Normalisation vérifiée du langage Lustre

Timothy Bourke, Paul Jeanmaire, Basile Pesin, Marc Pouzet

Normalisation vérifiée du langage Lustre

Timothy Bourke[1,2], Paul Jeanmaire[1,2], Basile Pesin[1,2], Marc Pouzet[2,1]

[1] Inria Paris, France
[2] Département d'informatique de l'École normale supérieure, CNRS, PSL University, Paris, France

Résumé

Lustre est un langage synchrone à flots de données conçu pour programmer des systèmes embarqués. Dans le cadre du projet Vélus, nous avons développé et formalisé dans Coq un compilateur qui accepte une forme normalisée du langage et la compile vers du code impératif. Si cette forme réduite prend en charge un code généré depuis une interface utilisateur basée sur les schémas-blocs, nous voulons offrir au programmeur la possibilité de manipuler le langage complet.

Dans cet article nous présentons l'étape de normalisation, qui transforme le langage de programmation en langage normalisé. Cette transformation est décomposée en trois étapes afin de simplifier les preuves de correction. Pour établir la préservation de la sémantique, il est nécessaire de démontrer que les trois passes préservent certaines propriétés statiques et dynamiques du langage. En particulier, il faut prouver le lien entre le typage des horloges et la sémantique dynamique pour pouvoir raisonner sur la suite de la compilation.

1 Introduction

La conception de systèmes embarqués critiques est souvent basée sur des modèles de type schémas-blocs qui permettent la définition et l'interconnexion de comportements temporels. Un tel modèle peut être compris comme une composition récursive de séquences infinies, les flots de valeurs à calculer pas à pas, et compilé dans un code impératif pour une exécution cyclique. C'est l'idée principale derrière le langage académique Lustre [13] et son descendant industriel Scade 6 [10]. Le projet Vélus [1] [5, 6] vise à formaliser les particularités de ces langages, leurs systèmes de types [11] et leurs schémas de compilation [3] dans l'assistant de preuve Coq [12].

Les précédents travaux sur Vélus ont traité une forme restreinte du langage. Cette forme suffit pour traduire les programmes représentés graphiquement et est une étape préalable à la génération de code impératif. Mais elle ne représente qu'un sous-ensemble du vrai langage source et elle est moins commode pour les programmes écrits ou lus dans un éditeur. En outre, le traitement de la forme plus générale dans un assistant de preuve soulève d'intéressantes questions techniques. On s'intéresse donc ici au passage de la forme générale du langage à la forme normalisée. La transformation en elle-même est presque banale ; le défi est de vérifier sa correction dans un assistant de preuve. Notre contribution est de présenter une implémentation qui facilite cette tâche et les invariants principaux qui sous-tendent notre preuve formelle.

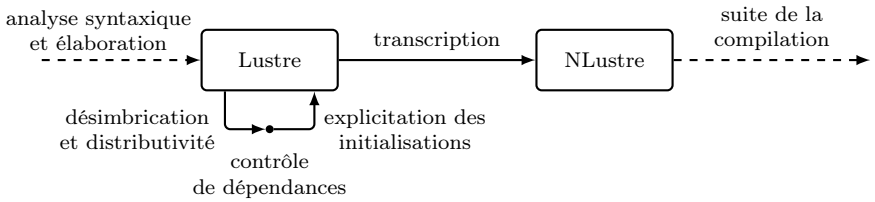

FIGURE 1 – Passes de normalisation

1. https://velus.inria.fr

```
node count_down(res:bool; n:int)
returns (cpt:int)
let
  cpt = if res then n else (n fby (cpt - 1));
tel

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
        (count_down((edge, n) when ck))
        (0 when not ck);
  o = v > 0;
tel
```

FIGURE 2 – Exemple : source Lustre

```
node count_down(res:bool, n:int32)
returns (cpt:int32)
var $114:int32, $215:int32, $214:bool;
let
  $214 = true fby false;
  $215 = 0 fby (cpt - 1);
  $114 = if $214 then n else $215;
  cpt = if res then n else $114;
tel

node rising_edge_retrigger(i:bool, n:int32)
returns (o:bool)
var edge:bool, ck:bool, v:int32,
    $122:bool, $121:bool,
    $124:int32 when ck;
let
  $122 = false fby (not i);
  edge = i and $122;
  $121 = false fby o;
  ck = edge or $121;
  $124 = count_down(edge when ck, n when ck);
  v = merge ck $124 (0 when not ck);
  o = v > 0;
tel
```

FIGURE 3 – Exemple : NLustre généré

i	F	**T**	F	F	F	F	F	T	F	T	F	F	F	F	···
n	3	**3**	3	3	3	3	3	3	3	3	3	3	3	3	···
edge	F	**T**	F	F	F	F	F	T	F	T	F	F	F	F	···
ck	F	**T**	**T**	**T**	**T**	F	F	T	T	T	T	T	T	F	···
edge when ck		**T**	F	F	F			T	F	T	F	F	F		···
count_down(...)		**3**	**2**	**1**	**0**			3	2	3	2	1	0		···
v	0	**3**	**2**	**1**	**0**	0	0	3	2	3	2	1	0	0	···
o	F	**T**	**T**	**T**	**F**	F	F	T	T	T	T	T	F	F	···

FIGURE 4 – Exemple d'une trace du nœud `rising_edge_retrigger`

Les parties du compilateur Vélus concernées par cette transformation sont schématisées dans la figure 1. L'analyse syntaxique suivie d'une passe d'élaboration ajoutant les annotations de typage produit un programme dans la syntaxe abstraite *Lustre*. La passe de normalisation est chargée de mettre le programme sous forme normalisée. On la sépare en deux transformations source à source. La première désimbrique les éléments qui ne doivent se trouver qu'à la racine d'une expression et distribue les opérateurs sur les listes d'arguments. La deuxième explicite des initialisations pour simplifier la forme des opérateurs séquentiels. Après la désimbrication, on vérifie que le programme ne contient pas de dépendance circulaire. Ensuite, la passe de transcription transforme le programme dans la syntaxe abstraite *NLustre* qui encode les exigences de la forme normalisée et sert de point d'entrée pour la suite de la compilation.

Exemple. Le programme en figure 2 définit un opérateur typique de l'automatique [14, §1.2.2] qui est fourni par les bibliothèques de Scade et de Simulink. Il est composé de deux fonctions de

flots, également appelées *nœuds*. La normalisation le transforme dans le programme en figure 3.

Le premier nœud s'appelle `count_down`, prend en entrée deux flots, `res` et `n`, et rend en sortie un flot `cpt`. Dans le corps du nœud, une équation définit le flot associé à `cpt` avec une expression conditionnelle qui prend la valeur de l'entrée `n` lorsque `res` est vrai et sinon la valeur définie par la sous-expression `n fby (cpt - 1)`. L'opérateur `fby`, « *followed by* », produit un flot en décalant d'un instant le flot à droite et en remplissant le vide initial ainsi créé avec la première valeur du flot à gauche. Ici, le résultat est un flot `cpt` qui compte à rebours de la valeur courant de `n` en recommençant chaque fois que `res` est vrai.

La normalisation de `count_down` donne le nœud du même nom dans la figure 3. Dans l'équation de `cpt`, le `fby` a été désimbriqué et remplacé par la nouvelle variable locale `$114`. Ensuite, une variable d'initialisation `$214` a été ajoutée et utilisée dans la définition de `$114` pour choisir entre la valeur initiale venant de `n` et toutes les autres valeurs venant de `$215`. Par conséquent, tous les `fby`s sont initialisés par une constante dans l'expression à gauche. Pour ce nœud, il n'était pas nécessaire de distribuer un opérateur sur une liste d'arguments.

Le deuxième nœud, `rising_edge_retrigger`, attend un front montant, c'est-à-dire, un F suivi directement d'un T, sur son entrée `i` et émet ensuite la valeur T n fois sur son unique sortie. Une trace est donnée dans la figure 4. On y voit en caractères gras le premier front montant sur l'entrée `i`, la valeur de `n` à cet instant et, sur la dernière ligne, la réponse du nœud sur `o`. Le corps du nœud est composé de quatre équations, dont trois pour les variables locales du nœud. La première équation détecte les fronts montants en considérant chaque paire de valeurs successives d'`i`. L'équation de `ck` détermine si un compte à rebours est actif. C'est le cas si un front montant vient d'être détecté ou si la sortie était vraie dans l'instant antérieur.

