoptée par les concepteurs de LUCID est une sémantique de Scott [8]. Ainsi, il est possible de définir un flot tel que "si la cinquième valeur est égale à `1` alors la première est égale à `0`". Donc la définition d'un flot n'est pas soumise à un principe de causalité qui dit grosso-modo qu'une action dans l'avenir ne peut interférer sur le passé. L'objectif des concepteurs de LUCID était de proposer un langage d'usage général basé sur les flots. La sémantique de Scott choisie pour LUCID et la difficulté de compiler efficacement les programmes expliquent certainement que le langage se soit peu répandu. Nous verrons plus loin que les langages synchrones flot de données, en se concentrant sur les programmes manipulant des flots infinis et en choisissant la sémantique de Kahn, ont réconcilié l'élégance de LUCID et l'efficacité d'implantations séquentielles.

1.3.2 Les réseaux de Kahn

Le modèle flot de données ne s'est pas révélé très convaincant pour la programmation de systèmes transformationnels, qui sont fondamentalement des programmes séquentiels qui terminent. En revanche, grâce aux travaux de Gilles Kahn [59], ce modèle permet de donner une sémantique simple à des systèmes de processus déterministes concurrents. Il se révèle être un bon modèle pour décrire des systèmes qui ne doivent pas s'arrêter et en particulier les systèmes réactifs.

Un réseau de Kahn est constitué d'un ensemble de processus séquentiels déterministes communiquant par envois de messages (figure 1.2). Chaque processus est écrit dans un langage séquentiel disposant de deux primitives de communication asynchrones (envoi et attente de message). La communication est réalisée par des files d'attente (buffers), de taille non bornée à l'avance. L'émission de message est non bloquante alors que l'attente de message est bloquante (lorsque la file est vide). Les canaux sont supposés fiables et le délai d'émission borné à l'avance.

[7] ou *true-dependences*, pour reprendre la terminologie classique en compilation [29].

[8] Un flot f à valeurs dans V est une fonction de $\mathbb{N}$ dans $V_\perp$, avec $V_\perp$ munis de l'ordre plat ($\perp$ étant le plus petit élément) et $f_1 \leq f_2$ ssi $f_1(x) \leq f_2(x)$.

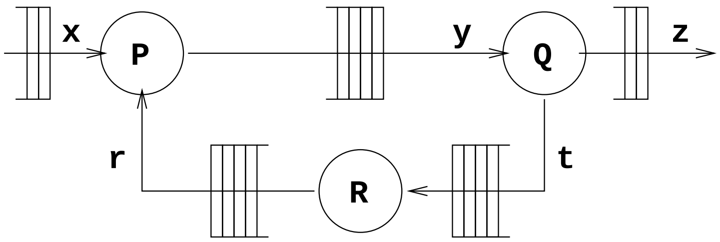


FIG. 1.2 – Un réseau de Kahn

Nous introduisons la définition de réseau de Kahn synchrone comme suit.

Définition 1 (Exécution synchrone d'un réseau de Kahn) *Un réseau de Kahn est synchrone s'il possède une exécution à buffer borné : il existe une constante $k \in \mathbb{N}$ telle que pour tout canal X et tout instant n, la file d'attente du canal X à l'instant n est de taille inférieure ou égale à k.*

Cette définition peut être simplifiée en imposant que les canaux contiennent au plus une valeur à chaque instant. Les circuits synchrones correspondent donc à des réseaux de Kahn où les canaux contiennent exactement une valeur à chaque instant [106].

Parce que le comportement d'un processus à un instant donné ne dépend pas seulement de la valeur de son entrée courante mais aussi des entrées qu'il a reçues jusque là, Kahn propose de décrire la sémantique d'un opérateur par une fonction de l'*histoire* de ses entrées vers l'*histoire* de ses sorties, autrement dit une fonction de *flots*. Cette proposition est connue sous le nom de *principe de Kahn*. Chaque canal de communication X définit alors le flot $X = x_1, ..., x_n, ...$ des valeurs passant sur ce canal et la sémantique d'un réseau de Kahn sera donnée par la solution d'un système d'équations entre ces flots. Pour décrire précisément la sémantique de ces réseaux, nous faisons les rappels suivants :

Soit D un ensemble (dont les éléments seront notés en minuscules). On note D^*, l'ensemble des suites finies d'éléments de D : $D^* = \cup_{n \in \mathbb{N}} D^n$ où D^n désigne les suites de longueur égale à n. D^∞ désigne l'ensemble des suites infinies d'éléments de D. On note $D^\omega = D^* \cup D^\infty$ (et ses éléments seront notés en majuscules).

D^ω contient la suite vide ϵ, $x.S$ désigne la suite dont le premier élément est x et le reste est S et $\bullet$ est la concaténation de suites. La longueur d'une suite S est notée $|S|$.

$(D^\omega, \leq)$ est muni de l'ordre préfixe $\leq$ ($X \leq Y$ s'il existe Z tel que $X \bullet Z = Y$). $(D^\omega, \leq)$ est un *ordre partiel complet* (cpo) : le plus petit élément est la suite ϵ. Avec l'ordre préfixe, toute partie dirigée est de la forme $P = \{S_1, ..., S_n, .../S_i \leq S_{i+1}\})$ et elle admet un sup dans D^ω qui est $\cup_i S_i$.

Une fonction $f : D_1^\omega \rightarrow D_2^\omega$ est *monotone* si et seulement si $\forall X, Y, X \leq Y \Rightarrow f(X) \leq f(Y)$ et elle est *continue* si et seulement si pour toute partie dirigée $P = \{S_0, ...|S_i \leq S_{i+1}\}$, $sup(f(P)) = f(sup(P))$.

D'après le *principe de Kahn*, un processus à une entrée et une sortie d'un réseau de Kahn est interprété par une fonction continue f de l'*histoire* de son entrée $X = x_1, ..., x_n, ...$ (suite des valeurs que l'on peut observer). Sa sortie est une suite Y définie par $Y = f(X)$. Ainsi, un canal de type D est interprété par son histoire ($\in D^\omega$). Avec l'ordre préfixe sur les séquences, la *monotonie* correspond au fait que les sorties sont produites progressivement, au fur et à mesure que les entrées sont lues. La lecture d'une nouvelle entrée ne remet pas en cause ce qui a été produit jusque là. La *continuité* renforce la propriété de monotonie en interdisant des définitions telles que ($f(X) = \epsilon$ pour tout $X \in D^*$ et $f(X) = X$ si $X \in D^\infty$).

Cette interprétation d'un processus à une entrée et une sortie se prolonge simplement à des processus à plusieurs entrées et plusieurs sorties de sorte que la sémantique du réseau est donnée par la plus petite solution d'un système d'équations :

$$\begin{aligned}&Y_1 = f_1(X_1, ..., X_n)\\&...\\&Y_p = f_p(X_1, ..., X_n)\end{aligned}$$

Cette solution existe d'après le théorème de Kleene et peut être obtenue par itérations successives (à partir de ϵ pour chaque canal défini récursivement). Ainsi, un réseau de Kahn définit encore une fonction continue de ses entrées et peut être composée arbitrairement avec d'autres réseaux.

Dans la suite nous identifierons un réseau de Kahn avec le système d'équations associé (le réseau n'est qu'une représentation graphique).

De très nombreux travaux ont été menés sur les réseaux de Kahn : ordonnancement [62, 81, 30], ajout d'opérateurs non déterministes et étude de leur expressivité [80]. Les réseaux de Kahn ont naturellement un pouvoir expressif limité : pas de non-déterminisme, pas de test à l'absence, nombre d'opérateurs et connections entièrement déterminées statiquement, etc. Cette limitation s'est cependant révélée être un atout pour les concepteurs de systèmes réactifs. Les réseaux de Kahn n'ont pas la complexité des modèles asynchrones généraux et correspondent bien aux habitudes des ingénieurs du domaine.

1.3.3 Propriétés abstraites et concrètes des réseaux de Kahn

Nous définissons maintenant certaines familles de fonctions de flots. Si X est un flot alors $X(n)$ désigne sa valeur à l'instant n et $X[1..n]$ désigne le préfixe de longueur n de X. La forme générale d'une fonction de flot est donnée par l'équation $Y(n) = f(X[1..g_n])(n)$ qui dit que la n-ième valeur du flot $f(X)$ est obtenue à partir du préfixe $X[1..g_n]$ de X.

Définition 2 (Fonctions combinatoires) *Les fonctions combinatoires sont des fonctions de flot dont la valeur courante de la sortie Y ne dépend que de la valeur courante des entrées X_i. Ces fonctions sont aussi dites sans mémoire :*

> $f : A_1^\omega \times \ldots \times A_k^\omega \to B^\omega$ *est une fonction combinatoire si pour tout n,*
> $Y(n) = f(X_1(n), \ldots, X_k(n))$

Remarque 1 Un réseau de Kahn formé seulement de fonctions combinatoires définit une fonction combinatoire de ses entrées. A chaque instant n, tout canal du réseau contient exactement une valeur scalaire. □

Définition 3 (Fonctions préservant les longueurs) *Une fonction préserve les longueurs lorsqu'elle émet autant de sorties qu'elle reçoit d'entrées.*

> *Soit f telle que pour tout n, $Y(n) = (f(X[1..g_n]))(n)$. f préserve les longueurs si et seulement si pour tout n, $n = g_n$.*

Cette définition s'étend aux fonctions n-aires. Soit f telle que pour tout n, $Y(n) = (f(X_1[1..g_n^1], ..., X_k[1..g_n^k]))(n)$. f préserve les longueurs si et seulement si pour tout n, $n = g_n^1 = ... = g_n^k$.

Remarque 2 Un réseau de Kahn formé seulement de fonctions préservant les longueurs est synchrone. □

Remarque 3 Une fonction causale [106] ne dépend que du passé de ses entrées (et éventuellement de tout son passé). Une fonction causale est continue et préserve les longueurs. Les fonctions causales sont aussi appelées *synchrones* dans [67]. □

Le caractère causal ou combinatoire d'une fonction est une propriété *abstraite* car il ne dit rien sur l'aspect calculatoire de la fonction (e.g., le temps de calcul, la mémoire utilisée durant le calcul). On peut seulement les tester avec la sémantique de Kahn. Une propriété est *concrète* lorsqu'elle concerne l'aspect opérationnel des fonctions.

Pour qu'une fonction puisse dépendre du passé de ses entrées, il faut que le processus qui l'implante puisse stocker des valeurs dans des variables internes. Nous appelons *état* ou *état interne* ou *mémoire*, un ensemble de variables auxiliaires manipulées par le programme implantant une fonction. Soit $f : D_1^\infty \times ... \times D_n^\infty \to D^\infty$, une fonction causale et l'équation $Y = f(X_1, ..., X_k)$. Alors :

– f peut être implantée par deux fonctions concrètes f_v et f_m et un état initial M_0 tels que, pour tout n :

$$\begin{array}{l} Y(n) = f_v\, M_{n-1}\, X_1(n) \ldots X_k(n) \\ M(n) = f_m\, M_{n-1}\, X_1(n) \ldots X_k(n) \end{array}$$

où M_n désigne l'état interne à l'instant n. f peut donc être calculée de manière itérative (séquentielle) par une boucle.

```
(* initialisation de l'etat *)
M := M_0 ;
(* etat permanent *)
loop
  read (X1 ... Xk) ;
  Y := f_v M X1 ... Xk ;
  M := f_m M X1 ... Xk ;
  emit Y
end
```

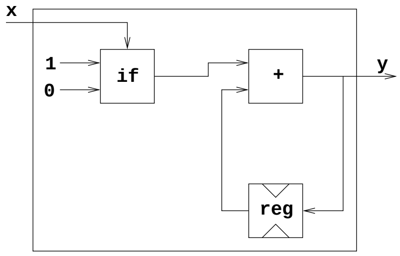
Nous verrons dans le chapitre 5, une manière systématique de produire le code séquentiel correspondant à certains réseaux de Kahn..

Remarque 4 La taille des états M_n (notée $|M_n|$ si besoin) n'est pas nécessairement bornée dans le cas où f dépend de tout son passé. Par exemple, la fonction f telle que $f(X[1..n])$ est égal à l'entier le plus proche de $X(n)$ parmi $X[1..(n-1)]$ est une fonction préservant les longueurs. Elle fait référence à toute l'histoire de X. □

Exemple 1 Une fonction de décalage (ou *registre*) telle que $\texttt{reg}(X) = 0.X$ est une fonction causale. On a $\texttt{reg}(X)(1) = 0$ et $\texttt{reg}(X)(n) = X(n-1)$.

De plus, cette fonction est *temps-réel* car son résultat courant peut être calculé en un temps borné à priori et utilisant une quantité de mémoire également bornée à priori. □

Un réseau de Kahn est *cyclique* lorsque certaines sorties sont utilisées comme entrées.



$$\begin{array}{rcl} Z & = & \texttt{if } X \texttt{ then } 1 \texttt{ else } 0 \\ Y & = & Z + \texttt{reg}(Y) \end{array}$$

FIG. 1.3 – Un réseau de Kahn cyclique

Exemple 2 La suite :

$$\begin{array}{rcl} Y(1) & = & \texttt{if } X(1) \texttt{ then } 1 \texttt{ else } 0 \\ Y(n) & = & (\texttt{if } X(n) \texttt{ then } 1 \texttt{ else } 0) + Y(n-1) \end{array}$$

compte le nombre de fois où $X(n)$ est vrai. Le réseau associé est cyclique. Il est représenté dans la figure 1.3. □

Naturellement, les noeuds d'un réseau de Kahn ne préservent pas nécessairement les longueurs.

Définition 4 (Echantillonnage, sur-échantillonnage) *Une fonction est une fonction d'é-*chantillonnage *lorsqu'elle produit moins de sorties qu'elle n'a d'entrées, elle est une fonction de* sur-échantillonnage *lorsqu'elle produit plus de sorties qu'elle n'a d'entrées.*

Soit $f : A^\omega \rightarrow B^\omega$ et l'équation $Y = f(X)$. Pour tout n, il existe g_n tel que $Y(n) = f(X[1..g_n])$.
- *f est une fonction d'échantillonnage lorsque pour tout n, $n \leq g_n$*
- *f est une fonction de sur-échantillonnage lorsque pour tout n, $n \geq g_n$*

Exemple 3 La fonction `tl` qui oublie sa première entrée ($\texttt{tl}(x.s) = s$) ne préserve pas les longueurs. La fonction de bégaiement `2` telle que $\texttt{2}(\epsilon) = \epsilon$ et $\texttt{2}(x.s) = x.x.\texttt{2}(s)$ est une fonction de sur-échantillonnage. Elle calcule la suite $(x_{\lfloor n/2 \rfloor})_{n \in I\!N}$. La fonction `1/2` telle que $\texttt{1/2}(\epsilon) = \epsilon$ et $\texttt{1/2}(x.y.s) = x.\texttt{1/2}(s)$ qui oublie une entrée sur deux est une fonction d'échantillonnage. Elle calcule la sous-suite $(x_{2n})_{n \in I\!N}$. □

Une fonction ne préservant pas les longueurs telle que `tl` peut aussi s'écrire : $\texttt{tl}(x.S) = \epsilon \bullet S$. Ainsi, dans une équation $Y = \texttt{tl}(X)$, le canal Y contient ϵ au premier instant puis $X(n)$ aux suivants.

Définition 5 (Fonctions à mémoire bornée) *Une fonction f est dite à mémoire bornée lorsque sa sortie peut être calculée par une fonction ne dépendant que d'une information finie sur le passé de ses entrées. Autrement dit, en reprenant les notations précédentes, il existe deux fonctions f_v et f_m et un état initial M_0 de taille finie tels que pour tout n, $|M_n| = |M_0|$.*

Cette définition n'est pas entièrement satisfaisante car elle caractérise une fonction abstraite par une propriété de l'une de ses implantations possibles. Pour être plus précis, il faut expliciter la notion d'état, ce que nous ferons au chapitre 5.

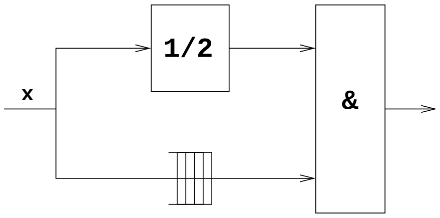


FIG. 1.4 – Un réseau de Kahn non synchrone

Exemple 4 La propriété d'exécution synchrone n'est pas stable par composition dès lors que l'on compose des fonctions ne préservant pas les longeurs [18]. Le réseau donné dans la figure 1.4 et calculant la suite $(x_{2n} \& x_n)_{n \in I\!N}$ où & est le "et" logique appliqué point à point n'admet pas d'exécution synchrone alors qu'il est formé uniquement de fonctions synchrones. Les deux flots n'avançant pas au même rythme, à l'instant n, $n/2$ valeurs de x seront accumulées le long d'un canal. □

1.3.4 Langages fonctionnels paresseux

Aujourd'hui, des langages fonctionnels paresseux tels que Haskell [31] permettent de décrire très facilement des structures infinies telles que les flots, en les exprimant comme des types abstraits. Les systèmes d'équations sur les flots peuvent être écrits directement et être exécutés. Ces langages fournissent des mécanismes d'abstraction puissants comme la curryfication et l'ordre supérieur.

Les langages paresseux souffrent cependant d'une certaine inefficacité. Pour y remédier, Wadler [108] proposa de nouvelles techniques de compilation telles que "l'exécution sans liste". L'idée est d'éviter de construire des listes intermédiaires dans les programmes manipulant des listes, dès lors que les éléments de ces listes peuvent être consommés au moment où ils sont fabriqués. Plus tard, cette technique fut généralisée aux structures récursives quelconques et appellé "deforestation" [109]. Les techniques de deforestation se sont ensuite développées [38, 61, 102] au point d'être intégrées au compilateur de HASKELL [37].

Nous reviendrons dans le chapitre 3 sur l'utilisation des langages paresseux pour traiter les flots.

1.4 Le flot de données synchrone réactif

Dans les années 70-80, les ingénieurs en automatique ou en traitement du signal ont dû faire face au passage de systèmes analogiques à des calculateur numériques. Ils avaient l'habitude de décrire leurs sytèmes par des équations récurrentes sur des flots (des signaux), formalisme naturel pour raisonner sur leurs systèmes et trouvaient difficile de les traduire sous forme de programmes séquentiels impératifs. Certains proposèrent donc des méthodes de traduction automatique de ces équations sous forme de programmes impératifs. Par exemple, certains des programmes de l'Airbus A320 ont été obtenus de cette manière. Ceci a conduit les chercheurs à proposer des langages spécialisés (LUSTRE, SIGNAL), des boites à outils (PTOLEMY) associés à des compilateurs dédiés [9, 43, 62] : ces langages furent appelés *langages data-flow synchrones*.

Le langage LUSTRE [9] repose sur l'idée de programmer directement à partir d'équations sur des flots proches des programmes LUCID mais en adoptant une sémantique de Kahn. Cependant, seul un petit sous-ensemble des réseaux de Kahn est autorisé :

- un programme LUSTRE définit un réseau de Kahn éventuellement cyclique mais *statique* (composé d'un nombre fini d'opérateurs et de connections). Ce programme est un système d'équations de flots récursifs mais il n'y a pas de fonctions récursives [10].
- dans un programme LUSTRE, le temps logique est représenté naturellement par la succession des valeurs d'un flot et son interprétation synchrone est claire : si f est une fonction de flots (qui préserve les longueurs), la valeur à l'instant n de la sortie de f est *synchrone* avec les valeurs de ses entrées à l'instant n. Le caractère synchrone des programmes a aussi une interprétation en terme de réseau : le réseau de Kahn possède une exécution à buffer borné (et même nulle !). Ainsi, le réseau donné dans la figure 1.4 n'est pas synchrone et sera rejeté statiquement.

Les compilateurs des langages synchrones ont donc développé des méthodes permettant de ne conserver que les réseaux de Kahn synchrones et temps réel. Ceci est obtenu en :

- définissant une sémantique opérationnelle synchrone ne conservant que les programmes pouvant s'exécuter de manière synchrone ;
- fournissant une analyse statique rejetant les réseaux flot de données ne pouvant s'exécuter de cette manière. Cette analyse est souvent appelée *calcul d'horloge* ou *vérification de consistance* ;
- en rejetant les programmes ne pouvant être calculés séquentiellement. Cette analyse est appelée *analyse de causalité* ;
- et finalement, en fournissant des méthodes de compilation produisant des programmes séquentiels pour les programmes ayant passé les vérifications précédentes.

Les compilateurs des langages synchrones sont assez proches des méthodes d'élimination des listes de Wadler, i.e., ils permettent de représenter un flot paresseux d'un type donné par une seule valeur de ce type.

1.5 Le flot de données synchrone

Les langages data-flow synchrones sont limités au domaine de la programmation des systèmes temps-réel et leur pouvoir d'abstraction est réduit. Ils n'autorisent pas la récursion ou la pleine fonctionnalité et leurs types de données sont limités à des types élémentaires ou des types abstraits.

Les raisons de ces restrictions paraissent assez claires : les systèmes temps-réel doivent interagir continûment avec leur environnement et ceci peut être garanti si leur réaction peut s'implanter en utilisant une mémoire et un temps de réaction finis. Ces restrictions peuvent donc être vues comme des conditions suffisantes pour garantir la réactivité.

Nous nous sommes demandés si les techniques développées jusque là pour les langages synchrones (e.g., analyses statiques, méthodes de compilation) ne pouvaient pas être généralisées pour être appliquées à des langages data-flow plus généraux ayant des mécanismes d'abstraction, d'application de fonction et de récursivité. Cela permettrait d'atteindre les deux objectifs suivants :

[9] Le nom LUSTRE vient de LUcid Synchrone et Temps-REel.

[10] Les réseaux de Kahn peuvent être récursifs (e.g., un réseau de calcul des nombres premiers par la méthode du crible d'Eratosthene) mais il n'y a pas de création dynamique de connections.

– enrichir le pouvoir expressif des langages réactifs, en particulier avec des traits présents dans les langages fonctionnels : polymorphisme de type et synthèse des types, types de données et définitions par cas, ordre supérieur, etc. Cela permettrait de répondre aux besoins exprimés par les utilisateurs traditionnels des langages synchrones et en particulier de Scade, la version industrielle de Lustre.
– trouver les restrictions à la programmation data-flow générale permettant de bénéficier des méthodes d'analyse et de compilation efficace développées pour le synchrone. Ceci permettrait de proposer un nouveau paradigme de programmation efficace, basé sur l'itération sur les flots, pour décrire des systèmes dynamiques.

Nous avons remarqué que ces deux objectifs pouvaient être atteints en définissant un langage de flot synchrone fonctionnel et en caractérisant son sous-ensemble temps-réel à la compilation. Il en résulte un langage strictement plus expressif que Lustre mais gardant ses caractéristiques pour la programmation de systèmes temps-réels.

1.5.1 Lucid Synchrone

Nous avons donc conçu un langage fonctionnel de flots synchrones, appelé Lucid Synchrone, dont la restriction au premier ordre est largement inspirée de Lustre. Celui-ci étant un langage fonctionnel, il est apparu intéressant de construire Lucid Synchrone au dessus d'un langage fonctionnel existant afin de faciliter les échanges entre les deux langages. Nous avons choisi Ocaml [63] comme langage cible du compilateur avec l'idée suivante : le langage cible fournira un modèle d'évaluation strict — donc efficace — pour programmer les calculs effectués dans l'instant alors que le langage source nous fournira des objets synchrones — donc efficaces — pour parler du temps.

Dans le chapitre 2, nous commençons par un bref aperçu de la version actuelle du langage. Une présentation plus détaillée est disponible dans [93]. Le langage est disponible à l'adresse : `www-spi.lip6.fr/lucid-synchrone`.

Le chapitre 3 présente la sémantique de Lucid Synchrone (une partie de la sémantique présentée ici est publiée dans [19, 20]). Nous montrons d'abord comment *simuler* les langages flot de données synchrones et Lucid Synchrone en particulier en quelques lignes de Haskell. Nous donnons ensuite une sémantique synchrone sous forme de règles SOS (*Stuctural Operational Semantics*). Celle-ci est fortement inspirée des règles de sémantique comportementale d'Esterel mais en restant dans un cadre flot de données. Cette sémantique est présentée en détail dans la thèse de Grégoire Hamon que j'ai encadrée [46]. Dans cette sémantique, seuls les programmes synchrones peuvent être exécutés. Ainsi, comme dans le cas de Lustre, nous avons besoin d'un calcul d'horloge pour éliminer statiquement les programmes ne pouvant être exécutés de manière synchrone. Le calcul d'horloge est présenté dans le chapitre suivant (chapitre 4) et est exprimé sous forme d'un système de types dépendants. Ce système garantit l'exécution synchrone des programmes, i.e., *well clocked programs cannot go wrong !*. Ce calcul est plus général que celui de Lustre puisqu'il permet de synthétiser les horloges et est compatible avec l'ordre supérieur. Ce calcul a été publié dans [19, 20] et a été décrit dans Coq [26] par Sylvain Boulmé et Grégoire Hamon [13]. Nous montrons ensuite comment, à partir de ce calcul, dériver plusieurs implantations possibles. Nous terminons par un calcul simplifié proche d'un calcul de type *à la ML*. Ce système est basé sur le calcul de type dépendant original mais en remplaçant les dépendances par une forme affaiblie de types existentiels. Ce système est présenté en détail dans la thèse de Grégoire Hamon [46]. La fin du chapitre est consacrée à une présentation informelle d'autres analyses telles que l'analyse de causalité et l'analyse d'initialisation. Celles-ci ont été publiées dans dans [28, 25] et l'analyse de causalité est décrite en détail dans la thèse de Pascal Cuoq que j'ai co-encadrée [27].

Dans le chapitre 5, nous présentons une sémantique opérationnelle basée sur la théorie des co-algèbres [57] qui nous permet de transformer de manière systématique des définitions co-inductives en définitions co-itératives. Ceci nous permet de dériver une méthode de compilation vers du code impératif basée sur la co-itération. Ce travail a d'abord été publié dans [21]. [22] est l'article de référence de Lucid Synchrone. Dans la conclusion (chapitre 6), nous discutons des travaux reliés au notre et présentons les activités en cours et les perspectives de développement du langage.

Chapitre 2

Lucid-Synchrone

Ce chapitre est une présentation tutorielle des principaux traits de Lucid Synchrone. On ne peut parler de Lucid Synchrone sans parler de ses deux "parents" que sont Ocaml [63] d'une part et Lustre de l'autre. Le lecteur familier des langages fonctionnels à la ML peut le voir comme un (petit sous-ensemble) de Ocaml manipulant des flots synchrones. Le lecteur familier de Lustre préférera sans doute le voir comme une extension fonctionnelle de Lustre. Le langage ayant largement évolué depuis la première implantation de 1995, nous présentons ici la version courante (v2) [1]. Nous reviendrons dans le chapitre 4 sur les différences majeures avec les versions précédentes.

2.1 Un langage à la ML sur des flots

2.1.1 Les flots comme objets de base

Types de base étendus aux flots

Lucid Synchrone est un langage de flots de données et à ce titre il ne permet de manipuler que des flots, ou suites infinies de valeurs. Ainsi, parler d'un objet x revient en fait à parler de la suite infinie $x_0, x_1, x_2, ...$ des valeurs successives de x. De même, la constante `1`, ne désignera pas la valeur scalaire entière 1, mais la suite infinie $1, 1, 1, ...$. Toutes les constantes des types de base (e.g. entiers, booléens, flottants, caractères, chaînes de caractères) sont ainsi étendues en flots.

Toute valeur étant un flot, on ne parlera pas, par exemple d'un flot d'entiers, mais simplement d'un entier. La suite $1, 1, ...$ sera simplement désignée par `1`.

Application point-à-point des opérations de base

Les opérations classiques sur des valeurs scalaires sont également étendues implicitement aux flots et appliquées point-à-point. Par exemple :

`c`	t	f	t	f	f	...
`x`	x_0	x_1	x_2	x_3	x_4	...
`y`	y_0	y_1	y_2	y_3	y_4	...
`x + y`	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$	...
`if c then x else y`	x_0	y_1	x_2	y_3	y_4	...

