PSL
UNIVERSITÉ PARIS

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à l'École normale supérieure

Une sémantique dénotationnelle pour un compilateur synchrone vérifié

Soutenue par
Paul Jeanmaire
Le 20 décembre 2024

École doctorale n° 386
Sciences Mathématiques de Paris Centre

Spécialité
Informatique

Composition du jury :

Xavier Rival	Inria	*Président*
Sylvain Boulmé	Grenoble INP	*Rapporteur*
Christine Paulin-Mohring	Université Paris-Saclay	*Rapporteuse*
Michael Mendler	Université de Bamberg	*Examinateur*
César A. Muñoz	NASA	*Examinateur*
Yannick Zakowski	Inria	*Examinateur*
Timothy Bourke	Inria	*Co-encadrant*
Marc Pouzet	École normale supérieure	*Co-encadrant*

1794
ENS | PSL

Une sémantique dénotationnelle pour un compilateur synchrone vérifié

Paul Jeanmaire

20 décembre 2024

Remerciements

Je remercie en premier lieu les rapporteurs de cette thèse, Sylvain Boulmé et Christine Paulin-Mohring. Leurs travaux sur la modélisation des langages à flots de données ont été, par leur élégance et leur précision, une source inestimable d'inspiration et d'admiration.

Je remercie chaleureusement mes encadrants, Tim et Marc. C'est grâce à leur bienveillance, leur disponibilité et leur grande patience à mon égard que j'ai pu mener à bien ce travail difficile mais passionnant. Je les remercie infiniment de m'avoir accueilli dans l'équipe Parkas, où j'ai eu la chance de côtoyer d'autres jeunes chercheurs, aussi redoutablement efficaces que Basile, aussi virtuoses que Baptiste ou aussi érudits que Grégoire. Je remercie Christine, puis Laurence, d'avoir rendu possible et agréable mon séjour à l'Inria, et en dehors. Je remercie tous les autres, membres un jour de l'équipe ou non, avec qui mes rapports furent divers et toujours enrichissants : Guillaume, Jean-Baptiste, Adrien, Loïc, Lélio, Albert, Hector, Ada, Astyax, Charles, Jean-Christophe, Jiawei, Rémy, Victor, et j'en oublie. Je remercie également la communauté Synchron, lieu d'échanges techniques et sincères, qui m'a accueilli avec bienveillance.

Je remercie les membres du département d'informatique que j'ai eu le très grand plaisir de côtoyer au fil des années : Pierre, Chien-Chung, Lise-Marie, Tatiana, Pierre, Yann, Michaël, Valérie, Mohamed, Ky, Jérôme, Jérôme, Xavier, Valentin, Bernadette, Josselin, Naïm et tous les autres.

Je remercie l'ENS d'offrir un environnement de travail aussi confortable et foisonnant. Merci au club Trouvères, au club de baby-foot, au service des concours et à Noureddine, merci à Éric et aux autres cuisiniers.

Je remercie mes amis et partenaires musicaux de tous bords, pour avoir rendu mon séjour parisien si épanouissant. Je remercie toute ma famille, pour ces moments heureux mais parfois difficiles. Enfin, je remercie Judith, sans qui la vie serait moins douce, et aussi moins salée.

Table des matières

Chapitre 1

Introduction

Gilles Kahn introduit, en 1974, un modèle à flots de données pour un langage de programmation parallèle. Les composants d'un réseau de Kahn [1], assimilables à ceux d'un système distribué, communiquent entre eux sur des canaux équipés de files d'attente dont la taille peut croître arbitrairement. Son idée, alors fondatrice, consiste à modéliser les canaux comme des flots de valeurs et les composants des réseaux comme des fonctions de transformation de flots. En résulte une élégante sémantique, fonctionnelle et modulaire, reposant sur des propriétés de continuité des fonctions de flots.

C'est dans la lignée de ces travaux qu'apparaissent les langages synchrones, comme Esterel [2] ou Lustre [3]. En s'inspirant des notions venant de l'automatique et de traitement du signal, les langages synchrones ajoutent un mécanisme d'*horloges* au modèle de Kahn, imposant une exécution synchronisée des différents composants. L'avantage d'une telle approche est double. D'une part, les langages synchrones permettent de décrire des systèmes sous forme d'équations récursives contraignant les flots de valeurs circulant sur les signaux, à la manière des réseaux de Kahn. Ce sont donc des langages de *spécification*, grâce auxquels l'utilisateur peut se concentrer sur la modélisation du comportement attendu du système, indépendamment de toute notion de calcul ou d'évaluation. D'autre part, les restrictions imposées au langage, comme les contraintes d'horloges, garantissent une exécution en mémoire bornée et permettent une génération de code séquentiel très efficace, en vue d'une exécution sur des processeurs embarqués. Cette tâche de transformation de la spécification en un programme exécutable, délicate et sujette aux erreurs, est décrite dans plusieurs travaux [4] [5] et déléguée à des outils spécialisés, tels que la suite industrielle Scade [6] ou les compilateurs

[1] Kahn, 1974
[2] Berry et Cosserat, 1984
[3] Halbwachs *et al.*, 1991
[4] Raymond, 1991
[5] Biernacki *et al.*, 2008
[6] Colaço *et al.*, 2017

académiques Heptagon [7] et Vélus [8]. C'est dans ce dernier, développé à l'Inria, que cette thèse trouve son cadre.

Vélus est un compilateur synchrone formellement vérifié dans l'assistant de preuve Coq [9], c'est-à-dire accompagné d'une preuve de correction de ses algorithmes de compilation. La contribution majeure de ce projet est donc l'encodage d'une sémantique de flots pour le langage d'entrée, ainsi que tous les mécanismes permettant de raisonner sur cette sémantique au sein de l'assistant de preuve. Si les algorithmes de compilation des langages synchrones sont déjà bien établis et documentés, leur implémentation dans Coq se révèle extrêmement délicate. Il en va de même pour le langage d'entrée, dont plusieurs modèles sont décrits dans la littérature, mais peu ont été encodés formellement dans un assistant de preuve. C'est cette question, ainsi que celle du raisonnement interactif sur les programmes, qui motive principalement le travail de cette thèse.

1.1 Sémantique formelle et compilation vérifiée

Le langage pris en charge par Vélus, nommé ici simplement *Lustre*, correspond à un sous-ensemble de Lucid Synchrone [10] sans fonctions d'ordre supérieur, ou encore à un mélange de Lustre v4 et Scade 6, avec notamment un opérateur de réinitialisation modulaire.

En Lustre, les composants d'un système sont représentés par des *nœuds*, opérant sur des flots de valeurs.

```
node countdown (n : int; r : bool)
returns (t : int)
let
  t = case r (true => n)
             (false => n fby (t - 1));
tel
```

La sémantique d'un nœud consiste, pour chaque combinaison de flots de valeurs associés aux entrées, ici n et r, en des flots associés aux sorties, ici t seulement, qui satisfont les équations du programme. On parle alors d'une sémantique *relationnelle*. Il est d'usage de représenter cette sémantique en donnant une grille, ou chronogramme, correspondant à une exécution possible du programme.

n	4	4	4	4	4	4	5	5	5	...
r	F	F	F	F	T	F	F	T	F	...
t	4	3	2	1	4	3	2	5	4	...

[7] Heptagon Developers, 2017
[8] Bourke *et al.*, 2017
[9] Coq Development Team, 2020
[10] Caspi et Pouzet, 1995

Dans cette instance du compteur descendant, la sémantique de t est un flot qui commence à la même valeur que n puis décroît d'une unité jusqu'à ce qu'une valeur vraie soit rencontrée dans r, auquel cas le processus se répète.

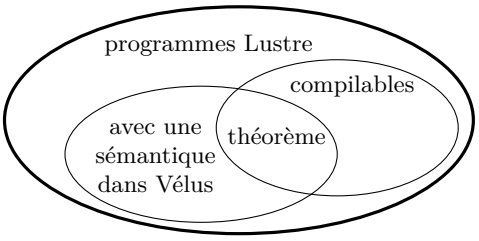
Se pose alors la question d'existence et d'unicité d'une telle sémantique : si plusieurs flots satisfont les équations, doit-on considérer que le système est non-déterministe ? s'il n'existe pas de flots qui satisfont les équations, doit-on interdire la génération de code ? peut-on caractériser quels flots d'entrée justifient l'existence d'une sémantique ? La réponse à ces interrogations est au cœur des choix de conception d'un compilateur vérifié.

Dans Vélus, dont la génération de code assembleur est assurée par le compilateur vérifié CompCert [11], l'existence d'une sémantique ou, plus formellement, l'existence d'un élément dans la relation définie par la sémantique, est une hypothèse nécessaire à la preuve de correspondance entre le programme d'entrée et le code généré.

Théorème (correction de la compilation, simplifié)**.**

si $\mathsf{compile}\ P_L = \mathsf{OK}\,(P_{asm})$
et $P_L \vdash \mathit{ins} \Downarrow \mathit{outs}$
alors $P_{asm}(\mathit{ins}) \simeq \mathit{outs}$.

Le résultat de ce théorème est considérable. Pour tout programme Lustre P_L compilé avec succès en un programme assembleur P_{asm}, s'il existe une sémantique d'entrée *ins* et de sorties *outs* pour P_L, alors l'exécution de P_{asm} sur *ins* résultera en *outs*. Autrement dit, Vélus transforme un système d'équations qui comporte des solutions en un programme exécutable calculant véritablement ces solutions.



Deux classes de programmes se dessinent alors, ceux qui sont acceptés lors de la compilation et ceux qui possèdent une sémantique. Le théorème de correction de Vélus ne s'applique qu'aux programmes se situant à l'intersection des deux classes. Pour les autres, les hypothèses du théorème ne sont pas satisfaites et aucune garantie n'est apportée sur le résultat de la compilation. Si les conséquences sont limitées dans le cas de programmes non compilables, elles sont critiques pour des programmes compilés ne possédant pas de sémantique, l'utilisateur pouvant alors exécuter, sans en être averti, du code ne correspondant pas au programme source.

[11] Leroy, 2009a

Déterminer, pour un programme donné, l'existence ou non d'une sémantique est en général un problème indécidable, d'autant plus qu'il dépend des valeurs des entrées, inconnues lors de la compilation. Il est possible d'aborder le problème en définissant un interpréteur de référence. Cette approche, développée notamment pour le langage Zélus avec Zrun [12], et de façon vérifiée pour le langage C avec CompCert [13], permet d'exécuter directement le programme source sur des entrées déterminées et d'utiliser le résultat de ce calcul pour produire un témoin de la relation de sémantique. Il faut alors tenir compte de toutes les exécutions du programme, y compris celles menant à des cas d'erreurs, blocage, ou tout autre « comportement indéfini ».

Dans cette thèse, nous proposons d'appliquer une approche similaire. En développant pour Vélus un modèle dénotationnel constructif standard, nous spécifions formellement tous les comportements possibles d'un programme source. Nous prouvons ensuite que les exécutions sans erreurs calculent des flots qui satisfont bien les équations du programme en question.

Théorème (nouvelle correction de la compilation, simplifié)**.**

si $\mathsf{compile}\ P_L = \mathsf{OK}\,(P_{asm})$

et $\mathsf{no\text{-}run\text{-}time\text{-}errors}\ P_L\ ins$

alors $\exists outs, P_L \vdash ins \Downarrow outs \wedge P_{asm}(ins) \simeq outs.$

Avec ce nouveau résultat de correction du compilateur, l'existence d'une sémantique du programme P_L n'est plus un prérequis mais une conséquence.

L'exclusion des erreurs à l'exécution reste un point important à résoudre, et c'est tout l'objet du prédicat no-run-time-errors. Beaucoup d'erreurs, pouvant survenir à l'exécution d'un programme écrit dans un langage comme C ou OCaml, sont évitées en Lustre grâce aux restrictions appliquées au langage source, comme une absence de récursivité, un typage fort, une gestion implicite de la mémoire, etc. Nous démontrons, dans l'assistant de preuve, qu'après les analyses statiques effectuées par le compilateur, seules les erreurs de calcul arithmétique ou logique restent possibles à l'exécution. Ces dernières ne peuvent pas toujours être détectées par des analyses statiques car elles dépendent bien souvent des valeurs données en entrée du programme.

Justifier l'absence de telles erreurs fait l'objet de nombreux travaux, notamment dans le domaine de l'interprétation abstraite [14]. Dans le cadre de cette thèse nous proposons une analyse, beaucoup plus simple, consistant à rejeter tout programme utilisant la division entière ou le décalage logique avec une deuxième opérande non définie statiquement, ainsi que la conversion entre un type flottant et un type entier. Nous montrons que cette analyse définit une classe, intéressante en pratique, de programmes pour lesquels le

[12] Pouzet, 2021
[13] Leroy, 2009b
[14] Kästner *et al.*, 2010

prédicat no-run-time-errors est valide, et donc pour lesquels l'existence d'une sémantique est garantie, quels que soient les flots d'entrée.

1.2 Vérification interactive de programmes

L'expression et la vérification automatique de propriétés est un domaine très étudié depuis la création des langages synchrones. Le schéma de compilation, par traduction vers un système de transitions, rend les systèmes synchrones particulièrement bien adaptés à une vérification par model-checking. Les travaux sur les observateurs synchrones [15] proposent de tirer parti de la compositionnalité du langage pour exprimer des propriétés comme des nœuds du système, en utilisant donc la même syntaxe. La composition du système et des nœuds de propriétés, correspondant à un produit synchrone d'automates, peut ensuite être analysée par un outil de model-checking. Contrairement aux systèmes parallèles asynchrones, qui nécessitent de considérer tous les entrelacements possibles de l'exécution des composants, les systèmes synchrones permettent de limiter l'explosion combinatoire lors de l'analyse. Ces idées ont donné lieu à l'implémentation d'outils de vérification automatique très efficaces. C'est le cas de Lesar [16], outil de model-checking pour Lustre utilisant les diagrammes de décision binaire, Kind 2 [17], basé sur un encodage SAT des propriétés et un raisonnement par k-induction [18], ou encore des outils de la suite industrielle Scade.

La vérification interactive dans un assistant de preuve constitue une approche tout à fait complémentaire. Tandis que les propriétés définies au moyen d'observateurs sont, par définition, limitées par l'expressivité du langage source, elles sont beaucoup plus générales dans une logique plus expressive. En Coq, il est par exemple possible d'utiliser des quantificateurs, universels ou existentiels, ou bien de formuler des assertions arbitrairement complexes sur des théories arithmétiques ou temporelles. On pourrait alors aussi envisager la vérification de *modèles paramétrés*, comme une bibliothèque de calcul sur des entrées de taille arbitraire, un système distribué dont le nombre d'agents est un paramètre, etc. Le traitement des horloges et des machines à états, insatisfaisant dans les outils automatiques, pourrait également être abordé interactivement. La contrepartie reste, bien sûr, l'absence quasi-totale d'automatisation du raisonnement. Des travaux [19] sur la preuve déductive de programmes Lustre ont été réalisés dans ce sens au moyen d'une axiomatisation de la sémantique du langage et d'un générateur d'obligations dans l'assistant de preuve PVS [20].

[15] Halbwachs *et al.*, 1993
[16] Halbwachs *et al.*, 1992
[17] Champion *et al.*, 2016
[18] Sheeran *et al.*, 2000
[19] Dumas, 2000; Mikáč, 2005
[20] Dumas et Caspi, 2000

Nous proposons d'aborder la question au sein de Vélus. Le théorème de correction du compilateur assure que les valeurs des flots définis par les équations d'un programme Lustre sont effectivement les valeurs calculées successivement par le code s'exécutant sur la machine. Ce résultat rend donc possible un raisonnement formel de bout en bout, c'est-à-dire de pouvoir raisonner sur le modèle mathématique du programme source tout en ayant la garantie que les propriétés formulées s'appliquent également au programme cible. Si cette idée de combiner preuve interactive et compilation vérifiée a déjà été expérimentée à plusieurs reprises, pour des langages impératifs, comme dans le projet VST [21], et pour des langages fonctionnels, comme dans le projet CakeML [22], elle n'a pas encore été appliquée aux langages synchrones. Dans cette thèse, par la formalisation en Coq d'un nouveau modèle sémantique et par l'étude de plusieurs principes de raisonnement sur les flots de données, nous posons les bases nécessaires, mais pas encore suffisantes, à un raisonnement interactif sur les programmes Lustre.

Sémantique synchrone, sémantique de Kahn

Les travaux [23] sur la preuve déductive de programmes Lustre insistent sur l'importance du modèle sémantique à adopter afin de faciliter le raisonnement. Ils questionnent notamment la représentation des différentes échelles de temps au sein d'un programme. En Lustre, les valeurs des flots sont *échantillonnées*, c'est-à-dire disponibles seulement à certains instants, déterminés par une *horloge* associée à chaque expression. La manipulation des horloges est réalisée par deux constructions du langage.

```
node actdef (ck : bool; x : int when ck)
returns (y : int)
let
 y = merge ck (true => x)
             (false => 0 when not ck);
tel
```

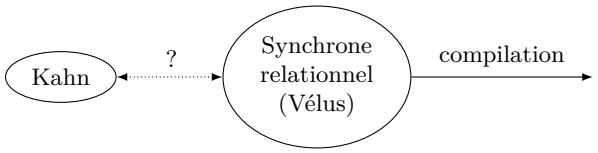
Le `when` introduit un sous-échantillonnage. Dans le nœud actdef ci-contre, les valeurs de x ne sont disponibles que lorsque celles de ck sont vraies, on dit alors que ck est l'horloge de x. Les valeurs du `merge`, au contraire, sont toujours disponibles. Lorsque ck est vrai, la valeur est prise dans x, et lorsque ck est faux, la valeur est prise dans la deuxième branche. Les constantes étant aussi interprétées par des flots échantillonnés, il est nécessaire de spécifier l'horloge de l'expression 0. Ici, il s'agit de l'horloge complémentaire de ck. Lorsque la valeur d'un flot n'est pas disponible, on dit qu'elle est absente. Cette information est rendue explicite par le modèle synchrone, grâce à l'ajout d'éléments abs dans le chronogramme.

[21] Appel *et al.*, 2014
[22] Guéneau *et al.*, 2017
[23] Dumas, 2000

ck	T	T	F	F	T	F	T	T	...
x	4	4	abs	abs	4	abs	5	5	...
`0 when not ck`	abs	abs	0	0	abs	0	abs	abs	...
y	4	4	0	0	4	0	5	5	...

Dans Vélus, la cohérence des horloges est assurée par un système de types dédié, inspiré de celui de Lucid Synchrone [24]. Le bon « alignement » des flots avec leurs horloges est ainsi démontré dès la compilation. L'information de présence ou d'absence des valeurs, centrale dans le modèle synchrone de Vélus, est utilisée très largement lors de la génération de code séquentiel. En revanche, le raisonnement sur les flots échantillonnés est compliqué par la multiplication de cas, due aux possibles absences dans les flots, et ce schéma semble peu adapté à la preuve interactive de programmes [25]. Une idée serait alors d'« effacer » les éléments abs du modèle synchrone afin de se retrouver dans le contexte, plus favorable à la vérification, de flots non échantillonnés mais éventuellement finis. Cette opération revient en fait à passer d'une sémantique synchrone à une sémantique de Kahn [26].

Nous proposons dans cette thèse d'explorer la relation entre le modèle synchrone et le modèle de Kahn à travers la formalisation en Coq des différentes construction de Vélus, et notamment celle de réinitialisation modulaire [27]. Ce travail est en grande partie rendu possible grâce à une bibliothèque [28] dédiée à la modélisation des flots dans les réseaux de Kahn. Bien que la procédure d'effacement des absences dans les flots soit facile à définir, nous verrons que la gestion des erreurs à l'exécution reste un problème dans l'énoncé de correspondance entre les deux modèles.



[24] Colaço et Pouzet, 2003
[25] Dumas, 2000
[26] Caspi et Pouzet, 1996
[27] Bourke *et al.*, 2020
[28] Paulin-Mohring, 2009

Plan des chapitres

Cette thèse est organisée en cinq chapitres. La présente introduction en constitue le premier. Le second chapitre est d'abord consacré à une description générale du compilateur Vélus, de ses mécanismes, de sa correction et des différents aspects de son langage d'entrée. Nous présentons ensuite différentes approches pour un raisonnement interactif sur les programmes, basées à la fois sur l'état de l'art et sur nos propres expériences au sein du compilateur. Dans le troisième chapitre, nous décrivons l'encodage en Coq d'une sémantique synchrone dénotationnelle pour le langage d'entrée, incluant notamment une nouvelle définition de l'opérateur de réinitialisation modulaire. Nous donnons, pour chaque nouvel opérateur introduit, les conditions de son équivalence avec sa version relationnelle existant dans la sémantique du compilateur. Dans le quatrième chapitre, nous démontrons des propriétés fondamentales du modèle dénotationnel comme la productivité des fonctions de flots, la correction du typage ou encore la préservation de certaines propriétés algébriques. Enfin, en conclusion, nous rassemblons tous les éléments de cette thèse pour établir un nouveau théorème de correction de la compilation, dans lequel l'existence d'une sémantique au programme d'entrée n'est plus supposée mais calculée. Nous décrivons également une procédure d'analyse statique rejetant les programmes dont l'exécution peut conduire à une erreur arithmétique ou logique, seul cas invalidant les hypothèses du nouveau théorème de correction.

Note sur le développement

Le développement Coq associé à cette thèse, disponible en ligne à l'adresse suivante, est distribué sous la licence non commerciale Inria.

`https://github.com/INRIA/velus/tree/paul-thesis`

La branche `paul-thesis` dérive de Vélus dans sa version `emsoft23`, suite aux travaux de Basile Pesin sur la compilation vérifiée des automates hiérarchiques. Bien que le traitement des automates sorte du cadre de cette thèse, nous avons choisi ce point de départ pour bénéficier de l'infrastructure développée par Basile, permettant notamment de raisonner sur la causalité des programmes. En définissant une nouvelle sémantique pour le langage d'entrée, nous n'avons pas modifié le compilateur à proprement parler. La quasi-totalité des contributions de cette thèse se trouve donc dans le répertoire `src/Lustre/Denot`, totalisant environ 23 000 lignes de code. Pour faciliter la lecture de ce document, les définitions, théorèmes, fonctions et autre prédicats inductifs sont donnés tantôt dans leur notation Coq, tantôt dans des notations mathématiques standards, sous forme de règles d'inférence ou de pseudo-code. Dans tous les cas, nous avons inséré des hyperliens permettant de se référer rapidement aux véritables définitions du développement.

Chapitre 2

Vérification des compilateurs et des programmes

Dans ce chapitre, nous décrivons les différents éléments constituant le cadre formel de notre problématique. Nous présentons Vélus, sa genèse, ses langages intermédiaires, ses principes de compilation et surtout la sémantique relationnelle de son langage d'entrée. Nous proposons d'abord d'utiliser directement cette relation pour raisonner sur le comportement des programmes, mais nous nous heurtons aux nombreuses complications liées à son intégration dans un compilateur. Nous explorons alors d'autres modèles à flots de données susceptibles de faciliter les raisonnements et discutons de leur éventuel encodage dans Coq.

2.1 Le compilateur Vélus

2.1.1 Contexte industriel et lien avec Scade

Certains systèmes embarqués critiques, notamment dans le domaine de l'avionique, de l'automobile ou du ferroviaire, sont élaborés à l'aide d'une technologie de type schéma-bloc. Dans les outils industriels Scade Suite [1], par exemple, les utilisateurs conçoivent les systèmes en assemblant virtuellement des composants, à la manière de circuits numériques. Ces composants peuvent être de natures différentes, allant d'opérations logiques simples sur des signaux à des itérateurs ou des automates hiérarchiques (exemple en figure 2.1). Cette approche est dite *dirigée par le modèle* car les planches ainsi conçues correspondent à une description fonctionnelle du système. L'implémentation concrète de programmes exécutables à partir de ces modèles est laissée à des outils spécialisés. Dans le cas de Scade, les composants des planches du modèle graphique correspondent exactement aux constructions d'un langage synchrone à flots de données, inspiré de Lustre

[1] ANSYS/Esterel Technologies, 2024

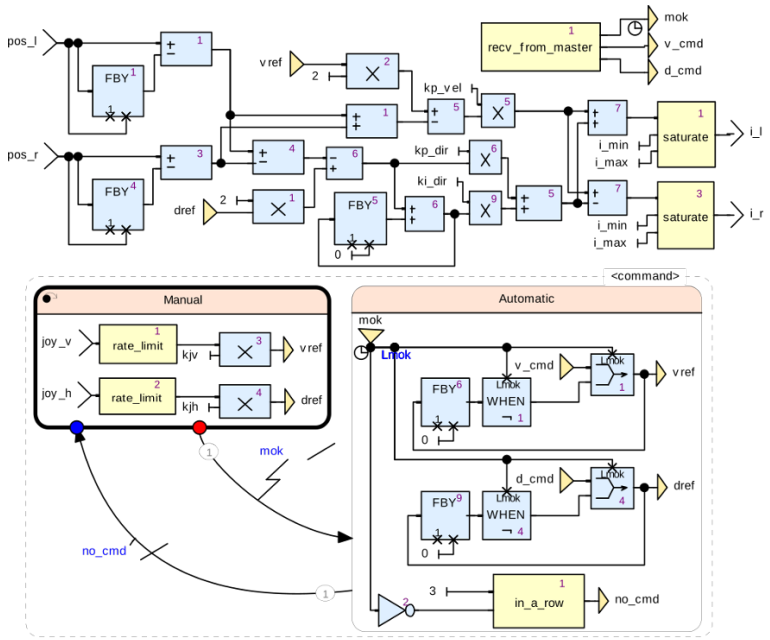


FIGURE 2.1 – Une planche Scade adapté d'un modèle de fauteuil roulant électrique, par Timothy Bourke. Les lignes représentent des signaux, les petites boîtes rectangulaires des opérateurs (+, -, fby, mux, etc.) et les plus grandes boîtes des structures de contrôle avec transitions.

et nommé Scade 6 [2]. La traduction de programmes Scade en programmes bas-niveau C ou Ada est effectuée par le compilateur Scade KCG, dont la fiabilité est assurée par un processus de certification extrêmement rigoureux, basé sur différentes méthodes de test et de couverture de code.

L'ambition du compilateur Vélus, développé à l'Inria, est toute autre. En s'appuyant sur l'assistant de preuve Coq, Vélus vise à proposer une description formelle, la plus précise possible, du processus de compilation d'un langage synchrone. Cela passe d'abord par une formalisation complète de la sémantique de chaque langage manipulé, puis par l'explicitation de tous les invariants nécessaires à la preuve de correction des différents algorithmes impliqués dans la compilation. Bien que le résultat de ce projet – une preuve de correction de bout en bout de la compilation – soit remarquable, l'application d'un tel compilateur dans un contexte industriel n'est pas encore envisageable. En effet, l'effort du projet est principalement orienté vers les preuves et non vers l'infrastructure nécessaire à une maintenance sur le long terme et une certification industrielle. D'autre part, le langage pris en charge par Vélus inclut les constructions fondamentales d'un langage synchrone, mais pas toutes celles nécessaires à la programmation des systèmes en général.

[2] ANSYS/Esterel Technologies, 2016

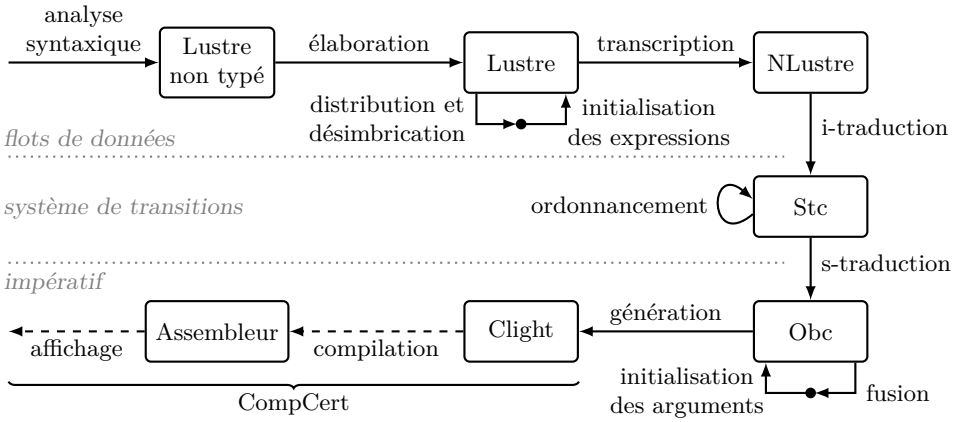


FIGURE 2.2 – Architecture du compilateur

C'est le cas des tableaux de taille statique, utilisés intensivement dans les applications de traitement du signal ou les réseaux de neurones, et dont l'ajout dans Vélus constitue pour l'instant un véritable défi technique.

2.1.2 Architecture

La chaîne de compilation implémentée par Vélus, représentée en figure 2.2, suit un schéma classique introduit dans les années 80 puis formalisé [3] pour la génération modulaire de code impératif. Parmi les autres compilateurs suivant ce schéma, on peut citer Heptagon [4], Scade KCG [5], Lustre v6 [6], LustreC [7]. Le processus de compilation est articulé autour des transformations appliquées à plusieurs langages intermédiaires. Avec l'*élaboration*, seule étape susceptible d'échouer, Vélus commence par analyser le programme d'entrée et lui attribuer des annotations de types et d'horloges. Un programme Lustre élaboré avec succès obtient la garantie, via des preuves de prédicats en Coq, de la bonne formation de ses éléments syntaxiques et de l'absence de chaîne de dépendance cyclique entre les variables (plus de détails à ce propos seront donnés en section 4.1). Plusieurs transformations sont appliquées au programme en vue de sa *transcription*, dans le langage normalisé NLustre. La forme restreinte de NLustre est ensuite exploitée pour faciliter la preuve d'optimisations flots de données sur le programme. Viennent enfin les transformations vers le langage Stc, dédié à l'ordonnancement des équations du programme en blocs, puis Obc, langage impératif décrivant explicitement la gestion de la mémoire du programme, et Clight, sous-ensemble du langage C.

[3] Biernacki *et al.*, 2008
[4] Heptagon Developers, 2017
[5] ANSYS/Esterel Technologies, 2016
[6] Jahier *et al.*, 2019
[7] Brun *et al.*, 2023

La suite de la compilation est assurée par CompCert [8], qui se charge de la génération de code assembleur pour différentes architectures matérielles.

Si chaque langage intermédiaire possède ses caractéristiques propres, à la fois syntaxiques et sémantiques, certaines constructions sont partagées. Les expressions correspondant aux opérations de base sur les flots, comme l'échantillonnage ou le délai, sont les mêmes du langage d'entrée à Stc. Leur interprétation diffère néanmoins selon le modèle de flots, *co-inductif* en Lustre et *instantané* en Stc. Les opérations sur les données numériques primitives, comme les additions, divisions et autres opérations flottantes ou bit-à-bit sont quant à elles définies par CompCert. Leur spécification complète, qui va de l'analyse lexicale des constantes numériques à l'implémentation d'un modèle de calcul pour différents types d'unités arithmétiques et logiques, tout en respectant les conventions de la norme C11, est particulièrement complexe. C'est pourquoi les langage intermédiaires de Vélus incorporent, par l'intermédiaire de types paramétrés, les opérateurs arithmétiques et logiques directement depuis CompCert.

2.1.3 Principes de compilation

Bien que le schéma de compilation en lui-même ne soit pas l'objet de cette thèse, il est intéressant de constater son effet sur un programme synchrone. Considérons le nœud Lustre *countup*, spécifiant un simple compteur pouvant être remis à zéro par l'activation d'une condition r.

```
node countup (r : bool)
returns (t : int)
let
  t = if r then 0 else (0 fby (t + 1));
tel
```

L'équation de t définit un flot qui vaut zéro aux instants où celui de r est vrai et sinon vaut sa propre valeur calculée à l'instant précédent, plus un. Dans le cas où r est faux à l'instant initial, aucune valeur précédente de t n'est disponible, et c'est le membre gauche du délai initialisé fby qui précise la valeur à utiliser, en l'occurrence zéro. L'exécution séquentielle d'une instance de *countup*, calculant successivement toutes les valeurs du flot t, ne nécessite de stocker qu'une seule valeur, celle de t, qui sera éventuellement lue puis mise à jour à chaque instant logique de l'exécution. Le code Clight généré par la compilation déclare cette mémoire, propre à chaque instance du nœud, sous la forme d'une structure, aussi appelée *état*.

```
struct countup { int t_st; };
```

L'exécution à proprement parler s'articule autour de deux fonctions manipulant explicitement cet état. La première sert à initialiser l'état avec les valeurs déclarées comme initiales dans le programme source.

[8] Leroy, 2009b

```
void countup_reset(struct countup *self) { (*self).t_st = 0; }
```

La seconde calcule la valeur des variables de sortie en fonction de l'état et des valeurs des entrées, puis met à jour l'état.

```
int countup_step(struct countup *self, unsigned char r) {
  register int t;
  switch(r) {
    case 0:
      t = (*self).t_st; break;
    default:
      t = 0;
  };
  (*self).t_st = t + 1;
  return t;
}
```

L'exécution séquentielle typique du programme réactif consiste alors en une alternance infinie entre lecture des entrées et calcul des sorties.

```
void main() {
  int r, t;
  struct countup st;
  countup_reset(&st);
  while(true) {
    r = read(...); // lecture d'une variable volatile
    t = countup_step(&st,r);
  };
}
```

L'*hypothèse synchrone*, s'appliquant à Lustre et autres langages de la même famille, consiste à supposer que la lecture des entrées et le calcul des sorties s'effectue dans un même instant logique. Pour satisfaire un tel critère, le compilateur doit générer une fonction *step* dont le temps d'exécution est inférieur à l'intervalle entre l'arrivée de deux valeurs sur une entrée.

La gestion de la mémoire est aussi un aspect important à prendre en compte dans la programmation de systèmes embarqués. Le compilateur doit être capable de borner statiquement la quantité de mémoire nécessaire à l'exécution de *step*, pour garantir que l'exécution à l'infini de la boucle de réaction n'entraîne aucune consommation de mémoire incontrôlée.

C'est par l'intermédiaire d'un mécanisme statique d'horloges, contrôlant l'accès aux données selon leur disponibilité dans le temps, qu'un compilateur Lustre est en mesure de rejeter les programmes ne s'exécutant pas en mémoire bornée. Les règles strictes du système d'horloges servent aussi à garantir certains aspects de la transformation en code impératif correct. Par exemple, le nœud *csum* ci-dessous calcule dans t la somme cumulative des valeurs de x, mais seulement aux instants où ck est vrai.

```
node csum (ck : bool; x : int)
returns (t : int when ck)
let
  t = 0 fby t + (x when ck);
tel
```

La disponibilité de t aux instants où ck est vrai est indiquée par une annotation dans sa déclaration de type. Le compilateur s'appuie sur cette information pour placer le calcul de la mise à jour de t dans une instruction conditionnelle.

```
int csum_step(struct csum *self, unsigned char ck, int x) {
  register int t;
  switch(ck) {
    case 0:
      t = 0; break; // valeur arbitraire
    default:
      t = (*self).t_st + x;
      (*self).t_st = t;
  };
  return t;
}
```

L'affectation arbitraire de zéro à t lorsque ck est faux est nécessaire pour satisfaire les contraintes d'initialisation d'arguments [9] prévues dans la norme C, implémentée par CompCert.

Ce dernier exemple illustre l'importance pour le compilateur de connaître la disponibilité des valeurs d'un flot. D'un point de vue sémantique, que nous développerons en section 2.1.5, le modèle intègre cette information par l'utilisation de valeurs échantillonnées. À tout instant, une valeur peut être soit *présente* et donc manipulable directement, soit *absente*, auquel cas aucune opération, affectation ou lecture, ne doit être effectuée dessus.

2.1.4 Le langage d'entrée, syntaxe et typage

Le langage Lustre pris en charge par Vélus correspond en réalité à un sous-ensemble du langage Scade 6, lui-même inspiré de Lustre v4 et de Lucid Synchrone. Dans ses récents travaux [10], Basile Pesin propose une extension de Vélus aux machines à états hiérarchiques reposant notamment sur un mécanisme sophistiqué de blocs locaux. Dans cette thèse cependant, nous nous appuyons sur le langage tel qu'il était défini en 2021, avant l'introduction des automates. Les présents travaux constituent donc un premier pas vers la formalisation d'une sémantique dénotationnelle pour un langage plus riche, mais le traitement des automates pose encore de nombreux défis.

La syntaxe complète prise en charge par le compilateur, présentée en figure 2.3 servira de support au reste de cette thèse. Sa représentation en

[9] Bourke et Pouzet, 2019
[10] Pesin, 2023

Coq sous forme d'arbre de syntaxe abstraite, donnée en figure 2.4, contient des annotations, informations supplémentaires obtenues par les différents algorithmes de typage que nous décrirons plus loin. Un programme s'organise autour d'expressions associées à des opérations de flots et de déclarations structurantes comme celles des variables, équations, types ou nœuds.

Opérateurs Les opérations unaires et binaires sur les types primitifs sont issues des définitions de CompCert. Seules les opérations sur les entiers et les flottants, qui sont appliquées point à point aux éléments des flots, sont conservées. Il n'y a en particulier pas de gestion explicite de la mémoire en Lustre, et donc aucune manipulation de pointeurs. Cela permet d'abstraire les opérateurs de CompCert comme de simples fonctions, qui ne dépendent pas d'un état de la mémoire.

Listes En Vélus, il est possible de définir plusieurs variables par équation. Par exemple, si f est un nœud à une entrée et deux sorties, alors l'équation `x,y = f(a)` est valide. En suivant le même principe, si g est un nœud à trois entrées et une sortie, alors `z = g(b,f(a))` est aussi une équation valide. Ce genre de combinaison, bien que peu intuitif, peut se retrouver dans des programmes générés par des outils de type schéma-bloc. Son encodage dans une syntaxe abstraite typée est assez délicat car il nécessite un principe d'associativité pour valider à la fois `g(b,f(a))` et `g(f(c),d)`, par exemple. Dans Vélus, ce problème est esquivé par l'utilisation de listes plutôt que de n-uplets pour encoder des sous-expressions. Ainsi, le typage des objets de type `exp` en Coq ne requiert pas de calcul d'associativité. La vérification de la cohérence des tailles des listes de sous-expressions s'incorpore quant à elle naturellement dans un prédicat de bonne formation, décrit ci-dessous.

Typage et environnements Le nombre de flots définis par une expression e ainsi que leurs types et leurs horloges sont des informations stockées dans les annotations de l'arbre de syntaxe et qui peuvent être respectivement obtenues à l'aide des fonctions Coq suivantes.

```
Definition numstreams (e: exp) : nat := ...
Definition typeof (e: exp) : list type := ...
Definition clockof (e: exp) : list clock := ...
```

On dit qu'un programme est *bien typé* s'il satisfait un prédicat inductif assurant la cohérence globale des annotations de typage. Une preuve de ce prédicat est obtenue en cas de succès de la première étape de compilation (cf. figure 2.2). Il s'agit d'un système très classique dont nous ne décrivons ici que les cas les plus intéressants.