L'équation de `v` utilise les deux opérateurs d'échantillonnage `when` et `merge`. Un `when` filtre un premier flot en fonction de la valeur d'un deuxième. Par exemple, l'expression `edge when ck` prend la valeur de celle de `edge` seulement lorsque `ck` est vrai, comme on peut voir dans la trace. On dit alors que le flot est « présent ». Aux autres instants, laissés vides dans la figure, on dit que le flot est « absent ». Dans l'expression `count_down((edge, n) when ck)`, le premier nœud est appliqué à deux flots qui sont tous les deux filtrés par `ck`. Le cadencement de cette instance est donc plus lent que celui de son contexte. Un `merge` fusionne deux flots en fonction de la valeur de son premier argument. Lorsque le premier flot est vrai, une valeur est prise du deuxième flot qui doit être présent et le troisième flot doit être absent, et inversement. Donc, la valeur de `v` vient de l'instance de `count_down` quand `ck` est vrai et d'un flot de zéros sinon.

Enfin, la dernière équation assure que la sortie n'est vraie que lorsque le compte à rebours a une valeur positive.

Le résultat de la normalisation de `rising_edge_retrigger` est présenté dans la figure 3. Dans l'équation de `v`, l'instance de nœud a été desimbriquée et remplacée par la variable `$124`. Dans l'expression définissant celle-ci se trouve la seule application de la distributivité dans cet exemple : l'opérateur `when` a été distribué sur la liste d'arguments `(edge, n)`. En général, la distributivité s'applique aussi aux arguments de `merge`, `if-then-else` et `fby`.

Les `fby`s dans les définitions d'`edge` et de `ck` ont été désimbriqués mais, ayant déjà des constantes à gauche, ils n'ont pas été autrement modifiés.

Sommaire. La syntaxe et la sémantique de Lustre et de NLustre formalisé dans Vélus sont présentées dans la section 2. La présentation des passes indiquées en figure 1 commence dans la section 3 avec la dernière, la transcription, avant de revenir dans les sections 4 et 5 sur la désimbrication/distributivité et explicitation des initialisations. Cette organisation, qui ne reflète pas l'ordre de traitement d'un programme, facilite la description d'une propriété sémantique qui est nécessaire et pour le passage entre Lustre et NLustre et pour l'explicitation des initialisations.

2 Lustre dans Vélus

Dans cette partie, nous présentons la formalisation de Lustre dans Vélus. Nous introduisons d'abord les objets syntaxiques qui seront manipulés par l'algorithme de normalisation, puis le modèle sémantique du langage.

2.1 Syntaxes

La syntaxe des expressions et équations du langage Lustre est donnée en figure 5. Une équation associe une liste d'identifiants à une liste d'expressions. Les `fby`s et les instanciations de nœuds peuvent apparaître sans restriction. Par ailleurs, les opérateurs `when`, `merge`, `if-then-else` et `fby` peuvent être appliqués à des listes de sous-expressions. Une expression peut donc produire plusieurs flots. Cette fonctionnalité permet d'imbriquer librement les instances de nœud, qui peuvent rendre plusieurs flots, dans d'autres expressions.

$$
\begin{array}{rcl}
e & ::= & c \\
& | & x \\
& | & \diamond\ e \\
& | & e \oplus e \\
& | & e^+ \texttt{ fby } e^+ \\
& | & e^+ \texttt{ when } x \\
& | & \texttt{merge } x\ e^+\ e^+ \\
& | & \texttt{if } e \texttt{ then } e^+ \texttt{ else } e^+ \\
& | & f\ \texttt{(}\ e^+\ \texttt{)} \\
\\
eq & ::= & x^+ \texttt{ = } e^+ \texttt{ ;}
\end{array}
$$

FIGURE 5 – Equations de Lustre

$$
\begin{array}{rcl}
e & ::= & c \\
& | & x \\
& | & \diamond\ e \\
& | & e \oplus e \\
& | & e \texttt{ when } x \\
\\
ce & ::= & e \\
& | & \texttt{merge } x\ ce\ ce \\
& | & \texttt{if } e \texttt{ then } ce \texttt{ else } ce \\
\\
eq & ::= & x \texttt{ = } ce \\
& | & x \texttt{ = } c \texttt{ fby } e \\
& | & x^+ \texttt{ = } f\ \texttt{(}\ e^+\ \texttt{)}
\end{array}
$$

FIGURE 6 – Equations de NLustre

$$
\begin{array}{rcl}
n & ::= & \texttt{node } f\ \texttt{(}\ d^+\ \texttt{) returns (}\ d^+\ \texttt{)} \\
& & \texttt{var } d^*\ \texttt{;} \\
& & \texttt{let } eq^*\ \texttt{tel}
\end{array}
\qquad
\begin{array}{rcl}
d & ::= & x^{ck}_{ty} \\
\\
G & ::= & n^+
\end{array}
$$

FIGURE 7 – Syntaxe des nœuds

La syntaxe des nœuds est décrite dans la figure 7. Un nœud peut recevoir plusieurs entrées (au moins une) et produire plusieurs sorties (au moins une). De plus, il peut déclarer des variables locales. Le nœud est ensuite constitué d'une liste d'équations.

Dans la syntaxe abstraite du langage, les expressions sont annotées par leur type et leur horloge statique. On notera plus loin dans cet article e^{ck} pour l'expression e annotée par l'horloge ck. Dans Lustre le système d'horloges est, à l'instar d'un système de types [11], un moyen de vérifier statiquement qu'un programme est synchrone et donc qu'il s'exécute dans un mémoire de taille bornée. Ainsi, les opérandes d'une opération arithmétique binaire doivent avoir la même horloge tandis que les horloges des membres d'un `merge` doivent être complémentaires, comme c'est le cas dans l'exemple de la figure 2, ligne 13. Une expression comme `x + (x when c)` n'est pas synchrone car la taille du tampon nécessaire au calcul dépend directement des valeurs prises par `c`, qui ne sont en général pas déterminées à la compilation.

Une horloge est définie par $ck ::= \bullet \mid ck\ \texttt{on}\ x$. Elle peut être l'horloge de base, s'activant à chaque instant, ou bien représenter un échantillonnage sur les valeurs d'une variable booléenne x.

Le système d'horloges contraint ces annotations. Par exemple, si l'expression `e` a pour horloge $\bullet$, alors l'expression `e when x` a pour horloge $\bullet$ `on x`.

Le langage normalisé présenté dans la figure 6 est plus contraint que le langage Lustre. Les expressions y sont hiérarchisées en expressions simples et expressions de contrôle. On y catégorise trois types d'équations, contenant soit une expression de contrôle, soit un `fby`, soit une instanciation. Par ailleurs, on n'y trouve plus de listes de sous-expressions. Cela signifie que dans le langage normalisé, toutes les expression sauf les instanciations représentent exactement un flot. Les nœuds normalisés sont, comme ceux de Lustre, composés d'une liste d'équations. Cette forme normalisée est moins commode pour un programmeur, mais constitue une étape essentielle de la compilation puisqu'elle permet d'isoler les `fby`s et instanciations de nœuds qui nécessitent un traitement particulier lors des transformations ultérieures vers du code impératif.

L'objectif de l'algorithme de normalisation présenté plus loin sera donc de transformer un programme de la première syntaxe vers la seconde, tout en préservant la sémantique que nous détaillons maintenant.