[1]Le langage est accessible à l'adresse `www-spi.lip6.fr/lucid-synchrone`.

Tuples et flots

Il est possible de construire des tuples de flots et ces tuples peuvent encore être vus comme des flots. Ainsi :

`c`	t	f	t	f	f	...
`x`	x_0	x_1	x_2	x_3	x_4	...
`y`	y_0	y_1	y_2	y_3	y_4	...
`if c then (x,y) else (0,0)`	(x_0, y_0)	$(0,0)$	(x_2, y_2)	$(0,0)$	...	

***Commentaire* 1** *Le point de vue adopté ici sur la représentation des tuples est celui de* OCAML *qui distingue les valeurs de type suivants :*
- $T_1 \times (T_2 \times T_3)$
- $(T_1 \times T_2) \times T_3$
- $T_1 \times T_2 \times T_3$

En LUSTRE, *en revanche, les valeurs de ces trois types sont interchangeables (i.e., les valeurs sont considérées modulo associativité). Nous préférons la première approche, plus simple et compatible avec la synthèse de type et le polymorphisme.* □

***Commentaire* 2** *Le typage des tuples est un peu délicat si l'on veut conserver l'esprit flot de données où toute valeur est un flot (c'est à dire un objet qui peut évoluer au cours du temps). Normalement, il faudrait considérer les flots de couples et les couples de flots, leurs constructeurs/destructeurs associés et les fonctions de conversion d'une représentation à l'autre. Assez rapidement, cette programmation devient pénible et pousse à la schizophrénie. Les couples de flots peuvent être représentés par des flots de couples et nous gardons donc la deuxième représentation.*

Nous verrons dans le chapitre 4 les conséquences de ce choix sur les horloges. □

2.1.2 Opérations temporelles : décalages et initialisation

Délai initialisé

L'opération de décalage initialisé `fby` (ou *followed by*) est emprunté à LUCID [5], l'ancêtre des langages flot de données. Le comportement du décalage initialisé est donné dans le diagramme suivant. Au premier instant il retourne la première valeur de son premier argument puis retourne la valeur précédente de son deuxième argument.

x	x_0	x_1	x_2	x_3	x_4	...
y	y_0	y_1	y_2	y_3	y_4	...
x `fby` y	x_0	y_0	y_1	y_2	y_3	...

Délai unaire et initialisation

Le délai initialisé n'est cependant pas toujours très commode et pousse parfois à une écriture lourde et redondante. Ceci nous conduit à considérer deux nouveaux opérateurs : l'opérateur d'initialisation `->` et le délai non initialisé `pre`.

x	x_0	x_1	x_2	x_3	x_4	$\dots$
y	y_0	y_1	y_2	y_3	y_4	$\dots$
`pre` x	nil	y_0	y_1	y_2	y_3	$\dots$
`pre` $x+1$	nil	y_0+1	y_1+1	y_2+1	y_3+1	$\dots$
x `->` y	x_0	y_1	y_2	y_3	y_4	$\dots$
x `->` (`pre` y)	x_0	y_0	y_1	y_2	y_3	$\dots$

La valeur initiale de `pre` x est notée *nil*. Cette valeur représente l'indéfini et elle est absorbante. `fby` est un raccourci pour `->pre`. De plus, `->` et `pre` peuvent eux-mêmes être exprimés en fonction de `fby` :

$$\begin{array}{rcl} x \texttt{ fby } y & \equiv & x \texttt{ -> pre } y \\ \texttt{pre } x & \equiv & nil \texttt{ fby } x \\ x \texttt{ -> } y & \equiv & \texttt{if } (true \texttt{ fby } false) \texttt{ then } x \texttt{ else } y \end{array}$$

L'expression ($true$ `fby` $false$) n'étant vraie qu'au premier instant, la conditionnelle retourne la première valeur de x au premier instant, puis la valeur courante de y. Il est souvent plus naturel d'écrire les équations en séparant l'invariant (écrit en utilisant `pre`) de la phase d'initialisation. Ces deux nouvelles constructions ont donc été ajoutées au langage [2].

Malheureusement, ce confort a un prix : (`pre` x) peut introduire des valeurs indéfinies dans le programme et il faudra s'assurer que le comportement du programme n'en dépend pas. Pour cette raison, nous avons développé une analyse statique appelée *analyse d'initialisation* donnant des conditions suffisantes de bonne initialisation [25]. Elle est décrite dans le chapitre 4.

***Commentaire* 3** *La valeur (nil) n'est pas une valeur manipulée explicitement à l'exécution mais est représentée par une valeur quelconque (donnée par le compilateur). Il est donc essentiel de s'assurer que le comportement du programme ne dépend pas de cette valeur d'initialisation.* □

***Commentaire* 4** *On retrouve ici un sujet de discussion récurrent sur les variables dans les langages généralistes et la nécessité ou pas de forcer une initialisation au moment de leur définition. Les langages dits "impératifs" permettent de déclarer des variables impératives puis de les initialiser. C'est alors au programmeur (ou au compilateur) de s'assurer que ces variables ont bien été initialisées avant d'être lues. D'autres langages (e.g.,* OCAML*) imposent par construction que les variables définies soient initialisées. Cette approche, plus prudente, est aussi plus sûre. Elle n'est pourtant pas entièrement satisfaisante car il n'existe pas toujours de bonne valeur d'initialisation* [3]. □

2.1.3 Nommer une valeur

Déclarations globales

Un programme est composé d'une suite de déclarations de valeurs globales appelées *noeuds*. La construction `let` définit les valeurs non récursives globales alors que la construc-

[2] J'ai longuement hésité avant de sauter le pas mais il a fallu se rendre à l'évidence : l'utilisation de `pre` et `->` m'était beaucoup plus naturelle que l'utilisation seule de `fby`.

[3] Décider, par exemple, d'initialiser une matrice à zéro, faute d'autre valeur pertinente pour ensuite exécuter un code d'initialisation complexe n'est pas très élégant.

tions `let/rec` définit les valeurs globales récursives. Ces valeurs peuvent être des flots ou des fonctions. Ainsi, le calcul de la suite des entiers naturels s'écrit :

```
let rec nat = 0 -> pre nat + 1
```

`0`	0	0	0	0	0	0	...
`1`	1	1	1	1	1	1	...
`pre nat`	*nil*	0	1	2	3	4	...
`pre nat + 1`	*nil*	1	2	3	4	5	...
`0 -> pre nat + 1`	0	1	2	3	4	5	...

Une fonction calculant la somme de ses entrées s'écrit :

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

ou encore :

```
let sum x =
  let rec s = x + (0 fby s) in s
```

Un détecteur de front (montant ou descendant) s'écrit :

```
let edge c = c <> (c fby c)
```

`c`	c_0	c_0	c_1	c_1	c_2	c_3	...
`c fby c`	c_0	c_0	c_0	c_1	c_1	c_2	...
`edge c`	f	f	t	f	t	t	...

`edge` est une fonction globale de flots.

Pour les programmes que nous venons d'écrire, le compilateur synthétise les types et les horloges et l'on obtient le résultat :

```
node nat :  int
node nat :: 'a
node sum :  int -> int
node sum :: 'a -> 'a
node edge :  'a -> 'a
node edge :: 'a -> 'a
```

Pour chaque déclaration, le compilateur indique les types (`:`) et les horloges (`::`) synthétisés. L'horloge `'a` de la constante globale `nat` indique que `nat` peut être utilisé à n'importe quelle horloge. L'horloge `'a -> 'a` du noeud `edge` indique que la fonction préserve les longueurs : elle renvoie une valeur pour chaque entrée. Nous reviendrons sur les horloges par la suite.

***Commentaire* 5** *On peut remarquer que la fonction* `edge` *est polymorphe : elle peut être appliquée indifféremment à des flots d'entiers, de booléens, de couples, etc. On pourra écrire l'expression :*

```
edge (if c then (x,y) else (0,0))
```

C'est une différence essentielle avec Lustre *qui ne fournit pas de mécanisme pour écrire du code générique.* □

Définitions locales

Les variables peuvent être définies localement en utilisant une construction `let/in` ou `let rec/in`. De plus, les définitions peuvent être mutuellement récursives. Par exemple :

```
(* compter modulo n *)
let sample n =
  let rec cpt = n -> if ok then n else pre cpt - 1
  and ok = pre cpt = 1 in
  ok
```

définit localement deux flots `cpt` et `ok` dépendant l'un de l'autre.

***Commentaire* 6** *Les déclarations locales et globales sont celles que l'on peut faire en* OCAML. *En* LUSTRE, *seules les fonctions de flots peuvent être déclarées globalement et les déclarations locales ne peuvent être imbriquées.* □

2.1.4 Réinitialisation

LUCID SYNCHRONE permet de réinitialiser l'état interne d'une fonction de manière modulaire : tous les flots redémarrent avec leur valeur initiale. Par exemple :

```
let sum_every_c dx c =
  reset sum dx every c
```

réinitialise les états internes de `sum` à chaque instant où la condition `c` est vraie. Plus précisément, le flot interne `0 fby s` de `sum` recommence avec sa valeur initiale `0` à chaque instant où `c` est vraie.

```
node sum_every_c : int -> bool -> int
node sum_every_c :: 'a -> 'a -> 'a
```

La réinitialisation est une forme particulière d'application de fonction recevant un paramètre supplémentaire booléen.

2.2 Les horloges

Le mécanisme d'horloge permet de décrire des flots (et donc des communications) ayant des rythmes différents. Il permet de manipuler des processus évoluant en parallèle sur des rythmes différents et de les combiner.

L'horloge d'un flot est une séquence booléenne définissant les instants où ce flot est défini. Certaines opérations vont permettre de produire un flot plus lent (échantillonnage) ou plus rapide (sur-échantillonnage). L'information d'horloge est synthétisée par le compilateur par un mécanisme appelé *calcul d'horloge* décrit dans le chapitre 4.

2.2.1 Filtrer une valeur *(échantillonnage)*

`when` est une fonction d'échantillonnage permettant de faire communiquer des processus rapides avec des processus plus lents en échantillonnant un flot suivant une condition booléenne. Dans l'exemple ci-dessous, la valeur courante de (`x when c`) est présente lorsque `x` et `c` sont présents et que `c` est vrai. Si `x` et `c` sont sur l'horloge `a`, l'horloge de (`x when c`) sera notée (`a on c`).

`a`	t	t	t	t	...
`c`	f	t	f	t	...
`x`	x_0	x_1	x_2	x_3	...
`x when c`		x_1		x_3	...
`true on c`	f	t	f	t	...

Les horloges comme (`a on c`) donnent des informations sur le comportement temporel des programmes flot de données.

Maintenant la fonction `sum` peut être utilisée sur un rythme plus lent en échantillonnant son entrée :

```
let mywhen x y = x when y

let sampled_sum x y = sum(x when y)

node mywhen :  'a -> clock -> 'a
node mywhen :: 'a -> (_c0:'a) -> 'a on _c0
node sampled_sum :  int -> clock -> int
node sampled_sum :: 'a -> (_c0:'a) -> 'a on _c0
```

`sampled_sum` a une horloge fonctionnelle qui suit la structure de son type (de données). Cette horloge indique que pour toute horloge `'a`, si le premier argument de la fonction a l'horloge `'a` et le second argument nommé `_c0` a la même horloge `'a`, alors le résultat a l'horloge `'a on _c0` (les variables des fonctions sont renommées dans l'horloge pour éviter les conflits de noms). Une expression sur l'horloge `'a on _c0` est présente lorsque l'horloge `'a` est vraie et le flot booléen `_c0` est présent et vrai.

Maintenant, cette somme échantillonnée peut être instanciée avec une constante d'horloge. Par exemple :

```
(* 10hz *)
let clock ten_hz = sample 10
let sum_ten_hz dx = sampled_sum dx ten_hz

node ten_hz :  clock
node ten_hz :: 'a
node sum_ten_hz :  int -> int
node sum_ten_hz :: 'a -> 'a on ten_hz
```

Le mot clef `clock` permet d'introduire un nom d'horloge (ici `ten_hz`) à partir d'un flot booléen. Ce nom d'horloge peut alors être utilisé pour échantillonner un flot en utilisant `when`.

Commentaire 7 *Une horloge (cl* `on` *c) est donc bien formée lorsque c est un nom introduit par le mot clef* `clock`*. C'est une différence fondamentale avec les versions précédentes du compilateur (versions* `v1.x`*) où c pouvait être n'importe quelle expression booléenne (comme c'est le cas en* Lustre*). Seuls les flots booléens introduits avec une construction* `let/clock` *peuvent servir à filtrer un flot.*

Nous reviendrons dans le chapitre 4 sur cette restriction par rapport aux versions précédentes du compilateur. □

L'opérateur `whenot` est l'opérateur symétrique. e `whenot` c produit un flot échantillonné sur le flot `not` c.

2.2.2 Combiner deux valeurs *(sur-échantillonnage)*

`merge` permet à des processus lents de communiquer avec des processus plus rapides en combinant des sous-séquences en des séquences plus longues.

c	f	t	f	t	...
x		x_0		x_1	...
y	y_0		y_1		...
merge c x y	y_0	x_0	y_1	x_1	...

Ainsi, l'opérateur `merge` permet de définir une fonction permettant de stocker la valeur d'un signal entre deux occurrences de celui-ci. Cet opérateur correspond à l'opérateur `current` de Lustre.

```
let mymerge c x y = merge c x y
let hold ydef c x = y
    where rec y = merge c x ((ydef -> pre y) whenot c))

node mymerge :  clock -> 'a -> 'a -> 'a
node mymerge :: (_c0:'a) -> 'a on _c0 -> 'a on not _c0 -> 'a
node hold :  'a -> clock -> 'a -> 'a
node hold :: 'a -> (_c0:'a) -> 'a on _c0 -> 'a
```

Nous lui ajoutons une valeur par défaut `ydef` utilisée jusqu'au premier instant où `x` est présent.

La construction `merge` permet d'exprimer une forme simple de sur-échantillonnage, i.e., des fonctions dont la sortie est plus rapide que l'entrée. Elle permet d'écrire une fonction imposant une horloge lente à son entrée.

```
let clock half = h
let over x = hold 1 half x
let stuttering = over nat

node half :  clock
node half :: 'a
node over :  int -> int
node over :: 'a on half -> 'a
node stuttering :  int
node stuttering :: 'a
```

***Commentaire* 8** `merge` *se comporte très différemment de la conditionnelle* `if/then/else`. `merge` *compose deux flots complémentaires et a donc un aspect* paresseux. *C'est la version flot de donnée de la conditionnelle classique (paresseuse) des langages séquentiels. La conditionnelle* `if/then/else` *est un opérateur strict dont les arguments partagent la même horloge.*

Les horloges permettent d'exprimer des structures de contrôle. Le compilateur utilise les informations d'horloges pour générer du code impératif efficace de sorte qu'une expression est calculée seulement lorsque son horloge est vraie. □

2.2.3 Composition d'horloges

Les horloges peuvent être imbriquées. Considérons par exemple, la description d'une montre.

```
(* 1/60 *)
let clock sixty = sample 60

let hour_minute_second second =
  let minute = second when sixty in
  let hour = minute when sixty in
  hour,minute,second

node sixty :  clock
node sixty :: 'a
node hour_minute_second :  'a -> 'a * 'a * 'a
node hour_minute_second ::
  'a -> 'a on sixty on sixty * 'a on sixty * 'a
```

Un flot sur l'horloge `'a on sixty on sixty` est présent seulement un instant tous les 3600 instants.

2.2.4 Programmes non synchrones

Certains programmes ne sont pas bien typés du point de vue des horloges [4]. Le plus typique étant :

```
let rec half = true -> pre (not half)
let wrong_clocked x =
  let clock clk = half in
  x + (x when clk)
```

Ce programme ajoute le flot `x` au sous-flot `x` obtenu en filtrant une entrée sur deux. Ainsi, cette fonction devrait calculer la séquence $(x_n + x_{2n})_n$ qui ne peut clairement pas être exécutée en mémoire bornée comme nous l'avons vu dans le chapitre 1. Le résultat de la compilation est :

```
File "tutorial.ls", line 38, characters 7-17:
>  x + (x when clk)
>       ^^^^^^^^^^
This expression has clock 'b on ?_clk0,
but is used with clock 'b.
```

`?_clk0` définit le nom d'horloge introduit par la construction `clock`.

Le calcul d'horloge n'est bien entendu pas complet et il rejette de nombreux programmes pouvant être exécutés de manière synchrone. Le calcul d'horloge ayant évolué depuis les premières versions, je reviendrai sur ses limitations dans le chapitre 4.

[4] Le terme bien *cadencé* serait-il plus naturel ?

2.2.5 Horloges et opérations à mémoire

Il est important de comprendre l'influence des horloges sur les opérations à mémoire (e.g., `->`, `pre`) comme dans l'exemple suivant :

`c`	f	t	f	t	f	...
`1`	1	1	1	1	1	...
`sum 1`	1	2	3	4	5	...
`(sum 1) when c`		2		4		...
`1 when c`		1		1		...
`sum (1 when c)`		1		2		...
`x`	x_0	x_1	x_2	x_3	x_4	...
`x when c`		x_1		x_3		...
`pre x`	*nil*	x_0	x_1	x_2	x_3	...
`pre (x when c)`		*nil*		x_1		...
`(pre x) when c`		x_0		x_2		...

Ainsi, dès qu'une fonction f contient un opérateur à mémoire, échantillonner ses entrées n'est pas équivalent à échantillonner ses sorties.

2.3 Traits expérimentaux : horloges implicites et filtrage

Nous présentons ici deux extensions expérimentales. La première permet de retrouver la construction `current` de LUSTRE et une forme affaiblie de `default` de SIGNAL. La seconde permet de faire du filtrage sur les flots en le basant sur le mécanisme des horloges.

La construction `on` peut être utilisée comme constructeur, en argument d'une fonction. Ceci permet de retrouver le `current` de LUSTRE.

```
let current init (x on c) =
  let rec o = merge c x ((init fby o) whenot c) in o

node current :  'a -> 'a -> 'a
node current :: 'a -> 'a on _c0 -> 'a
```

`current` attend un argument `x` d'horloge `'a on c` pour rendre un flot sur l'horloge dite immédiatement plus rapide `'a` (pour reprendre la terminologie de LUSTRE). Cette opérateur retourne un flot qui vaut `x` lorsque `x` est présent et qui vaut la valeur précédente de `x` sinon. Il est initialisé avec la valeur `init` jusqu'à la première occurrence vraie de `c`.

***Commentaire* 9** *A la différence de* LUSTRE, *l'opérateur* `current` *attend un second argument* `init` *permettant de donner une valeur initiale au flot jusqu'à la première occurrence vraie de* `c`. *En* LUSTRE, *l'opérateur* `current` *est un opérateur primitif du langage et il est unaire. L'implantation actuelle de l'analyse d'initialisation de* LUCID SYNCHRONE *impose l'ajout d'une valeur d'initialisation. Cette contrainte pourrait être levée dans le futur.* □

De même, on peut programmer un opérateur de combinaison de deux flots complémentaires.

```
let default (x on c) y = merge c x y

node default :  'a -> 'a -> 'a
node default :: 'a on _c0 -> 'a on not _c0 -> 'a
```

***Commentaire* 10** *La construction* `on` *en paramètre permet donc d'éviter certaines des redondances qu'impose une programmation à base d'horloges (e.g., comme dans une expression de la forme* `merge clk (f (x when clk)) (g (x whenot clk))`*).*

Cette construction n'ajoute aucun pouvoir expressif supplémentaire : le code généré de la fonction `current`*, par exemple, se comporte exactement comme si l'utilisateur avait ajouté explicitement l'argument d'horloge.* □

Les types somme et les produits (e.g., enregistrements) peuvent être définis de manière similaire à OCAML et nous avons conservé la même syntaxe [5].

Nous les illustrons dans un codage "naif" des événements valués d'ESTEREL. Nous définissons :

```
type 'a event = Nothing | Just of 'a
let nothing = Nothing
let emit(x) = Just(x)
let present x = match x with Nothing -> false | _ -> true
let absent x = not (present x)
```

Les constructions `present/then` ou `present/else` de ESTEREL s'expriment alors comme des conditionnelles *à la* LUSTRE.

```
(* compte le nombre d'occurrences d'un evenement *)
let count e = o where
  rec o = if present e then 1 -> pre o else pre o
```

Le langage permet d'effectuer du filtrage avec une construction `match/with` *à la* OCAML mais celle-ci s'applique à des flots. Cette construction est une généralisation du `merge`, et suit donc les mêmes règles d'horloge.

Considérons l'exemple d'un programme qui compte le nombre de `Top` entre deux `Click`.

```
type mouse = Click | Top

(* count the number of top between two click *)
let count e =
  let rec o =
    0 -> match e with
      Top on clk -> ((pre o) when clk) + 1
    | Click -> 0 in
  o
```

[5] Dans le mode de compilation temps-réel, les définitions inductives sont cependant interdites.

Le filtrage compose plusieurs sous-suites complémentaires, autrement dit, des flots sur des horloges complémentaires. Le filtrage ci-dessus a deux branches. Chaque branche définit sa propre horloge, l'une désignant les instants où `e = Top` et l'autre désignant les instants où `e = Click`.

La branche `Top on clk -> ((pre o) when clk) + 1` indique qu'aux instants où la condition `e = Top` est vraie (`clk` étant l'horloge de la branche), le résultat est égal à la valeur précédente de `o` plus un. Aux instants où `e = Click`, le résultat est `0`.

Le diagramme de l'exécution est montré ci-dessous.

e	Click	Top	Top	Click	Top	Click	...
clk	f	t	t	f	t	f	...
Top on clk		Top	Top		Top		...
(pre o) when clk + 1		1	2		1		...
Click	Click			Click		Click	...
0	0			0		0	...
o	0	1	2	0	1	0	...

Ce programme peut aussi être écrit :

```
let count e =
  let rec o =
    0 -> match e, pre o with
      Top, o -> o + 1
    | _ -> 0 in
  o
```

Si l'on revient à la première forme pour la fonction `count`, on peut noter la différence entre une expression `(pre o) when clk` qui désigne "la valeur de `o` au dernier instant global" et `pre (o when clk)` qui désigne "la valeur de `o` au dernier instant où `clk` était vrai".

***Commentaire* 11** *La construction de filtrage ici est similaire à la construction* `merge`*. Elle bénéficie (et parfois souffre) des mêmes contraintes d'horloges. Ainsi, écrire simplement* `(pre o) + 1` *entraine une erreur d'horloge puisque la variable* `o` *est sur l'horloge de base et doit être filtrée sur l'horloge de la branche. Elle force donc à écrire un code parfois redondant (les variables libres, en particulier, doivent être filtrées explicitement en utilisant le* `when`*). Ceci a l'avantage de lever bien des ambiguïtés (e.g., est-ce la dernière valeur globale de* `e` *où la dernière valeur qu'avait* `e`*, la dernière fois que la branche était active ?).*

Bien entendu, à partir de cette construction, il est possible de proposer une construction simplifiée ayant un comportement de structure de bloc où les variables libres sont filtrées implicitement (une variable libre `x` *est remplacée implicitement par* `x when c` *où* `c` *est l'horloge de la branche).*

La construction de filtrage est une manière de programmer les modes *de Maraninchi et Rémond [65]. Un codage systématique des modes vers* LUCID SYNCHRONE *a été réalisé par Grégoire Hamon [48]. La compilation du filtrage produit du code efficace, en tous points comparable au code généré par la compilation des modes proposée dans [66] : seul le code de la branche active à un instant donné est exécuté.* □

***Commentaire* 12** *Il est facile de proposer une construction de filtrage basée sur la généralisation de la conditionnelle stricte* `if/then/else` *plutôt que* `merge` *(et les horloges). Il s'agit alors d'un opérateur strict où toutes les branches sont exécutées à chaque instant mais où tous les résultats sont ignorés sauf un.*

Il ne s'agit donc pas d'une vraie structure de contrôle (spécifiant du code à n'exécuter que de temps en temps). L'opération de filtrage basée sur les horloges présentée ici est une structure de contrôle au point que sa structure est préservée dans le code généré. □

2.4 Ordre supérieur

Le langage étant fonctionnel, les fonctions sont des objets de base qui peuvent être donnés en argument à des fonctions ou retournés par des fonctions. Par exemple, la fonction `it` est un itérateur série (i.e., elle itère une fonction sur un flot de valeurs).

```
let it f init x = y where
  rec y = f x (init fby y)

node it :: ('a -> 'b -> 'b) -> 'b -> 'a -> 'b
```

telle que :

```
let sum x = it (+) 0 x
```

SCADE propose une opération appelée *condition d'activation*. Cette opération est *cablée* dans le compilateur et nous montrons ici comment la programmer à partir des autres constructions du langage. Une condition d'activation prend une fonction `f`, une horloge `clk`, une valeur par défaut `default`, une entrée `input` et calcule `f(input when clk)`. Elle replace le résultat sur l'horloge de base.

```
let cond_act f clk default input =
  let rec o =
    merge clk (f (input when clk))
              ((default -> pre o) whenot clk) in
  o

node cond_act :  ('a -> 'b) -> clock -> 'b -> 'a -> 'b
node cond_act :: ('a on _c0 -> 'a on _c0) -> (_c0:'a)
                                         -> 'a -> 'a -> 'a
```

En utilisant la condition d'activation, on peut calculer la somme d'un flot en éliminant les entrées stables.

```
let stable_sum input =
  let clock c = input <> (input fby input) in
  cond_act sum c 0 input
```

Tous les programmes que nous avons définis sont des systèmes temps-réel : les programmes sont compilés vers du code impératif séquentiel répondant en temps et en mémoire bornés quelle que soit leur entrée.

***Commentaire* 13** *Le codage proposé ici pour la condition d'activation (à partir de* `merge` *et* `when`*) est celui qui a été retenu dans le compilateur* ReluC *de* Scade*.* Scade *ne disposant pas de traits d'ordre supérieur, son compilateur traduit une opération (*`cond_act f clk default input`*) où* `f` *est le nom d'un noeud par la définition que nous avons donnée.*

Le compilateur ReluC *est capable ensuite de produire du code impératif efficace* □

2.5 Programmes synchrones non temps-réel

Il est possible de décrire des programmes synchrones non temps-réel en utilisant la récursion. Considérons, par exemple, le célèbre crible d'*Eratosthene*.

```
let first x = v
  where rec v = x fby v

let rec sieve x =
  let clock filter = (x mod (first x))<> 0
  in  merge filter
            (sieve (x when filter))
            (true fby false)

let rec nat = 2 fby (nat + 1)

let clock sieve = sieve nat

let primes = nat when sieve

node first :  'a -> 'a
node first :: 'a -> 'a
node sieve :  int -> bool
node sieve :: 'a -> 'a
node nat :  int
node nat :: 'a
node primes :  int
node primes :: 'a on sieve
```

Ce programme n'est plus temps-réel puisque le temps et la mémoire pour répondre augmentent au cours du temps. Une nouvelle instance de `sieve` est créée à chaque fois qu'elle reçoit un nombre premier, c'est à dire un nombre multiple d'aucune entrée reçue jusque là. Ainsi, au n-ième instant, la mémoire utilisée est proportionnelle au nombre de nombres premiers compris dans $[2..n]$.

***Commentaire* 14** *Le compilateur propose un mode permettant de rejeter les programmes synchrones non temps-réel. Dans l'implantation actuelle, les programmes manipulant des valeurs dynamiques (e.g., types inductifs, fonctions récursives) sont alors interdits.* □

D'autres exemples sont présentés dans [93] et dans la distribution. Lucid Synchrone décrit donc des réseaux de Kahn synchrones (d'où le titre de l'article [20]) en reprenant la philosophie de Lucid et de Lustre où les valeurs de base sont des flots et en leur ajoutant certains traits fonctionnels de ML.

Lorsque les programmes ont passé les diverses analyses (e.g., typage, calcul d'horloges, causalité, initialisation), ils sont compilés en des programmes séquentiels impératifs traduits en Ocaml.

Chapitre 3

Sémantique

Dans ce chapitre, nous donnons deux sémantiques à Lucid Synchrone. La première est une sémantique dénotationnelle de flots (finis et infinis). Très simple, elle permet de décrire Lustre et ses extensions en quelques lignes de Haskell.