La règle de bon typage d'une opération binaire $e_1 \oplus e_2$, donnée ci-dessous sous forme de règle d'inférence puis en Coq, spécifie par exemple que les deux

Expressions			
e	::=	c	constante
	|	C	constante d'une énumération
	|	x	variable
	|	$\diamond\ e$	opération unaire
	|	$e \oplus e$	opération binaire
	|	e^+ `fby` e^+	délai initialisé
	|	e^+ `when` $C(x)$	échantillonnage
	|	`merge` x $(C$ `=>` $e^+)^+$	fusion de flots échantillonnés
	|	`case` e $(C$ `=>` $e^+)^+$ $e^?$	conditionnelle avec cas par défaut
	|	$f(e^+)$	instance de nœud
	|	(`restart` f `every` e) (e^+)	instance de nœud avec réinitialisation
Équations			
eq	::=	x^+ `=` e^+;	
Types d'horloge			
ck	::=	$\bullet$	horloge de base
	|	ck `on` $C(x)$	sous-horloge
Types de données			
ty	::=	τ	type primitif
	|	tx	type énuméré
Déclarations			
dt	::=	`type` tx `=` $(\mid C)^+$	déclaration de type énuméré
d	::=	x^{ck}_{ty}	déclaration de variable
Composants			
n	::=	`node` $f(d^+)$ `returns` (d^+)	nœud
		$(\texttt{var}\ d^+)^?$ `let` eq^+ `tel`	
g	::=	$dt^*\ n^+$	programme

FIGURE 2.3 – Syntaxe abstraite de Lustre

```
Inductive type :=
| Tprimitive : ctype → type
| Tenum     : ident → list ident → type.

(* indice d'un type énuméré *)
Definition enumtag := nat.

Inductive clock : Type :=
| Cbase : clock
| Con   : clock → ident → type * enumtag → clock.

(* annotations *)
Definition ann : Type := type * clock.
Definition lann : Type := (list type) * clock.

Inductive exp : Type :=
| Econst : cconst → exp
| Eenum  : enumtag → type → exp
| Evar   : ident → ann → exp
| Eunop  : unop → exp → ann → exp
| Ebinop : binop → exp → exp → ann → exp
| Efby   : list exp → list exp → list ann → exp
| Ewhen  : list exp → (ident * type) → enumtag → lann → exp
| Emerge : ident * type → list (enumtag * list exp) → lann → exp
| Ecase  : exp → list (enumtag * list exp) → option (list exp) →
lann → exp
| Eapp   : ident → list exp → list exp → (list ann) → exp.

Definition equation : Type := list ident * list exp.

Record node : Type := mk_node {
  n_name    : ident;
  n_in      : list (ident * ann);
  n_out     : list (ident * ann);
  n_vars    : list (ident * ann);
  n_eqs     : list equation;

  n_ingt0   : 0 < length n_in;
  n_outgt0  : 0 < length n_out;
  (* ... *)
}.

Record global : Type := mk_global {
  types  : list type;
  nodes  : list node;
}.
```

FIGURE 2.4 – Syntaxe abstraite de Lustre, en Coq.

(Il s'agit ici de la syntaxe 2021, avant l'ajout des automates hiérarchiques à Vélus. Pour des raisons pratiques, nous basons notre développement sur la nouvelle syntaxe et définissions un prédicat restreignant l'arbre aux constructions effectivement prises en charge dans cette thèse. L'idée étant, à terme, de lever cette restriction.)

sous-expressions e_1 et e_2 doivent être récursivement bien typées. De plus, leurs types doivent être des singletons, compatibles avec l'opérateur binaire $\oplus$. Ce dernier point est spécifié par la fonction `type_binop` de CompCert, qui gère les éventuelles promotions lors d'opérations arithmétiques.

$$\frac{\Gamma \vdash e_1 : [ty_1] \qquad \Gamma \vdash e_2 : [ty_2] \qquad \vdash \oplus_{ty_1 \times ty_2} : ty}{\Gamma \vdash e_1 \oplus e_2 : [ty]}$$

```
Inductive wt_exp {Γ} : exp → Prop :=
| (* ... *)
| wt_Ebinop: ∀⊕ e1 e2 ty1 ty2 ty ck,
    wt_exp e1 →
    wt_exp e2 →
    typeof e1 = [ty1] →
    typeof e2 = [ty2] →
    type_binop ⊕ ty1 ty2 = Some ty →
    wt_exp (Ebinop ⊕ e1 e2 (ty, ck))
| (* ... *)
```

L'environnement Γ utilisé localement pour le typage d'un nœud Lustre coïncide avec les déclarations des variables présentes dans la syntaxe. Il associe à chaque identifiant de variable un type ty et une horloge ck. La règle de typage d'une variable en tant qu'expression force l'association de x à ty dans Γ et vérifie, via le prédicat inductif `wt_clock`, que les variables apparaissant dans l'horloge ck sont bien déclarées de type énuméré. L'association de x à ck dans Γ est effectuée dans un second temps, lors du typage des horloges.

```
| wt_Evar: ∀x ty ck,
    HasType Γ x ty →
    wt_clock Γ ck →
    wt_exp (Evar x (ty, ck))
```

Par la suite, nous distinguerons, pour chaque nœud du programme, l'environnement Γ_{E} des variables d'entrée et Γ_{S} celui des variables locales et de sortie. La cohérence des environnements en eux-mêmes est assurée par plusieurs prédicats, spécifiant notamment l'absence de redondance parmi les variables. Ainsi, dans un nœud bien typé, nous avons $\Gamma = \Gamma_{\mathrm{E}} \uplus \Gamma_{\mathrm{S}}$.

Système d'horloges La cohérence des annotations d'horloges est assurée de façon similaire. On dit qu'un programme est *bien cadencé* s'il satisfait un prédicat inductif encodant les contraintes du système d'horloges classique de Lustre [11]. Le cas des opérateurs d'échantillonnage when et merge est essentiel, tandis que les autres constructions sont transparentes vis-à-vis des horloges.

$$\frac{\Gamma(x) = ck \qquad \forall i,\ \Gamma \vdash es_i : ck}{\Gamma \vdash es \text{ when } C(x) : ck \text{ on } C(x)}$$

[11] Colaço et Pouzet, 2003

```
Inductive wc_exp {Γ} : exp → Prop :=
| (* ... *)
| wc_Ewhen: ∀ es x tx ty C tys ck,
    HasClock Γ x ck →
    Forall wc_exp es →
    Forall (eq ck) (clocksof es) →
    wc_exp (Ewhen es (x, tx) C (tys, (Con ck x (ty, C))))
| (* ... *)
```

Une expression $es_1, \ldots, es_n$ when $C(x)$ est bien cadencée et possède l'horloge ck on $C(x)$ à condition que x soit déclarée d'horloge ck et que toutes les sous-expressions $es_1, \ldots, es_n$ soient bien cadencées et d'horloge ck. Les différentes annotations de type tx, ty, et tys sont contraintes de façon orthogonale par le système décrit au paragraphe précédent.

À l'inverse, une expression merge x $(C_1 => es_1) \ldots (C_n => es_n)$ se voit attribuer l'horloge ck lorsque chaque branche es_i est d'horloge ck on $C_i(x)$.

$$\frac{\Gamma(x) = ck \qquad \forall i,\ \Gamma \vdash es_i : ck \text{ on } C_i(x)}{\Gamma \vdash \text{merge } x\ (C_i => es_i)_i : ck}$$

```
| wc_Emerge: ∀ x tx es tys ck,
    HasClock Γ x ck →
    Forall (λ '(Ci, es) ⇒ Forall wc_exp es) es →
    Forall (λ '(Ci, es) ⇒
       Forall (eq (Con ck x (tx, Ci))) (clocksof es)) es →
    wc_exp (Emerge (x, tx) es (tys, ck))
```

2.1.5 Sémantique synchrone relationnelle

Définir la sémantique d'un programme Lustre consiste à associer un flot à chaque variable, de sorte que toutes les équations soient satisfaites. Dans Vélus, qui adopte un modèle synchrone, les valeurs des flots sont de type svalue, échantillonnées grâce à un indicateur d'absence explicite.

$$\text{svalue} := v \mid \text{abs}$$

Les contraintes sur les flots du programme sont encodées avec des prédicats mutuellement inductifs définissant pour chaque nœud une relation entre les flots d'entrée et les flots de sortie. Pour un nœud f dans un programme G, la règle associant les flots d'entrée xs aux flots de sortie ys impose l'existence d'une définition (syntaxique) de nœud n pour f et d'une histoire H associant xs aux variables d'entrée et ys aux variables locales et de sortie.

$$\frac{\begin{array}{c} G.f = n \qquad H(n.\text{in}) = xs \qquad H(n.\text{out}) = ys \\ H : \text{ident} \to \text{Stream svalue} \qquad \forall eq \in n.\text{eqs},\ G, H, (\text{base-of } xs) \vdash eq \end{array}}{G \vdash f(xs) \Downarrow ys}$$

Chaque équation définie dans n introduit une contrainte supplémentaire sur l'histoire, paramétrée par une horloge de base, flot de booléens qui spécifie les instants auxquels les valeurs sont rendues disponibles, relativement au cycle global du programme. Au sein d'un nœud, l'horloge de base est instanciée par $\mathsf{base\text{-}of}\ xs$, vrai si l'une au moins des entrées xs est présente :

$$\mathsf{base\text{-}of}\ (xs_1, \ldots, xs_n) \equiv \left(\bigvee_{i=1}^{n} \mathsf{hd}\ xs_i \neq \mathsf{abs}\right) \cdot \mathsf{base\text{-}of}\ (\mathsf{tl}\ xs_1, \ldots, \mathsf{tl}\ xs_n).$$

Une équation $x = e$ est satisfaite lorsque la sémantique de l'expression e correspond au flot associé à x dans H.

$$\frac{G, H, bs \vdash e \Downarrow [s] \qquad s \equiv H(x)}{G, H, bs \vdash x = e}$$

Les différentes règles pour les expressions lient la syntaxe à des opérateurs de flots définis dans Coq. Par exemple, la sémantique d'une expression constante c l'associe à un unique flot s, lui-même défini par l'opérateur const, tenant compte de l'échantillonnage imposé par l'horloge de base. L'égalité extensionnelle de flots utilisée dans les prédicats, notée $\equiv$, est définie co-inductivement dans la bibliothèque standard de Coq.

$$\frac{s \equiv \mathsf{const}\ bs\ c}{G, H, bs \vdash c \Downarrow [s]} \qquad \begin{aligned} \mathsf{const}\ (\mathsf{T} \cdot bs)\ v &= v \cdot \mathsf{const}\ bs\ v \\ \mathsf{const}\ (\mathsf{F} \cdot bs)\ v &= \mathsf{abs} \cdot \mathsf{const}\ bs\ v \end{aligned}$$

La règle des opérateurs binaires se décompose de façon similaire avec d'une part un prédicat contraignant les sous-expressions et d'autre part un prédicat co-inductif $\textsc{lift}^{\oplus}$ décrivant l'application de l'opérateur $\oplus$ aux valeurs successives des flots. Les règles pour $\textsc{lift}^{\oplus}$ ne s'appliquent que si les valeurs des deux flots sont bien échantillonnées de la même façon. De plus, lorsque deux valeurs sont présentes, l'application de $\oplus$, qui est un opérateur partiel, ne doit pas échouer et produire une valeur v.

$$\frac{G, H, bs \vdash e_1 \Downarrow [vs_1] \qquad G, H, bs \vdash e_2 \Downarrow [vs_2] \qquad \textsc{lift}^{\oplus}\ vs_1\ vs_2\ vs}{G, H, bs \vdash e_1 \oplus e_2 \Downarrow [vs]}$$

$$\frac{\textsc{lift}^{\oplus}\ xs\ ys\ rs \qquad v_1 \oplus v_2 = \mathsf{Some}\ v}{\textsc{lift}^{\oplus}\ (v_1 \cdot xs)\ (v_2 \cdot ys)\ (v \cdot rs)} \qquad \frac{\textsc{lift}^{\oplus}\ xs\ ys\ rs}{\textsc{lift}^{\oplus}\ (\mathsf{abs} \cdot xs)\ (\mathsf{abs} \cdot ys)\ (\mathsf{abs} \cdot rs)}$$

La règle du délai initialisé $e_1\, \mathsf{fby}\, e_2$, plus complexe car nécessitant de stocker une valeur et de garder la synchronisation de ses entrées, se décompose en deux prédicats co-inductifs. Le premier, FBY, propage la valeur présente du flot de gauche et mémorise celle du flot de droite en la passant comme paramètre au prédicat $\textsc{fby}_1$. Ce dernier fonctionne de façon similaire en propageant la dernière valeur mémorisée du flot de droite. Ces deux prédicats sont *insensibles* à l'absence, dans la mesure où une valeur abs rencontrée est directement propagée et n'impacte pas l'éventuelle valeur mémorisée.

$$\frac{\text{FBY}_1\ v_2\ xs\ ys\ rs}{\text{FBY}\ (v_1 \cdot xs)\ (v_2 \cdot ys)\ (v_1 \cdot rs)} \qquad \frac{\text{FBY}\ xs\ ys\ rs}{\text{FBY}\ (\mathsf{abs} \cdot xs)\ (\mathsf{abs} \cdot ys)\ (\mathsf{abs} \cdot rs)}$$

$$\frac{\text{FBY}_1\ v_2\ xs\ ys\ rs}{\text{FBY}_1\ v\ (v_1 \cdot xs)\ (v_2 \cdot ys)\ (v \cdot rs)} \qquad \frac{\text{FBY}_1\ v\ xs\ ys\ rs}{\text{FBY}_1\ v\ (\mathsf{abs} \cdot xs)\ (\mathsf{abs} \cdot ys)\ (\mathsf{abs} \cdot rs)}$$

Bien que les valeurs du flot xs ne soient jamais propagées par le prédicat FBY_1, il est nécessaire de conserver leur synchronisation. Sans information sur `y`, par exemple, l'équation `x = y fby x + 1` ne peut être calculée.

Les règles relationnelles des autres opérateurs seront données et comparées plus en détail avec leurs analogues en sémantique de Kahn dans la section 3.3.

Exemple Le nœud Lustre suivant, adapté d'un exemple du manuel de Lucid Synchrone [12], décrit une opération courante en traitement du signal. La sortie o doit être gardée vraie pendant n instants dès qu'un front montant (passage de F à T) est détecté sur l'entrée i. Le décompte des instants depuis le dernier front montant est réalisé par une instance du nœud `countdown(n,r)`, défini en introduction, qui compte en décroissant à partir de n et reprend à n lorsque le signal r est vrai.

```
node rising_edge_retrigger (i : bool; n : int)
returns (o : bool)
var edge, ck : bool; v : int;
let
  edge = i and (false fby (not i));
  ck = edge or (false fby o);
  v = merge ck
        (countdown((n,edge) when ck))
        (0 when not ck);
  o = v > 0;
tel
```

Étant donnés des flots échantillonnés pour i, n et o, la sémantique du nœud est constituée d'une histoire H présentée ici sous forme de chronogramme et satisfaisant les prédicats décrits précédemment.

$H(i)$	F	T	abs	F	F	F	abs	F	T	F
$H(n)$	3	3	abs	3	3	3	abs	3	3	3
$H(o)$	F	T	abs	T	T	F	abs	F	T	T
base-of$(H(i), H(n))$	T	T	F	T	T	T	F	T	T	T
$H(edge)$	F	T	abs	F	F	F	abs	F	T	F
$H(ck)$	F	T	abs	T	T	T	abs	F	T	T
`countdown((n,edge) when ck)`	abs	3	abs	2	1	0	abs	abs	3	2
`0 when not ck`	0	abs	abs	abs	abs	abs	abs	0	abs	abs
$H(v)$	0	3	abs	2	1	0	abs	0	3	2

[12] Pouzet, 2006, §1.2.2

Ce style de sémantique est couramment utilisé pour des langages manipulant des processus qui interagissent entre eux, comme Esterel. L'existence de flots satisfaisant les prédicats permet de raisonner sur le comportement d'un programme sans se soucier de la façon d'obtenir ces flots, qui requiert souvent une lourde mécanique de calcul de point fixe d'équations mutuellement récursives. Par conséquent, dans le modèle relationnel de Lustre, la composition parallèle des nœuds coïncide précisément avec la conjonction logique des prédicats relationnels. Plus généralement, un modèle relationnel n'énumère que les cas pertinents d'un point de vue sémantique. Toute autre configuration de l'histoire ne respectant pas les règles d'introduction des prédicats est *de facto* exclue du modèle relationnel. Dans Vélus, l'existence d'une histoire satisfaisant les équations est un prérequis au théorème de correction principal et apparaît comme hypothèse dans la plupart des résultats intermédiaires. Une simple inversion (au sens de Coq) des prédicats inductifs décrits ci-avant permet d'éliminer les cas qui ne correspondent pas à une exécution valide du programme.

En contrepartie, de grandes précautions doivent être prises lors de la définition de ces prédicats afin qu'ils ne soient pas incohérents. Si le cas des constantes et des opérations binaires est simple et bien compris, celui des automates ou de la réinitialisation modulaire (cf. section 3.4) l'est beaucoup moins. De plus, ces descriptions des opérateurs de flots, à l'exception de `const`, ne sont pas calculatoires et ne permettent pas d'en déduire directement une fonction d'interprétation des expressions.

2.1.6 Causalité

Dans le modèle relationnel, les flots de l'histoire sont supposés infinis et respectant les équations, sans pour autant fournir de procédure permettant de les calculer. De fait, dans certains cas, une telle histoire existe mais ne peut être calculée sans résoudre des contraintes numériques, ce qui n'est pas prévu dans le langage.

```
node bad (...)
returns (x,y : int)
let
  x = y * 2;
  y = x - 3;
tel
```

x	6	6	6	6	6	6	...
y	3	3	3	3	3	3	...

Dans d'autres cas, plusieurs histoires valides existent mais le compilateur ne peut décider de laquelle implémenter.

```
node bad (...)
returns (x : int)
let
  x = x;
tel
```

x	3	4	5	6	7	8	...
x	42	42	42	42	42	52	...
x	...						

Ces situations sont gérées dans Vélus, comme dans d'autres langages synchrones, grâce à une analyse de dépendance entre les variables définies dans un même nœud [13]. Tout programme dont les dépendances instantanées forment un cycle, comme c'est le cas dans les deux exemples précédents, est systématiquement rejeté à l'élaboration. Cette analyse, basée sur un algorithme de tri topologique du graphe des dépendances, produit également une liste ordonnée servant de support à un schéma d'induction sur les variables. Plus de détails à ce propos seront donnés en section 4.1.

2.1.7 Correction de la compilation

Le principal résultat de correction démontré dans Vélus reflète les choix de conception présentés dans les sections précédentes.

Théorème (correction de la compilation)**.**

$$\begin{aligned}
\textbf{si}\quad & \mathsf{compile}\ G\ f = \mathsf{OK}\ asm \\
\textbf{et}\quad & \mathsf{welltyped\text{-}inputs}\ G\ f\ xs \\
\textbf{et}\quad & G \vdash f(xs) \Downarrow ys \\
\textbf{alors}\quad & asm \Downarrow \langle \mathsf{Load}(xs(i)) \cdot \mathsf{Store}(ys(i)) \rangle_{i=0}^{\infty}.
\end{aligned}$$

Ce théorème s'applique à tout nœud f défini dans un programme G. Le succès de la procédure compile appliquée à G implique la validation de toutes les analyses statiques. Un tel programme est donc bien typé, bien cadencé et satisfait les contraintes statiques de causalité. Le prédicat welltyped-inputs assure que les flots d'entrée xs sont bien cohérents vis-à-vis des déclarations du nœud f, autant pour le typage que pour l'échantillonnage des valeurs. Enfin, l'existence de flots de sortie ys satisfaisant la sémantique relationnelle est supposée. Sous ces hypothèses, le programme compilé asm, alternance de lecture des entrées et d'écriture des sorties, a la garantie de fournir les mêmes valeurs que le programme source.

Traces d'exécution Dans le modèle sémantique de CompCert, un programme assembleur est mis en relation avec un ensemble de *traces* d'évènements, qui peuvent dépendre de l'environnement dans lequel le programme est exécuté. En fixant à xs les valeurs successives des variables volatiles correspondant aux entrées du programme, la garantie est apportée que les valeurs écrites successivement dans les variables de sortie correspondent bien à celles de ys, et ce de façon infinie.

Si l'existence d'un élément dans la sémantique relationnelle est supposée, son unicité est un résultat prouvé de façon indépendante, sous l'hypothèse de causalité. Ainsi, sauf pour un programme ambigu, comme ceux décrits en section 2.1.6, la relation est déterministe.

[13] Bourke *et al.*, 2023a

Théorème (déterminisme de la sémantique[14])**.**

si is-causal G
et $G \vdash f(xs) \Downarrow ys_1$
et $G \vdash f(xs) \Downarrow ys_2$
alors $ys_1 \equiv ys_2$.

Ces principaux résultats mettent en lumière l'ambivalence de l'encodage d'une sémantique relationnelle de Lustre. L'utilisation de prédicats rend les définitions mécanisées proches de celles des articles fondateurs et permet de modéliser des équations mutuellement récursives comme de simples conjonctions logiques. Le choix de ne pas représenter les erreurs permet de s'abstraire de certaines considérations propres à la compilation. Dans le nœud suivant par exemple, le calcul du flot de t échoue si la valeur de l'entrée est nulle.

```
node fail_on_0 (x : int)
returns (y : int)
var t : int;
let
  t = 3 / x;
  y = 3 + x;
tel
```

Or la variable locale t, n'étant pas utilisée directement, sera très certainement éliminée du programme cible par une optimisation du compilateur. Il serait alors possible d'en tenir compte afin de donner une sémantique à ce programme sur l'entrée zéro, qui s'exécute en fait correctement. Mais ce serait au prix de l'introduction de détails du schéma de compilation, potentiellement compliqués ou arbitraires, dans le modèle d'entrée qui se veut aussi simple que possible. En Clight en revanche, une erreur ou un comportement indéfini à l'exécution est modélisé explicitement par un constructeur `Goes_wrong` associé à une trace finie d'évènements observables survenus avant l'erreur. L'un des corollaires au théorème de correction de CompCert exprime la préservation sémantique en termes de préfixes de traces, sans invalider toute l'exécution en cas d'erreur.

```
Theorem transf_c_program_preserves_initial_trace:
  ∀ (p : Clight.program) (tp : Asm.program) (t : trace),
  transf_c_program p = OK tp →
  c_program_has_initial_trace p t →
  asm_program_has_initial_trace tp t.
```

Le programme assembleur généré a ainsi la garantie de se comporter comme le programme source, et ce jusqu'à une éventuelle erreur. Cela est souhaitable pour des programmes séquentiels réalisant des entrées/sorties, notamment à des fins de debogage, et pour lesquels les effets de bord sont autant, sinon plus importants que l'issue du programme en lui-même.

[14] Bourke *et al.*, 2023a

Dans la seconde partie de ce chapitre, nous explorons les différentes possibilités qui nous sont offertes de raisonner sur le comportement des programmes, en manipulant directement leur représentation dans Coq.

2.2 Modèles pour le raisonnement interactif

La problématique des preuves interactives de programmes Lustre a été abordée très largement par Cécile Dumas dans sa thèse [15], qui propose une étude approfondie des principes de preuve adaptés à Lustre. Il apparaît dans son travail que la représentation des flots, a fortiori le modèle sémantique du langage, joue un rôle crucial dans l'application des différents principes de preuve. Elle montre en particulier qu'il est impossible d'obtenir un principe de preuve véritablement utilisable en pratique avec un modèle de flots synchrone, que celui-ci manipule des éléments absents explicitement, comme dans Vélus, ou non, auquel cas un système d'indices sophistiqué est nécessaire. En revanche, elle montre qu'un modèle du langage en termes de réseaux de Kahn, dans lequel les contraintes de synchronisation sont relâchées, fournit naturellement un principe d'induction bien plus approprié au raisonnement interactif. Elle développe dans ce sens une axiomatisation des principes de preuve de Kahn en PVS ainsi qu'un outil de génération d'obligations de preuve à partir d'un programme Lustre et de sa spécification. Cet outil sera plus tard démontré en partie incorrect, puis ré-implémenté par Jan Mikáč [16].

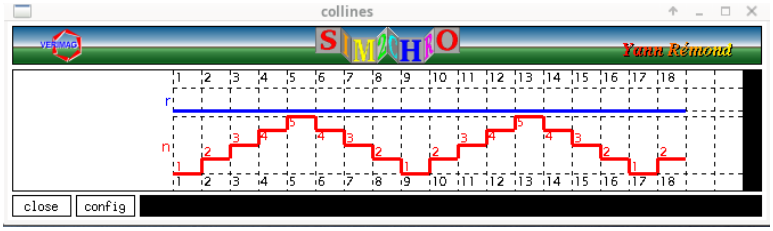
Nous abordons ici le même questionnement dans le cadre du projet Vélus. En tirant parti de la vérification du compilateur, nous pouvons espérer obtenir des garanties de bout en bout : une propriété démontrée sur l'exécution du programme source l'est aussi, par correction, sur l'exécution du programme cible. L'enjeu est donc de reproduire ou de proposer des mécanismes de preuves de programmes Lustre, en tenant compte à la fois des observations de Cécile Dumas et des spécificités liées à la compilation vérifiée dans Coq.

2.2.1 Manipulation du modèle synchrone relationnel

Une première idée est de raisonner directement sur les prédicats de la sémantique relationnelle en s'appuyant sur le formalisme développé dans Vélus. Nous avons développé à cet effet l'outil lusgen, largement inspiré de clightgen [17], permettant de charger la syntaxe abstraite d'un programme Lustre comme une structure de données directement dans une session Coq interactive. Prenons comme exemple le nœud suivant, définissant un flot d'entiers et dont l'exécution est représentée ci-dessous graphiquement grâce à l'outil Luciole fourni avec la distribution Lustre v4 [18].

[15] Dumas, 2000
[16] Mikáč, 2005
[17] `https://github.com/AbsInt/CompCert/tree/master/export`
[18] Raymond, 1992

```
node collines (r : bool) returns (n : int)
var up : bool; pn : int;
let
  up = true fby ((up and n < 5) or (not up and n <= 1));
  pn = 0 fby n;
  n = if up then pn + 1 else pn - 1;
tel
```



Notons que l'entrée r n'est là que pour satisfaire les contraintes du langage, interdisant les nœuds sans entrées. Le fichier Coq généré contient une suite de définitions constituant l'arbre de syntaxe abstraite du programme.

```
(* raccourcis vers les types primitifs de CompCert *)
Definition tint : type := Tint I32 Signed.
Definition tbool : type := Tint IBool Unsigned.
(* variables, de type [ident] *)
Definition _collines := 12%positive.
Definition _n := 14%positive.
Definition _pn := 16%positive.
Definition _r := 13%positive.
Definition _up := 15%positive.
(* déclarations, de type [list (ident * (type * clock))] *)
Definition in_collines := [(_r, (tbool, Cbase))].
Definition out_collines := [(_n, (tint, Cbase))].
Definition vars_collines := [(_up, (tbool, Cbase)); (_pn, ...)].
(* équations *)
Definition eqs_collines : list equation :=
[ ([_n],[Ecase (Evar _up (tbool, Cbase))
         [(T,[(Ebinop Oadd (Evar _pn tint)
              (Econst (Cint (Int.repr 1) I32 Signed)) tint)]);
          (F,[(Ebinop Osub (Evar _pn tint)
              (Econst (Cint (Int.repr 1) I32 Signed)) tint)])]
         None _]);
  ([_pn],[Efby [Econst (Cint (Int.repr 0) I32 Signed)]
            [Evar _n (tint, Cbase)]]);
  ([_up], ...) ].
(* noeud *)
Program Definition node_collines : node :=
  mk_node _collines in_collines out_collines
    vars_collines eqs_collines _ _ _. (* obligations *)
(* programme *)
Definition G : global := mk_global _ [node_collines].
```

L'utilisateur peut ensuite énoncer une propriété à démontrer sur l'exécution du programme. En l'occurrence, le prédicat co-inductif `Forall_Str P s`, spécifiant que la propriété P doit être satisfaite par tous les éléments d'un flot s, est appliqué directement au flot de sortie et assure que toutes ses valeurs sont présentes, entières et comprises entre 1 et 5. L'hypothèse `SEM` correspond à l'existence d'une sémantique relationnelle, i.e. $G \vdash \mathsf{collines}(xs) \Downarrow ys$.

```
Lemma spec_collines : ∀ xs ys
  (SEM : sem_node G _collines [xs] [ys]),
  Forall_Str (λ v ⇒ match v with
   | present (Vint i) ⇒
       Int.lt Int.zero i && Int.lt i (Int.repr 6) = true
   | absent ⇒ ⊥
   end) ys.
```

Équations et prédicats Plusieurs inversions successives de l'hypothèse `SEM` sont nécessaires pour retrouver dans le contexte les prédicats relatifs aux différentes équations. Par souci de lisibilité, nous utilisons le mécanisme de notation fourni par Coq pour afficher plus simplement les éléments de la syntaxe. Ainsi, le terme `Ebinop Oadd (Evar _pn tint) (Econst ...)` s'affiche `(_pn + 1)` dans l'hypothèse H_8, par exemple. Chaque équation de la forme $x = e$ introduit deux hypothèses, l'une associant un flot à x dans l'histoire et l'autre mettant en relation ce flot et l'expression e par la sémantique. C'est le cas des hypothèses (H_3, H_7) et (H_4, H_8). Nous avons inversé une fois le prédicat `sem_exp` associé à l'équation de *up* pour obtenir H_5 et H_6, ce qui introduit un flot intermédiaire s_2 et fait apparaître dans H_5 une instance du prédicat FBY tel que défini en section 2.1.5.

```
xs, ys, s1, s2, s3 : Stream svalue
H : History
bs := base_of [xs]
H1 : Env.find _r H = Some xs
H2 : Env.find _up H = Some s1
H3 : Env.find _pn H = Some s3
H4 : Env.find _n H = Some ys
H5 : fby (const bs (Venum 1)) s2 s1
H6 : sem_exp H bs ((_up and _n < 5) or (not _up and _n ≤ 1)) s2
H7 : sem_exp H bs (0 fby _n) s3
H8 : sem_exp H bs (if _up then (_pn + 1) else (_pn - 1)) ys
============================
Forall_Str (λ v ⇒ match v with
 | present (Vint i) ⇒
     Int.lt Int.zero i && Int.lt i (Int.repr 6) = true
 | _ ⇒ ⊥
 end) ys.
```

Une difficulté qui apparaît immédiatement est celle de la compréhensibilité du contexte de preuve. À la différence d'un outil de model-checking, qui encoderait le programme comme un ensemble de formules logiques à donner à un solveur automatique, une approche interactive doit à tout prix conserver la lisibilité du contexte, des hypothèses et des formules à prouver. L'apparition de flots intermédiaires à chaque inversion en constitue un obstacle majeur et semble inhérent à la manipulation d'une sémantique relationnelle. D'autre part, la nature des hypothèses ne permet pas de véritable raisonnement équationnel. Il est par exemple impossible de remplacer directement *pn* par sa définition dans H_8 pour ensuite raisonner avec des propriétés algébriques de l'addition et du FBY.

Principes de preuve Plusieurs techniques peuvent être employées afin de démontrer la propriété. L'utilisation de la tactique `cofix` étant particulièrement délicate en raison des critères syntaxiques imposés par Coq, nous encapsulons son effet dans un schéma d'induction proche de celui proposé par Cécile Dumas [19].

```
Theorem Forall_Str_ind :
  ∀ (A C : Type) (ctx : C) (ys : Stream A)
    (upd_ctx : C → C)
    (P : A → Prop)
    (Q : C → Stream A → Prop)
    (Hp : P (hd ys))
    (Hq : Q ctx ys)
    (Tp : ∀ctx ys, P (hd ys) → Q ctx ys → P (hd (tl ys)))
    (Tq : ∀ctx ys, Q ctx ys → Q (upd_ctx ctx) (tl ys)),
    Forall_Str P ys.
```

L'objet *ctx* s'instancie en les données du contexte. Dans notre exemple il s'agit des flots et de l'histoire, soit la structure (xs, s_1, s_2, s_3, H). L'invariant Q s'instancie en les relations utiles à la preuve de P. Dans notre exemple il s'agit de la conjonction des hypothèses H_1 à H_8, obtenues par inversion des prédicats de sémantique. Pour prouver que la propriété P tient sur chaque élément de *ys*, il faut d'abord prouver qu'elle tient sur la tête du flot (hypothèse H_p) puis que si Q tient sur un contexte environnant *ys* alors P tient sur le second élément de *ys* (hypothèse T_p). Il faut aussi prouver que la relation Q tient sur le contexte initial (hypothèse H_q) et qu'elle est conservée par application de tl à *ys* (hypothèse T_q). Dans notre exemple, nous pouvons instancier la fonction de mise à jour du contexte très simplement :

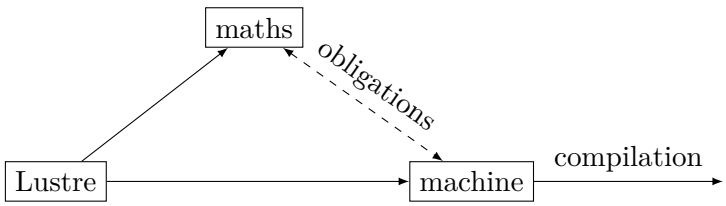
$$upd\_ctx := \lambda\,(xs, s_1, s_2, s_3, H) \to (\mathsf{tl}\ xs, \mathsf{tl}\ s_1, \mathsf{tl}\ s_2, \mathsf{tl}\ s_3, \mathsf{map}\ \mathsf{tl}\ H).$$

Malheureusement, comme l'a déjà remarqué Cécile Dumas, instancier Q avec l'ensemble des prédicats du contexte relationnel n'est pas suffisant

[19] Dumas, 2000, §9.2

dès lors qu'ils impliquent le FBY. L'hypothèse H_5, par exemple, est vraie initialement mais n'est pas conservée par *upd_ctx*. Il faudrait alors énoncer manuellement une généralisation de Q pour les prédicats relatifs au FBY et FBY_1, en renforçant éventuellement l'hypothèse sur la valeur mémorisée par FBY_1. De plus, la prise en compte de valeurs `abs` dans les flots multiplie le nombre de cas à traiter et alourdit considérablement le raisonnement.

Arithmétique Une autre difficulté que nous rencontrons concerne la gestion des types de données, notamment numériques. Par défaut, les valeurs transportées par les flots échantillonnés de la sémantique de Vélus sont identiques à celles définies dans Clight et reflètent toute la complexité du traitement des valeurs numériques, entières et flottantes. Par exemple, la fonction `Int.repr : Z → int`, définie dans CompCert et utilisée dans la spécification du programme ci-dessus, convertit un entier relatif en entier machine dont la représentation dépend de l'architecture ciblée. De même, la sémantique de l'addition `Oadd` est définie selon le type de ses opérandes et peut entraîner des débordements. Raisonner avec de telles définitions est nécessaire pour obtenir une preuve de bout en bout mais s'avère souvent compliqué, même sur des programmes simples, car nombre de propriétés algébriques ne sont pas garanties sur les entiers modulo. Nous proposons de remédier à ce problème grâce à un prototype inspiré des travaux sur AutoCorres, en Isabelle [20], ou encore des outils comme Frama-C ou Atelier B.



L'idée est de fournir, pour chaque nœud Lustre, deux interprétations relationnelles. La première correspond à celle du compilateur, où les opérateurs arithmétiques viennent de CompCert, tandis la seconde associe aux opérateurs leur interprétation mathématique « idéalisée », facilitant le raisonnement interactif. La correspondance des deux interprétations est assurée sous certaines conditions, notamment de non-débordement des opérations entières, déchargées dans des obligations de preuve.

Formellement, nous définissons le modèle mathématique sous forme de règles d'inférence décrivant conjointement la relation sémantique et les obligations. Dans ce modèle, l'environnement ρ associe aux variables du programme un flot dont les éléments sont des objets mathématiques.

$$\rho : \mathsf{ident} \to \mathsf{Stream}\ (\mathbb{Z} \mid \mathbb{N} \mid \dots)$$

[20] Greenaway *et al.*, 2014

À des fins de simplicité, nous ne traitons dans ce prototype qu'une forme restreinte du langage, proche de NLustre, forme normalisée de Lustre dans le compilateur. Dans ce sous-langage, en particulier, un fby ne peut apparaître qu'à la racine du membre droit d'une équation. De plus, nous ne considérons que le cas des flots non échantillonnés, et excluons pour l'instant les opérateurs when et merge. Dans ce contexte, un flot s peut être vu comme une fonction totale $\mathbb{N} \to \mathsf{val}$ dont la i-ème valeur est $s(i)$, toujours présente. Deux règles sont alors nécessaires pour les équations, l'une pour le cas combinatoire et l'autre pour le cas du fby. Chaque jugement de la forme $\rho \vdash \ldots ; P$ définit une contrainte sur l'environnement ainsi que des obligations, collectées dans la formule P. Les jugements des expressions sont de plus paramétrés par un entier i, déterminant l'instant auquel s'applique l'obligation.

$$\frac{\forall i \quad \rho, i \vdash e \Downarrow \rho(x) \; ; \; P}{\rho \vdash x = e \; ; \; \forall i, P} \qquad \frac{\begin{array}{cc} \rho(x)(0) = s_0(0) & \forall i, \rho(x)(i+1) = s(i) \\ \rho, 0 \vdash e_0 \Downarrow s_0 \; ; \; P_0 & \forall i \quad \rho, i \vdash e \Downarrow s \; ; \; P \end{array}}{\rho \vdash x = e_0 \,\mathsf{fby}\, e \; ; \; P_0 \wedge \forall i, P}$$

Le cas du fby se décompose donc en deux jugements, l'un pour l'instant initial et l'autre pour les instants suivants. À la différence de Vélus, l'absence d'échantillonnage n'impose pas de synchronisation entre les deux flots s et s_0.

Dans certaines règles, nous utilisons le prédicat suivant, emprunté à la bibliothèque du projet VST [21] et caractérisant la validité d'une valeur entière vis-à-vis de sa représentation machine.

$$\mathsf{repable\_signed}\ (v : \mathbb{Z}) := \mathsf{Int.min\_signed} \leq v \leq \mathsf{Int.max\_signed}$$

Les règles de la figure 2.5 ci-contre donnent un aperçu des contraintes et obligations introduites par les différentes expressions. Certains opérateurs, comme le or logique (donné ici) ou encore and, not ou les comparaisons n'introduisent pas d'obligations particulières car leur interprétation mathématique est quasi similaire à celle de la machine. D'autres comme l'addition, l'opposé (donnés ici) ou la soustraction et la division sur les entiers, nécessitent de connaître des bornes sur le résultat du calcul afin de garantir sa préservation en arithmétique machine.