2.2 Modèle sémantique de Lustre

La sémantique de Lustre associe à chaque expression une liste de flots de valeurs. La formalisation présentée ici impose des relations entre flots infinis de valeurs représentés par le type co-inductif `stream value`, où `value` représente des valeurs présentes ou absentes. On notera les valeurs présentes $\langle v \rangle$ et les absences $\langle\rangle$. On note le jugement $G, H, bs \vdash e \Downarrow \boldsymbol{vs}$ pour « dans le programme G, sous l'historique H et sur le rythme bs, l'expression e produit les flots $\boldsymbol{vs}$ ». Dans ce jugement, le paramètre global G contient les nœuds du programme. L'historique H associe des flots aux noms apparaissant dans le nœud. L'historique correspond en fait aux chronogrammes donnés plus hauts. Enfin, le flot de base bs donne le rythme d'exécution du nœud. C'est un flot de booléens (`stream bool`), vrai si l'une au moins des entrées du nœud est présente. Il est défini par la fonction `base-of : list (stream value)` $\rightarrow$ `stream bool` qui calcule l'union des présences à chaque instant.

De manière générale, la typographie en **gras** dénote une liste. On note ainsi, par simplicité, $G, H, bs \vdash \boldsymbol{es} \Downarrow \boldsymbol{vs}$ pour le cas où $\boldsymbol{es}$ est une liste d'expressions. Dans ce cas, $\boldsymbol{vs}$ correspond à la concaténation des listes de flots issues de chaque expression. Dans les jugements ci-dessous $\boldsymbol{vs}$ est bien une liste de flots, qu'on parle d'une seule expression e ou de plusieurs expressions $\boldsymbol{es}$ car, comme on l'a vu plus haut, l'une des subtilités de notre mécanisation est qu'une expression peut correspondre à plusieurs flots.

La sémantique des expressions est donnée par des fonctions ou relations co-inductives, qui apparaissent ensuite dans des règles sémantiques définies par induction sur la syntaxe des expressions. Ces opérateurs sont appliqués point-à-point sur des listes de flots.

La règle de sémantique pour les constantes, Sconst, est donnée dans la figure 8. Elle associe l'expression c à une liste d'un seul élément. Ce flot est construit avec l'opérateur `const` qui permet d'échantillonner les constantes sur un flot de base. En l'occurrence, la règle de sémantique contraint ce flot à être celui du nœud. Ainsi les constantes sont toutes émises au rythme le plus rapide du nœud.

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

Figure 8 – Sémantique des constantes

Une règle plus complexe est celle du `if-then-else`. Dans la figure 9, on présente la sémantique de l'opérateur co-inductif ite. Celle-ci s'interprète facilement : si la condition et les branches sont absentes, alors le résultat est absent. Sinon, si la condition et les branches sont présentes, on choisit la branche en fonction de la valeur de la condition, qui doit être T ou F. La valeur de ite est combinatoire. Elle ne dépend que des valeurs de ses sous-expressions au même instant. On constate aussi que ite est une fonction partielle. Elle n'est pas définie si par exemple, la valeur reçue en condition n'est pas booléene, ou si la condition est présente mais l'une des deux branches est absente. Le symbole $\doteq$ ne sert qu'à faciliter la lecture : il ne s'agit ici que d'un prédicat quaternaire. La notation double barre indique que les règles sont établies par co-induction sur les flots, et non par induction sur une structure syntaxique.

$$\frac{\mathsf{ite}\ cs\ \boldsymbol{ts}\ \boldsymbol{fs} \doteq \boldsymbol{vs}}{\mathsf{ite}\ (\langle\rangle \cdot cs)\ (\langle\rangle \cdot \boldsymbol{ts})\ (\langle\rangle \cdot \boldsymbol{fs}) \doteq \langle\rangle \cdot \boldsymbol{vs}} \qquad \frac{\mathsf{ite}\ cs\ \boldsymbol{ts}\ \boldsymbol{fs} \doteq \boldsymbol{vs}}{\mathsf{ite}\ v\ (\langle \mathrm{T}\rangle \cdot cs)\ (\langle t\rangle \cdot \boldsymbol{ts})\ (\langle f\rangle \cdot \boldsymbol{fs}) \doteq \langle t\rangle \cdot \boldsymbol{vs}}$$

$$\frac{\mathsf{ite}\ cs\ \boldsymbol{ts}\ \boldsymbol{fs} \doteq \boldsymbol{vs}}{\mathsf{ite}\ v\ (\langle \mathrm{F}\rangle \cdot cs)\ (\langle t\rangle \cdot \boldsymbol{ts})\ (\langle f\rangle \cdot \boldsymbol{fs}) \doteq \langle f\rangle \cdot \boldsymbol{vs}}$$

FIGURE 9 – Relation co-inductive ite

La règle sémantique du `if-then-else` est donnée dans la figure 10. Elle prend comme hypothèse l'existence de sémantiques pour les sous-expressions. La dernière hypothèse applique l'opérateur ite sur les valeurs issues de ces expressions. La règle du `if-then-else` est stricte, c'est à dire que les deux branches doivent avoir une sémantique, quelle que soit la valeur de la condition.

$$\textsc{Site}\ \frac{G, H, bs \vdash e \Downarrow [s] \qquad G, H, bs \vdash \boldsymbol{e_t} \Downarrow \boldsymbol{ts} \qquad G, H, bs \vdash \boldsymbol{e_f} \Downarrow \boldsymbol{fs} \qquad \mathsf{ite}\ s\ \boldsymbol{ts}\ \boldsymbol{fs} \doteq \boldsymbol{vs}}{G, H, bs \vdash \texttt{if}\ e\ \texttt{then}\ \boldsymbol{e_t}\ \texttt{else}\ \boldsymbol{e_f} \Downarrow \boldsymbol{vs}}$$

FIGURE 10 – Sémantique du `if-then-else`

On présentera les autres opérateurs co-inductifs au besoin, plus loin dans ce document. On ne détaillera pas les règles sémantiques, qui sont similaires à celle du `if-then-else`. Ces règles sont présentées dans leur intégralité dans [9, p. 37].

Les équations, dont la sémantique est donnée sous la forme $G, H, bs \vdash eq$, ne produisent pas de valeurs, contrairement aux expressions. Le rôle d'une équation est de contraindre l'historique H qui paramètre le jugement. On observe en effet que la règle SEQ donnée dans la figure 11 contraint l'historique à associer les noms à gauche de l'équation aux valeurs produites par les expressions à droite de l'équation.

La sémantique d'un nœud f dans un programme G, notée $G \vdash f(\boldsymbol{xs}) \Downarrow \boldsymbol{ys}$ associe les flots d'entrées $\boldsymbol{xs}$ aux flots de sorties $\boldsymbol{ys}$. Elle est définie par le jugement SNODE dans la figure 11. Elle exige l'existence d'un historique H contraint par chacune des équations du nœud. On voit que H n'apparaît pas dans la conclusion de la règle qui n'expose donc pas les variables internes.

$$\textsc{Seq}\ \frac{G, H, bs \vdash \boldsymbol{e} \Downarrow H(\boldsymbol{x})}{G, H, bs \vdash \boldsymbol{x} = \boldsymbol{e}}$$

$$\textsc{Snode}\ \frac{\texttt{node}(G, f) \doteq n \qquad H(n.\mathbf{in}) = \boldsymbol{xs} \qquad H(n.\mathbf{out}) = \boldsymbol{ys} \qquad \forall eq \in n.\mathbf{eqs},\ G, H, (\texttt{base-of}\ \boldsymbol{xs}) \vdash eq}{G \vdash f(\boldsymbol{xs}) \Downarrow \boldsymbol{ys}}$$

FIGURE 11 – Sémantique de l'équation et du nœud

3 Transcription et correction du système d'horloges

Dans cette section nous présentons la transcription, dernière étape du processus de normalisation du langage. Cette passe présente peu d'intérêt algorithmique car il suffit de vérifier que le programme d'entrée est normalisé puis d'appliquer une transformation directe sur chaque constructeur. Néanmoins un point délicat est soulevé lors de la preuve de préservation sémantique que nous résolvons en prouvant la correction du système d'horloges. Cette propriété s'appliquant à tout programme Lustre, même non normalisé, nous l'utilisons également dans la preuve de correction d'une des passes en amont. Il est plus facile de motiver le besoin d'une telle propriété dans le cadre de la transcription, c'est pourquoi nous préférons la décrire au préalable.