Cette représentation n'est cependant qu'une *simulation* qui nécessite un modèle d'exécution coûteux ne donnant aucune garantie sur les temps de réaction. Nous montrons donc comment donner une sémantique opérationnelle synchrone caractérisant les programmes pouvant être exécutés de manière synchrone.

La constatation que les programmes Lustre pouvaient être simulés en quelques lignes de Haskell a été faite dans l'article [20] dans lequel une sémantique opérationnelle synchrone était donnée. Nous revenons sur la formulation initialement proposée de cette sémantique synchrone et proposons une formulation plus naturelle et plus simple. Celle-ci n'a pas été publiée mais est présentée dans la thèse de Grégoire Hamon [46].

3.1 Sémantique flot de données

A des détails syntaxiques près, les programmes Lustre et Lucid Synchrone peuvent être entièrement écrits dans un langage fonctionnel auquel on ajoute un jeu de primitives de manipulation de flots [1].

Considérons un petit noyau ML auquel nous ajoutons des opérations de manipulation de flot. (`fby`) est une opération de décalage, (`when`) est un opérateur d'échantillonnage, (`merge`) est un opérateur de combinaison de flots. (`extend`) est un opérateur d'application point-à-point et (`const`) permet de transformer une valeur en un flot constant. v désigne une valeur immédiate (e.g., un booléen, un entier (i), un opérateur).

$$\begin{array}{lcl}
e & ::= & x \mid e(e) \mid \lambda x.e \mid \texttt{rec}\ x.e \\
 & & \mid \texttt{extend}\ e\ e \mid \texttt{const}\ e \\
 & & \mid \texttt{merge}\ e\ e\ e \mid e\ \texttt{fby}\ e \mid e\ \texttt{when}\ e \mid v \\
v & ::= & \texttt{true} \mid \texttt{false} \mid i \mid \ldots \\
 & & \quad \mid \texttt{+} \mid \ldots
\end{array}$$

La figure 3.1 donne une interprétation dans la sémantique de Kahn des opérations flot de données de ce noyau.

[1] Lustre se décrit dans le sous-ensemble de premier ordre de ce langage

$$
\begin{array}{lcl}
D^\omega & = & D^\infty + D^* \\
\\
\texttt{const}\ v & = & v.\texttt{const}\ v \\
\texttt{extend}\ \epsilon\ xs & = & \epsilon \\
\texttt{extend}\ xs\ \epsilon & = & \epsilon \\
\texttt{extend}\ (f.fs)\ (x.xs) & = & (f\ x).\texttt{extend}\ fs\ xs \\
\\
\epsilon\ \texttt{fby}\ y & = & \epsilon \\
x.xs\ \texttt{fby}\ ys & = & x.ys \\
\\
\epsilon\ \texttt{when}\ cs & = & \epsilon \\
xs\ \texttt{when}\ \epsilon & = & \epsilon \\
x.xs\ \texttt{when}\ \texttt{true}.cs & = & x.(xs\ \texttt{when}\ cs) \\
x.xs\ \texttt{when}\ \texttt{false}.cs & = & xs\ \texttt{when}\ cs \\
\\
\texttt{merge}\ \epsilon\ xs\ ys & = & \epsilon \\
\texttt{merge}\ cs\ \epsilon\ ys & = & \epsilon \\
\texttt{merge}\ cs\ xs\ \epsilon & = & \epsilon \\
\texttt{merge}\ \texttt{true}.cs\ x.xs\ ys & = & x.\texttt{merge}\ cs\ xs\ ys \\
\texttt{merge}\ \texttt{false}.cs\ xs\ y.ys & = & y.\texttt{merge}\ cs\ xs\ ys
\end{array}
$$

FIG. 3.1 – Sémantique de Kahn

- `const` permet de produire un flot constant à partir d'une valeur. Ainsi, `const 1` s'écrit simplement `1` en LUSTRE et en LUCID SYNCHRONE puisque toutes les valeurs importées du langage hôte sont implicitement transformées en des flots.
- `extend` est l'application point à point. Elle permet, par exemple, de "lifter" les opérations scalaires importées de OCAML. Ainsi, l'addition s'écrit [2].

  ```
  plus x y = extend (extend (const (λ x.x + y)) x y
  ```

- `when` et `merge` sont les opérations de filtrage et de composition de flots. L'opération `when` émet la valeur courante de sa première entrée lorsque la valeur courante de la condition est vraie et est silencieux sinon. La fonction `merge` émet la valeur courante de sa deuxième entrée si la valeur courante de sa première entrée est vraie. Elle émet la valeur courante de sa troisième entrée sinon.

Les opérations flot de données sont continues pour l'ordre préfixe. La sémantique dénotationnelle des autres constructions (abstraction, application, récursion) est classique et nous ne la donnons pas ici (voir, par exemple [74, 99]). La sémantique dénotationnelle d'un noyau de LUCID SYNCHRONE est donnée dans [25].

Ce petit noyau étant purement fonctionnel, il peut être simulé en quelques lignes de HASKELL [54] en représentant les flots par des listes. Le code est donné dans la figure 3.2. Le constructeur de listes en HASKELL est noté "`:`". Pour chaque définition, nous donnons le type synthétisé par HASKELL.

[2] Une fonction n-aire f peut être transformée en une fonction de flots en écrivant `lift_n f x_1...x_n = extend (extend ... (const f) x_1)...x_n`.

```
module Streams where
constant x = x : (constant x)
constant : : a -> [a]

extend (f : fs) (x : xs) = (f x) : (extend fs xs)
extend : : [a -> b] -> [a] -> [b]

(x : xs) `fby` y = x : y
fby : : [a] -> [a] -> [a]

(x : xs) `when` (True : cs) = (x : (xs `when` cs))
(x : xs) `when` (False : cs) = xs `when` cs
when : : [a] -> [Bool] -> [a]

merge (True : c) (x : xs) y = x : (merge c xs y)
merge (False : c) x (y : ys) = y : (merge c x ys)
merge : : [Bool] -> [a] -> [a] -> [a]
```

FIG. 3.2 – LUSTRE et LUCID SYNCHRONE en HASKELL

3.2 Sémantique flot de donnée synchrone

Ce codage des flots par des structures de données paresseuses permet de décrire des réseaux de Kahn quelconques. Comme le rappelle l'exemple de la figure 3.3, certains programmes ne peuvent s'exécuter de manière synchrone. L'exécution de ce programme HASKELL s'interrompt faute de mémoire suffisante (l'exception `Heap overflow` est levée). Ceci est dû à la présence des opérations ne préservant pas les longueurs (`when` et `merge`).

Plus grave, certains programmes synchrones conduisent également à des exécutions qui échouent car un nombre non borné de valeurs intermédiaires sont stockées. C'est le cas avec le programme (écrit dans la syntaxe LUCID SYNCHRONE) :

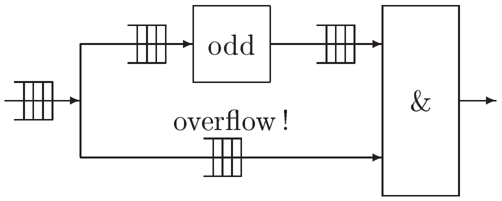
```
let main () =
    let rec nat = 0 -> pre nat + 1 in
    (merge false (nat when false) 0) + nat
```

Pour modéliser les exécutions synchrones, nous décrivons maintenant une sémantique flot de donnée de plus bas niveau où l'espace des valeurs instantanées est complété avec une valeur spéciale marquant l'absence. Cette valeur est notée *abs*. Ceci nous conduit à définir les flots infinis de valeurs et les flots complétés avec des valeurs marquant l'absence.

$$
\begin{array}{lcl}
\mathit{Stream}\,(A) & = & A.\mathit{Stream}\,(A) \\
\mathit{Value}\,(A) & = & \mathit{abs} + A \\
\mathit{Clocked\text{-}Stream}\,(A) & = & \mathit{Stream}\,(\mathit{Value}\,(A)) \\
\mathit{Clock} & = & \mathit{Stream}\,(\mathit{bool})
\end{array}
$$

Un flot (élément de *Clocked-Stream* (A)) est une suite de valeurs présentes ou absentes. L'horloge d'un flot est un flot de booléens (sans absence) indiquant les instants où une valeur est

```
let  odd x =
       let half = (const false) ‘fby‘ (not half) in
       x ‘when‘ half
in
       x & (odd x)
```



`x`	=	x_0	x_1	x_2	x_3	x_4	x_5	...
`half`	=	`false`	`true`	`false`	`true`	`false`	`true`	...
`x when half`	=		x_1		x_3		x_5	...
`x & (odd x)`	=		$x_0\&x_1$		$x_1\&x_3$		$x_2\&x_5$	...

FIG. 3.3 – Un exemple de programme non synchrone

présente :

$$\begin{array}{lcl} \texttt{clock}\ (abs.xs) & = & \texttt{false.clock}\ (xs) \\ \texttt{clock}\ (x.xs) & = & \texttt{true.clock}\ (xs) \end{array}$$

La sémantique modifiée est donnée dans la figure 3.4. Elle est construite ainsi :

Flots constants (`const`) : Cet opérateur est polymorphe puisqu'il doit produire (ou pas) suivant son environnement. Pour cette raison, nous lui ajoutons un argument supplémentaire d'horloge. Nous le notons `cl_const` et verrons plus tard comment synthétiser cet argument supplémentaire.

Application point à point (`extend`) : Nous devons décider ici ce qu'il faut faire en présence d'un opérateur binaire, lorsqu'un des arguments a une valeur et pas l'autre. Trois choix sont possibles :

1. stocker la valeur dans une variable d'état implantant un file (FIFO) jusqu'à ce que l'autre entrée soit présente,
2. générer une erreur à l'exécution,
3. ou rejeter statiquement cette situation.

Dans un contexte de programmation temps-réel — et rester donc cohérent avec LUSTRE — nous choisissons la troisième solution. Ainsi, l'opérateur `extend` attend deux flots de même horloge. Si ce n'est pas le cas, ce filtrage incomplet lèvera une erreur à l'exécution. Afin d'éliminer de telles erreurs, nous fournirons des conditions statiques — un *calcul d'horloges* — garantissant que les programmes s'exécutent correctement. Ces règles d'horloge jouent un rôle central mais pour simplifier la présentation, elles seront données dans le chapitre 4.

$$
\begin{array}{lcl}
\texttt{cl\_const}\,v\,(\texttt{true}.cl) & = & v.\texttt{cl\_const}\,v\,cl \\
\texttt{cl\_const}\,v\,(\texttt{false}.cl) & = & abs.\texttt{cl\_const}\,v\,cl \\
\texttt{base} & = & true.\texttt{base} \\
\texttt{const}\ v & = & \texttt{cl\_const}\,v\,\texttt{base} \\
\\
\texttt{extend}\ f.fs\ x.xs & = & (f\ x).\texttt{extend}\ fs\ xs \\
\texttt{extend}\ abs.fs\ abs.xs & = & abs.\texttt{extend}\ fs\ xs \\
\\
abs.xs\ \texttt{fby}\ abs.ys & = & abs.xs\ \texttt{fby}\ ys \\
x.xs\ \texttt{fby}\ y.ys & = & x.\texttt{fby1}\,y\,(xs)\,(ys) \\
\texttt{fby1}\,v\ abs.xs\,abs.ys & = & abs.\texttt{fby1}\,v\,xs\,ys \\
\texttt{fby1}\,v\ w.xs\ s.ys & = & v.\texttt{fby1}\,s\,xs\,ys \\
\\
abs.xs\ \texttt{when}\ abs.cs & = & abs.xs\ \texttt{when}\ cs \\
x.xs\ \texttt{when}\ \texttt{true}.cs & = & x.xs\ \texttt{when}\ cs \\
x.xs\ \texttt{when}\ \texttt{false}.cs & = & abs.xs\ \texttt{when}\ cs \\
\\
\texttt{merge}\ abs.cs\ abs.xs\ abs.ys & = & abs.\texttt{merge}\ cs\ xs\ ys \\
\texttt{merge}\ \texttt{true}.cs\ x.xs\ abs.ys & = & x.\texttt{merge}\ cs\ xs\ ys \\
\texttt{merge}\ \texttt{false}.cs\ abs.xs\ y.ys & = & y.\texttt{merge}\ cs\ xs\ ys
\end{array}
$$

FIG. 3.4 – Flots synchrones

Filtrage et composition de flots (`when` et `merge`) L'opérateur de filtrage attend deux arguments de même horloge. L'horloge du résultat dépend de la condition booléenne (c). Si les arguments ont l'horloge (cl), nous dirons que l'horloge du résultat est (cl `on` c) de sorte que :

$$
\begin{array}{lcl}
\texttt{true}.cl\ \texttt{on}\ \texttt{true}.cs & = & \texttt{true}.cl\ \texttt{on}\ cs \\
\texttt{true}.cl\ \texttt{on}\ \texttt{false}.cs & = & \texttt{false}.cl\ \texttt{on}\ cs \\
\texttt{false}.cl\ \texttt{on}\ abs.cs & = & \texttt{false}.cl\ \texttt{on}\ cs
\end{array}
$$

On peut noter ici que l'horloge est un flot simple sans valeur absente.

La définition du `merge` est totale lorsqu'une l'une des branches produit une valeur alors que l'autre est silencieuse.

Délais (`fby`) : L'opérateur de décalage impose que les horloges des ses arguments et de son résultat soient identiques.

Dans cette sémantique, toutes les fonctions de manipulation de flot préservent les longueurs et les programmes sont donc évaluables de manière synchrone. Cette sémantique étant partielle, nous avons introduit des conditions d'horloge garantissant une exécution sans échec. L'idée de compléter l'espace des valeurs instantanées avec une valeur marquant l'absence a été utilisée dans [20, 21]. La sémantique flot de donnée présentée ici a été entièrement décrite en COQ par Sylvain Boulmé et Grégoire Hamon [13]. Une représentation des flots avec absence explicite a également été utilisée pour un codage de SIGNAL en COQ [77].

3.3 Exécution paresseuse et simulation en Haskell

La représentation sous forme de listes paresseuses a été utilisée abondamment pour implanter Fran [33], une librairie Haskell dédiée à la programmation d'animations graphiques. Fran a depuis servi de base à des travaux sur la programmation fonctionnelle réactive (*Functional Reactive Programming* ou *FRP*) [110, 111].

La possibilité d'écrire directement la sémantique de Lustre dans un langage tel que Haskell nous a été extraordinairement utile et nous l'avons exploité dès le début. Elle nous a permis de mettre au point des constructions flot de données (e.g., la mise en séquence, le reset) bien avant d'avoir à en étudier la compilation. L'intérêt de Haskell comme langage de prototypage de langages est largement discuté dans [34]. C'est aujourd'hui un argument fort pour l'utiliser dans la conception de langages dédiés (*Domain Specific Languages*).

Le codage de Lustre en Haskell est intéressant pour plusieurs raisons :

- Les programmes sont, à quelques détails syntaxiques près, directement écrits en Haskell en définissant une librairie d'opérateurs *ad-hoc* (figure 3.2). Le mécanisme d'exécution de Haskell agit comme un interprète des constructions de Lustre.
- Un programme Lustre manipule des structures devant être évaluées de manière paresseuse et est composé d'équations récursives de flots. Les récursions de flots de Lustre s'écrivent telles quelles en Haskell.
- Les programmes Lustre sont typés gratuitement par le compilateur Haskell et les types sont inférés.
- Cette approche permet de changer aisément la représentation des valeurs du langage embarqué. En interprétant les opérateurs de flots par des opérations de manipulation de réseaux data-flow (*net-lists*), par exemple, l'évaluation d'un programme manipulant des flots s'apparente à une compilation du programme vers un circuit. Et le résultat de cette compilation peut à nouveau être traduit vers du code séquentiel, par exemple. Cette *compilation par exécution* est l'approche suivie dans les librairies Lava [12] et Hawk [51] construites au dessus de Haskell et dédiées à la conception de circuits synchrones.
- Le codage effectué permet de disposer de toutes les propriétés du langage hôte (types de données, récursivité).

L'idée d'une simulation d'un Lustre sans horloges en Haskell a été reprise récemment par Gordon Pace [79]. Le codage proposé exploite avantageusement les mécanismes de surcharge de Haskell qui permettent d'écrire les programmes dans une syntaxe proche de celle de Lustre [3].

Il était plus simple de choisir un langage paresseux tel que Haskell plutôt qu'un langage strict tel que Ocaml pour réaliser ce codage. Un codage vers Ocaml a cependant été réalisé. Il repose sur l'utilisation de primitives de gestion de la paresse (opérateurs `lazy/force`).

3.4 Concevoir un langage et des analyses dédiées

L'utilisation d'un langage paresseux (ou seulement d'une évaluation paresseuse des flots) comme support d'exécution ne nous semble pas satisfaisant dans un contexte de programmation de systèmes temps-réel. Nous avons rapidement abandonné l'idée de décrire Lustre et

[3] Par exemple, `1` est surchargé et désigne tantôt le flot constant `const 1`, tantôt la valeur scalaire `1` suivant son contexte d'utilisation.

ses extensions comme de simples librairies de HASKELL (ou de OCAML). Nous avons préféré suivre une approche plus traditionnelle identifiant un langage et en lui donnant une sémantique statique, dynamique et une méthode de compilation. Cette approche est également adoptée aujourd'hui par les concepteurs de *FRP* [111]. Les raisons principales ayant motivé ce choix sont les suivantes :

- La paresse nécessite un mécanisme d'exécution coûteux en temps et en mémoire, difficilement compatible avec les ressources modestes des processeurs sur lesquels les systèmes réactifs sont exécutés. L'exigence de sûreté des systèmes réactifs nous conduit à préférer un modèle d'exécution le plus simple possible.
- L'utilisation d'un ramasse miettes (*Garbage Collector*), l'impossibilité de déterminer statiquement quels sont les calculs effectués à chaque instant ainsi que l'ordre d'évaluation sont peu compatibles avec le besoin d'établir une borne maximum de temps de réaction (en particulier lorsque le système est échantillonné).
- L'embarquement de LUSTRE en HASKELL ne permet pas de disposer des analyses statiques dédiées (e.g., calcul d'horloge, causalité). En plongeant LUSTRE dans HASKELL, ces analyses devraient alors être étendues à tout HASKELL.
- Il est indispensable de disposer de diagnostics précis en cas d'erreur (de type, d'horloge, etc.) ce qui conduit à manipuler explicitement les arbres de syntaxe du programme et donc, de sortir de l'approche où le langage dédié est décrit par une librairie.
- La simulation de LUSTRE en HASKELL permet d'avoir une traduction facile vers des *net-list* dans lequel toutes les informations sur la structure du programme source ont disparu et le code séquentiel généré alors se révèle peu efficace.

Il nous est donc paru plus sage d'étendre progressivement LUSTRE avec des traits d'ordre supérieur et de développer des analyses statiques, une sémantique opérationnelle et des méthodes de compilation dédiées. La différence fondamentale entre LUCID SYNCHRONE et HASKELL est donc que le compilateur de LUCID SYNCHRONE rejette plus de programmes. En contre partie, le compilateur fournit des garanties sur le code obtenu et une aide à la mise au point des programmes.

3.5 Sémantique synchrone

Les sémantiques précédentes permettent de disposer d'une *simulation* à peu de frais. Elles ne permettent pas d'exprimer des propriétés opérationnelles fondamentales telles que l'exécution en temps et mémoire bornée et de comprendre comment *compiler* les programmes en des programmes séquentiels impératifs simples pouvant s'exécuter dans un environnement modeste (en particulier sans *ramasse miettes*).

Nous présentons ici une sémantique opérationnelle caractérisant les programmes pouvant être exécutés de manière synchrone. Nous donnons une formulation par des règles SOS [84], à la manière des règles de la sémantique d'ESTEREL [11] et de celles données dans [45] pour un langage d'équations de flots.

La première formulation est parue dans [20]. Compliquée et incomplète, elle a été abandonnée au profit d'une formulation moins classique basée sur la co-itération et que nous développerons dans le chapitre 5. Nous profitons cependant de ce document pour revenir sur une présentation de la sémantique opérationnelle synchrone sous forme de règles SOS. Elle est décrite dans la thèse de Grégoire Hamon [46].

3.5.1 Un noyau synchrone flot de données

Nous considérons ici un noyau flot de données dans lequel les programmes Lucid Synchrone peuvent être traduits aisément. Nous adoptons cependant une syntaxe un peu spéciale pour simplifier les règles de sémantique opérationnelle. Un programme est formé de déclarations (D) et d'expressions (e). Une déclaration peut être une composition de déclarations, une déclaration liant une variable à une expression ($x = e$) ou une déclaration liant une variable à une application ($x = e(e)$). Une expression (e) est construite à partir de valeurs immédiates importées et étendues aux flots (i), une application point-à-point d'un opérateur importé (op) [4], une opération de manipulation de flots ou une abstraction. L'opération de décalage `pre` reçoit un argument supplémentaire d'initialisation i, noté en indice.

$$
\begin{array}{lcl}
D & ::= & D \texttt{ and } D \mid x = e \mid x = e(e) \\
e & ::= & op(e,e) \mid \texttt{merge}\ e\ e\ e \mid e \texttt{ when } e \\
 & & \mid i \mid x \\
 & & \mid \texttt{pre}_i\ e \\
d & ::= & e \texttt{ with } D \\
 & & \mid \lambda x.e \texttt{ with } D \\
i & ::= & \texttt{true} \mid \texttt{false} \mid ... \\
op & ::= & + \mid ...
\end{array}
$$

Par rapport aux programmes Lucid Synchrone que nous avons vu dans le chapitre 2, il n'y a pas de définitions imbriquées et le résultat d'une application de fonction doit être nommé. Ainsi, une expression Lucid Synchrone imbriquée de la forme : `let rec` $x =$ `let` $y = e_1$ `in` e_2 `in` e_3 s'écrit dans ce format : $y = e_1 \texttt{ and } x = e_2 \texttt{ and } res = e_3 \texttt{ with } res$ et une application $e_1(e_2(x))$ s'écrit $x_1 = e_2(x) \texttt{ and } x_2 = e_1(x_1) \texttt{ with } x_2$.

Le passage par ce format intermédiaire est avant tout d'ordre esthétique et rien n'empêcherait de décrire la sémantique opérationnelle sur le vrai langage. Il permet cependant de décrire plus simplement les règle de sémantique en évitant évitant les problèmes de portée et de capture de noms.

3.5.2 Sémantique comportementale

La sémantique est décrite par des règles de réaction. Pour cela, on définit le langage des valeurs instantanées (v), les environnements de réaction (R) et trois prédicats de réaction.

$$
\begin{array}{ll}
\text{Valeurs :} & v \ ::= \ c \mid \epsilon \\
 & c \ ::= \ i \mid \lambda x.e \texttt{ with } D \\
\text{Environnement :} & R \ ::= \ [v_1/x_1, ..., v_n/x_n] \\
\text{Réaction :} & R \vdash e_1 \xrightarrow{v} e_2 \\
 & R \vdash D \xrightarrow{R'} D' \\
\text{Exécution :} & R.S \vdash D : R'.S'
\end{array}
$$

$Dom(R)$ désigne le domaine de R. $Var(R)$ désigne la liste des variables libres présentes dans R. R_1, R_2 désigne un environnement obtenu en concaténant les environnements R_1 et R_2. R_1, R_2 est possible lorsque $Dom(R_2) \cap (Dom(R_1) \cup Var(R_1)) = \emptyset$, autrement dit, il n'y ait pas de conflits de noms.

[4] On se limite ici aux opérations binaires.

Le prédicat $R \vdash e_1 \xrightarrow{v} e_2$ signifie que le programme e_1, soumis à l'environnement de réaction R, produit une valeur v et se transforme en e_2. Le prédicat $R \vdash D \xrightarrow{R'} D'$ signifie que la déclaration émet un environnement R' et se transforme en D'. L'exécution d'une déclaration, soumise à une suite d'environnements de réaction $S = R_1.R_2...$ produit une suite d'environnements de réaction $S' = R'_1.R'_2...$ tel que :

$$\frac{R \vdash D \xrightarrow{R} D' \quad S \vdash D' : S'}{R.S \vdash D : R.S'}$$

Les définitions des réactions des expressions et des déclarations sont données dans les figures 3.5 et 3.6.

Habituellement, la sémantique de LUSTRE est donnée à des programmes ayant passé au préalable le calcul d'horloge et où chaque opération à mémoire (e.g., `pre`, `current`) est annotée avec son horloge [97] (ou simplement à des programmes sans horloges, ce qui revient au même [45]). Nous préférons adopter un point de vue plus classique dans lequel la sémantique dynamique précède la sémantique statique : la sémantique dynamique du langage décrit comment les valeurs sont produites et la sémantique statique exprime des conditions suffisantes de “bonne” exécution. La sémantique opérationnelle synchrone est donc donnée directement au programme avant qu'il n'ait passé le typage ou le calcul d'horloges.

L'exécution d'un programme consiste en une suite de *réactions* à un environnement donné (i.e., l'état des variables libres du programme). A chaque instant, toutes les sous-expressions du programme effectuent une réaction pour produire leur valeur courante mais certaines réactions peuvent être silencieuses (à la manière du calcul synchrone SCCS de Milner [71]).

Détaillons les règles.