On exprime la correspondance entre les valeurs des deux modèles à l'aide de la fonction `Int.signed : int → Z` qui envoie les entiers machines sur les entiers relatifs et qui satisfait la relation $\mathsf{Int.repr}(\mathsf{Int.signed}\ n) = n$, pour tout entier machine n. L'équivalence notée $\rho \simeq_d H$ d'un environnement ρ et d'une histoire H sur un domaine d de variables est définie par la condition suivante :

$$\forall x \in d,\ \forall i \in \mathbb{N},\ \exists v,\ H(x)(i) = v \wedge \mathsf{Int.signed}\ v = \rho(x)(i).$$

Nous démontrons la correction du système d'obligations sur l'ensemble des équations $x_1 = e_1; \ldots; x_n = e_n$ d'un nœud du langage. On nomme *ins*

[21] Appel *et al.*, 2014

$$\frac{\rho(x)(i) = s(i)}{\rho, i \vdash x \Downarrow s\ ;\ \top} \qquad \frac{s(i) = c \qquad c : \mathbb{Z}}{\rho, i \vdash c \Downarrow s\ ;\ \mathsf{repable\_signed}\ c}$$

$$\frac{\rho, i \vdash e \Downarrow s_1\ ;\ P \qquad s(i) = -_{\mathbb{Z}} s_1(i)}{\rho, i \vdash -e \Downarrow s\ ;\ P \wedge \mathsf{Int.min\_signed} < s_1(i)}$$

$$\frac{\rho, i \vdash e_1 \Downarrow s_1\ ;\ P_1 \qquad \rho, i \vdash e_2 \Downarrow s_2\ ;\ P_2 \qquad s(i) = s_1(i) +_{\mathbb{Z}} s_2(i)}{\rho, i \vdash e_1 + e_2 \Downarrow s\ ;\ P_1 \wedge P_2 \wedge \mathsf{repable\_signed}\ (s_1(i) +_{\mathbb{Z}} s_2(i))}$$

$$\frac{\rho, i \vdash e_1 \Downarrow s_1\ ;\ P_1 \qquad \rho, i \vdash e_2 \Downarrow s_2\ ;\ P_2 \qquad s(i) = s_1(i) \vee s_2(i)}{\rho, i \vdash e_1\ \mathsf{or}\ e_2 \Downarrow s\ ;\ P_1 \wedge P_2}$$

$$\frac{\begin{array}{c}\rho, i \vdash e_1 \Downarrow s_1\ ;\ P_1 \qquad \rho, i \vdash e_2 \Downarrow s_2\ ;\ P_2 \\ \rho, i \vdash e \Downarrow s_e\ ;\ P_e \qquad s(i) = s_e(i)?s_1(i) : s_2(i)\end{array}}{\rho, i \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow s\ ;\ \mathsf{if}\ s_e(i)\ \mathsf{then}\ P_e \wedge P_1\ \mathsf{else}\ P_e \wedge P_2}$$

FIGURE 2.5 – Règles d'obligations

le domaine des variables d'entrée et $outs = \{x_1, \ldots, x_n\}$ celui des variables locales et de sortie. Nous montrons que si les équations satisfont bien le principe de causalité syntaxique décrit en section 2.1.6 alors le théorème suivant s'applique et les environnements des deux modèles sémantiques coïncident.

Théorème (correction du prototype)**.**

si $\rho \simeq_{ins} H$
et $\forall k \in [1 \ldots n] \quad H, (bs := \mathsf{T}^\omega) \vdash x_k = e_k$
et $\forall k \in [1 \ldots n] \quad \rho \vdash x_k = e_k\ ;\ P_k$
et $P_1 \wedge \cdots \wedge P_n$
alors $\rho \simeq_{outs} H$.

Un corollaire immédiat de ce résultat est que si les obligations $P_1, \ldots, P_n$ sont vérifiées et qu'un propriété est vraie sur les flots de l'environnement du modèle mathématique, alors elle l'est aussi sur les flots du modèle machine et, par correction du compilateur, sur les valeurs successives des traces du programme assembleur généré.

En appliquant ce principe à l'exemple des collines en Lustre, on obtient un environnement formé des trois flots *up*, *n* et *pn* avec leur définition relationnelle impliquant des opérateurs mathématiques. Quatre sous-buts sont générés, l'un pour l'invariant du programme à démontrer et les autres pour les obligations relatives aux trois équations. Celles de *up* et *pn* sont

trivialement vraies tandis que celle de n ainsi que l'invariant se démontrent aisément par récurrence sur i et en réécrivant les hypothèses du contexte.

```
(* switch_init i v0 vs := match i with O ⇒ v0 |S i ⇒ vs i end. *)
up : stream bool
n, pn : stream Z
H_up : ∀i, up i =
       switch_init i true
         (λ j ⇒ up j && (n j <? 5) || negb (up j) && (n j ≤? 1))
H_n : ∀i, n i = (if up i then pn i + 1 else pn i - 1)
H_pn : ∀i, pn i = switch_init i 0 n
============================
 ∀ i : nat, 0 < n i < 6

subgoal 2 is:
 ∀ i : nat, switch_init i ⊤ (λ _ : nat ⇒ ⊤ )
subgoal 3 is:
 ∀ i : nat,
 if up i
 then -2147483648 ≤ pn i + 1 ≤ 2147483647
 else -2147483648 ≤ pn i - 1 ≤ 2147483647
subgoal 4 is:
 ∀ i : nat, switch_init i ⊤ (λ _ : nat ⇒ ⊤ )
```

Cette approche semble donc prometteuse car elle permet une manipulation très naturelle des équations à l'aide des tactiques d'arithmétique disponibles dans Coq (`auto with arith`, `lia`...). Le raisonnement sur les contraintes numériques pourrait être poussé beaucoup plus loin en utilisant des méthodes d'interprétation abstraite, développées notamment dans les projets Astrée [22] et Verasco [23], mais cela sortirait du cadre de cette thèse.

Leçons L'expérience menée dans cette partie nous montre qu'il est possible mais difficile de raisonner directement avec la sémantique implémentée dans le compilateur. Le traitement des opérateurs machine peut toutefois être facilité grâce à un mécanisme d'abstraction des calculs et des valeurs. La question de la lisibilité du contexte et de la manipulation des opérateurs temporels reste la principale difficulté car le modèle de Vélus opère sur des flots échantillonnés et des prédicats synchronisants, comme FBY/FBY$_1$. Il est beaucoup plus simple de raisonner sur cet opérateur sans la synchronisation, par simple disjonction de cas, comme l'illustre l'exemple ci-dessus avec `switch_init`. Nous verrons en section 2.2.3 qu'un modèle de Kahn offre justement cette possibilité.

[22] Boillot et Feret, 2023
[23] Jourdan *et al.*, 2015

2.2.2 Modèle synchrone à types dépendants

En 2001, Sylvain Boulmé et Grégoire Hamon ont proposé une formalisation en Coq de la sémantique dénotationnelle de Lucid Synchrone [24], langage fonctionnel inspiré de Lustre avec des constructions d'ordre supérieur. Ils encodent, par l'intermédiaire d'un plongement superficiel, les types des flots et des horloges du langage comme des types dépendants en Coq. À la différence d'un modèle relationnel, qui définit des contraintes sur un ensemble de flots, un modèle dénotationnel décrit une procédure déterministe permettant de calculer ces flots. En Lucid Synchrone, comme en Lustre, la sémantique usuelle consiste à calculer par itérations successives le plus petit point fixe de l'ensemble des équations d'un nœud. C'est cette procédure que les auteurs proposent d'encoder. Le développement initial a été remarquablement analysé et mis à jour par Chantal Keller en 2008 dans un projet d'études [25].

Dans leur formalisme, la famille des types des flots dépendants $\mathsf{Str}\ A\ c$ est paramétrée par A, le type des valeurs et c, une horloge qui spécifie la disponibilité des valeurs à chaque cycle. Le type de l'horloge c est clock, synonyme de $\mathsf{Stream\ bool}$, flot de valeurs booléennes. Le constructeur des flots dépendants a le type suivant :

$$\forall c : \mathsf{clock} \to \mathsf{Ivalue}\ A\ (\mathsf{hd}\ c) \to \mathsf{Str}\ A\ (\mathsf{tl}\ c) \to \mathsf{Str}\ A\ c.$$

Le premier argument, souvent inféré, spécifie l'horloge du flot. Le second est l'élément en tête du flot. Son type Ivalue dispose de constructeurs pour les valeurs présentes, $v : A \to \mathsf{Ivalue}\ A\ \mathsf{T}$, absentes, $\mathsf{abs} : \mathsf{Ivalue}\ A\ \mathsf{F}$ et indéterminées, $\mathsf{Fail} : \mathsf{Ivalue}\ A\ \mathsf{T}$. Ces dernières représentent à la fois le résultat d'un calcul qui échoue, notamment par propagation d'un autre Fail, ou plus généralement une valeur non encore disponible au moment du calcul. Elles sont utilisées pour définir un flot cadencé $\bot(c)$, absent lorsque c est vraie, Fail sinon, à partir duquel le point fixe d'une équation est calculé. La sémantique de l'opérateur when, par exemple, est définie par deux fonctions co-récursives.

```
CoFixpoint sp_on (c : clock) (lc : samplStr bool c) : clock :=
  Cons (match hd lc with
        | Val x ⇒ x
        | Abs ⇒ false
        | Fail ⇒ true
        end) (sp_on (tl lc)).

CoFixpoint sp_when (c : clock) (x : Str A c) (b : Str bool c)
                                  : Str A (sp_on c b) := ...
```

La première calcule le flot correspondant à la sous-horloge induite et la seconde effectue l'échantillonnage du flot de valeurs. Les arguments x

[24] Boulmé et Hamon, 2001b,a

[25] http://www.lri.fr/~keller/Documents-etudes/Systemes-synchrones/rapport.pdf

et b doivent avoir la même horloge c tandis que le résultat a bien une sous-horloge dépendant de la valeur de b. Des fonctions de flots sont introduites similairement pour les autres opérateurs du langage.

D'autre part, les auteurs proposent un opérateur générique de point fixe permettant de calculer à la fois les solutions d'équations mutuellement récursives et le résultat des nœuds qui, contrairement à ceux de Lustre, peuvent s'appeler récursivement.

```
Definition rec1 : ∀ (B : Set) (C : B → clock),
  ((∀ x : B, Str A (C x)) → ∀ x : B, Str A (C x)) →
  ∀ y : B, Str A (C y) := ...
```

Des hypothèses supplémentaires sont ensuite nécessaires pour démontrer que les flots obtenus par ce combinateur ne contiennent pas d'élément Fail et ne sont pas des solutions erronées ou dégénérées.

La sémantique d'un programme Lucid Synchrone s'obtient donc en composant ces différentes constructions avec des λ-abstractions de Coq et des paires dépendantes d'horloges et de flots. L'expression des règles de typage et de cadencement sous forme de types dépendants permet de définir les opérateurs sémantiques comme des fonctions en considérant uniquement les cas pertinents, comme dans un modèle relationnel. Les cas non désirés sont éliminés par construction des termes.

Le principal obstacle à l'adoption de cette approche dans Vélus semble être l'imbrication des contraintes entre les types représentant la syntaxe d'un programme en cours de compilation et ceux représentant sa sémantique. Le compilateur doit construire ces valeurs, validant essentiellement les règles de typage dans le processus et perdant ainsi l'un des avantages du plongement superficiel. De plus, pour prouver les propriétés des programmes, il faut manipuler des termes dépendants arbitraires à l'aide de tactiques, ce qui, par expérience, est particulièrement difficile et pas souhaitable dans un processus de vérification interactif.

2.2.3 Modèle dénotationnel de Kahn

Les langages synchrones, Lustre en particulier, utilisent leur système d'horloges pour former un sous-ensemble de réseaux de Kahn ayant la propriété de s'exécuter en mémoire bornée. Comme nous l'avons constaté, les spécificités liées aux horloges et à leur synchronisation compliquent fortement le raisonnement sur l'exécution des programmes. Cécile Dumas montre dans sa thèse qu'il est beaucoup plus approprié d'« ignorer » l'hypothèse synchrone et de se placer dans le contexte plus général du modèle de Kahn, dont nous rappelons ici les éléments principaux. Pour plus de détails, nous renvoyons le lecteur intéressé aux notes de cours de Marc Pouzet [26].

[26] Pouzet, 2019

Domaine des flots Soit V un ensemble de valeurs. Les flots du modèle sont définis comme l'ensemble $V^\infty = V^* \cup V^\omega$ des séquences finies ou infinies d'éléments de V. On appelle $\bot$ le flot vide et on note $v \cdot s$ le flot de tête $v \in V$ et de queue $s \in V^\infty$. Le domaine des flots est équipé de l'ordre préfixe des séquences, noté $s_1 \leq s_2$ si s_1 est un préfixe de s_2. Par exemple, $\bot \leq a \cdot \bot \leq a \cdot b \cdot \cdots$ mais $a \cdot b \cdot \bot \not\leq a \cdot c \cdot \bot$. La structure $(V^\infty, \leq, \bot)$ ainsi définie constitue un CPO (*complete partial order*) dont chaque sous-ensemble totalement ordonné S possède une borne supérieure, notée $\operatorname{lub} S$.

Continuité et point fixe Une fonction f sur un CPO est continue au sens de Scott si elle commute avec l'opération de borne supérieure. Autrement dit, pour toute chaîne C, $f(\operatorname{lub} C) = \operatorname{lub}\{f(c), c \in C\}$. Une fonction continue au sens de Scott est aussi monotone. Dans le cas des flots, une fonction monotone préserve l'ordre préfixe. Si cette fonction décrit l'évolution d'un programme, sa monotonicité correspond à l'idée intuitive que les valeurs des flots sont calculées séquentiellement et ne sont plus jamais modifiées par la suite. Le théorème de Kleene assure l'existence, pour chaque fonction continue f, d'un plus petit point fixe noté $\operatorname{fix} f$. Il est obtenu par itérations successives de f à partir de l'élément $\bot$, soit $\operatorname{fix} f = \operatorname{lub}\{f^n(\bot), n \in \mathbb{N}\}$. En découle un principe de preuve, nommé « induction continue » :

$$P(\bot) \wedge (\forall x, P(x) \Rightarrow P(f\ x)) \Rightarrow P(\operatorname{fix} f).$$

Ce résultat est valable pour toute propriété P *admissible*, c'est-à-dire que si P tient sur tous les éléments d'une chaîne C, alors elle doit aussi tenir sur $\operatorname{lub} C$. C'est le cas de toute propriété de sûreté sur les éléments d'un flot.

Sémantique à flots de données Dans son article fondateur [27], Gilles Kahn propose d'appliquer l'opérateur de point fixe pour trouver les solutions d'équations de flots mutuellement récursives. Il montre que dans les langages à flots de données, les équations sont de la forme $y = f(x)$, où f est la composition de fonctions continues, dont on peut extraire un point fixe dans V^∞. Ce principe s'applique directement aux équations Lustre. Nous donnons en figure 2.6 la définition des fonctions primitives (continues) du langage [28], dans une forme simplifiée, sans type énumérés. Notons en particulier le traitement fondamentalement différent des opérateurs merge et if-then-else, ainsi que la simplicité du fby. Pour être totales, ces définitions doivent être complétées avec le cas du flot vide sur un de leurs arguments. Ainsi, seul le fby produit une valeur lorsque son second argument est vide, ce qui permet de calculer le point fixe non trivial d'une équation comme `x = 0 fby x + 1`. Une équation ne respectant pas le principe de causalité, comme `x = x + 1`, se voit attribuer la sémantique $\bot$, au même titre que `x = y when false`, qui est bien causale

[27] Kahn, 1974
[28] Colaço et Pouzet, 2003, figure 1

$$
\begin{array}{ll}
\mathsf{const}\ c & = c \cdot \mathsf{const}\ c \\
\mathsf{lift}_{\oplus}\ (v_1 \cdot s_1)\ (v_2 \cdot s_2) & = (v_1 \oplus v_2) \cdot (\mathsf{lift}_{\oplus}\ s_1\ s_2) \\
\mathsf{fby}\ (v_1 \cdot s_1)\ s_2 & = v_1 \cdot s_2 \\
\mathsf{when}\ (v \cdot s)\ (\mathsf{T} \cdot s_c) & = v \cdot (\mathsf{when}\ s\ s_c) \\
\mathsf{when}\ (v \cdot s)\ (\mathsf{F} \cdot s_c) & = \mathsf{when}\ s\ s_c \\
\mathsf{whenot}\ (v \cdot s)\ (\mathsf{T} \cdot s_c) & = \mathsf{whenot}\ s\ s_c \\
\mathsf{whenot}\ (v \cdot s)\ (\mathsf{F} \cdot s_c) & = v \cdot (\mathsf{whenot}\ s\ s_c) \\
\mathsf{merge}\ (\mathsf{T} \cdot s_c)\ (v_1 \cdot s_1)\ s_2 & = v_1 \cdot (\mathsf{merge}\ s_c\ s_1\ s_2) \\
\mathsf{merge}\ (\mathsf{F} \cdot s_c)\ s_1\ (v_2 \cdot s_2) & = v_2 \cdot (\mathsf{merge}\ s_c\ s_1\ s_2) \\
\mathsf{if\text{-}then\text{-}else}\ (\mathsf{T} \cdot s_c)\ (v_1 \cdot s_1)\ (v_2 \cdot s_2) & = v_1 \cdot (\mathsf{if\text{-}then\text{-}else}\ s_c\ s_1\ s_2) \\
\mathsf{if\text{-}then\text{-}else}\ (\mathsf{F} \cdot s_c)\ (v_1 \cdot s_1)\ (v_2 \cdot s_2) & = v_2 \cdot (\mathsf{if\text{-}then\text{-}else}\ s_c\ s_1\ s_2) \\
\hline
\mathsf{lift}_{\oplus}\ s_1\ s_2 & = \bot \quad \text{si } s_1 = \bot \text{ ou } s_2 = \bot \\
\mathsf{fby}\ s_1\ s_2 & = \bot \quad \text{si } s_1 = \bot \\
\mathsf{when}\ s\ s_c & = \bot \quad \text{si } s = \bot \text{ ou } s_c = \bot \\
\mathsf{whenot}\ s\ s_c & = \bot \quad \text{si } s = \bot \text{ ou } s_c = \bot \\
\mathsf{merge}\ s_c\ s_1\ s_2 & = \bot \quad \text{si } s_c, s_1 \text{ ou } s_2 = \bot \\
\mathsf{if\text{-}then\text{-}else}\ s_c\ s_1\ s_2 & = \bot \quad \text{si } s_c, s_1 \text{ ou } s_2 = \bot
\end{array}
$$

FIGURE 2.6 – Sémantique de Kahn des primitives Lustre

$$
\begin{array}{ll}
\mathsf{const}\ c\ (\mathsf{F}\cdot bs) & = \mathsf{abs}\cdot \mathsf{const}\ c\ bs \\
\mathsf{const}\ c\ (\mathsf{T}\cdot bs) & = c\cdot \mathsf{const}\ c\ bs \\
\mathsf{lift}_{\oplus}\ (\mathsf{abs}\cdot s_1)\ (\mathsf{abs}\cdot s_2) & = \mathsf{abs}\cdot (\mathsf{lift}_{\oplus}\ s_1\ s_2) \\
\mathsf{lift}_{\oplus}\ (v_1\cdot s_1)\ (v_2\cdot s_2) & = (v_1\oplus v_2)\cdot (\mathsf{lift}_{\oplus}\ s_1\ s_2) \\
\mathsf{fby}\ (abs\cdot s_1)\ (abs\cdot s_2) & = abs\cdot \mathsf{fby}\ s_1\ s_2 \\
\mathsf{fby}\ (v_1\cdot s_1)\ (v_2\cdot s_2) & = v_1\cdot \mathsf{fby1}\ v_2\ s_1\ s_2 \\
\mathsf{fby1}\ v\ (abs\cdot s_1)\ (abs\cdot s_2) & = abs\cdot \mathsf{fby1}\ v\ s_1\ s_2 \\
\mathsf{fby1}\ v\ (v_1\cdot s_1)\ (v_2\cdot s_2) & = v\cdot \mathsf{fby1}\ v_2\ s_1\ s_2 \\
\mathsf{when}\ (\mathsf{abs}\cdot s)\ (\mathsf{abs}\cdot s_c) & = \mathsf{abs}\cdot (\mathsf{when}\ s\ s_c) \\
\mathsf{when}\ (v\cdot s)\ (\mathsf{T}\cdot s_c) & = v\cdot (\mathsf{when}\ s\ s_c) \\
\mathsf{when}\ (v\cdot s)\ (\mathsf{F}\cdot s_c) & = \mathsf{abs}\cdot \mathsf{when}\ s\ s_c \\
\mathsf{whenot}\ (\mathsf{abs}\cdot s)\ (\mathsf{abs}\cdot s_c) & = \mathsf{abs}\cdot (\mathsf{whenot}\ s\ s_c) \\
\mathsf{whenot}\ (v\cdot s)\ (\mathsf{F}\cdot s_c) & = v\cdot (\mathsf{whenot}\ s\ s_c) \\
\mathsf{whenot}\ (v\cdot s)\ (\mathsf{T}\cdot s_c) & = \mathsf{abs}\cdot \mathsf{whenot}\ s\ s_c \\
\mathsf{merge}\ (\mathsf{abs}\cdot s_c)\ (\mathsf{abs}\cdot s_1)\ (\mathsf{abs}\cdot s_2) & = \mathsf{abs}\cdot (\mathsf{merge}\ s_c\ s_1\ s_2) \\
\mathsf{merge}\ (\mathsf{T}\cdot s_c)\ (v_1\cdot s_1)\ (\mathsf{abs}\cdot s_2) & = v_1\cdot (\mathsf{merge}\ s_c\ s_1\ s_2) \\
\mathsf{merge}\ (\mathsf{F}\cdot s_c)\ (\mathsf{abs}\cdot s_1)\ (v_2\cdot s_2) & = v_2\cdot (\mathsf{merge}\ s_c\ s_1\ s_2) \\
\mathsf{if\text{-}then\text{-}else}\ (\mathsf{abs}\cdot s_c)\ (\mathsf{abs}\cdot s_1)\ (\mathsf{abs}\cdot s_2) & = \mathsf{abs}\cdot (\mathsf{if\text{-}then\text{-}else}\ s_c\ s_1\ s_2) \\
\mathsf{if\text{-}then\text{-}else}\ (\mathsf{T}\cdot s_c)\ (v_1\cdot s_1)\ (v_2\cdot s_2) & = v_1\cdot (\mathsf{if\text{-}then\text{-}else}\ s_c\ s_1\ s_2) \\
\mathsf{if\text{-}then\text{-}else}\ (\mathsf{F}\cdot s_c)\ (v_1\cdot s_1)\ (v_2\cdot s_2) & = v_2\cdot (\mathsf{if\text{-}then\text{-}else}\ s_c\ s_1\ s_2) \\
\hline
\mathsf{const}\ c\ bs & = \bot \quad \text{si } bs = \bot \\
\mathsf{lift}_{\oplus}\ (\mathsf{abs}\cdot s_1)\ (v_2\cdot s_2) & = \mathsf{err} \\
\mathsf{merge}\ (\mathsf{F}\cdot s_c)\ (v_1\cdot s_1)\ (\mathsf{abs}\cdot s_2) & = \mathsf{err} \\
\vdots &
\end{array}
$$

FIGURE 2.7 – Sémantique synchrone des primitives Lustre

mais ne produit aucune valeur. Cette spécificité rend le modèle de Kahn particulièrement difficile à encoder dans Coq, qui requiert que les calculs sur les structures infinies, dont les flots, soient productifs.

Exemple On peut donner un sémantique au nœud des collines en appliquant l'opérateur de point fixe sur toutes les équations à la fois. L'ensemble des flots up, pn et n est défini par la formule suivante.

$$\begin{aligned}
&\text{fix}\,(f := \lambda\,\{up, pn, n\} \to \\
&\quad \{\ up \mapsto \mathsf{fby}\ (\mathsf{const}\ \mathsf{T})\ (\mathsf{lift}_{\mathsf{or}}\ (\mathsf{lift}_{\mathsf{and}}\ up\ (\mathsf{lift}_{<}\ n\ (\mathsf{const}\ 5)))\\
&\qquad\qquad (\mathsf{lift}_{\mathsf{and}}\ (\mathsf{lift}_{\neg}\ up)\ (\mathsf{lift}_{\leq}\ n\ (\mathsf{const}\ 1))))\\
&\quad\ pn \mapsto \mathsf{fby}\ (\mathsf{const}\ 0)\ n\\
&\quad\ n \mapsto \mathsf{if\text{-}then\text{-}else}\ up\ (\mathsf{lift}_{+}\ pn\ (\mathsf{const}\ 1))\ (\mathsf{lift}_{-}\ pn\ (\mathsf{const}\ 1))\ \})
\end{aligned}$$

L'évaluation du point fixe itère la fonction f à partir de flots vides.

$$\{\bot,\bot,\bot\} \xrightarrow{f} \{\mathsf{T}\cdot\bot, 0\cdot\bot, \bot\} \xrightarrow{f} \{\mathsf{T}\cdot\bot, 0\cdot\bot, 1\cdot\bot\} \xrightarrow{f} \{\mathsf{T}\cdot\mathsf{T}\cdot\bot, 0\cdot 1\cdot\bot, 1\cdot\bot\}\cdots$$

On peut ensuite utiliser l'induction continue pour prouver que les valeurs du flot n sont bien comprises entre 1 et 5. Le cas de base $\bot$ est trivial. Il faut ensuite montrer que si la propriété est vraie sur n alors elle l'est aussi sur le flot $n' := f(\{up, pn, n\}).n$, projection sur n de l'ensemble après une itération de f. En déroulant l'opérateur fby et en remplaçant up et pn par leurs définitions respectives dans n', on obtient que :

— $\mathsf{hd}\ n' = \mathsf{if}\ \mathsf{T}\ \mathsf{then}\ (0+1)\ \mathsf{else}\ (0-1) = 1$

— $\mathsf{tl}\ n' = \mathsf{if\text{-}then\text{-}else}\ (\mathsf{lift}_{\mathsf{or}}\ (\mathsf{lift}_{\mathsf{and}}\ up\ (\mathsf{lift}_{<}\ n\ (\mathsf{const}\ 5)))$
$(\mathsf{lift}_{\mathsf{and}}\ (\mathsf{lift}_{\neg}\ up)\ (\mathsf{lift}_{\leq}\ n\ (\mathsf{const}\ 1))))$
$(\mathsf{lift}_{+}\ n\ (\mathsf{const}\ 1))\ (\mathsf{lift}_{-}\ n\ (\mathsf{const}\ 1))$.

La propriété est immédiatement vérifiée sur la tête $\mathsf{hd}\ n'$. L'implication est également très simple à vérifier sur la queue $\mathsf{tl}\ n'$, car cette dernière a une définition purement combinatoire, n'impliquant plus de fby.

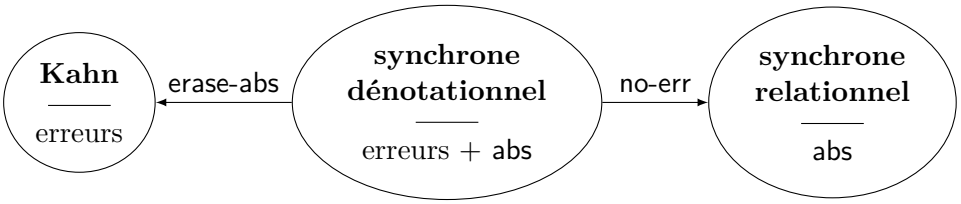
2.2.4 Modèle dénotationnel synchrone

Le modèle synchrone impose au modèle de Kahn des contraintes temporelles strictes par l'intermédiaire de la valeur spéciale abs, afin de garantir des communications en mémoire bornée. Le modèle synchrone constitue une sorte de spécification du code séquentiel que génère le compilateur : à chaque cycle logique de l'exécution, une valeur d'un flot peut être présente, et donc calculée et éventuellement utilisée, ou bien absente, auquel cas elle est indisponible pour les opérations du cycle courant. Formellement, il s'agit de se placer dans le domaine des flots $(V \cup \mathsf{abs})^{\infty}$, et de redéfinir les primitives

du langage selon les équations de la figure 2.7 [29]. Comme dans le modèle synchrone relationnel, précédemment décrit en section 2.1.5, l'opérateur des constantes est paramétré par un flot de booléens *bs* définissant l'horloge de base du nœud, et le fby est découpé en deux fonctions. À nouveau, ces définitions ne sont pas totales et doivent être complétées pour traiter non seulement le cas des flots vides, mais aussi celui d'erreurs de synchronisation, survenant par exemple lorsque l'opérateur lift est appliqué à deux flots qui ne sont pas échantillonnés de la même façon. Si le comportement à adopter en cas d'erreur est arbitraire dans un cadre purement théorique, il l'est beaucoup moins lorsqu'il s'agit d'encoder le modèle sémantique dans un compilateur vérifié en Coq. Nous proposerons au chapitre suivant un traitement des erreurs permettant de raisonner conjointement sur le typage des programmes et l'absence d'erreurs à l'exécution.

2.3 Discussion

Après comparaison des différentes représentations du langage, nous faisons le même constat que Cécile Dumas dans sa thèse : le modèle des réseaux de Kahn est celui qui semble convenir le mieux à la preuve interactive de programmes Lustre. Il ne nécessite pas de raisonner sur la synchronisation des flots et fournit un principe de preuve tout à fait adapté à des propriétés de sûreté. La question qui nous intéresse désormais est celle de son applicabilité à Vélus. En plus de définir en Coq le modèle de Kahn, il faut démontrer sa pertinence vis-à-vis du modèle relationnel existant dans le compilateur. Les deux modèles étant de nature fondamentalement différente, il semble difficile d'établir une correspondance directe entre eux. Nous proposons de décomposer le problème en encodant aussi la sémantique dénotationnelle synchrone présentée en section 2.2.4.



La gestion des erreurs constitue l'un des principaux défis à cette approche car les modèles dénotationnels sont, par définition, constructifs et définissent des fonctions totales sur la syntaxe. Les cas d'erreurs doivent donc être caractérisés précisément, contrairement au traitement du modèle relationnel.

Le second défi concerne l'opérateur de réinitialisation modulaire, qui n'existe pour l'instant que dans le compilateur et son modèle relationnel [30].

[29] Colaço et Pouzet, 2003, figure 2

[30] Bourke *et al.*, 2020

Quelques définitions ont été proposées dans la littérature mais n'ont jamais été encodées en Coq et leur type d'horloge soulève encore des questions auxquelles il convient de répondre avant de prouver la correspondance.

Enfin, il faut définir formellement la relation entre les flots des trois modèles. Si les flots des modèles synchrones sont identiques aux erreurs près, ce n'est pas le cas de ceux de Kahn, qui ne sont pas échantillonnés. Il nous faut alors définir une procédure d'effacement des `abs`, qui s'apparente à un passage au quotient et qui, nous le verrons, interagit de façon problématique avec la gestion des erreurs.

Chapitre 3

Modèle synchrone dénotationnel

Nous encodons une sémantique synchrone dénotationnelle pour Vélus en nous appuyant sur les travaux de Christine Paulin-Mohring, qui a développé une bibliothèque générale [1] pour les CPO, les fonctions continues et leur application aux équations de flots et aux réseaux de Kahn. Cette bibliothèque, particulièrement bien fournie mais complexe, n'est pas strictement nécessaire pour encoder un modèle synchrone car ce dernier garantit la productivité des opérateurs par l'ajout explicite de valeurs absentes. Nous choisissons cependant d'adopter ce formalisme en vue d'implémenter la procédure d'effacement des absences évoquée au chapitre précédent et qui peut, dans certains cas, conduire à la manipulation de flots finis.

La première partie de chapitre est dédiée à la description de la bibliothèque sous-jacente. Nous donnons ensuite tous les détails nécessaires à la définition d'une sémantique synchrone dénotationnelle pour le langage d'entrée de Vélus : valeurs, opérateurs, compositions. Dans le même temps, nous énonçons les conditions de la correspondance entre ce nouveau modèle et celui existant dans le compilateur. Nous donnons aussi, de façon plus schématique, les conditions de sa correspondance avec le modèle de Kahn.

3.1 Sémantique dénotationnelle constructive

Nous décrivons dans cette section les mécanismes fondamentaux de la bibliothèque CPO ainsi que ses composants utiles à la définition d'une sémantique pour Vélus. Une particularité de cette bibliothèque est qu'elle est *constructive* : les définitions des bornes supérieures de séquences monotones sont données sous forme de fonctions calculables, sans axiomes, qui peuvent par conséquent être extraites de Coq vers du code exécutable.

[1] Paulin-Mohring, 2009

3.1.1 Ordres complets partiels

Les définitions en Coq calquent les structures de la théorie des domaines par l'intermédiaire d'enregistrements dépendants, à la manière de *type classes*. Un élégant système de notations et de coercitions permet de manipuler et de comparer de façon très naturelle les éléments de différents domaines.

Un ensemble *partiellement ordonné* est constitué d'un support et d'une relation binaire réflexive et transitive.

```
Record ord : Type := mk_ord
{ tord :> Type;
  Ole : tord → tord → Prop;
  Ole_refl : ∀x : tord, Ole x x;
  Ole_trans : ∀x y z : tord, Ole x y → Ole y z → Ole x z }.
Infix "⪯ " := Ole.
```

On note $x \preceq y$ lorsque deux élément x et y satisfont `Ole`, qui s'étend à une relation d'équivalence $x \simeq y$ si $x \preceq y$ et $y \preceq x$. Parmi les instances de cette classe nous distinguons $\mathbb{N}_0$, l'ensemble `nat` de Coq équipé de l'ordre naturel sur les entiers, ou encore $O_1 \to_o O_2$ l'ensemble des fonctions entre deux ensembles ordonnés O_1 et O_2, qui constitue lui-même un ensemble ordonné, avec comme relation d'ordre $f \preceq g$ si $\forall x, f\ x \preceq g\ x$.

Une fonction $f : O_1 \to_o O_2$ est *monotone* si $\forall x\ y, x \preceq y \to f\ x \preceq f\ y$.

```
Definition monotonic (O1 O2 : ord) (f : O1 → O2) :=
  ∀ x y, x ⪯ y → f x ⪯ f y.
Record fmono (O1 O2 : ord) : Type := mk_fmono
{ fmonot :> O1 → O2;
  fmonotonic: monotonic fmonot }.
```

Un enregistrement dépendant permet là encore d'encapsuler une fonction Coq et sa preuve de monotonicité. On note $O_1 \to_m O_2$ l'ensemble ordonné des fonctions monotones. L'opérateur de composition $f \circ g$, avec $f : O_1 \to_m O_2$ et $g : O_2 \to_m O_3$ est monotone, de type $O_1 \to_m O_3$. Dans ce contexte, une *chaîne* totalement ordonnée d'un ensemble O est un élément de $\mathbb{N}_0 \to_m O$.

Un (ω-)CPO étend la structure d'ensemble ordonné avec un plus petit élément, noté $\bot$, et un opérateur lub permettant de calculer la plus petite borne supérieure d'une chaîne ordonnée.

```
Record cpo : Type := mk_cpo
{ tcpo :> ord;
  ⊥ : tcpo;
  lub : (natO -m> tcpo) → tcpo;
  Dbot : ∀x : tcpo, ⊥ ⪯ x;
  le_lub : ∀(f : natO -m> tcpo) (n : nat), f n ⪯ lub f;
  lub_le : ∀(f : natO -m> tcpo) (x : tcpo),
    (∀ n, f n ⪯ x) → lub f ⪯ x }.
```

Si D_1 et D_2 sont des CPO, une fonction monotone $F : D_1 \to_m D_2$ est *continue* au sens de Scott si pour toute chaîne h, $F(\mathrm{lub}\, h) \preceq \mathrm{lub}(F \circ h)$. On note $D_1 \to_c D_2$ l'ensemble des fonctions continues de D_1 dans D_2, formant lui aussi un CPO.

```
Definition continuous (D1 D2 : cpo) (f:D1 -m> D2) :=
  ∀ h : natO -m> D1, f (lub h) ≼ lub (f ∘ h).
Record fconti (D1 D2 : cpo): Type := mk_fconti
{  fcontit : D1 -m> D2;
   fcontinuous : continuous fcontit }.
```

Définir des fonctions continues de cette manière est pénible car il faut construire manuellement un enregistrement constitué de la fonction et d'une preuve de sa continuité. Combiner des fonctions continues via des applications partielles ou l'opérateur de composition $\circ$ reste difficile à faire directement et il faut souvent passer par le langage de tactiques de Coq, ce qui mène à des définitions illisibles. Il est alors d'usage de définir un objet par tactiques puis d'énoncer une *équation caractéristique* permettant de le spécifier et de le manipuler simplement. La bibliothèque définit plusieurs instances intéressantes de CPO et de fonctions avec leurs équations caractéristiques.

Produit Le produit binaire `D1*D2` de Coq possède une structure de CPO dont les principales fonctions de manipulation sont continues.

$$\begin{aligned}
\mathsf{pair} &: D_1 \to_c D_2 \to_c D_1 * D_2 \\
\mathsf{fst} &: D_1 * D_2 \to_c D_1 \\
\mathsf{snd} &: D_1 * D_2 \to_c D_2 \\
\mathsf{curry} &: (D_1 * D_2 \to_c D_3) \to_c D_1 \to_c D_2 \to_c D_3 \\
\mathsf{uncurry} &: (D_1 \to_c D_2 \to_c D_3) \to_c D_1 * D_2 \to_c D_3
\end{aligned}$$

Produit indexé La notion de produit est généralisée à un ensemble arbitraire I d'indices. À partir d'une famille $D : I \to \mathit{cpo}$ est défini le CPO produit Π_D, qui peut être vu comme une fonction dépendante $(i : I) \to D\ i$.

$$\begin{aligned}
\mathsf{proj} &: \forall i \in I, \Pi_D \to_c D\ i \\
\mathsf{mapi} &: (\forall i, D_1\ i \to_c D_2\ i) \to \Pi_{D_1} \to_c \Pi_{D_2}
\end{aligned}$$

Les équations caractéristiques spécifient bien que $\mathsf{proj}\ i\ p = p\ i$ et que $\mathsf{mapi}\ f\ p\ i = f\ (p\ i)$. Cela permet notamment de manipuler des environnements de valeurs. Le plongement profond induit par la structure du compilateur, nous le verrons, ne permet pas de tirer parti directement de la dépendance du produit indexé. Nous instancierons I par ident, l'ensemble des identifiants des variables d'un programme, et D par une fonction constante.

Produit fini Le produit fini $D^k, k \in \mathbb{N}$ a aussi une structure de CPO.

3.1.2 Points fixes

Toute l'infrastructure développée autour des ordres et de la continuité permet d'obtenir une formalisation constructive du théorème de Kleene. Soit $F : D \to_c D$ une fonction continue, et $\mathsf{iter}\ F$ la fonction monotone de type $\mathbb{N}_0 \to_m D$ définie par $\mathsf{iter}\ F\ 0 = \bot$ et $\mathsf{iter}\ F\ (n+1) = F(\mathsf{iter}\ F\ n)$. Alors on peut définir un combinateur de point fixe :

$$\mathsf{fixp} := \lambda F.\, \mathrm{lub}\,(\mathsf{iter}\ F) : (D \to_c D) \to_c D.$$

L'équation caractéristique découlant de cette définition assure la propriété souhaitée, à savoir $\mathsf{fixp}\ F \simeq F\ (\mathsf{fixp}\ F)$.