En Lustre, les deux membres (gauche et droit) d'un `fby` sont des expressions qui produisent des flots et l'élaboration garantit que ces deux expressions possèdent la même annotation d'horloge. La sémantique du `fby` est formalisée en Coq par la relation co-inductive fby présentée dans la figure 12. Celle-ci utilise un opérateur fby1 qui retarde chaque valeur présente dans le flot de droite à la prochaine présence, en la conservant dans son premier argument. L'opérateur fby permet lui d'initialiser le délai avec la première valeur présente du flot de gauche. On note que les valeurs du flot xs sont ignorées mais pas leur statut (présent/absent), ce qui force la synchronisation du statut des valeurs successives de xs et ys. La forme de ces définitions est classique, cf., par exemple, Colaço et Pouzet [11, §3.2].

$$\frac{\mathsf{fby1}\ v\ xs\ ys \doteq vs}{\mathsf{fby1}\ v\ (\langle\rangle \cdot xs)\ (\langle\rangle \cdot ys) \doteq \langle\rangle \cdot vs} \qquad \frac{\mathsf{fby1}\ y\ xs\ ys \doteq vs}{\mathsf{fby1}\ v\ (\langle x\rangle \cdot xs)\ (\langle y\rangle \cdot ys) \doteq \langle v\rangle \cdot vs}$$

$$\frac{\mathsf{fby}\ xs\ ys \doteq vs}{\mathsf{fby}\ (\langle\rangle \cdot xs)\ (\langle\rangle \cdot ys) \doteq \langle\rangle \cdot vs} \qquad \frac{\mathsf{fby1}\ y\ xs\ ys \doteq vs}{\mathsf{fby}\ (\langle x\rangle \cdot xs)\ (\langle y\rangle \cdot ys) \doteq \langle x\rangle \cdot vs}$$

FIGURE 12 – Relation co-inductive fby

En NLustre, le membre gauche du `fby` est une constante qui n'est pas interprétée comme un flot mais comme un paramètre d'initialisation statique, ce qui facilite ensuite la génération de code impératif en fournissant directement la valeur d'initialisation pour la cellule mémoire correspondante. L'interprétation du `fby`, simplifiée, est alors donnée par une fonction totale de type `value` × `stream value` → `stream value` présentée sous forme relationnelle en figure 13.

$$\frac{\mathsf{fby}_{\mathsf{NL}}\ v\ ys = vs}{\mathsf{fby}_{\mathsf{NL}}\ v\ (\langle\rangle \cdot ys) = \langle\rangle \cdot vs} \qquad \frac{\mathsf{fby}_{\mathsf{NL}}\ y\ ys = vs}{\mathsf{fby}_{\mathsf{NL}}\ v\ (\langle y\rangle \cdot ys) = \langle v\rangle \cdot vs}$$

FIGURE 13 – Définition relationnelle de $\mathsf{fby}_{\mathsf{NL}}$

Cette distinction sémantique se propage également dans les autres constructions. Considérons l'équation `x = true fby (not x)`, valide dans les deux langages. En Lustre, l'expression `true` est une constante cadencée sur l'horloge de base et produit le flot $\langle\mathrm{T}\rangle \cdot \langle\mathrm{T}\rangle \cdot \langle\mathrm{T}\rangle \cdots$. En considérant l'interprétation du fby, on déduit que le seul flot possible pour la variable x est $\langle\mathrm{T}\rangle \cdot \langle\mathrm{F}\rangle \cdot \langle\mathrm{T}\rangle \cdot \langle\mathrm{F}\rangle \cdots$. En NLustre en revanche, la valeur `true` n'est que le paramètre initial de $\mathsf{fby}_{\mathsf{NL}}$ et n'impose pas de rythme au flot de x. Par conséquent, tout flot de la forme $(\langle\rangle^* \cdot \langle\mathrm{T}\rangle \cdot \langle\rangle^* \cdot \langle\mathrm{F}\rangle)^\omega$ est une interprétation valide de x.

Cet aspect non déterministe n'est pas souhaitable dans le cadre de la programmation de systèmes critiques car il laisse au compilateur le choix du comportement à adopter. Il est donc nécessaire d'imposer une contrainte supplémentaire sur la sémantique NLustre. La solution adoptée dans Vélus est de faire correspondre le flot d'une expression avec l'annotation d'horloge de celle-ci, déterminée lors de l'élaboration. On introduit donc une notion de sémantique pour les horloges, interprétées par des flots booléens. L'horloge de base $\bullet$ doit s'évaluer en le flot de base de l'environnement et une horloge échantillonnée `ck on x` ne doit s'évaluer en T que lorsque la sous-horloge `ck` est vraie, que la valeur de `x` est présente et égale à T ; elle s'évalue en F si `ck` est vraie et la condition fausse ou si `ck` est F et `x` absente ; elle est indéfinie sinon. On surcharge la notation $H, bs \vdash ck \Downarrow b$ pour dire que l'horloge ck produit le flot booléen b.

L'intégration de la sémantique des horloges se fait au niveau des expression en NLustre. Ici tl désigne l'un des deux destructeurs de flot, qu'on étend aux environnements par composition.

$$\frac{H, bs \vdash e \Downarrow \langle v\rangle \cdot s \qquad H, bs \vdash ck \Downarrow \mathrm{T} \cdot b \qquad \mathsf{tl}\ H, \mathsf{tl}\ bs \vdash e^{ck} \Downarrow s}{H, bs \vdash e^{ck} \Downarrow \langle v\rangle \cdot s}$$

$$\frac{H, bs \vdash e \Downarrow \langle\rangle \cdot s \qquad H, bs \vdash ck \Downarrow \mathrm{F} \cdot b \qquad \mathsf{tl}\ H, \mathsf{tl}\ bs \vdash e^{ck} \Downarrow s}{H, bs \vdash e^{ck} \Downarrow \langle\rangle \cdot s}$$

FIGURE 14 – Règles de l'alignement

Les règles données dans la figure 14 décrivent le principe d'*alignement* du flot d'une expression avec son horloge : à chaque instant, la valeur d'une expression est présente si et seulement si son horloge est vraie. Dans la suite, on notera $H, bs \vdash e^{ck} \Downarrow s$ pour signifier que l'expression e produit un flot s bien aligné avec l'horloge ck. En Lustre en revanche, les annotations d'horloges ne sont pas interprétées par le modèle sémantique, il faut démontrer que l'alignement en est une conséquence.

3.1 Correction du système d'horloges

Le prédicat d'alignement des horloges et des valeurs constitue le principal obstacle dans la preuve de préservation sémantique de la transcription. En plus d'imposer une sémantique déterministe aux programmes NLustre, l'alignement apporte une information de présence ou d'absence des valeurs à chaque instant qui s'avère importante lors de la preuve de correction du code impératif généré [7, §3.4]. Le résultat central de la transcription consiste en une preuve de cette propriété, pour tout nœud Lustre bien cadencé, ordonnançable et possédant une sémantique.

Théorème 1. *Pour tout environnement H, pour tous flots $s_1, \ldots, s_n$, $s'_1, \ldots, s'_m$, pour tout nœud Lustre de signature* `node f` $(x_1^{ck_1}, \ldots, x_n^{ck_n})$ `returns` $(y_1^{ck'_1}, \ldots, y_m^{ck'_m})$*, si*

- $bs \equiv$ `base-of`$(s_1, \ldots, s_n)$
- $H, bs \vdash x_1^{ck_1} \Downarrow s_1, \ldots, x_n^{ck_n} \Downarrow s_n$
- `f`$(s_1, \ldots, s_n) \Downarrow s'_1, \ldots, s'_m$
- $H \vdash y_1 \Downarrow s'_1, \ldots, y_m \Downarrow s'_m$

alors

$H, bs \vdash y_1^{ck'_1} \Downarrow s'_1, \ldots, y_m^{ck'_m} \Downarrow s'_m$.

On note que les flots des entrées du nœud sont supposés alignés avec leurs horloges. Par un raisonnement inductif sur l'ensemble du programme, on peut se débarrasser de cette contrainte pour tous les nœuds à l'exception du nœud principal, qui reçoit ses entrées de l'extérieur. L'ajout de cette nouvelle hypothèse au théorème principal du compilateur est justifié et n'introduit pas de faiblesse dans la preuve de correction. Il n'est pas possible de garantir une sémantique pour une exécution dans laquelle les entrées du programme ne sont pas bien formées.