- Les règles sont considérées à renommage près.
- Une constante scalaire importée peut être émise ou pas (règles (IM) et (IM-[])) pourvu qu'elle soit synchrone avec les processus qui la consomment.
- La valeur d'une variable est celle qui est définie dans l'environnement de réaction (règle (TAUT)). Cette valeur peut être absente.
- Les règles (OP) et (OP-[]) indiquent que les entrées et les sorties de l'application point à point d'un opérateur doivent être synchrones : elles sont soit toutes absentes soit toutes présentes en même temps.
- Un opérateur de décalage émet sa valeur si son entrée émet une valeur. Sinon, il reste silencieux (règles (PRE)).
- L'opération `when` émet la valeur courante de sa première entrée lorsque la condition est vraie (règles (WHEN)). L'opération `merge` permet de combiner deux flots complémentaires (règles (MERGE)). Comme pour les autres opérations synchrones, les opérations `when` et `merge` sont silencieuses lorsque leurs entrées le sont.
- Une équation $x = e$ produit une réaction nouvelle $[v/x]$ si e produit une valeur v. Ce faisant, e peut faire référence à la valeur courante de e (règle (DEF)).
- Enfin, un système d'équations multiples est évalué en parallèle à chaque instant (règle (AND)).
- La figure 3.6 donne la sémantique des fonctions et de leur application. Une fonction produit une fonction (règle (ABS)). Une application consiste à ajouter à l'ensemble des déclarations courantes, les déclarations du corps de la fonction, la déclaration du calcul de l'argument et la déclaration correspondant au calcul du résultat (règle (APP)).

$$(\text{IM-}[])\quad R \vdash i \xrightarrow{\epsilon} i \qquad (\text{IM})\quad R \vdash i \xrightarrow{i} i \qquad (\text{TAUT})\quad R[v/x] \vdash x \xrightarrow{v} x$$

$$(\text{OP-}[])\ \frac{R \vdash e_1 \xrightarrow{\epsilon} e'_1 \quad R \vdash e_2 \xrightarrow{\epsilon} e'_2}{R \vdash op(e_1, e_2) \xrightarrow{\epsilon} op(e'_1, e'_2)} \qquad (\text{OP})\ \frac{R \vdash e_1 \xrightarrow{i_1} e'_1 \quad R \vdash e_2 \xrightarrow{i_2} e'_2 \quad i = op(i_1, i_2)}{R \vdash op(e_1, e_2) \xrightarrow{i} op(e'_1, e'_2)}$$

$$(\text{PRE-}[])\ \frac{R \vdash e \xrightarrow{\epsilon} e'}{R \vdash \mathtt{pre}_i\ e \xrightarrow{\epsilon} \mathtt{pre}_i\ e'} \qquad (\text{PRE})\ \frac{R \vdash e \xrightarrow{j} e'}{R \vdash \mathtt{pre}_i\ e \xrightarrow{i} \mathtt{pre}_j\ e'}$$

$$(\text{WHEN-}[])\ \frac{R \vdash e_1 \xrightarrow{\epsilon} e'_1 \quad D \vdash e_2 \xrightarrow{\epsilon} e'_2}{R \vdash e_1\ \mathtt{when}\ e_2 \xrightarrow{\epsilon} e'_1\ \mathtt{when}\ e'_2} \qquad (\text{WHEN-t})\ \frac{R \vdash e_1 \xrightarrow{i} e'_1 \quad D \vdash e_2 \xrightarrow{\mathtt{true}} e'_2}{R \vdash e_1\ \mathtt{when}\ e_2 \xrightarrow{i} e'_1\ \mathtt{when}\ e'_2}$$

$$(\text{WHEN-f})\ \frac{R \vdash e_1 \xrightarrow{i} e'_1 \quad R \vdash e_2 \xrightarrow{\mathtt{false}} e'_2}{R \vdash e_1\ \mathtt{when}\ e_2 \xrightarrow{\epsilon} e'_1\ \mathtt{when}\ e'_2}$$

$$(\text{MERGE-}[])\ \frac{R \vdash e_1 \xrightarrow{\epsilon} e'_1 \quad R \vdash e_2 \xrightarrow{\epsilon} e'_2 \quad R \vdash e_3 \xrightarrow{\epsilon} e'_3}{R \vdash \mathtt{merge}\ e_1\ e_2\ e_3 \xrightarrow{\epsilon} \mathtt{merge}\ e'_1\ e'_2\ e'_3}$$

$$(\text{MERGE-t})\ \frac{R \vdash e_1 \xrightarrow{true} e'_1 \quad R \vdash e_2 \xrightarrow{i} e'_2 \quad R \vdash e_3 \xrightarrow{\epsilon} e'_3}{R \vdash \mathtt{merge}\ e_1\ e_2\ e_3 \xrightarrow{i} \mathtt{merge}\ e'_1\ e'_2\ e'_3}$$

$$(\text{MERGE-f})\ \frac{R \vdash e_1 \xrightarrow{false} e'_1 \quad R \vdash e_2 \xrightarrow{\epsilon} e'_2 \quad R \vdash e_3 \xrightarrow{i} e'_3}{R \vdash \mathtt{merge}\ e_1\ e_2\ e_3 \xrightarrow{i} \mathtt{merge}\ e'_1\ e'_2\ e'_3}$$

$$(\text{DEF})\ \frac{R[v/x] \vdash e \xrightarrow{v} e'}{R \vdash x = e \xrightarrow{[v/x]} x = e'} \qquad (\text{AND})\ \frac{R, R_2 \vdash D_1 \xrightarrow{R_1} D'_1 \quad R, R_1 \vdash D_2 \xrightarrow{R_2} D'_2}{R \vdash D_1\ \mathtt{and}\ D_2 \xrightarrow{R_1, R_2} D'_1\ \mathtt{and}\ D'_2}$$

$$(\text{WITH})\ \frac{R \vdash D \xrightarrow{R'} D' \quad R, R' \vdash e \xrightarrow{v} e'}{R \vdash e\ \mathtt{with}\ D \xrightarrow{v} e'\ \mathtt{with}\ D'}$$

Fig. 3.5 – Sémantique comportementale synchrone

$$\text{(APP)}\quad \frac{R \vdash e_1 \overset{\lambda y.e\, \mathtt{with}\, D}{\rightarrow} e_1' \quad R \vdash x = e \;\mathtt{and}\, y = e_2 \;\mathtt{and}\, D \overset{R'}{\rightarrow} D'}{R \vdash x = e_1(e_2) \overset{R'}{\rightarrow} D'}$$

$$\text{(ABS)}\quad R \vdash \lambda x.e\, \mathtt{with}\, D \overset{\lambda x.e\, \mathtt{with}\, D}{\rightarrow} \lambda x.e\, \mathtt{with}\, D$$

FIG. 3.6 – Sémantique comportementale synchrone des fonctions et applications

Remarque 5 (Sémantique partielle et absence de règles de calcul) L'aspect synchrone de cette sémantique vient de l'absence de certaines règles de calcul. Si l'on tient compte de tous les cas de production ou de non production des valeurs des arguments des opérations point-à-point, par exemple, nous devrions considérer quatre cas au lieu de deux. Ainsi, il n'y a aucun moyen d'évaluer une expression lorsqu'une partie seulement de ses arguments est présente ni de stocker des résultats intermédiaires dans un buffer. Ceci constitue la différence fondamentale avec une sémantique classique de flots.

Une exécution s'apparente donc à une exécution sans échec : on dira qu'un programme est synchrone s'il admet une exécution sans échec dans la sémantique donnée précédemment. Le rôle du calcul d'horloge sera donc de rejeter les programmes ne pouvant être exécutés de manière synchrone. A la manière d'un calcul de types, son rôle sera de garantir l'absence de certaines erreurs à l'exécution. □

Remarque 6 (Non déterminisme) On peut noter le caractère indéterministe de cette sémantique puisqu'une constante, par exemple, peut indifféremment produire une valeur ou ne pas en produire. Ce qui compte ici, c'est que le générateur de constante soit *synchrone* avec ses divers consommateurs.

L'information synthétisée par le calcul d'horloge permet de transformer le programme en un programme annoté ne testant pas la présence de ses entrées. □

Remarque 7 (Programme clos et exécution synchrone) Il peut paraître surprenant que tout programme clos (sans entrées) puisse toujours s'exécuter de manière synchrone. Il suffit, en effet, que toutes ses réactions internes restent infiniment silencieuses. Ce cas pathologique est cependant sans grand intérêt : nous nous intéressons ici aux programmes synchrones pouvant réagir avec l'environnement extérieur et donc ayant des entrées (des variables libres) et sinon, où au moins une des variables réagit à chaque instant. Cet environnement extérieur fournit donc nécessairement des entrées et/ou demande des résultats. En pratique, le programme principal D est exécuté dans un contexte où toutes ses variables libres (ses entrées) reçoivent une valeur présente ($[c_1/x_1, ..., c_n/x_n] \vdash D \overset{R}{\rightarrow} D'$). □

Cette sémantique et ses propriétés sont décrites plus en détail dans [46].

3.6 Extension : processus qui terminent

Dans ce qui précède, nous nous sommes limités à des programmes ne terminant pas. Les programmes produisent donc des flots infinis. A chaque instant, un processus produit une valeur et se réécrit en un nouveau processus ou reste silencieux et se réécrit simplement. On peut cependant constater que la sémantique de Kahn permettait de modéliser le flots infinis mais aussi les flots finis, c'est à dire des processus pouvant s'arrêter.

Nous profitons de ce document pour présenter une extension du langage avec des constructions permettant de manipuler des processus qui terminent.

L'ajout des processus qui terminent permet de retrouver un style de programmation impératif *à la* ESTEREL. Nous l'illustrons sur deux exemples simples. Nous ajoutons de nouvelles constructions au noyau synchrone et étendons la sémantique opérationnelle en conséquence.

Nous commençons par un exemple classique de programme impératif qu'il est *facile* de programmer en ESTEREL et peu naturel en LUSTRE. L'exemple choisi est celui du contrôleur de souris [5]. Il reçoit le signal `Click` ou `Top`, émet `Double` s'il a reçu deux `Click` en moins de quatre `Top` et émet `Simple` sinon.

Nous supposons ici que les événements d'entrée et les événements de sorties n'ont pas lieu en même temps.

```
type input = Click | Top
type output = Simple | Double

let count n c = ok where
  rec o = if c then n -> pre o - 1 else n -> pre o
  and ok = o = 1

let emit_once x = do emit(x) until true done

let control input =
  loop
    do nothing until input = Click then                          (* 1 *)
    do nothing until input = Click then emit_once(Simple)        (* 2 *)
               until count 4 (input = Top) then emit_once(Double) (* 3 *)
  end
```

La fonction `emit_once x` émet la valeur `Just(x)` [6] puis s'arrête (la construction `do/until` dure au moins un instant).

La fonction `control` émet `nothing` jusqu'à ce que `input = Click` soit vrai (ligne `(* 1 *)`). A l'instant ou la condition est vraie, le point de contrôle passe à l'instruction suivante. La construction `then` est donc une construction de mise en séquence. La fonction émet la valeur absente `nothing` jusqu'à la présence d'un `Click` ou si quatre `Top` ont été rencontrés. A l'instant où l'une des conditions est vraie, la branche correspondante est exécutée (branche `(* 2 *)` ou `(* 3 *)`. A cet instant, le signal `Simple` ou `Double` est émis et le corps de la boucle s'arrête. La construction `loop` indique que le corps de la boucle doit être redémarré à chaque fois que l'itération précédente est terminée.

Dans le deuxième exemple, on considère la programmation d'une fonction d'intégration qui est définie à partir de deux fonctions primitives. On suppose l'existence d'une fonction booléenne `precision`.

[5] Quelle est la paternité de cet exemple ? Je l'ai vu écrit en LUSTRE, en ARGOS et en ESTEREL. Cet exemple s'écrit trivialement en ESTEREL alors qu'il peut être redoutable pour un programmeur LUSTRE débutant.

[6] La définition de `emit` a été donnée dans le chapitre 2.

```
let rectangle init dx = x where
  rec x = init -> pre x + dx * dt
let trapeze init dx = x where
  rec x = init -> pre x + ((pre dx) + dx) / 2) * dt

let integr i dx = x where
  rec x =
    loop
      do rectangle px dx until c then
      do trapeze px dx until not c done
    end
  and px = 0 -> pre x
  and c = precision x < eps
```

Le langage est étendu avec de nouvelles constructions. Nous nous limitons à trois constructions simples. La construction `do/until` permet d'arrêter un processus ; l'opération `do/then` est une construction de mise en séquence et `loop` est l'opération d'itération. Pour alléger les notations, nous écrivons `do` e `until` c au lieu de `do` e `until` c `done`.

$$
\begin{array}{rcl}
e & ::= & \ldots \mid \texttt{do}\, d\, \texttt{until}\, d \\
 & & \mid \texttt{do}\, d\, \texttt{then}\, d \\
 & & \mid \texttt{loop}\, d
\end{array}
$$

Pour modéliser la terminaison d'un processus, nous modifions les réactions de la manière suivante :

- $R \vdash e_1 \xrightarrow{v} e_2$ signifie que dans R le programme e_1 émet une valeur v et se réécrit en e_2
- $R \vdash e_1 \xrightarrow{v} *$ signifie que dans R, le programme e émet v et se termine.

Le langage des valeurs instantanées et les environnements de réaction restent inchangés. L'exécution d'une déclaration, soumise à une suite d'environnements de réactions $S = R_1.R_2...$ produit une suite d'environnements, éventuellement finie, $S' = R'_1.R'_2...$ telle que :

$$
\frac{R \vdash D \xrightarrow{R} D' \quad S \vdash D' : S'}{R.S \vdash D : R.S'} \qquad \frac{R \vdash D \xrightarrow{R} *}{R.S \vdash D : R}
$$

Les définitions des réactions des nouvelles opérations sont données dans la figure 3.7. Détaillons les règles.

- Les deux arguments d_1 et d_2 d'une opération `do/until` sont calculés à chaque instant et la valeur courante de d_1 est émise (règle (DO-f)). Le processus termine lorsque la valeur courante de d_2 est vraie (règle (DO-t)). Une opération `do/until` permet donc d'interrompre un calcul à la manière de l'opération `abort/when` d'ESTEREL.
- L'opération de mise en séquence `do/then` est une opération de reprise du calcul. L'expression d_1 est évaluée jusqu'à ce que le processus termine (règle (THEN)). Dès que le processus d_1 meurt, le programme se comporte comme d_2 (règle (THEN*)).
- L'opération `loop` est une itération de la séquence (règle (LOOP)).
- Comme pour le calcul original, les règles sont silencieuses lorsque tous leurs arguments le sont.

$$\text{(DO-f)}\ \frac{R \vdash d_1 \xrightarrow{i_1} d_1' \quad R \vdash d_2 \xrightarrow{\mathtt{false}} d_2'}{R \vdash \mathtt{do}\, d_1\, \mathtt{until}\, d_2 \xrightarrow{i_1} \mathtt{do}\, d_1'\, \mathtt{until}\, d_2'} \qquad \text{(DO-t)}\ \frac{R \vdash d_1 \xrightarrow{i_1} d_1' \quad R \vdash d_2 \xrightarrow{\mathtt{true}} d_2'}{R \vdash \mathtt{do}\, d_1\, \mathtt{until}\, d_2 \xrightarrow{i_1} *}$$

$$\text{(THEN)}\ \frac{R \vdash d_1 \xrightarrow{i_1} d_1'}{R \vdash \mathtt{do}\, d_1\, \mathtt{then}\, d_2 \xrightarrow{i_1} \mathtt{do}\, d_1'\, \mathtt{then}\, d_2} \qquad \text{(THEN*)}\ \frac{R \vdash d_1 \xrightarrow{i_1} * \quad R \vdash d_2 \xrightarrow{i_2} d_2'}{R \vdash \mathtt{do}\, d_1\, \mathtt{then}\, d_2 \xrightarrow{i_2} d_2}$$

$$\text{(LOOP)}\ \frac{R \vdash \mathtt{do}\, d\, \mathtt{then}\, \mathtt{loop}\, d \xrightarrow{i} d'}{R \vdash \mathtt{loop}\, d \xrightarrow{i} d'}$$

$$\text{(DO-[])}\ \frac{R \vdash d_1 \xrightarrow{[]} d_1' \quad R \vdash d_2 \xrightarrow{[]} d_2'}{R \vdash \mathtt{do}\, d_1\, \mathtt{until}\, d_2 \xrightarrow{[]} \mathtt{do}\, d_1'\, \mathtt{until}\, d_2'}$$

$$\text{(THEN-[])}\ \frac{R \vdash d_1 \xrightarrow{[]} d_1'}{R \vdash \mathtt{do}\, d_1\, \mathtt{then}\, d_2 \xrightarrow{[]} \mathtt{do}\, d_1'\, \mathtt{then}\, d_2}$$

$$\text{(LOOP-[])}\ \frac{R \vdash d \xrightarrow{[]} d'}{R \vdash \mathtt{loop}\, d \xrightarrow{[]} \mathtt{loop}\, d'}$$

FIG. 3.7 – Sémantique des constructions impératives

Nous modifions maintenant les règles pour les autres constructions du langage. Nous décidons que lorsqu'un processus contient un processus qui se termine et qui n'est pas à gauche d'une opération de mise en séquence (`do/then`), le processus complet se termine aussi ($*$ est absorbant). Pour alléger l'écriture, nous adoptons le raccourci d'écriture suivant. Si k désigne une expression e ou le processus mort $*$, on notera :

$$\begin{array}{lcl}
op(k_1, k_2) & = & * \text{ si } k_1 = * \text{ ou } k_2 = * \\
\mathtt{pre}_i\ k & = & * \text{ si } k = * \\
k_1\ \mathtt{when}\ k_2 & = & * \text{ si } k_1 = * \text{ ou } k_2 = * \\
\mathtt{merge}\ k_1\ k_2\ k_3 & = & * \text{ si } k_1 = * \text{ ou } k_2 = * \text{ ou } k_3 = * \\
(x = k) & = & * \text{ si } k = * \\
D_1\ \mathtt{and}\ D_2 & = & * \text{ si } D_1 = * \text{ ou } D_2 = *
\end{array}$$

Nous pouvons alors conserver les règles données dans la figure 3.5 et 3.6, étant entendu que les continuations sont simplifiées par les règles données ci-dessus. Par exemple :

$$\text{(OP*)}\ \frac{R \vdash e_1 \xrightarrow{i_1} e_1' \quad R \vdash e_2 \xrightarrow{i_2} * \quad i = op(i_1, i_2)}{R \vdash op(e_1, e_2) \xrightarrow{i} *}$$

Ainsi, le processus $op(e_1, e_2)$ s'arrête dès que l'un de ses arguments s'arrête. Il est possible d'interdire la combinaison d'un processus qui s'arrête avec un processus qui ne s'arrête pas en s'appuyant sur le mécanisme d'horloges. Pour cela, l'expression e_2 recevra une horloge différente de e_1 et le programme sera rejeté.

Remarque 8 [Boucles instantanées] La construction `loop` peut provoquer des boucles instantanées (e.g, `loop do x until true done end`). Ceci peut être évité raisonnablement en imposant que la construction `do/until` dure au moins un instant (en considérant `do` e `until false ->` c au lieu de `do` e `until` c).

Ceci pourrait être exprimé directement dans les règles en distinguant l'instant initial des autres instants (à la manière du traitement de la construction `abort/immediate` de ESTEREL). Dès lors que l'on dispose d'un opérateur d'initialisation `->`, je trouve plus naturel de l'utiliser ici. □

Chapitre 4

Horloges et autres analyses

Le rôle du calcul d'horloge est de définir un ensemble de conditions vérifiables statiquement permettant de garantir qu'une expression pourra s'exécuter de manière synchrone. Ces horloges sont ensuite utilisées pour générer un code séquentiel où les expressions ne sont calculées qu'aux instants où leurs horloges sont vraies. Dans ce chapitre, nous revenons sur le premier calcul d'horloges de Lucid Synchrone vu comme un système de types (4.1) et décrivons les évolutions successives de ce calcul. Nous montrons brièvement l'utilisation des horloges dans le processus de compilation (4.2). Nous montrons enfin comment, à partir du calcul initial, nous avons dérivé une forme simplifiée proche d'un calcul de type de ML avec une forme limitée de types existentiels (4.4).

Ce premier calcul d'horloges, décrit sous forme d'un système de types, a été réalisé avec Paul Caspi. Il est paru dans [19, 20]. Il a été implanté en 1995 dans la première version du compilateur. Ses évolutions ont toutes été implantées. Une simulation complète de ce calcul a été faite en Coq par Sylvain Boulmé et Grégoire Hamon [13]. Les horloges y sont directement représentées par des types dépendants de Coq. Le calcul simplifié *à la ML* n'a pas encore été publié. Il est implanté dans la version actuelle du compilateur.

L'originalité de ces calculs d'horloge est de représenter les horloges comme un langage de types. Grâce au lien étroit entre types et horloges, les horloges peuvent être utilisées comme *discipline de programmation* et non plus comme le résultat d'une simple analyse rejetant certains "mauvais" programmes. Nous pensons qu'elles peuvent être une aide précieuse au programmeur pour structurer les différents rythmes d'un programme. Le calcul d'horloge fait donc partie de l'interface du langage (comme c'est le cas du calcul de types en ML). Naturellement, ce point de vue est tenable si ce calcul est simple à utiliser et son comportement est facilement prévisible par le programmeur. Le rapprochement entre types et horloges, et plus précisément entre typage *à la ML* et calcul d'horloge permet aussi de récupérer les résultats théoriques et les techniques d'implantation. Ainsi, notre implantation reprend largement des techniques développées dans le compilateur Ocaml (e.g., unification destructive, niveaux pour la généralisation).

Les horloges en Lustre doivent être déclarées et ne sont pas polymorphes. Cela les rend difficiles à utiliser et elles restent peu utilisées en pratique. Le calcul d'horloges de Lucid Synchrone peut être considéré comme une extension à l'ordre supérieur du calcul d'horloges de Lustre, en exprimant les horloges comme des types et en ajoutant du polymorphisme. Les horloges de Signal sont plus expressives que celles de Lustre mais nous ne savons pas comment les décrire par un calcul de types simple. Une extension d'ordre supérieur de Signal associée à un calcul d'horloge basé sur un calcul de types avec effets a été proposée par Jean-Pierre Talpin [104]. Cette extension n'ayant pas, à ma connaissance, été implantée, je ne peux dire si les horloges proposées sont d'un usage facile.

Un nouveau compilateur de SCADE (la version industrielle de LUSTRE) a été développé à Telelogic SA (et maintenant à Esterel Technologies) en 2000 – 2001 par Jean-Louis Colaço [24]. Le calcul d'horloge implanté dans ce compilateur s'est fondé sur le premier calcul d'horloge de LUCID SYNCHRONE (4.1). Ce calcul a été expérimenté sur des exemples réels ($\geq$ 20000 lignes de code).

4.1 Un calcul de types dépendants

Nous décrivons d'abord le système initial développé en 1995 avec Paul Caspi et implanté dans la première version de LUCID SYNCHRONE. La formulation du calcul d'horloge donnée ici est celle d'un système de types dépendants classique [40].

Les horloges sont décomposées en des schémas d'horloges (σ) et des horloges (cl). Une horloge cl est soit une variable (β), soit une information de présence d'un flot (s), soit un produit d'horloges ($cl \times cl$), soit une horloge fonctionnelle ($\forall x : cl.cl$). Une horloge fonctionnelle (dépendante) $\forall x : cl_1.cl_2$ sera écrite simplement $cl_1 \rightarrow cl_2$ lorsque x n'apparaît ni dans les variables libres de cl_1, ni dans celles de cl_2 [1]. Les horloges de présence (s) peuvent être des variables (α) ou des horloges filtrées par des expressions (cl `on` e).

Pour éviter les captures de noms, on suppose que les programmes sont considérés à α-conversion près.

Le but du calcul d'horloges est de produire des jugements de la forme $H \vdash e : cl$ signifiant que "l'expression e a l'horloge cl dans l'environnement H".

$$
\begin{array}{lll}
\sigma & ::= & \forall \beta_1, ..., \beta_n.\forall \alpha_1, ..., \alpha_m.cl \\
cl & ::= & \forall x : cl.cl \mid cl \times cl \mid \langle s \rangle \mid \beta \\
s & ::= & s \ \texttt{on}\ e \mid \alpha \\
 & & \\
H & ::= & [x_1 : \sigma_1, ..., x_n : \sigma_n] \\
 & & \text{où pour tous } i,\ j \text{ tels que } j \leq i,\ x_i \notin FV(cl_j) \\
 & & \\
\text{jugement} & ::= & H \vdash e : cl
\end{array}
$$

Remarque 9 La présentation du langage d'horloges diffère légèrement de celle donnée dans [20, 22]. Dans le calcul initial, nous n'avions pas distingué les horloges des flots ($\langle s \rangle$) des horloges (cl) et une horloge pouvait être de la forme (cl `on` e). Cependant, le calcul d'horloge étant réalisé après typage, les horloges de la forme ($cl_1 \times cl_2$ `on` e) ou encore (($cl_1 \rightarrow cl_2$) `on` e) ne peuvent être construites.

La définition donnée ci-dessous précise donc la définition donnée antérieurement et caractérise plus finement les horloges bien formées. Elle est aussi plus fidèle aux diverses implantations réalisées. □

Le calcul d'horloge s'effectue en déclarant les horloges suivantes pour les primitives du

[1] Une horloge $\forall x : cl_1.cl_2$ est affichée $(x : cl_1) \rightarrow cl_2$ par le compilateur.

$$\text{(IM)}\quad H \vdash i : cl \qquad\qquad \text{(OP)}\ \frac{H \vdash e_1 : \langle s \rangle \quad H \vdash e_2 : \langle s \rangle}{H \vdash op(e_1, e_2) : \langle s \rangle}$$

$$\text{(FUN)}\ \frac{H, x : cl_1 \vdash e : cl_2}{H \vdash \lambda x.e : \forall x : cl_1.cl_2} \qquad \text{(APP)}\ \frac{H \vdash e_1 : \forall x : cl_2.cl_1 \quad H \vdash e_2 : cl_2}{H \vdash e_1\,(e_2) : cl_1[e_2/x]}$$

$$\text{(LET)}\ \frac{H \vdash e_1 : cl_1 \quad H, x : gen_H(cl_1) \vdash e_2 : cl_2}{H \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : cl_2[e1/x]}$$

$$\text{(REC)}\ \frac{H, x : cl \vdash e : cl}{H \vdash \mathtt{rec}\, x.e : cl}$$

$$\text{(INST)}\ \frac{cl = inst(H(x))}{H \vdash x : cl}$$

FIG. 4.1 – Un calcul d'horloge dépendant

langage.

$$\begin{array}{rcll} H_0 & = & [\mathtt{pre} & : \forall \alpha.\langle \alpha \rangle \rightarrow \langle \alpha \rangle, \\ & & \mathtt{fby} & : \forall \alpha.\langle \alpha \rangle \rightarrow \langle \alpha \rangle \rightarrow \langle \alpha \rangle, \\ & & \mathtt{when} & : \forall \alpha.\langle \alpha \rangle \rightarrow \forall x : \langle \alpha \rangle.\langle \alpha\ \mathtt{on}\ x \rangle \\ & & \mathtt{merge} & : \forall \alpha.\forall x : \alpha.\langle \alpha\ \mathtt{on}\ x \rangle \rightarrow \langle \alpha\ \mathtt{on\ not}\ x \rangle \rightarrow \langle \alpha \rangle \\ & & \mathtt{fst} & : \forall \beta_1, \beta_2.\beta_1 \times \beta_2 \rightarrow \beta_1, \\ & & \mathtt{snd} & : \forall \beta_1, \beta_2.\beta_1 \times \beta_2 \rightarrow \beta_2] \end{array}$$

L'horloge d'un délai `fby` est celle de ses deux arguments, qui doivent chacun être sur la même horloge. L'horloge d'une expression e `when` c dépend de la valeur de son second argument. Ceci est représenté par la construction `on`. Une expression `merge` e_1 e_2 e_3 utilise l'entrée courante de e_2 lorsque la valeur courante de e_1 est vraie ; sinon, elle utilise l'entrée de e_3. Une telle construction est donc correcte si l'horloge de e_2 est celle de e_1, limitée aux instants où e_1 est vrai, et l'horloge de e_3 est celle de e_1, limitée aux instants où e_1 est faux.

Nous désignons par $FV(\sigma)$, l'union des variables libres d'horloge (β) de cl et des variables libres de présence (α) de σ et nous étendons cette notation aux environnements. L'instanciation d'un schéma d'horloge et la généralisation d'une horloge sont définis ainsi :

$$\begin{array}{lcl} inst(cl) & = & cl \\ inst(\forall \vec{\beta}.\vec{\alpha}.cl') & = & cl'[\vec{cl}/\vec{\beta}][\vec{s}/\vec{\alpha}] \\ gen_H(cl) & = & \forall \beta_1, ..., \beta_n.\forall \alpha_1, ..., \alpha_m.cl \\ & & \text{ où pour tous } \beta_i,\ \alpha_i \notin FV(H) \end{array}$$

Le calcul d'horloge est donné dans la figure 4.1. Il est construit de la manière suivante :

– une constante a n'importe quelle horloge (règle (IM)). Nous écrivons ici simplement i pour `const` i.

– l'application point à point impose que toutes les entrées et le résultat soient de même horloge (règle (OP)).
– les règles de calcul d'horloge de l'abstraction et de l'application sont les règles classiques d'un système de types dépendants (règles (FUN) et (APP)). La règle d'horloge d'une abstraction doit garder une trace de la variable abstraite puisque celle-ci peut apparaître dans l'horloge du résultat. La règle de l'application est cohérente avec la règle précédente : l'horloge d'une application est l'horloge du résultat de la fonction instanciée avec son paramètre effectif.
– Les horloges sont généralisées aux définitions (règle (LET)) et instanciées aux points d'utilisation des variables (règle (INST)), comme c'est le cas en ML [70].
– La règle de typage des récursions est la règle classique de ML (règle (REC)). Nous verrons plus loin pourquoi il est préférable de prendre le schéma μ proposé par Mycroft [76].

4.2 Compilation et utilisation des horloges

Une fois que le calcul d'horloge a été réalisé, toute expression du programme est annotée par son horloge et le programme est transformé en un nouveau programme dans lequel les horloges ont été traduites en des expressions booléennes. Les horloges sont utilisées dans un but d'optimisation : il n'y a plus de test dynamique de présence/absence (i.e., pas de représentation explicite de (abs)) et elles sont utilisées pour optimiser le contrôle et générer des arbres de conditionnelles.

Les principes de cette transformation ont été publiés dans [21] et sont plus détaillés dans [22]. Nous en donnons ici un aperçu.