3.1.3 Type des flots

Le CPO des flots finis et infinis, support du modèle de Kahn, nécessite un encodage particulier en Coq. Rappelons d'abord le type des flots de la bibliothèque standard, utilisé dans la sémantique relationnelle de Vélus.

```
CoInductive Stream (A : Type) : Type :=
  Cons : A → Stream A → Stream A.
```

L'interprétation co-inductive de cette structure de donnée assure que tout objet de type Stream est bien un flot infini de valeurs de type A. Il pourrait être tentant d'introduire un constructeur supplémentaire pour autoriser la terminaison.

```
CoInductive Stream′ (A : Type) : Type :=
  | Nil : Stream′ A
  | Cons : A → Stream′ A → Stream′ A.
```

En résulte un type de donnée isomorphe aux *lazy lists* décrites au chapitre 13 du Coq'Art [2]. Si la plupart des opérations sur les flots infinis se transposent aux flots possiblement finis, ce n'est pas le cas du filtrage, composant essentiel d'un langage échantillonné comme Lustre.

```
CoFixpoint filter A (P : A → bool) (s : Stream′ A) : Stream′ A :=
  match s with
  | Nil ⇒ Nil
  | Cons a s ⇒ if P a then Cons a (filter A P s)
                else filter A P s
  end.
(* Unguarded recursive call in "filter A P s". [...] *)
```

[2] Bertot et Castéran, 2004

En effet, le mécanisme de garde syntaxique imposé par Coq n'autorise pas d'appel co-récursif sans d'abord l'appel à un constructeur. Si ce critère n'est pas satisfaisant en général, il est ici tout à fait justifié car une telle définition du filtrage n'est pas productive. L'idée dans la bibliothèque CPO, empruntée à des travaux sur la co-induction [3], est d'ajouter un constructeur ϵ signifiant intuitivement « pas disponible » et s'intercalant entre deux éléments du flot, notamment dans la branche `else` de la fonction de filtrage ci-dessus.

```
CoInductive StreamEps (A : Type) : Type :=
  | Eps : StreamEps A → StreamEps A
  | Cons : A → StreamEps A → StreamEps A.
```

Un flot de type Stream_ϵ est infini au sens de Coq mais représente les flots possiblement finis du modèle de Kahn. Nous utiliserons dans la suite la catégorisation suivante :

— on appelle **flot vide** le flot ϵ^ω constitué d'une infinité d'éléments ϵ ;

— un flot est **non vide** s'il contient au moins un constructeur `Cons` ;

— un flot est **fini** s'il termine par la séquence ϵ^ω ;

— un flot est **infini** s'il continent une infinité de constructeurs `Cons`.

Comme ces propriétés ne sont pas décidables, une fonction de flot ne peut s'appuyer dessus pour déterminer son résultat. Nous verrons dans la suite comment les fonctions de la bibliothèque gèrent les occurrences d'éléments ϵ afin de représenter des calculs sur les différentes catégories de flots.

Notation À des fins de lisibilité, nous adoptons dans le texte des notations légèrement plus souples que celles de Coq. Nous notons $a \cdot x$ pour le flot `Cons a x` et $\epsilon \cdot x$ pour le flot `Eps x`. Nous utilisons la même notation $\cdot$ pour les flots du modèle relationnel, sans que cela ne cause de véritable ambiguïté.

Flots en Isabelle Un encodage des flots possiblement finis est documenté dans la thèse d'Olaf Müller [4] et dans l'article associé [5]. Ces travaux s'appuient sur la logique HOLCF en Isabelle. Le type des flots y est défini comme une instance d'un domaine récursif, correspondant aux lazy lists :

$$\textbf{domain} \quad (\alpha)\mathsf{seq} = \mathsf{nil} \mid (\mathsf{HD} : \alpha)\#(\textbf{lazy}(\mathsf{TL} : (\alpha)\mathsf{seq})).$$

En HOLCF, le type des domaines récursifs est lui-même déclaré comme un CPO, ce qui amène à distinguer trois catégories de flots : les flots infinis, constitués d'une infinité de constructeurs #, les flots finis, se terminant par le constructeur nil et les flots *partiels*, finissant par l'élément $\bot$ du CPO. Les

[3] Capretta, 2005
[4] Müller, 1998, §6.2
[5] Müller *et al.*, 1999

fonctions sur les flots sont ensuite définies par disjonction sur ces trois cas. Comme la théorie des domaines sous-jacente est axiomatisée, c'est-à-dire que les bornes supérieures des chaînes sont supposées existantes, et non calculées explicitement, les fonctions de flots n'ont pas besoin d'être productives comme en Coq. Cela permet d'éviter l'utilisation d'éléments ϵ et de raisonner plus confortablement sur la forme des flots. En contrepartie, aucune extraction vers du code exécutable n'est envisageable.

3.1.4 CPO et fonctions de flots

L'ordre préfixe du modèle de Kahn est défini sur les flots de type Stream_ϵ par les deux règles co-inductives suivantes.

$$\frac{x \preceq y}{\epsilon \cdot x \preceq y} \qquad \frac{\exists n, y = \epsilon^n \cdot a \cdot z \qquad x \preceq z}{a \cdot x \preceq y}$$

De cette façon, l'ordre ne concerne bien que les valeurs des constructeurs `Cons`, l'existence des `Eps` étant totalement transparente. Nous avons par exemple la chaîne $\epsilon^\omega \preceq \epsilon \cdot \epsilon \cdot a \cdot \epsilon^\omega \preceq a \cdot \epsilon \cdot b \cdot \epsilon^\omega \preceq \epsilon \cdot a \cdot b \cdot \epsilon \cdot c \cdot \epsilon^\omega \preceq \ldots$

Le calcul d'une borne supérieure pour de telles séquences monotones de flots, défini dans la bibliothèque [6], est par conséquent très lent car il ne peut préjuger que de l'existence d'éléments `Cons` dans les flots successifs, et pas de leur position relative au sein d'un flot. En prenant comme plus petit élément $\bot = \epsilon^\omega$, l'ensemble ordonné des Stream_ϵ possède bien une structure de CPO.

Plusieurs fonctions sur les flots sont disponibles de façon primitive dans la bibliothèque, d'autres ont été implémentées par nos soins. Toutes sont continues et définies par des tactiques. Nous donnons en figure 3.1 leurs types et leurs équations caractéristiques. La relation d'équivalence $\simeq$ dérivée de l'ordre préfixe sur les flots est transparente vis-à-vis des ϵ, c'est-à-dire que pour tout flot x, nous avons $\epsilon \cdot x \simeq x$. Cela permet d'obtenir des équations à la Haskell/Isabelle très naturelles, notamment pour la fonction filter.

3.1.5 Outils de preuve

L'utilisation de la bibliothèque CPO ouvre la porte à de nombreux mécanismes de preuve permettant de raisonner à la fois sur les fonctions sémantiques et sur les programmes en eux-mêmes. Nous détaillons ici quelques outils utiles par la suite.

Réécriture Un CPO muni de sa relation d'équivalence $\simeq$ constitue un setoïde et bénéficie à ce titre des fonctionnalités de la réécriture généralisée dans Coq [7]. Grâce au mécanisme de *type classes*, toute fonction continue

[6] Paulin-Mohring, 2009, §3.2.3
[7] Sozeau, 2009

$\mathsf{rem} : \mathsf{Stream}_\epsilon\ A \to_c \mathsf{Stream}_\epsilon\ A$
$\mathsf{rem}\ (a \cdot x) \simeq x$
$\mathsf{rem}\ \bot \simeq \bot$

$\mathsf{app} : \mathsf{Stream}_\epsilon\ A \to_c \mathsf{Stream}_\epsilon\ A \to_c \mathsf{Stream}_\epsilon\ A$
$\mathsf{app}\ (a \cdot x)\ y \simeq a \cdot y$
$\mathsf{app}\ \bot\ y \simeq \bot$

$\mathsf{first} : \mathsf{Stream}_\epsilon\ A \to_c \mathsf{Stream}_\epsilon\ A$
$\mathsf{first}\ (a \cdot x) \simeq a \cdot \bot$
$\mathsf{first}\ \bot \simeq \bot$

$\mathsf{map} : (A \to B) \to \mathsf{Stream}_\epsilon\ A \to_c \mathsf{Stream}_\epsilon\ B$
$\mathsf{map}\ f\ (a \cdot x) \simeq (f\ a) \cdot \mathsf{map}\ f\ x$
$\mathsf{map}\ f\ \bot \simeq \bot$

$\mathsf{filter} : (A \to \mathtt{bool}) \to \mathsf{Stream}_\epsilon\ A \to_c \mathsf{Stream}_\epsilon\ A$
$\mathsf{filter}\ f\ (a \cdot x) \simeq a \cdot \mathsf{filter}\ f\ x \quad \text{si } f\ a = \mathsf{T}$
$\mathsf{filter}\ f\ (a \cdot x) \simeq \mathsf{filter}\ f\ x \quad \text{si } f\ a = \mathsf{F}$
$\mathsf{filter}\ f\ \bot \simeq \bot$

$\mathsf{zip} : (A \to B \to C) \to \mathsf{Stream}_\epsilon\ A \to_c \mathsf{Stream}_\epsilon\ B \to_c \mathsf{Stream}_\epsilon\ C$
$\mathsf{zip}\ f\ (a \cdot x)\ (b \cdot y) \simeq (f\ a\ b) \cdot \mathsf{zip}\ f\ x\ y$
$\mathsf{zip}\ f\ \bot\ y \simeq f\ x\ \bot \simeq \bot$

$\mathsf{take} : \mathtt{nat} \to \mathsf{Stream}_\epsilon\ A \to_c \mathsf{Stream}_\epsilon\ A$
$\mathsf{take}\ 0\ x \simeq \bot$
$\mathsf{take}\ (n+1)\ (a \cdot x) \simeq a \cdot (\mathsf{take}\ n\ x)$
$\mathsf{take}\ n\ \bot \simeq \bot$

FIGURE 3.1 – Fonctions primitives de manipulation des flots.

La nomenclature adoptée ici est celle de la bibliothèque CPO, elle-même calquée sur les définitions de l'article de Gilles Kahn. Ainsi, la fonction rem correspond au destructeur usuel des flots (tl/tail) tandis que la fonction app correspond à l'opérateur -> en Lustre.

sur un CPO définit automatiquement un morphisme compatible avec les relations $\preceq$ et $\simeq$, ce qui permet une manipulation des termes très fluide. Il devient par exemple très simple de substituer un flot par sa définition.

```
(* rem_cons : ∀ a x, rem (cons a x) ≃ x *)
x, y : Streamε A
a : A
H : x ≃ cons a y
============================
y ≃ rem x
< rewrite H, rem_cons; reflexivity.
```

La réécriture d'une inégalité dans le but est aussi possible, mais uniquement dans une position contravariante.

```
(* rem_app_le : ∀ x y, rem (app x y) ≼ y *)
x, y, z : Streamε A
H : x ≼ app y z
============================
rem x ≼ z.
< rewrite H; apply rem_app_le.
```

Co-induction La relation préfixe entre deux flots est définie par un prédicat co-inductif. Il est possible de la prouver directement, ce qui implique de raisonner avec la définition concrète des flots et en particulier avec les éléments ϵ. Pour faciliter l'interaction avec la tactique `cofix`, nous utilisons l'astuce de *coinduction loading*[8]. Pour prouver l'inégalité entre deux termes t_1 et t_2 de type Stream_ϵ, nous commençons par introduire deux nouveaux flots x et y et généraliser les variables $s_1, \ldots, s_n$ du contexte de t_1 et t_2.

$$\frac{H : \ \forall x\ y\ s_1 \ldots s_n, x \simeq t_1[s_1; \ldots; s_n] \to y \simeq t_2[s_1; \ldots; s_n] \to x \preceq y}{t_1[s_1; \ldots; s_n] \preceq t_2[s_1; \ldots; s_n]}$$

L'application de `cofix` sur ce nouveau but permet d'obtenir une hypothèse de co-induction H dont il est aisé pour Coq de vérifier la productivité des instances. L'idée est simplement d'appliquer les équivalences, réécritures et autres théorèmes aux arguments de H et non à sa conclusion. L'analyse de cas sur x génère ensuite deux sous-buts :

— $\epsilon \cdot x \simeq t_1[s_1; \ldots; s_n] \to y \simeq t_2[s_1; \ldots; s_n] \to \epsilon \cdot x \preceq y$;

— $a \cdot x \simeq t_1[s_1; \ldots; s_n] \to y \simeq t_2[s_1; \ldots; s_n] \to a \cdot x \preceq y$.

[8] Endrullis *et al.*, 2013, §3

Le premier se démontre immédiatement en appliquant le constructeur de l'inégalité et en réécrivant $\epsilon \cdot x \simeq x$ avant d'instancier H pour conclure. Le second nécessite un raisonnement sur t_1 et t_2 afin de déterminer la forme de y et de pouvoir instancier H. Au final, ce principe de preuve est en adéquation avec la représentation interne des flots : le cas ϵ a été introduit pour satisfaire les critères de productivité de Coq et ne nécessite qu'un traitement administratif dans les preuves ; seul le cas du flot non vide, véritablement intéressant, est laissé à l'utilisateur.

Équivalence de flots L'équivalence de la bibliothèque est définie par la double inégalité $x \simeq y := x \preceq y \wedge y \preceq x$. Il n'est donc pas possible d'utiliser directement la tactique `cofix` pour prouver l'équivalence de deux flots. Deux lemmes standards, remarquablement illustrés par Olaf Müller dans sa thèse[9], peuvent toutefois s'appliquer. Le premier, proposé dans la bibliothèque, correspond à un principe de bi-simulation nécessitant la synthèse d'une relation R compatible avec l'opérateur rem. Ici, seul le cas de flots non-vides est à démontrer, le cas ϵ est traité silencieusement.

$$\frac{\begin{array}{c}\forall x\ y\ z\ t, x \simeq z \to y \simeq t \to R\ x\ y \to R\ z\ t \\ \forall x\ y, R\ x\ y \to (x \neq \bot \vee y \neq \bot) \to \mathsf{first}\ x \simeq \mathsf{first}\ y \\ \forall x\ y, R\ x\ y \to (x \neq \bot \vee y \neq \bot) \to R\ (\mathsf{rem}\ x)\ (\mathsf{rem}\ y)\end{array}}{\forall x\ y, R\ x\ y \to x \simeq y}$$

Le second, utile essentiellement dans les cas combinatoires, se démontre par une simple récurrence sur les entiers naturels.

$$\frac{\forall n, \mathsf{take}\ n\ x \simeq \mathsf{take}\ n\ y}{x \simeq y}$$

Enfin, nous démontrons un principe de « préfixe infini », très simple à utiliser.

$$\frac{x \preceq y \qquad \mathsf{infinite}\ x}{x \simeq y} \qquad\qquad \frac{\exists a\ y,\ x \simeq a \cdot y \qquad \mathsf{infinite}\ (\mathsf{rem}\ x)}{\mathsf{infinite}\ x}$$

Induction et sûreté Le principe d'induction de Scott est implémenté dans la bibliothèque et s'applique à toute fonction continue $F : D \to_c D$.

$$\frac{\mathrm{admissible}\ P \qquad P\ \bot \qquad (\forall x, P\ x \to P\ (F\ x))}{P\ (\mathsf{fixp}\ F)}$$

Rappelons qu'une propriété P est admissible si elle est compatible avec l'opérateur de borne supérieure, c'est-à-dire si pour toute chaîne h d'éléments de D, $(\forall n, P(h\,n)) \to P(\mathrm{lub}\,h)$. C'est le cas de beaucoup de propriétés qui

[9] Müller, 1998, §6.4

nous intéressent dans le cadre de cette thèse, et notamment les propriétés de sûreté sur les flots, exprimées à l'aide du prédicat co-inductif suivant.

$$\mathsf{Forall}_\epsilon : (A \to \mathsf{Prop}) \to \mathsf{Stream}_\epsilon\ A \to \mathsf{Prop}$$

$$\frac{\mathsf{Forall}_\epsilon\ Q\ x}{\mathsf{Forall}_\epsilon\ Q\ (\epsilon \cdot x)} \qquad \frac{Q\ a \qquad \mathsf{Forall}_\epsilon\ Q\ x}{\mathsf{Forall}_\epsilon\ Q\ (a \cdot x)}$$

En revanche, l'induction continue ne permet pas de prouver des propriétés de vivacité, comme l'infinitude d'un flot ou d'autres propriétés relatives aux nombre d'éléments dans un flot.

3.1.6 Remarques et travaux connexes

Extraction Toutes les fonctions de la bibliothèque CPO sont constructives et définies sans axiomes, ce qui les rend éligibles à l'extraction vers du code OCaml, en vue de leur exécution. Dans la pratique, la hiérarchie des structures décrite en section 3.1.1 résulte en un code extrait particulièrement compliqué, notamment en raison des coercitions qui ne sont pas exploitées comme en Coq. De plus, le temps de calcul de l'algorithme du point fixe, quadratique, devient rédhibitoire lors de l'imbrication de plusieurs appels à fixp.

Autres encodages D'autres approches ont été proposées pour définir les fonctions de flots en Coq, notamment par Yves Bertot et Ekaterina Komendantskaya [10] ou encore Vlad Rusu et David Nowak [11]. Ces deux encodages garantissent la productivité des fonctions de flots au moyen d'un prédicat. Par exemple, la fonction filter P n'est productive que si l'utilisateur apporte la preuve qu'une infinité d'éléments du flot à filtrer satisfont effectivement le prédicat P. Une telle approche ne peut pas s'appliquer aux opérateurs des réseaux de Kahn, qui sont bloquants par nature.

Plus récemment, les *Interaction Trees* [12] ont été proposés comme un environnement permettant de définir des sémantiques dénotationnelles en Coq. Leur encodage des calculs infinis, potentiellement bloquants ou divergents, repose sur la même astuce que la bibliothèque CPO, en satisfaisant les critères de productivité de Coq à l'aide d'un constructeur spécial τ. La structure monadique permettant de représenter élégamment les calculs purement séquentiels ne semble toutefois pas appropriée au modèle des réseaux de Kahn, où les calculs s'effectuent à la fois dans l'instant (évaluation d'expressions) et dans le temps (composition parallèle d'équations et de nœuds).

[10] Bertot et Komendantskaya, 2008
[11] Rusu et Nowak, 2022, §3
[12] Xia *et al.*, 2019

3.2 Valeurs

3.2.1 Flots avec erreurs

Notre modèle dénotationnel associe des flots aux expressions et des fonctions de flots aux nœuds. Contrairement au modèle du compilateur, il est défini pour tous les programmes, y compris ceux ne possédant pas de témoin dans la sémantique relationnelle. Pour gérer tous les cas possibles, il faut notamment pouvoir représenter des exécutions possiblement erronées. Nous classifions les différentes erreurs en quatre catégories.

— Les *erreurs de données*, qui résultent d'une combinaison incorrecte de valeurs. Elles sont exclues par le système de types du compilateur.

— Les *erreurs de synchronisation*, qui résultent d'une combinaison incorrecte d'absence et de présence de valeurs. Elle sont exclues par le système d'horloges du compilateur.

— Les *erreurs à l'exécution (run-time errors)*, qui sont causées par un échec d'opération arithmétique ou logique. Ces erreurs ne peuvent pas toujours être détectées statiquement et nécessitent un traitement particulier.

— Les *erreurs de causalité*, qui entraînent un blocage du calcul. Dans Vélus, elles sont exclues par une analyse des dépendances entre les variables du programme.

Nous définissions le type sevalue des valeurs synchrones « possiblement erronées » en introduisant explicitement un constructeur pour chaque type d'erreur rencontrée dans l'exécution. À titre de comparaison, nous rappelons aussi le type svalue du modèle relationnel et le type des valeurs transportées par les flots, qui peuvent être des constantes énumérées ou scalaires.

$$\begin{aligned} \mathsf{sevalue} &:= v \mid \mathsf{abs} \mid \mathsf{err}_{\mathsf{ty}} \mid \mathsf{err}_{\mathsf{sync}} \mid \mathsf{err}_{\mathsf{rt}} \\ \mathsf{svalue} &:= v \mid \mathsf{abs} \\ v : \mathsf{value} &:= C \mid i : \texttt{int} \mid l : \texttt{long} \mid f : \texttt{float} \end{aligned}$$

Les erreurs de causalité sont de nature différente. Dans un modèle dénotationnel, une erreur de causalité bloque le calcul des itérations, ce qui aboutit à un point fixe finissant par $\bot$. Dans le modèle synchrone, une erreur de causalité résulte donc en un flot fini. Nous modélisons naturellement ce type d'erreur en utilisant comme domaine Stream_ϵ sevalue, celui des flots possiblement finis ou erronés, contre Stream svalue dans le modèle relationnel.

Dans le modèle de Kahn en revanche, un flot fini n'est pas toujours synonyme d'erreur de causalité. Il est d'ailleurs impossible de distinguer le point fixe d'une équation non causale comme `x = x` de celui d'une équation causale comme `x = y when false`.

Notations Nous écrirons simplement err lorsque la distinction entre $\mathsf{err}_{\mathsf{ty}}$, $\mathsf{err}_{\mathsf{sync}}$, ou $\mathsf{err}_{\mathsf{rt}}$ n'est pas nécessaire. Nous abrégerons aussi Stream_ϵ sevalue en Stream_ϵ lorsque le type des données véhiculées est rendu clair par le contexte.

3.2.2 Propriétés et conversions

Plusieurs aspects sont à prendre en compte lors de la conversion des flots entre les différents modèles.

Type des données Nous traduisons les valeurs des flots grâce à une simple analyse de cas projetant les valeurs erronées sur une valeur absente.

$$\begin{aligned}\texttt{val\_val} : \mathsf{Stream}_\epsilon\ \mathsf{sevalue} &\to \mathsf{Stream}_\epsilon\ \mathsf{svalue}\\ &:= \mathsf{map}\ (\lambda v \to v \mid \mathsf{abs} \mid \mathsf{err} \to \mathsf{abs})\end{aligned}$$

Flots infinis Nous pouvons convertir un flot dénotationnel en un flot de Coq à condition qu'il soit équipé d'une preuve d'infinitude.

```
Inductive is_cons : Streamε A → Prop :=
  | icEps : ∀x, is_cons x → is_cons (Eps x)
  | icCon : ∀a x, is_cons (Con a x).

CoInductive infinite (x: Streamε A) : Prop :=
  inf_intro : is_cons x → infinite (rem x) → infinite x.
```

Par sa formulation, le prédicat inductif `is_cons` contient l'information nécessaire à l'extraction de la valeur en tête d'un flot non vide. En itérant `uncons` à l'infini, nous obtenons la fonction de conversion `S_of_Se` désirée.

```
Lemma uncons : ∀(x: Streamε A),
  is_cons x → { a : A & { s : Streamε A | x ≃ cons a s }}.

CoFixpoint S_of_Se (x : Streamε A) (Inf : infinite x) : Stream A :=
    match Inf with
    | inf_intro Hc Infr ⇒
        Streams.Cons (projT1 (uncons Hc)) (S_of_Se (rem x) Infr)
    end.
```

La manipulation dans Coq d'une telle fonction est problématique car on ne peut pas réécrire directement sur des hypothèses dépendantes.

```
x, y : Streamε A
Inf : infinite x
H : x ≃ y
============================
Forallε (λ e ⇒ ...) (S_of_Se x Inf)

< Fail rewrite H.
(* Error: no constraint can apply on a dependent argument *)
```

Une façon de remédier à ce problème consiste à construire de nouvelles preuves d'infinitude à chaque étape de réécriture, en utilisant par exemple le lemme suivant.

```
Lemma S_of_Se_eq :
  ∀ (x y : Streamϵ A) (Hx : infinite x) (H : x ≃ y),
  ∃ Hy, S_of_Se x Hx ≡ S_of_Se y Hy.
```

La fonction de conversion inverse, en revanche, ne nécessite pas de preuve supplémentaire car les flots de Coq sont infinis par définition.

```
CoFixpoint Se_of_S (x : Stream A) : Streamϵ A :=
  match x with
  | Streams.Cons a x ⇒ cons a (Se_of_S x)
  end.
```

Flots sans erreurs Nous caractérisons l'absence d'erreurs dans les flots dénotationnels au moyen des prédicats suivants.

$$
\begin{aligned}
\mathsf{safe}_{\mathsf{ty}} \quad &:= \mathsf{Forall}_\epsilon\ (\lambda e.e \neq \mathsf{err}_{\mathsf{ty}}) \\
\mathsf{safe}_{\mathsf{sync}} \quad &:= \mathsf{Forall}_\epsilon\ (\lambda e.e \neq \mathsf{err}_{\mathsf{sync}}) \\
\mathsf{safe}_{\mathsf{rt}} \quad &:= \mathsf{Forall}_\epsilon\ (\lambda e.e \neq \mathsf{err}_{\mathsf{rt}}) \\
\mathsf{safe}\ x \quad &:= \mathsf{safe}_{\mathsf{ty}}\ x \wedge \mathsf{safe}_{\mathsf{sync}}\ x \wedge \mathsf{safe}_{\mathsf{rt}}\ x
\end{aligned}
$$

Nous dirons d'un flot satisfaisant le prédicat safe qu'il est *sans erreurs*.

Traduction En composant les fonctions `val_val` et `S_of_Se` décrites dans les paragraphes précédents, nous obtenons une traduction d'un flot du modèle synchrone dénotationnel en un flot du modèle synchrone relationnel. Si le flot est sans erreurs, alors la correspondance est complète et commute avec `Se_of_S`. Les preuves d'infinitude sont nécessaires et omniprésentes dans les parties du développement Coq traitant de la correspondance des deux modèles synchrones. Dans ce document, nous choisissons de passer ces éléments sous silence en adoptant la notation informelle suivante pour les flots infinis.

$$\lceil \cdot \rceil : \mathsf{Stream}_\epsilon\ \mathsf{sevalue} \to \mathsf{Stream}\ \mathsf{svalue}$$

Effacement des absences Pour passer aux flots du modèle de Kahn, nous implémentons la procédure d'effacement des absences par un simple filtrage, sans changer le type des données.

$$\lfloor \cdot \rfloor := \mathsf{filter}\ (\lambda e.e \neq \mathsf{abs}) : \mathsf{Stream}_\epsilon\ \mathsf{sevalue} \to \mathsf{Stream}_\epsilon\ \mathsf{sevalue}$$

3.3 Sémantique des opérateurs de flots

Dans le modèle dénotationnel de Lustre, chaque expression du langage est associée à un opérateur synchrone sur les flots échantillonnés. Nous décrivons dans cette section chacun de ces opérateurs, en rappelant d'abord leur définition existante dans le modèle relationnel du compilateur, puis en donnant une définition constructive dans notre modèle synchrone dénotationnel. Nous esquissons ensuite les principaux résultats de correction liés à la correspondance entre les deux modèles.

3.3.1 Constantes

L'opérateur des constantes est défini relativement au flot de l'horloge de base du nœud courant, qui spécifie l'emplacement des absences dans le temps. C'est la seule construction à être partagée à l'identique par les deux modèles. Elle ne produit jamais d'erreurs et le flot résultant est de même longueur que celui de l'horloge de base.

$$\begin{aligned}&\mathsf{const}\ c : \mathsf{Stream}_\epsilon\ \mathsf{bool} \to_c \mathsf{Stream}_\epsilon\ \mathsf{sevalue}\\ &\quad := \mathsf{map}\ (\lambda b.\mathsf{if}\ b\ \mathsf{then}\ c\ \mathsf{else}\ \mathsf{abs})\end{aligned}$$

Dans le modèle de Kahn en revanche, un flot constant ne dépend d'aucune horloge et produit la même valeur à l'infini.

$$\mathsf{const}\ c := c \cdot \mathsf{const}\ c$$

3.3.2 Opérations unaires et binaires

Une opération arithmétique ou logique unaire $\diamond$ s'applique aux valeurs d'un flot échantillonné en laissant passer les absences, comme le spécifient les deux règles du prédicat relationnel LIFT$^\diamond$. L'opérateur $\diamond$, défini dans la bibliothèque de CompCert, est susceptible d'échouer sur certaines entrées et de retourner la valeur None. Ce cas est banni de la relation par la seconde règle du prédicat. Après un examen minutieux des algorithmes de CompCert, nous avons identifié les cas problématiques, tous relatifs aux conversions entre flottants et entiers, et les avons répertoriés en section 5.2. Dans la fonction $\mathsf{lift}^\diamond$ du modèle dénotationnel, ce cas d'erreur résulte en $\mathsf{err}_{\mathsf{rt}}$.

$$\frac{\text{LIFT}^\diamond\ xs\ rs}{\text{LIFT}^\diamond\ (\mathsf{abs} \cdot xs)\ (\mathsf{abs} \cdot rs)}$$

$$\frac{\text{LIFT}^\diamond\ xs\ rs \qquad \diamond v = \mathsf{Some}\ r}{\text{LIFT}^\diamond\ (v \cdot xs)\ (r \cdot rs)}$$

$$\begin{aligned}&\mathsf{lift}^\diamond : \mathsf{Stream}_\epsilon \to_c \mathsf{Stream}_\epsilon\\ &\mathsf{lift}^\diamond := \mathsf{map}\,(\lambda\ \mathsf{abs} \to \mathsf{abs}\\ &\qquad\qquad \mid v \to (\mathsf{match}\ \diamond v\ \mathsf{with}\\ &\qquad\qquad\qquad \mid \mathsf{Some}\ r \to r\\ &\qquad\qquad\qquad \mid \mathsf{None} \to \mathsf{err}_{\mathsf{rt}})\\ &\qquad\qquad \mid \mathsf{err} \to \mathsf{err})\end{aligned}$$

Nous énonçons ensuite le principal résultat de correspondance entre ces deux formulations de l'opérateur.

Lemme 3.1. Pour tout flot $xs : \mathsf{Stream}_\epsilon$, si $\mathsf{lift}^\diamond\ xs$ est infini et sans erreurs, alors xs est infini et sans erreurs et $\textsc{lift}^\diamond\ \lceil xs \rceil\ \lceil \mathsf{lift}^\diamond\ xs \rceil$.

Lemme 3.2. Pour tout flot $xs : \mathsf{Stream}_\epsilon$, $\lfloor \mathsf{lift}^\diamond\ xs \rfloor \simeq \mathsf{lift}^\diamond\ \lfloor xs \rfloor$.

Le prédicat pour une opération binaire $\oplus$ nécessite en plus que les deux flots passés en paramètre soient synchronisés.

$$\frac{\textsc{lift}^\oplus\ xs\ ys\ rs}{\textsc{lift}^\oplus\ (\mathsf{abs} \cdot xs)\ (\mathsf{abs} \cdot ys)\ (\mathsf{abs} \cdot rs)} \qquad \frac{\textsc{lift}^\oplus\ xs\ ys\ rs \qquad v_1 \oplus v_2 = \mathsf{Some}\ r}{\textsc{lift}^\oplus\ (v_1 \cdot xs)\ (v_2 \cdot ys)\ (r \cdot rs)}$$

Dans la version dénotationnelle, une erreur de synchronisation résulte en $\mathsf{err}_{\mathsf{sync}}$ tandis qu'un échec d'opérateur résulte toujours en $\mathsf{err}_{\mathsf{rt}}$. Là encore, nous avons identifié toutes les sources d'échec possibles, dont la plus emblématique est la division entière par zéro.

$$\begin{aligned}
&\mathsf{lift}^\oplus : \mathsf{Stream}_\epsilon \to_c \mathsf{Stream}_\epsilon \to_c \mathsf{Stream}_\epsilon \\
&\mathsf{lift}^\oplus := \mathsf{zip}\,(\lambda\ \mathsf{abs}, \mathsf{abs} \to \mathsf{abs} \\
&\qquad\qquad \mid v_1, v_2 \to (\mathsf{match}\ v_1 \oplus v_2\ \mathsf{with} \\
&\qquad\qquad\qquad\qquad \mid \mathsf{Some}\ r \to r \\
&\qquad\qquad\qquad\qquad \mid \mathsf{None} \to \mathsf{err}_{\mathsf{rt}}) \\
&\qquad\qquad \mid \mathsf{err}, _ \mid _, \mathsf{err} \to \mathsf{err} \\
&\qquad\qquad \mid _, _ \to \mathsf{err}_{\mathsf{sync}})
\end{aligned}$$

Lemme 3.3. Pour tous flots $xs, ys : \mathsf{Stream}_\epsilon$, si $\mathsf{lift}^\oplus\ xs\ ys$ est infini et sans erreurs, alors xs et ys sont infinis et sans erreurs et $\textsc{lift}^\oplus\ \lceil xs \rceil\ \lceil ys \rceil\ \lceil \mathsf{lift}^\oplus\ xs\ ys \rceil$.

Lemme 3.4. Pour tous flots $xs, ys : \mathsf{Stream}_\epsilon$, si $\mathsf{lift}^\oplus\ xs\ ys$ est sans erreurs, alors $\lfloor \mathsf{lift}^\oplus\ xs\ ys \rfloor \simeq \mathsf{lift}^\oplus\ \lfloor xs \rfloor\ \lfloor ys \rfloor$.

3.3.3 Délai initialisé

L'opérateur `fby` prend deux entrées, un flot initial et un flot à décaler. La définition de Kahn, dans laquelle l'absence de valeurs n'est pas explicite, est simple : la tête du flot initial est ajoutée au flot à décaler, comme un préfixe. C'est précisément le comportement de la fonction app définie dans la bibliothèque CPO, cf. figure 3.1, et qui s'avère être particulièrement agréable à manipuler lors d'une vérification interactive. Dans une sémantique synchrone en revanche, les valeurs absentes des deux flots doivent rester *alignées*. C'est ce qu'imposent les règles du prédicat relationnel standard dans la littérature [13].

[13] Colaço et Pouzet, 2003, figure 2

$$\frac{\text{FBY}\ xs\ ys\ rs}{\text{FBY}\ (\mathsf{abs}\cdot xs)\ (\mathsf{abs}\cdot ys)\ (\mathsf{abs}\cdot rs)} \qquad \frac{\text{FBY}_1\ v_2\ xs\ ys\ rs}{\text{FBY}\ (v_1\cdot xs)\ (v_2\cdot ys)\ (v_1\cdot rs)}$$

$$\frac{\text{FBY}_1\ v\ xs\ ys\ rs}{\text{FBY}_1\ v\ (\mathsf{abs}\cdot xs)\ (\mathsf{abs}\cdot ys)\ (\mathsf{abs}\cdot rs)} \qquad \frac{\text{FBY}_1\ v_2\ xs\ ys\ rs}{\text{FBY}_1\ v\ (v_1\cdot xs)\ (v_2\cdot ys)\ (v\cdot rs)}$$

La relation FBY propage les absences jusqu'à ce qu'une valeur apparaisse sur les deux flots passés en argument. La valeur du flot initial est propagée et celle du flot à décaler est mémorisée dans la relation FBY_1, qui propage elle aussi les absences. Lorsque deux valeurs sont présentes sur les flots de FBY_1, la valeur v mémorisée est propagée, celle du flot initial est ignorée, car seule sa présence compte, et celle du flot à décaler est mémorisée.

La définition dénotationnelle que nous proposons est plus compliquée en raison de son rôle dans le calcul des points fixes. Considérons par exemple le compteur `x = 0 fby (x + 1)`. Une définition de `fby` qui lirait ses deux entrées avant de produire la sortie conduirait nécessairement à un point fixe égal à $\bot$. Il faut donc que l'opérateur lise le flot initial, produise une valeur et ensuite seulement, lise le flot à décaler. Nous concrétisons cette idée par la définition de quatre fonctions continues mutuellement récursives.

$$\begin{aligned}
\mathsf{fby}\ (\mathsf{abs}\cdot xs)\ ys &:= \mathsf{abs}\cdot \mathsf{fby}_\mathsf{A}\ xs\ ys\\
\mathsf{fby}\ (v\cdot xs)\ ys &:= v\cdot \mathsf{fby}'_1\ \mathsf{None}\ xs\ ys\\
\mathsf{fby}\ (\mathsf{err}\cdot xs)\ ys &:= \mathsf{err}\cdot \mathsf{map}\ (\lambda x.\ \mathsf{err})\ xs\\
\\
\mathsf{fby}_\mathsf{A}\ xs\ (\mathsf{abs}\cdot ys) &:= \mathsf{fby}\ xs\ ys\\
\mathsf{fby}_\mathsf{A}\ xs\ (\mathsf{err}\cdot ys) &:= \mathsf{map}\ (\lambda x.\ \mathsf{err})\ xs\\
\mathsf{fby}_\mathsf{A}\ xs\ (v\cdot ys) &:= \mathsf{map}\ (\lambda x.\ \mathsf{err}_\mathsf{sync})\ xs\\
\\
\mathsf{fby}'_1\ \mathsf{None}\ xs\ (v\cdot ys) &:= \mathsf{fby}_1\ v\ xs\ ys\\
\mathsf{fby}'_1\ (\mathsf{Some}\ v)\ xs\ (\mathsf{abs}\cdot ys) &:= \mathsf{fby}_1\ v\ xs\ ys\\
\mathsf{fby}'_1\ _\ xs\ (\mathsf{err}\cdot ys) &:= \mathsf{map}\ (\lambda x.\ \mathsf{err})\ xs\\
\mathsf{fby}'_1\ _\ xs\ (_\cdot ys) &:= \mathsf{map}\ (\lambda x.\ \mathsf{err}_\mathsf{sync})\ xs\\
\\
\mathsf{fby}_1\ v\ (\mathsf{abs}\cdot xs)\ ys &:= \mathsf{abs}\cdot \mathsf{fby}'_1\ (\mathsf{Some}\ v)\ xs\ ys\\
\mathsf{fby}_1\ v\ (v_x\cdot xs)\ ys &:= v\cdot \mathsf{fby}'_1\ \mathsf{None}\ xs\ ys\\
\mathsf{fby}_1\ v\ (\mathsf{err}\cdot xs)\ ys &:= \mathsf{err}\cdot \mathsf{map}\ (\lambda x.\ \mathsf{err})\ xs
\end{aligned}$$

Les fonctions fby et fby_1 lisent sur le flot initial et produisent une valeur, tandis que fby_A et fby'_1 lisent sur le flot à décaler. Lorsque fby lit une valeur absente sur le flot initial, elle produit une valeur absente avant d'invoquer fby_A, qui vérifie ensuite l'apparition d'une absence sur le flot à décaler, et sinon produit une erreur de synchronisation. Lorsque fby lit une valeur présente sur le flot initial, elle la propage et invoque fby'_1 pour lire et mémoriser

une valeur présente sur le flot à décaler. La fonction fby_1 applique le même principe avec une valeur mémorisée v. En produisant ainsi des sorties de façon spéculative, ces définitions assurent la productivité d'équations récursives satisfaisant le principe de causalité. En contrepartie, une éventuelle erreur sur le flot de droite ou une mauvaise synchronisation des deux flots est détectée avec un instant de retard. L'utilisation de map dans les cas d'erreurs est nécessaire pour conserver la longueur des flots. Nous verrons au chapitre 4 comment cette propriété est ensuite utilisée, notamment pour démontrer la productivité de la sémantique synchrone.

Lemme 3.5. Pour tous flots $xs, ys : \mathsf{Stream}_\epsilon$, si fby xs ys est infini et sans erreurs, alors xs et ys sont infinis et sans erreurs et FBY $\lceil xs \rceil$ $\lceil ys \rceil$ $\lceil \mathsf{fby}\ xs\ ys \rceil$.

Lemme 3.6. Pour tous flots $xs, ys : \mathsf{Stream}_\epsilon$, si fby xs ys est sans erreurs, alors $\lfloor \mathsf{fby}\ xs\ ys \rfloor \preceq \mathsf{app}\ \lfloor xs \rfloor\ \lfloor ys \rfloor$.

