Analyse de dépendance vérifiée pour un langage synchrone à flot de données

Timothy Bourke[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

Vélus est une formalisation d'un langage synchrone à flots de données et de sa compilation dans l'assistant de preuve Coq. Il inclut une définition de la sémantique dynamique du langage, un compilateur produisant du code impératif, et une preuve de bout en bout que le compilateur préserve la sémantique des programmes.

Dans cet article, on spécifie dans Vélus la sémantique de deux structures d'activation présentes dans les compilateurs modernes : switch et déclarations locales. Ces nouvelles constructions nécessitent une adaptation de l'analyse statique de dépendance de Vélus, qui produit un graphe acyclique comme témoin de la bonne formation d'un programme. On utilise ce témoin pour construire un schéma d'induction propre aux programmes bien formés. Ce schéma permet de démontrer le déterminisme du modèle sémantique dans Coq.

1 Introduction

Le projet Vélus [3, 4, 5] formalise un langage synchrone à flot de données dans l'assistant de preuve Coq [11]. Ce langage est un sous-ensemble du langage industriel Scade 6 [9], qui est utilisé pour la spécification de systèmes de contrôle embarqués critiques. Scade 6 s'appuie lui-même sur des langages de recherche tels que Lustre [6] et Lucid Synchrone [16]. Vélus intègre un compilateur produisant du code impératif Clight, langage d'entrée du compilateur vérifié CompCert [13], ainsi qu'une sémantique à flots de données pour son langage d'entrée, et une preuve de bout en bout mettant en relation cette sémantique et celle du code assembleur généré.

Dans cet article, on ajoute à Vélus deux nouvelles structures d'activation présentes dans Lucid Synchrone [7] et Scade 6 [9] : le `switch`, et le bloc de déclarations locales. La sémantique à flots de données du langage source de Vélus est étendue pour traiter ces constructions. De précédents travaux se sont concentrés sur la preuve de correction du schéma de compilation de Vélus. On a étendu ces preuves pour traiter le `switch` et les déclarations locales. Dans cet article, on s'intéresse plutôt à la preuve de propriétés dynamiques du modèle sémantique, comme l'adéquation des flots calculés aux définitions statiques d'un programme (typage des valeurs et échantillonage), ou son déterminisme : pour une entrée donnée, une seule sortie est possible. Ces propriétés dépendent de la bonne formation du programme et en particulier de l'absence de cycle de dépendance dans ses définitions. L'encodage de cette notion de dépendance dans Coq est rendue plus complexe par l'ajout de structures d'activation.

Un exemple : contrôle des portes d'une télécabine On présente un programme Lustre contrôlant l'ouverture et la fermeture de la porte d'une télécabine hypothétique. Un moteur pas à pas bipolaire est utilisé pour déplacer les portes le long de leurs rails. Le tableau de la figure 1 indique le signal à appliquer à chacun des pôles du moteur pour le faire tourner dans le sens horaire. Ce comportement peut être implémenté dans Vélus à l'aide de deux équations utilisant chacune l'opérateur de délai initialisé `fby`. Pour faire tourner le moteur dans le sens anti-horaire, il faut inverser cette séquence d'activation. Le nœud `control_moteur` gère ce contrôle de bas

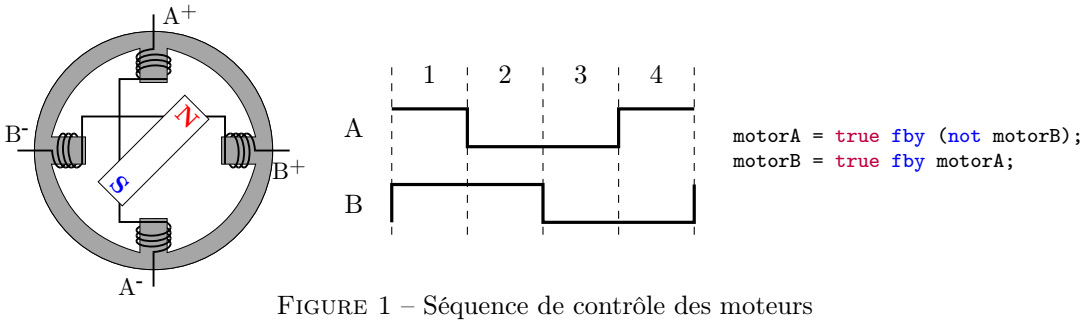

FIGURE 1 – Séquence de contrôle des moteurs

```
type moteurDir = Ouvre | Ferme | Bloque

node control_moteur(dir : moteurDir) returns (moteurA, moteurB : bool)
let
  switch dir
  | Ouvre do
    moteurA = true fby (not moteurB);
    moteurB = true fby moteurA;
  | Ferme do
    moteurA = true fby moteurB;
    moteurB = true fby (not moteurA);
  | Bloque do
    (moteurA, moteurB) = (false, false);
  end
tel
```

FIGURE 2 – Contrôle des moteurs

niveau. La structure d'activation `switch` active le bon comportement en fonction du paramètre `dir`, dans le sens horaire pour `Ouvre`, anti-horaire pour `Ferme`, ou stoppé pour `Bloque`.

Ce nœud est instancié par le nœud `control_porte`, présenté en figure 3, qui choisit la direction des moteurs. Quand la cabine est en dehors de la station, les portes sont fermées. La station est équipée de 8 repères fixés au rail. Quand la cabine passe un repère, l'entrée `repere` vaut `true`. Les portes doivent s'ouvrir entre les repères 1 et 3, rester ouvertes jusqu'au repère 6, puis se fermer avant le repère 8. Le compteur `nbRepere` est déclaré localement à la branche `true` du `switch`. Il est incrémenté chaque fois qu'un repère est détecté, jusqu'à `8`. Pour le mettre à jour, on utilise la construction `last`, introduite dans [8], qui permet d'accéder à la valeur précédente d'un flot, même calculée dans l'autre branche du `switch` interne. La déclaration `last` doit être accompagnée d'une expression d'initialisation, pour donner une valeur à `last nbRepere` au premier instant.

Un fabriquant de capteurs (hypothétique également) propose d'ajouter une redondance permettant de vérifier si les portes peuvent être ouvertes sans danger. Pour cela, il faut que la cabine soit proche du sol et que sa vitesse verticale soit suffisamment faible. Un capteur infrarouge placé sous la cabine permet de mesurer la distance avec le sol. Le manque de visibilité (neige, brouillard) peut parfois empêcher le capteur de fonctionner correctement pendant le trajet de la cabine. Pour pallier à ce problème, un accéléromètre permet de mesurer l'accélération verticale de la cabine. Cette mesure est alors intégrée pour calculer la vitesse courante, puis la position verticale. L'inconvénient de ce deuxième capteur est que de petites erreurs de mesure peuvent faire dériver l'estimation. Le nœud `ouvrir_ok` combine donc les données de ces capteurs pour vérifier que la porte peut être ouverte sans danger. Si le capteur infrarouge n'est pas

```
node control_porte(en_station : bool; repere : bool) returns (moteurA, moteurB : bool)
var dir : moteurDir;
let
  switch en_station
  | true do
    var last nbRepere : int = 0;
    let
      switch repere
      | true do nbRepere = (last nbRepere + 1) mod 8;
      | false do nbRepere = last nbRepere
      end;
      dir = if nbRepere > 0 and nbRepere < 3 then Ouvre
            else if nbRepere > 5 then Ferme
            else Bloque;
    tel
  | false do dir = Bloque;
  end;
  (moteurA, moteurB) = control_moteur(dir);
tel
```

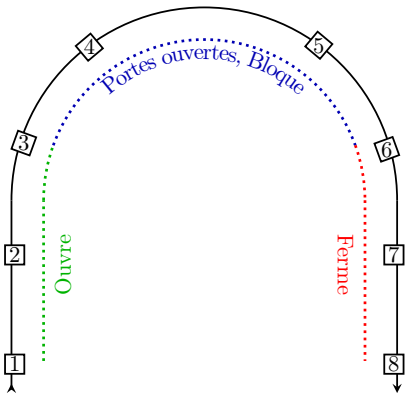

FIGURE 3 – Contrôle des portes

disponible, la valeur de `captdist` est négative et on utilise alors la valeur de l'accéléromètre `accely`. On note que les dépendances de ce programme commutent : dans la branche `true`, `vy` dépend de `y`, alors que c'est le contraire dans la branche `false`.

```
node ouvrir_ok(y_max, vy_max, captdist, accely, dt : double) returns (ok : bool)
var last vy : double = 0.0; last y : double = 0.0;
let
  switch (captdist > 0.0)
  | true do
    y = captdist;
    vy = (y - last y) / dt;
  | false do
    vy = last vy + accely * dt;
    y = last y + vy * dt;
  end;
  ok = y < y_max and vy < vy_max;
tel
```

FIGURE 4 – Vérification de la position et vitesse de la cabine

La suite présente une formalisation de la sémantique du `switch` et des déclarations locales dans Vélus. La section 3 adapte l'analyse de dépendance introduite dans [5, §2.3], et détaille son implémentation en Coq. On s'intéresse particulièrement au traitement des déclarations locales et de la commutation de dépendance. En section 4, on détaille le schéma d'induction introduit dans [5, §2.3], qu'on applique pour prouver que la sémantique de Vélus est déterministe.

La documentation du développement Coq décrit dans cet article, ainsi qu'une démonstration de l'exemple présenté ci-dessus sont accessibles à `https://velus.inria.fr/jfla2023`.