Les programmes manipulant des flots sont transformés en des programmes manipulant des flots avec horloges. Cette transformation suit exactement le calcul d'horloges et est obtenue en formulant un jugement de la forme :

$$H \vdash e : cl \Rightarrow e'$$

et signifiant que l'expression e d'horloge cl est transformée en une expression e' définie dans le noyau suivant (essentiellement un sous-ensemble de LUCID SYNCHRONE sans opérations d'échantillonnage/sur-échantillonnage). Ici, les constantes reçoivent un argument supplémentaire donnant leur horloge. Cet argument est donné entre crochets.

$$\begin{array}{lcl}
e & ::= & i[pres] \mid op(e,e) \mid x \\
 & & \mid e \texttt{ fby } e \\
 & & \mid \texttt{let } x = e \texttt{ in } e \mid \texttt{let rec } x = e \texttt{ in } e \\
 & & \mid \lambda x.e \\
 & & \mid pres \\
pres & ::= & pres \texttt{ on } e \mid x
\end{array}$$

Le système est détaillé dans [22] et nous ne le donnons pas.

Exemple 5 Par exemple, la fonction suivante (nommons la f) :

```
λ (x1,x2).
  let y = x1 fby y in
  let z = y + x2 + 1 in
  z : ∀α.⟨α⟩ → ⟨α⟩
```

est transformé en en :

```
λα. λ (x1,x2).
     let y = x1 fby y in
     let z = y + x2 + 1[α] in
     z
```

Toutes les constantes reçoivent leur horloge en paramètre. Une abstraction d'horloge est introduite à tout point de généralisation des horloges. Si `m1`, `m2` et `c` sont d'horloge α_1, une application de la forme :

```
r = f (m1 when c + 2, m2 when c) : α1 on c
```

est transformée en :

```
r = f (α1 on c) (m1 + 2[α1 on c], m2)
```

Une application est introduite à tout point d'instanciation d'horloge. Ici, f reçoit en paramètre l'horloge de son argument. On peut noter que la construction `when` disparaît : `x when c` se comporte comme `x`. Il en sera de même pour `merge` qui sera traduit vers une conditionnelle.

□

Le programme obtenu peut recevoir une sémantique synchrone où les flots sont complétés avec des absences explicites. Cependant, la manipulation explicite des absences est coûteuse et inutile en pratique et peut être éliminée en utilisant les information d'horloge. Plutôt que d'annoter les constantes, la transformation va annoter les opérations avec leur horloge qui va agir comme des gardes : une expression est calculée seulement si son horloge est vraie, sinon, la valeur précédemment calculée est rendue. Ainsi, les deux expressions précédentes sont transformées en :

```
λα. λ (x1,x2).
     let y = x1 fby[α] y in
     let z = (y +[α] x2 +[α] 1) in
     z
```

et en :

```
r = f (α1 on c) (m1 +[α1 on c] 2, m2))
```

L'une des optimisations classiques des compilateurs synchrones consiste alors à factoriser les gardes. Dans l'exemple ci-dessus, une seule garde sera ainsi générée pour `y + x2 + 1`. Cette optimisation a été réalisée par Grégoire Hamon durant sa thèse.

Le calcul d'horloge présenté ici est correct vis à vis de la sémantique : tout programme ayant une horloge, s'exécute de manière synchrone. Le résultat est énoncé et démontré dans [22].

4.3 Discussion

Ce calcul d'horloge est une extension du calcul d'horloge de Lustre. Par rapport à Lustre, les horloges sont inférées plutôt que vérifiées ; le calcul est basé sur l'unification de types plutôt qu'un calcul de point-fixe et il est étendu aux constructions d'ordre supérieur.

Le calcul présenté ici n'est pas celui utilisé dans le chapitre 2 de présentation du langage. C'est le calcul originel de Lucid Synchrone, implanté dans les versions `1.x` du compilateur. Nous présenterons plus loin (4.4) le nouveau calcul.

Nous discutons ici informellement des principaux traits de ce calcul, de son implantation, des extensions réalisées et des principales différences avec celui de Lustre.

L'implantation de ce calcul d'horloge a été faite dans l'esprit des implantations classiques des systèmes de types (non dépendants) de ML en imposant donc des restrictions majeures.

Unification d'horloges : Dans ce système, l'unification est différente de l'unification classique (de premier ordre) puisque les horloges peuvent contenir des expressions. Dans l'implantation nous nous sommes limités à l'égalité syntaxique entre expressions apparaissant dans les horloges. Ainsi :

– cl_1 `on` e_1 et cl_2 `on` e_2 sont unifiables si cl_1 et cl_2 le sont et
– e_1 et e_2 sont égaux syntaxiquement

Ainsi, les deux programmes suivants sont rejetés.

```
let y = x <> 0 in
(1 when y) + (2 when x <> 0)

(1 when true) + (1 when ((λ x.true) z))
```

Une telle restriction a cependant deux qualités. Le calcul d'horloge devient simple à utiliser et son fonctionnement est prévisible facilement. En pratique, un programmeur connaît toujours les horloges des expressions et attend une simple confirmation que le programme est correctement écrit. Le calcul d'horloge faisant partie de l'interface du langage — au même titre que les types font partie des langages ML — sa simplicité d'utilisation est essentielle. Ce calcul d'horloge est aussi relativement simple à implanter.

Une égalité d'horloges plus ambitieuse a été expérimentée dans laquelle les expressions sont mises sous forme normale (par application des β-réductions) et considérées modulo renommage (α-conversion). Elle est difficile à réaliser, coûteuse en temps et d'un intérêt pratique discutable et je l'ai abandonnée : je n'ai jamais trouvé d'exemple intéressant de programme justifiant une telle implantation et ne pouvant être écrit (souvent plus simplement) dans une forme acceptable par le système précédent.

Le choix d'avoir une égalité d'horloges simple a été poursuivi dans le système actuel au point de limiter cette égalité à une égalité de nom : deux expressions d'horloge sont égales si elles ont le même nom [2]. Nous y reviendrons dans la section 4.4.

Traitement des fonctions récursives et schéma mu : La règle d'horloge des récursions ne permet pas de typer les fonctions récursives de flot telle que le crible d'Eratosthene présenté dans le chapitre 2. En effet, une telle fonction est correctement typée (au sens des horloges) si elle s'appelle récursivement plus rarement, et donc avec une horloge différente.

De telles fonctions peuvent être correctement typées en utilisant la règle de Mycroft [76] dit *schéma mu*.

$$\text{(REC)}\ \frac{H, x : gen_H(cl) \vdash e : cl}{H \vdash \texttt{rec}\, x.e : cl}$$

Comme le note Mycroft, les récursions peuvent être typées en utilisant un algorithme itératif, semi décidable. Dans l'implantation nous avons limité la profondeur de cette itération à deux :

[2] cl_1 `on` x s'unifie avec cl_2 `on` x si cl_1 et cl_2 sont unifiables et si x est un nom.

les programmes sont rejetés si un point fixe n'est pas atteint jusque là. Un algorithme pour implanter efficacement le schéma mu est proposé dans [83] (utilisé pour une analyse décrite dans [64]). Il n'a pas été utilisé ici.

Forme prénexe pour les types dépendants : Dans l'implantation de ce système, les horloges des fonctions sont décomposées en deux catégories : l'horloge dépendante $(x : cl) \to cl'$ lorsque x apparaît dans cl ou cl' et l'horloge non dépendante $cl \to cl'$ sinon. Nous appliquons ensuite une restriction supplémentaire : les horloges bien formées sont celles où une horloge dépendante ne peut apparaître à la gauche d'un type flèche. Ceci permet de définir un système dirigé par la syntaxe (et d'obtenir l'unicité des horloges). Sinon, le programme suivant peut recevoir les deux horloges suivantes. Dans l'implantation, seule la première horloge est valide.

```
let f x g = (g x) + (1 when x)
node f : (x :a) -> (a -> a on x) -> a on x
node f : (x :a) -> ((y :a) -> a on y) -> a on x
```

Curryfication vs n-uplet d'arguments : Le langage étant essentiellement un sous-ensemble de ML, il privilégie une écriture curryfiée des fonctions. Les calculs de type dépendants imposent un ordre gauche-droite dans les dépendances : un paramètre de fonction ne peut dépendre que des paramètres placés à sa gauche. Ainsi, le programme :

```
let f c x y = o where
    rec nat = 0 -> pre nat + 1
    and o = x + (nat when c) + (y when c)

node f : (c :'a) -> 'a on c -> 'a -> 'a on c
```

est correctement typé (du point de vue des horloges) alors que le suivant ne l'est pas :

```
let f x c y = o where
    rec nat = 0 -> pre nat + 1
    o = x + (nat when c) + (y when c)
```

car `x`, d'horloge `a on c` ne peut dépendre de `c` qui n'est pas encore connu. On retrouve ici la contrainte imposée dans le calcul d'horloge de LUSTRE. Dans la signature d'horloge d'un noeud LUSTRE, les horloges de paramètres doivent dépendre de paramètres déjà définis. Ainsi, le premier programme s'écrit en LUSTRE :

```
node f(c : bool ; x : (int when c) ; y : int) returns (o : (int when c)) ;
var nat : int ;
let nat = 0 -> pre nat + 1 ;
    o = x + (nat when c) + (y when c) ;
tel ;
```

et la traduction en LUSTRE du second est rejetée.

Types produits et horloges en résultat : Les n-uplets sont pris en compte en ajoutant les règles classiques (couples et fonctions d'accès). Le langage est aussi étendu pour que les fonctions puissent prendre en paramètre des motifs simples formés de n-uplets de motifs ou de variables. Pour obtenir un système plus confortable, le système est étendu pour permettre l'écriture de :

```
let f (x, c, y) = o where
    rec nat = 0 -> pre nat + 1
    and o = x + (nat when c) + (y when c)

node f : 'a on c * (c :'a) * 'a -> 'a on c
```

On autorise ici qu'une fonction dépende d'une partie de ses arguments. Ceci s'obtient en étendent le type $(x : cl_1) \to cl_2$ en $(pat : cl_1) \to cl_2$ où pat désigne un motif (composé de variables).

Se pose ensuite le problème de rendre un flot dont l'horloge ne dépend ni d'une entrée de la fonction ni d'une variable plus globale mais d'une variable calculée localement. Ceci est nécessaire pour pouvoir écrire des signatures de noeuds *masquant* totalement leur implantation. Ainsi, le programme suivant calcule la suite des racines de ses entrées par la méthode de Newton. Il calcule une condition `ok` qui est vraie lorsque la différence entre les deux valeurs successives de la racine `uc` est inférieure à `eps` et émet une valeur seulement à ces instants. Ici, `ok` est abstrait et calculé à l'intérieur de la fonction.

```
let eps = 0.001
let root input = (output,ok) where
    rec x = if ok input (0 -> pre x)
    and uc = input -> (pre uc) / 2 + (x / 2 * pre uc)
    and ok = true -> (uc - pre uc <= eps)
    and output = uc when ok

node root : 'a -> 'a on ok * (ok : 'a)
```

`ok` n'est pas une entrée de la fonction mais est calculée localement. Dans l'horloge de `root`, `ok` est une variable quantifiée existentiellement et l'horloge se lit : “pour toute entrée `input` d'horloge `'a`, il existe (un unique) couple `(output,ok)` d'horloge `'a on ok * 'a`”.

Le type des fonctions devient $(pat_1 : cl_1) \to (pat_2 : cl_2)$ où les variables de pat_1 sont quantifiées universellement et celles de pat_2, existentiellement. Cette extension a été dérivée des systèmes avec type existentiels [99, 74].

Ainsi, les horloges peuvent dépendre de variables qui sont, soit globales, soit des paramètres d'entrée, soit des paramètres de sortie. Ici, les quantifications imposent naturellement un ordre gauche-droite : cl_1 dépend seulement de variables (plus) globales et des noms apparaissant dans pat_1 et cl_2 peut dépendre en plus des noms apparaissant dans pat_2. Le sur-échantillonnage propre à Signal est obtenu en autorisant les variables de cl_1 à dépendre aussi de variables de pat_2. En reprenant l'exemple précédent :

```
let eps = 0.001
let root input = (output,ok) where
    rec x = merge ok input ((0 -> pre x) whenot ok)
    and uc = input -> (pre uc) / 2 + (x / 2 * pre uc)
    and ok = true -> (uc - pre uc <= eps)
    and output = uc when ok

node root : 'a on ok -> 'a on ok * (ok : 'a)
```

Cette fois, une entrée `input` est lue seulement aux instants où `ok` est vrai alors qu'elle était lue à chaque instant dans l'écriture précédente. La fonction `root` sur-échantillonne son entrée : elle effectue un calcul sur une horloge interne `'a` alors que son entrée a une horloge plus lente `'a on ok`. Ce programme ne peut pas être écrit en LUSTRE qui interdit le sur-échantillonnage.

Ici encore, les horloges fonctionnelles sont décomposées en deux entités syntaxiques : les horloges sans quantifications ($cl_1 \rightarrow cl_2$) et les autres ($(pat_1 : cl_1) \rightarrow (pat_2 : cl_2)$). Et seules les horloges en forme prénexe (aucune quantification sur des variables à gauche des flèches) sont autorisées. Ces extensions ont toutes été implantées (dans les versions `1.x` successives), de 1997 à 1999. Elles sont toujours disponibles.

Mono-horloge et constantes polymorphes : On peut noter que les schémas d'horloges peuvent être quantifiés par rapport à plusieurs variables d'horloge. En LUSTRE, ces schémas ne peuvent contenir qu'une variable d'horloge par noeud, appelée l'horloge de base du noeud. Ainsi :

```
let f (x,y) = (x+1, y+2)
f : ('a * 'b) -> ('a * 'b) ← ici
f : ('a * 'a) -> ('a * 'a) ← en LUSTRE
```

De plus, il n'y a pas de constantes *polymorphes* en LUSTRE : les constantes doivent être sur l'horloge de base du noeud. Ainsi, l'expression `1 + (x when half)` est rejetée de même que le programme suivant [3] :

```
let rec half = true -> pre (not (half))
let f x = (x when half) when half
node f : 'a -> 'a on half on half
```

Le résultat de `(f x)` est donc présent une fois sur quatre. Les constantes polymorphes doivent être expliquées (comment dire que le flot `half` a deux horloges différentes ?). Dans le langage, les flots définis à *top-level* et les flots définis localement sont traités différemment. Les déclarations globales de valeurs définissent des constantes (c'est à dire des valeurs ne dépendant pas d'entrées). Ainsi, `half` est vu comme la définition d'un processus (c'est un noeud non encore instancié pour reprendre la terminologie propre à LUSTRE).

```
let rec half = true -> pre (not half)
```

est un raccourci (*i.e., se compile comme*) :

```
let process_half () = half where
    rec half = true -> pre (not half) in half
```

Ensuite, toute instance de `half` peut avoir sa propre horloge et donc :

```
let f x = (x when half) when half
```

est un raccourci pour :

```
let f x = (x when process_half())
             when (process_half(() when process_half()))
```

[3] L'exemple de la montre vu au chapitre 2 était une illustration des horloges polymorphes.

Ce dernier programme est tout à fait valide en LUSTRE mais il me paraît beaucoup moins lisible que la version précédente. Le polymorphisme d'horloge permet aussi d'écrire `1 + (x when c)` qui est rejeté en LUSTRE.

Limitations du calcul d'horloge : Il y a deux formes de limitations du calcul d'horloge qui ont pour conséquence de rejeter des programmes qui pourraient être acceptés.

La première vient du fait que l'unification d'horloge est purement syntaxique et n'interprète jamais les formules booléennes. Ainsi :

```
let tl x = x when (false -> true)
let rejected x = x + tl x
```

est rejeté alors qu'il est synchrone (son réseau de Kahn s'évalue à buffer borné, ou dit autrement, il est *deforestable*). Ce type de programmes peut, la plupart du temps, être réécrit en rendant les buffers explicites.

L'autre source de rejet vient de la faiblesse du langage d'horloges. Considérons, par exemple, le programme suivant :

```
let rec generic_reset f (c, x) =
merge (whilenot c)
      (f (x when (whilenot c)))
      (generic_reset f ((c,x) when (not (whilenot c))))
```

`generic_reset` attend une fonction `f`, une condition booléenne `c` et une entrée `x`. Elle commence à exécuter `f x` jusqu'à ce que `c` soit vrai puis recommence une nouvelle exécution, etc. Cette fonction est donc un opérateur de ré-initialisation générique.

Ce programme est rejeté par le calcul d'horloge parce que `f` est utilisé avec un (nombre infini) d'horloges différentes. Ce rejet vient d'une limitation intrinsèque du polymorphisme de ML, sur lequel le calcul d'horloge est fondé. Seules les horloges des variables introduites par des `let` peuvent être généralisées. L'extension du système avec du polymorphisme d'ordre supérieur est une voie à envisager. Ce programme peut alors recevoir l'horloge : $\forall\alpha.(\forall\alpha_1.\langle\alpha_1\rangle \rightarrow \langle\alpha_1\rangle) \rightarrow \langle\alpha\rangle \times \langle\alpha\rangle \rightarrow \langle\alpha\rangle \times \langle\alpha\rangle$.

La présence d'expressions dans les horloges et la β-réduction effectuée par le calcul peuvent, sur certains exemples, rendre le temps du calcul d'horloge prohibitif. On peut s'en convaincre aisément en construisant un opérateur de mise en séquences basé sur les horloges. Considérons un opérateur `do_waiting` mettant en séquence deux fonctions `f` et `g`. Il exécute `f` sur son entrée jusqu'à ce qu'une condition `c` soit vraie puis exécute `g` :

```
let whilenot c = w where rec w = true -> (not c) & (pre w)
let do_waiting c f g (init, x) = o where
    rec o = merge (whilenot c) (f (init, x) when whilenot c))
                  (g (pre o, x) when not (whilenot c))

node do_waiting :
     (c : 'a)
     -> (('a on whilenot c * 'a on whilenot c) -> ('a on whilenot c)
     -> (('a on not whilenot c * 'a on not whilenot c)
         -> ('a on not whilenot c)
     -> 'a * 'a -> 'a
```

Le calcul d'horloge effectuant des β-reductions, la taille des horloges et le temps pour les calculer peut croître exponentiellement avec la taille du programme. Il suffit de considérer une suite de la forme :

```
let id x = x
let main =
    do_waiting (count 1) id
    do_waiting (count 2) id
    ...
    do_waiting (count 10) id id
```

Cependant, nous n'avons jamais rencontré ce problème de croissance en écrivant des programmes à la main. Ce problème a été découvert en janvier 2000 par Grégoire Hamon lors de l'écriture d'un outil de génération automatique de programmes Lucid Synchrone.

Puisque le calcul d'horloges est un cas particulier de système de types dépendants, deux questions se sont posées rapidement :

- Comment internaliser ce système dans un environnement d'aide à la preuve disposant de type dépendants (e.g., Coq) ? Ceci permettait, d'une part, de vérifier sa correction et, d'autre part, de prouver des programmes en profitant des informations d'horloges. Cette expérience a été menée par Sylvain Boulmé et Grégoire Hamon dans [13].
- Comment internaliser (un sous-ensemble seulement) dans un système de type proche de celui d'un langage de la famille ML ? C'est cette expérience que nous décrivons ici et qui a été implantée dans la version récente du compilateur. Ce nouveau calcul est plus simple que le premier calcul, ne souffre pas des problèmes de croissance dus aux dépendances et reprend l'essentiel de l'implantation classique d'un système de type à la ML. Ce calcul est cependant beaucoup moins expressif que le calcul initial.

4.4 Un calcul d'horloge à la ML

Nous présentons maintenant un calcul simplifié construit à partir du calcul précédent. Il a été largement illustré dans le chapitre 2. Il repose sur une idée simple : réduire au maximum le système pour qu'il se rapproche du système de type de ML. Les horloges de flots ne peuvent plus contenir que des noms de variables ; ces noms doivent être introduits explicitement avec une construction `let clock` et le type des fonctions est le type classique (non dépendant) $(cl_1 \rightarrow cl_2)$. Le langage est étendu avec une opération d'introduction d'horloges.

$$e \quad ::= \quad \texttt{let clock } x = e \texttt{ in } e$$

Le langage d'horloge est défini ci-dessous. Il est composé de schémas d'horloges (σ) et d'horloges simples (cl). Les schémas d'horloge peuvent être quantifiés sur des variables d'horloges (β), des variables d'horloges pour les flots (α) et des variables d'expressions (X). Une horloge peut être fonctionnelle ($cl \rightarrow cl$), un produit ($cl \times cl$), une variable (β), une information de présence $\langle s \rangle$ ou une information de présence associée à une valeur $(c : s)$. Lorsqu'un flot e a le type $(c : s)$, il est égal à c et est présent lorsque s est vrai (son horloge est $\langle s \rangle$). c peut être soit une variable (X), soit une constante (l). Une information $(c : s)$ va permettre d'introduire des dépendances.

$$
\begin{array}{lll}
\sigma & ::= & \forall \beta_1, ..., \beta_n.\forall \alpha_1, ..., \alpha_m.\forall X_1, ..., X_k.cl \\
cl & ::= & cl \rightarrow cl \mid cl \times cl \\
 & & \mid \beta \mid \langle s \rangle \mid (c : s) \\
s & ::= & s \texttt{ on } c \mid s \texttt{ on not } c \mid \alpha \\
c & ::= & l \mid X \\
l & ::= & l_1 \mid ... \mid l_n ... \\
 & & \\
H & ::= & [x_1 : cl_1, ..., x_n : cl_n] \\
 & & \\
\text{jugements} & ::= & H \vdash e : cl
\end{array}
$$

Les primitives du langages sont typées en considérant l'environnement initial suivant :

$$
\begin{array}{lll}
H_0 & = & [\texttt{pre} \quad : \forall \alpha.\langle \alpha \rangle \rightarrow \langle \alpha \rangle, \\
 & & \;\texttt{->} \quad : \forall \alpha.\langle \alpha \rangle \rightarrow \langle \alpha \rangle \rightarrow \langle \alpha \rangle, \\
 & & \;\texttt{when} \quad : \forall \alpha.\forall X.\langle \alpha \rangle \rightarrow (X : \alpha) \rightarrow \langle \alpha \texttt{ on } X \rangle \\
 & & \;\texttt{whenot} \quad : \forall \alpha.\forall X.\langle \alpha \rangle \rightarrow (X : \alpha) \rightarrow \langle \alpha \texttt{ on not } X \rangle \\
 & & \;\texttt{merge} \quad : \forall \alpha.\forall X.(X : \alpha) \rightarrow \langle \alpha \texttt{ on } X \rangle \rightarrow \langle \alpha \texttt{ on not } X \rangle \rightarrow \langle \alpha \rangle \\
 & & \;\texttt{fst} \quad : \forall \beta_1, \beta_2.\beta_1 \times \beta_2 \rightarrow \beta_1, \\
 & & \;\texttt{snd} \quad : \forall \beta_1, \beta_2.\beta_1 \times \beta_2 \rightarrow \beta_2]
\end{array}
$$

Les horloges sont généralisées et instanciées de la manière suivante :

$$
\begin{array}{lll}
inst(cl) & = & cl \\
inst(\forall \vec{\beta}.\vec{\alpha}.\vec{X}.cl') & = & cl'[\vec{cl}/\vec{\beta}][\vec{s}/\vec{\alpha}][\vec{c}/\vec{X}] \\
gen_H(cl) & = & \forall \beta_1, ..., \beta_n.\forall \alpha_1, ..., \alpha_m.\forall X_1, ..., X_k.cl \\
 & & \text{ où les } \beta_i,\ \alpha_i,\ X_i \notin FV(H)
\end{array}
$$

$FV(H)$ désigne l'ensemble des variables libres de H. Les règles du calcul d'horloge sont données dans la figure 4.2.

Excepté la règle d'introduction d'horloges, les règles d'horloge sont celles du système de types de ML. La règle (DEF-clock)) introduit un nom neuf l qui ne doit pas apparaître dans les hypothèses. La fonction $names(cl)$ rend la listes des étiquettes l de cl.

4.5 Discussion

Implantation efficace : Le système présenté a été entièrement implanté. L'implantation s'est inspirée très largement de l'approche proposée par François Pottier [69] pour implanter le système de Laüfer et Odersky [60].

Les optimisations classiques ont étées réalisées (e.g., unification destructive, niveau pour la généralisation). Il en résulte un code très efficace et le temps du calcul d'horloge est maintenant similaire au temps mis par la synthèse des types.

Sa proximité avec un calcul de type classique le rend aussi plus accessible à un programmeur ML.

$$\text{(IM)}\quad H \vdash i : \langle s \rangle \qquad \text{(OP)}\ \frac{H \vdash e_1 : \langle s \rangle \quad H \vdash e_2 : \langle s \rangle}{H \vdash op(e_1, e_2) : \langle s \rangle}$$

$$\text{(FUN)}\ \frac{H, x : cl_1 \vdash e : cl_2}{H \vdash \lambda x.e : cl_1 \rightarrow cl_2} \qquad \text{(APP)}\ \frac{H \vdash e_1 : cl_2 \rightarrow cl_1 \quad H \vdash e_2 : cl_2}{H \vdash e_1\,(e_2) : cl_1}$$

$$\text{(INST)}\ \frac{cl = inst(H(x))}{H \vdash x : cl}$$

$$\text{(LET)}\ \frac{H \vdash e_1 : cl_1 \quad H, x : gen_H(cl_1) \vdash e_2 : cl_2}{H \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : cl_2}$$

$$\text{(REC)}\ \frac{H, x : cl_1 \vdash e_1 : cl_1 \quad H, x : gen_H(cl_1) \vdash e_2 : cl_2}{H \vdash \texttt{let rec}\, x = e_1\ \texttt{in}\ e_2 : cl_2}$$

$$\text{(LET-clock)}\ \frac{H \vdash e_1 : \langle s \rangle \quad H, x : gen_H((l : s)) \vdash e_2 : cl_2 \quad l \notin names(H, cl_2)}{H \vdash \texttt{let clock}\ x = e_1\ \texttt{in}\ e_2 : cl_2}$$

FIG. 4.2 – Un calcul d'horloge simplifié

Enregistrement et filtrage de types somme par horloges : Les enregistrements ont été ajoutés au langage et sont vus comme des n-uplets particuliers. Le calcul d'horloge impose que tous les champs d'un enregistrement soient synchronisés sur la même horloge.

Cette restriction rend le calcul d'horloge très simple : une expression $e.label$ est d'horloge $\langle s \rangle$ si e est d'horloge $\langle s \rangle$ et un enregistrement $\{_1 = e_1\ ; \ldots ; l_n = e_n\}$ est d'horloge $\langle s \rangle$ si tous les l_i sont d'horloge $\langle s \rangle$. Elle semble parfois trop contraignante. Nous ne savons pas, pour le moment, comment disposer d'un calcul d'horloge simple permettant de distinguer les champs d'une structure.

Les types sommes ont également été incorporés. Ici, les constructeurs sont “liftés” au niveau des flots. Un constructeur `C` d'arité zéro désigne un flot infini constant alors qu'une expression `C(e)` désigne l'application point à point de `C` au flot `e`. Les règles d'horloges de ces constructions sont détaillées dans [46].

Restriction de portée dans l'utilisation des horloges Ce calcul impose des conditions sur la portée des horloges définies localement. Lorsqu'une horloge est introduite avec la construction `clock`, celle ci ne doit pas échapper à sa portée lexicale. Par exemple :

```
let escape_up x =
  let clock c = true in
  merge c (x + (1 when c)) 0

File "tutorial.ls", line 53-55, characters 4-70:
```

```
>....escape_up x =
>  let clock c = true in
>  merge c (x + (1 when c)) 0
The clock of this expression depends on ?_c0 which escape its scope.
```

Ce programme est rejeté parce que la construction `let clock` introduit un nom neuf `?_c0` qui ne peut échapper à la portée locale du `let`. Ici, le programme est rejeté parce que la variable `x` devrait être d'horloge `'a on ?_c0`.

```
let escape_down x =
  let clock c = true in
  x when c

File "tutorial.ls", line 53-55, characters 4-54:
>....escape_down x =
>  let clock c = true in
>  x when c
The clock of this expression depends on ?_c0 which escape its scope.
```

Ce programme est aussi rejeté car la fonction rend une expression sur l'horloge `'a on ?_c0` alors que `?_c0` doit rester local (`?_c0` doit être un nom neuf, jamais utilisé).

Ainsi, les horloges neuves introduites par une construction `clock` ont une durée de vie limitée à leur bloc.