Notons qu'après effacement des absences, notre définition du délai correspond bien à celle de Kahn dans la mesure où elle en constitue un préfixe. En revanche, l'équivalence n'est pas vérifiée en raison de l'asymétrie intrinsèque du délai. Considérons par exemple l'instance suivante.

xs	abs	x_1	x_2	abs	x_3	abs	abs	abs $\cdots$
ys	abs	y_1	y_2	abs	y_3	abs	abs	abs $\cdots$
fby xs ys	abs	x_1	y_1	abs	y_2	abs	abs	abs $\cdots$

Le calcul du fby est bloqué par la lecture du flot initial xs tandis que l'opérateur de Kahn ne le lit pas. Nous avons donc :

$$\begin{aligned}\mathsf{app}\ \lfloor xs \rfloor\ \lfloor ys \rfloor &\simeq x_1 \cdot y_1 \cdot y_2 \cdot y_3 \cdot \bot \\ &\not\preceq x_1 \cdot y_1 \cdot y_2 \cdot \bot \simeq \lfloor \mathsf{fby}\ xs\ ys \rfloor.\end{aligned}$$

3.3.4 Échantillonnage

L'opérateur `when`, dont les règles apparaissent ci-dessous, est paramétré par une constante C d'un type énuméré et prend deux entrées, un flot de condition et un flot d'arguments à échantillonner. Comme précédemment les deux flots d'entrée doivent être synchronisés, sinon le prédicat relationnel ne tient pas. Lorsque la valeur présente du flot de condition correspond au paramètre C, la valeur en tête de l'argument est propagée, sinon une absence est produite. L'idée générale de cet opérateur est de filtrer le flot d'arguments en fonction du flot de condition tout en maintenant la synchronisation des deux flots.

$$\frac{\text{WHEN}_C\ cs\ xs\ rs}{\text{WHEN}_C\ (\mathsf{abs} \cdot cs)\ (\mathsf{abs} \cdot xs)\ (\mathsf{abs} \cdot rs)}$$

$$\frac{\text{WHEN}_C\ cs\ xs\ rs \qquad v_c = C}{\text{WHEN}_C\ (v_c \cdot cs)\ (v \cdot xs)\ (v \cdot rs)} \qquad \frac{\text{WHEN}_C\ cs\ xs\ rs \qquad v_c \neq C}{\text{WHEN}_C\ (v_c \cdot cs)\ (v \cdot xs)\ (\mathsf{abs} \cdot rs)}$$

En pratique, le test d'égalité entre la valeur v et la constante énumérée C n'est pas immédiat. En effet, chaque expression de la syntaxe a le type `exp`, qui n'est pas paramétré par le type des valeurs transportées dans les flots, cf. figure 2.4. Par conséquent, les flots du modèle relationnel ont tous le même type $\mathsf{Stream\ svalue}$ et la valeur v, de type value, peut ne pas désigner une constante d'énumération mais un entier ou un flottant. La situation est identique dans le modèle dénotationnel et nécessite donc une analyse de cas.

$$
\begin{aligned}
&\mathsf{when}_C : \mathsf{Stream}_\epsilon \to_c \mathsf{Stream}_\epsilon \to_c \mathsf{Stream}_\epsilon \\
&\mathsf{when}_C := \mathsf{zip}\,(\lambda\ \mathsf{abs}, \mathsf{abs} \to \mathsf{abs} \\
&\qquad\qquad \mid C, v \to v \\
&\qquad\qquad \mid C', v \to \mathsf{abs} \\
&\qquad\qquad \mid v_c, v \to \mathsf{err}_{\mathsf{ty}} \\
&\qquad\qquad \mid \mathsf{err}, \_ \mid \_, \mathsf{err} \to \mathsf{err} \\
&\qquad\qquad \mid \_, \_ \to \mathsf{err}_{\mathsf{sync}})
\end{aligned}
$$

Si la valeur du flot de condition est présente et désigne bien une constante énumérée (peu importe le type d'énumération), alors elle est comparée à C pour le filtrage. Si elle ne désigne pas une constante énumérée, alors il s'agit d'une erreur dans les types des valeurs véhiculées dans les flots et $\mathsf{err}_{\mathsf{ty}}$ est propagé. Dans le cas d'une mauvaise synchronisation, $\mathsf{err}_{\mathsf{sync}}$ est propagé.

Lemme 3.7. Pour tous flots cs, xs, si $\mathsf{when}_C\ cs\ xs$ est infini et sans erreurs, alors cs et xs sont infinis et sans erreurs et $\textsc{when}_C\ \lceil cs \rceil\ \lceil xs \rceil\ \lceil \mathsf{when}_C\ cs\ xs \rceil$.

Le `merge` effectue l'opération inverse du `when` : à partir de flots échantillonnés de façon complémentaire, il reconstruit le flot des valeurs fusionnées. L'opération porte ici sur un type énuméré de constructeurs $C_1, \ldots, C_n$. Lorsque la valeur du flot de condition est présente et correspond à l'un des constructeurs C_i, alors toutes les valeurs des flots des n arguments doivent être absentes, sauf la i-ème, qui doit être présente et propagée.

$$
\frac{\textsc{merge}\ cs\ [xs_1, \ldots, xs_n]\ rs}{\textsc{merge}\ (\mathsf{abs} \cdot cs)\ [\mathsf{abs} \cdot xs_1; \ldots; \mathsf{abs} \cdot xs_n]\ (\mathsf{abs} \cdot rs)}
$$

$$
\frac{\textsc{merge}\ cs\ [xs_1, \ldots, xs_i, \ldots, xs_n]\ rs \qquad v_c = C_i}{\textsc{merge}\ (v_c \cdot cs)\ [\mathsf{abs} \cdot xs_1; \ldots; v \cdot xs_i; \ldots; \mathsf{abs} \cdot xs_n]\ (v \cdot rs)}
$$

Les deux règles relationnelles ci-dessus sont définies dans Coq en termes de prédicats `Forall` et `Exists` sur la liste des flots $[xs_1; \ldots; xs_n]$. Les listes standard de Coq ne possèdent pas de structure de CPO intéressante car leur taille peut varier, ce qui empêche d'appliquer un opérateur de comparaison point à point sur les éléments de deux listes. L'opérateur dénotationnel opère plutôt sur des vecteurs, ou produits finis, de taille fixe.

$$
\mathsf{merge} : \mathsf{Stream}_\epsilon \to_c \mathsf{Stream}_\epsilon^n \to_c \mathsf{Stream}_\epsilon
$$

Le calcul de fusion doit à la fois produire une valeur et vérifier la bonne synchronisation des flots du vecteur. Prenons par exemple l'instance suivante.

cs	C_1	abs	C_3	C_2	C_1	...
xs_1	v_1	abs	abs	abs	v_1	...
xs_2	abs	abs	abs	v_2	abs	...
xs_3	abs	abs	v_3	abs	v_3	...
merge cs (xs_1, xs_2, xs_3)	v_1	abs	v_3	v_2	$\mathsf{err}_{\mathsf{sync}}$	...

L'opérateur merge traite ses arguments colonne par colonne, en commençant par examiner la valeur de cs. Si elle est absente, il parcourt les n flots du vecteur à la recherche d'absences et produit $\mathsf{err}_{\mathsf{sync}}$ si la valeur d'un des flots est présente. Si la valeur de cs est présente, il commence par vérifier qu'elle correspond à l'un des constructeurs C_i et produit $\mathsf{err}_{\mathsf{ty}}$ sinon. Il parcourt ensuite les n flots du vecteur à la recherche d'absences, sauf sur le i-ème flot, où il attend une valeur présente. Il produit $\mathsf{err}_{\mathsf{sync}}$ si l'une des conditions échoue. Nous implémentons ce comportement à l'aide de l'accumulateur foldi sur les vecteurs qui étend fold en donnant accès à l'indice de l'élément en cours d'accumulation.

$$\mathsf{foldi} : (\mathbb{N} \to B \to_c A \to_c A) \to_c A \to_c B^n \to_c A$$

Nous utilisons également zip_3, l'extension naturelle de zip à trois flots.

$$\begin{aligned}
\mathsf{merge}\ cs\ xss &:= \mathsf{foldi}\ (\lambda i.\mathsf{zip}_3\ (\mathsf{fmerge}\ i)\ cs)\ (\mathsf{start\text{-}acc}\ cs)\ xss \\
\mathsf{start\text{-}acc} &:= \mathsf{map}\ (\lambda\ \mathsf{abs} \to \mathsf{abs} \mid v \to \mathsf{err}_{\mathsf{ty}} \mid \mathsf{err} \to \mathsf{err}) \\
\mathsf{fmerge}\ i\ c\ x\ a &:= \mathsf{match}\ (c \stackrel{?}{=} C_i), x, a\ \mathsf{with} \\
&\quad \mid \mathsf{abs}, \mathsf{abs}, \mathsf{abs} \to \mathsf{abs} \\
&\quad \mid \mathsf{T}, v, \mathsf{err}_{\mathsf{ty}} \to v \\
&\quad \mid \mathsf{F}, \mathsf{abs}, a \to a \\
&\quad \mid \mathsf{err}_{\mathsf{ty}}, _, _ \to \mathsf{err}_{\mathsf{ty}} \\
&\quad \mid _, _, _ \to \mathsf{err}_{\mathsf{sync}}
\end{aligned}$$

À chaque instant, la fonction start-acc initialise un accumulateur pour foldi en fonction de la valeur du flot de condition. Lorsque la condition est présente, l'accumulateur s'initialise en $\mathsf{err}_{\mathsf{ty}}$, dans l'espoir d'être remplacé par une valeur présente rencontrée sur la branche correspondant à la condition (2^{e} cas du filtrage dans fmerge). Les valeurs des autres branches, supposément absentes, relèvent du 3^{e} cas du filtrage. Si la condition est absente, l'itérateur s'initialise en abs et filtre toutes les branches dans l'attente d'éléments absents.

Lemme 3.8. Pour tous flots cs : Stream_ϵ, xss : $\mathsf{Stream}_\epsilon^n$, si merge cs xss est infini et sans erreurs, alors cs et les xss sont infinis et sans erreurs et MERGE $\lceil cs \rceil$ $\lceil xss \rceil$ $\lceil \mathsf{merge}\ cs\ xss \rceil$.

Contrairement au fby, l'opérateur merge que nous avons défini ici en utilisant zip_3 lit une valeur sur chacune de ses entrées avant de produire une sortie. Ce comportement peut conduire à un blocage lorsque merge est appelé récursivement, comme nous le verrons en section 3.4.2. Dans Vélus cependant, l'analyse de causalité exclut d'office ce cas de figure.

3.3.5 Conditionnelle

L'opérateur de conditionnelle `case` effectue une analyse de cas sur les valeurs d'un type énuméré. La notation booléenne `if then else`, que nous avons adoptée dans les différents exemples de programmes Lustre, s'applique aux types à deux constructeurs. La sémantique de la conditionnelle ressemble à celle du `merge` mais impose la même synchronisation à tous ses arguments, « jetant » ainsi les valeurs des flots non sélectionnés.

$$\frac{\text{CASE } cs\ [xs_1, \ldots, xs_n]\ rs}{\text{CASE } (\mathsf{abs} \cdot cs)\ [\mathsf{abs} \cdot xs_1; \ldots; \mathsf{abs} \cdot xs_n]\ (\mathsf{abs} \cdot rs)}$$

$$\frac{\text{CASE } cs\ [xs_1, \ldots, xs_i, \ldots, xs_n]\ rs \qquad v_c = C_i}{\text{CASE } (v_c \cdot cs)\ [x_1 \cdot xs_1; \ldots; v \cdot xs_i; \ldots; x_n \cdot xs_n]\ (v \cdot rs)}$$

Le compilateur offre également la possibilité de définir un cas par défaut, évitant ainsi d'avoir à spécifier toutes les branches d'un type énuméré.

```
x = case e of (C1 => e1) (C2 => e2) (_ => d);
```

Le prédicat relationnel intègre cette fonctionnalité en prenant comme argument supplémentaire le flot ds de la branche par défaut.

$$\frac{\text{CASE}'\ cs\ [xs_1, \ldots, xs_m]\ ds\ rs}{\text{CASE}'\ (\mathsf{abs} \cdot cs)\ [\mathsf{abs} \cdot xs_1; \ldots; \mathsf{abs} \cdot xs_m]\ (\mathsf{abs} \cdot ds)\ (\mathsf{abs} \cdot rs)}$$

$$\frac{\text{CASE}'\ cs\ [xs_1, \ldots, xs_i, \ldots, xs_m]\ ds\ rs \qquad v_c = C_i}{\text{CASE}'\ (v_c \cdot cs)\ [v_1 \cdot xs_1; \ldots; v_i \cdot xs_i; \ldots; v_m \cdot xs_m]\ (v_d \cdot ds)\ (v_i \cdot rs)}$$

$$\frac{\text{CASE}'\ cs\ [xs_1, \ldots, xs_i, \ldots, xs_m]\ ds\ rs \qquad v_c = C_{i \neq 1,\ldots,m}}{\text{CASE}'\ (v_c \cdot cs)\ [v_1 \cdot xs_1; \ldots; v_m \cdot xs_m]\ (v_d \cdot ds)\ (v_d \cdot rs)}$$

L'implémentation dans la sémantique dénotationnelle s'appuie sur les mêmes mécanismes que merge, en initialisant un accumulateur selon la valeur du flot de condition. Dans le cas de case', l'accumulateur est initialisé avec la valeur, si elle existe, du flot par défaut.

$$
\begin{aligned}
\mathsf{case}\ cs\ xss &:= \mathsf{foldi}\ (\lambda i.\mathsf{zip}_3\ (\mathsf{fcase}\ i)\ cs)\ (\mathsf{start\text{-}acc}\ cs)\ xss \\
\mathsf{fcase}\ i\ c\ x\ a &:= \mathsf{match}\ (c \stackrel{?}{=} C_i), x, a\ \mathsf{with} \\
&\quad \mid \mathsf{abs}, \mathsf{abs}, \mathsf{abs} \to \mathsf{abs} \\
&\quad \mid \_, \_, \mathsf{abs} \to \mathsf{err}_{\mathsf{sync}} \\
&\quad \mid \mathsf{T}, v, \mathsf{err}_{\mathsf{ty}} \to v \\
&\quad \mid \mathsf{T}, v, v' \to v \\
&\quad \mid \mathsf{F}, v, a \to a \\
&\quad \mid \_, \_, \_ \to \mathsf{err}_{\mathsf{sync}} \\
\mathsf{start\text{-}acc}' &:= \mathsf{zip}\ (\lambda\ \mathsf{abs}, \mathsf{abs} \to \mathsf{abs} \mid c, d \to d \mid \_, \_ \to \mathsf{err}_{\mathsf{sync}}) \\
\mathsf{case}'\ cs\ xss\ ds &:= \mathsf{foldi}\ (\lambda i.\mathsf{zip}_3\ (\mathsf{fcase}\ i)\ cs)\ (\mathsf{start\text{-}acc}'\ cs\ ds)\ xss
\end{aligned}
$$

Le filtrage effectué par la fonction fcase diffère de celui de fmerge car il s'attend à des valeurs synchronisées sur toutes les branches. Le 3e cas du filtrage, lorsque l'accumulateur est $\mathsf{err}_{\mathsf{ty}}$, survient en l'absence de valeur par défaut tandis que le 4e permet de remplacer la valeur par défaut de l'accumulateur par la valeur de la branche désignée par la condition.

Lemme 3.9. Pour tous flots $cs : \mathsf{Stream}_\epsilon$, $xss : \mathsf{Stream}_\epsilon^n$, si case $cs\ xss$ est infini et sans erreurs, alors cs et les xss sont infinis et sans erreurs et CASE $\lceil cs \rceil\ \lceil xss \rceil\ \lceil \mathsf{case}\ cs\ xss \rceil$.

Lemme 3.10. Pour tous flots $cs, ds : \mathsf{Stream}_\epsilon$, $xss : \mathsf{Stream}_\epsilon^m$, si case′ $cs\ ds\ xss$ est infini et sans erreurs, alors cs, ds et les xss sont infinis et sans erreurs et CASE′ $\lceil cs \rceil\ \lceil xss \rceil\ \lceil ds \rceil\ \lceil \mathsf{case}'\ cs\ ds\ xss \rceil$.

3.4 Réinitialisation modulaire

La réinitialisation de nœuds en Lustre est beaucoup plus récente que les opérateurs synchrones classiques, et a aussi été beaucoup moins étudiée. Pour illustrer son intérêt, considérons l'exemple suivant.

```
node countdown (n : int; r : bool)
returns (t : int)
let
  t = if r then n else (n fby (t - 1));
tel
(* ... *)
x = countdown (n, edge);
```

Ici la variable x instancie le nœud `countdown`, qui reprend la valeur n dès qu'un signal est activé sur son deuxième argument. Une alternative équivalente à ce programme utilisant la réinitialisation serait la suivante.

```
node countdown (n : int)
returns (t : int)
let
  t = n fby (t - 1);
tel
(* ... *)
x = (restart countdown every edge) (n);
```

Le deuxième argument r est désormais inutile car la réinitialisation est effectuée au niveau de l'appel du nœud. L'opérateur de réinitialisation s'applique à l'instance du nœud, qui inclut les variables et toutes les éventuelles instances imbriquées. Cette approche est donc tout à fait modulaire et ne nécessite pas de modifier manuellement les variables internes du nœud à réinitialiser. Pour ce qui est de la compilation, l'opération de réinitialisation correspond grosso modo à un appel à la fonction *reset* suivi d'un appel à la fonction *step* (cf. section 2.1.3).

Cette construction constitue un élément central dans la définition des automates hiérarchiques de Scade 6 et Lucid Synchrone [14], où les équations d'un état peuvent être réinitialisées lors d'une transition. La réinitialisation a d'abord été formalisée sous forme de réseau récursif [15] puis a été introduite dans Vélus dans une forme plus générale [16] avant d'être utilisée pour la compilation vérifiée des automates [17]. Sa définition en termes de réseaux de Kahn nécessite toutefois d'être clarifiée avant de pouvoir être encodée dans un modèle dénotationnel, ce que nous proposons de faire dans cette section.

Nous commençons par expliquer l'encodage de la réinitialisation modulaire dans Vélus, qui repose sur un prédicat de masquage des environnements, puis nous en définissons deux versions calculatoires, chacune avec ses propres

[14] Colaço *et al.*, 2005
[15] Caspi, 1994; Hamon et Pouzet, 2000
[16] Bourke *et al.*, 2020
[17] Bourke *et al.*, 2023b

contraintes de synchronisation. Nous démontrons ensuite que lorsque ces contraintes coïncident, les trois formulations sont équivalentes.

3.4.1 Masques : réinitialisation dans Vélus

La sémantique relationnelle de Vélus est modulaire dans le sens où un nœud du langage est considéré comme un objet n'exposant que ses flots d'entrée et de sortie. Ainsi, le prédicat $G \vdash f(xs) \Downarrow ys$, spécifiant que le nœud f du programme G met en relation les entrées xs et les sorties ys, ne dépend pas d'un environnement global de variables et peut s'instancier arbitrairement. Ce principe est exploité pour modéliser le comportement d'un nœud réinitialisé comme une suite d'instances disjointes de ce même nœud, toutes séparées par un signal de réinitialisation. L'opérateur $\mathsf{mask}^i\ rs\ xs$ utilisé à cet effet permet d'isoler, via un masque d'absences, les valeurs du flot xs entre le i et le $(i+1)$-ème signal de réinitialisation reçu sur le flot rs. Le chronogramme ci-dessous illustre le principe de fonctionnement du masquage sur une instance réinitialisée f de `countdown`.

xs	3	3	3	3	abs	3	3	3	3	...
rs	F	F	T	F	F	F	T	abs	F	...
$\mathsf{mask}^0\ rs\ xs$	3	3	abs	abs	abs	abs	abs	abs	abs	...
$f(\mathsf{mask}^0\ rs\ xs)$	3	2	abs	abs	abs	abs	abs	abs	abs	...
$\mathsf{mask}^1\ rs\ xs$	abs	abs	3	3	abs	3	abs	abs	abs	...
$f(\mathsf{mask}^1\ rs\ xs)$	abs	abs	3	2	abs	1	abs	abs	abs	...
$\mathsf{mask}^2\ rs\ xs$	abs	abs	abs	abs	abs	abs	3	3	3	...
$f(\mathsf{mask}^2\ rs\ xs)$	abs	abs	abs	abs	abs	abs	3	2	1	...
ys	3	2	3	2	abs	1	3	2	1	...

La sémantique d'un nœud réinitialisé est obtenue en combinant toutes les pseudo-instances définies par les masquages successifs :

$$\forall k, G \vdash f(\mathsf{mask}^k\ rs\ xs) \Downarrow \mathsf{mask}^k\ rs\ ys.$$

La fonction mask est quant à elle définie co-inductivement sur les deux flots passés en argument.

$$\begin{aligned}
&\mathsf{mask}^k\ rs\ xs && := \mathsf{mask}^k_0\ rs\ xs \\
&\mathsf{mask}^k_{k'}\ (\mathsf{abs} \cdot rs)\ (x \cdot xs) && := (\mathsf{if}\ k' = k\ \mathsf{then}\ x\ \mathsf{else}\ \mathsf{abs}) \cdot \mathsf{mask}^k_{k'}\ rs\ xs \\
&\mathsf{mask}^k_{k'}\ (\mathsf{F} \cdot rs)\ (x \cdot xs) && := (\mathsf{if}\ k' = k\ \mathsf{then}\ x\ \mathsf{else}\ \mathsf{abs}) \cdot \mathsf{mask}^k_{k'}\ rs\ xs \\
&\mathsf{mask}^k_{k'}\ (\mathsf{T} \cdot rs)\ (x \cdot xs) && := (\mathsf{if}\ k' + 1 = k\ \mathsf{then}\ x\ \mathsf{else}\ \mathsf{abs}) \cdot \mathsf{mask}^k_{k'+1}\ rs\ xs
\end{aligned}$$

Cette fonction filtre le flot xs en propageant ses valeurs, commençant à la k-ième valeur vraie lue sur rs et jusqu'à la prochaine valeur vraie, et en propageant sinon des absences.

3.4.2 Réinitialisation dénotationnelle

Un opérateur dénotationnel de réinitialisation peut être défini constructivement en utilisant des fonctions de flots primitives et des instances de nœuds récursives [18], comme en Lucid Synchrone :

$$\begin{aligned}\mathsf{reset}_f\ rs\ xs := \mathsf{let}\ cs = \mathsf{true\text{-}until}\ rs\ \mathsf{in}\\ \mathsf{merge}\ cs\ (f\ (\mathsf{when}_\mathsf{T}\ cs\ xs))\\ (\mathsf{reset}_f\ (\mathsf{when}_\mathsf{F}\ cs\ rs)\ (\mathsf{when}_\mathsf{F}\ cs\ xs)).\end{aligned}$$

Grâce à une fonction auxiliaire true-until, le flot local cs est vrai jusqu'à ce qu'une première valeur vraie soit lue sur rs, auquel cas il devient faux immédiatement et pour toujours. Le merge retourne donc les valeurs d'une instance initiale de f jusqu'à ce que cs devienne faux, puis il retourne récursivement les valeurs de la prochaine instance.

En général, les programmes récursifs ne peuvent pas être compilés pour être exécutés en temps et en mémoire bornée. Cependant, comme le remarquent à juste titre les auteurs de cette définition, une fois la seconde branche du merge empruntée, la première n'a plus d'utilité car cs reste faux pour toujours. Cela s'apparente donc à une forme de récursion terminale, tout à fait compilable pour une exécution bornée. Bien que Vélus ne propose pas encore de telles fonctionnalités, nous pouvons exprimer cette construction récursive à l'aide du combinateur de point fixe de la bibliothèque CPO. De cette façon, nous proposons une définition mathématique, indépendante des problématiques d'utilisation des ressources.

Deux problèmes apparaissent lors de la définition de reset_f. Le premier se manifeste lorsqu'un signal de réinitialisation est reçu sur rs dès le premier instant de l'exécution, ce qui ne doit pas causer de récurrence instantanée dans la seconde branche du merge. La définition originale de true-until gère effectivement ce cas en produisant toujours vrai initialement, ignorant ainsi le signal de réinitialisation au premier instant.

$$\mathsf{true\text{-}until}\ rs = (\mathsf{fby}\ \mathsf{T}^\omega\ \mathsf{F}^\omega) \vee (\neg\ rs \wedge\ \mathsf{fby}\ \mathsf{T}^\omega\ (\mathsf{true\text{-}until}\ rs))\ [19]$$

Le second problème que nous avons détecté, concerne l'opérateur merge en lui-même. Sa définition dans le modèle synchrone nécessite qu'il inspecte chacune de ses branches afin de vérifier leur bonne synchronisation, cf. figure 2.7. Lorsque la condition est vraie, la valeur de la première branche doit être présente et la valeur de la seconde absente, et réciproquement lorsque la condition est fausse. Dans le reset défini ci-dessus, cela revient à forcer une récurrence instantanée à chaque instant de l'exécution et aboutit en un point fixe égal à $\bot$. Une façon d'éliminer ce problème est de donner, comme en figure 3.2, une autre interprétation à l'opérateur de fusion, à la manière de ce que nous avons proposé pour le délai initialisé.

[18] Caspi, 1994; Hamon et Pouzet, 2000

[19] Nous notons ici $\wedge$, $\vee$, et $\neg$ pour $\mathsf{zip}_\wedge$, $\mathsf{zip}_\vee$ et $\mathsf{map}_\neg$, respectivement.

$$
\begin{array}{ll}
\mathsf{expecta}\ (\mathsf{abs} \cdot s) & \simeq s \\
\mathsf{expecta}\ (v \cdot s) & \simeq \mathsf{map}\ (\lambda\_.\mathsf{err}_{\mathsf{sync}})\ s \\
\mathsf{expecta}\ (\mathsf{err} \cdot s) & \simeq \mathsf{map}\ (\lambda\_.\mathsf{err})\ s \\
\mathsf{merge}\ (\mathsf{abs} \cdot s_c)\ s_1\ s_2 & \simeq \mathsf{abs} \cdot (\mathsf{merge}\ s_c\ (\mathsf{expecta}\ s_1)\ (\mathsf{expecta}\ s_2)) \\
\mathsf{merge}\ (\mathsf{T} \cdot s_c)\ s_1\ s_2 & \simeq \mathsf{app}\ s_1\ (\mathsf{merge}\ s_c\ (\mathsf{rem}\ s_1)\ (\mathsf{expecta}\ s_2)) \\
\mathsf{merge}\ (\mathsf{F} \cdot s_c)\ s_1\ s_2 & \simeq \mathsf{app}\ s_2\ (\mathsf{merge}\ s_c\ (\mathsf{expecta}\ s_1)\ (\mathsf{rem}\ s_2)) \\
\mathsf{merge}\ (\mathsf{err} \cdot s_c)\ s_1\ s_2 & \simeq \mathsf{map}\ (\lambda\_.\mathsf{err})\ (\mathsf{err} \cdot s_c)
\end{array}
$$

FIGURE 3.2 – Un opérateur merge optimiste

Avec cette nouvelle définition, l'opérateur ne se base que sur la valeur de la condition pour sélectionner, de façon spéculative, une valeur à produire. En particulier, si une absence est attendue sur l'une des branches alors l'opérateur ne le vérifie qu'après avoir propagé la valeur de l'autre branche. Cette définition, qui permet de débloquer la fonction reset, ne permet cependant pas de détecter toutes les erreurs de synchronisation. Si le flot de condition est toujours absent par exemple, alors des absences sont propagées, quels que soient les flots des deux branches. Détecter de telles erreurs nécessiterait de définir d'autres fonctions auxiliaires, peu intéressantes dans notre cas.

Notons que dans le modèle synchrone à types dépendants, mentionné en section 2.2.2, la vérification de la bonne synchronisation des flots est effectuée paresseusement par typage, et n'a donc pas besoin d'être rendue explicite dans les opérateurs synchrones. Les auteurs encodent d'ailleurs l'opérateur de réinitialisation exactement comme dans la sémantique de Lucid Synchrone [20].

L'opérateur merge ainsi défini satisfait bien la propriété souhaitée lorsque le flot de condition est faux pour toujours.

Lemme 3.11. Pour tous flots s_f et s infinis et sans erreurs, si $s_f \simeq (\mathsf{abs} \mid \mathsf{F})^\omega$ et que les absences de s_f et s sont alignées,

$$\mathsf{merge}\ s_f\ \mathsf{abs}^\omega\ s \simeq s.$$

Cependant, le maintien à l'infini d'une synchronisation entre les deux branches s_f et s est difficile à établir en général, et surtout lorsque le merge est appelé récursivement. Nous démontrons donc une propriété plus faible mais applicable dans le cas du reset.

Lemme 3.12. Pour tous flots s_f, s_c, s_1, s_2 infinis et sans erreurs, si $s_f \simeq (\mathsf{abs} \mid \mathsf{F})^\omega$ et que les absences de s_f et s_c sont alignées,

$$\mathsf{merge}\ s_f\ \mathsf{abs}^\omega\ (\mathsf{merge}\ s_c\ s_1\ s_2) \simeq \mathsf{merge}\ s_c\ s_1\ s_2.$$

[20] Boulmé et Hamon, 2001b, §4.2.1

3.4.3 Conditions sur le calcul d'horloge

La définition de reset_f énoncée précédemment ne fait appel qu'à des fonctions Lustre comme fby, merge et when. Elle constitue donc un programme valide dans la sémantique de Kahn, qui gère la récursivité de façon primitive. Après effacement des absences, les opérateurs d'échantillonnage retrouvent des propriétés qui ne nécessitent pas de synchronisation des flots. En particulier, lorsque le flot local $cs = \mathsf{true\text{-}until}\ rs$ devient faux pour toujours, suite à la réception d'un signal sur rs, le raisonnement suivant est valide :

$$
\begin{aligned}
&\mathsf{merge}\ cs\ (f\ (\mathsf{when_T}\ cs\ xs))\ (\mathsf{reset}_f\ (\mathsf{when_F}\ cs\ rs)\ (\mathsf{when_F}\ cs\ xs)) \\
&\simeq \mathsf{merge}\ \mathsf{F}^\omega\ (f\ (\mathsf{when_T}\ \mathsf{F}^\omega\ xs))\ (\mathsf{reset}_f\ (\mathsf{when_F}\ \mathsf{F}^\omega\ rs)\ (\mathsf{when_F}\ \mathsf{F}^\omega\ xs)) \\
&\simeq \mathsf{reset}_f\ (\mathsf{when_F}\ \mathsf{F}^\omega\ rs)\ (\mathsf{when_F}\ \mathsf{F}^\omega\ xs) \qquad (1) \\
&\simeq \mathsf{reset}_f\ rs\ xs \qquad (2)
\end{aligned}
$$

car dans la sémantique de Kahn, les contraintes d'alignement sont levées :

(1) $\forall s_1\ s_2$, $\mathsf{merge}\ \mathsf{F}^\omega\ s_1\ s_2 \simeq s_2$;

(2) $\forall s$, $\mathsf{when_F}\ \mathsf{F}^\omega\ s \simeq s$.

Cette définition ne peut toutefois pas s'appliquer directement à notre modèle car dans Vélus, aucune synchronisation n'est imposée entre le flot de réinitialisation rs et le flot d'argument xs. Cela s'observe notamment dans le chronogramme donné en exemple précédemment, et plus généralement dans la définition de mask, qui autorise une valeur absente sur rs et présente sur xs, et vice versa. L'opérateur de masquage est donc *sensible aux absences*, ce qui va à l'encontre des principes du modèle de Kahn. Nous l'observons, par exemple, en considérant deux instances réinitialisées du nœud `countdown`.

xs_1	3	3	3	3	3	...
rs_1	F	F	abs	T	F	...
ys_1	3	2	1	3	2	...

xs_2	3	3	3	3	3	...
rs_2	F	F	T	abs	F	...
ys_2	3	2	3	2	1	...

Dans Vélus, l'absence sur les flots rs_1 et rs_2 est considérée et utilisée comme une valeur fausse. Or, après l'effacement des absences, nous avons $\lfloor rs_1 \rfloor \simeq \lfloor rs_2 \rfloor$ et $\lfloor xs_1 \rfloor \simeq \lfloor xs_2 \rfloor$. Si l'on se place dans l'interprétation de Kahn de la réinitialisation, nous obtenons $\mathsf{reset}_f\ \lfloor rs_1 \rfloor\ \lfloor xs_1 \rfloor \simeq \mathsf{reset}_f\ \lfloor rs_2 \rfloor\ \lfloor xs_2 \rfloor$, ce qui entre en contradiction avec la définition de Vélus, où $ys_1 \not\simeq ys_2$.

Cette ambiguïté est néanmoins levée dès que les absences des flots rs et xs sont alignées entre elles et avec celles du flot de sortie de f. C'est d'ailleurs une restriction qu'impose le système d'horloges de Lucid Synchrone [21], comme l'illustre le message d'erreur suivant obtenu à la compilation :

[21] Pouzet, 2006, p.76

```
let node main c =
  let n = 5 -> if c then 5 else 0 in
> reset (countdown n) every (true when c)
> ^
This expression has clock 'b,
but is used with clock (__c0:'c).
```

Dans cet exemple, nous forçons les variables n et c à avoir la même horloge `'b`, impossible à unifier avec l'horloge de l'expression `true when c`, d'où l'erreur. Dans Scade 6 ou Vélus en revanche, la sémantique du langage d'entrée prend en charge la réinitialisation sur des opérandes d'horloge arbitraire et la compilation est capable de générer du code séquentiel en adéquation. Nous montrons qu'en ajoutant la contrainte d'horloge, les deux formulations de la réinitialisation modulaire coïncident.

Lemme 3.13. Pour tous flots rs et xs alignés et sans erreurs, et pour toute instance f d'un nœud Lustre bien formé dont les sorties sont sans erreurs et alignées avec les entrées :

$$\forall k, f\ (\mathsf{mask}^k\ rs\ xs) \simeq \mathsf{mask}^k\ rs\ (\mathsf{reset}_f\ rs\ xs).$$

La démonstration de ce résultat requiert des propriétés sur le comportement de la fonction f, notamment vis-à-vis des absences. Nous proposons d'expliciter ces propriétés en section 3.4.5 et de démontrer au chapitre 4 qu'elles sont vraies de toute interprétation d'un nœud Lustre bien formé. Notons aussi que la fonction mask de Vélus est définie sur des flots de type Stream svalue. Le lemme ci-dessus fait plutôt référence à une définition de la même fonction, mais sur des flots de type Stream_ϵ sevalue.

3.4.4 Réinitialisation sans contraintes

Nous devons définir une procédure pour calculer la réinitialisation modulaire dans notre modèle dénotationnel synchrone, sans contrainte sur les horloges des flots d'entrée et de réinitialisation. Comme nous l'avons compris dans la section précédente, une telle définition doit nécessairement être sensible aux absences, ce qui pose problème dans une interprétation de Kahn. L'absence de synchronisation ne pose en revanche aucun problème à la compilation. Dans Vélus, la compilation d'automates hiérarchiques fait d'ailleurs apparaître des opérations de réinitialisation imbriquées dont les conditions de réinitialisation n'ont aucune raison d'être synchronisées [22]. Nous proposons d'encoder un opérateur dénotationnel alternatif [23].

[22] Bourke *et al.*, 2023b, figure 19
[23] Gérard, 2013, §2.6.4.2

$$
\begin{aligned}
&\mathsf{sreset}_f\ rs\ xs && := \mathsf{sreset}'_f\ rs\ xs\ (f\ xs)\\
&\mathsf{sreset}'_f\ (\mathsf{T}\cdot rs)\ xs\ ys && \simeq \mathsf{sreset}'_f\ (\mathsf{F}\cdot rs)\ xs\ (f\ xs)\\
&\mathsf{sreset}'_f\ (\mathsf{F}\cdot rs)\ (x\cdot xs)\ (y\cdot ys) && \simeq y\cdot(\mathsf{sreset}'_f\ rs\ xs\ ys)\\
&\mathsf{sreset}'_f\ (\mathsf{abs}\cdot rs)\ (x\cdot xs)\ (y\cdot ys) && \simeq y\cdot(\mathsf{sreset}'_f\ rs\ xs\ ys)
\end{aligned}
$$

Il applique f à xs initialement et à chaque fois qu'un signal est reçu sur le flot rs, passant le résultat du calcul dans un flot auxiliaire ys. Les valeurs de ys sont propagées une par une tant que l'absence ou le faux sont rencontrés sur rs et que des valeurs sont lues sur xs. Nous prouvons la cohérence de cette définition vis-à-vis de celle par masquage dans Vélus, et nous montrons que si l'on rétablit les contraintes de synchronisation, alors les deux versions dénotationnelles correspondent.

Lemme 3.14. Pour tous flots rs et xs sans erreurs et pour toute instance f,

$$\forall k, f\ (\mathsf{mask}^k\ rs\ xs) \simeq \mathsf{mask}^k\ rs\ (\mathsf{sreset}_f\ rs\ xs).$$

Lemme 3.15. Si de plus, rs et xs sont alignés et que f est une instance d'un nœud Lustre dont les sorties sont alignées avec les entrées,

$$\mathsf{sreset}_f\ rs\ xs \simeq \mathsf{reset}_f\ rs\ xs.$$

Pour simplifier la présentation, nous n'avons considéré ici que le cas des nœuds à une seule entrée et une seule sortie. Nous verrons en section 3.5 comment utiliser les environnements pour généraliser leur signature.

3.4.5 Éléments de preuve

Les lemmes 3.13,3.14 et 3.15 de la section précédente formalisent le comportement des différents opérateurs de réinitialisation appliqués à une instance f d'un nœud Lustre dans notre sémantique synchrone dénotationnelle. En plus de la continuité, nous identifions trois propriétés que doit satisfaire la fonction f afin d'interagir correctement avec l'opérateur de masquage. Intuitivement, ces trois propriétés correspondent aux trois segments d'un flot masqué, comme l'illustre le chronogramme suivant.

xs	3	3	3	3	3	3	3	3	3	3	...
rs	F	T	F	T	F	F	F	T	abs	F	...
$\mathsf{mask}^2\ rs\ xs$	abs	abs	abs	3	3	3	3	abs	abs	abs	...
$f(\mathsf{mask}^2\ rs\ xs)$	abs	abs	abs	3	2	1	0	abs	abs	abs	...

Le premier segment concerne les éléments arrivant avant le k-ième signal reçu sur rs, où k est l'indice de masquage (2 dans cet exemple). Le second segment concerne les éléments arrivant entre le k et le $k+1$-ième signal sur rs. Le dernier segment concerne les éléments arrivant après le $k+1$-ième signal.

Indépendance à l'absence initiale La fonction f doit commuter avec un élément absent arrivant sur son entrée. Formellement,

$$\forall xs,\ f\ (\mathsf{abs} \cdot xs) \simeq \mathsf{abs} \cdot (f\ xs).$$

Cela signifie qu'une absence reçue en entrée est non seulement propagée mais aussi qu'elle n'impacte pas l'état interne de f. C'est d'ailleurs l'idée derrière la notion d'horloge de base d'un nœud : lorsque ses arguments sont absents, un nœud et ses composants internes ne sont pas activés. Dans le cas d'une fonction f à plusieurs arguments, cette propriété n'est valide que lorsque toutes les entrées sont absentes, ce qui est plus faible que l'insensibilité aux absences, mentionnée précédemment et incompatible avec le modèle de Kahn.