2 Sémantique à flots de données de Vélus

La syntaxe abstraite du langage d'entrée de Vélus est présentée en figure 5. Le langage des expressions étend celui présenté en [5]. Une expression peut faire référence à une constante primitive c ou énumérée C, un flot nommé avec x, et à sa valeur précédente avec `last` x. Les

opérateurs unaires et binaires arithmétiques et logiques habituels sont disponibles. Le `fby` introduit un délai initialisé par les expressions à sa gauche. La flèche d'initialisation `->` permet de donner une valeur particulière à l'instant initial. Le `when` échantillonne son premier argument selon la valeur d'un flot énuméré, et le `merge` fusionne des flots échantillonnés complémentaires. L'opérateur `case` est une généralisation de `if-then-else` pour une condition énumérée. Pour ces cinq dernières constructions, chaque opérande peut contenir une liste (non-vide) de sous-expressions (notée e^+) ; cela permet à une expression de définir plusieurs flots. Une expression peut enfin appeler un autre nœud du programme. Le compilateur supporte aussi le `reset` modulaire, traité formellement dans [4].

Dans [5], un bloc peut seulement définir une équation. On ajoute les structures d'activation `switch`, et les déclarations locales. Chaque déclaration est annotée par son type et son horloge statique. On y ajoute, pendant l'élaboration du programme, une étiquette unique pour chaque flot, ou deux pour un flot défini avec `last`. Par ailleurs, chaque branche d'un `switch` est annotée par une fonction de renommage de ces étiquettes σ. On verra plus loin comment ces étiquettes sont utilisées lors de l'analyse de dépendance d'un programme. Chaque nœud contient un bloc et spécifie ses entrées et sorties. Un programme est constitué d'une liste de définitions de types énumérés, suivie d'une liste non vide de nœuds.

$$
\begin{array}{l}
e ::= c \\
\quad \mid C \\
\quad \mid x \\
\quad \mid \texttt{last}\ x \\
\quad \mid \diamond\ e \quad \mid \quad e \oplus e \\
\quad \mid e^+\ \texttt{fby}\ e^+ \\
\quad \mid e^+\ \texttt{->}\ e^+ \\
\quad \mid e^+\ \texttt{when (}\ x\ \texttt{=}\ C\ \texttt{)} \\
\quad \mid \texttt{merge}\ x\ \texttt{(}\ C\ \texttt{=>}\ e^+\ \texttt{)}^+ \\
\quad \mid \texttt{case}\ e\ \texttt{of (}\ C\ \texttt{=>}\ e^+\ \texttt{)}^+ \\
\quad \mid f\ \texttt{(}\ e^+\ \texttt{)} \\
\quad \mid \texttt{(}\ \texttt{reset}\ f\ \texttt{every}\ e\ \texttt{)(}\ e^+\ \texttt{)}
\end{array}
\qquad
\begin{array}{l}
blk ::= x^+ \texttt{ = } e^+ \texttt{ ;} \\
\quad \mid \texttt{var}\ loc^*\ \texttt{let}\ blk^+\ \texttt{tel} \\
\quad \mid \texttt{switch}\ e\ \texttt{(}\ C\ \texttt{do}_\sigma\ blk^+\ \texttt{)}^+\ \texttt{end} \\
\\
d ::= x^{ck}_{ty} \\
\\
loc ::= d : \alpha_x \quad \mid \quad \texttt{last}\ d : (\alpha_x, \alpha_{\texttt{last}\,x}) = e : \\
\\
td ::= \texttt{type}\ tx\ \texttt{=}\ C^+ \\
\\
n ::= \texttt{node}\ f\ \texttt{(}\ (d : \alpha_x)^+\ \texttt{)}\ \texttt{returns}\ \texttt{(}\ (d : \alpha_x)^+\ \texttt{)}\ blk \\
\\
G ::= td^*\ n^+
\end{array}
$$

FIGURE 5 – Syntaxe abstraite de Lustre

2.1 Noyau du langage

On s'intéresse maintenant au modèle sémantique de Vélus, défini par un ensemble de règles, encodées comme des prédicats inductifs sur la syntaxe abstraite. Ce modèle contraint des flots infinis, représentés en Coq par le type co-inductif `Stream A := Cons : A → Stream A → Stream A`. Dans la suite, on note $x \cdot xs$ pour `Cons x xs`. Ces flots étant infinis, on utilise la relation d'équivalence entre flots $\equiv$ de la bibliothèque standard, dont la définition est rappelée ci-dessous.

Définition 1 (Equivalence de flots)**.**

$$x \cdot xs \equiv y \cdot ys \quad \textbf{\textit{ssi}} \quad x = y \quad \textbf{\textit{et}} \quad xs \equiv ys$$

On rappelle d'abord le modèle sémantique de Vélus déjà présenté dans [5]. Ces règles encodent, sous forme de relation entre flots, la sémantique décrite dans [10]. On présente en figure 6 les règles centrales de ce modèle. Le jugement $G, H, bs \vdash e \Downarrow vss$ se lit « sous l'environnement global G, l'historique H et l'horloge de base bs, l'expression e produit les flots vss ». Comme dit plus

haut, une expression peut produire plusieurs flots, et *vss* est donc une liste. Dans les règles suivantes, on note $[vs]$ quand cette liste ne contient qu'un élément. L'historique H associe un flot à chaque entrée, sortie et déclaration locale du nœud. Dans la suite, on note $x \in \mathsf{dom}(H)$ le jugement « il y a un flot associé à x dans H ». Si c'est le cas, $H(x)$ est défini comme le flot associé à x dans H. La première règle de figure 6 indique que, pour donner une sémantique à l'expression x, on lit le flot associé à x dans l'historique.

Un flot peut être échantillonné, c'est-à-dire que ses valeurs ne sont pas toujours présentes. Dans Vélus, on caractérise explicitement les absences (notées $\langle\!\rangle$) et présences (notées $\langle\!\langle v \rangle\!\rangle$). Le « rythme » d'un flot échantillonné est caractérisé par un flot booléen, son horloge. L'horloge de base *bs* est celle des flots les plus rapides du nœud. La fonction base-of permet de la calculer : elle est vraie à un instant si et seulement si au moins une entrée du nœud est présente. Intuitivement, le nœud est actif à chaque fois qu'au moins l'une de ses entrées est présente. L'horloge *bs* est utilisée par la deuxième règle pour donner le rythme d'une constante avec la fonction const.

Le jugement $G, H, bs \vdash blk$ indique que la sémantique d'un bloc *blk* peut être donnée sous le contexte G, H, bs. Un bloc ne produit pas de valeur, mais induit un ensemble de contraintes sur l'historique H. La règle pour l'équation est rappelée en figure 6 : les valeurs associées aux noms à gauche de l'équation doivent être celles calculées par les expressions à droite.

Enfin, l'environnement global G contient l'ensemble des nœuds du programme. Pour donner la sémantique d'un nœud de G, il faut donner un historique H respectant les contraintes des blocs du nœud. Les flots associés aux entrées et sorties du nœud sont lus dans H.

$$\frac{H(x) \equiv vs}{G, H, bs \vdash x \Downarrow [vs]} \qquad \frac{}{G, H, bs \vdash \mathsf{c} \Downarrow [\mathsf{const}\ bs\ \mathsf{c}]}$$

$$\begin{array}{lcl} \mathsf{const}\ (\mathtt{T} \cdot bs)\ \mathsf{c} & \equiv & \langle\!\langle \mathsf{c} \rangle\!\rangle \cdot \mathsf{const}\ bs\ \mathsf{c} \\ \mathsf{const}\ (\mathtt{F} \cdot bs)\ \mathsf{c} & \equiv & \langle\!\rangle \cdot \mathsf{const}\ bs\ \mathsf{c} \end{array}$$

$$\frac{G, H, bs \vdash es \Downarrow (vs_1, \ldots, vs_n) \qquad \forall i \in 1..n,\ H(x_i) \equiv vs_i}{G, H, bs \vdash (x_1, \ldots, x_n) = es}$$

$$\frac{\begin{array}{c} G(f) = \mathtt{node}\ f(x_1, \ldots, x_n)\ \mathtt{returns}\ (y_1, \ldots, y_m)\ blk \\ \forall i \in 1..n,\ H(x_i) \equiv xs_i \qquad \forall i \in 1..m,\ H(y_i) \equiv ys_i \qquad G, H, (\mathsf{base\text{-}of}\ (xs_1, \ldots, xs_n)) \vdash blk \end{array}}{G \vdash f(xs_1, \ldots, xs_n) \Downarrow (ys_1, \ldots, ys_m)}$$

FIGURE 6 – Règles sémantiques centrales [5]

La sémantique de chaque autre opérateur du language d'expressions est exprimée au moyen d'une fonction de flots. On donne ici l'exemple de l'opérateur de délai initialisé `fby`, qui permet d'accéder à la valeur des flots aux instants précédents. Ce fonctionnement est encodé par les opérateurs fby et fby1, présentés en figure 7. La première valeur présente du flot de gauche est produite par fby, tandis que les valeurs du deuxième flot sont conservées dans le premier argument de fby1 jusqu'à la présence suivante. Pour donner une sémantique à l'expression `es0 fby es1`, on distribue l'opérateur fby sur les flots produits par les sous-expressions.