On peut aussi remarquer que ce théorème s'applique à tout nœud Lustre ordonnançable (hypothèse discutée en section **??**) possédant une sémantique, et pas uniquement aux nœuds sous forme normalisée. Ce théorème peut donc être considéré comme une propriété des programmes Lustre, indépendamment de l'étape de transcription, dénotant la correction du système d'horloges : si un nœud est bien cadencé alors les flots des variables sont alignés avec leurs horloges.

3.2 Induction et ordonnancement

La preuve du théorème 1 s'effectue à plusieurs niveaux, équations puis expressions. L'alignement du flot d'une expression avec son horloge se démontre par induction structurelle, avec les informations du système d'horloges. Par définition, les constantes sont cadencées sur l'horloge • et sont donc alignées avec le flot de base du nœud. L'autre cas de base, la variable, est plus délicat : on ne dispose de l'alignement que s'il s'agit d'une variable d'entrée du nœud ou qu'elle a déjà été traitée dans le raisonnement. Il est alors nécessaire de choisir un ordre sur les variables du programme.

L'existence d'un tel ordre est une hypothèse raisonnable faite sur les équations Lustre et correspond à l'absence de définitions cycliques [13, §III.A]. Chaque cycle de dépendances dans un nœud doit contenir au moins un opérateur `fby`, ce qui permet de briser la circularité. On dit qu'une variable est *libre à gauche* d'une expression si elle n'apparaît pas à droite d'un `fby`. Dans ce contexte, une liste d'équations $eq_1, \ldots, eq_n$ est ordonnancée si toute variable libre à gauche d'une équation eq_i est définie soit comme une entrée du nœud, soit par une équation $eq_{j<i}$. Par exemple `x = true fby not x; y = z when x` est ordonnancée si `z` est une entrée du nœud, mais `y = 2 * y + z` n'est pas ordonnançable.

En raisonnant par récurrence sur les équations ordonnancées, on peut maintenir l'invariant suivant sur les variables libres à gauche : si la variable est déclarée d'horloge ck et produit le flot de valeurs s dans l'environnement H, alors ck est alignée avec s. En effet, l'équation eq_1 ne peut contenir que les variables d'entrées pour lesquelles on a l'invariant par hypothèse ; puis par ordonnancement, une variable libre à gauche dans une équation $eq_{i>1}$ a été définie par une précédente équation $eq_{j<i}$, ou bien c'est une entrée, et l'on peut utiliser l'hypothèse de récurrence pour obtenir l'alignement.

Il est important de noter que l'absence de dépendance circulaire entre les variables d'un programme n'implique pas l'ordonnançabilité de ses équations. Par exemple `x,y = (1,x)` est une équation valide en Lustre qui ne nous permet pas d'exploiter le raisonnement décrit précédemment. Cette limitation n'est pas un obstacle à la preuve de correction de la transcription car de telles équations ne sont pas normalisées. Cependant on souhaiterait l'éliminer afin d'obtenir un résultat plus général. Une piste envisageable serait de gérer plus finement les dépendances entre les variables au sein d'une équation. Une autre serait d'appliquer une passe de désimbrication des équations avant de commencer le raisonnement inductif du paragraphe précédent.

Contrôle de dépendances. L'ordonnançabilité d'un programme est établie par une analyse de graphe. À chaque équation du nœud est associé un sommet, et des arcs sont ajoutés pour chaque équation utilisant une variable définie par une autre. Vérifier l'ordonnançabilité du programme revient alors à vérifier l'absence de cycle dans le graphe.

4 Désimbrication et distributivité

On revient maintenant sur les deux passes qui précèdent la transcription et qui mettent un programme Lustre sous forme normalisée. Il faut d'abord désimbriquer les instanciations et les `fby`s, en les plaçant dans leurs propres équations. On profite aussi de cette passe pour distribuer les opérateurs `fby`, `when`, `merge` et `if-then-else` sur leurs listes d'opérandes. Cela permet de produire un code où chaque expression, sauf une instanciation, représente exactement un flot.

4.1 Schémas de compilation

On note $\lfloor e \rfloor = ([e'_1, ..., e'_m], \boldsymbol{eqs})$ la normalisation de l'expression e. Celle-ci produit une liste d'expressions, à cause de la distribution des opérateurs. Par exemple, dans l'exemple de la figure 2, la sous-expression `(edge, n) when ck` se normalise en `(edge when ck, n when ck)`. La normalisation produit de plus des équations contenant les sous-expressions qui ont été désimbriquées. Pour simplifier, on notera $([e'_1, \ldots, e'_m], \boldsymbol{eqs'}) \leftarrow \lfloor e_1, \ldots, e_n \rfloor$ la normalisation d'une liste d'expressions $e_1, \ldots, e_n$, où $e'_1, \ldots, e'_m$ est la concaténation des listes d'expressions produites, et $\boldsymbol{eqs'}$ l'union des listes d'équations.

On présente dans la figure 15 les schémas de normalisation pour quelques cas. Les cas de la constante et de la variable ne changent pas l'expression et n'introduisent pas de nouvelle équation. Le cas des opérateurs binaires est défini par récursion, avec concaténation des équations introduites pour chaque sous-expression. Par contrainte de typage, on sait qu'une seule expression sera renvoyée par chaque appel récursif. Dans le cas du `when`, on effectue les appels récursifs, puis on distribue le `when` autour de chaque résultat de l'appel. Dans le cas du `fby`, il faut désimbriquer les expressions après distribution. On génère donc une nouvelle variable par expression et on renvoie comme expression la liste de ces variables. Pour l'instanciation, on ne distribue pas, mais on doit désimbriquer l'expression produite après les appels récursifs. On génère alors le nombre de variables correspondant au nombre de flots renvoyés par le nœud

instancié. Les cas du `merge` et du `if-then-else` ne sont pas détaillés car similaires au cas du `fby`. Cela dit, on évite de désimbriquer les expressions de contrôle directement imbriquées, comme spécifié dans la grammaire de la figure 6.

$$
\begin{aligned}
\lfloor \mathtt{c} \rfloor &= ([\mathtt{c}], [\,]) \\
\lfloor \mathtt{x} \rfloor &= ([\mathtt{x}], [\,]) \\
\lfloor e_1 \oplus e_2 \rfloor &= ([e'_1], \boldsymbol{eqs}'_1) \leftarrow \lfloor e_1 \rfloor \\
&\quad\ ([e'_2], \boldsymbol{eqs}'_2) \leftarrow \lfloor e_2 \rfloor \\
&\quad\ ([e'_1 \oplus e'_2], \boldsymbol{eqs}'_1 \cup \boldsymbol{eqs}'_2) \\
\lfloor (e_1, \ldots, e_n)\ \mathtt{when}\ b \rfloor &= ([e'_1, \ldots, e'_m], \boldsymbol{eqs}') \leftarrow \lfloor e_1, \ldots, e_n \rfloor \\
&\quad\ ([e'_1\ \mathtt{when}\ b, \ldots, e'_m\ \mathtt{when}\ b], \boldsymbol{eqs}') \\
\lfloor (e_1, \ldots, e_n)\ \mathtt{fby}\ (f_1, \ldots, f_m) \rfloor &= ([e'_1, \ldots, e'_k], \boldsymbol{eqs}'_1) \leftarrow \lfloor e_1, \ldots, e_n \rfloor \\
&\quad\ ([f'_1, \ldots, f'_k], \boldsymbol{eqs}'_2) \leftarrow \lfloor f_1, \ldots, f_m \rfloor \\
&\quad\ ([x_1, \ldots, x_k], [x_1 = e'_1\ \mathtt{fby}\ f'_1, \ldots, x_k = e'_k\ \mathtt{fby}\ f'_k] \cup \boldsymbol{eqs}'_1 \cup \boldsymbol{eqs}'_2) \\
\lfloor f(e_1, \ldots, e_n) \rfloor &= ([e'_1, \ldots, e'_m], \boldsymbol{eqs}') \leftarrow \lfloor e_1, \ldots, e_n \rfloor \\
&\quad\ ([x_1, \ldots, x_k], [(x_1, \ldots x_k) = f(e'_1, \ldots, e'_m)] \cup \boldsymbol{eqs}')
\end{aligned}
$$

FIGURE 15 – Schémas de normalisation des expressions

On ajoute à ces cas généraux des traitements particuliers, permettant de minimiser les transformations du programme. Si l'on rencontre une instanciation directement à droite d'une équation, on n'a pas besoin d'introduire de nouvelle équation, puisque l'équation est déjà sous forme normalisée. Il en va de même pour les `fby`s directement à droite d'une équation. Eviter les transformations inutiles permet de produire des programmes plus courts et lisibles. On a aussi pu prouver formellement que la fonction est idempotente : pour tout programme G, $\lfloor \lfloor G \rfloor \rfloor = \lfloor G \rfloor$.