Commutativité du préfixe La fonction f doit produire ses sorties au même rythme qu'elle lit ses entrées. Formellement,

$$\forall n\ xs\ ys, \mathsf{take}\ n\ xs \simeq \mathsf{take}\ n\ ys \to \mathsf{take}\ n\ (f\ xs) \simeq \mathsf{take}\ n\ (f\ ys).$$

Cela signifie notamment que la fonction n'a pas besoin de plus de n éléments sur son entrée pour produire n éléments sur sa sortie. Cette propriété constitue une véritable caractérisation du modèle synchrone et n'est trivialement pas vraie dans le modèle de Kahn (penser au filtrage). Nous avons aussi établi une version plus forte de ce résultat, mais qui n'est pas strictement nécessaire dans le cas du masquage :

$$\forall n\ xs,\ f\ (\mathsf{take}\ n\ xs) \simeq \mathsf{take}\ n\ (f\ xs).$$

Alignement des absences Si toutes les entrées de f sont absentes à un instant donné, alors toutes ses sorties doivent l'être également dans ce même instant. Cette propriété, conséquence de la correction du système d'horloges, permet d'isoler le dernier segment d'un flot masqué. Contrairement à l'indépendance aux absences, elle ne spécifie rien sur l'état interne de f. Nous l'encodons de la façon suivante, dans sa forme générale :

$$\forall n\ xs,\ \mathsf{first}\ (\mathsf{rem}^n\ xs) \preceq \mathsf{abs}^\omega \to \mathsf{first}\ (\mathsf{rem}^n\ (f\ xs)) \preceq \mathsf{abs}^\omega.$$

L'inégalité par préfixe est suffisante. Utiliser une équivalence $\simeq$ reviendrait ici à demander en plus à ce que le flot de sortie de la fonction f soit au moins aussi long que le flot d'entrée xs, et donc à prouver une autre forme de commutativité du préfixe. Dans le cas du masquage, une forme plus faible de cette propriété, mais aussi plus facile à démontrer, est suffisante :

$$\forall n\ xs,\ \mathsf{rem}^n\ xs \preceq \mathsf{abs}^\omega \to \mathsf{rem}^n\ (f\ xs) \preceq \mathsf{abs}^\omega.$$

Nous avons démontré ces trois propriétés des programmes Lustre, indépendamment les unes des autres, dans notre modèle synchrone dénotationnel en Coq. Chacun de ces résultats fait appel à des mécaniques de preuve bien distinctes que nous détaillerons au chapitre 4.

3.4.6 Bonus : définition fantaisie de la réinitialisation

Nous avons développé, en collaboration avec Guillaume Baudart, une définition constructive alternative de la réinitialisation modulaire. L'idée est de calculer à chaque instant une nouvelle instance de la fonction f et de ne la sélectionner que lorsqu'un signal de réinitialisation est effectivement reçu.

$$\mathsf{greset}_f\ rs\ xs := \mathsf{slicer}\ ((f\ xs) \cdot \mathsf{iter}\ f\ xs)\ rs$$

$$\mathsf{iter} : (\mathsf{Stream}\ A \to \mathsf{Stream}\ B) \to \mathsf{Stream}\ A \to \mathsf{Stream}\ (\mathsf{Stream}\ B)$$
$$\mathsf{iter}\ f\ xs \simeq (f\ xs) \cdot \mathsf{iter}\ f\ (\mathsf{rem}\ xs)$$

$$\mathsf{slicer} : \mathsf{Stream}\ (\mathsf{Stream}\ B) \to \mathsf{Stream}\ \mathsf{bool} \to \mathsf{Stream}\ B$$
$$\mathsf{slicer}\ ((x \cdot xs) \cdot ys \cdot ss)\ (\mathsf{F} \cdot rs) \simeq x \cdot (\mathsf{slicer}\ (xs \cdot ss)\ rs)$$
$$\mathsf{slicer}\ (xs \cdot (y \cdot ys) \cdot ss)\ (\mathsf{T} \cdot rs) \simeq y \cdot (\mathsf{slicer}\ (ys \cdot ss)\ rs)$$

La fonction iter calcule de façon paresseuse toutes les instances de f sur xs dans le temps et les stocke dans un flot de flots. La fonction slicer choisit de propager ou non les valeurs des instances successives selon si un signal est reçu sur rs. Un éventuel signal reçu au premier instant est ignoré grâce à la redondance de la première instance introduite dans la définition de greset. Cet opérateur, sans doute équivalent à sreset défini en section 3.4.4, ne présente pas d'intérêt particulier dans notre cas. Nous ne l'avons pas encodé dans Coq.

3.5 Interprétation et composition

Nous avons décrit dans les sections précédentes tous les composants nécessaires à l'encodage dans Coq des différents opérateurs de flots du langage. Dans cette section, nous présentons l'infrastructure permettant d'interpréter les éléments des différents niveaux de la syntaxe Lustre.

3.5.1 Environnements

Variables Au sein d'un nœud, chaque variable est interprétée par un flot échantillonné de valeurs possiblement erronées, de type $\mathsf{Stream}_\epsilon\ \mathsf{sevalue}$. Le plongement profond imposé par la structure du compilateur ne permet pas d'utiliser un type de données dédié à l'énumération des variables déclarées dans un programme. Les identifiants de variables de l'arbre de syntaxe sont tous des objets de type ident := positive, des entiers naturels strictement positifs. Nous définissons SEnv, le type des environnements de flots, comme une instance du CPO des produits indexés, sans dépendance sur l'indice.

$$\mathsf{SEnv} := \Pi\ (\lambda i : \mathsf{ident} \to \mathsf{Stream}_\epsilon\ \mathsf{sevalue})$$

Avec cet encodage, tout entier strictement positif possède son entrée dans un environnement, même s'il ne correspond à aucune variable déclarée. Nous prendrons soin dans la suite d'associer systématiquement le flot vide $\bot$ à de tels identifiants.

Nœuds Au sein d'un programme, chaque nœud est interprété comme une fonction continue des flots d'entrée dans les flots de sortie. Ici encore, les identifiants des nœuds sont de type ident et peuvent ne pas être attribués dans le programme. Nous définissons FEnv, le type des environnements de fonctions de flots comme un produit indexé non dépendant.

$$\mathsf{FEnv} := \Pi\,(\lambda i : \mathsf{ident} \to (\mathsf{SEnv} \to_c \mathsf{SEnv}))$$

3.5.2 Imbrication de points fixes

Comme dans les réseaux de Kahn [24] et dans leur formalisation avec la bibliothèque CPO, la sémantique d'un nœud est définie comme le plus petit point fixe de ses équations. Nous adaptons cette approche à notre plongement profond avec des équations de la forme $x^+ = e^+$, où x^+ est une liste de variables locales ou de sortie et e^+ est une liste d'expressions les définissant. La longueur des deux listes est susceptible de différer car une seule expression peut définir plusieurs flots (cf. remarques en section 2.1.4). Ce point qui semble anodin au premier abord entraîne, nous le verrons, de nombreuses complications lors de l'encodage en Coq.

Équations Le point fixe des équations d'un nœud n est calculé relativement à un environnement env_G : FEnv dénotant les nœuds déclarés dans le programme et à un environnement env_I : SEnv associant les variables d'entrée de n à leurs flots de valeurs. Formellement, nous le définissons comme :

$$\mathsf{fixp}\,(\mathsf{denot_N}\ n\ env_G\ env_I) : \mathsf{SEnv}.$$

Parmi les stratégies envisageables pour définir la fonction $\mathsf{denot_N}$, chargée de mettre à jour le résultat en itérant sur les équations $n.\mathsf{eqs}$, nous choisissons celle consistant à interpréter toutes les équations dans le même contexte. Cela permet notamment de rendre le calcul indépendant de l'ordre des équations dans la liste.

$$\begin{aligned}&\mathsf{denot_N}\ n\ env_G\ env_I : \mathsf{SEnv} \to_c \mathsf{SEnv} :=\\ &\quad \lambda env.\ \mathsf{fold\_right}\ (\lambda eq.\ \mathsf{denot_{EQ}}\ eq\ env_G\ env_I\ env)\ n.\mathsf{eqs}\ \bot.\end{aligned}$$

Cette fonction prend comme paramètre *env*, la précédente approximation d'un environnement des variables locales et de sortie. Elle calcule un nouvel

[24] Kahn, 1974, §3

environnement env' en accumulant, à partir de l'environnement vide $\bot$, les flots des variables définies par les différentes équations.

$$\begin{aligned}&\mathsf{denot}_{\mathsf{EQ}}\ (xs = es)\ env_G\ env_I\ env : \mathsf{SEnv} \to_c \mathsf{SEnv} :=\\ &\quad \mathsf{let}\ ss = \mathsf{denot}_{\mathsf{E}}\ es\ env_G\ env_I\ env\ \mathsf{in}\\ &\quad \lambda env'.\ env' \uplus \{xs \mapsto ss\}.\end{aligned}$$

La fonction $\mathsf{denot}_{\mathsf{E}}$ que nous définirons ci-après calcule les flots d'une liste d'expressions. Rien dans la syntaxe des équations ne permet de garantir que l'ensemble ss ainsi calculé soit de même taille que l'ensemble xs des variables. Il s'agit d'une contrainte de typage exprimée de façon orthogonale par un prédicat, sur lequel il est impossible de faire une analyse de cas dans le cadre d'une fonction constructive comme $\mathsf{denot}_{\mathsf{EQ}}$. La notation $\uplus$ adoptée ici cache donc un examen des longueurs de xs et ss ainsi qu'une procédure permettant de retrouver le flot associé à une variable dans le nouvel environnement calculé. Si l'un des tests échoue, un flot constant d'$\mathsf{err}_{\mathsf{ty}}$ est placé dans env'.

À titre d'exemple, considérons l'évaluation d'un nœud constitué des deux équations `x = 0 fby y + 1` et `y = x`. Ce nœud ne possédant pas d'entrées, nous avons initialement $env_I = \bot$ et $env = \bot$. La fonction $\mathsf{denot}_{\mathsf{N}}$ évalue les deux équations dans l'environnement env pour obtenir $env' = \bot \uplus \{x \mapsto 0 \cdot \bot\} \uplus \{y \mapsto \bot\}$. Ce nouvel environnement sert d'argument à $\mathsf{denot}_{\mathsf{N}}$ dans l'itération suivante du calcul du point fixe, pour obtenir $env'' = \bot \uplus \{x \mapsto 0 \cdot \bot\} \uplus \{y \mapsto 0 \cdot \bot\}$. En répétant cette opération, nous convergeons bien vers une solution $env = \{x \mapsto 0 \cdot 1 \cdot 2 \cdot 3 \cdots; y \mapsto 0 \cdot 1 \cdot 2 \cdot 3 \cdots; \_ \mapsto \bot\}$.

Programmes La sémantique d'un programme complet G est obtenue en prenant le point fixe de la dénotation des nœuds définie précédemment.

$$\begin{aligned}&\mathsf{denot}\ G : \mathsf{FEnv} := \mathsf{fixp}\ (\mathsf{denot}_{\mathsf{G}}\ G)\\ &\mathsf{denot}_{\mathsf{G}}\ G : \mathsf{FEnv} \to_c \mathsf{FEnv} :=\\ &\quad \lambda env_G\ (f : \mathsf{ident})\ env_I.\ \mathsf{fixp}\ (\mathsf{denot}_{\mathsf{N}}\ (G.f)\ env_G\ env_I).\end{aligned}$$

Ici, f est l'identifiant du nœud à interpréter, et $G.f$ est la fonction permettant d'obtenir le corps de la définition syntaxique du nœud dans le programme. Si le nœud n'existe pas dans G alors la fonction vide $\bot$ est renvoyée.

Expressions La sémantique d'une expression est donnée, dans le cas général, par un ensemble de flots. Ceci est dû aux instances de nœuds, qui sont des expressions du langage et qui peuvent produire plusieurs flots en sortie. L'information, notée $|e|$, du nombre de flots que doit produire une expression e est disponible sous forme d'annotations dans la syntaxe (cf. figure 2.4). Il est indispensable d'exploiter cette information pour déterminer

$\mathsf{denot_E}\ e = \mathsf{err_{ty}}^{\omega}$
si aucun des cas ci-dessous ne s'applique

$\mathsf{denot_E}\ c := \mathsf{const}\ c\ (\mathsf{base\text{-}of}\ env_I)$
$\mathsf{denot_E}\ C := \mathsf{const}\ C\ (\mathsf{base\text{-}of}\ env_I)$

$\mathsf{denot_E}\ x := env_I(x) \quad$ si $x \in \Gamma_{\mathrm{E}} \setminus \Gamma_{\mathrm{S}}$
$\mathsf{denot_E}\ x := env(x) \quad$ si $x \in \Gamma_{\mathrm{S}} \setminus \Gamma_{\mathrm{E}}$

$\mathsf{denot_E}\ (\diamond\, e) := \mathsf{lift}^{\diamond}\ (\mathsf{denot_E}\ e)$
si $|e| = 1$
$\mathsf{denot_E}\ (e_1 \oplus e_2) := \mathsf{lift}^{\oplus}\ (\mathsf{denot_E}\ e_1)\ (\mathsf{denot_E}\ e_2)$
si $|e_1| = 1$ et $|e_2| = 1$
$\mathsf{denot_E}\ (es_1\ \mathtt{fby}\ es_2) := \mathsf{fby}^{\uparrow}\ (\mathsf{denot_E}\ es_1)\ (\mathsf{denot_E}\ es_2)$
si $|es_1| = |es_2|$
$\mathsf{denot_E}\ (es\ \mathtt{when}\ C(x)) := (\mathsf{when}_C\ (\mathsf{denot_E}\ x))^{\uparrow}\ (\mathsf{denot_E}\ es)$

$\mathsf{denot_E}\ (\mathtt{merge}\ x\ ess) := (\mathsf{merge}\ (\mathsf{denot_E}\ x))^{\uparrow\uparrow}\ (\mathsf{denot_E}\ ess)$
si $\exists n, \forall i, |ess_i| = n$
$\mathsf{denot_E}\ (\mathtt{case}\ e\ ess) := (\mathsf{case}\ (\mathsf{denot_E}\ e))^{\uparrow\uparrow}\ (\mathsf{denot_E}\ ess)$
si $|e| = 1$ et $\exists n, \forall i, |ess_i| = n$
$\mathsf{denot_E}\ (\mathtt{case}\ e\ ess\ e_d) := (\mathsf{case}'\ (\mathsf{denot_E}\ e))^{\uparrow\uparrow}\ (\mathsf{denot_E}\ ess)\ (\mathsf{denot_E}\ e_d)$
si $|e| = 1$ et $\exists n, \forall i, |ess_i| = |e_d| = n$
$\mathsf{denot_E}\ (f\ es) := env_G\ f\ (\mathsf{denot_E}\ es)$
si $|es| = |(G.f).\mathsf{in}|$
$\mathsf{denot_E}\ ((\mathtt{restart}\ f\ \mathtt{every}\ e_r)\ es) := \mathsf{sreset}\ (env_G\ f)\ (\mathsf{denot_E}\ e_r)\ (\mathsf{denot_E}\ es)$
si $|es| = |(G.f).\mathsf{in}|$ et $|e_r| = 1$

FIGURE 3.3 – Interprétation des expressions

la bonne instance de CPO à utiliser dans l'interprétation. La fonction de dénotation calcule donc un produit fini, dont la taille dépend de l'expression :

$$\mathsf{denot_E}\ e\ env_G\ env_I\ env : (\mathsf{Stream}_{\epsilon}\ \mathsf{sevalue})^{|e|}.$$

Dans la suite, nous surchargeons cette fonction afin qu'elle traite à la fois les expressions et les listes d'expressions. Chaque expression du langage est récursivement associée à l'un des opérateurs synchrones détaillés en section 3.3. Nous donnons en figure 3.3 ci-dessus les différents cas possibles, y compris ceux erronés, en omettant les paramètres env_G, env_I et env qui restent inchangés lors des appels récursifs. La fonction base-of calcule

l'horloge de base du nœud courant, vraie lorsque l'une au moins des entrées de env_I est présente. Dans la figure, la notation $\mathsf{op}^{\uparrow}$ désigne l'application continue d'un opérateur op à un ou plusieurs vecteurs de flots. La notation $\mathsf{op}^{\uparrow\uparrow}$ désigne l'application continue de op à une matrice de flots. Lors d'un appel de nœud, avec ou sans réinitialisation, les vecteurs de flots passés en argument à l'instance de env_G doivent être convertis en un environnement, et l'environnement résultant doit lui-même être converti en un vecteur de flots. Par souci de lisibilité, nous omettons ces conversions dans la figure.

Les conditions d'égalité des longueurs spécifiées sous les équations de la figure 3.3, indispensables pour définir une fonction totale, entraînent des complications liées à l'utilisation de types dépendants. Dans le cas es_1 `fby` es_2 par exemple, l'opérateur $\mathsf{fby}^{\uparrow}$ ne s'applique qu'à deux vecteurs de même taille. Il faut comparer dynamiquement les valeurs $n = |es_1|$ et $m = |es_2|$ pour garantir que le même nombre de flots est calculé par les deux appels récursifs. En Coq, une analyse de cas sur `Nat.eq_dec m n` permet d'obtenir constructivement une preuve `eqm` de l'égalité entre les deux valeurs, utilisée ensuite par la fonction `eq_rect` de façon à convertir le vecteur $ss_2 : \mathsf{Stream}_{\epsilon}^{m}$ en un vecteur de type $\mathsf{Stream}_{\epsilon}^{n}$. Le même procédé est utilisé pour s'assurer que le résultat de $\mathtt{fby}^{\uparrow}$ est de même longueur que l'annotation.

```
(* denot_exp (Efby es1 es2 ann) := *)
let n := list_sum (List.map numstreams es1) in
let m := list_sum (List.map numstreams es2) in
let ss1 := denot_exps es1 in   (* ss1 : Stream_ε^n *)
let ss2 := denot_exps es2 in   (* ss2 : Stream_ε^m *)
match Nat.eq_dec m n, Nat.eq_dec n (length ann) with
| left eqm, left eqann ⇒
    eq_rect _ _ (fby↑ ss1 (eq_rect _ _ ss2 _ eqm)) _ eqann
| _, _ ⇒ const errTy
end
```

3.5.3 Correction

Nous pouvons à présent utiliser les composants de la sémantique dénotationnelle pour former un témoin satisfaisant les prédicats de la sémantique relationnelle. Au sein d'un nœud f d'un programme G, en supposant l'environnement env_I infini et sans erreurs sur le domaine Γ_{E} des variables d'entrée de f, nous définissons des éléments suivants.

- $env_G := \mathsf{denot}\ G$, l'environnement global du programme.
- $env := env_G\ f\ env_I$, l'environnement de f, supposé infini et sans erreurs sur le domaine Γ_{S} des variables locales et de sortie.
- $bs := \lceil \mathsf{base\text{-}of}\ env_I \rceil$, l'horloge de base de f.
- $H := \lambda x. \begin{cases} \lceil env_I(x) \rceil & \text{si } x \in \Gamma_{\mathrm{E}} \\ \lceil env(x) \rceil & \text{si } x \in \Gamma_{\mathrm{S}} \end{cases}$, l'histoire de f.

En combinant les lemmes énoncés dans la section 3.3, et grâce à un principe d'induction mutuellement récursive sur la structure du programme, nous prouvons conjointement que l'interprétation dénotationnelle des expressions et des nœuds respecte bien les prédicats de la sémantique relationnelle.

Lemme 3.16. Pour toute expression e dont la dénotation dans env_G, env_I et env est infinie et sans erreurs,

$$G, H, bs \vdash e \Downarrow \lceil \mathsf{denot_E}\ e\ env_G\ env_I\ env \rceil.$$

Théorème 3.17. Pour tout nœud f et tout environnement env_I, si la dénotation de f appliquée à env_I dans env_G est infinie et sans erreurs,

$$G \vdash f(\lceil env_I \rceil) \Downarrow \lceil env_G\ f\ env_I \rceil.$$

3.6 Bilan

Dans ce chapitre, nous avons détaillé l'encodage d'une sémantique dénotationnelle synchrone pour le langage d'entrée pris en charge par Vélus. Grâce à un mécanisme de calcul des points fixes, cette sémantique permet d'interpréter de façon constructive tous les programmes de la syntaxe, y compris ceux rejetés par le compilateur car mal typés ou ne respectant pas le principe de causalité. Notre modèle dénotationnel donne donc une sémantique aux analyses statiques du compilateur, avec une vérification dynamique.

Nous avons démontré que si l'interprétation dénotationnelle résulte en des flots infinis et sans erreurs, alors ces flots constituent un témoin des prédicats de la sémantique relationnelle, validant ainsi le théorème de correction de la compilation. Nous avons donc remplacé l'obligation de fournir un modèle de la sémantique par l'obligation de prouver des propriétés d'un objet bien défini. La preuve de ces propriétés fait l'objet du prochain chapitre.

D'autre part, nous avons démontré qu'après effacement des absences dans les flots, notre interprétation des différents opérateurs coïncide avec celle de Kahn, toujours à condition que les flots soient sans erreurs.

Chapitre 4

Propriétés du modèle

Nous avons démontré que la sémantique dénotationnelle synchrone de Lustre implémentée à l'aide de la bibliothèque CPO constitue bien un modèle de la sémantique relationnelle définie dans Vélus, à condition que les flots calculés soient infinis et sans erreurs et que les fonctions associées aux nœuds satisfassent certaines relations algébriques. Dans ce chapitre, nous démontrons que la plupart de ces propriétés peuvent être décidées grâce aux différentes analyses statiques effectuées lors de la compilation.

Nous commençons par expliquer notre utilisation du principe de causalité syntaxique pour prouver l'infinitude des flots calculés. Nous prouvons ensuite les propriétés algébriques des fonctions de flots, décrites au chapitre précédent, mais cette fois par un raisonnement sémantique. Enfin, nous combinons ces deux techniques, syntaxique et sémantique, pour prouver la sûreté du système de types de Vélus. Cela nécessite d'introduire un prédicat caractérisant précisément les occurrences d'erreurs arithmétiques ou logiques, seules erreurs qui ne peuvent pas toujours être exclues statiquement.

4.1 Infinitude des flots et causalité

À la différence du modèle de Kahn, le modèle synchrone utilise l'élément distingué abs dans les flots pour contrôler la synchronisation des calculs et du rythme global du programme. L'exécution à l'infini d'un programme dans le modèle synchrone résulte en une alternance de valeurs présentes et d'éléments abs, la position de ces derniers étant déterminée par les règles du système d'horloges. Un problème de causalité dans le programme entraîne en revanche un blocage de l'exécution et donc une impossibilité de calculer le résultat. Dans le modèle synchrone, cela se traduit par des flots en sortie de taille finie, là où des flots infinis, éventuellement constitués d'une infinité d'éléments abs, sont attendus.

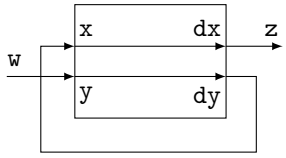
Considérons à nouveau l'exemple de la section 2.1.6.

```
node bad (...) returns (x : int)
let
  x = x;
tel
```

Parmi tous les points fixe de l'équation de x, la sémantique dénotationnelle considère le plus petit pour l'ordre préfixe, soit le flot vide $\perp$. Vélus refuse de compiler ce genre de programmes sur la base d'une heuristique classique [1] consistant à analyser les dépendances entre les variables d'un nœud et à chercher des cycles dans le graphe résultant. Si une dépendance instantanée (sans délai) d'une variable sur elle-même est détectée, alors l'analyse rejette le programme. Cette caractérisation est toutefois une sur-approximation de l'ensemble des programmes causaux. Considérons par exemple le programme formé des deux nœuds suivants.

```
node id (x,y : int) returns (dx,dy : int)
let
  dx,dy = (x,y);
tel

node main (w : int) returns (z : int)
var v : int;
let
  z,v = id (v, w);
tel
```

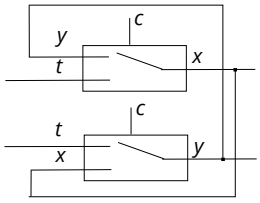


Le premier nœud branche directement ses sorties sur ses entrées, tandis que le second l'instancie sur une entrée et une sortie. Avec l'analyse de dépendances implémentée dans Vélus, le second nœud est considéré comme incorrect car la définition de v dépend de l'instance `id (v,w)`, qui dépend elle-même de v. Ce programme est pourtant bien causal car la première sortie du nœud `id` ne dépend pas de la seconde entrée. Nous remarquons d'ailleurs l'absence de cycle dans la représentation graphique du programme. Mais dans Vélus, le compilateur n'effectue pas d'analyse aussi précise. D'autres compilateurs, comme Zélus [2], implémentent un système de types expressif permettant d'encoder plus finement les relations de dépendance entre variables d'un même nœud et acceptent l'exemple précédent. La compilation de tels programmes est délicate et nécessite des techniques d'*inlining* [3]ou de génération de multiples fonctions `step` [4] afin de pouvoir calculer un ordonnancement des équations problématiques.

D'autres programmes reposent sur une notion de causalité dynamique, comme celui de l'exemple suivant.

[1] Caspi *et al.*, 1987, §4.1
[2] Benveniste *et al.*, 2014
[3] `https://github.com/INRIA/zelus/blob/main/compiler/rewrite/markfunctions.ml`
[4] Pouzet et Raymond, 2009

```
node commute (c : bool; t : int)
returns (x,y : int)
let
  x = if c then y else t;
  y = if c then t else x;
tel
```

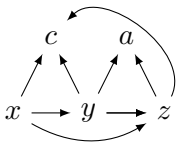


Bien que les équations soient syntaxiquement inter-dépendantes, ce nœud ne présente aucun problème de causalité, car la dépendance entre les variables x et y s'inverse selon la valeur de c. L'analyse de ce type de programme est difficile et le schéma de compilation implémenté par Vélus ne permet actuellement pas de les compiler. Notons que dans cet exemple précis, notre définition de la sémantique synchrone dénotationnelle est insuffisante, pour les mêmes raisons que celles décrites en section 3.4.2. Notre interprétation de l'opérateur if-then-else/case examine chacune des deux branches avant de produire une valeur, ce qui aboutirait ici en un flot systématiquement vide. Il faudrait alors définir un opérateur alternatif, qui produirait sa valeur de façon optimiste avant de lire le flot de la branche non sélectionnée, à la manière de l'opérateur merge que nous avons proposé en figure 3.2 du chapitre précédent.

4.1.1 Principe d'induction causale

Dans Vélus, un nœud est *causal* si ses variables peuvent être ordonnées en une suite $x_1, \ldots, x_n$ telle que x_i ne dépende pas instantanément de $x_{j \geq i}$. Les variables à droite d'un `fby` ne comptent pas comme dépendances instantanées. Par exemple, dans l'équation `z = x fby y`, la variable z dépend instantanément de x (on note ici $x < z$) mais pas de y. Cette relation de dépendance est formalisée dans Vélus pour le langage dans sa forme générale, ce qui implique notamment la prise en charge des expressions imbriquées, comme dans l'exemple suivant.

```
node caus (c : bool; a : int)
returns (x,y,z : int)
let
  x,y,z = if c then (y,z,a)
          else (z, id (0 fby x, a));
tel
```



Grâce à une analyse précise des expressions, la procédure implémentée dans le compilateur peut calculer l'ordre c, a, z, y, x des variables satisfaisant les contraintes de dépendances. Lorsqu'une telle suite existe, elle sert de témoin au principe d'induction général.

Théorème 4.1 (principe d'induction causale [5])**.** Pour tout nœud causal et bien typé dans un environnement de variables Γ et pour toute propriété P sur les listes de variables :

si $P\ [\,]$
et $\forall x \in \Gamma, \forall l,$ si $P\ l \wedge (\forall y, y < x \to y \in l)$ alors $P\ (x :: l)$
alors $\exists l, P\ l \wedge \mathsf{Permutation}\ l\ \Gamma.$

L'environnement de typage Γ, représenté par une liste d'associations des variables et de leurs types (omis ici) est directement extrait de l'arbre de syntaxe abstraite du programme et ne respecte en général pas l'ordre des dépendances, c'est pourquoi l'usage d'une permutation est nécessaire. Notons que cela n'a pas d'impact lorsque la propriété P est du type `Forall` sur les listes. Ce théorème permet, au sein d'une équation d'un nœud, d'examiner l'expression définissant une variable tout en disposant d'une hypothèse d'induction sur les variables dont elle dépend instantanément. Dans le nœud `caus` donné ci-dessus, il est par exemple possible de prouver une propriété sur le flot de x tout en supposant cette même propriété sur les flots de y et z. En revanche, lors de l'examen de l'expression définissant y, le principe d'induction ne donne aucune hypothèse sur x car ce dernier apparaît à droite d'un `fby`.

Cet outil a notamment permis de démontrer le déterminisme de la sémantique relationnelle (cf. section 2.1.7) ainsi que la correction du système d'horloges [6]. Nous montrons à présent qu'il permet aussi de prouver l'infinitude des flots calculés dans la sémantique synchrone dénotationnelle.

4.1.2 Productivité de la sémantique synchrone

Dans le modèle des flots possiblement finis de la bibliothèque CPO, nous définissons et notons la longueur minimale d'un flot s de la façon suivante :

$$|s| \geq n := \mathsf{is\text{-}cons}\ (\mathsf{rem}^{n-1}\ s).$$

Autrement dit, s doit contenir au moins n occurrences du constructeur `Cons`. Dans la suite, nous utilisons la même notation $|\cdot| \geq \cdot$ pour désigner la longueur minimale des flots d'une liste ou d'un environnement.

Théorème 4.2. Un flot s est infini si et seulement si $\forall n,\ |s| \geq n$.

Nous examinons ensuite tous les opérateurs définis en section 3.3 et déterminons une borne inférieure sur la longueur des flots résultants.

Propriété 4.1 (préservation des longueurs)**.**

1. $\forall c\ bs\ n,\ |bs| \geq n \to |\mathsf{const}\ c\ bs| \geq n$

[5] Pesin, 2023, §3.1
[6] Bourke *et al.*, 2021

2. $\forall s\, n,\ |s| \geq n \rightarrow |\mathsf{lift}^{\diamond}\ s| \geq n$
3. $\forall s_1\ s_2\ n,\ |s_1| \geq n \rightarrow |s_2| \geq n \rightarrow |\mathsf{lift}^{\oplus}\ s_1\ s_2| \geq n$
4. $\forall s_1\ s_2\ n,\ |s_1| \geq n \rightarrow |s_2| \geq n \rightarrow |\mathsf{fby}\ s_1\ s_2| \geq n$
5. $\forall s_c\ s\ n,\ |s_c| \geq n \rightarrow |s| \geq n \rightarrow |\mathsf{when}_C\ s_c\ s| \geq n$
6. $\forall\, s_c\ ss\ n,\ |s_c| \geq n \rightarrow (\forall i,\ |ss_i| \geq n) \rightarrow |\mathsf{merge}\ s_c\ ss| \geq n$
7. $\forall\, s_c\ ss\ n,\ |s_c| \geq n \rightarrow (\forall i,\ |ss_i| \geq n) \rightarrow |\mathsf{case}\ s_c\ ss| \geq n$
8. $\forall\, s_c\ s_d\ ss\ n,\ |s_c| \geq n \rightarrow |s_d| \geq n \rightarrow (\forall i,\ |ss_i| \geq n) \rightarrow |\mathsf{case}'\ s_c\ s_d\ ss| \geq n$
9. $\forall f\ s_r\ s\ n,\ |s_r| \geq n \rightarrow |s| \geq n \rightarrow (\forall s\, m,\ |s| \geq m \rightarrow |f\ s| \geq m) \rightarrow |\mathsf{sreset}_f\ s_r\ s| \geq n.$

L'opérateur de délai vérifie en outre une relation cruciale pour le raisonnement sur la productivité des programmes.

Propriété 4.2 (Productivité du délai)**.**

10. $\forall s_1\ s_2\ n,\ |s_1| \geq n+1 \rightarrow |s_2| \geq n \rightarrow |\mathsf{fby}\ s_1\ s_2| \geq n+1.$

Notons qu'aucune hypothèse sur les flots, autre que leur longueur minimale, n'est nécessaire pour démontrer ces propriétés. En particulier, le traitement des valeurs erronées par les différents opérateurs que nous avons définis est compatible avec la préservation des longueurs, ce qui n'est pas le cas d'un opérateur arrêtant sa production en cas d'erreur, par exemple. En exploitant la propriété 4.1, nous pouvons déterminer, par récurrence structurelle sur la syntaxe, une borne sur la longueur des flots d'une expression en fonction de la longueur des flots de son environnement.

Lemme 4.3. Pour tout entier n, expression e et environnements env_G, env_I et env, si $|env_I| \geq n$ et $|env| \geq n$ et $\forall f\, s\, m,\ |s| \geq m \rightarrow |env_G\ f\ s| \geq m$, alors $|\mathsf{denot}_{\mathsf{E}}\ e\ env_G\ env_I\ env| \geq n$.

En utilisant ce résultat et la propriété 4.2, nous obtenons, à nouveau par induction sur la syntaxe, une relation de récurrence sur la longueur des flots de la dénotation d'une expression.

Lemme 4.4. Pour tout entier n, expression e et environnements env_G, env_I et env, si $|env_I| \geq n+1$ et $|env| \geq n$ et $\forall f\, s\, m,\ |s| \geq m \rightarrow |env_G\ f\ s| \geq m$ et si pour toute variable x instantanée (c'est-à-dire pas à droite d'un `fby`) dans e, $|env(x)| \geq n+1$, alors $|\mathsf{denot}_{\mathsf{E}}\ e\ env_G\ env_I\ env| \geq n+1$.

Pour étendre ce résultat à un nœud causal nd, nous instancions env par le point fixe de ses équations. Formellement, $env := \mathsf{fixp}\,(\mathsf{denot}_{\mathsf{N}}\ nd\ env_G\ env_I)$. Cela nous permet d'appliquer le principe d'induction causale donné dans le théorème 4.1, avec comme propriété $P := \texttt{Forall}\ (\lambda x.\ |env(x)| \geq n+1)$.

Lemme 4.5. Soit $env := \mathsf{fixp}\,(\mathsf{denot}_{\mathsf{N}}\ nd\ env_G\ env_I)$ le point fixe d'un nœud causal nd. Pour tout n, si $|env_I| \geq n+1$ et $|env| \geq n$ et $\forall f\, s\, m,\ |s| \geq m \rightarrow |env_G\ f\ s| \geq m$, alors $|env| \geq n+1$.

Autrement dit, si les entrés env_I du nœud sont définies au moins jusqu'au rang $n+1$, que ses variables internes le sont au moins jusqu'au rang n et que les autre nœuds du programme préservent la longueur minimale de flots, alors les sorties du nœud sont définies au moins jusqu'au rang $n+1$. Nous utilisons ce résultat et une simple récurrence sur les entiers naturels pour garantir la productivité d'un nœud causal. Le cas de base avec $n = 0$ est trivial.

Lemme 4.6. Pour tout n, env_G, env_I et tout nœud causal nd, si $|env_I| \geq n$ et $\forall f\, s\, m$, $|s| \geq m \rightarrow |env_G\ f\ s| \geq m$, alors $|\mathsf{fixp}\ (\mathsf{denot_N}\ nd\ env_G\ env_I)| \geq n$.

Dans Vélus, les nœuds d'un programme sont déclarés séquentiellement en une liste $nd_1, \ldots, nd_n$ au sein de laquelle chaque nœud nd_i ne peut instancier que des nœuds $nd_{j<i}$. Nous pouvons donc éliminer l'hypothèse sur l'environnement global env_G du lemme 4.6 en utilisant une induction sur l'ordre des nœuds du programme. Nous concluons en passant à l'infini avec le théorème 4.2.

Théorème 4.7. Pour tout nœud f d'un programme compilable G, si env_I est infini sur le domaine des entrées de f alors $\mathsf{denot}\ G\ f\ env_I$ est infini sur le domaine des sorties de f.

4.1.3 Discussion

Dans cette section, nous avons prouvé la productivité à l'infini des programmes Lustre lorsqu'ils satisfont les critères syntaxiques de causalité imposés par le compilateur. Le raisonnement sur les fonctions de flots menant aux propriétés 4.1 et 4.2 est assez naturel, même avec les définitions complexes de la bibliothèque CPO. En revanche, l'infrastructure autour du principe d'induction causale et le raisonnement sur l'imbrication des composants du langage sont lourds et représentent dans le développement environ 1500 lignes d'énoncés et de preuves. Heureusement, les propriétés qu'il nous reste à établir sont, contrairement à l'infinitude des flots, admissibles au sens de Scott et peuvent être démontrées par induction continue sur les points fixes, sans utiliser le théorème 4.1 ni les hypothèses syntaxiques de causalité.

4.2 Commutativité du préfixe

La propriété que nous vérifions ici est nécessaire à la preuve d'équivalence des deux formulations de la réinitialisation modulaire, énoncée en section 3.4.4. Elle conditionne le nombre d'éléments en entrée nécessaires à une fonction de flots f pour calculer sa sortie :

$$\forall n\ xs,\ f\ (\mathsf{take}\ n\ xs) \simeq \mathsf{take}\ n\ (f\ xs).$$

Cette propriété est indépendante de la préservation des longueurs en raison du comportement de l'opérateur take, extrayant *au plus* n éléments d'un flot.

Proposition 4.8. L'opérateur take satisfait les relations suivantes.

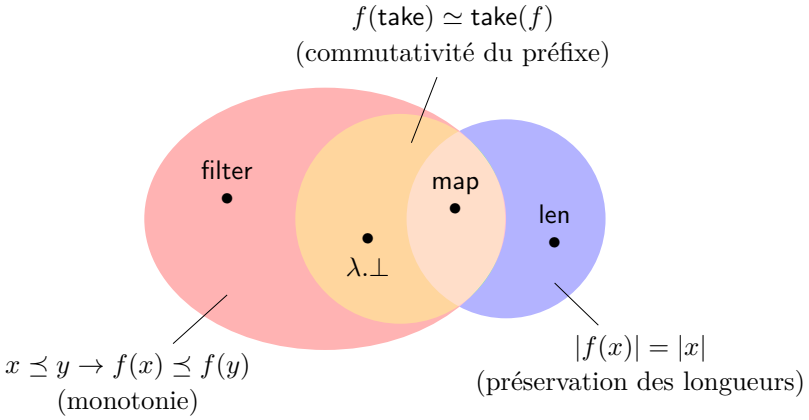
— $\forall xs\ n,\ \mathsf{take}\ n\ xs \preceq xs$

— $\forall xs\ n, |xs| \geq n \to |\mathsf{take}\ n\ xs| = n$

— $\forall xs\ n, |xs| < n \to |\mathsf{take}\ n\ xs| = |xs|$.

Remarque : ici encore, la longueur des flots notée $|\cdot| = \cdot$ ne peut pas être calculée mais fait plutôt référence à un prédicat.

Par exemple, la fonction constante $\lambda.\bot$ retournant le flot vide sur toutes ses entrées ne préserve pas les longueurs mais commute bien avec take. Nous pouvons classifier les différentes propriétés des fonctions de flots selon le diagramme suivant et donner une instance de chaque catégorie.



La fonction filter est l'exemple typique d'un opérateur monotone ne préservant pas les longueurs, il est très simple de vérifier qu'il ne commute pas avec take. La fonction len, définie par les équations suivantes, préserve bien la longueur de son argument mais n'est pas monotone pour l'ordre préfixe.

$$\begin{aligned}
&\mathsf{len}\ \bot && \simeq \bot \\
&\mathsf{len}\ (v_1 \cdot \bot) && \simeq 1 \cdot \bot \\
&\mathsf{len}\ (v_1 \cdot v_2 \cdot \bot) && \simeq 2 \cdot 2 \cdot \bot \\
&\mathsf{len}\ (v_1 \cdot v_2 \cdot v_3 \cdot \bot) && \simeq 3 \cdot 3 \cdot 3 \cdot \bot \\
&\dots
\end{aligned}$$

Ce genre de fonction ne nous intéresse pas vraiment car on ne peut pas les programmer dans notre modèle. Les autres inclusions du diagramme sont quant à elles des conséquences des propositions suivantes.