Ajout de dépendances et/ou de vrais types existentiels : La construction `let clock` limite la portée du nouveau nom d'horloge au bloc courant. Cette limitation est ennuyeuse dans la mesure où elle empêche d'écrire de nombreux programmes qui étaient acceptés dans le système précédent (e.g., les deux exemples ci-dessus). L'exemple de sur-échantillonnage vu précédemment et que l'on aimerait écrire :

```
let eps = 0.001
let root input = output where
    rec x = merge ok input ((0 -> pre x) whenot ok)
    and uc = input -> (pre uc) / 2 + (x / 2 * pre uc)
    and clock ok = true -> (uc - pre uc <= eps)
    and output = uc when ok
```

est, lui aussi, rejeté car `ok` ne peut rester local à son bloc. Cet exemple était accepté dans le système précédent. Cela nous conduit à considérer un calcul d'horloge dans lesquel les horloges peuvent être quantifiées existentiellement. Cette étude est en cours.

4.6 Autres analyses

Le rôle du calcul d'horloge est de donner des conditions d'exécution synchrone d'un réseau de Kahn. Le réseau s'exécute sans avoir besoin d'instructions de synchronisation, d'attendre et donc de stocker des valeurs intermédiaires.

Un réseau peut cependant contenir des rebouclages instantanés ou *boucles de causalité* de sorte qu'il n'est pas possible de produire séquentiellement les valeurs du flot. Dans un contexte de programmation de systèmes réactifs, ces boucles doivent être rejetées statiquement par une analyse appelée *analyse de causalité*.

De plus, le retard unaire (`pre`) peut introduire des valeurs indéfinies (notées *nil*) dans les flots (car $\texttt{pre}(x_0...x_n...) = nil.x_0...x_n...$ et il est nécessaire de s'assurer que le comportement du système ne dépend pas de ces valeurs indéfinies.

Nous décrivons brièvement les deux analyses envisagées pour LUCID SYNCHRONE. Plusieurs analyses de causalité ont été implantées dans le compilateur dont l'une est décrite dans [28, 27]. L'analyse d'initialisation utilisée actuellement est basée la technique présentée dans [25].

4.6.1 Recherche des boucles de causalité

Dans le monde réel, les lois de la causalité interdisent l'utilisation d'informations du futur dans le présent. De telles interférences ne peuvent se produire dans le contexte d'un langage de flots synchrones sans horloges, car il n'existe pas d'opérateur pour exprimer le futur [4]. Cependant, les calculs étant supposés instantanés, la causalité peut être enfreinte en utilisant dans l'instant la valeur même qui est en train d'être définie. C'est le cas avec la fonction :

```
let causality_loop dx dt = x where
    rec x = x + dx * dt
```

où simplement l'équation `rec x = x + 1` car le flot `x` dépend instantanément de lui-même. Les définitions récursives de flots causales sont celles qui définissent des flots infinis dont les valeurs peuvent être obtenues par itérations successives. Dans la sémantique de Kahn, la plus petite solution de l'équation `rec x = x + dx * dt` est ϵ (car $(\lambda\ \texttt{x.}\ \texttt{x + dx * dt})(\epsilon) = \epsilon$). Nous dirons que de telles équations ne sont pas *productives*.

De même, les définitions récursives telles que `rec x = x` et `rec x = -(1 + x`^{2}`)/2` sont valides en tant qu'équations mais ne définissent pas des flots infinis calculables par itération successives et devront donc être rejetées.

Par contre, grâce au retard introduit par l'opérateur `pre`, l'expression `1 + pre x` ne dépend plus instantanément de `x` ; la définition récursive `rec x = 1 + pre x`, qui dénote le flot des entiers strictement positifs, est valide, et doit être acceptée.

Dans le cas de définitions mutuellement récursives, il serait trop restrictif d'imposer à chaque expression à la droite d'un signe `=` de contenir un `pre`. Par exemple, `rec x = y + 1 and y = pre x` est une définition valide de deux flots `x` et `y`.

L'analyse des boucles de causalité (et le caractère productif de définitions récursives) a été étudiée dans divers contextes tels que : les programmes de flots [101, 107], les définitions inductives et co-inductives dans la théorie des types [39, 3] ; les définitions de valeurs paresseuses potentiellement infinies dans les langages paresseux [55]. Ces analyses sont toutes basées sur un critère *syntaxique* reposant essentiellement sur des propriétés de longueurs de suites que l'on peut énoncer (informellement) par :

$$\frac{\forall n, |x| = n \Rightarrow |e| > n}{\texttt{rec}\, x = e \text{ est bien formé}}$$

[4]La fonction `tl` est définie par : `tl(x) = x when (false -> true)`.

en considérant, dans le cas des suites que $|x.e| > |e|$. Toutes les analyses proposées reposent donc sur une méthode de comptage. Il s'agit de vérifier que la fonction $(\lambda x.e)$ est strictement croissante (elle est monotone par construction). Ces méthodes s'expriment aisément sous forme de règles de typage et peuvent être appliquées à des types de données quelconques [39, 55]. Ces analyses sont *syntaxiques* car la bonne définition ne dépend pas des valeurs instantanées des flots mais seulement de la présence de constructeurs/destructeurs.

En se limitant à un noyau synchrone sans horloge, la causalité d'une récursion revient à vérifier simplement :

$$\frac{(\lambda x.e)(\epsilon) \neq \epsilon}{\mathtt{rec}\, x = e \text{ est bien formé}}$$

Seule les fonctions f telles que $f(\epsilon) \neq \epsilon$ peuvent donc être itérées [5].

L'analyse de causalité s'exprime simplement sous forme d'un système de types avec sous-typage pour un noyau de flots sans horloges. Nous donnons très informellement le système de types et les règles dans la figure 4.3. Les types des flots sont notés s. Le type d'un flot peut être une variable (α) où une variable surlignée plusieurs fois par un “chapeau”. Cette notation des types avec des compteurs exprimés par un nombre de “chapeaux” est due à Gimenez [39]. Ainsi, un flot de type $\overline{s}$ est un flot dont la longueur est strictement plus grande qu'un flot dont le type est s.

Pour simplifier l'écriture, je ne rappelle pas les règles classiques d'instanciation et de généralisation (elles sont similaires à celles données dans la section 4.4).

Cependant, ce système est basé sur un système avec sous-typage qui s'avère coûteux sur des programmes de grande taille. D'où l'idée de définir un calcul basé seulement sur du typage classique (reposant seulement sur l'unification).

Cette analyse prend en fait la forme d'un système de types avec rangées. Elle comporte le polymorphisme des systèmes de type de ML et HASKELL. La principale différence est que toutes les expressions qui auraient eu un type de base (int, float, bool, ...) dans un système classique ont ici un type de la forme $\{\varphi\}$. La rangée φ encode les dépendances de l'expression en question.

On souhaite bien sûr pouvoir, pour une même expression, indiquer ses dépendances vis-à-vis de plusieurs variables de récursion (voire toutes). C'est pour cette raison que φ est une rangée ([112],[98]) dont les étiquettes sont les noms des variables de récursion. Les rangées ont été conçues à l'origine pour introduire des enregistrements extensibles en ML. D'une manière générale, elles permettent dans certains cas (dont le nôtre) d'utiliser le polymorphisme pour remplacer le sous-typage. Le système est décrit dans [28] et la thèse de Pascal Cuoq [27].

4.6.2 Recherche des erreurs d'initialisation

L'ajout d'un opérateur de retard non initialisé `pre` nécessite une analyse statique rejetant les programmes non initialisés. Une analyse par typage a été développée en collaboration avec Jean-Louis Colaço. Cette analyse a été implantée dans le compilateur de LUCID SYNCHRONE et dans un compilateur de SCADE [6]. Elle est publiée dans [25].

[5] Cette définition est duale de celle des fonctions *strictes* : f est stricte lorsque $f(\perp) = \perp$. La reconnaissance des fonctions strictes est un standard des compilateurs optimisant des langages paresseux.

[6] Dans cette étude, l'aspect *laboratoire* de LUCID SYNCHRONE pour prototyper des extensions et des analyses pour LUSTRE a bien fonctionné : l'analyse a été mise au point et expérimentée dans LUCID SYNCHRONE puis implantée et testée sur des exemples réels dans SCADE.

$$
\begin{array}{lll}
\sigma & ::= & \forall \beta_1, ..., \beta_n . \alpha_1, ..., \alpha_n . t \\
t & ::= & t \to t \mid t \times t \mid \beta \mid s \\
s & ::= & \overline{s} \mid \alpha \\
\\
H & ::= & [\sigma_1/x_1, ..., \sigma_n/x_n] \\
\\
\mathit{jugement} & ::= & H \vdash e : t
\end{array}
$$

$$
\text{(IM)}\quad H \vdash i : t \qquad\qquad \text{(OP)}\ \frac{H \vdash e_1 : s \quad H \vdash e_2 : s}{H \vdash \mathit{op}(e_1, e_2) : s}
$$

$$
\text{(VAR)}\quad H, x : \sigma \vdash x : \mathit{inst}(\sigma)
$$

$$
\text{(APP)}\ \frac{H \vdash e_1 : t_2 \to t_1 \quad H \vdash e_2 : t_2}{H \vdash e_1(e_2) : t_1} \qquad \text{(ABS)}\ \frac{H, x : t_1 \vdash e : t_2}{H \vdash \lambda x.e : t_1 \to t_2}
$$

$$
\text{(REC)}\ \frac{H, x : s \vdash e_1 : \overline{s} \quad H, x : s \vdash e_2 : t}{H \vdash \texttt{let rec}\, x = e_1 \ \texttt{in}\ e_2 : t} \qquad \text{(LET)}\ \frac{H \vdash e_1 : t_1 \quad H, x : \mathit{gen}_H(t_1) \vdash e : t_2}{H \vdash \texttt{let}\ x = e_1 \ \texttt{in}\ e_2 : t_2}
$$

$$
\text{(SUB)}\ \frac{H \vdash e : \overline{s}}{H \vdash e : s} \qquad \text{(PRE)}\quad H \vdash \texttt{pre} : s_1 \to s_2
$$

FIG. 4.3 – Causalité par sous-typage

Nous donnons ici un aperçu du problème et de la solution adoptée. Les détails techniques se trouvent dans [25]. Le comportement des opérateurs `->` et `pre` est rappelé ci-dessous.

`x`	x_0	x_1	x_2	x_3	...
`pre x`	*nil*	x_0	x_1	x_2	...
`y`	y_0	y_1	y_2	y_3	...
`y fby x`	y_0	x_0	x_1	x_2	...
`y -> x`	y_0	x_1	x_2	x_3	...
`y -> pre x`	y_0	x_0	x_1	x_2	...
`y -> pre (pre x)`	y_0	*nil*	x_0	x_1	...

La valeur de `pre x` n'est pas définie au premier instant et est notée *nil*. Une valeur indéfinie peut être éliminée en insérant un opérateur d'initialisation (`->`) et l'on peut remarquer que le délai initialisé `fby` est un racourci pour `->pre`. Le rôle de l'*analyse d'initialisation* est de vérifier que ces valeurs indéfinies n'affecteront pas le comportement du programme et qu'en particulier, aucun flot produit par le programme principal ne contiendra de valeur indéfinie *nil*.

Comme nous l'avons dit dans le chapitre 2, le délai non initialisé `pre` n'était pas présent dans les premières versions du langage. Les équations de flots étaient écrites en utilisant l'opérateur `fby` de LUCID. Le délai non initialisé `pre` été ajouté par souci de compatibilité avec LUSTRE et parce qu'il permet d'écrire les équations de manière plus "naturelle" et plus concise [7].

Si une analyse d'initialisation est nécessaire, cette analyse doit rester simple et rapide pour une raison essentielle : un programme peut toujours être sur-initialisé sans changer sa sémantique. Par exemple, le programme suivant :

```
let f x =
    let rec half = true -> pre half in
    let rec o = if half then x else pre o in
    o
```

produit un flot toujours défini si `x` est toujours défini car `half` est vrai au premier instant. La bonne initialisation de `o` demande une analyse un peu fine qui fait intervenir les valeurs de certains flots (ici `half`). Ce programme peut être réécrit immédiatement en un programme (sur-initialisé) :

```
let f x =
    let rec half = true -> pre half in
    let rec o = if half then x else x -> pre o in
    o
```

pour lequel l'analyse d'initialisation sera beaucoup plus simple. On peut donc plus facilement se satisfaire d'une analyse d'initialisation élémentaire que dans le cas de l'analyse de causalité. Dans le cas des boucles de causalité, l'ajout d'un délai *change* la sémantique du programme. Les programmes sur-initilisés étant moins lisibles et compilés vers du code moins efficace (à cause des redondances), on recherchera tout de même une analyse rejetant peu de programmes bien initialisés.

Une analyse triviale consiste à vérifier que tout délai est précédé syntaxiquement par une opération d'initialisation. Cependant, cette analyse ne permet pas d'accepter :

```
let switch c = o where
    rec o = c -> if c then not (pre o)
                 else pre o
```

Il est également utile, parfois, de définir une fonction qui n'initialise pas nécessairement toutes ses sorties et/ou qui n'impose pas que toutes ses entrées soient initialisées, laissant le soin au contexte de gérer l'initialisation. Ainsi, la présence de valeurs *nil* dans des flots intermédiaires doit être acceptée.

Une autre approche consiste à donner une valeur initiale par défaut aux délais non initialisés (e.g., `false` dans le cas du délai booléen) [8]. Cette approche, simple en apparence, est complexe à mettre en oeuvre et semble nécessiter un mécanisme de surcharge : il y a autant d'opération `pre` qu'il y a de types possibles pour son argument et il faut donner la valeur par défaut associé à `pre(x)` pour chaque nouvelle déclaration de type. Le comportement du programme dépend d'une valeur initiale implicite et il y a peu de chance pour que cette valeur initiale soit toujours la plus adéquate [9].

[7] Lorsque j'écris un système d'équations de flots, je considère d'abord l'invariant (en utilisant seulement `pre`) puis j'ajoute des valeurs d'initialisation.

[8] C'est l'approche choisie dans la version (`V7`) d'Esterel : `pre(x)` est un flot booléen initialisé à false et `pre1(x)` est un flot booléen initialisé à `true`.

[9] D'où l'introduction de `pre` et `pre1` en Esterel.

Une approche ambitieuse consiste à utiliser des outils de vérification (e.g., NP-TOOLS, LESAR) en traduisant le problème de l'initialisation en un problème booléen. Bien que théoriquement possible, cette approche nous paraît coûteuse, non modulaire (au moins dans le cas de l'ordre supérieur) et inutile pour les nombreux programmes réels que nous avons pu analyser. De plus, la maîtrise de la complexité (pratique) de l'analyse et l'obtention de diagnostics précis et clairs — ce qui est essentiel dans un compilateur — sont loin d'être évidentes.

Nous avons donc choisi de mettre au point une analyse d'initialisation spécifique. Nous adoptons une approche très modeste de type *1-bit* où les flots sont considérés comme étant, soit toujours définis, soit toujours définis sauf éventuellement au premier instant. Cette formulation s'exprime facilement sous forme d'un système de types avec sous-typage. Dans ce système, une expression toujours initialisée reçoit le type **0** alors qu'une expression dont la première valeur est éventuellement indéfinie reçoit le type **1**. La relation $\mathbf{0} \leq \mathbf{1}$ exprime alors le fait qu'*un flot initialisé peut être utilisé là où un flot non initialisé est attendu*. Par exemple :

```
let deriv x = s where
  rec s = x - pre x
```

reçoit le type $\mathbf{0} \rightarrow \mathbf{1}$ qui signifie que l'entrée `x` doit être initialisée alors que sa sortie ne l'est pas. En effet, `pre x` a le type **1** si `x` a le type **0**. L'opérateur `(-)` attend deux arguments de même type et donc, après affaiblissement du type de `x`, la variable `s` reçoit le type **1**. Ainsi, le programme suivant est rejeté :

```
let deriv2 x = s where
  rec s = deriv (deriv (x))
```

puisque `deriv x` est de type **1** alors que `deriv` attend une valeur de type **0**.

Cette abstraction *1-bit* vient des propriétés particulières de `->` et de `pre`. En particulier, l'opérateur `->` ne peut éliminer une valeur *nil* apparaissant au deuxième instant dans un flot `pre(pre x)`. C'est la raison pour laquelle une expression `pre(pre x)` est rarement utilisée. Naturellement, l'utilisation d'un double décalage ne signifie pas que le programme soit incorrect (dès lors que la valeur n'est pas utilisée dans le calcul final). Cependant, l'absence de *nil* dans le résultat dépend alors de propriétés booléennes du programme rendant l'analyse d'initialisation plus complexe. Une expression `pre(pre x)` sera donc rejetée par notre analyse.

L'analyse d'initialisation s'exprime comme un système classique de type avec une relation de sous-typage [1, 86, 87] où les types de base sont **0** et **1**. Nous avons réalisé (après coup) que le système était quasiment identique au système minimal utilisé dans [2] pour étudier le problème de la simplification des types. Le système est modulaire et peut s'implanter en utilisant les techniques classiques. Nous avons récupéré les méthodes décrites dans [2, 87] pour simplifier les types et rendre l'implantation efficace.

Cette analyse a été implantée dans le compilateur de LUCID SYNCHRONE. Elle a été également implantée dans le compilateur RELUC de SCADE par Jean-Louis Colaço et testée sur des exemples de taille réelle (plusieurs milliers de lignes). En pratique, l'abstraction *1-bit*, bien que très grossière, donne de bons résultats [10]. L'analyse est précise : la plupart du temps, les programmes rejetés contiennent effectivement des problèmes d'initialisation; l'analyse est rapide : des programmes réels de plusieurs milliers de lignes ont été analysés en quelques secondes; les diagnostics d'erreurs sont pertinents : il suffit souvent d'ajouter une opération d'initialisation à l'endroit de l'erreur pour obtenir un programme qui est accepté

[10] Je dois avouer que je ne m'attendais pas à de tels résultats au départ.

Domaine d'application	nb. de lignes de LUSTRE analysées	temps d'analyse
transport	$\sim$ 17000	$\leq$ 55 sec.
hélicoptère	$\sim$ 45000	$\leq$ 11 sec.
avionique	$\sim$ 3000	$\leq$ 1 sec.
automobile	$\sim$ 1000	$\leq$ 0.1 sec.

FIG. 4.4 – Temps d'analyse dans le compilateur RELUC (automne 2001)

par l'analyse. Enfin, dans le compilateur SCADE, cette nouvelle analyse a permis de réduire d'un facteur dix en pratique le nombre d'alarmes (des *warning*), par rapport à la version précédente du compilateur. La figure 4.4 reprend les chiffres donnés dans l'article [25] sur les temps d'analyses dans RELUC. Depuis, plusieurs optimisations ont été incorporées. La version actuelle améliore de plus d'un facteur dix le temps d'analyse au point d'être négligeable par rapport aux temps des autres phases de compilation. Il me semble que cela confirme notre opinion de traiter ce problème par une méthode dédiée plutôt que de s'en remettre à des outils généraux de vérification de propriétés booléennes.

Chapitre 5

Co-itération et code séquentiel

La co-itération est apparue récemment comme un concept essentiel pour raisonner et prouver des propriétés sur des programmes flot de données [82, 73] et même pour les compiler et les optimiser. Son intérêt vient du fait qu'elle permet de représenter les structures de données infinies (telles que les flots) de manière efficace plutôt que d'avoir à les représenter de manière paresseuse. Ceci explique qu'elle soit utilisée *implicitement* dans la compilation de Lustre ou Scade.

Dans ce chapitre nous décrivons l'intérêt de la co-itération dans un but de compilation. Elle permet d'expliquer simplement ce qui est fait traditionnellement dans les compilateurs de Lustre et Scade et de l'étendre à l'ordre supérieur. Elle a été implantée dans le compilateur de Lucid Synchrone. La partie théorique de ce travail a d'abord été publiée dans [21]. Une version plus détaillée contenant les preuves des propriétés énoncées dans ce chapitre sont données dans [22]. Le travail sur la co-itération a été mené avec Paul Caspi.

Ce chapitre n'explique pas la compilation d'un programme flot de données vers du code séquentiel efficace telle qu'elle est faite dans la version actuelle du compilateur. La question des optimisations réalisées (e.g. inlining, factorisation des conditions) n'est pas non plus abordée (bien qu'elles représentent l'essentiel en nombre de ligne de la génération de code). Cependant, toutes les versions du compilateur que nous avons réalisées reposent sur les principes énoncés dans ce chapitre.

5.1 Code séquentiel en boucle simple

La sémantique de flots (chapitre 3), même si elle est exécutable, ne peut servir de base pour implanter un système réactif : elle nécessite un mécanisme d'exécution coûteux ne donnant aucune garantie sur les temps de réaction et la mémoire utilisés.

Dans la compilation de Lustre, une fonction de flot ($f : Stream(T_1) \to Stream(T_2)$) servant de programme principal est compilée vers du code séquentiel dit en *boucle simple*. Ce code séquentiel est déterminé par un *état initial* (disons $M_0 : S$), une fonction de transition ($f_v : S \to T \to T_2$) et une fonction mémoire ($f_m : S \to T \to S$). Le programme principal (l'entrée est i et la sortie est o) est implanté par une boucle :

```
(* initialisation de l'etat *)
M := M0 ;
(* etat permanent *)
loop
```

```
  read i ;
  o := f_v M i ;
  M := f_m M i ;
  emit o
end
```

Dans le programme obtenu, les flots sont représentés par des valeurs scalaires et il n'y a plus d'allocation (ou désallocation) dynamique de données (autre que celle effectuée au premier instant) [1].

Il existe essentiellement deux compilateurs de programmes LUSTRE, un compilateur "académique" et celui de SCADE, la version industrielle de LUSTRE. Ces compilateurs font des choix un peu différents que nous rappelons ici. Le compilateur académique compile les programmes sous forme de code séquentiel impératif après dépliage complet des appels de fonctions, de sorte que la compilation s'applique seulement à un paquet d'équations de flots. Il s'agit essentiellement d'un sous-ensemble sans appel de fonctions (exceptées les fonctions externes) ni fonctions du noyau introduit dans le paragraphe 3.5.1 du chapitre 3. La compilation vers du code séquentiel dit en *boucle simple* d'un système d'équations de flots consiste en un ordonnancement des équations et en l'application d'un certain nombres d'optimisations locales (e.g., factorisation de la mémoire, du contrôle). Le temps de compilation et la taille du code produit sont proportionnels à la taille du système d'équations. Des techniques d'énumération de certaines variables d'états permettent d'obtenir du code efficace sous forme d'automate fini, voire même d'automate fini minimal [97, 45]. Cependant, le temps de compilation et/ou la taille du code obtenu pour produire un automate peuvent être exponentiellement plus grands que la taille du source. Ceci explique que la compilation vers du code en boucle simple (ou qui s'en rapproche) soit préférée en pratique [2]. Le choix fait par le compilateur académique de déplier au préalable tous les appels de noeuds s'explique par l'impossibilité de compiler séparément certains noeuds ne contenant pourtant pas de boucles de causalité ([43] rappelle l'exemple classique). On peut noter qu'à cause du dépliage préalable des fonctions, le partage de code présent dans le source a disparu : les seuls appels de fonction correspondent aux fonctions externes.

Le compilateur de SCADE (la version industrielle de LUSTRE) est guidé par trois impératifs : la *traçabilité* du code (c'est à dire la possibilité de relire le code et faire le lien avec le programme source) ; la *compilation séparée* (c'est à dire la possibilité de compiler la définition d'un noeud indépendamment de ses contextes d'utilisation, condition nécessaire à l'écriture de programmes de grande taille) et la nécessité de pouvoir être *certifié*. Ces impératifs imposent des choix de compilation différents. En particulier, les fonctions sont compilées séparément et ne sont pas dépliées systématiquement [3]. La compilation séparée impose la présence de délais (e.g., `pre`) supplémentaires dans l'écriture des équations. Le compilateur réalise essentiellement des optimisations locales, de faible coût et ne changeant pas la complexité du programme de sorte que le programmeur peut avoir une idée assez juste de ce qui est calculé à chaque instant (ce qui est loin d'être clair dans le cas d'une compilation vers des automates). Dans ce mode de compilation, le dépliage est utilisé dans un but d'optimisation (comme c'est le cas

[1] En pratique, le code généré correspond plutôt à la construction d'une seule fonction $step : S \to T_1 \to T_2 \times S$ et d'un état initial où $step\, M\, i = (f_v\, M\, i, f_m\, M\, i)$ qui permet de partager les calculs communs entre la production du résultat et la construction du nouvel état.

[2] Les techniques d'énumération sont au coeur des outils de vérification automatique (e.g., outil LESAR [44] de LUSTRE).

[3] En fait, le dépliage est à la discrétion de l'utilisateur qui le contrôle manuellement.

dans les compilateurs des langages généralistes). Le code généré par le compilateur SCADE contient donc des appels de fonctions internes et doit donc gérer le partage de code. Une méthode de compilation séparée des programmes LUSTRE, imposant moins de contraintes sur les programmes que l'approche choisie dans SCADE est proposée par Pascal Raymond [96].

C'est en étudiant le problème de la compilation séparée vers du code séquentiel, sans dépliage statique préalable de toute la fonctionnalité (et par conséquent, pour trouver une explication simple de la compilation de SCADE) que nous nous sommes intéressés à la co-itération. Pour ce faire, nous avons poursuivi l'approche choisie dans la sémantique flot de données en utilisant HASKELL comme un langage de prototypage — une *règle à calcul* évoluée en quelque sorte — pour mettre au point un jeu de combinateurs.

5.2 Co-itération

Les flots (infinis) à valeur dans V sont définis par le type co-inductif :

$$Stream(V) = \texttt{Cons}\, V\ (Stream(V))$$

Les flots sont construits avec le constructeur `Cons`. Les valeurs de ce type étant infinies, les expressions doivent être évaluées de manière paresseuse. L'idée de la co-itération est de considérer un flot comme un processus de calcul séquentiel [82, 73] qui est défini par :

$$Cstream(V, S) = \texttt{Co}_\texttt{F}\ (S \to V \times S)\ S$$

où $\texttt{Co}_F$ est un constructeur d'arité 2. Un processus est composé :

- d'une fonction de transition de type $S \to V \times S$ calculant, à partir d'un état, une valeur et un nouvel état ;
- d'un état initial de type S

Les flots *concrets* sont reliés aux flots par la fonction `run` qui itére le processus de calcul :

$$\begin{array}{rcl} run(\texttt{Co}_\texttt{F}\, t\, s) & = & let\, v, s' = t(s)\, in \\ & & \texttt{Cons}\, v\ (run(\texttt{Co}_\texttt{F}\, t\, s')) \end{array}$$

Remarque 10 La définition de $Cstream(V, S)$ rappelle donc la sémantique opérationnelle synchrone ($R \vdash e \xrightarrow{v} e'$)) où un état est représenté par une expression. □

Grâce à la co-itération, les définitions récursives des opérations `const`, `extend` ou `pre` (vues dans le chapitre 3) peuvent être transformées en des définitions co-itératives non récursives.

Ces *combinateurs* peuvent être directement écrits en HASKELL (à quelques détails syntaxiques près) et peuvent donc être exécutés [4]. Leur définition est donnée dans la figure 5.1. Nous donnons, le type de chaque combinateur.

- `const` fabrique un processus constant, sans mémoire (que l'on représente ici par `Nil`)
- `extend` émet une valeur en calculant un pas de ses deux arguments et en appliquant la valeur courante de son premier argument à la valeur de son deuxième. La mémoire utilisée pour le calcul est l'union des deux mémoires.

[4]Nous prenons quelques libertés avec la syntaxe HASKELL pour faciliter la lecture. Une fonction est notée ici $\lambda x.e$ (au lieu de `\x->e`), les définitions récursives sont signalées en utilisant le mot clef `rec` (alors qu'elles sont récursives par défaut en HASKELL) et le type produit est noté `*` (au lieu de `,`).

```
data State(s) = Nil | St(s)

let const v = CoF (λ s.(v,s)) Nil
const : t -> Cstream t (State(s))

let extend (CoF df if) (CoF dv iv) =
           CoF (λ (sf,sv). let f, sf' = df sf in
                           let v, sv' = dv sv in
                           (f v), (sf', sv')) (if,iv)
extend : Cstream (t -> t') s' -> Cstream t s -> Cstream t' (s * s')

let pre v (CoF t i) =
        CoF (λ (memo, s).memo, (t s)) (v,i)
pre : t -> Cstream t s -> Cstream t (t * s)
```

FIG. 5.1 – Définition concrètes d'opérateurs de flots

– L'opération `pre` émet à chaque instant la valeur stockée dans un état (`memo`) et met à jour son état avec la valeur courante de son entrée.