4.2 Génération de variables

Comme on l'a vu plus haut, désimbriquer un `fby` ou une instanciation nécessite l'introduction de nouvelles variables locales dans le nœud. Dans le contexte de programmation fonctionnelle pure de Coq, on implémente la génération de variables par une monade d'état.

L'état manipulé est de type `fresh_st = (ident × list (ident × (ty × ck)))`. Celui-ci permet de conserver le prochain identifiant à générer, ainsi qu'une liste des variables générées, chacune accompagnée de son annotation de type et d'horloge. Cette liste contient les nouvelles variables locales introduites dans le nœud.

On exploite la représentation des identifiants par des entiers positifs. Chaque appel à la fonction `fresh_ident : (ty × ck) → fresh_st → (ident × fresh_st)` renvoie un nouvel identifiant, ainsi qu'un nouvel état avec l'identifiant à générer incrémenté, et l'identifiant juste généré sauvegardé dans la liste. Ainsi tous les identifiants générés par la monade sont plus grands que l'identifiant utilisé pour initialiser la monade. Il est donc important de bien choisir cet identifiant d'initialisation. La solution la plus simple est de prendre un identifiant plus grand que tout ceux apparaissant dans le nœud. Ainsi l'on sait que tous les identifiants générés sont plus grands et donc différents.

En plus de résoudre le problème de la génération de variables dans un contexte fonctionnel pur, cette manipulation explicite d'un état permet de raisonner sur la sémantique des équations.

4.3 Raisonnement sur la sémantique

On décrit maintenant la preuve de correction de cette passe de normalisation. Il s'agit de montrer que la sémantique d'un programme est préservée par la transformation.

Théorème 2 (Préservation de la sémantique)**.** *La passe de désimbrication et distributivité préserve la sémantique des programmes.*

$$\forall G\ f\ \boldsymbol{xs}\ \boldsymbol{ys},\quad G \vdash f(\boldsymbol{xs}) \Downarrow \boldsymbol{ys} \quad \Rightarrow \quad \lfloor G \rfloor \vdash f(\boldsymbol{xs}) \Downarrow \boldsymbol{ys}$$

Intéressons-nous au cœur de la fonction de normalisation : le traitement des expressions. Il faut prouver que la normalisation d'une expression e ayant une sémantique $\boldsymbol{vs}$ produit des expressions $\mathbf{es}'$ ayant la même sémantique. On veut également que les équations $\boldsymbol{eqs}'$ produites aient une sémantique. Le noyau de cette propriété d'induction est donné ci-dessous.

$$\begin{aligned}
&\lfloor e \rfloor = (\boldsymbol{es}', \boldsymbol{eqs}') \Rightarrow \\
&G, H, bs \vdash e \Downarrow \boldsymbol{vs} \Rightarrow \\
&\exists H',\ (H \sqsubseteq H' \ \wedge\ G, H', bs \vdash \boldsymbol{es}' \Downarrow \boldsymbol{vs} \ \wedge\ G, H', bs \vdash \boldsymbol{eqs}')
\end{aligned}$$

Lors d'une opération de désimbrication, on va ajouter une nouvelle variable au nœud. Pour donner une sémantique aux nouvelles équations et expressions, il faut donc étendre l'historique H avec cette nouvelle variable. La sémantique des $\boldsymbol{es}'$ et $\boldsymbol{eqs}'$ sera donc donnée en fonction d'un nouvel historique H'. On veut que H' « raffine » H, noté $H \sqsubseteq H'$. On reviendra sur cette notion de raffinement plus loin.

La preuve est établie par induction sur la syntaxe des expressions. Les cas intéressants sont ceux du `fby` et de l'instanciation, où la désimbrication a lieu. Quand on désimbrique la sous-expression e_x de l'expression e, on introduit une nouvelle équation $x = e_x$ et on substitue x à e_x dans e. On note le résultat de cette substitution $e' = e[e_x \mapsto x]$. On veut prouver que si $G, H, bs \vdash e \Downarrow \boldsymbol{vs}$, alors il existe un nouvel historique H' étendu avec la nouvelle variable x, tel que $G, H', bs \vdash e' \Downarrow \boldsymbol{vs}$. De plus, H' doit satisfaire la sémantique imposée par la nouvelle équation produite par la désimbrication, $G, H', bs \vdash x = e_x$. Pour répondre à ces contraintes nous construisons explicitement H', une extension de H dans laquelle on associe à la variable x le flot produit par e_x. En notant ce flot v_x, on a $G, H, bs \vdash e_x \Downarrow v_x$.

Comme x est un nom frais issu de la monade de génération de noms, on peut prouver qu'il n'apparaît pas dans le domaine de H. On sait alors de H' que

— $H'(x) = v_x$ et

— $\forall y\ v, H(y) = v \rightarrow H'(y) = v$ (H' raffine H, noté $H \sqsubseteq H'$).

C'est suffisant pour prouver que H' donne une sémantique à l'expression et à l'équation produite. En effet, il est facile de prouver que si H' raffine H, alors H' donne la même sémantique que H à toute expression, en particulier e_x et e. De plus, on sait que $H'(x) = v_x$, qui est bien le flot associé à e_x.

Des travaux précédents [1] avaient mis la notion de substitution au cœur du raisonnement. Au contraire, l'objet central de notre induction est l'historique H donné comme un existentiel.

Après avoir normalisé toutes les équations, on obtient un H' raffinant H et satisfaisant les contraintes des équations produites. Ce H' peut servir comme existentiel donnant la sémantique du nœud, en suivant la règle SNODE donnée en figure 11. Comme H' raffine H, on sait que les sorties du nœud normalisé sont bien les mêmes qu'avant normalisation.

On donne en **??** la forme désimbriquée du nœud `count_down` présenté en figure 2. On illustre en **??** comment la forme désimbriquée du programme se comporte. Le nouvel historique associe un flot de valeurs à la variable `$114` et toujours le même flot à `cpt`.

```
node count_down(res:bool; n:int)
returns (cpt:int)
let $114 = n fby (cpt - 1);
    cpt = if res then n else $114;
tel
```

FIGURE 16 – `count_down` désimbriqué

res	F	T	F	F	F	F	T	F	···
n	3	3	3	3	3	3	3	3	···
$114	3	2	2	1	0	-1	-2	2	···
cpt	3	3	2	1	0	-1	3	2	···

FIGURE 17 – Exécution de `count_down`

Traiter la distributivité ajoute une petite difficulté. L'une des propriétés de la fonction de normalisation est que si $\lfloor e \rfloor = (\boldsymbol{es'}, \boldsymbol{eqs'})$, alors la longueur de $\boldsymbol{es'}$ correspond au nombre de flots issus de e. Dans l'implémentation des schémas donnés dans la figure 15, les annotations de typage sont utilisées pour générer les $\boldsymbol{es'}$. Il faut donc savoir que celles-ci sont correctes, c'est-à-dire, que e est bien typée. C'est une hypothèse peu coûteuse, puisque les programmes à normaliser ont été élaborés au préalable. L'élaboration étant prouvée correcte, on sait que ces programmes sont effectivement bien typés.

5 Explicitation des initialisations

La passe de normalisation précédente met un programme sous forme désimbriquée et distribuée. Nous souhaitons aussi que les `fby`s ne soient initialisés que par des constantes et non par des expressions. C'est nécessaire pour compiler les `fby`s dans la suite de la chaine.