On note que l'opérateur `fby` est insensible aux absences, c'est à dire qu'on peut en insérer ou supprimer sans changer son comportement fondamental. Il prend une valeur du flot de gauche et l'ajoute devant le flot de droite. Cette idée s'étend aux nœuds, qui associent aux absences en entrée des absences en sortie. Les nœuds sont donc eux aussi insensibles aux absences.

$$\begin{array}{lcl} \mathsf{fby}\ (\langle\rangle \cdot xs)\ (\langle\rangle \cdot ys) & \equiv & \langle\rangle \cdot \mathsf{fby}\ xs\ ys \\ \mathsf{fby}\ (\langle v_1\rangle \cdot xs)\ (\langle v_2\rangle \cdot ys) & \equiv & \langle v_1\rangle \cdot \mathsf{fby1}\ v_2\ xs\ ys \end{array}$$

$$\begin{array}{lcl} \mathsf{fby1}\ v_0\ (\langle\rangle \cdot xs)\ (\langle\rangle \cdot ys) & \equiv & \langle\rangle \cdot \mathsf{fby1}\ v_0\ xs\ ys \\ \mathsf{fby1}\ v_0\ (\langle v_1\rangle \cdot xs)\ (\langle v_2\rangle \cdot ys) & \equiv & \langle v_0\rangle \cdot \mathsf{fby1}\ v_2\ xs\ ys \end{array}$$

$$\frac{\begin{array}{c} G, H, bs \vdash es_0 \Downarrow (xs_1, \ldots, xs_n) \\ G, H, bs \vdash es_1 \Downarrow (ys_1, \ldots, ys_n) \\ \forall i \in 1...n,\ \mathsf{fby}\ xs_i\ ys_i \equiv vs_i \end{array}}{G, H, bs \vdash es_0\ \texttt{fby}\ es_1 \Downarrow (vs_1, \ldots, vs_n)}$$

FIGURE 7 – Sémantique de l'opérateur de délai initialisé `fby`

$$\frac{\begin{array}{c} G, H, bs \vdash e \Downarrow [cs] \\ \forall i,\ G, \mathsf{when}^{C_i}\ (H, bs)\ cs \vdash blks_i \end{array}}{G, H, bs \vdash \texttt{switch}\ e\ (C_i\ \texttt{do}\ blks_i)_i\ \texttt{end}}$$

$$\begin{array}{lcl} \mathsf{when}^C\ (\langle\rangle \cdot xs)\ (\langle\rangle \cdot ys) & \equiv & \langle\rangle \cdot \mathsf{when}^C\ xs\ ys \\ \mathsf{when}^C\ (\langle v\rangle \cdot xs)\ (\langle C\rangle \cdot ys) & \equiv & \langle v\rangle \cdot \mathsf{when}^C\ xs\ ys \\ \mathsf{when}^C\ (\langle v\rangle \cdot xs)\ (\langle C'\rangle \cdot ys) & \equiv & \langle\rangle \cdot \mathsf{when}^C\ xs\ ys \end{array}$$

FIGURE 8 – Sémantique du switch

2.2 Structure d'activation switch

Pour modéliser la sémantique du `switch`, on s'appuie sur l'idée d'insensibilité aux absences présentée ci-dessus. Une branche doit être insensible, dans le même sens que pour les `fby` et les nœuds, quand l'expression de garde ne correspond pas à l'étiquette de la branche. De plus, le `switch` doit combiner les flots produits par les branches, en choisissant à chaque instant les valeurs de la branche désignée par l'expression de garde, les valeurs de toutes les autres branches étant absentes. Dans les instants ou l'expression de garde est absente, toutes les valeurs de toutes les branches sont absentes.

On concrétise ces intuitions par la règle du `switch` définie en figure 8, à gauche. L'opérateur when, appliqué à l'historique et à l'horloge de base, est au cœur de cette modélisation. Pour chaque branche, il est appliqué avec le flot de l'expression de garde cs et l'étiquette C_i de la branche. Ainsi, il rend la branche insensible quand le flot de garde ne correspond pas à l'étiquette, et ne reporte les valeurs produites par la branche que quand elles sont présentes.

La définition de when pour un flot est donnée en figure 8, à droite. C'est une fonction partielle : ses deux entrées doivent être présentes en même temps. Pour appliquer when à l'horloge de base, un flot booléen sans absence explicite, ce critère est relaxé, et on interprète les absences dans le flot produit par when comme le booléen false. Les historiques étant représentés comme des fonctions partielles, on peut définir when appliqué à un historique par $(\mathsf{when}^C\ H\ cs)(x) = \mathsf{when}^C\ (H(x))\ cs$. Cela signifie que $(\mathsf{when}^C\ H\ cs)(x)$ est défini si et seulement si le flot associé à x dans H a le même rythme que cs. Ainsi, la sémantique des sous-blocs du `switch` est donnée dans un environnement plus restreint que H. Cela correspond d'ailleurs à un critère de typage statique du langage, que nous ne détaillerons pas ici.

2.3 Déclarations locales

Comme on l'a vu dans les exemples, l'utilisateur peut arbitrairement imbriquer des déclarations locales, y compris dans la branche d'un `switch`. On introduit en figure 10 une définition pour la sémantique d'un bloc de déclarations locales. La première prémisse de la règle spécifie le domaine d'un historique local H' qui associe un flot à chaque déclaration locale du bloc. Cet historique H' vient compléter l'historique global H pour donner une sémantique aux sous-blocs, comme indiqué dans la troisième prémisse. Cette formalisation est inspirée de celle présentée

dans [7, §3.2]. L'opérateur d'union d'historique + est défini par

$$(H_1 + H_2)(x) = \begin{cases} H_2(x) & \text{si } x \in \mathsf{dom}(H_2) \\ H_1(x) & \text{sinon.} \end{cases}$$

Le cas où $x \in \mathsf{dom}(H_1)$ et $x \in \mathsf{dom}(H_2)$ ne peut en fait pas se produire, puisque Vélus rejette statiquement le masquage d'une déclaration globale par une déclaration locale. Dans le cas contraire, on pourrait écrire le programme de la figure 9, où la variable `x` dont dépend l'horloge de `y` est masquée. Pour éviter ce problème, il serait nécessaire d'introduire un niveau d'indirection dans les types d'horloges [8, Fig. 3], ce qu'on n'a pas fait dans Vélus.

```
node f(x : bool; y : int when x) returns (z : int)
var x : int;
let
    x = y + 1;
    z = x;
tel
```

FIGURE 9 – Programme avec masquage, rejeté par Vélus

Variables partagées Les flot locaux peuvent être déclarés comme partagés en utilisant le mot clé `last`. Pour traiter cette fonctionnalité, la définition d'historique est enrichie pour associer un flot différent à `x` et à `last x`, comme dans [17] ; on y accède respectivement avec $H(\mathtt{x})$ et $H(\mathtt{last\ x})$. Dans le développement Coq, l'historique est encodé par une paire d'environnements, l'un pour les variables courantes et l'autre pour les `last`. Les règles de sémantique pour `last` sont présentées en figure 10. La deuxième règle indique que l'expression `last x` produit le flot associé à `last x` dans l'historique. Cette association est contrainte au point de définition de `x` par la première règle. Le flot associé à `last x` correspond à celui associé à `x`, retardé par un délai initialisé par le flot de l'expression d'initialisation `e`.

$$\frac{\forall x,\ x \in \mathsf{dom}(H') \iff x \in \mathit{locs} \qquad \forall x\ e,\ (\mathtt{last}\ x = e) \in \mathit{locs} \implies G, H + H', bs \vdash \mathtt{last}\ x = e \qquad G, H + H', bs \vdash \mathit{blks}}{G, H, bs \vdash \mathtt{var}\ \mathit{locs}\ \mathtt{let}\ \mathit{blks}\ \mathtt{tel}}$$

$$\frac{H(\mathsf{last}\ x) \equiv vs}{G, H, bs \vdash \mathtt{last}\ x \Downarrow [vs]}$$

$$\frac{G, H + H', bs \vdash e \Downarrow [vs_0] \qquad H'(x) \equiv vs_1 \qquad H'(\mathsf{last}\ x) \equiv \mathsf{fby}\ vs_0\ vs_1}{G, H + H', bs \vdash \mathtt{last}\ x = e}$$

FIGURE 10 – Sémantique des déclarations locales et `last`

3 Analyse de dépendance

Dans la section précédente, on a présenté la sémantique dynamique de Vélus. Cependant, ce modèle seul ne suffit pas à garantir les propriétés attendues du langage. Considérons l'équation

`x = x`. Un nœud la contenant ne serait pas déterministe, puisqu'elle n'applique aucune contrainte au flot associé à `x`, qui peut prendre n'importe quelle valeur. A l'inverse, l'équation `x = x + 1` n'admet pas de sémantique, alors qu'elle est bien typée. De plus, de telles équations ne peuvent pas être compilées, puisque dans le code impératif produit, une valeur doit être calculée avant son utilisation. C'est aussi le cas pour les systèmes d'équations contenant un cycle, comme par exemple, `x = y + 1; y = x * 2`.

De nombreux travaux se sont penchés sur l'analyse statique de ces dépendances. Les analyses les plus fines [12, 1, 9] les modélisent sous forme de systèmes de types. Ces formulations permettent de donner à un nœud un type indiquant les dépendances entre entrées et sorties. Elles peuvent par exemple accepter l'équation `(x, y) = f(x)` si la première sortie de `f` ne dépend pas instantanément de son entrée. Le schéma de compilation doit alors être adapté pour produire un programme ordonnançable, soit en expansant les appels de nœud [9], soit en les décomposant en plusieurs appels ordonnançables séparément [18, 14]. Le schéma de compilation de Vélus est bien plus simple : chaque nœud est une « boite noire », et les appels sont atomiques [2]. L'exemple est donc rejeté. L'analyse de dépendance de Vélus introduite dans [5, §2.3] est basée sur une analyse de graphe. Dans cette section, on étend cette analyse pour traiter les nouvelles structures d'activation, et on décrit l'implémentation Coq de l'algorithme d'analyse de graphe.