Proposition 4.9. Pour toute fonction f compatible avec la relation d'équivalence $\simeq$, si $\forall n\ xs,\ f\ (\mathsf{take}\ n\ xs) \simeq \mathsf{take}\ n\ (f\ xs)$ alors f est monotone.

Démonstration. Soient $xs \preceq ys$ deux flots ordonnés par l'ordre préfixe. Si xs est infini alors $xs \simeq ys$ et $f\ xs \simeq f\ ys \preceq f\ ys$. Sinon xs est un préfixe fini de ys donc $\exists n, xs \simeq \mathsf{take}\ n\ ys$, d'où $f\ xs \simeq f\,(\mathsf{take}\ n\ ys) \simeq \mathsf{take}\ n\ (f\ ys) \preceq f\ ys$. □

Proposition 4.10. Pour toute fonction monotone f préservant les longueurs, c'est-à-dire si $\forall xs,\ |f\ xs| = |xs|$, alors $\forall n\ xs,\ f\ (\mathsf{take}\ n\ xs) \simeq \mathsf{take}\ n\ (f\ xs)$.

Démonstration. On remarque d'abord que $f\ (\mathsf{take}\ n\ xs) \preceq f\ xs$, par monotonie de f, et que $\mathsf{take}\ n\ (f\ xs) \preceq f\ xs$. Il suffit donc de montrer que ces deux préfixes de $f\ xs$ sont de même longueur. Soit $m = |xs| = |f\ xs|$. Si $m \geq n$ alors $|\mathsf{take}\ n\ (f\ xs)| = n$ et $|\mathsf{take}\ n\ xs| = n$, d'où $|f\ (\mathsf{take}\ n\ xs)| = n$. Si $m < n$ alors $|\mathsf{take}\ n\ (f\ xs)| = m$ et $|\mathsf{take}\ n\ xs| = m$, d'où $|f\ (\mathsf{take}\ n\ xs)| = m$. □

4.2.1 Application à Vélus

Dans un premier temps, nous démontrons que chaque opérateur de notre sémantique synchrone commute bien avec take.

Propriété 4.3 (Commutativité du préfixe et des opérateurs synchrones)**.**

1. $\forall c\ bs\ n,\ \mathsf{take}\ n\ (\mathsf{const}\ c\ bs) \simeq \mathsf{const}\ c\ (\mathsf{take}\ n\ bs)$
2. $\forall s\ n,\ \mathsf{take}\ n\ (\mathsf{lift}^{\diamond}\ s) \simeq \mathsf{lift}^{\diamond}\ (\mathsf{take}\ n\ s)$
3. $\forall s_1\ s_2\ n,\ \mathsf{take}\ n\ (\mathsf{lift}^{\oplus}\ s_1\ s_2) \simeq \mathsf{lift}^{\oplus}\ (\mathsf{take}\ n\ s_1)\ (\mathsf{take}\ n\ s_2)$
4. $\forall s_1\ s_2\ n,\ \mathsf{take}\ n\ (\mathsf{fby}\ s_1\ s_2) \simeq \mathsf{fby}\ (\mathsf{take}\ n\ s_1)\ (\mathsf{take}\ n\ s_2)$
5. $\forall s_c\ s\ n,\ \mathsf{take}\ n\ (\mathsf{when}_C\ s_c\ s) \simeq \mathsf{when}_C\ (\mathsf{take}\ n\ s_c)\ (\mathsf{take}\ n\ s)$
6. $\forall s_c\ ss\ n,\ \mathsf{take}\ n\ (\mathsf{merge}\ s_c\ ss) \simeq \mathsf{merge}\ (\mathsf{take}\ n\ s_c)\ ((\mathsf{take}\ n)^{\uparrow}\ ss)$
7. $\forall s_c\ ss\ n,\ \mathsf{take}\ n\ (\mathsf{case}\ s_c\ ss) \simeq \mathsf{case}\ (\mathsf{take}\ n\ s_c)\ ((\mathsf{take}\ n)^{\uparrow}\ ss)$
8. $\forall\ s_c\ s_d\ ss\ n,$
   $\mathsf{take}\ n\ (\mathsf{case}'\ s_c\ s_d\ ss) \simeq \mathsf{case}'\ (\mathsf{take}\ n\ s_c)\ (\mathsf{take}\ n\ s_d)\ ((\mathsf{take}\ n)^{\uparrow}\ ss)$
9. $\forall f\ s_r\ s\ n$, si $\forall s\ m,\ \mathsf{take}\ m\ (f\ s) \simeq f\ (\mathsf{take}\ m\ s)$ alors
   $\mathsf{take}\ n\ (\mathsf{sreset}_f\ s_r\ s) \simeq \mathsf{sreset}_f\ (\mathsf{take}\ n\ s_r)\ (\mathsf{take}\ n\ s)$

La preuve du cas n° 4 est assez délicate en raison de la complexité de l'encodage du fby dans notre modèle, avec ses quatre fonctions mutuellement récursives, cf. section 3.3.3. La preuve du n° 5 en revanche, est immédiate car when est défini comme instance de zip. Nous utilisons ensuite ces propriétés pour étendre le résultat à la dénotation d'une expression.

Lemme 4.11. Pour tout entier n, expression bien typée e et environnements env_G, env_I, env, si $\forall f\ s\ m,\ \mathsf{take}\ m\ (env_G\ f\ s) \simeq env_G\ f\ (\mathsf{take}\ m\ s)$ alors $\mathsf{take}\ n\ (\mathsf{denot}_{\mathsf{E}}\ e\ env_G\ env_I\ env) \simeq \mathsf{denot}_{\mathsf{E}}\ e\ env_G\ (\mathsf{take}\ n\ env_I)\ (\mathsf{take}\ n\ env)$.

La preuve se fait très naturellement par récurrence sur la structure de l'expression et par réécriture des équations de la propriété 4.3 sauf dans le cas des variables, dont la preuve est faite par réflexivité. L'hypothèse de bon typage est importante : la dénotation d'une expression mal typée peut être $\mathsf{err}_{\mathsf{ty}}{}^{\omega}$, or $\mathsf{take}\ n\ \mathsf{err}_{\mathsf{ty}}{}^{\omega} \not\simeq \mathsf{err}_{\mathsf{ty}}{}^{\omega}$. En suit le même résultat, sur les nœuds.

Lemme 4.12. Pour tout entier n, nœud bien typée nd et environnements env_G, env_I, env, si $\forall f\ s\ m$, $\mathsf{take}\ m\ (env_G\ f\ s) \simeq env_G\ f\ (\mathsf{take}\ m\ s)$ alors $\mathsf{take}\ n\ (\mathsf{denot_N}\ nd\ env_G\ env_I\ env) \simeq \mathsf{denot_N}\ nd\ env_G\ (\mathsf{take}\ n\ env_I)\ (\mathsf{take}\ n\ env)$.

La dernière étape consiste à prouver le résultat sur la dénotation d'un programme G. Cette fois, plutôt que d'utiliser une récurrence sur l'ordre de déclaration des nœuds du programme, nous utilisons le principe d'induction continue sur le calcul de l'environnement global $env_G := \mathsf{denot}\ G = \mathsf{fixp}(\mathsf{denot_G}\ G)$. Les trois obligations générées par l'application du principe sont données ici sous forme de prémisses d'une règle d'inférence.

$$\frac{\begin{array}{r}\text{admissible } (\lambda env_G.\ \forall f\ s\ m,\ \mathsf{take}\ m\ (env_G\ f\ s) \simeq env_G\ f\ (\mathsf{take}\ m\ s)) \\ \forall n,\ \mathsf{take}\ n\ \bot \simeq \bot \\ \forall env_G,\ (\forall f\ s\ m,\ \mathsf{take}\ m\ (env_G\ f\ s) \simeq env_G\ f\ (\mathsf{take}\ m\ s)) \rightarrow \\ \forall f\ env_I\ n,\ \mathsf{take}\ n\ (\mathsf{fixp}\ (\mathsf{denot_N}\ (G.f)\ env_G\ env_I)) \\ \simeq \mathsf{fixp}\ (\mathsf{denot_N}\ (G.f)\ env_G\ (\mathsf{take}\ n\ env_I))\end{array}}{\forall f\ env_I\ n,\ \mathsf{take}\ n\ (\mathsf{denot}\ G\ f\ env_I) \simeq \mathsf{denot}\ G\ f\ (\mathsf{take}\ n\ env_I)}$$

Les deux premières, soit l'admissibilité de la propriété et son applicabilité à l'environnement vide initial, sont facilement démontrables. La troisième, correspondant à l'étape d'induction, est obtenue en déroulant la définition de $\mathsf{denot_G}$, qui est lui-même un point fixe de $\mathsf{denot_N}$, cf. section 3.5.2. Cette obligation fournit l'hypothèse sur env_G nécessaire à l'utilisation du lemme 4.12. Nous ne pouvons appliquer l'induction continue simultanément sur les deux points fixes car leurs arguments sont différents : env_I d'une part et $\mathsf{take}\ n\ env_I$ d'autre part. Nous scindons l'obligation en deux sous-buts que nous prouvons par induction continue sur les membres gauches, et en utilisant le lemme 4.12 :

— $\forall f\ env_I\ n,\ \mathsf{take}\ n\ (\mathsf{fixp}\ (\mathsf{denot_N}\ (G.f)\ env_G\ env_I)) \preceq \mathsf{fixp}\ (\mathsf{denot_N}\ (G.f)\ env_G\ (\mathsf{take}\ n\ env_I))$

— $\forall f\ env_I\ n,\ \mathsf{fixp}\ (\mathsf{denot_N}\ (G.f)\ env_G\ (\mathsf{take}\ n\ env_I)) \preceq \mathsf{take}\ n\ (\mathsf{fixp}\ (\mathsf{denot_N}\ (G.f)\ env_G\ env_I))$.

Théorème 4.13. Pour tout entier n, programme bien typé G, identifiant f et environnement env_I, $\mathsf{take}\ n\ (\mathsf{denot}\ G\ f\ env_I) \simeq \mathsf{denot}\ G\ f\ (\mathsf{take}\ n\ env_I)$.

4.2.2 Discussion

Nous avons démontré que tout programme Lustre bien typé satisfait la commutativité du préfixe. pré-requis à la preuve de correspondance entre les deux modèles synchrones énoncée au chapitre précédent. Nous avons employé le principe d'induction continue fourni par la bibliothèque CPO. En plus d'être agréable à l'utilisation, ce principe permet de mener un raisonnement sémantique au niveau du programme, et non syntaxique, comme le requière la preuve d'infinitude des flots ou la plupart des preuves de correction des passes de compilation. Cette section représente environ 800 lignes de code.

4.3 Indépendance à l'absence initiale

Un autre pré-requis aux théorèmes du chapitre précédent est qu'une fonction f associée à un nœud Lustre ne réagisse pas lorsque son entrée est absente initialement, soit :

$$\forall xs,\ f\ (\mathsf{abs} \cdot xs) \simeq \mathsf{abs} \cdot (f\ xs),$$

et dans le cas général d'un nœud à n entrées et m sorties :

$$\forall xss,\ f\ (\mathsf{abs} \cdot xss_1, \ldots, \mathsf{abs} \cdot xss_n) \simeq (\underbrace{\mathsf{abs}, \ldots, \mathsf{abs}}_{m}) \cdot (f\ xss).$$

Notons que cette propriété n'est pas vérifiée dans tous les langages synchrones. L'exemple suivant, notamment, est valide en Lucid Synchrone.

```
let clock half = h where rec h = false fby not h

let node stutter(x) = o where
  rec o = merge half x ((true fby o) whenot half)
```

Le flot booléen `half` alterne entre une valeur fausse et une valeur vraie, tandis que le nœud `stutter` propage son entrée lorsqu'elle est présente et la répète, via la mémoire du `fby`, lorsqu'elle est absente. Les annotations de type d'horloge inférées par le compilateur sont les suivantes :

```
val half :: 'a
val stutter :: 'a on half -> 'a
```

L'argument de `stutter` est donc attendu seulement lorsque `half` est vrai, tandis que sa sortie est présente à chaque instant où `half` est présent. Le flot `half` étant faux initialement, cet exemple viole la condition d'indépendance à l'absence initiale.

En Lustre en revanche, le typage des horloges est moins expressif et n'autorise pas la dépendance des horloges sur des définitions non locales, il est obligatoire que l'un des arguments du nœud en définisse l'horloge de base. Par conséquent, si tous les arguments du nœud sont absents, l'horloge de base du nœud est absente également. C'est précisément ce qui nous permet d'obtenir la propriété d'indépendance à l'absence initiale.

Dans notre modèle, où les fonctions de flots sont définies par le plus petit point fixe d'un ensemble d'équations, nous ne pouvons pas appliquer le principe d'induction continue pour prouver cette propriété. En effet, le cas initial de l'induction, où $f = \bot$, n'est pas vérifié : $\bot\ (\mathsf{abs} \cdot xs) \simeq \bot \not\simeq \mathsf{abs} \cdot \bot$.

Démontrer cette propriété reviendrait en fait à établir une sorte de productivité de la sémantique synchrone, restreinte au cas où les entrées sont absentes et cela nécessiterait, comme nous l'avons observé en section 4.1.2, un mécanisme de raisonnement sur la causalité du programme afin de vérifier qu'aucun inter-blocage n'empêche la production d'une valeur absente. Une façon d'éviter ces complications est de ne prouver qu'un seul côté de l'équivalence :

$$\forall xs,\ f\ (\mathsf{abs} \cdot xs) \preceq \mathsf{abs} \cdot (f\ xs).$$

Ce résultat se démontre par induction continue sur f, exactement de la même manière que la commutativité du préfixe décrite à la section précédente. Nous employons ici la notation $\mathsf{abs}\cdot env_I$ pour désigner l'ajout d'un élément abs en tête de tous les flots de l'environnement env_I.

Théorème 4.14. Pour tout programme bien typé G, identifiant f et environnement env_I, $\mathsf{denot}\ G\ f\ (\mathsf{abs}\cdot env_I) \preceq \mathsf{abs}\cdot(\mathsf{denot}\ G\ f\ env_I)$.

Il est ensuite possible de combiner ce résultat et celui d'infinitude des flots donné au théorème 4.7 afin d'établir une équivalence $\simeq$ grâce au principe de « préfixe infini ». L'ensemble des énoncés et preuves en Coq de cette section représente environ 500 lignes de code.

4.4 Sûreté du typage

Il nous reste à démontrer l'alignement des abs sur les flots d'entrée et de sortie, ainsi que l'absence de valeurs erronées dans les flots calculés. Ces deux propriétés sont en fait des conséquences, tout au moins en partie, de la sûreté du typage, qui inclut notamment la correction du système d'horloges. Le langage pris en charge par Vélus est équipé des deux systèmes de types propres à son schéma de compilation, l'un assurant la cohérence des données et l'autre celle des horloges. Nous prouvons ici la sûreté de ces deux systèmes vis-à-vis de notre sémantique synchrone dénotationnelle. En faisant le lien entre les annotations de typage et les propriétés des environnements d'évaluation, nous montrons qu'un programme bien typé ne peut produire que des flots respectant ces annotations.

4.4.1 Adéquation des systèmes de types

L'exemple suivant illustre le fonctionnement général du typage de Vélus.

```
node count_when (c : bool; n : int) returns (o : bool)
var v : int; t : int when c;
let
  t = countdown ((n, false) when c);
  v = merge c (true => t) (false => 0 when not c);
  o = v >= 0;
tel
```

Dans ce nœud les variables c et o sont déclarées de type booléen, un type énuméré à deux constructeurs, et les variables n, v et t de type entier. L'absence d'annotation de type d'horloge pour c, n, o et v signifie qu'elles sont par défaut sur l'horloge de base $\bullet$, tandis que t est explicitement déclarée d'horloge $\bullet$ on $\mathsf{T}(c)$.

Ces informations sont véhiculées par un environnement de typage Γ obtenu lors des analyses statiques :

$$\Gamma = \Gamma_{\mathrm{E}} \uplus \Gamma_{\mathrm{S}} : \mathsf{ident} \to \mathsf{type} * \mathsf{clock}.$$

Il est formé de l'environnement Γ_{E} des variables d'entrée et de Γ_{S}, celui des variables locales et de sortie, qui dans notre exemple satisfont les contraintes suivantes :

$$\begin{aligned} \Gamma_{\mathrm{E}}(c) &= (\mathtt{bool}, \bullet) & \Gamma_{\mathrm{S}}(o) &= (\mathtt{bool}, \bullet) \\ \Gamma_{\mathrm{E}}(n) &= (\mathtt{int}, \bullet) & \Gamma_{\mathrm{S}}(v) &= (\mathtt{int}, \bullet) \\ & & \Gamma_{\mathrm{S}}(t) &= (\mathtt{int}, \bullet \ \mathsf{on}\ \mathsf{T}(c)). \end{aligned}$$

L'environnement env : $\mathsf{ident} \to \mathsf{Stream}_\epsilon\ \mathsf{sevalue}$ obtenu en interprétant le nœud sur un environnement d'entrées env_I est en adéquation avec Γ si les flots qu'il contient respectent les annotations des variables. Dans notre exemple, les valeurs du flot $env(o)$ doivent être booléennes et celles de $env(v)$ et $env(t)$ doivent être entières. De plus, les valeurs de $env(o)$ et $env(v)$ doivent être présentes seulement lorsque l'horloge de base est vraie et celles de $env(t)$ seulement lorsque $env_I(c)$ est vrai, comme dans l'instance suivante.

$env_I(c)$	F	T	T	abs	T	F	F	T	T	abs	T
$env_I(n)$	3	3	3	abs	3	3	3	3	3	abs	3
horloge de base $\bullet$	T	T	T	F	T	T	T	T	T	F	T
$env(t) : \bullet$ on $\mathsf{T}(c)$	abs	3	2	abs	1	abs	abs	0	-1	abs	-2
$env(v) : \bullet$	0	3	2	abs	1	0	0	0	-1	abs	-2
$env(o) : \bullet$	T	T	T	abs	T	T	T	T	F	abs	F

Le typage dépendant des horloges nécessite donc d'interpréter les annotations afin d'en vérifier l'adéquation. Notons que dans notre exemple l'annotation d'horloge de t ne dépend que de l'entrée c, mais qu'en général, le typage autorise aussi les dépendances sur des variables locales ou de sortie.

4.4.2 Interprétation d'horloges

Nous distinguons deux notions d'horloge : celle d'un flot échantillonné et celle calculée à partir d'une annotation de typage. Les deux notions coïncident lorsque la sûreté du typage est établie.

Définition 4.1. On appelle *horloge abstraite* d'un flot échantillonné le flot booléen défini par la fonction suivante :

$$\begin{aligned} &\mathsf{clock\text{-}of} : \mathsf{Stream}_\epsilon\ \mathsf{sevalue} \to_c \mathsf{Stream}_\epsilon\ \mathsf{bool} \\ &\qquad := \mathsf{map}\ (\lambda v \to \mathsf{T} \mid \mathsf{abs} \mid \mathsf{err} \to \mathsf{F}). \end{aligned}$$

Cette définition est parfois appelée simplement « l'horloge d'un flot » mais nous préférons ici la différencier clairement de toute notion d'horloge statique.

Définition 4.2. On calcule le *flot d'horloge de base* d'un environnement *env* défini sur une liste l de variables en utilisant l'opérateur d'horloge abstraite :

$$\mathsf{base\text{-}of} : \mathsf{list\ ident} \to \mathsf{SEnv} \to_c \mathsf{Stream}_\epsilon\ \mathsf{bool}$$
$$\mathsf{base\text{-}of}\ [\,]\ env := \mathsf{F}^\omega$$
$$\mathsf{base\text{-}of}\ (x :: l)\ env := \mathsf{zip}\ (\vee)\ (\mathsf{clock\text{-}of}\ env(x))\ (\mathsf{base\text{-}of}\ l\ env).$$

Dans la suite nous ignorerons le premier paramètre et noterons simplement base-of *env*. C'est cette même fonction base-of qui est utilisée dans la dénotation des constantes du langage, cf. figure 3.3, activées lorsque l'une au moins des entrées du nœud est présente.

Notons que l'opérateur zip utilisé ici est bloquant, comme tout opérateur sur les flots de Kahn. Pour déterminer la valeur de l'horloge de base il faut, à chaque instant, extraire la tête de tous les flots de la liste. Par conséquent, si l'un au moins des flots de *env* est vide, alors base-of $env = \bot$.

Définition 4.3. On définit *l'interprétation d'une annotation d'horloge* dans un environnement (bs, env_I, env), où bs est un flot de booléens, comme suit :

$$\mathsf{denot_C} : \mathsf{clock} \to \mathsf{Stream}_\epsilon\ \mathsf{bool} \to \mathsf{SEnv} \to \mathsf{SEnv} \to \mathsf{Stream}_\epsilon\ \mathsf{bool}$$
$$\mathsf{denot_C}\ \bullet\ bs\ env_I\ env := bs$$
$$\mathsf{denot_C}\ (ck\ \mathsf{on}\ C(x))\ bs\ env_I\ env :=$$
$$\begin{cases} \mathsf{zip}\ (\lambda v\ c.\ v = C \wedge c)\ env_I(x)\ (\mathsf{denot_C}\ ck\ bs\ env_I\ env) & \text{si } x \in \Gamma_\mathrm{E} \\ \mathsf{zip}\ (\lambda v\ c.\ v = C \wedge c)\ env(x)\ (\mathsf{denot_C}\ ck\ bs\ env_I\ env) & \text{si } x \in \Gamma_\mathrm{S}. \end{cases}$$

En reprenant l'instance du nœud `count_when`, on constate l'exacte correspondance entre l'horloge abstraite des flots, calculée avec clock-of, et l'interprétation de leur annotation d'horloge dans (base-of env_I, env_I, env).

$env_I(c)$	F	T	T	abs	T	F	F	T	T	abs	T
$env_I(n)$	3	3	3	abs	3	3	3	3	3	abs	3
$\mathsf{denot_C}\ \bullet = bs$	T	T	T	F	T	T	T	T	T	F	T
$\mathsf{denot_C}\ (\bullet\ \mathsf{on}\ \mathsf{T}(c))$	F	T	T	F	T	F	F	T	T	F	T
$env(t)$	abs	3	2	abs	1	abs	abs	0	-1	abs	-2
clock-of $env(t)$	F	T	T	F	T	F	F	T	T	F	T
$\vdots$											

Il n'est en général pas possible d'exiger l'équivalence entre l'horloge abstraite et le flot associé à l'annotation, car elle n'est pas vérifiée tout au long du calcul du point fixe des équations. Initialement par exemple, l'environnement env_I des entrées peut contenir des flots arbitrairement longs tandis que l'environnement *env* est initialisé à $\bot$. Une relation de préfixe, en revanche, suffit pour établir la correction du système d'horloges. On suppose donc, pour toute variable x déclarée d'horloge ck :

$$\mathsf{clock\text{-}of}\ env(x) \preceq \mathsf{denot_C}\ ck\ bs\ env_I\ env.$$

Notons qu'en général, la relation opposée ne tient pas : par définition, ni les variables d'entrée ni les constantes du langage ne peuvent « dépasser » l'horloge de base. Nous démontrerons que cette relation est satisfaite à toute étape du calcul de la dénotation d'un programme bien cadencé, mais dans le contexte plus large de la preuve d'adéquation des environnements d'évaluation avec les environnements de typage. Cela passe notamment par le typage des données des flots.

4.4.3 Sûreté du typage des données

Les valeurs des flots de Vélus sont toutes du type value, désignant des scalaires ou des constantes énumérées. Le prédicat `wt_value` est défini pour garantir le bon typage d'une valeur vis-à-vis d'une annotation de type.

```
Inductive wt_value : value → type → Prop :=
| WtS: ∀c t, wt_cvalue v t → wt_value (Vscalar v) (Tprimitive t)
| WtE: ∀n tx tn, n < length tn → wt_value (Venum n) (Tenum tx tn).
```

Le typage d'un scalaire est assuré par un prédicat `wt_cvalue`, sur les valeurs et annotations de CompCert. Le typage d'une constante énumérée, représentée par un entier n, garantit qu'elle ne dépasse pas le nombre de constructeurs du type auquel elle appartient. Nous utilisons ce prédicat pour définir le bon typage des valeurs d'un flot.

```
Definition wt_stream ty : Streamϵ sevalue → Prop :=
  Forallϵ (λ v ⇒ match v with
                 | pres v ⇒ wt_value v ty
                 | abs | err _ ⇒ ⊤
                 end).
```

Avec cette définition, toute valeur absente ou erronée est considérée comme bien typée. Cela nous permet d'établir, via les règles données en figure 4.1 ci-contre, la préservation du typage des données par les opérateurs synchrones. La pré-condition sur les annotations ty_1 et ty_2 dans les règles des opérateurs arithmétiques est identique à celle du typage statique et permet de garantir la compatibilité avec l'opérateur $\diamond$ ou $\oplus$. Nous ne donnons pas de règle pour l'opérateur sreset car ce dernier requiert une pré-condition plus élaborée sur le comportement de la fonction à réinitialiser, que nous décrirons plus tard.

4.4.4 Sûreté du typage d'horloges

Les règles relatives au typage d'horloge sont plus délicates à énoncer en raison du traitement des erreurs. Considérons l'instance suivante.

$s_c := env(c) : \bullet$	T	T	F	T	F	T	T
$s_1 : \bullet$ on T(c)	1	2	abs	4	abs	6	7
$s_2 : \bullet$ on F(c)	abs	abs	3	abs	5	$\mathsf{err}_{\mathsf{ty}}$	abs
$s :=$ merge s_c $[s_1; s_2] : \bullet$	1	2	3	4	5	$\mathsf{err}_{\mathsf{ty}}$	7

$$\frac{\mathsf{wt\text{-}value}\ c\ ty}{\mathsf{wt\text{-}stream}\ ty\ (\mathsf{const}\ c\ bs)} \qquad \frac{\mathsf{wt\text{-}stream}\ ty_1\ s \qquad \vdash \diamond_{ty_1} : ty}{\mathsf{wt\text{-}stream}\ ty\ (\mathsf{lift}^{\diamond}\ s)}$$

$$\frac{\mathsf{wt\text{-}stream}\ ty_1\ s_1 \qquad \mathsf{wt\text{-}stream}\ ty_2\ s_2 \qquad \vdash \oplus_{ty_1 \times ty_2} : ty}{\mathsf{wt\text{-}stream}\ ty\ (\mathsf{lift}^{\oplus}\ s_1\ s_2)}$$

$$\frac{\mathsf{wt\text{-}stream}\ ty\ s_1 \qquad \mathsf{wt\text{-}stream}\ ty\ s_2}{\mathsf{wt\text{-}stream}\ ty\ (\mathsf{fby}\ s_1\ s_2)}$$

$$\frac{C < \texttt{length tn} \qquad \mathsf{wt\text{-}stream}\ (\texttt{Tenum tx tn})\ s_c \qquad \mathsf{wt\text{-}stream}\ ty\ s}{\mathsf{wt\text{-}stream}\ ty\ (\mathsf{when}_C\ s_c\ s)}$$

$$\frac{\mathsf{wt\text{-}stream}\ (\texttt{Tenum tx tn})\ s_c \qquad \forall i, \mathsf{wt\text{-}stream}\ ty\ ss_i}{\mathsf{wt\text{-}stream}\ ty\ (\mathsf{merge}\ s_c\ ss)}$$

$$\frac{\mathsf{wt\text{-}stream}\ (\texttt{Tenum tx tn})\ s_c \qquad \forall i, \mathsf{wt\text{-}stream}\ ty\ ss_i}{\mathsf{wt\text{-}stream}\ ty\ (\mathsf{case}\ s_c\ ss)}$$

$$\frac{\mathsf{wt\text{-}stream}\ (\texttt{Tenum tx tn})\ s_c \qquad \mathsf{wt\text{-}stream}\ ty\ s_d \qquad \forall i, \mathsf{wt\text{-}stream}\ ty\ ss_i}{\mathsf{wt\text{-}stream}\ ty\ (\mathsf{case}'\ s_c\ s_d\ ss)}$$

FIGURE 4.1 – Préservation du typage des données par les opérateurs synchrones

En prenant les définitions de la section 4.4.2, nous avons $\mathsf{clock\text{-}of}\ s_c \preceq \mathsf{denot_C}\ \bullet$, $\mathsf{clock\text{-}of}\ s_1 \preceq \mathsf{denot_C}\ (\bullet\ \mathsf{on}\ \mathsf{T}(c))$ et $\mathsf{clock\text{-}of}\ s_2 \preceq \mathsf{denot_C}\ (\bullet\ \mathsf{on}\ \mathsf{F}(c))$, car l'horloge abstraite d'un élément err vaut F. Les flots s_c, s_1 et s_2 respectent donc leur annotation d'horloge. Cependant, en raison de l'erreur survenant sur s_2, nous avons $\mathsf{clock\text{-}of}\ s \not\preceq \mathsf{denot_C}\ \bullet$. Nous ne pouvons donc pas traiter les erreurs de façon transparente comme avec le typage des données, et devons plutôt supposer qu'aucune erreur n'intervient lors du calcul.

Considérons aussi l'instance suivante.

$s_1 : \bullet$	16	16	16	16	16	16	16
$s_2 : \bullet$	16	8	4	2	1	0	-1
$s := \mathsf{lift}^{\mathrm{div}}\ s_1\ s_2 : \bullet$	1	2	4	8	16	$\mathsf{err_{rt}}$	-16

Les flots s_1 et s_2 sont bien échantillonnés sur l'horloge de base, mais le résultat s ne l'est pas. Contrairement au cas de figure précédent, il ne suffit pas de supposer l'absence d'erreur dans les flots des opérandes, il faut aussi supposer que l'opération arithmétique n'échoue pas. Nous utilisons à cet effet les prédicat $\mathsf{safe_{rt}} := \mathsf{Forall}_\epsilon\ (\lambda e.e \neq \mathsf{err_{rt}})$ et $\mathsf{safe} := \mathsf{Forall}_\epsilon\ (\lambda e.e \neq \mathsf{err})$.

La figure 4.2 ci-contre donne les règles d'alignement des horloges abstraites des opérateurs synchrones. Pour améliorer la lisibilité, nous omettons les arguments bs, env_I, et env de la fonction $\mathsf{denot_C}$.

4.4.5 Adéquation des environnements

Les propriétés de sûreté des deux systèmes de types que nous avons décrites précédemment sont admissibles. Nous les combinons dans l'objectif de prouver par induction continue la sûreté du typage sur les environnements issus de la dénotation des programmes.

Définition 4.4. On dit qu'un environnement dénotationnel (bs, env_I, env) est en adéquation avec un environnement de typage $\Gamma = \Gamma_\mathrm{E} \uplus \Gamma_\mathrm{S}$ et on note :

$$\mathsf{wf\text{-}env}\ \Gamma\ bs\ env_I\ env$$

si, pour toute variable x telle que $\Gamma(x) = (ty, ck)$ et pour tout flot $s_x := env_I(x)$ si $x \in \Gamma_\mathrm{E}$ ou $s_x := env(x)$ si $x \in \Gamma_\mathrm{S}$,

1. $\mathsf{wt\text{-}stream}\ ty\ s_x$
2. $\mathsf{clock\text{-}of}\ s_x \preceq \mathsf{denot_C}\ ck\ bs\ env_I\ env$
3. $\mathsf{safe}\ s_x$.

Autrement dit, tous les flots de l'environnement doivent être sans erreurs et respecter leur annotation de type et d'horloge dans Γ.

Le choix du flot bs à l'instanciation du prédicat d'adéquation est très important. Considérons par exemple le programme suivant, bien typé dans un environnement Γ tel que $\Gamma(x) = \Gamma(y) = \Gamma(z) = (\texttt{int}, \bullet)$.

```
node f (x y : int) return (z : int)
let
  z = x;
tel
```

Prenons un environnement d'entrée env_I tel que $env_I(x) = 1^\omega$ et $env_I(y) = \bot$. Le point fixe de la dénotation du nœud f est un environnement env tel que $env(z) = 1^\omega$. Or, comme nous le remarquions en section 4.4.2, nous avons $\mathsf{base\text{-}of}\ env_I = \bot$, il est donc impossible d'instancier bs par $\mathsf{base\text{-}of}\ env_I$ pour obtenir $\mathsf{wf\text{-}env}\ \Gamma\ (\mathsf{base\text{-}of}\ env_I)\ env_I\ env$, car $\mathsf{clock\text{-}of}\ env(z) \not\preceq \bot$.

Notre solution consiste à quantifier sur l'existence d'une *complétion* du flot d'horloge de base satisfaisant le prédicat d'adéquation :

$$\exists bs,\ \mathsf{base\text{-}of}\ env_I \preceq bs \wedge \mathsf{wf\text{-}env}\ \Gamma\ bs\ env_I\ env.$$

Nous utilisons cette caractérisation pour démontrer la préservation de l'adéquation des environnements par l'opération de réinitialisation modulaire.

$$\frac{bs \preceq bs'}{\mathsf{clock\text{-}of}\ (\mathsf{const}\ c\ bs) \preceq bs'} \qquad \frac{\mathsf{clock\text{-}of}\ s \preceq \mathsf{denot_C}\ ck \qquad \mathsf{safe_{rt}}\ (\mathsf{lift}^{\diamond}\ s)}{\mathsf{clock\text{-}of}\ (\mathsf{lift}^{\diamond}\ s) \preceq \mathsf{denot_C}\ ck}$$

$$\frac{\begin{array}{c}\mathsf{clock\text{-}of}\ s_1 \preceq \mathsf{denot_C}\ ck \qquad \mathsf{safe_{rt}}\ (\mathsf{lift}^{\oplus}\ s_1\ s_2)\\ \mathsf{clock\text{-}of}\ s_2 \preceq \mathsf{denot_C}\ ck\end{array}}{\mathsf{clock\text{-}of}\ (\mathsf{lift}^{\oplus}\ s_1\ s_2) \preceq \mathsf{denot_C}\ ck}$$

$$\frac{\begin{array}{ll}\mathsf{clock\text{-}of}\ s_1 \preceq \mathsf{denot_C}\ ck & \mathsf{safe}\ s_1\\ \mathsf{clock\text{-}of}\ s_2 \preceq \mathsf{denot_C}\ ck & \mathsf{safe}\ s_2\end{array}}{\mathsf{clock\text{-}of}\ (\mathsf{fby}\ s_1\ s_2) \preceq \mathsf{denot_C}\ ck}$$

$$\frac{\begin{array}{ll}\mathsf{clock\text{-}of}\ s \preceq \mathsf{denot_C}\ ck & \mathsf{safe}\ s\\ \mathsf{clock\text{-}of}\ env_I(x) \preceq \mathsf{denot_C}\ ck & \mathsf{safe}\ env_I(x) \qquad x \in \Gamma_{\mathrm{E}}\end{array}}{\mathsf{clock\text{-}of}\ (\mathsf{when}_C\ env_I(x)\ s) \preceq \mathsf{denot_C}\ (ck\ \mathsf{on}\ C(x))}$$

$$\frac{\begin{array}{ll}\mathsf{clock\text{-}of}\ s \preceq \mathsf{denot_C}\ ck & \mathsf{safe}\ s\\ \mathsf{clock\text{-}of}\ env(x) \preceq \mathsf{denot_C}\ ck & \mathsf{safe}\ env(x) \qquad x \in \Gamma_{\mathrm{S}}\end{array}}{\mathsf{clock\text{-}of}\ (\mathsf{when}_C\ env(x)\ s) \preceq \mathsf{denot_C}\ (ck\ \mathsf{on}\ C(x))}$$

$$\frac{\begin{array}{ll}\forall i, \mathsf{clock\text{-}of}\ ss_i \preceq \mathsf{denot_C}\ (ck\ \mathsf{on}\ C_i(x)) & \forall i, \mathsf{safe}\ ss_i\\ \mathsf{clock\text{-}of}\ env_I(x) \preceq \mathsf{denot_C}\ ck & \mathsf{safe}\ env_I(x) \qquad x \in \Gamma_{\mathrm{E}}\end{array}}{\mathsf{clock\text{-}of}\ (\mathsf{merge}\ env_I(x)\ ss) \preceq \mathsf{denot_C}\ ck}$$

$$\frac{\begin{array}{ll}\forall i, \mathsf{clock\text{-}of}\ ss_i \preceq \mathsf{denot_C}\ (ck\ \mathsf{on}\ C_i(x)) & \forall i, \mathsf{safe}\ ss_i\\ \mathsf{clock\text{-}of}\ env(x) \preceq \mathsf{denot_C}\ ck & \mathsf{safe}\ env(x) \qquad x \in \Gamma_{\mathrm{S}}\end{array}}{\mathsf{clock\text{-}of}\ (\mathsf{merge}\ env(x)\ ss) \preceq \mathsf{denot_C}\ ck}$$

$$\frac{\begin{array}{ll}\forall i, \mathsf{clock\text{-}of}\ ss_i \preceq \mathsf{denot_C}\ ck & \forall i, \mathsf{safe}\ ss_i\\ \mathsf{clock\text{-}of}\ s_c \preceq \mathsf{denot_C}\ ck & \mathsf{safe}\ s_c\end{array}}{\mathsf{clock\text{-}of}\ (\mathsf{case}\ s_c\ ss) \preceq \mathsf{denot_C}\ ck}$$

$$\frac{\begin{array}{ll}\forall i, \mathsf{clock\text{-}of}\ ss_i \preceq \mathsf{denot_C}\ ck & \forall i, \mathsf{safe}\ ss_i\\ \mathsf{clock\text{-}of}\ s_c \preceq \mathsf{denot_C}\ ck & \mathsf{safe}\ s_c\\ \mathsf{clock\text{-}of}\ s_d \preceq \mathsf{denot_C}\ ck & \mathsf{safe}\ s_d\end{array}}{\mathsf{clock\text{-}of}\ (\mathsf{case}'\ s_c\ s_d\ ss) \preceq \mathsf{denot_C}\ ck}$$

FIGURE 4.2 – Horloge abstraite des opérateurs synchrones dans un environnement (bs, env_I, env)

Lemme 4.15. Pour toute fonction de flots f associée à un nœud Lustre bien typé dans un environnement Γ, pour tous flots bs, s_r et tout environnement env_I,

$$\begin{array}{rl}
\textbf{si} & \forall env'_I\ bs',\ \mathsf{base\text{-}of}\ env'_I \preceq bs' \wedge \mathsf{wf\text{-}env}\ \Gamma\ bs'\ env'_I\ \bot \rightarrow \\
& \qquad \mathsf{wf\text{-}env}\ \Gamma\ bs'\ env'_I\ (f\ env'_I) \\
\textbf{et} & \mathsf{base\text{-}of}\ env_I \preceq bs \\
\textbf{et} & \mathsf{wf\text{-}env}\ \Gamma\ bs\ env_I\ \bot \\
\textbf{alors} & \mathsf{wf\text{-}env}\ \Gamma\ bs\ env_I\ (\mathsf{sreset}_f\ s_r\ env_I).
\end{array}$$

Le dernier point à traiter avant de pouvoir démontrer l'adéquation des environnements dans notre modèle est celui des erreurs à l'exécution.