5.1 Schéma de compilation

Nous définissons un sous-ensemble des expressions, les « constantes lentes », composées de constantes pouvant être entourées d'opérateurs `when`. Ces `when`s permettent de maintenir le bon cadencement des équations contenant des `fby`s. Ils sont ensuite éliminés lors de la transcription.

Notons c^{ck} la constante c entourée de `when`s de manière à être cadencée sur l'horloge ck. Par exemple, $\texttt{true}^{\bullet\ \texttt{on b on c}}$ représente `true when b when c`. Notons, par ailleurs, $\texttt{def}_{ty}$ une constante « par défaut » du type ty. Notre langage manipulant des types booléens, entiers et réels, il est toujours possible de choisir une constante du type approprié (false, 0, 0.0).

On veut transformer une équation `x = (e0 fby e)`^{ck} où `e0` n'est pas déjà une constante lente. L'horloge ck de l'expression est importante, puisqu'elle permettra de générer les constantes lentes. La transformation donne alors les trois équations suivantes.

$$\lfloor \texttt{x = (e0 fby e)}^{ck} \rfloor_{fby} = \begin{cases} \texttt{xinit = true}^{ck}\ \texttt{fby false}^{ck}\texttt{;} \\ \texttt{px =}\ \texttt{def}_{ty}^{ck}\ \texttt{fby e;} \\ \texttt{x = if xinit then e0 else px;} \end{cases}$$

Le rôle de la première équation est de choisir le premier instant de présence pour le flot d'horloge de ck, l'horloge de l'équation à transformer. La deuxième équation retarde le flot de `e`. La troisième utilise le flot d'initialisation `xinit` pour choisir entre l'expression d'initialisation `e0` et le flot retardé de `e`. Ces trois équations sont effectivement normalisées, car seules des constantes lentes apparaissent à gauche des `fby`s.

Un exemple de cette transformation est celle de `count_down` désimbriquée, voir la **??**, en forme normalisée, voir la figure 3.

Pour générer les nouvelles variables `xinit` et `px`, nous réutilisons la monade de génération de noms présentée dans la section 4.2.

Remarquons que pour deux équations contenant des `fby`s cadencées sur la même horloge, le schéma ci-dessus produirait deux équations d'initialisation identiques. Par exemple, normaliser l'équation `(x, y) = (x0, y0) fby (y, x)` causerait cette redondance. Cela est à éviter, en particulier parce que tout `fby` demandera un espace mémoire dans le programme impératif issu de la compilation. On utilise donc l'état de la monade de génération de nom pour mémoïser et réutiliser les équations d'initialisation déjà générées. Cette optimisation complexifie les preuves de correction.

5.2 Raisonnement sur la sémantique

Comme pour la passe précédente, nous montrons que la sémantique d'un programme est préservée par cette passe de normalisation.

Théorème 3 (Préservation de la sémantique)**.** *L'explicitation des initialisations préserve la sémantique des programmes.*

$$\forall G\ f\ \boldsymbol{xs}\ \boldsymbol{ys},\quad G \vdash f(\boldsymbol{xs}) \Downarrow \boldsymbol{ys} \quad \Rightarrow \quad \lfloor G \rfloor_{fby} \vdash f(\boldsymbol{xs}) \Downarrow \boldsymbol{ys}$$

Cette fois, le cœur du problème est de prouver que la sémantique de l'équation `x = e0 fby e` est préservée par les nouvelles équations introduites. Comme dans la preuve du **??**, il faut étendre l'historique donnant la sémantique des expressions pour inclure les nouveaux noms. Cette transformation ajoute aussi une nouvelle difficulté. À partir de la sémantique d'un `fby`, il faut donner la sémantique pour une construction incluant des constantes lentes, deux `fby`s, et un `if-then-else`.

Constantes lentes $\mathtt{c}^{ck}$. Donner la sémantique d'une constante `c` est trivial. Comme on peut voir dans la figure 8, il s'agit du flot `const bs c`. Il est plus difficile de donner une sémantique à $\mathtt{c}^{ck}$. Prenons l'exemple de $\mathtt{c} \bullet^{\mathtt{on\ b1\ on\ b2}} = \mathtt{c\ when\ b1\ when\ b2}$. En se rappelant la règle du `when`, on voit qu'il est nécessaire que le flot issu de `c when b1` soit aligné avec le flot de `b1`. Par induction sur ck, on prouve que $\mathtt{c}^{ck}$ a une sémantique si l'horloge ck en a une.

Pour obtenir cette sémantique, on utilise le résultat d'alignement développé dans la section 3.1 pour prouver l'existence d'un flot d'horloge b pour l'annotation ck. Le flot de $\mathtt{c}^{ck}$ est alors $\mathsf{const}\ b\ c$, et est aligné avec les flots issus de `e0` et `e`. Utiliser cette preuve demande de nouvelles hypothèses sur notre programme. Il faut savoir qu'il est bien typé, bien cadencé et ordonnaçable.

Équations de délai. On veut maintenant donner les sémantiques de $\mathtt{true}^{ck}$ `fby` $\mathtt{false}^{ck}$ et $\mathtt{def}_{ty}^{ck}$ `fby e`. Les flots correspondant à ces expressions peuvent être explicitement construits par la fonction totale $\mathsf{fby}_{\mathsf{NL}}$. La figure 17 en présente une définition fonctionnelle qui est équivalente à la définition par prédicats co-inductifs de la figure 13.

La fonction $\mathsf{fby}_{\mathsf{NL}}$ correspond à l'opérateur fby de Lustre donné dans la figure 12. En effet, si $y0$ est un flot constant de v aligné avec y, on a `fby y0 y` $\doteq$ $\mathsf{fby}_{\mathsf{NL}}$ `v y`. En particulier, les flots $\mathtt{v}_{init}$ `=` $\mathsf{fby}_{\mathsf{NL}}$ `true (const bs false)` et $\mathtt{v}_{px}$ `=` $\mathsf{fby}_{\mathsf{NL}}\ \mathtt{def}_{ty}$ `y` (avec `y` le flot associé à `e`) sont bien les flots associés à ces deux nouvelles expressions.

Sélection de valeur avec `if-then-else`. Il reste à déduire de la sémantique de l'équation initiale `x = e0 fby e` une sémantique pour `x = if xinit then e0 else px`. Inverser l'hypothèse de sémantique établit `fby y0 y` $\doteq$ `z`, où `y0`, `y` et `z` sont les flots associés à `e0`, `e` et `x`. On peut alors prouver la relation `ite (`$\mathsf{fby}_{\mathsf{NL}}$ `true (const bs false)) y0 (`$\mathsf{fby}_{\mathsf{NL}}\ \mathtt{def}_{ty}$ `y)` avec

$$\mathsf{fby_{NL}}\ v\ (\langle\rangle \cdot xs) = \langle\rangle \cdot\ \mathsf{fby_{NL}}\ v\ xs$$
$$\mathsf{fby_{NL}}\ v\ (\langle x\rangle \cdot xs) = \langle v\rangle \cdot\ \mathsf{fby_{NL}}\ x\ xs$$

FIGURE 18 – Définition fonctionnelle de $\mathsf{fby_{NL}}$

`res`	F	T	F	F	F	F	T	F	···
`n`	3	3	3	3	3	3	3	3	···
$214	T	F	F	F	F	F	F	F	···
$215	0	2	2	1	0	-1	-2	2	···
$114	3	2	2	1	0	-1	-2	2	···
`cpt`	3	3	2	1	0	-1	3	2	···

FIGURE 19 – Exécution de `count_down` normalisé

les règles co-inductives du `ite` de la figure 9. On construit cette preuve par co-induction, en suivant les règles de fby de la figure 12.

Dans la **??**, nous présentons comme exemple de ces nouveaux flots la trace de la forme normalisée de `count_down`.Le flot de $114, qui est maintenant construit à partir des flots de $214 et $215, est inchangé.

5.3 Préservation de l'ordonnançabilité

Comme expliqué dans la section précédente, la preuve de préservation de la sémantique lors de l'explicitation des initialisations utilise la preuve de correction du système d'horloges. Celle-ci utilise une hypothèse d'ordonnançabilité sur le programme. Il faut donc savoir que le programme G traité par cette passe est ordonnançable. Nous obtenons cette propriété en exécutant le contrôle de dépendances affiché dans la figure 1 et décrit dans la section 3.2.