3.1 Règles de dépendances pour le langage

A chaque flot `x` du programme est associé une étiquette α_x. Cette indirection permet de donner une étiquette différente à deux déclarations locales utilisant le même nom. La suite définit d'abord la fonction calculant l'ensemble des variables utilisées instantanément dans une expression avant de présenter formellement la relation de dépendance entre deux étiquettes.

3.1.1 Variables utilisées instantanément

Pour analyser les cycles de dépendance dans un programme, il faut d'abord déterminer l'ensemble de variables utilisées par une expression. Considérons l'équation `x = 0 fby (x + 1)`. Elle ne doit pas induire de cycle de dépendance puisque `x` ne dépend que de sa valeur précédente. On ne s'intéressera donc qu'aux variables utilisées *instantanément* dans une expression, c'est-à-dire celles n'apparaissant pas à droite d'un `fby`.

Une fonction pour ce calcul est proposée dans [2, Fig. 3], mais ne s'applique que sur un programme normalisé. Notre analyse doit être un peu plus précise pour traiter le langage général. Par exemple, l'équation `(x, y) = if c then (0, x) else (1, 0)` ne contient pas de cycle : `y` dépend de `x`, mais `x` ne dépend pas de lui même. Effectivement, cette équation est normalisée [5] en deux équations `x = if c then 0 else 1` et `y = if c then x else 0`, dans lesquelles l'absence de cycle est évidente. Pour traiter cette subtilité, on définit la fonction $\mathsf{UsedInst}_\Gamma(e)[k]$ qui collecte les étiquettes des variables utilisées instantanément pour définir le k-ième flot produit par l'expression e. Dans cet exemple, $k = 0$ désigne la définition `x = if c then 0 else 1`, et $k = 1$ désigne la définition `y = if c then x else 0`. L'environnement Γ associe le nom d'un flot à son étiquette. Il est construit à partir des déclarations encodées dans la syntaxe abstraite.

On donne en figure 11 quelques cas intéressants de la définition de cette fonction, qui est encodée comme un prédicat inductif comme dans le développement Coq. Pour une variable, on récupère l'annotation correspondante dans l'environnement. Pour une expression composée comme le `if-then-else`, on considère le flot de la condition, et les k-ième flots des deux alternatives. Pour traiter une liste de sous expressions, il faut choisir le k-ième élément de la liste aplatie des flots. Pour cela, on calcule le nombre de flots de chaque expression avec

$$\frac{\Gamma(x) = \alpha_x}{\alpha_x \in \mathsf{UsedInst}_\Gamma(x)[0]} \qquad \frac{\alpha \in \mathsf{UsedInst}_\Gamma(e)[0] \qquad k < \mathsf{numstreams}(\texttt{if}\ e\ \texttt{then}\ es_0\ \texttt{else}\ es_1)}{\alpha \in \mathsf{UsedInst}_\Gamma(\texttt{if}\ e\ \texttt{then}\ es_0\ \texttt{else}\ es_1)[k]}$$

$$\frac{\alpha \in \mathsf{UsedInst}_\Gamma(es_0)[k]}{\alpha \in \mathsf{UsedInst}_\Gamma(\texttt{if}\ e\ \texttt{then}\ es_0\ \texttt{else}\ es_1)[k]} \qquad \frac{\alpha \in \mathsf{UsedInst}_\Gamma(es_1)[k]}{\alpha \in \mathsf{UsedInst}_\Gamma(\texttt{if}\ e\ \texttt{then}\ es_0\ \texttt{else}\ es_1)[k]}$$

$$\frac{\alpha \in \mathsf{UsedInst}_\Gamma(e)[k]}{\alpha \in \mathsf{UsedInst}_\Gamma(e :: es)[k]} \qquad \frac{\alpha \in \mathsf{UsedInst}_\Gamma(es)[k]}{\alpha \in \mathsf{UsedInst}_\Gamma(e :: es)[\mathsf{numstreams}(e) + k]}$$

$$\frac{\alpha \in \mathsf{UsedInst}_\Gamma(es_0)[k]}{\alpha \in \mathsf{UsedInst}_\Gamma(es_0\ \texttt{fby}\ es_1)[k]} \qquad \frac{\alpha \in \mathsf{UsedInst}_\Gamma(es)[k'] \qquad k < \mathsf{numstreams}(f(es))}{\alpha \in \mathsf{UsedInst}_\Gamma(f(es))[k]}$$

FIGURE 11 – Quelques règles pour UsedInst

$\mathsf{numstreams}(e)$. Pour un `fby`, on ne prends pas en compte les variables apparaissant à droite, comme expliqué plus tôt. Enfin, un appel de nœud est atomique : l'expression `f(es)` dépend de toutes les variables utilisées instantanément dans `es`.

3.1.2 Variables définies et dépendances

On souhaite maintenant établir les contraintes de dépendance induites par un bloc. On note $\Gamma \vdash blk \mid \alpha_2 \xleftarrow{dep} \alpha_1$ pour indiquer que sous l'environnement Γ, dans le bloc blk, α_2 dépend de α_1. Les règles qui définissent ces contraintes dépendent aussi d'une fonction $\mathsf{Def}_\Gamma(blk)$ qui calcule les étiquettes définies par un bloc blk.

Equations : Les étiquettes définies par une équation sont celles des variables à gauche de l'équation. Une équation induit immédiatement une dépendance entre l'étiquette du k-ième flot défini, et les variables utilisées instantanément dans le k-ième flot des expressions. Par exemple, dans `(x, y) = (0, x fby z)`, `x` ne dépend d'aucune étiquette et `y` dépend seulement de `x`.

$$\frac{\Gamma(x_i) = \alpha_{x_i}}{\alpha_{x_i} \in \mathsf{Def}_\Gamma((x_1, \ldots, x_n) = es)} \qquad \frac{\Gamma(x_i) = \alpha_{x_i} \qquad \alpha \in \mathsf{UsedInst}_\Gamma(es)[i]}{\Gamma \vdash (x_1, \ldots, x_n) = es \mid \alpha_{x_i} \xleftarrow{dep} \alpha}$$

FIGURE 12 – Etiquettes définies et dépendances pour une équation

Déclarations locales : Les déclarations locales introduisent de nouvelles étiquettes $locs$. Pour traiter les sous-blocs d'une déclaration, il faut les prendre en compte, en étendant l'environnement avec $+$, défini comme : $\forall x, (\Gamma + locs)(x) = \alpha_x \leftrightarrow (\Gamma(x) = \alpha_x \vee locs(x) = \alpha_x)$. Les règles de calcul de Def et de dépendance utilisant cet opérateur sont données en figure 13. On note que les étiquettes d'une déclaration locale échappent à leur portée. C'est nécessaire pour construire un graphe de dépendance global au nœud. Un invariant statique, non présenté ici, garantit par ailleurs que toutes les étiquettes du nœud sont distinctes.

$$\frac{\alpha \in \mathsf{Def}_{(\Gamma+locs)}(blks)}{\alpha \in \mathsf{Def}_{\Gamma}(\texttt{var}\ locs\ \texttt{let}\ blks\ \texttt{tel})} \qquad \frac{(\Gamma + locs) \vdash blks \mid \alpha_y \xleftarrow{dep} \alpha_x}{\Gamma \vdash \texttt{var}\ locs\ \texttt{let}\ blks\ \texttt{tel} \mid \alpha_y \xleftarrow{dep} \alpha_x}$$

$$\frac{(\texttt{last}\ y : (\alpha_y, \alpha_{\texttt{last}y}) = e) \in locs \qquad \alpha_x \in \mathsf{UsedInst}_{(\Gamma+locs)}(e)[0]}{\Gamma \vdash \texttt{var}\ locs\ \texttt{let}\ blks\ \texttt{tel} \mid \alpha_{\texttt{last}y} \xleftarrow{dep} \alpha_x}$$

FIGURE 13 – Etiquettes définies et dépendances pour une déclaration locale

Last : Si l'utilisateur déclare un flot local `x` avec une valeur `last` initiale, on ajoute également une étiquette distincte pour `last x`. Cela permet de traiter `last x` comme un flot à part, qui ne dépend que des flots apparaissant dans son expression d'initialisation. Ainsi, les blocs présentés en figure 14 sont valides (mais ne produisent pas le même flot pour `x`). En revanche, celui présenté en figure 15 ne l'est pas, puisque `x` dépend immédiatement de `last x` et inversement.

```
var last x = 0;
let x = last x + 1; tel
```

FIGURE 14 – Déclarations `last` correctes

```
var last x = x + 1;
let x = 0; tel
```

```
var last x = x + 1;
let x = last x; tel
```

FIGURE 15 – Déclaration `last` cyclique

Switch-Case : Le cas du switch est plus complexe. On rappelle le nœud présenté en figure 4. Il ne serait pas accepté si on associait les étiquettes $\alpha_{\texttt{y}}$ et $\alpha_{\texttt{vy}}$ à `y` et `vy` respectivement : il y aurait un cycle. On peut en revanche associer des étiquettes $\alpha_{\texttt{y}_T}$ et $\alpha_{\texttt{y}_F}$ aux définitions de `y` dans les branches `true` et `false`, et de même pour `vy`. Les deux branches étant mutuellement exclusives, il ne peut pas y avoir d'interaction entre $\alpha_{\texttt{y}_T}$ et $\alpha_{\texttt{y}_F}$. L'étiquette $\alpha_{\texttt{y}}$ dépend par contre de ces étiquettes locales aux branches. De même, tous ces flots dépendent de la condition qui détermine quelle branche est activée, et donc de $\alpha_{\texttt{captdist}}$. Le graphe de dépendance pour l'exemple est présenté en figure 16. Il est bien acyclique.