Il reste maintenant à traiter le cas des fonctions, des applications et de la récursion. Dans le cas des fonctions et des applications, une solution consiste à déplier statiquement tous les appels de fonction jusqu'à l'obtention d'un paquet d'équations de flots mutuellement récursives. L'autre solution cherche à conserver la structure du code et correspond à la solution implantée depuis longtemps par les ingénieurs de SCADE : une fonction de flots f est compilée vers une fonction f_C écrite dans un langage impératif et paramétrée par une structure de données représentant sa mémoire. C'est ce que nous expliquons maintenant.

Normalement, une fonction de type :

$$f : \mathit{Stream}(V) \rightarrow \mathit{Stream}(V')$$

devrait être compilée en une fonction :

$$f : \mathit{Cstream}(V, S) \rightarrow \mathit{Cstream}(V', S')$$

Une fonction devrait donc pouvoir utiliser une valeur de type $\mathit{Cstream}(V, S)$, c'est à dire la *recette* de son argument. Si l'on observe le cas des opérations flot de données considérées jusque là et qui préservent les longueurs, ceci est inutilement complexe :

La valeur courante de la sortie de $f(x)$ dépend seulement de la valeur courante de l'entrée x (et éventuellement d'un état interne)

D'où l'idée de définir un combinateur d'application spécial correspondant à ce que nous appellerons une *application synchrone*. Une application synchrone évalue à chaque instant la valeur courante de ses deux entrées et la valeur courante de son corps. Sa définition est donnée dans la figure 5.2.

La mémoire nécessaire au calcul d'une application synchrone est composée de la mémoire nécessaire au calcul des deux arguments et de la mémoire locale de la fonction. Nous choisissons d'initialiser la mémoire locale avec la valeur spéciale `Nil`. Nous dirons qu'une fonction est synchrone lorsqu'elle peut être appliquée de manière synchrone.

```
let co_apply (CoF tf sf) (CoF te se) =
    CoF (λ (st,sf,se). let (vf, sf') = tf sf in
                       let (ve, se') = te se in
                       let (v, st') = vf ve st
                       in (v, (st',sf',se')))
        (Nil,sf,se)
co_apply : Cstream (t -> State s' -> t' * (State s')) sf
         -> Cstream t s -> Cstream t' ((State s') * sf * s)

let co_lambda f =
    const (λv,s. let CoF t s0 = f (const v) in
                 s1 = case s of Nil -> s0 | St(s) -> s
                 v1,s2 = t s1
                 v1,St(s2))
co_lambda : (Cstream t (State s) -> Cstream t' s')
            -> Cstream (t -> State s' -> t' * State(s')) (State(sf))

let co_rec (CoF t i) =
    CoF((λ (se,s). let v,s' = t s in
                   let rec ve, se' = v ve se in
                   (ve,(se',s')))
        (Nil,i)
co_rec : Cstream(t -> State s -> t * (State s)) s'
        -> Cstream t (State(s) * s')
```

Fig. 5.2 – Application, abstraction et récursion synchrones

Définition 6 (Fonction synchrone) *Une fonction f est synchrone si elle est l'image d'une application synchrone, i.e., il existe f' telle que $f =$* `co_apply`(f')

En observant la signature du combinateur d'application, on peut noter qu'il transforme un flot de fonctions (à mémoire) en une fonction de flots. Le combinateur pour l'abstraction synchrone transforme les fonctions de flots en des flots de fonctions. Sa définition est donnée dans la figure 5.2. La correction de ces combinateurs repose sur la propriété suivante.

Théorème 1 (Correction)
Pour tous f, e, `co_apply (co_lambda(co_apply(`f`))) `e` = co_apply `f e

Il reste à traiter le cas de la récursion. Par exemple, l'équation de flots (`+` désigne l'opération d'addition point à point) :

```
rec fibo = pre 1 (fibo + pre 0 fibo)
```

doit pouvoir se compiler en :

```
fibo = CoF (λ (memo1,memo2).memo1, (memo1 + memo2, memo1)) (1,0)
```

d'où l'idée de définir un combinateur de récursion `co_rec`. Sa définition est donnée dans la figure 5.2. En observant son type, on peut noter que l'opération de récursion transforme une récursion sur les flots en une récursion sur les valeurs scalaires. Nous ajoutons ici le mot clef `let rec` pour signaler que `ve`, la valeur retournée par la fonction de transition peut dépendre d'elle même [5]. Cet opérateur de récursion est correct pourvu que la propriété suivante soit vérifiée.

Théorème 2 (Correction) *Pour tout f,* `co_apply` f `(co_rec f) = co_rec(`f`)`.

Deux cas peuvent arriver dans une récursion. S'il s'agit d'une récursion d'ordre zéro (i.e., de flot) alors, soit le premier élément du couple `ve,se' = vf ve se` dépend de `ve` et nous avons une *boucle de causalité* (qui se traduit par une boucle infinie en HASKELL), soit il n'en dépend pas et l'exécution réussit. Par exemple, l'équation `x = x + (const 1)` n'est pas causale puisque `x` dépend instantanément de lui-même. S'il s'agit d'une récursion fonctionnelle alors le caractère borné de l'exécution peut dépendre de conditions sémantiques (comme c'était le cas pour l'exemple du crible d'Eratosthene du chapitre 2) [6].

5.3 Combinateurs de compilation

Les combinateurs décrits dans la section précédente ont peu d'intérêt pour exécuter les programmes et ils n'ont pas été conçus pour cela. Vus comme des *combinateurs de compilation*, c'est à dire comme autant de règles de compilation, ils servent à obtenir la fonction de transition associée à un programme. Le code de cette fonction de transition est écrit dans un langage de plus bas niveau (OCAML dans le cas de LUCID SYNCHRONE).

Par exemple, l'expression `pre 0 (const 1 + pre 2 (x * y))` se compile aisément en :

```
CoF((λ (memo1, memo2).memo1, (1 + memo2, x * y)), (0, 2))
```

où `+` et `*` sont les opérations scalaires usuelles. Cette compilation peut être obtenue par simple exécution du programme. Chaque combinateur se comporte comme un transformateur de code compilé : lorsqu'il est appliqué au code compilé pour produire la valeur de son entrée, il rend le code compilé pour produire la valeur de son résultat.

Nous dérivons les règles de compilation en remarquant que la fonction de transition (t) d'un processus $\mathtt{Co_F}\, t\, i$ (dont le type est $S \to V \times S$) peut s'écrire sous la forme : (λ `sig.` `let` d `in` (v`,`st)) où sig (de type S) désigne la liste des variables d'état, d est une liste d'équations servant au calcul de la valeur de la fonction de transition (v) et au nouvel état (st). Nous formulons de nouvelles définitions pour les opérateurs dont les valeurs s'écrivent cette fois : $\mathrm{Co}(sig, d, v, st, i)$. Les définitions sont données dans la figure 5.3.

Nous utilisons les notations suivantes : [] désigne une liste vide ; $\bullet$ est la concaténation de listes ; $x + l$ est une liste dont la tête est x et le reste est l. $Vars(d)$ désigne la liste des variables libres d'une liste d'équations d. $Vars(sig)$ désigne la liste des variables apparaissant dans la liste sig.

[5] Ce mot clef est inutile est HASKELL car les définitions sont récursives par défaut.

[6] Dans le compilateur de LUCID SYNCHRONE, l'analyse de causalité est limitée au équations de flots et ne s'applique pas aux récursions fonctionnelles.

$$
\begin{array}{lcl}
\texttt{co\_const}\, v & = & \mathrm{Co}([],[],v,[],[]) \\
\texttt{co\_extend}\ \mathrm{Co}(sig_1,d_1,v_1,s_1,i_1) & = & \mathrm{Co}\ \ (sig_1 \bullet sig_2 \\
\qquad\qquad\ \ \mathrm{Co}(sig_2,d_2,v_2,s_2,i_2) & & \qquad ,d_1 \bullet d_2 \\
 & & \qquad ,v_1\,(v_2), s_1 \bullet s_2 \\
 & & \qquad ,i_1 \bullet i_2) \\
\texttt{co\_pre}\, v_0\ \mathrm{Co}(sig,d,v,s,i) & = & \mathrm{Co}\ \ (m + sig \\
 & & \qquad ,d \\
 & & \qquad ,m, v + s \\
 & & \qquad ,v_0 + i) \\
 & & \text{si } m \notin Vars(d) \cup Vars(sig) \\
\texttt{co\_var}\, x & = & \mathrm{Co}([],[],x,[],[]) \\
\texttt{co\_lambda}\, x\ (\mathrm{Co}(sig,d,v,s,i)) & = & \mathrm{Co}\ \ ([] \\
 & & \qquad ,[] \\
 & & \qquad ,\lambda x, st.\texttt{let}\,(sig = \ \texttt{case}\, st\, \texttt{of} \qquad) + d\ \ ,[] \\
 & & \qquad\qquad\qquad\qquad\quad\ \texttt{|}\ Nil\ \texttt{->}\ i \\
 & & \qquad\qquad\qquad\qquad\quad\ \texttt{|}\ St(m)\ \texttt{->}\ m \\
 & & \qquad\qquad\quad \texttt{in}\, v, St(s) \\
 & & \qquad ,[]) \\
 & & \text{si } st, m \notin Vars(d) \cup Vars(sig) \\
\texttt{co\_apply}\ \mathrm{Co}(sig_1,d_1,v_1,s_1,i_1) & = & \mathrm{Co}\ \ (m + sig_1 \bullet sig_2 \\
\qquad\qquad\ \mathrm{Co}(sig_2,d_2,v_2,s_2,i_2) & & \qquad ,d_1 \bullet d_2 \bullet (v, m' = v_1\,(v_2\,(m))) \\
 & & \qquad ,v, m' + s_1 \bullet s_2 \\
 & & \qquad ,Nil + i_1 \bullet i_2) \\
\texttt{co\_let}\, x\ \mathrm{Co}(sig_1,d_1,v_1,s_1,i_1) & = & \mathrm{Co}\ \ (sig_1 \bullet sig_2 \\
\qquad\qquad\ \mathrm{Co}(sig_2,d_2,v_2,s_2,i_2) & & \qquad ,d_1 \bullet (x = v_1) + d_2 \\
 & & \qquad ,v_2, s_1 \bullet s_2 \\
 & & \qquad ,i_1 \bullet i_2) \\
 & & \text{si } x \notin Vars(d_1) \cup Vars(sig_1) \\
\texttt{co\_rec}\, x\ \mathrm{Co}(sig,d,v,s,i) & = & \mathrm{Co}\ \ (sig \\
 & & \qquad ,(x = v) + d \\
 & & \qquad ,x, s \\
 & & \qquad ,i)
\end{array}
$$

Fig. 5.3 – Combinateurs de compilation

5.3.1 Ordonnancement dynamique et évaluation paresseuse

Ces règles de compilation produisent du texte qu'il est facile ensuite de fournir à un langage paresseux tel que HASKELL. Le résultat $\mathrm{Co}(sig, d, v, st, i)$ est traduit vers un programme HASKELL de la forme (`CoF (λ sig. let` d `in (`v`,`st`))` i). d est un système d'équations mutuellement récursives qui doit être évalué de manière paresseuse. Par exemple, le programme :

```
let rec x =
    let m = x + (const 2)
    let p = pre 1 x + 4 in
    let nat = pre 2 nat + (const 1) in
    p + nat + pre 3 m + i
```

est traduit vers le code HASKELL suivant [7] :

```
CoF (λ (memo1, memo2, memo3). let { m = x + 2 ;
                                     p = memo1 + 4 ;
                                     nat = memo2 + 1 ;
                                     x = p + nat + memo3 + i } in
                              x, (x, nat, m))
    (1, 2)
```

Ce programme est un programme paresseux valide dont l'ordre d'évaluation des équations ne suit pas celui de leur définition (`m` doit être calculé après `x` qui doit être calculé après `p` et `nat`).

Le résultat de la compilation se traduit aisément dans un langage strict (e.g., OCAML) en incorporant des instructions de gestion de la paresse pour l'évaluation des équations mutuellement récursives.

5.3.2 ou ordonnancement statique et évaluation stricte

L'exécution paresseuse de la fonction de transition présente peu d'intérêt. Pour des raison d'efficacité (et rester cohérent avec la compilation de LUSTRE), il est préférable de traduire le code obtenu vers du code séquentiel non paresseux : un flot est représenté par une variable (au sens impératif du terme). A un instant donné, cette variable contient la valeur courante du flot et les équations doivent pouvoir être ordonnées statiquement. C'est le choix fait dans le compilateur LUCID SYNCHRONE. Le langage vers lequel les programmes sont traduits est OCAML.

Le système d'équations mutuellement récursives d doit pouvoir être transformé en une séquence de définitions de valeurs pouvant être écrite en OCAML. Ceci est obtenu par un calcul classique des composantes fortement connexes [100]. Après un tel ordonnancement, le programme précédent s'écrit :

```
CoF (λ (memo1, memo2, memo3). let p = memo1 + 4 in
                              let nat = memo2 + 1 in
                              let x = p + nat + memo3 + i in
```

[7] Pour rester cohérent avec les notations précédentes, nous conservons le symbole d'abstraction λ. En HASKELL, `let {` p_1 `=` e_1 `;...;`p_n `=` e_n `} in` e est une définition locale de plusieurs valeurs mutuellement récursives.

```
                                   let m = x + 2 in
                                   x, (x, nat, m))
    (1, 2)
```

Le corps de la fonction de transition s'évalue séquentiellement, sans paresse.

La traduction vers du code séquentiel obtenu par un ordonnancement des équations est faite de manière modulaire : une fonction est compilée une fois pour toute indépendamment de ses différentes utilisations. Cet ordonnancement est réalisé au moment de la compilation du corps d'une fonction.

L'ordonnancement statique n'est pas toujours possible. Considérons le programme suivant :

```
let f g =
    let rec o = g (o + (const 1)) in o

let m = f (fun x -> (const 0) fby o + (const 2))
```

`m` est le flot des entiers naturels plus grands que `1`. Dans `f`, le flot `o` est défini récursivement. Le code compilé de `f` s'écrit, grosso-modo :

```
CoF (λ g,state. let { o = g (o + 1) state ; }
                in o
    Nil
```

Dans le code obtenu, la récursion `o = g (o + 1) state` est une récursion sur une valeur scalaire. Ce programme s'évalue de manière paresseuse mais ne peut s'évaluer de manière séquentielle (stricte). La présence de l'appel à `g` ajoute un *faux* cycle de dépendances. Ici, plusieurs approches sont envisageables :

- rejeter statiquement `f` et les programmes pour lesquels le code ne peut être ordonné statiquement ;
- s'en remettre à une exécution paresseuse des programmes ;
- déplier statiquement tous les appels de fonctions ;
- déplier seulement les fonctions contenant des équations récursives ne pouvant être ordonnées statiquement.

La première approche est celle adoptée dans le compilateur de Lucid Synchrone. Dans la première version du compilateur, l'analyse de causalité était effectuée *à posteriori* : un programme était causal si le système d'équations associé pouvait être ordonné statiquement. Les analyses de causalité développées par la suite ont consisté à rejeter les programmes ne pouvant pas être traduits vers du code séquentiel. Cette approche oblige a rejeter des programmes qui ne contiennent pourtant pas de boucles de causalité (au sens du chapitre 4) parce que les calculs ne peuvent être ordonnés statiquement. Les trois autres approches, elles, permettent d'évaluer les programmes sans boucle de causalité (ayant passé l'analyse décrite dans [27, 28]). La deuxième approche a été expérimentée par Pascal Cuoq et est décrite dans sa thèse. La troisième approche est celle choisie par le compilateur académique de Lustre ou les langages de description de circuits synchrones (e.g., Lava [12]). Toute la fonctionnalité est éliminée statiquement. Nous n'avons pas expérimenté la quatrième approche. Une autre approche consiste à annoter les fonctions devant être dépliées statiquement (c'est le choix suivi par Scade) et à conditionner l'analyse de causalité en conséquence. Dans ce cas, une récursion (rec o = $f(o)$) où ($f(o) = e$) n'est pas rejetée dès lors que (rec o = e) peut être ordonné statiquement.

L'ordonnancement statique vers du code séquentiel nécessite d'ajouter des délais supplémentaires dans les récursions ou de réécrire les programmes. Ainsi, dans le programme :

```
let f g =
    let rec o = g (((const 0) fby o) + (const 1)) in o
let m = f (fun x -> x + (const 2))
```

f peut être compilée séparément vers du code séquentiel.

L'ordonnancement statique peut être source d'inefficacité. Par exemple :

```
let f x =
    let rec nat = pre 0 nat + (const 1) in
    x fby nat
let rec nat2 = pre 1 nat2 + 1 in
f nat2
```

Le calcul de nat2 dans (f nat2), nécessaire seulement au premier instant, est effectué à chaque instant.

Le choix d'un mode d'évaluation strict a cependant plusieurs avantages. La compilation est rapide, modulaire et les tests que nous avons réalisés montrent que le code généré est bien plus efficace que le code paresseux. Surtout, de nombreuses optimisations peuvent être incorporées à la compilation et le code donné au compilateur OCAML est mieux optimisé que sa version paresseuse. L'absence de paresse au cours de l'exécution permet de déterminer *syntaxiquement* ce qui est calculé à chaque instant et facilite l'écriture et la compréhension des programmes. En pratique, l'ordonnancement statique des récursions (imposant parfois l'ajout de délais supplémentaires) n'est jamais apparu gênant. Finalement, un mode d'évaluation strict se prête mieux à l'utilisation d'opérations importées de OCAML et effectuant des effets de bord.

5.4 Discussion

Flots et objets : Grâce à la sémantique co-itérative, on peut établir un lien entre la définition d'une fonction de flot synchrone et la définition d'une classe dans un langage objet, et donc entre flot et objet. Le lien entre flots et les objets (en utilisant des techniques venant des co-algèbres) a été étudié par Jacobs [56]. Une fonction synchrone de flot définit une classe particulière ayant une seule méthode (*step*) de calcul de la réaction et un ensemble de variables d'instance définissant l'état interne. L'application de cette fonction correspond à la création d'un objet de la classe. Une fonction f de type $\mathit{Stream}(t_1) \rightarrow \mathit{Stream}(t_2)$ et d'implantation (co_lambda x $(\mathrm{Co}((s_1, ..., s_k), d, v, (st_1, ..., st_k), (i_1, ..., i_k))))$ définit une classe (écrite dans une syntaxe OCAML) :

```
class f =
      object
      mutable val s1 = i1 ;
      ...
      mutable val sk = ik ;
```

```
      method step (x : t1) =
                  let d in
                  s1 <- st1 ;
                  ...
                  sk <- stk ;
                  v
end
```

Lorsqu'une fonction de flots est instanciée, c'est à dire appelée en un point de programme, elle définit un objet de la classe qui lui est associée.

Cette relation entre fonctions de flots et classes peut être exploitée au point de proposer un compilateur traduisant les programmes LUSTRE vers des programmes objet écrits en OCAML : les noeuds LUSTRE sont traduits vers des classes et les objets sont créés à chaque point d'appel de noeud. Une traduction utilisant les objets a été expérimentée pour SCADE par Jean-Louis Colaço.

Des expériences d'extension de langages objets avec des traits synchrones ont été menées par Poigné & all. [85, 17].

Modification en place des états et des flots, valeurs constantes : Dans un processus $\mathtt{Co_F}\, t\, i$, l'état peut être représenté par un enregistrement et modifié physiquement. Une représentation à base d'enregistrements (avec champs modifiables) a été utilisée dans la version `1.0` du compilateur. La version `2.0` représente l'état par des références. De même, les arguments (et les résultats) d'une fonction sont stockés dans l'état de la fonction et représentés par des références. Ceci évite de les repasser en paramètre à chaque réaction. De manière très schématique, une fonction de flots ($\mathtt{co\_lambda}\, x\, (\mathrm{Co}((s_1, ..., s_k), d, v, (st_1, ..., st_k), (i_1, ..., i_k))))$ est représentée par :

```
let f x =
    let s1 = ref i1 in
    ...
    let sk = ref ik in
    fun () -> let x = !x in
              let d in
              s1 <- st1 ;
              ...
              sk <- stk ;
              v
```

Le code constant (valeurs immédiates, fonctions) est remonté du corps de la fonction de transition pour n'être évalué qu'une seule fois.

Caractérisation d'un sous-ensemble temps-réel : Il est facile de trouver des conditions suffisantes caractérisant un sous-ensemble temps-réel du langage : les programmes temps-réel sont ceux dont les récursions sont gardées par des délais — les programmes ont passé l'analyse de causalité — et qui ne contiennent pas de récursion fonctionnelle. Dans le compilateur LUCID SYNCHRONE, la récursion fonctionnelle est rejetée par un argument de typage (une option de compilation permet de rejeter les récursions fonctionnelles).

Typage concret et mémoire utilisée : Le choix d'utiliser un langage fonctionnel typé comme langage cible de la compilation s'est révélé très avantageux. En plus de la possibilité de faire communiquer du code écrit en OCAML et en LUCID SYNCHRONE, le typage "supplémentaire" effectué par OCAML sur le code compilé facilite grandement la mise au point du compilateur. Les phases de compilation liées à la fonctionnalité (e.g., lambda-lifting, représentation des données) sont confiées à OCAML, allégeant d'autant l'écriture du compilateur [8].

On peut remarquer que les combinateurs de compilation (et le code qu'ils génèrent) sont typés. Les types des expressions écrites en utilisant ces combinateurs s'apparentent à des types avec *effets* [103]. Le type *concret* d'une expression e est $Cstream(t, S)$, c'est à dire un couple donnant le type au sens classique et un effet S déterminant ici la mémoire utilisée pour le calcul. Les règles de typage (synthétisées par le compilateur HASKELL) pour les combinateurs `co_lambda`, `co_apply` et `co_rec` sont très voisines de règles *ad-hoc* de typage de l'abstraction, de l'application et de la récursion d'un système de types avec effets. Par exemple, le type de `co_lambda` correspond à la règle classique de typage :

$$\frac{H, x : t_1 \vdash e : t_2, S'}{H \vdash \lambda x.e : t_1 \stackrel{S'}{\rightarrow} t_2, S}$$

en notant $t_1 \stackrel{S'}{\rightarrow} t_2$ pour $t_1 \rightarrow State(S') \rightarrow t_2 \times State(S')$. S est une variable d'effet non liée (et pouvant être généralisée). Elle exprime le fait qu'une abstraction n'utilise pas de mémoire.

Les types synthétisés indiquent la mémoire nécessaire au calcul. En présence de fonctions récursives, les programmes nécessitent une mémoire non bornée : les programmes ne peuvent être typés sans avoir recours à une synthèse de types cycliques (option `rec_type` de OCAML).

Autres implantations des combinateurs : Nous n'avons pas cherché à construire un jeu de combinateurs qui soient efficaces à tout prix. Le but était avant tout de déterminer un jeu de constructions minimal et se composant bien. Nous souhaitions aussi que ces combinateurs soient typés et écrits dans un langage purement fonctionnel.

D'autres implantations sont donc possibles. Le test à `Nil` effectué à chaque instant lors d'une réaction est un test coûteux et peut être évité. Nous donnons ici une définition des combinateurs (figure 5.4) dans le cas où le langage est de premier ordre (LUSTRE) [9].

Fonctions ne préservant pas les longueurs : Les fonctions ne préservant pas les longueurs peuvent être traitées en ajoutant une valeur explicite marquant l'absence comme nous l'avons fait au chapitre 3. Cette partie est détaillée dans [22].

En pratique, la représentation explicite de l'absence à l'exécution est inutile et coûteuse puisque toutes les expressions savent en utilisant l'horloge si leur valeur est présente ou pas. Les différentes versions du compilateur ont utilisé l'information d'horloge pour éliminer les tests de présence/absence des flots et pour exécuter une expression seulement lorsque son horloge est vraie. Les instructions portant la même horloge sont rassemblées sous la même conditionnelle.

[8]Un compilateur partiel générant du code C a également été écrit pour pouvoir programmer des robots LEGO. J'ai conscience que la compilation vers un langage de plus bas niveau tel que C est un passage obligé pour que le langage puisse être utilisé pour programmer des applications temps-réel traditionnelles.

[9]Le code produit par le compilateur de LUCID SYNCHRONE évite également ce test à *nil*.

$$
\begin{array}{lcl}
\texttt{co\_lambda}\, x\ (\mathrm{Co}(sig, d, v, s, i)) & = & \mathrm{Co}\ \ ([] \\
 & & \quad ,[] \\
 & & \quad ,\lambda x, st.\texttt{let}\, d \quad ,[] \\
 & & \qquad\qquad \texttt{in}\, v, s \\
 & & \quad ,([], i)) \\
 & & \text{si } st, m \notin Vars(d) \cup Vars(sig) \text{ et } x \notin Vars(i) \\
 & & \\
\texttt{co\_apply}\ \ \mathrm{Co}(sig_1, d_1, v_1, s_1, (i_1, i)) & = & \mathrm{Co}\ \ (m + sig_1 \bullet sig_2 \\
\qquad\qquad\quad \mathrm{Co}(sig_2, d_2, v_2, s_2, i_2) & & \quad , d_1 \bullet d_2 \bullet (v, m' = v_1\,(v_2\,(m))) \\
 & & \quad , v, m' + s_1 \bullet s_2 \\
 & & \quad , i + i_1 \bullet i_2)
\end{array}
$$

FIG. 5.4 – Une autre implantation des combinateurs de compilation

Chapitre 6

Conclusion

Nous terminons ce document par une comparaison de notre travail avec des travaux qui lui sont reliés et discutons des perspectives.

6.1 Travaux connexes

Le concept de synchronisme est apparu dans le domaine des systèmes réactifs pour qualifier les systèmes parallèles avançant pas à pas (*locked-step mode*). Les processus mis en parallèles avancent de manière synchrone, en effectuant chacun une transition [71]. L'intérêt de son étude dans un cadre flot de données est double :

- les systèmes réactifs ont un comportement infini parfaitement pris en compte par des notions de co-algèbres et de types co-inductifs [57]. Il ne semble cependant pas que l'idée de fonction synchrone ait été identifiée dans ce contexte ;
- les langages de programmation flot de données (Lustre, Signal) font partie des langages synchrones et ils sont largement utilisés. Aucun de ces langages ne contient cependant de notions d'ordre supérieur, et plus généralement de traits présents dans les langages fonctionnels typés (e.g., synthèse des types, polymorphisme).

Boussinot [15] propose un modèle d'exécution de réseaux de processus, consistant en une exécution (synchrone) d'une étape de chaque processus à chaque instant. La sémantique opérationnelle est donc assez proche de la notre. Cependant, il n'y a pas de calcul d'horloges ou de méthode de compilation qui permettent de compiler les communications de manière efficace.

Dans [94], Pucella propose une extension de Standard ML avec des traits réactifs issus de Reactive-C [14]. Cette intégration permet de bénéficier des traits de haut niveau du langage hôte (e.g., ordre supérieur, types de données, synthèse des types). Cette approche diffère cependant de la notre dans la mesure où elle est basée sur un style de programmation impératif *à la* Esterel plutôt que sur un style flot de données ; les constructions réactives sont interprétées dans le langage hôte alors qu'elles sont compilées en Lucid Synchrone ; il n'y a non plus de garanties sur le comportement du programme.

L'idée de la programmation flot de données synchrone peut être reliée à l'idée d'exécution n'utilisant pas de structure intermédiaire (*listlessness* et *deforestation*) [109]. La deforestation consiste à éliminer les instructions d'allocation et de désallocation des structures intermédiaires dans les programmes. De nombreuses méthodes de deforestation ont été proposées par la suite [38, 37, 61]. Cependant, la deforestation de programmes d'ordre supérieurs a été peu étudié.