Cependant, on veut également savoir que le programme $\lfloor G \rfloor_{fby}$ produit par l'explicitation des initialisations de G est ordonnançable car cette propriété est requise dans la passe suivante, la transcription.

Il faut donc prouver que la passe d'explicitation des initialisations préserve l'ordonnançabilité. L'optimisation introduite dans la section 5.1 rend la preuve plus complexe que ce à quoi on pourrait s'attendre. En effet, les variables correspondant aux initialisations introduites peuvent être utilisées par plusieurs expressions. Pour rétablir un ordre d'exécution sur le nouveau programme, il faut montrer que l'on peut placer ces équations avant toutes celles qui en dépendent. De plus, les équations d'initialisation dépendent elles-mêmes d'autres variables à cause de `when`s introduits.

Pour prouver l'existence d'un tel ordre, il faut manipuler un invariant spécifiant qu'il existe un ordonnancement pour les équations du nœud dans lequel toutes les équations d'initialisation apparaissent avant les équations de la forme `x = e0 fby e` cadencées sur la même horloge. Mécaniser et manipuler cet invariant en Coq introduit quelques technicités qu'on ne détaillera pas ici. Cependant, il permet effectivement de prouver la préservation de l'ordonnançabilité.

6 Conclusion

Nous avons présenté l'implémentation et les preuves de correction dans un assistant de preuve d'une passe de normalisation pour le langage Lustre. Cette passe a été intégrée dans notre prototype. Le résultat est une chaîne de compilation pour le langage de la figure 5 et un théorème qui lie la sémantique d'un programme source, c'est-à-dire, la relation entre les flots des entrées et des sorties, à celle du code généré. En fait, le prototype prend également en compte la réinitialisation modulaire [6], mais son traitement ne pose pas de pro-

blème particulier et nous ne présentons pas les détails ici. De même, nous traitons l'opérateur d'initialisation (−>) en adaptant directement l'approche décrite dans la section 5, puisque `e0 -> e` ≡ `if (true fby false) then e0 else e`. Cet opérateur est normalement utilisé avec le délai non initialisé, « `pre` », qui ne fait pas encore partie de notre langage. En plus des questions posées pour le `fby` de NLustre dans la section 3, l'inclusion de `pre` demanderait de considérer la sous-spécification de sa valeur initiale.

Dans la plupart des cas, l'ajout des listes au langage source ne complique pas vraiment les preuves dans Coq. On utilise quelques lemmes sur `Forall`n pour éviter des inductions imbriquées et pour se concentrer sur un unique flot arbitraire. Dans certains cas, l'absence d'un lien biunivoque entre syntaxe (expressions) et sémantique (flots correspondants) rendent les preuves encore plus fastidieuses. Des problèmes plus profonds sont posés par la possibilité qu'un nœud prenne des entrées et produise des sorties sur des cadences moins rapides que son horloge de base. Il a fallu étendre la formalisation d'horloges [8] pour incorporer des horloges « anonymes » qui, dans un programme non normalisé, représentent les flots booléens provenant d'instances de nœuds imbriquées. Après quelques itérations, nous avons fini par trouver une formalisation assez simple. Mais, à part exiger que les entrées et les sorties partagent la même cadence, on ne peut pas éviter de passer entre historiques locaux augmentés avec flots anonymes et historiques internes à une instance de nœud, tout en substituant les variables passées en argument aux paramètres formels. Et cela rend le raisonnement beaucoup plus difficile.

Travaux connexes. Nous avons implémenté une passe de normalisation directement dans Coq. Une approche différente a été poursuivie par C. Auger et ses collègues [1, 2]. Ils découpent leur algorithme en deux parties. La première introduit de nouvelles équations et la seconde élimine les tuples [1, §7.1]. Ils n'ont pas besoin, comme nous, d'expliciter des initialisations, car dans leur langage source [1, §5.1] les `fby`s ont toujours une constante à gauche. La première passe appelle une fonction externe qui calcule les équations à introduire et la substitution à appliquer. Le résultat est ensuite validé par une fonction Coq facile à implémenter [1, §7.2.1].

Boulmé et Hamon [4] montrent comment spécifier un plongement peu profond dans Coq d'un language plus riche que Lustre. Dans leur approche, la propriété d'alignement est garantie par un encodage direct dans le système de types de Coq. Nous ne savons pas exploiter cette approche dans le plongement profond nécessaire pour spécifier un compilateur.

Travail futur. Alors que l'alignement des flots avec leurs horloges doit être une propriété de tout programme Lustre qui vérifie les conditions de typage, la preuve décrite dans la section 3.1 ne la démontre que pour les programmes dans lesquels il est possible d'ordonner les équations d'un nœud d'après leurs interdépendances. De même, le contrôle de dépendances est général, il vérifie l'absence de cycles instantanés entre variables, mais son témoin est traduit dans un prédicat sur la possibilité d'ordonner strictement les équations. C'est une des raisons pour lesquelles ce contrôle est invoqué après la passe de désimbrication et distributivité (cela évite aussi de démontrer que la propriété est préservée par cette passe). Nous voudrions donc développer un principe d'induction basé sur une notion de causalité plus abstraite qui permettrait de raisonner sur les propriétés générales des programmes acceptés par notre compilateur. La préservation d'une telle notion de causalité devrait être plus facile à établir que pour celle de la section 5.3.

Remerciements. Ce travail a été soutenu par Bpifrance « Programme d'Investissements d'Avenir » dans le projet ES3CAP et le projet ANR JCJC FidelR 19-CE25-0014-01 « Fidelity in Reactive Systems Design and Compilation ».

Bibliographie

[1] Cédric Auger. *Compilation certifiée de SCADE/LUSTRE*. PhD thesis, Université Paris Sud 11, Orsay, France, April 2013.

[2] Cédric Auger, Jean-Louis Colaço, Gregoire Hamon, and Marc Pouzet. A formalization and proof of a modular Lustre code generator. In preparation, 2014.

[3] Dariusz Biernacki, Jean-Louis Colaço, Gregoire Hamon, and Marc Pouzet. 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)*, pages 121–130, Tucson, AZ, USA, June 2008. ACM Press.

[4] Sylvain Boulmé and Grégoire Hamon. Certifying synchrony for free. In Robert Nieuwenhuis and Andrei Voronkov, editors, *Proceedings of the 8th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2001)*, volume 2250 of *Lecture Notes in Elecrical Engineering*, pages 495–506, Havana, Cuba, December 2001. Springer.

[5] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. 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, June 2017. ACM Press.

[6] Timothy Bourke, Lélio Brun, and Marc Pouzet. 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, January 2020.

[7] Timothy Bourke, Pierre-Évariste Dagand, Marc Pouzet, and Lionel Rieg. Vérification de la génération modulaire du code impératif pour Lustre. In Julien Signoles and Sylvie Boldo, editors, *28^{ièmes} Journées Francophones des Langages Applicatifs (JFLA 2017)*, pages 165–179, Gourette, Pyrénées, France, January 2017.

[8] Timothy Bourke and Marc Pouzet. Arguments cadencés dans un compilateur Lustre vérifié. In Nicolas Magaud and Zaynah Dargaye, editors, *30^{ièmes} Journées Francophones des Langages Applicatifs (JFLA 2019)*, pages 109–124, Les Rousses, Haut-Jura, France, January 2019.

[9] Lélio Brun. *Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset*. PhD thesis, PSL Research University, June 2020.

[10] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. 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, September 2017. IEEE Computer Society.

[11] Jean-Louis Colaço and Marc Pouzet. Clocks as first class abstract types. In R. Alur and I. Lee, editors, *Proceedings of the 3rd International Conference on Embedded Software (EMSOFT 2003)*, volume 2855 of *Lecture Notes in Elecrical Engineering*, pages 134–155, Philadelphia, PA, USA, October 2003. Springer.

[12] Coq Development Team. *The Coq proof assistant reference manual*. Inria, 2020.

[13] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9) :1305–1320, September 1991.

[14] Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, April 2006.