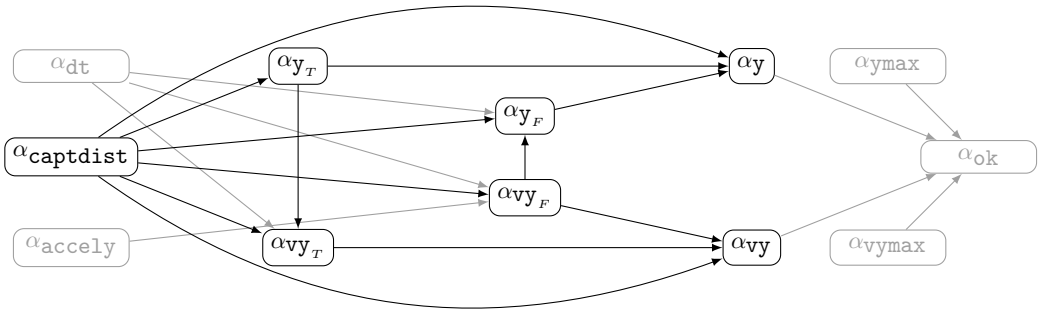

FIGURE 16 – Graphe de dépendance pour la figure 4

Pour encoder ces changements d'associations, chaque branche d'un `switch` est annotée dans la syntaxe abstraite par une fonction σ qui s'applique à l'environnement pour modifier les étiquettes. On omet cette annotation dans la présentation des règles où elle n'est pas utile. Un invariant statique non présenté ici nous garantit que les étiquettes renommées par σ sont celles

des flots définis par le `switch`. On a

$$\sigma(\Gamma)(x) = \begin{cases} \sigma(x) & \text{si } x \in \sigma \\ \Gamma(x) & \text{sinon.} \end{cases}$$

La figure 17 présente les règles de dépendances pour le `switch`. Les étiquettes définies par un `switch` sont celles des sous-blocs après renommage par σ. La deuxième règle y ajoute les étiquettes correspondantes avant le renommage. La fonction σ est aussi utilisée pour calculer les dépendances des sous-blocs. Toutes les étiquettes des flots définis par le `switch` dépendent de la condition. La dernière règle ajoute une dépendance entre chaque étiquette interne α_x et l'étiquette correspondante externe α_y, pour « connecter » les flots définis.

Cet étiquetage reflète le schéma de compilation présenté dans [8]. Une nouvelle variable est introduite pour chaque variable définie et chaque branche, et un `merge` combine les valeurs de ces nouvelles variables pour définir la variable originale.

$$\frac{\alpha \in \mathsf{Def}_{\sigma_i(\Gamma)}(blks_i)}{\alpha \in \mathsf{Def}_\Gamma(\texttt{switch}\ e\ (C_i\ \texttt{do}_{\sigma_i}\ blks_i)_i\ \texttt{end})} \qquad \frac{x \in \sigma_i \qquad \Gamma(x) = \alpha}{\alpha \in \mathsf{Def}_\Gamma(\texttt{switch}\ e\ (C_i\ \texttt{do}_{\sigma_i}\ blks_i)_i\ \texttt{end})}$$

$$\frac{\sigma_i(\Gamma) \vdash blks_i \mid \alpha_1 \xleftarrow{dep} \alpha_2}{\Gamma \vdash \texttt{switch}\ e\ (C_1\ \texttt{do}_{\sigma_i}\ blks_i)_i\ \texttt{end} \mid \alpha_1 \xleftarrow{dep} \alpha_2}$$

$$\frac{\alpha_x \in \mathsf{UsedInst}_\Gamma(e)[0] \qquad \alpha_y \in \mathsf{Def}_\Gamma(\texttt{switch}\ e\ (C_i\ \texttt{do}\ blks_i)_i\ \texttt{end})}{\Gamma \vdash \texttt{switch}\ e\ (C_i\ \texttt{do}\ blks_i)_i\ \texttt{end} \mid \alpha_y \xleftarrow{dep} \alpha_x}$$

$$\frac{x \in \sigma_i \qquad \Gamma(x) = \alpha_y \qquad \sigma_i(\Gamma)(x) = \alpha_x}{\Gamma \vdash \texttt{switch}\ e\ (C_i\ \texttt{do}_{\sigma_i}\ blks_i)_i\ \texttt{end} \mid \alpha_y \xleftarrow{dep} \alpha_x}$$

FIGURE 17 – Etiquettes définies et dépendances pour un `switch`

3.2 Analyse du graphe de dépendance

Les relations de dépendance décrites plus haut nous permettent de construire le graphe de dépendance d'un programme. On exprime ce graphe en Coq par une table associant chaque étiquette à la liste de ses prédécesseurs. Cette représentation ne garantit bien sûr pas l'absence de cycles dans le graphe. C'est le prédicat inductif AcyGraph, introduit dans [5], qui caractérise les graphes acycliques. Il est défini par les trois règles rappelées dans la figure 18. Ce prédicat est paramétré par un ensemble de sommets et un ensemble d'arcs. Un graphe vide est acyclique. Ajouter un sommet préserve l'acyclicité. Cependant, on ne peut ajouter un arc entre deux sommets α_x et α_y que s'ils sont distincts, et qu'il n'existe pas d'arc retour dans la clôture transitive des arcs existants.

On veut maintenant développer une fonction vérifiant qu'un graphe de dépendance construit selon les règles de la section 3.1 satisfait le prédicat d'acyclicité. On s'appuie sur un algorithme bien connu de recherche en profondeur pour traverser les prédécesseurs de chaque sommet dans le graphe. Pour justifier de sa terminaison et de sa correction, on utilise les types dépendants de Coq pour encoder les invariants de l'algorithme. On utilise l'extension `Program` [19] qui permet

$$\frac{}{\mathsf{AcyGraph}\ \emptyset\ \emptyset} \qquad \frac{\mathsf{AcyGraph}\ V\ A}{\mathsf{AcyGraph}\ (V \cup \{\alpha\})\ A} \qquad \frac{\mathsf{AcyGraph}\ V\ A \qquad \alpha_x, \alpha_y \in V \qquad \alpha_x \neq \alpha_y \qquad \alpha_y \nrightarrow_A^+ \alpha_x}{\mathsf{AcyGraph}\ V\ (A \cup \{\alpha_x \to \alpha_y\})}$$

FIGURE 18 – Graphe acyclique par construction

de définir la partie calculatoire de l'algorithme en Gallina, le language de programmation à la ML de Coq, et de traiter séparément les obligations de preuve avec le langage de tactiques Ltac.

On s'intéresse d'abord à la partie calculatoire, présentée figure 19, en ignorant les types dépendants utilisés pour les entrées et sorties. L'appel `dfs' s x v` recherche récursivement les prédécesseurs de `x` dans le `graph`. Elle renvoie l'ensemble des sommets visités à l'issue de son exécution. L'ensemble `v` représente les sommets déjà visités. L'ensemble `s` contient les sommets sur le chemin parcouru depuis le sommet initial. Ces ensembles sont représentés par le type `PS.t`. On utilise la monade `option` pour représenter la réussite ou l'échec de l'algorithme. La fonction échoue (ligne 13) si elle rencontre un sommet $\texttt{x} \in \texttt{s}$, ce qui signifie qu'il y a un cycle dans le graphe. Sinon, si $\texttt{x} \in \texttt{v}$, c'est qu'on connaît déjà tous ses prédécesseurs, donc on peut renvoyer `v` (ligne 16). Si `x` reste à traiter, on récupère la liste de ses prédécesseurs (ligne 18), et on appelle récursivement `dfs'` sur chacun, après avoir ajouté `x` à `s` (ligne 21). Pour itérer sur la liste des prédécesseurs, on utilise `ofold_left : (B → A → option B) → list A → option B → option B`, qui se comporte comme `fold_left`, mais s'interrompt si la fonction retourne une erreur (ligne 22).

Aux lignes 12 et 15, on utilise la construction `match/with`, plutôt qu'un `if/then/else` pour traiter la condition booléenne. C'est une contrainte liée à l'utilisation de `Program` qui, si on utilise `if/then/else`, ne conserve pas assez d'information dans les obligations de preuves générées.

Terminaison : La fonction `dfs'` ne respecte pas le critère de récursion gardée de Coq où chaque appel récursif doit être effectué sur un sous terme de l'argument initial. On utilise `Program Fixpoint` pour définir une fonction récursive en utilisant une « mesure », c'est-à-dire une fonction des arguments vers un entier naturel. Si cette mesure décroît strictement à chaque appel récursif, la fonction termine, puisque $(\mathbb{N}, \leq)$ est bien fondé. Pour cet algorithme, on sait que l'ensemble `s` grandit à chaque appel récursif, et qu'il est inclus dans l'ensemble des sommets du graphe. La mesure `num_remaining`, définie ligne 3 décroît donc strictement.

Correction : L'algorithme est correct si sa réussite indique qu'un témoin d'AcyGraph peut être associé au graphe analysé. Il est difficile de raisonner à posteriori sur un programme écrit avec `Program Fixpoint`, car le programme Coq généré contient des termes complexes justifiant de sa terminaison. À la place, on intègre les invariants de l'algorithme dans le type de la fonction.