4.4.6 Programmes et expressions sans erreurs

L'absence d'erreurs lors de l'exécution d'opérations arithmétiques ou logiques ne peut pas être prouvée en général par la correction d'une analyse statique, car elle dépend de la dénotation du programme et de ses entrées. Nous introduisons donc un prédicat qui peut être vu comme une *obligation*, à démontrer pour chaque instance d'un programme G. Nous le donnons sous la forme de deux prédicats inductifs :

$$\begin{array}{l}
\mathsf{no\text{-}rte\text{-}exp}\ env_G\ env_I\ env : \mathsf{exp} \rightarrow \mathsf{Prop} \\
\mathsf{no\text{-}rte\text{-}node}\ env_G\ env_I\ env : \mathsf{node} \rightarrow \mathsf{Prop}.
\end{array}$$

Les règles d'introduction du prédicat no-rte-exp (*no run-time errors*) pour les expressions sont données en figure 4.3 ci-contre, tandis que l'unique règle s'appliquant au cas des nœuds consiste à supposer que toutes les expressions à droite des équations sont sans erreurs :

$$\frac{\forall (xs = es) \in n.\mathsf{eqs}, \forall i,\ \mathsf{no\text{-}rte\text{-}exp}\ env_I\ env\ es_i}{\mathsf{no\text{-}rte\text{-}node}\ env_I\ env\ n}$$

Les seules règles non triviales pour les expressions sont celles des opérations unaires et binaires, consistant à supposer que l'évaluation point à point d'un opérateur $\diamond$ ou $\oplus$ ne produit pas de valeur $\mathsf{err_{rt}}$ dans le contexte d'exécution (env_I, env). Les analyses décrites précédemment nous autorisent à supposer, par l'intermédiaire de l'hypothèse wt-stream, le bon typage des données des flots des sous-expressions. Enfin, nous enveloppons ces définitions dans un unique prédicat spécifiant l'absence d'erreurs arithmétiques ou logiques à l'exécution d'un programme G :

$$\frac{\begin{array}{c}\mathsf{no\text{-}rte\text{-}node}\ env_I\ (\mathsf{denot}\ G\ f\ env_I)\ G.f \\ \forall f' \neq f,\ \forall env'_I,\ \mathsf{no\text{-}rte\text{-}node}\ env'_I\ (\mathsf{denot}\ G\ f'\ env'_I)\ G.f'\end{array}}{\mathsf{no\text{-}run\text{-}time\text{-}errors}\ G\ f\ env_I}$$

$$\frac{}{\mathsf{no\text{-}rte\text{-}exp}\ c} \qquad \frac{}{\mathsf{no\text{-}rte\text{-}exp}\ C} \qquad \frac{}{\mathsf{no\text{-}rte\text{-}exp}\ x}$$

$$\frac{\begin{array}{c}\mathsf{no\text{-}rte\text{-}exp}\ env_G\ e \\ (\mathsf{wt\text{-}stream}\ (\mathsf{type\text{-}of}\ e)\ (\mathsf{denot}_{\mathsf{E}}\ e\ env_G\ env_I\ env) \rightarrow \\ \mathsf{safe}_{\mathsf{rt}}\ (\mathsf{denot}_{\mathsf{E}}\ (\diamond\, e)\ env_G\ env_I\ env))\end{array}}{\mathsf{no\text{-}rte\text{-}exp}\ (\diamond\, e)}$$

$$\frac{\begin{array}{c}\mathsf{no\text{-}rte\text{-}exp}\ e_1 \qquad \mathsf{no\text{-}rte\text{-}exp}\ e_2 \\ (\mathsf{wt\text{-}stream}\ (\mathsf{type\text{-}of}\ e_1)\ (\mathsf{denot}_{\mathsf{E}}\ e_1\ env_G\ env_I\ env) \\ \wedge\, \mathsf{wt\text{-}stream}\ (\mathsf{type\text{-}of}\ e_2)\ (\mathsf{denot}_{\mathsf{E}}\ e_2\ env_G\ env_I\ env) \rightarrow \\ \mathsf{safe}_{\mathsf{rt}}\ (\mathsf{denot}_{\mathsf{E}}\ (e_1 \oplus e_2)\ env_G\ env_I\ env))\end{array}}{\mathsf{no\text{-}rte\text{-}exp}\ (e_1 \oplus e_2)}$$

$$\frac{\mathsf{no\text{-}rte\text{-}exp}\ e}{\mathsf{no\text{-}rte\text{-}exp}\ (e\ \texttt{when}\ x)} \qquad \frac{\forall i, \mathsf{no\text{-}rte\text{-}exp}\ es_i}{\mathsf{no\text{-}rte\text{-}exp}\ (\texttt{merge}\ x\ es)}$$

$$\frac{\mathsf{no\text{-}rte\text{-}exp}\ e \qquad \forall i, \mathsf{no\text{-}rte\text{-}exp}\ es_i}{\mathsf{no\text{-}rte\text{-}exp}\ (\texttt{case}\ e\ es)}$$

$$\frac{\mathsf{no\text{-}rte\text{-}exp}\ e \qquad \mathsf{no\text{-}rte\text{-}exp}\ e_d \qquad \forall i, \mathsf{no\text{-}rte\text{-}exp}\ es_i}{\mathsf{no\text{-}rte\text{-}exp}\ (\texttt{case}\ e\ e_d\ es)} \qquad \frac{\forall i, \mathsf{no\text{-}rte\text{-}exp}\ es_i}{\mathsf{no\text{-}rte\text{-}exp}\ (f\ es)}$$

$$\frac{\mathsf{no\text{-}rte\text{-}exp}\ e_r \qquad \forall i, \mathsf{no\text{-}rte\text{-}exp}\ es_i}{\mathsf{no\text{-}rte\text{-}exp}\ ((\texttt{restart}\ f\ \texttt{every}\ e_r)\ es)}$$

FIGURE 4.3 – Règles du prédicat no-rte-exp
dans un environnement (env_G, env_I, env)

L'idée de cette dernière règle est d'instancier un prédicat no-rte-node pour chaque nœud du programme, avec comme environnement local *env* le résultat de la dénotation du nœud. Les entrées du nœud principal f (le *main* du programme) sont passées en paramètre, tandis que celles des autres nœuds sont quantifiées universellement, définissant ainsi une sur-approximation des programmes sans erreurs.

Nous verrons d'une part, en section suivante, que la propriété définie par ces différents prédicats est suffisante pour exclure les erreurs $\mathsf{err}_{\mathsf{rt}}$ et ainsi conclure sur la sûreté des exécutions. D'autre part, nous décrirons au chapitre suivant une procédure permettant de valider statiquement cette propriété sur une large classe de programmes.

La forme du prédicat no-run-time-errors n'est toutefois pas entièrement satisfaisante, car il est défini inductivement sur la syntaxe du programme et non sur sa dénotation. La preuve d'une instance de ce prédicat doit donc reposer sur les mécanismes syntaxiques parfois laborieux du compilateur et non sur le principe d'induction continue. Une façon de traiter ce problème serait de démontrer la relation suivante.

Conjecture 4.1. Pour tout nœud f d'un programme G et pour tout environnement env_I, $\mathsf{safe}_{\mathsf{rt}}\ (\mathsf{denot}\ G\ f\ env_I) \rightarrow \mathsf{no\text{-}run\text{-}time\text{-}errors}\ G\ f\ env_I$.

La preuve d'absences d'$\mathsf{err}_{\mathsf{rt}}$ se ferait donc plutôt par induction continue sur le point fixe de denot. Cela nécessiterait cependant d'inverser toutes les fonctions de flots utilisées dans la dénotation des expressions, là où les règles de no-rte-exp données en figure 4.3 sont, pour la plupart, triviales à appliquer. De plus, la conjecture 4.1 ne tient pas lorsque des variables locales d'un nœud instancié ne sont pas utilisées pour calculer ses sorties, car leur flot n'apparaît pas dans $\mathsf{denot}\ G\ f\ env_I$ alors que leur expression doit satisfaire no-rte-exp.

4.4.7 Conclusion

Nous avons à présent tous les éléments nécessaires pour démontrer, par induction continue, la sûreté du typage dans notre modèle. En combinant les propriétés décrites dans toute cette section et en supposant l'absence d'erreurs arithmétiques et logiques, nous démontrons que l'évaluation d'un nœud préserve l'adéquation des environnements.

Théorème 4.16. Pour tout nœud f d'un programme G, bien typé dans un environnement Γ, pour tout flot bs et tout environnement d'entrée env_I,

si $\mathsf{no\text{-}run\text{-}time\text{-}errors}\ G\ f\ env_I$
et $\mathsf{base\text{-}of}\ env_I \preceq bs$
et $\mathsf{wf\text{-}env}\ \Gamma\ bs\ env_I\ \bot$
alors $\mathsf{wf\text{-}env}\ \Gamma\ bs\ env_I\ (\mathsf{denot}\ G\ f\ env_I)$.

L'hypothèse d'adéquation initiale de l'environnement env_I, indispensable ici, est directement impliquée par l'hypothèse welltyped-inputs du principal théorème de correction du compilateur.

Chapitre 5

Conclusion et travaux futurs

5.1 Correction du compilateur

En encodant un modèle dénotationnel du noyau flot de données du langage d'entrée de Vélus, et en explicitant les conditions de sa correspondance avec le modèle relationnel existant, nous avons considérablement renforcé le principal théorème de correction du compilateur.

Théorème (correction, avant)**.**

si $\mathsf{compile}\ G\ f = \mathsf{OK}\ asm$
et $\mathsf{welltyped\text{-}inputs}\ G\ f\ xs$
et $G \vdash f(xs) \Downarrow ys$
alors $asm \Downarrow \langle \mathsf{Load}(xs(i)) \cdot \mathsf{Store}(ys(i)) \rangle_{i=0}^{\infty}$.

Théorème (correction, après)**.**

si $\mathsf{compile}\ G\ f = \mathsf{OK}\ asm$
et $\mathsf{welltyped\text{-}inputs}\ G\ f\ xs$
et $\mathsf{no\text{-}run\text{-}time\text{-}errors}\ G\ f\ xs$
alors $\exists ys, G \vdash f(xs) \Downarrow ys \wedge asm \Downarrow \langle \mathsf{Load}(xs(i)) \cdot \mathsf{Store}(ys(i)) \rangle_{i=0}^{\infty}$.

Notre nouveau théorème garantit l'existence de flots de sortie ys, instanciés en l'occurrence par $\lceil \mathsf{denot}\ G\ f\ xs \rceil$, qui satisfont le prédicat de sémantique relationnelle. En combinant cet énoncé et celui du déterminisme de la relation, démontré dans des travaux antérieurs [1], nous pouvons même garantir que ces ys sont les seuls flots à être mis en relation avec les xs par la sémantique du compilateur, et donc à correspondre à la trace du programme cible.

De par l'aspect constructif et donc potentiellement exécutable du modèle dénotationnel, il est théoriquement possible d'extraire du développement Coq un interprète de référence, permettant de tester l'exécution des programmes

[1] Bourke *et al.*, 2023a, §4.3

source. Les perspectives sont alors nombreuses : pouvoir répondre rapidement aux questions que l'on pourrait avoir sur le comportement d'un programme à flot de données, détecter des erreurs d'exécution qui invalideraient le théorème de correction, et même tester automatiquement et valider des passes de compilation d'un outil industriel [2]. Les performances insatisfaisantes du calcul de point fixe sont pour l'instant le principal obstacle à cette approche, mais plusieurs pistes d'amélioration sont envisageables. Nous pourrions par exemple introduire des algorithmes de mémoïsation, ou même énoncer l'équivalence de notre modèle avec une procédure co-itérative [3], plus efficace.

L'idée de disposer d'une sémantique exécutable et manipulable est d'autant plus importante que le compilateur Vélus ne préserve pas les erreurs d'exécution du langage source. Détecter d'éventuelles erreurs dans le code généré, à l'aide d'un outil comme Astrée [4] par exemple, ne permet donc pas de valider l'hypothèse no-run-time-errors. Il faut nécessairement analyser le programme source, ce que nous proposons de faire dans la prochaine section.

5.2 Détection statique des erreurs à l'exécution

Nous avons développé une analyse statique du code source Lustre, certes très simple, mais permettant de valider l'hypothèse no-run-time-errors sur une classe de programmes n'impliquant pas d'opérations « problématiques ».

Nous avons constaté dans les chapitres précédents que la seule source d'erreurs de la forme $\mathsf{err}_{\mathsf{rt}}$ était l'échec des opérateurs arithmétiques ou logiques notés $\diamond$ et $\oplus$, définis dans CompCert, qui interprète les normes du langage C, reflétant elles-mêmes le comportement de certains processeurs. Plus précisément, les trois fonctions correspondantes dans CompCert, `sem_unary_operation` et `sem_cast` pour $\diamond$, `sem_binary_operation` pour $\oplus$, sont partielles : leur résultat est `None` en cas d'échec, ce que nous traduisons en $\mathsf{err}_{\mathsf{rt}}$ dans notre modèle, cf. les définition $\mathsf{lift}^{\diamond}$ et $\mathsf{lift}^{\oplus}$ en section 3.3.2. Les cas d'échec, en fait assez peu nombreux, sont les suivants.

— La conversion d'un flottant en entier avec `sem_cast` échoue si la valeur de l'argument dépasse les bornes des entiers ou vaut `Infty` ou `Nan`.

— La division ou le modulo d'entiers échoue si la seconde opérande vaut 0.

— La division ou le modulo d'entiers *signés* échoue si la première opérande vaut `Int.min_signed` et la seconde opérande vaut -1.

— Le décalage par n bits d'un entier, à gauche ou à droite, échoue si n est supérieur ou égal à la taille d'un mot dans l'architecture ciblée.

En interdisant préventivement les conversions flottant/entier ou les divisions, modulos ou décalages dont la seconde opérande n'est pas constante,

[2] Colaço *et al.*, 2023
[3] Colaço *et al.*, 2023
[4] Kästner *et al.*, 2010

nous définissons une sur-approximation de la classe des programmes sans erreurs. Nous implémentons une procédure qui se limite à analyser localement les expressions, sans information sur l'environnement d'évaluation. Le pseudo-code ci-dessous donne un aperçu de cette analyse, qui répond vrai ou faux selon si l'expression binaire $e_1 \oplus e_2$ est acceptable ou non.

```
check_exp (Ebinop ⊕ e1 e2) = match ⊕,e2 with
 | Odiv, Econst (Vint i) ⇒ i ≠ 0 && i ≠ -1
 | Odiv, _ ⇒ false
 | Oshr, Econst (Vint i) ⇒ i <? Int.iwordsize
 | Oshr, _ ⇒ false
 | ...
```

Par exemple, l'expression `y >> 4` est considérée comme acceptable tandis que `y / t` est rejetée. Nous étendons cette analyse à une fonction check-ops G vérifiant toutes les expressions d'un programme G et dont il est assez simple de démontrer la correction.

Théorème 5.1. Pour tout nœud f dans un programme bien typé G et pour tous flots xs, si check-ops $G = \mathsf{T}$ alors no-run-time-errors $G\ f\ xs$.

Nous pouvons alors re-formuler le principal théorème de compilation.

Théorème (correction avec heuristique)**.**

si compile $G\ f = \mathsf{OK}\ asm$
et check-ops $G = \mathsf{T}$
et welltyped-inputs $G\ f\ xs$
alors $\exists ys, G \vdash f(xs) \Downarrow ys \wedge asm \Downarrow \langle \mathsf{Load}(xs(i)) \cdot \mathsf{Store}(ys(i)) \rangle_{i=0}^{\infty}$.

En pratique, le compilateur applique cette petite heuristique et émet un message d'avertissement lorsque les critères ne sont pas satisfaits, comme l'illustre la session suivante.

```
$ cat ok_div.lus
node f (x : int; y : int) returns (z : int)
let
  z = (x + y) / 2;
tel
$ velus ok_div.lus # pas d'avertissement
$ cat ko_div.lus
node f (x : int; t : int) returns (y : int)
let
  y = 0 fby (x + y) / t;
tel
$ velus ko_div.lus
Warning: cannot guarantee absence of arithmetic errors
```

En l'absence d'avertissement, le programme cible est garanti de s'exécuter à l'infini, sans erreurs, et d'exhiber le même comportement que le programme source dans notre sémantique dénotationnelle. Lorsqu'un avertissement est émis, la compilation s'effectue normalement mais l'obligation no-run-time-errors reste à démontrer pour valider le théorème de correction.

Bien que très simple, cette heuristique pourrait tout à fait s'appliquer à un programme de contrôle commande, dont les principales opérations numériques s'effectuent sur des flottants. En effet, une erreur dans les calculs flottants est représentée par la valeur spéciale mais bien typée `Nan`, n'interférant en rien avec les erreurs de notre modèle sémantique. De même, la restriction à un membre droit constant pour la division entière s'accommode parfaitement à une procédure de recherche dichotomique dans des tableaux de correspondance, où les indices sont systématiquement divisés par 2.

Il serait aussi tentant d'instrumenter le code source pour éviter certaines erreurs d'exécution. On pourrait par exemple remplacer une équation `y = x / t` par `y = if t = 0 then 0 else x / t`. Mais la sémantique synchrone impose de toujours évaluer les deux branches d'une conditionnelle avant de sélectionner la valeur à propager, ce qui dans notre cas ne fait que reporter le problème. Une solution serait d'introduire un échantillonnage.

```
c = (t = 0);
y = merge c (true => 0 when c) (false => x / t when not c);
```

Ce code ne génère aucune erreur arithmétique car sa branche de droite n'est évaluée que lorsque la nouvelle variable c est fausse et donc lorsque t est non nulle. Cependant, raisonner sur la relation entre les valeurs de c et t requiert une analyse plus sophistiquée, comme l'interprétation abstraite par exemple, tenant compte des environnements d'évaluation.

5.3 Lien avec la sémantique de Kahn

L'autre grande question de cette thèse était celle du raisonnement interactif et, dans cet objectif, de la définition d'une sémantique de Kahn pour le langage d'entrée. Tout au long du chapitre 3, nous avons démontré qu'après effacement des absences dans les flots, tous nos opérateurs synchrones, sauf celui de la réinitialisation modulaire, correspondent bien aux opérateurs usuels des réseaux de Kahn, à condition que les flots calculés soient sans erreurs. Si les erreurs de données et de synchronisation sont bien exclues par le typage du programme, les erreurs arithmétiques ou logiques restent un problème. Considérons par exemple l'instance suivante.

```
let f (x : int) returns (y : int)
let
  y = (16 / x) fby (16 + x);
tel
```

x	16	8	0	4	1	...
y_{K}	1	32	24	16	20	...
y_{SD}	1	32	$\mathsf{err}_{\mathsf{rt}}$	$\mathsf{err}_{\mathsf{rt}}$	$\mathsf{err}_{\mathsf{rt}}$	...

L'interprétation de Kahn de l'opérateur `fby` extrait la tête du flot de gauche

et la place en tête du flot de droite, résultant ici en le flot y_{K}. Dans notre modèle en revanche, la fonction fby maintient la synchronisation de ses deux opérandes, et propage les éventuelles valeurs erronées du flot de gauche, comme dans y_{SD}. Nous sommes donc en présence d'une exécution valide dans le modèle de Kahn, mais qui ne satisfait pas les contraintes de la sémantique du compilateur, qui doit être sans erreurs. C'est une différence fondamentale des deux modèles dont il est nécessaire de tenir compte. Nous avons constaté au chapitre 2, par une étude de travaux antérieurs et par notre propre expérience, que le modèle de Kahn est celui qui convient le mieux à un raisonnement interactif sur les programmes, notamment en raison de la simplicité de son `fby`. Or, ce modèle ne permet pas de détecter certaines erreurs d'exécution, que l'on souhaiterait pourtant démontrer inexistantes, ne serait-ce que pour valider l'hypothèse du théorème de correction de la compilation. Nous entrevoyons au moins deux pistes pour aborder ce problème.

1. Raisonner sur le modèle synchrone dénotationnel pour établir l'absence d'erreurs $\mathsf{err}_{\mathsf{rt}}$ dans les flots calculés et obtenir une preuve du prédicat no-run-time-errors. La compilation est donc certifiée correcte, ainsi que l'équivalence entre l'interprétation de Kahn et celle du modèle synchrone dénotationnel dans lequel toutes les absences ont été effacées. Les autres preuves de programmes peuvent se faire dans le modèle de Kahn avec une garantie de leur préservation sur le code généré. Cela nécessite de définir avec précaution le transfert des propriétés, invariants et autre pré-conditions sur les flots des deux modèles.
2. Identifier les points de divergence entre les deux modèles qui pourraient « cacher » des erreurs et les traiter au cas par cas. Dans notre dernier exemple, il s'agirait de remplacer l'équation de y par les deux équations suivantes, où t est une nouvelle variable locale au nœud.

   ```
   t = 16 / x;
   y = t fby (16 + x);
   ```

   Prouver que les flots de t et y sont sans erreurs $\mathsf{err}_{\mathsf{rt}}$ dans le modèle de Kahn suffirait à obtenir une preuve no-run-time-errors dans le modèle synchrone. Cette méthode est assimilable à de la transformation de programmes et doit donc venir avec des relations d'équivalence de programmes, des principes de substitutions de variables etc. Il faut aussi s'assurer que la substitution d'une expression à gauche d'un fby n'interfère pas dans la production d'erreurs. Une méthode moins invasive consisterait à générer des obligations, dans notre cas $\mathsf{safe}_{\mathsf{rt}}\,(\mathsf{denot}_{\mathsf{E}}\,(\texttt{16/x}))$.

5.4 Bilan

Nous avons encodé dans Vélus un nouveau modèle du langage d'entrée. Bien que les opérateurs sémantiques du noyau flot de données de Lustre soient étudiés et documentés depuis longtemps, leur implantation dans un compilateur vérifié a soulevé de nombreux défis. La rigueur imposée par l'utilisation d'un assistant de preuve comme Coq nous a poussé à étudier ces opérateurs dans les moindres détails et, dans le cas du délai initialisé, à en proposer une nouvelle définition, compatible avec le calcul de point fixe d'une sémantique dénotationnelle. Nous avons également défini formellement deux versions d'un opérateur de réinitialisation modulaire sur les fonctions de flots et démontré les conditions de leur équivalence, confortant au passage la définition relationnelle avec de la réinitialisation avec masquage.

L'impossibilité d'exécuter le code extrait de notre modèle dénotationnel est une déception mais ouvre la voie à d'autres approches. Nous pourrions par exemple encoder un interprète efficace du langage, co-itératif et constructif, et démontrer son équivalence avec notre modèle. Il serait alors judicieux de reconsidérer le choix de la théorie pour encoder le modèle dénotationnel : sans besoin impératif de constructivité, il n'est plus nécessaire de raisonner sur des flots instrumentés avec ϵ. Nos définitions des opérateurs synchrones avec erreurs, indépendantes de la représentation des flots, s'appliqueront à n'importe quel modèle synchrone dénotationnel.

De façon générale, notre travail fait le lien entre celui de Sylvain Boulmé et Grégoire Hamon, premiers à avoir encodé la sémantique dénotationnelle de Lucid Synchrone en Coq, et celui de Christine Paulin-Mohring, qui a astucieusement encodé celle des réseaux de Kahn. Dans le contexte de Vélus, nous poussons à l'extrême le raisonnement formel sur l'association entre syntaxe et sémantique des programmes Lustre, avec comme ambition de proposer le meilleur environnement de raisonnement interactif. Nos observations tendent à confirmer celles de Cécile Dumas, pour qui la bonne lisibilité du contexte de preuve est essentielle et passe avant tout par un choix de modèle sémantique approprié. Nous pouvons désormais raisonner sur un modèle dénotationnel du langage, par des réécritures ou en appliquant le principe d'induction continue, et tirer parti de la correction du compilateur pour voir les propriétés ainsi prouvées s'appliquer automatiquement au code assembleur généré. Nous exauçons à notre façon l'ambition formulée par Gérard Berry il y a bien longtemps :

> *« What You Prove Is What You Execute. »*

Bibliographie

ANSYS/Esterel Technologies (2016). *The SCADE 6 Language*. KCG-SRS-007/R/2.

ANSYS/Esterel Technologies (2024). SCADE Suite. `http://www.ansys.com/products/embedded-software/ansys-scade-suite`.

Appel, A. W., Dockins, R., Hobor, A., Beringer, L., Dodds, J., Stewart, G., Blazy, S. et Leroy, X. (2014). *Program Logics for Certified Compilers*. Cambridge University Press.

Benveniste, A., Bourke, T., Caillaud, B., Pagano, B. et Pouzet, M. (2014). A type-based analysis of causality loops in hybrid modelers. *In* Fränzle, M. et Lygeros, J., éditeurs : *Proceedings of the 17th International Conference on Hybrid Systems : Computation and Control (HSCC 2014)*, pages 71–82, Berlin, Germany. ACM Press.

Berry, G. et Cosserat, L. (1984). The ESTEREL synchronous programming language and its mathematical semantics. *In* Brookes, S. D., Roscoe, A. W. et Winskel, G., éditeurs : *Seminar on Concurrency*, volume 197 de *Lecture Notes in Elecrical Engineering*, pages 389–448, Pittsburg, USA. Springer.

Bertot, Y. et Castéran, P. (2004). *Interactive Theorem Proving and Program Development : Coq'Art : The Calculus of Inductive Constructions*. Springer.

Bertot, Y. et Komendantskaya, E. (2008). Inductive and coinductive components of corecursive functions in Coq. *Electronic Notes in Theoretical Computer Science*, 203(5):25–47.

Biernacki, D., Colaço, J.-L., Hamon, G. et Pouzet, M. (2008). Clock-directed modular code generation for synchronous data-flow languages. *In Proceedings of the 9th ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2008)*, LCTES, pages 121–130, Tucson, AZ, USA. ACM Press.

Boillot, J. et Feret, J. (2023). Symbolic transformation of expressions in modular arithmetic. *In Static Analysis : 30th International Symposium, SAS 2023*, Static Analysis : 30th International Symposium, SAS 2023, pages 84–113, Cascais, Portugal.

Boulmé, S. et Hamon, G. (2001a). Certifying synchrony for free. *In* Nieuwenhuis, R. et Voronkov, A., éditeurs : *Proceedings of the 8th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2001)*, volume 2250 de *Lecture Notes in Elecrical Engineering*, pages 495–506, Havana, Cuba. Springer.

Boulmé, S. et Hamon, G. (2001b). A clocked denotational semantics for Lucid-Synchrone in Coq. Rapport technique, LIP6.

Bourke, T., Brun, L., Dagand, P.-É., Leroy, X., Pouzet, M. et Rieg, L. (2017). A formally verified compiler for Lustre. *In Proceedings of the 2017 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 586–601, Barcelona, Spain. ACM Press.

Bourke, T., Brun, L. et Pouzet, M. (2020). Mechanized semantics and verified compilation for a dataflow synchronous language with reset. *Proceedings of the of the ACM on Programming Languages*, 4(POPL):1–29.

Bourke, T., Jeanmaire, P., Pesin, B. et Pouzet, M. (2021). Verified Lustre normalization with node subsampling. *ACM Transactions on Embedded Computing Systems*, 20(5s):Article 98. Presented at 21st International Conference on Embedded Software (EMSOFT 2021).

Bourke, T., Pesin, B. et Pouzet, M. (2023a). Analyse de dépendance vérifiée pour un langage synchrone à flot de données. *In* Bourke, T. et Demange, D., éditeurs : *JFLA 2023 - Les trente-quatrièmes Journées Francophones des Langages Applicatifs*, pages 101–120, Praz-sur-Arly, France.

Bourke, T., Pesin, B. et Pouzet, M. (2023b). Verified compilation of synchronous dataflow with state machines. *ACM Transactions on Embedded Computing Systems*, 22(5s):137 :1–137 :26. Presented at 23rd International Conference on Embedded Software (EMSOFT 2023).

Bourke, T. et Pouzet, M. (2019). Arguments cadencés dans un compilateur Lustre vérifié. *In* Magaud, N. et Dargaye, Z., éditeurs : *30^{ièmes} Journées Francophones des Langages Applicatifs (JFLA 2019)*, pages 109–124, Les Rousses, Haut-Jura, France.

Brun, L., Garion, C., Garoche, P.-L. et Thirioux, X. (2023). Equation-directed axiomatization of lustre semantics to enable optimized code validation. *ACM Transactions on Embedded Computing Systems*, 22(5s).

Presented at 23rd International Conference on Embedded Software (EMSOFT 2023).

Capretta, V. (2005). General recursion via inductive types. *Logical Methods in Computer Science (LMCS)*, 1(2).

Caspi, P. (1994). Towards recursive block diagrams. *Annual Review in Automatic Programming*, 18:81–85.

Caspi, P., Pilaud, D., Halbwachs, N. et Plaice, J. A. (1987). LUSTRE : A declarative language for programming synchronous systems. *In Proceedings of the 14th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1987)*, pages 178–188, Munich, Germany. ACM Press.

Caspi, P. et Pouzet, M. (1995). A functional extension to Lustre. *In* Orgun, M. et Ashcroft, E., éditeurs : *International Symposium on Languages for Intentional Programming*, Sydney, Australia. World Scientific.

Caspi, P. et Pouzet, M. (1996). Synchronous Kahn networks. *In Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 226–238, Philadelphia, PA, USA. ACM Press.

Champion, A., Mebsout, A., Sticksel, C. et Tinelli, C. (2016). The Kind 2 model checker. *In* Chaudhuri, S. et Farzan, A., éditeurs : *Proceedings of the 28th International Conference on Computer Aided Verification (CAV 2016), Part II*, volume 9780 de *Lecture Notes in Elecrical Engineering*, pages 510–517, Toronto, Canada. Springer.

Colaço, J.-L., Mendler, M., Pauget, B. et Pouzet, M. (2023). A constructive state-based semantics and interpreter for a synchronous data-flow language with state machines. *ACM Transactions on Embedded Computing Systems*, 22(5s):152 :1–152 :26. Presented at 23rd International Conference on Embedded Software (EMSOFT 2023).

Colaço, J.-L., Pagano, B. et Pouzet, M. (2005). A conservative extension of synchronous data-flow with state machines. *In* Wolf, W., éditeur : *Proceedings of the 5th ACM International Conference on Embedded Software (EMSOFT 2005)*, EMSOFT, pages 173–182, Jersey City, USA. ACM Press.

Colaço, J.-L., Pagano, B. et Pouzet, M. (2017). Scade 6 : A formal language for embedded critical software development. *In Proceedings of the 11th International Symposium on Theoretical Aspects of Software Engineering (TASE 2017)*, pages 4–15, Nice, France. IEEE Computer Society.

Colaço, J.-L. et Pouzet, M. (2003). Clocks as first class abstract types. *In* Alur, R. et Lee, I., éditeurs : *Proceedings of the 3rd International Conference on Embedded Software (EMSOFT 2003)*, volume 2855 de *Lecture Notes in Elecrical Engineering*, pages 134–155, Philadelphia, PA, USA. Springer.

Coq Development Team (2020). *The Coq proof assistant reference manual*. Inria.

Dumas, C. (2000). *Méthodes déductives pour la preuve de programmes Lustre*. Thèse de doctorat, Université Joseph Fourier, Grenoble, France.

Dumas, C. et Caspi, P. (2000). A PVS proof obligation generator for Lustre programs. *In* Parigot, M. et Voronkov, A., éditeurs : *Proceedings of the 7th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2000)*, volume 1955 de *Lecture Notes in Elecrical Engineering*, pages 179–188, Reunion Island, France. Springer.

Endrullis, J., Hendriks, D. et Bodin, M. (2013). Circular coinduction in Coq using bisimulation-up-to techniques. *In* Blazy, S., Paulin-Mohring, C. et Pichardie, D., éditeurs : *Proceedings of the 4th International Conference on Interactive Theorem Proving (ITP 2013)*, volume 7998 de *Lecture Notes in Elecrical Engineering*, pages 354–369, Rennes, France. Springer.

Gérard, L. (2013). *Programmer le parallélisme avec des futures en Heptagon un langage synchrone flot de données et étude des réseaux de Kahn en vue d'une compilation synchrone*. Thèse de doctorat, Université Paris-Sud, Paris, France.

Greenaway, D., Lim, J., Andronick, J. et Klein, G. (2014). Don't sweat the small stuff : Formal verification of c code without the pain. *In Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 45, Edinburgh, UK. ACM Press. `http://www.nicta.com.au/pub?id=7629`.

Guéneau, A., Myreen, M. O., Kumar, R. et Norrish, M. (2017). Verified characteristic formulae for CakeML. *In* Yang, H., éditeur : *26nd European Symposium on Programming (ESOP 2017), part of European Joint Conferences on Theory and Practice of Software (ETAPS 2017)*, volume 10201 de *Lecture Notes in Elecrical Engineering*, page 584–610, Uppsala, Sweden. Springer.

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

Halbwachs, N., Lagnier, F. et Ratel, C. (1992). Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Transactions on Software Engineering*, 18(9):785–793.

Halbwachs, N., Lagnier, F. et Raymond, P. (1993). Synchronous observers and the verification of reactive systems. *In* Nivat, M., Rattray, C., Rus, T. et Scollo, G., éditeurs : *Proceedings of the 3rd International Conference on Algebraic Methodology and Software Technology (AMAST'93)*, Twente, Netherlands. Workshops in Computing, Springer Verlag.

Hamon, G. et Pouzet, M. (2000). Modular resetting of synchronous data-flow programs. *In* Pfenning, F., éditeur : *Proceedings of the 2nd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2000)*, pages 289–300, Montreal, Canada. ACM.

Heptagon Developers (2017). *Heptagon/BZR manual*.

Jahier, E., Raymond, P. et Halbwachs, N. (2019). *The Lustre V6 Reference Manual*. Verimag, Grenoble.

Jourdan, J.-H., Laporte, V., Blazy, S., Leroy, X. et Pichardie, D. (2015). A formally-verified C static analyzer. *In Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015)*, pages 247–259, Mumbai, India. ACM Press.

Kahn, G. (1974). The semantics of a simple language for parallel programming. *In* Rosenfeld, J. L., éditeur : *Proceedings of the International Federation for Information Processing (IFIP) Congress 1974*, pages 471–475, Stockholm, Sweden. North-Holland.

Kästner, D., Wilhelm, S., Nenova, S., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A. et Rival, X. (2010). Astrée : Proving the absence of runtime errors. *In Proceedings of the 5th International Congress on Embedded Real-Time Software (ERTS^2 2010)*, Toulouse, France. AAAF/-SEE/SIA.

Leroy, X. (2009a). Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115.

Leroy, X. (2009b). A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446.

Mikáč, J. (2005). *Raffinement et preuves de systèmes Lustre*. Thèse de doctorat, Institut National Polytechnique de Grenoble - INPG.

Müller, O. (1998). *A Verification Environment for I/O Automata Based on Formalized Meta-Theory*. Thèse de doctorat, Technische Universität München, Germany.

Müller, O., Nipkow, T., von Oheimb, D. et Slotosch, O. (1999). HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191 – 223.

Paulin-Mohring, C. (2009). A constructive denotational semantics for Kahn networks in Coq. *In* Bertot, Y., Huet, G., Lévy, J.-J. et Plotkin, G., éditeurs : *From Semantics to Computer Science : Essays in Honour of Gilles Kahn*, pages 383–413. Cambridge University Press, Cambridge, UK.

Pesin, B. (2023). *Verified Compilation of a Synchronous Dataflow Language with State Machines*. Thèse de doctorat, PSL Research University.

Pouzet, M. (2006). *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud.

Pouzet, M. (2019). Clocks in Kahn process networks. Notes de cours.

Pouzet, M. (2021). ZRun, a denotational semantics for zelus. `https://github.com/marcpouzet/zrun/blob/master/src/coiteration.ml`.

Pouzet, M. et Raymond, P. (2009). Modular static scheduling of synchronous data-flow networks : An efficient symbolic representation. *In Proceedings of the 9th ACM International Conference on Embedded Software (EMSOFT 2009)*, pages 215–224, Grenoble, France. ACM Press.

Raymond, P. (1991). *Compilation efficace d'un langage déclaratif synchrone : le générateur de code Lustre-V3*. Thèse de doctorat, Grenoble INP.

Raymond, P. (1992). The Lustre V4 distribution. `http://www-verimag.imag.fr/The-Lustre-Toolbox.html`.

Rusu, V. et Nowak, D. (2022). Defining corecursive functions in Coq using approximations. *In 36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222, pages 12 :1–12 :24.

Sheeran, M., Singh, S. et Stålmarck, G. (2000). Checking safety properties using induction and a SAT-solver. *In* Hunt Jr., W. A. et Johnson, S. D., éditeurs : *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD 2000)*, pages 127–144, Austin, TX, USA. IEEE.

Sozeau, M. (2009). A new look at generalized rewriting in type theory. *Journal of Formalized Reasoning*, 2(1):41–62.

Xia, L.-y., Zakowski, Y., He, P., Hur, C.-K., Malecha, G., Pierce, B. C. et Zdancewic, S. (2019). Interaction Trees : Representing recursive and impure programs in Coq. *Proceedings of the of the ACM on Programming Languages*, 4(POPL).

RÉSUMÉ

Lustre est un langage synchrone de type flot de données. Sa structure temporelle discrète et son système d'horloges lui permettent de garantir untemps d'exécution et une consommation de mémoire bornés statiquement, favorisant ainsi son adoption dans le domaine de systèmes critiques, notamment au sein de l'outil industriel Scade. Vélus, développé à l'Inria, est un compilateur Lustre vérifié. En s'appuyant sur la définition d'une sémantique formelle pour chaque langage intermédiaire, il apporte une preuve qu'un programme Lustre et sa traduction en Clight (langage pris en charge par CompCert) exhibent des comportements identiques.

Dans cette thèse, nous développons un nouveau modèle du noyau flot de données du langage d'entrée, basé sur une sémantique dénotationnelle synchrone, et donnons les conditions exactes de son équivalence avec le modèle relationnel existant dans Vélus. Cette approche constructive permet d'obtenir une sémantique exécutable, renforçant ainsi le principal théorème de correction de la compilation. Grâce au principe d'induction de Scott, propre à la sémantique dénotationnelle, nous menons des raisonnements très naturels sur la dynamique des programmes, moyennant un traitement explicite des erreurs pouvant survenir à l'exécution. Enfin, nous explorons la possibilité de s'affranchir des contraintes de synchronisation du langage en proposant une correspondance formelle de notre modèle avec celui des réseaux de Kahn. Nous esquissons les principes de l'infrastructure nécessaire à un raisonnement vérifié de bout en bout sur les programmes compilables.

MOTS CLÉS

Assistants de preuve, preuves de programmes, systèmes embarqués, programmation synchrone, compilation vérifiée, sémantique dénotationnelle.

ABSTRACT

Lustre is a synchronous dataflow language. Its discrete temporal structure and clocking system guarantee statically bounded execution time and memory consumption, favoring its adoption in mission-critical systems, notably within the Scade industrial tool. Vélus, developed at Inria, is a verified Lustre compiler. Based on the definition of formal semantics for each intermediate language, it provides a proof that a Lustre program and its translation into Clight (the language supported by CompCert) exhibit identical behaviors.

In this thesis, we develop a new model of the input language's dataflow kernel, based on synchronous denotational semantics, and give the exact conditions for its equivalence with the existing relational model in Vélus. This constructive approach results in an executable semantics, reinforcing the main correctness theorem of the compilation. Thanks to Scott's induction principle, specific to denotational semantics, we are able to conduct very natural reasoning on program dynamics, but with explicit treatment of errors that may occur at runtime. Finally, we explore the possibility of freeing ourselves from language synchronization constraints by proposing a formal correspondence between our model and that of Kahn's networks. We outline the principles of the infrastructure required for an end-to-end verified reasoning on compilable programs.

KEYWORDS

Embedded systems, verified compilation, denotational semantics, proof of programs, synchronous programming, proof assistants.