Notre travail est également relié aux langages et outils de modélisation de circuits synchrones dans un cadre purement fonctionnel.

Dans DAISY [58], un circuit synchrone est défini par un ensemble d'équations (mutuellement récursives). Par rapport à LUCID SYNCHRONE, tous les flots présents dans le programme partagent la même horloge de base et les programmes sont évalués de manière paresseuse.

Plus récemment, plusieurs outils et librairies ont été proposés pour décrire des circuits à l'intérieur du langage fonctionnel généraliste HASKELL (e.g, HYDRA [78], LAVA [12], HAWK [51]). Les circuits sont modélisés directement en HASKELL en définissant des librairies d'opérateurs de flots (e.g., registre, flip-flop) et des opérations pour convertir les valeurs scalaires en des flots. Les équations de flot et les fonctions de flot s'écrivent d'une manière proche de celle qui est utilisée dans les langages synchrones flot de données. L'utilisation de HASKELL et de son système de type permettent de changer aisément de représentation des flots pour obtenir une vérification de propriété, une simulation ou une compilation vers un réseau d'opérateurs (*net-list*). Ce réseau peut ensuite être compilé vers du code séquentiel. Cependant, le code obtenu est peu satisfaisant à cause de la croissance du code due au dépliage systématique et à la disparition de la structure de contrôle du programme source. Il est peu probable que les informations sur la structure puissent être retrouvées dans le réseau afin de générer du code efficace.

Aucun de ces outils de dispose de notion d'horloge (toutes les opérations préservent les longueurs), de structure de contrôle, d'analyses modulaires facilitant l'écriture des programmes ou de méthode de compilation. Ces caractéristiques sont centrales dans notre approche. Ces langages restent avant tout dédiés à la conception de circuits synchrones alors que LUCID SYNCHRONE vise une compilation vers du code séquentiel.

FRAN [33] (pour *Functional Reactive Animation*) est une librairie bâtie au dessus de HASKELL et dédiée à la modélisation d'animations graphiques. FRAN est basée sur les notions de *comportement* et d'*événement*. Un comportement est une structure évoluant au cours du temps et de manière continue alors qu'un événement évolue de manière discrète. Ces structures sont implantées en HASKELL par des flots paresseux (précisément, des flots indicés par le temps pour les comportements et des flots de valeurs présentes/absentes pour les évènements). Les concepts introduits dans FRAN ont ensuite été repris dans FRP [110, 111] (pour *Functional Reactive Programming*), une librairie dédiée à la programmation réactive en HASKELL. La manière d'écrire les programmes en FRAN (et en FRP) est voisine de celle de LUCID SYNCHRONE [1]. FRAN et LUCID SYNCHRONE poursuivent cependant des objectifs différents. FRAN est avant tout un langage de modélisation des systèmes réactifs. Il étend HASKELL avec des structures pour parler du temps et récupère son modèle d'évaluation. Il n'y a pas, en FRAN, d'analyse statique, de méthode de compilation ni de garantie sur le caractère réactif du programme généré par la compilation. LUCID SYNCHRONE a été conçu comme un langage pour programmer les systèmes réactifs et il est donc un (petit) sous-ensemble de HASKELL. D'où la nécessité de disposer de systèmes de types, d'analyses de programmes et de méthodes de compilation. Le compilateur de LUCID SYNCHRONE va empêcher d'écrire

[1]Au point que, lors d'une visite à Kent, j'ai retranscrit directement en LUCID SYNCHRONE l'exemple de l'ascenseur écrit par Simon Thompson en FRAN [105]. Cette transcription a révèle que le programme initial contenait deux boucles de causalité !

de nombreux programmes qui peuvent être écrits en FRAN mais pour lesquels il n'est pas capable de garantir la bonne exécution (e.g., sans fuites de mémoire, sans boucle infinie). Pour paraphraser ce qui est écrit dans [27], *la différence essentielle entre* LUCID SYNCHRONE *et* FRAN *(et plus généralement* HASKELL*) est que nous rejetons plus de programmes !* En échange, le compilateur fournit une aide au programmeur et des garanties sur le code obtenu.

Récemment, un sous-ensemble de premier ordre de FRAN dédié au temps-réel a été identifié [111]. Une sémantique par réaction et une méthode de compilation proche des méthodes classiques des langages synchrones ont été proposés.

6.2 Discussion

Mon travail sur LUCID SYNCHRONE a débuté en 1995, à la suite de ma thèse, lors de mon arrivée à VERIMAG. A cette époque, Paul Caspi avait jeté les premières bases d'une extension de LUSTRE avec des traits fonctionnels. Ensemble, nous avons développé les fondements du langage. Le langage s'est ensuite développé à Paris 6 avec Grégoire Hamon et Pascal Cuoq.

Etant donné le caractère fonctionnel de LUSTRE, il était naturel de chercher à le rapprocher d'un langage fonctionnel. Cela nécessitait néanmoins de comprendre le synchronisme dans un cadre fonctionnel. Nous nous sommes d'abord appuyé sur l'analogie qui existe entre les techniques de déforestation proposées par Wadler pour les langages fonctionnels paresseux et les méthodes de compilation des langages à flots de données synchrones. Cependant, nous avons réalisé que les méthodes de deforestation existantes ne permettaient pas d'éliminer les listes intermédiaires de nombreux programmes facilement compilés par LUSTRE. La solution a été trouvée en utilisant les techniques venant des co-algèbres [57] permettant de faire le lien entre les flots et les automates et de donner une formulation simple à la compilation. Une fois le lien entre LUSTRE et ML établi, il était normal de rapprocher *calcul d'horloge* et *typage*. Ce calcul s'est avéré être un cas particulier de système de type dépendants. D'où l'idée, d'un coté de représenter les programmes LUCID SYNCHRONE par des termes typés de COQ, et de l'autre, de simplifier les horloges pour qu'elles se rapprochent des types de ML.

Cette généralisation a consisté à plonger les techniques classiques des langages synchrones dans un cadre fonctionnel général, en introduisant curryfication, abstraction, et application. Cela nous permet de donner un sens synchrone à des programmes d'ordre supérieur et de disposer ainsi d'un langage réactif strictement plus puissant que LUSTRE.

L'une des idées de départ était de concevoir un langage *laboratoire* pouvant être utilisé pour prototyper de nouvelles constructions. L'ordre supérieur et le polymorphisme permettent à l'utilisateur de définir son propre jeu d'opérateurs (e.g, la condition d'activation, des itérateurs, des reboucleurs de réseaux,...) et de les utiliser comme s'il s'agissait de constructions "cablées" dans le compilateur. Deux expériences de prototypage ont été menées :

- une traduction des automates de modes vers LUCID SYNCHRONE [48, 46]. Cette traduction repose sur l'utilisation d'une opération de réinitialisation, du `merge` (et de son extension n-aire). Le compilateur LUCID SYNCHRONE génère du code comparable en efficacité et en taille au code généré par la méthode de compilation dédiée proposée dans [66].
- un prototypage d'itérateurs de tableaux dans le cadre des extensions de LUSTRE (contrat GENIE II [35]). Les opérations de manipulation de tableau ont pu été ajoutées, avec peu d'effort, sous forme d'une librairie. Cette expérience a permis d'illustrer l'intérêt du polymorphisme de la compilation séparée.

De nombreux exemples de programmes ont été écrits au cours de ces années, le plus souvent par des étudiants effectuant des stages dans l'équipe et sur des exemples n'entrant pas dans le domaine classique du contrôle/commande (application graphiques, jeux). Ayant besoin d'un outil de simulation graphique de programmes écrits dans le langage, nous avons eu l'idée de le programmer directement dans le langage, à partir d'une librairie de composants graphiques (e.g., boutons, boites graphiques, lecture du clavier,...) tous définis en Lucid Synchrone. Ce travail a été publié dans [47].

Un nouveau compilateur de Scade (la version industrielle de Lustre) a été développé par la société Telelogic (et maintenant à Esterel Technologies) par Jean-Louis Colaço. Ce compilateur, appelé ReluC (*Retargatable Lustre Compiler*) s'est fondé en partie sur les travaux théoriques et l'implantation de Lucid Synchrone. Il implante un calcul d'horloge basé sur celui de Lucid Synchrone. L'analyse d'initialisation a été conçu conjointement et implantée à la fois dans le compilateur ReluC et dans celui de Lucid Synchrone. Ce compilateur a été expérimenté sur des exemples industriels réels (plusieurs dizaines de milliers de lignes de code).

Le langage est encore jeune et de nombreux problèmes restent à étudier. En particulier, une caractérisation des programmes temps-réel, i.e. s'exécutant en temps et mémoire bornée, reste à faire.

La récursion fonctionnelle de flot permet de décrire des systèmes dont la taille augmente au cours du temps, autrement dit des réseaux de Kahn dynamique (e.g., pile et files cablées, crible d'Eratosthene). Elle permet de retrouver, dans un cadre flot de données, certains des traits présents dans les SugarCubes de Boussinot et Susini [16]. Son étude approfondie reste à faire.

L'étude du calcul d'horloge et de ses extensions n'est pas achevée : le calcul d'horloge simplifié reste moins expressif que le calcul initial et son extension avec des horloges fonctionnelles polymorphes ou quantifiées existentiellement n'est pas claire encore.

Plusieurs structures de contrôle ont été expérimentées. [22] et [48] proposent deux approches différentes pour ajouter une opération de réinitialisation modulaire. Une construction de mise en séquence proche de la construction proposée dans la fin du chapitre 3 est intégrée au compilateur. Cependant, l'interaction entre horloge et structures de contrôle reste un problème difficile (il n'y a qu'à voir à quel point il est difficile d'ajouter les "modes" de Maraninchi et Rémond à Lustre sans le dénaturer complètement). L'ajout de flots finis (pour décrire des processus qui s'arrêtent) et leur compilation en code séquentiel efficace reste également à faire.

Le travail de Boulmé et Hamon [13] montre que les programmes de Lucid Synchrone peuvent être traduits facilement vers des termes Coq dont la vérification (du point de vue des types, des horloges, etc.) est prise en charge par Coq. Coq prend donc ici la place qu'avait Haskell au début de nos travaux. Deux extensions naturelles de ce travail peuvent être considérées. La première concerne la certification d'un compilateur. Le compilateur se charge de la partie "algorithmique" (e.g, synthèse des types, des horloges) et produit des termes annotés dont la vérification est effectuée par Coq. La seconde concerne la vérification et l'extraction de programmes synchrones en les fondant sur la notion "horloge = type".

Enfin, j'espère que ce travail aura contribué à réduire le fossé entre programmation réactive et programmation fonctionnelle d'usage général en montrant que plusieurs notions de l'une étaient souvent des cas particuliers de l'autre.

,lustre

Bibliographie

[1] Alexander Aiken and Edward Wimmers. Type inclusion constraints and type inference. In *Seventh Conference on Functional Programming and Computer Architecture*, pages 31–41, Copenhagen, Denmark, June 1993. ACM.

[2] Alexander Aiken, Edward Wimmers, and Jens Palsberg. Optimal Representations of Polymorphic Types with Subtyping. In *Theoretical Aspects of Computer Software (TACS)*, September 1997.

[3] Roberto M. Amadio and Solange Coupet-Grimal. Analysis of a guard condition in type theory. In *International Conference on Foundation of Software Science and Computation Structures (FoSSACS)*, volume 1378 of *Lecture Notes in Computer Science*, pages 48–62, Lisboa, Portugal, 1998.

[4] Charles André. Representation and Analysis of Reactive Behaviors : A Synchronous Approach. In *CESA*, Lille, july 1996. IEEE-SMC. Available at : `www-mips.unice.fr/~andre/synccharts.html`.

[5] E. A. Ashcroft and W. W. Wadge. *Lucid, the data-flow programming language*. Academic Press, 1985.

[6] E.A. Ashcroft and W.W. Wadge. Lucid, a formal system for writing and proving programs. *SIAM j. Comp.*, 3 :336–354, 1976.

[7] E.A. Ashcroft and W.W. Wadge. Lucid, a non procedural language with iteration. *Communications of the ACM*, 20(7) :519–526, 1977.

[8] Jean-Philippe Beaujard. Using SCADE to Develop Flight Control Systems. The A 340-600 experience Airbus France. Scade User Conference, Toulouse, june 2002.

[9] A. Benveniste, P. LeGuernic, and Ch. Jacquemot. Synchronous programming with events and relations : the SIGNAL language and its semantics. *Science of Computer Programming*, 16 :103–149, 1991.

[10] G. Berry. Real time programming : Special purpose or general purpose languages. *Information Processing*, 89 :11–17, 1989.

[11] G. Berry and G. Gonthier. The Esterel synchronous programming language, design, semantics, implementation. *Science of Computer Programming*, 19(2) :87–152, 1992.

[12] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava : Hardware design in haskell. In *International Conference on Functional Programming (ICFP)*. ACM, 1998.

[13] Sylvain Boulmé and Grégoire Hamon. Certifying Synchrony for free. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 2250, La Havana, Cuba, December 2001. Lecture Notes in

Artificial Intelligence, Springer-Verlag. Short version of *A clocked denotational semantics for Lucid-Synchrone in Coq*, available as a Technical Report (LIP6), at `http ://www-spi.lip6.fr/lucid-synchrone`.

[14] F. Boussinot. Reactive C : An Extension of C to Program Reactive Systems. *Software Practice and Experience*, 21(4) :401–428, 1991.

[15] F. Boussinot. Réseaux de processus réactifs. Technical Report 1588, INRIA Sophia-Antipolis, janvier 1992.

[16] Frédéric Boussinot and Jean-Ferdinand Susini. The sugarcubes tool box : A reactive java framework. *Software-Practice and Experience*, 28(14) :1531–1550, December 1998. Url : `http ://www.inria.fr/meije/rc/rc-project.html`.

[17] Reinhard Budde, G. Michele Pinna, and Axel Poigné. Coordination of synchronous programs. In *International Conference on Coordination Languages and Models*, number 1594 in Lecture Notes in Computer Science, 1999.

[18] P. Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94 :125–140, 1992.

[19] Paul Caspi and Marc Pouzet. A Functional Extension to Lustre. In M. A. Orgun and E. A. Ashcroft, editors, *International Symposium on Languages for Intentional Programming*, Sydney, Australia, May 3-5 1995. World Scientific.

[20] Paul Caspi and Marc Pouzet. Synchronous Kahn Networks. In *ACM SIGPLAN International Conference on Functional Programming*, Philadelphia, Pensylvania, May 1996.

[21] Paul Caspi and Marc Pouzet. A Co-iterative Characterization of Synchronous Stream Functions. In *Coalgebraic Methods in Computer Science (CMCS'98)*, Electronic Notes in Theoretical Computer Science, 28-29 March 1998. Extended version available as a VERIMAG tech. report no. 97–07)) at `www-spi.lip6.fr/~pouzet`.

[22] Paul Caspi and Marc Pouzet. Lucid Synchrone, a functional extension of Lustre. submitted to publication, 2001.

[23] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transaction on Programming Languages and Systems*, 8(2) :244–263, 1986.

[24] Jean-Louis Colaço. Smantique statique de lustre v6. Esterel Technologies, 2001.

[25] Jean-Louis Colaço and Marc Pouzet. Type-based Initialization of a Synchronous Dataflow Language. In *Synchronous Languages, Applications, and Programming*, volume 65. Electronic Notes in Theoretical Computer Science, 2002.

[26] The coq proof assistant, 2002. `http ://coq.inria.fr`.

[27] Pascal Cuoq. *Ajout de synchronisme dans les langages fonctionnels typés.* PhD thesis, Université Pierre et Marie Curie, Paris, France, october 2002.

[28] Pascal Cuoq and Marc Pouzet. Modular causality in a synchronous stream language. In *European Symposium on Programming (ESOP'01)*, Genova, Italy, April 2001.

[29] R. Cytron and J. Ferrante. What's in a name ? In *International Conference on Parallel Processing*, pages 19–27, August 1987.

[30] E. A. de Kock, G. Essink, W.J.M. Smits, P. van der Wolf, J.-Y. Brunel, W.M. Kruijtzer, and P. Lie. YAPI : Application Modeling for Signal Processing Systems. In *Design Automation Conference (DAC)*, 2000.

[31] J. Peterson (editor). The haskell library report. Technical report, Yale University, 1996.

[32] Jr Edmund M Clarke, Orna Grumberg, and Doron A Peled. *Model Checking.* The MIT Press, 1999.

[33] C. Elliot and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming (ICFP).* ACM, 1997.

[34] Conal Elliott. An embedded modeling language approach to interactive 3d and multimedia animation. *IEEE Transactions on Software Engineering*, 25(3) :291–308, May-June 1999.

[35] Jean-Louis Colaço et Marc Pouzet. Prototypages. Rapport final du projet GENIE II, Verilog SA, Janvier 2000.

[36] J. T. Feo, D. C. Cann, and R. R. Oldehoeft. A report on the Sisal language project. *Journal of Parallel and Distributed Computation*, 10 :349–366, 1990.

[37] A. Gill and S.L. Peyton Jones. Cheap deforestation in practice : An optimiser for haskell. In *Proceedings of the IFIP 13th World Computer Congress*, volume 1, pages 581–586, The Netherlands, August 28-September 1 1994. Elsevier Science Publishers.

[38] A. Gill, J. Launchbury, and S.L. Peyton Jones. A short cut to deforestation. In *6th Functional programming languages and computer architecture*, pages 223–232, Copenhagen, June 1993. ACM.

[39] Eduardo Gimenez. Structural recursive definitions in type theory. In *International Conference on Automata, Languages and Programming (ICALP)*, volume 1443 of *Lecture Notes in Computer Science*, pages 397–408, Denmark, July 1998. Springer-Verlag.

[40] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types. Cambridge Tracts in Theoretical Computer Science*, 7, 1989.

[41] N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer Academic Pub., 1993.

[42] N. Halbwachs. Synchronous programming of reactive systems, a tutorial and commented bibliography. In *Tenth International Conference on Computer-Aided Verification, CAV'98*, Vancouver (B.C.), June 1998. LNCS 1427, Springer Verlag.

[43] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9) :1305–1320, September 1991.

[44] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous dataflow language LUSTRE. *IEEE Transaction on Software Engineering*, 18(9) :785–793, September 1992. Available through anonymous ftp at `imag.fr :pub/SPECTRE/LUSTRE/PAPERS/lustre.tse.ps.gz`.

[45] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991.

[46] Grégoire Hamon. *Calcul d'horloge et Structures de Contrôle dans Lucid Synchrone, un langage de flots synchrones à la ML.* PhD thesis, Université Pierre et Marie Curie, Paris, France, september 2002.

[47] Grégoire Hamon and Marc Pouzet. Un simulateur synchrone pour lucid synchrone. In *Journées Francophones des Langages Applicatifs (JFLA)*, Morzine-Avoriaz, Février 1999. INRIA.

[48] Grégoire Hamon and Marc Pouzet. Modular Resetting of Synchronous Data-flow Programs. In *ACM International conference on Principles of Declarative Programming (PPDP'00)*, Montreal, Canada, September 2000.

[49] D. Harel. StateCharts : a Visual Approach to Complex Systems. *Science of Computer Programming*, 8-3 :231–275, 1987.

[50] David Harel and Amir Pnueli. On the development of reactive systems. In *Logic and Models of Concurrent Systems*, volume 13 of *NATO ASI Series*, pages 477–498. Springer Verlag, 1985.

[51] J. Hatthews, J. Launchbury, and B. Cook. Specifying microprocessors in hawk. In *International Conference on Computer Languages*. IEEE, 1998.

[52] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitive Approach*. Morgan Kaufmann Publishers Inc, 1994.

[53] John E Hopcroft and Jeffrey D Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.

[54] P. Hudak, S. Peyton Jones, and P. Wadler. Report on the programming language Haskell, a non strict purely functional language (version 1.2). *ACM SIGPLAN Notices*, 27(5), 1990.

[55] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *POPL*, 1996.

[56] Bart Jacobs. Coalgebraic reasoning about classes in object-oriented languages. In *Workshop Coalgebraic Methods in Computer Science (CMCS)*, volume 11 of *Electronic Notes in Computer Science*, 1998.

[57] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62 :222–259, 1997.

[58] Steven D. Johnson. Applicative programming and digital design. In *Principles of Programming Languages*, pages 218–227. ACM, 1984.

[59] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP 74 Congress*. North Holland, Amsterdam, 1974.

[60] Konstantin Läufer and Martin Odersky. An extension of ML with first-class abstract types. In *ACM SIGPLAN Workshop on ML and its Applications, San Francisco, California*, pages 78–91, June 1992.

[61] John Launchbury and Tim Sheard. Warm Fusion : Deriving Build-cats from Recursive Definitions. In *International Conference on Functional Programming Languages and Computer Architecture (FPCA)*, La Jolla, San Diego, June 25-28 1995. ACM Press.

[62] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers*, 36(2), 1987.

[63] Xavier Leroy. The Objective Caml system release 3.01. Documentation and user's manual. Technical report, INRIA, 2001.

[64] Xavier Leroy and François Pessaux. Type-based analysis of uncaught exceptions. *TOPLAS*, 2000.

[65] F. Maraninchi and Y. Rémond. Mode-automata : About modes and states for reactive systems. In *European Symposium On Programming*, Lisbon (Portugal), March 1998. Springer verlag.

[66] F. Maraninchi, Y. Rémond, and Y. Raoul. Matou : An implementation of mode-automata into dc. In *Compiler Construction*, Berlin (Germany), March 2000. Springer verlag.

[67] Florence Maraninchi. *Modélisation et validation des systèmes réactifs : un langage synchrone base d'automates.* PhD thesis, Université Joseph Fourier - Grenoble I, May 1997. Document d'Habilitation Diriger des Recherches.

[68] Florence Maraninchi and Yann Rémond. Argos : an automaton-based synchronous language. *Computer Languages*, (27) :61–92, 2001.

[69] Michel Mauny and Franois Pottier. An implementation of Caml Light with existential types. Technical Report 2183, INRIA, October 1993.

[70] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17 :348–375, 1978.

[71] Robin Milner. *Communication and Concurrency*, volume International Series in Computer Science. Prentice Hall, 1989.

[72] Robin Milner. *Communicating and Mobile Systems : the π-calculus.* Cambridge University Press, 1999.

[73] Paul S. Miner and Steven D. Johnson. Verification of an optimized fault-tolerant clock synchronization circuit. In Mary Sheeran and Satnam Singh, editors, *Designing Correct Circuits*, Electronic Workshops in Computing, Bastad, Sweden, September 1996. Springer-Verlag.

[74] John C. Mitchell. *Foundations for Programming Languages.* The MIT Press, 1996.

[75] Steven S Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[76] Alan Mycroft. Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet, editors, *Proceedings of the Sixth International Symposium on Programming, Toulouse*, pages 217–228. Springer-Verlag LNCS 167, April 1984.

[77] D. Nowak, J. R. Beauvais, and J. P. Talpin. Co-inductive axiomatisation of synchronous language. In *International Conference on Theorem Proving in Higher-Order Logics (TPHOLs'98).* Springer Verlag, October 1998.

[78] John O'Donnell. From transistors to computer architecture : Teaching functional circuit specification in hydra. In Springer-Verlag, editor, *Functional Programming Languages in Education*, pages 195–214, 1995.

[79] Gordon Pace. Haslu : An experimental version of an embedding of lustre in haskell. Tool available at `http ://www.inrialpes.fr/vasy/people/Gordon.Pace`, 2001.

[80] Prakash Panangaden and Vasant Shanbhogue. The Expressive Power of Indeterminate Dataflow Primitives. *Information and Computation*, 98(1), May 1992.

[81] Thomas M Park. *Bounded Scheduling of Process Networks.* PhD thesis, University of California at Berkeley, 1995.

[82] Christine Paulin-Mohring. Circuits as streams in Coq, verification of a sequential multiplier. Technical report, Laboratoire de l'Informatique du Parallélisme, September 1995. Available at `http ://www.ens-lyon.fr :80/LIP/lip/publis/`.

[83] Francois Pessaux. *Détection statique d'exceptions non rattrapées en Objective Caml.* PhD thesis, Université Pierre et Marie Curie (Paris 6), 2000.

[84] Gordon D. Plotkin. *A structural approach to operational semantics.* Daimi FN-19. University of Aarhus, Computer Science Department Aarhus University Ny Munkegade DK 8000 C Danemark, September 1981.

[85] Axel Poigné and Leszek Holenderski. On the combination of synchronous languages. In W.P.de Roever, editor, *Workshop on Compositionality : The Significant Difference*, volume LNCS 1536, pages 490–514, Malente, September 8-12 1997. Springer Verlag.

[86] François Pottier. Simplifying subtyping constraints. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 122–133, May 1996.

[87] François Pottier. Simplifying subtyping constraints : a theory. To appear in *Information & Computation*, August 2000.

[88] Marc Pouzet. *Compaction des langages Fonctionnels.* PhD thesis, Université Paris VII, October 1994.

[89] Marc Pouzet. Fine grain parallelisation of functional programs for VLIW or superscalars architectures. In *IFIP WG 10.3 International Conference on Applications in Parallel and Distributed Computing*, Caracas,Venezuela, April 1994.

[90] Marc Pouzet. The Program Compaction Revisited : the Functional Framework. In *International Conference on Parallel Processing (EURO-PAR'95)*, LNCS 966, Stockholm, Sweden, August 29-31 1995.

[91] Marc Pouzet. Une présentation fonctionnelle de la compaction de code. *Techniques et Sciences Informatiques (TSI)*, 15(7), 1996. Numéro spécial "langages applicatifs".

[92] Marc Pouzet. Using the parallel complexity of programs to improve compaction. In *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Boston, October 1996.

[93] Marc Pouzet. *Lucid Synchrone, version 2. Tutorial and reference manual.* Université Pierre et Marie Curie, LIP6, Mai 2001. Distribution available at : `www-spi.lip6.fr/lucid-synchrone`.

[94] Ricardo Pucella. Reactive programming in standard ml. In *IEEE International Conference on Computer Languages*, pages 48–57, 1998.

[95] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351, Berlin, 1982. Springer Verlag.

[96] Pascal Raymond. Compilation séparée de programmes lustre. Technical report, Institut Polytechnique de Grenoble, 1989.

[97] Pascal Raymond. *Compilation efficace d'un langage déclaratif synchrone : le générateur de code Lustre-v3.* PhD thesis, Institut National Polytechnique de Grenoble, 1991.

[98] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design.* MIT Press, 1993.

[99] John C. Reynolds. *Theories of Programming Languages.* Cambridge University Press, 1998.

[100] Robert Sedgewick. *Algorithms in C.* Addison Wesley, 2002.

[101] B. A. Sijtsma. On the productivity of recursive list definitions. *ACM Transactions on Programming Languages and Systems*, 11, October 1989.

[102] M. H. Sørensen, R. Glück, and N. D. Jones. Towards unifying partial evaluation, deforestation, supercompilation and GPC. In *European Symposium on Programming, LNCS 788*, pages 485–500. Springer Verlag, 1993.

[103] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111(2) :245–296, 1994.

[104] Jean-Pierre Talpin and David Nowak. A synchronous semantics of higher-order processes for modeling reconfigurable reactive systems. In *International Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS)*. Springer Verlag, December 1998.

[105] Simon Thompson. A functional reactive animation of a lift using fran. *Journal of Functional Programming*, 10(3) :245–26, May 2000.

[106] Jean Vuillemin. On Circuits and Numbers. Technical report, Digital, Paris Research Laboratory, 1993.

[107] W. W. Wadge. An extensional treatment of dataflow deadlock. *Theoretical Computer Science*, 13, 1981.

[108] P. Wadler. Listlessness is better than laziness : Lazy evaluation and garbage collection at compile time. In *ACM Symposium on LISP and Functional Programming*, pages 45–52, 1984.

[109] P. Wadler. Deforestation : transforming programs to eliminate trees. *Theoretical Computer Science*, 73 :231–248, 1990.

[110] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *International Conference on Programming Language, Design and Implementation (PLDI)*, 2000.

[111] Zhanyong Wan, Walid Taha, , and Paul Hudak. Real-time frp. In *International Conference on Functional Programming (ICFP)*, 2001.

[112] M. Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93, 1-15 1991.

Table des matières