L'invariant `visited s v` garantit d'abord que l'ensemble `s` des sommets dans le parcours et l'ensemble `v` des sommets visités sont bien disjoints (ligne 6). Par ailleurs, il indique qu'il existe un ensemble d'arcs `a`, tel que (1) le graphe formé par les sommets `v` et les arcs `a` est acyclique, et (2) pour chaque sommet `x` de `v`, tout arc allant vers `x` dans le `graph` analysé est présent dans `a`, ce qui est caractérisé par `has_arc`. Intuitivement, cela indique que le sous-graphe de `graph` réduit aux sommets de `v` est effectivement acyclique. On note que l'ensemble d'arcs `a` n'a pas besoin d'être calculé explicitement. Un tel existentiel disparaîtra dans le code OCaml extrait et ne coûtera donc pas de temps ni de mémoire à l'exécution. Par ailleurs, le type de `dfs'` garantit que le sommet `x` recherché est bien dans l'ensemble `v'` renvoyé, qui est un sur-ensemble de celui passé en entrée à la fonction.

Itérer `dfs` sur l'ensemble des sommets du graphe permet donc de construire un témoin AcyGraph pour l'ensemble des sommets du graphe. A chaque nœud du programme source analysé,

```
Variable graph : Env.t (list label).

Definition num_remaining (s : PS.t) : nat := Env.cardinal graph − PS.cardinal s.

Definition visited (s : PS.t) (v : PS.t) : Prop :=
  (∀ x, PS.In x s → ¬PS.In x v)
  ∧ ∃a, AcyGraph v a ∧ (∀ x, PS.In x v → ∃zs, Env.find x graph = Some zs ∧ (∀ y, In y zs → has_arc a y x)).

Program Fixpoint dfs′ (s : { p | ∀x, PS.In x p → Env.In x graph }) (x : ident)
                      (v : { v | visited s v }) {measure (num_remaining s)}
                      : option { v′ | visited s v′ & PS.In x v′ ∧ PS.Subset v v′ } :=
  match PS.mem x s with
  | true ⇒ None
  | false ⇒
    match PS.mem x v with
    | true ⇒ Some (exist2 _ _ v _ _)
    | false ⇒
      match Env.find x graph with
      | None ⇒ None
      | Some zs ⇒
        let s′ := exist _ (PS.add x s) _ in
        match ofold_left (fun v w ⇒ obind (dfs′ s′ w v) (fun v′ ⇒ Some (exist _ v′ _))) zs (Some v) with
        | None ⇒ None
        | Some v′ ⇒ Some (exist2 _ _ (PS.add x v′) _ _)
        end
      end
    end
  end.

Definition dfs (x : ident) (v : { v |visited PS.empty v })
               : option { v′ | visited PS.empty v′ & (PS.In x v′ ∧ PS.Subset v v′) } :=
  dfs′ (exist _ PS.empty _)
```

FIGURE 19 – Algorithme de recherche en profondeur certifiant

on peut donc associer un graphe dans lequel chaque dépendance entre étiquettes du nœud est reflétée par un arc. Si le graphe est acyclique, on dit que le nœud est causal, voir définition 2.

Définition 2 (Nœud « causal »)**.**

$$\mathsf{node\_causal}\ n := \exists V\, A, \mathit{AcyGraph}\, V\, A \wedge (\forall \alpha_x\, \alpha_y,\ \vdash n \mid \alpha_y \xleftarrow{dep} \alpha_x \implies \alpha_x \rightarrow_A \alpha_y)$$

4 Preuves par induction causale

Des travaux antérieurs [5, §2.3] présentent une approche permettant de construire un schéma d'induction pour les nœuds causaux. Dans la suite de cette section, on rappelle ce schéma avec plus de détails. On propose également un autre schéma d'induction basé sur UsedInst. On applique ensuite ces schémas pour prouver le déterminisme du modèle sémantique de Vélus.

4.1 Principe d'induction sur les étiquettes

Etant donné un graphe acyclique caractérisé par AcyGraph VA, on peut extraire un ordre topologique des étiquettes, sur lequel on peut aisément raisonner par induction. La figure 20 présente les règles de construction de cet ordre. La liste vide représente un ordre valide. Pour

ajouter un sommet à la liste, il faut que (1) ce soit bien un sommet du graphe, (2) qu'il n'apparaisse pas dans la liste, et (3) que tous les prédécesseurs, transitivement, de ce sommet soient déjà dans la liste. Pour ce dernier critère, on a prouvé que ne prendre que les arcs immédiats $(\alpha_y \to_A \alpha_x)$ donne une formulation équivalente. Cela dit, on préfère utiliser la formulation de la figure 20, qui est plus simple à manipuler dans nos preuves car elle correspond mieux à celle d'AcyGraph. En particulier, on établit l'existence d'un ordre topologique sur l'ensemble des sommets de tout graphe acyclique, comme énoncé dans la figure 20.

$$\frac{}{\mathsf{TopoOrder}\ (\mathsf{AcyGraph}\ V\ A)\ []} \qquad \frac{\mathsf{TopoOrder}\ (\mathsf{AcyGraph}\ V\ A)\ lord \qquad \alpha_x \in V \qquad \neg \mathsf{In}\ \alpha_x\ lord \qquad (\forall \alpha_y,\ \alpha_y \to_A^+ \alpha_x \implies \mathsf{In}\ \alpha_y\ lord)}{\mathsf{TopoOrder}\ (\mathsf{AcyGraph}\ V\ A)\ (\alpha_x :: lord)}$$

$$\mathsf{AcyGraph}\ V\ A \implies \exists\ lord,\ (\forall \alpha_x, \alpha_x \in V \iff \mathsf{In}\ \alpha_x\ lord) \wedge \mathsf{TopoOrder}\ (\mathsf{AcyGraph}\ V\ A)\ lord$$

FIGURE 20 – Définition et existence d'un ordre topologique

Le principe d'induction est présenté en lemme 1. On pose P_var : étiquette → Prop le prédicat à prouver pour chaque étiquette d'un nœud. La fonction locals collecte l'ensemble des étiquettes apparaissant dans les déclarations locales du bloc, y compris celles associées aux `last`. Le nœud doit être causal. Pour toute étiquette α_x, si toutes les étiquettes dont α_x dépend respectent P_var, alors α_x doit respecter P_var. Si ces deux conditions sont respectées, alors on sait que toutes les étiquettes apparaissant dans les entrées, sorties et déclarations locales du nœud respectent P_var.

Lemme 1 (Induction sur les étiquettes d'un nœud causal)**.**

$$\begin{aligned}
&\textbf{\textit{si}} \quad \mathsf{node_causal}\ (\texttt{node}\ (ins)\ \texttt{returns}\ (outs)\ blk)\\
&\textbf{\textit{et}} \quad (\forall \alpha_x,\ \mathsf{In}\ \alpha_x\ (ins + outs + \mathit{locals}\ blk) \implies\\
&\qquad\qquad (\forall \alpha_y, (ins + outs) \vdash blk \mid \alpha_x \xleftarrow{dep} \alpha_y \implies \mathsf{P_var}\ \alpha_y) \implies\\
&\qquad\qquad \mathsf{P_var}\ \alpha_x)\\
&\textbf{\textit{alors}} \quad (\forall \alpha_x, \mathsf{In}\ \alpha_x\ (ins + outs + \mathit{locals}\ blk) \implies \mathsf{P_var}\ \alpha_x)
\end{aligned}$$

La preuve de ce lemme utilise node_causal pour faire apparaître le graphe acyclique g associé au nœud. On utilise le mécanisme d'induction habituel de Coq sur le témoin de TopoOrder associé à g pour prouver que P_var est vraie, de proche en proche, pour chaque étiquette de l'ordre topologique. Pour une dépendance $\alpha_x \xleftarrow{dep} \alpha_y$ dans le nœud, il y a un arc de α_y vers α_x dans le graphe, donc α_y apparaît avant α_x dans l'ordre topologique, et donc, par induction, P_var α_y est vrai.

4.2 Principe d'induction sur les expressions

L'élément syntaxique central induisant une dépendance est l'équation. Considérons l'équation `(x, y) = es`. Le flot associé à `x` est le premier flot associé à `es`. Pour établir une propriété P_var sur ce flot, il faut d'abord raisonner par induction sur les expressions `es`. C'est l'objectif du deuxième schéma d'induction que l'on présente maintenant. On pose P_exp : exp → nat → Prop,

un prédicat sur le k-ième flot produit par une expression, et P_exps, un prédicat équivalent sur une liste d'expressions. Si toutes les variables utilisées instantanément du k-ième flot de l'expression e satisfont P_var, alors P_exp e k doit être satisfaite. Le schéma d'induction correspondant est présenté en lemme 2. Il suit la définition de UsedInst de la section 3.1.1.

Lemme 2 (Induction sur UsedInst)**.**

$$
\begin{array}{rl}
\textbf{\textit{si}} & \mathsf{P\_var}\ \alpha_x \implies \mathsf{P\_exp}\ x\ 0 \\
\textbf{\textit{et}} & \forall k, \mathsf{P\_exp}\ e\ 0 \wedge \mathsf{P\_exps}\ es_0\ k \wedge \mathsf{P\_exps}\ es_1\ k \implies \mathsf{P\_exp}\ (\texttt{if}\ e\ \texttt{then}\ es_0\ \texttt{else}\ es_1)\ k \\
\textbf{\textit{et}} & \forall k, \mathsf{P\_exps}\ es_0\ k \implies \mathsf{P\_exp}\ (es_0\ \texttt{fby}\ es_1)\ k \\
\textbf{\textit{et}} & \ldots \\
\textbf{\textit{alors}} & \forall e\ k, (\forall x, x \in \mathsf{UsedInst}_\Gamma(e)[k] \implies \mathsf{P\_var}\ x) \implies \mathsf{P\_exp}\ e\ k
\end{array}
$$

Pour chaque forme d'expression, l'utilisateur du schéma doit prouver que, si un ensemble d'hypothèses d'induction est vérifié, alors P_exp est vraie pour l'expression. Par exemple, dans le cas d'une variable, l'utilisateur doit prouver P_exp `x` 0, sous l'hypothèse que l'étiquette associée à `x` respecte P_var. Pour le `if-then-else`, on a une hypothèse pour la condition, et une pour chacun des deux opérandes. Le cas du `fby` est plus intéressant : conformément à la définition de UsedInst, on n'a qu'une hypothèse d'induction pour l'opérande de gauche. Quelque soit la propriété P_exp considérée, elle doit donc pouvoir être prouvée indépendemment de l'opérande de droite du `fby`. C'est le cas dans la preuve de déterminisme.

4.3 Application : déterminisme de la sémantique

On s'intéresse maintenant à la propriété de déterminisme du modèle sémantique, qui peut être exprimée comme « l'exécution d'un nœud pour une entrée donnée produit toujours la même sortie ». Cette propriété est formalisée par le théorème 1.

Théorème 1 (Déterminisme de la sémantique)**.**

$$
\begin{array}{rl}
\textbf{\textit{si}} & \mathsf{node\_causal}\ G(f) \\
\textbf{\textit{et}} & G \vdash f(xs) \Downarrow ys_1 \quad \textbf{\textit{et}} \quad G \vdash f(xs) \Downarrow ys_2 \\
\textbf{\textit{alors}} & ys_1 \equiv ys_2
\end{array}
$$

Ce théorème fait l'hypothèse de deux exécutions possibles du nœud. Cela signifie qu'il existe deux historiques distincts H_1 et H_2 qui satisfont les contraintes de ce nœud. Le but est de prouver qu'en réalité, ces historiques correspondent, pour chaque variable du nœud. Ce n'est vrai que pour les nœuds causaux. En effet, dans un nœud contenant la déclaration cyclique `x = x`, on peut associer n'importe quel flot à `x`.

Pour illustrer la structure générale de la preuve, on s'appuie sur le programme `after_n` présenté en figure 21. Ce programme renvoie `false` pendant `n` instants, puis `true`. Après cela, le compteur interne `c` n'est plus mis à jour pour éviter un dépassement d'entier négatif.

Dans ce programme, les valeurs de `c` et `b` à l'instant $n+1$ dépendent de leurs valeurs à l'instant n. Pour prouver que les flots associés à `c` et `b` dans H_1 et H_2 correspondent, on doit donc raisonner par induction sur n. L'induction doit être globale au nœud puisque la définition d'une variable peut dépendre de la valeur précédente de n'importe quelle autre : dans l'exemple, `c` dépend de la valeur précédente de `b`. On utilise la relation d'équivalence jusqu'à n définie ci-dessous. Si on peut prouver $xs_1 \equiv_n xs_2$ pour un n arbitraire, alors on peut prouver $xs_1 \equiv xs_2$.

```
node after_n(n : int) returns (b : bool)
var c : int;
let
  c = n fby (if b then c else (c − 1));
  b = c <= 0;
tel
```

FIGURE 21 – Programme `after_n`

Définition 3 (Equivalence jusqu'à n)**.**

$$xs \equiv_0 ys \qquad\qquad x \cdot xs \equiv_{n+1} y \cdot ys \quad \textbf{\textit{ssi}} \quad x = y \quad \textbf{\textit{et}} \quad xs \equiv_n ys$$

Le cas $n = 0$ de l'induction est trivial. Pour le cas inductif, on fait l'hypothèse que tous les flots de H_1 et H_2 sont égaux jusqu'à n, et on veut prouver qu'ils le sont toujours jusqu'à $n + 1$. La propriété qu'on veut prouver pour chaque étiquette $\mathsf{P\_var}$ dépend donc du rang n, et des deux historiques H_1 et H_2. Elle dépend aussi d'un environnement Γ pour faire le lien entre les noms des flots et leurs étiquettes. Cet environnement est déduit des déclarations des entrées, sorties, et variables locales du nœud. On définit donc

$$\mathsf{P\_var}\ \alpha_x := \forall x,\ \Gamma(x) = \alpha_x \implies H_1(x) \equiv_{n+1} H_2(x)$$

La preuve procède par induction sur l'ordre topologique des variables, comme décrit dans le lemme 1. Dans l'exemple, $\mathsf{P\_var}$ est immédiatement vraie pour `n` qui est une entrée. On la prouve ensuite pour `c` qui ne dépend instantanément que de `n`, puis pour `b` qui dépend de `c`.

Induction sur les expressions Pour chaque étiquette, il y a dans le nœud une équation `(`x_1`, ..., `x_m`) = es` définissant le flot correspondant. Par la règle sémantique de l'équation, on sait que le flot associé à l'étiquette de x_k est le k-ième flot produit par `es`. On montre donc le lemme ci-dessous qui indique que les k-ième flots produits correspondent jusqu'à $n + 1$. Pour le prouver, on utilise le principe d'induction du lemme 2.

Lemme 3 (Equivalence jusqu'à $n + 1$ pour les expressions)**.**

$$\begin{aligned}
&\textbf{\textit{si}} \quad \forall x,\ H_1(x) \equiv_n H_2(x)\\
&\textbf{\textit{et}} \quad \forall x\ \alpha_x,\ \alpha_x \in \mathsf{UsedInst}_\Gamma(e)[k] \implies \Gamma(x) = \alpha_x \implies H_1(x) \equiv_{n+1} H_2(x)\\
&\textbf{\textit{et}} \quad bs_1 \equiv_{n+1} bs_2\\
&\textbf{\textit{et}} \quad G, H_1, bs_1 \vdash e \Downarrow \mathbf{vss_1} \quad \textbf{\textit{et}} \quad G, H_2, bs_2 \vdash e \Downarrow \mathbf{vss_2}\\
\textbf{\textit{alors}} \quad &\mathbf{vss_1}[k] \equiv_{n+1} \mathbf{vss_2}[k]
\end{aligned}$$

Pour prouver ce résultat, il faut que les flots de H_1 et H_2 soient égaux jusqu'à n, ce qu'on sait par hypothèse d'induction. Il faut aussi que les flots associés aux variables utilisées instantanément pour le k-ième flot soient égaux jusqu'à $n + 1$. Cette différence vient du traitement de `fby`. Considérons l'équation `c = n fby (if b then c else (c - 1))` de l'exemple. Pour prouver que les flots associés à `c` sont égaux jusqu'à $n + 1$, il faut prouver que les flots associés à `n` le sont, puisque `c` en dépend instantanément. En revanche, on a seulement besoin de savoir que les flots associés aux variables à droites du `fby` sont égaux jusqu'à n, puisque leurs valeurs à l'instant $n + 1$ ne sont pas utilisées instantanément. Cette intuition est formalisée dans lemme 4, qui est prouvé par co-induction sur les définitions de fby et fby1.

Lemme 4 (Equivalence jusqu'à $n + 1$ pour fby)**.**

$$\textbf{\textit{si}} \quad xs_1 \equiv_{n+1} xs_2 \quad \textbf{\textit{et}} \quad ys_1 \equiv_n ys_2 \quad \textbf{\textit{alors}} \quad \mathsf{fby}\ xs_1\ ys_1 \equiv_{n+1} \mathsf{fby}\ xs_2\ ys_2$$

Induction sur la sémantique des déclarations locales Pour trouver l'équation définissant un flot, on procède par induction sur l'arbre de syntaxe des blocs. On rencontre une difficulté pour les flots déclarés localement, comme c dans l'exemple. En effet, la règle pour les déclarations locales de la figure 10 indique seulement qu'un historique interne existe et associe un flot à chaque déclaration interne, mais n'expose pas ces flots. Ainsi, la propriété P_var globale au nœud n'est pas suffisante pour traiter ces étiquettes locales.

Pour surmonter cette complication, on définit une version instrumentée de la sémantique des blocs. Elle met en parallèle les deux exécutions du bloc, en ajoutant localement des informations sur la correspondance des flots associés aux déclarations locales. Ces informations sont préservées pendant les inductions imbriquées sur n et sur l'ordre topologique.

$$\frac{\begin{array}{c}\forall x,\ x \in \mathsf{dom}(H_1') \iff x \in locs \qquad \forall x,\ x \in \mathsf{dom}(H_2') \iff x \in locs \\ \forall x\ e,\ (\texttt{last}\ x = e) \in locs \implies G, H_1 + H_1', bs \vdash \texttt{last}\ x = e \\ \forall x\ e,\ (\texttt{last}\ x = e) \in locs \implies G, H_2 + H_2', bs \vdash \texttt{last}\ x = e \\ G, H_1 + H_1', H_2 + H_2', bs_1, bs_2 \vdash_{n,lord} blks \qquad \forall x,\ x \in locs \implies H_1'(x) \equiv_n H_2'(x) \\ \forall x,\ locs(x) = \alpha \implies \alpha \in lord \implies H_1'(x) \equiv_{n+1} H_2'(x)\end{array}}{G, H_1, H_2, bs_1, bs_2 \vdash_{n,lord} \texttt{var}\ locs\ \texttt{let}\ blks\ \texttt{tel}}$$

FIGURE 22 – Sémantique instrumentée, cas de la déclaration locale (cf. figure 10)

On note le jugement instrumenté $G, H_1, H_2, bs_1, bs_2, \vdash_{n,lord} blk$. La règle pour les déclarations locales est donnée en figure 22. Elle suit la règle de sémantique donnée plus tôt pour les déclarations locales. Les deux dernières prémisses établissent la correspondance entre les deux exécutions. La prémisse $\forall x, x \in locs \implies H_1'(x) \equiv_n H_2'(x)$ impose que tous les flots locaux soient égaux jusqu'à n. Cela correspond à notre invariant d'induction sur n. De plus, la prémisse $\forall x, locs(x) = \alpha \implies \alpha \in lord \implies H_1'(x) \equiv_{n+1} H_2'(x)$ impose que les flots associés aux étiquettes de *lord* soient égaux jusqu'à $n+1$. Cette liste *lord* est un préfixe de l'ordre topologique manipulé par le principe d'induction du lemme 1. Cela correspond donc à notre invariant d'induction sur l'ordre topologique.

La structure de la preuve du théorème 1, en prenant en compte le modèle sémantique instrumenté, est la suivante : On procède par induction sur n. Par hypothèse d'induction, on sait que tous les flots du nœud sont égaux au rang n. On sait donc en particulier que tous les flots locaux le sont, mais on ne sait pas encore qu'ils sont égaux jusqu'à $n+1$. On pose $lord = []$, la liste (vide) des étiquettes pour lesquelles on sait que les flots sont égaux jusqu'à $n+1$, et on a donc $G, H_1, H_2, bs_1, bs_2, \vdash_{n,[]} blk$. On utilise alors le principe d'induction du lemme 1 pour prouver que les flots associés à chaque étiquette du nœud sont égaux jusqu'à $n+1$. Pour chaque étiquette α, on trouve l'équation définissant le flot correspondant et on utilise le lemme 3 comme décrit plus haut. Si l'étiquette est celle d'une déclaration locale, on ajoute α à la liste *lord*, et on obtient $G, H_1, H_2, bs_1, bs_2, \vdash_{n,\alpha::lord} blk$. À la fin de l'induction, on a $G, H_1, H_2, bs_1, bs_2, \vdash_{n,(\mathsf{locals}\ blk)} blk$. Autrement dit, on sait que tous les flots locaux sont égaux jusqu'à $n+1$, parce qu'ils sont tous dans la liste. C'est équivalent à incrémenter n, et vider la liste, comme énoncé formellement dans le lemme 5, ce qui termine la preuve.

Lemme 5 (Sémantique instrumentée, passage de n à $n+1$)**.**

$$\textbf{\textit{si}} \quad G, H_1, H_2, bs_1, bs_2, \vdash_{n,(\mathsf{locals}\ blk)} blk \quad \textbf{\textit{alors}} \quad G, H_1, H_2, bs_1, bs_2 \vdash_{n+1,[]} blk$$

5 Conclusion

Dans cet article, nous avons décrit l'ajout de deux nouvelles structures d'activation dans Vélus : le `switch` et les déclarations locales. Nous avons aussi ajouté à Vélus des variables partagées au moyen de l'opérateur `last`. Nous avons modélisé la sémantique à flot de données de ces structures. Les nouvelles règles sémantiques s'intègrent modulairement au modèle de Vélus, c'est-à-dire sans modifier les règles existantes. Nous avons aussi adapté l'analyse de dépendance de Vélus pour traiter ces nouvelles constructions. Nous avons présenté en détail un algorithme d'analyse de graphe générant un témoin de l'absence de cycle dans le graphe de dépendance d'un nœud. Enfin, nous avons défini un principe d'induction tirant parti de cette analyse.

Travaux connexes Les auteurs de [7] proposent une sémantique par réaction pour un language synchrone à flots de données avec structures d'activation. Dans [8], la sémantique des structures d'activation est donnée « par traduction » dans le langage noyau. De la même manière, notre règle pour le `switch` est définie en fonction de l'opérateur d'échantillonage `when`, qui est aussi utilisé lors de sa compilation. Ces articles formalisent également la sémantique et la compilation de machines à états hiérarchiques dans un langage synchrone à flots de données. Nous travaillons actuellement à l'ajout de cette construction au compilateur Vélus. Du point de vue de l'analyse de dépendance, elle ne pose à priori pas plus de difficultés que les `switch`. Sa sémantique est en revanche plus difficile à exprimer dans notre modèle relationnel à flots de données.

Notre implémentation de l'algorithme de recherche en profondeur a suivi, sans le savoir, le travail de F. Pottier. Il note [15, §4] que son algorithme, comme le notre, utilise sur la pile d'appel un espace proportionnel à la profondeur maximale de recherche, et qu'une implémentation en style récursif terminal avec pile explicite éviterait ce problème. Dans notre cas les graphes manipulés sont issu de programmes écrits par l'utilisateur et il y a donc peu de chance d'arriver à un débordement de la pile. Comme dans ce travail on utilise des types dépendants pour justifier la terminaison de l'algorithme, mais contrairement à lui on utilise la bibliothèque `Program` de M. Sozeau [19] pour éviter de programmer avec des tactiques.

Discussion Dans notre développement Coq, les règles sémantiques de Vélus présentées dans cet article sont encodées par des prédicats inductifs. Manipuler cet encodage dans un script de preuve Coq est commode, puisqu'on peut raisonner par introduction, inversion et induction. Entre autres, la preuve de correction des algorithmes de compilation pour ces structures, qui n'a pas été présentée dans cet article, est relativement directe.

Le principe d'induction sur un nœud causal présenté dans cet article est une idée centrale pour le langage. Si, dans cet article, nous ne l'avons appliqué qu'à la preuve du déterminisme du modèle sémantique, il nous a été utile pour établir d'autres propriétés. Prouver la correction du système d'horloge, c'est-à-dire l'adéquation du rythme des flots produits aux annotations statiques d'horloges, nécessite par exemple l'utilisation de ce principe. Cette propriété est indispensable pour démontrer la correction du compilateur.

Remerciements Nous remercions L. Brun pour ses contributions à Vélus, et en particulier son travail sur l'intégration des types énumérés, sans lesquels l'ajout de `switch` n'aurait pas été possible. Nous remercions également nos rapporteurs pour leurs remarques qui ont permis d'améliorer le contenu de cet article. Ce travail a été soutenu par le projet ANR JCJC FidelR 19-CE25-0014-01 « Fidelity in Reactive Systems Design and Compilation ».

Bibliographie

[1] Albert Benveniste, Timothy Bourke, Benoit Caillaud, Bruno Pagano et Marc Pouzet : A type-based analysis of causality loops in hybrid modelers. *In Proceedings of the 17th International Conference on Hybrid Systems : Computation and Control (HSCC 2014)*, pages 71–82, Berlin, Allemagne, avril 2014. ACM Press.

[2] Dariusz Biernacki, Jean-Louis Colaço, Gregoire Hamon et 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, États-Unis, juin 2008. ACM Press.

[3] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet et 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, Barcelone, Espagne, juin 2017. ACM Press.

[4] Timothy Bourke, Lélio Brun et 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, janvier 2020.

[5] Timothy Bourke, Paul Jeanmaire, Basile Pesin et Marc Pouzet : Verified Lustre normalization with node subsampling. *ACM Transactions on Embedded Computing Systems*, 20(5s):Article 98, octobre 2021.

[6] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs et John A. Plaice : 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, Allemagne, janvier 1987. ACM Press.

[7] Jean-Louis Colaço, Grégoire Hamon et Marc Pouzet : Mixing signals and modes in synchronous data-flow systems. *In Proceedings of the 6th ACM International Conference on Embedded Software (EMSOFT 2006)*, pages 73–82, Séoul, Corée du Sud, octobre 2006. ACM Press.

[8] Jean-Louis Colaço, Bruno Pagano et Marc Pouzet : A conservative extension of synchronous data-flow with state machines. *In Proceedings of the 5th ACM International Conference on Embedded Software (EMSOFT 2005)*, pages 173–182, Jersey City, États-Unis, septembre 2005. ACM Press.

[9] Jean-Louis Colaço, Bruno Pagano et 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, septembre 2017. IEEE Computer Society.

[10] Jean-Louis Colaço et Marc Pouzet : Clocks as first class abstract types. *In Proceedings of the 3rd International Conference on Embedded Software (EMSOFT 2003)*, volume 2855 de *Lecture Notes in Elecrical Engineering*, pages 134–155, Philadelphie, PA, États-Unis, octobre 2003. Springer.

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

[12] Pascal Cuoq et Marc Pouzet : Modular causality in a synchronous stream language. *In 10th European Symposium on Programming (ESOP 2001), part of European Joint Conferences on Theory and Practice of Software (ETAPS 2001)*, volume 2028 de *Lecture Notes in Elecrical Engineering*, pages 237–251, Gênes, Italie, avril 2001. Springer.

[13] Xavier Leroy : Formal verification of a realistic compiler. *Comm. ACM*, 52(7):107–115, 2009.

[14] Roberto Lublinerman, Christian Szegedy et Stavros Tripakis : Modular code generation from synchronous block diagrams : Modularity vs. code size. *In Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2009)*, pages 78–89, Savannah, GA, États-Unis, janvier 2009. ACM Press.

[15] François Pottier : Depth-first search and strong connectivity in Coq. *In Journées Francophones des Langages Applicatifs (JFLA)*, janvier 2015.

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

[17] Marc Pouzet : ZRun. https://github.com/marcpouzet/zrun/blob/master/src/coiteration.ml, 2021.

[18] Marc Pouzet et Pascal Raymond : 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, octobre 2009. ACM Press.

[19] Matthieu Sozeau : Subset Coercions in Coq. *Lecture Notes in Computer Science*, (4502):237–252, 2007.