Rapport de stage : Vérification de génération de code pour le modèle polyédrique

Nathanaël Courant, sous la supervision de Xavier Leroy

Mars-juillet 2018

Contexte général

La compilation formellement vérifiée permet de garantir qu'un programme compilé se conformera bien à la sémantique du code source correspondant, et donc que le compilateur n'aura pas introduit d'erreurs dans le code compilé. Cependant, cela se fait souvent au prix d'une dégradation de la performance du code produit, car les optimisations les plus agressives utilisées en pratique étant également les plus complexes à justifier correctes. Un des défis majeurs pour augmenter la performance du code produit par un compilateur formellement vérifié est de justifier la correction de telles optimisations. La compilation polyédrique est une technique qui permet d'effectuer et de justifier un grand nombre d'optimisations de boucles.

Problème étudié

Mon stage s'est concentré sur la génération de code séquentiel à partir d'une représentation polyédrique et sa formalisation en Coq. En effet, un générateur de code polyédrique formellement vérifié pourrait permettre à des compilateurs comme CompCert [Ler09] d'effectuer des optimisations de nids de boucles qu'ils ne font pour l'instant pas, ou encore d'écrire un compilateur formellement vérifié utilisant le modèle polyédrique pour un langage dédié comme Tensor Comprehensions [Vas+18], VOBLA [Bea+14] ou encore Alpha [QR], pour produire du code numérique de haute performance.

La question de la formalisation du modèle polyédrique a, à ma connaissance, été abordée uniquement dans les travaux non publiés d'Alexandre Pilkiewicz [Pil]. De plus, ces travaux ne traitent pas de la partie génération de code du modèle polyédrique, qui était la partie sur laquelle s'est concentré mon stage.

Contribution proposée

Les contributions de mon stage sont les suivantes : la formalisation de la sémantique de programmes polyédriques (section 2) ; la définition et for-

malisation de la sémantique du langage cible Loop (section 6) ainsi que du langage intermédiaire PolyLoop (section 4) ; l'implantation d'un générateur de code formellement vérifié transformant un programme polyédrique en un programme Loop, qui est composé des étapes d'élimination de l'ordonnancement (section 3), de génération de l'arbre de syntaxe (section 4), de simplification des contraintes redondantes (section 5), et enfin de génération de code Loop (section 6) ; et enfin, la vérification à l'aide de Coq de ce générateur de code (section 7).

Arguments en faveur de sa validité

Le générateur de code mentionné ci-dessus a été complètement implanté et vérifié en Coq comme produisant du code respectant toujours la sémantique du programme polyédrique source. De plus, il a été testé comme produisant effectivement du code, qui est d'une complexité similaire à celle attendue. Ainsi, sur l'exemple utilisé dans [Bas04], on obtient un résultat sensiblement équivalent à celui obtenu dans cet article pour la partie de la génération de code que nous avons implantée. Des exemples de résultats sont donnés en annexe C.

De plus, nous avons uniquement justifié le respect de la sémantique dans le cas de programmes terminant sans erreur. Cependant, le langage Loop étant déterministe et fortement normalisant si les instructions de base le sont, on peut étendre ce résultat sans trop de difficultés au cas de programmes ne terminant pas ou produisant une erreur, même si cela n'a pas été formalisé.

Bilan et perspectives

Ce travail est ainsi la première implantation formellement vérifiée de la partie de génération de code du modèle polyédrique, et est donc un premier pas vers l'intégration d'optimisations de nids de boucles dans des compilateurs formellement vérifiés. Elle permettrait également une compilation formellement vérifiée de langages dédiés pour faire du calcul numérique efficace.

Parmi les objectifs à court terme, il faudrait intégrer les autres optimisations présentées dans [Bas04] et dans [GVC15], comme la détection de pas de boucle, l'élimination de code mort et la réduction de la taille du code pour obtenir du code plus efficace et plus compact en sortie.

Les objectifs à long terme seraient principalement de compléter la formalisation du modèle polyédrique pour obtenir un générateur complet formellement vérifié, puis une possible intégration à CompCert, soit en compilant un langage dédié, soit en ajoutant des optimisations de nids de boucles dans la chaîne de compilation CompCert. Il faudrait également pour cela considérer le problème des dépassements de capacité arithmétiques, qui a été traité dans [Cue+12].

```
for (int i = 0; i < n; i++ ) {
  for (int j = 0; j < n; j++ ) {
    for (int k = 0; k < n; k++ ) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

(a) Multiplication de matrices

```
for (int m = 0; m < n; m++ ) {
  for (int k = 1; k < m; k++ ) {
    binom[m][k] =
      binom[m-1][k-1] + binom[m-1][k];
  }
}
```

(b) Triangle de Pascal

FIGURE 1 – Problèmes classiques de calcul numérique sous forme de nids de boucles

1 Introduction

1.1 Nids de boucles et leur optimisation

De nombreux problèmes de calcul numérique peuvent être formulés comme des calculs effectués par des boucles imbriquées. Ainsi, deux problèmes classiques (multiplication de matrices, et calcul du triangle de Pascal) sont donnés comme exemples en figure 1 (par souci de simplicité, on a omis l'initialisation).

Pour obtenir un binaire compilé plus efficace, les compilateurs optimisants traditionnels peuvent faire des transformations sur ces nids de boucles, par exemple, échanger deux boucles. Ainsi, sur le problème de multiplication de matrices, exécuter les boucles dans l'ordre `i`, `k`, `j` est plus rapide que l'ordre `i`, `j`, `k` spécifié par le code, car on a une meilleure localité des accès mémoire.

1.2 Le modèle polyédrique

Le modèle polyédrique peut être vu comme une représentation intermédiaire permettant de faire des opérations complexes sur des nids de boucles : séparation de boucles, échange de boucles, déplacement d'invariants de boucles en sont des cas particuliers. Il permet d'effectuer des optimisations agressives sur le corps d'un nid de boucles en réordonnant ses itérations, ce qui peut être bénéfique comme on l'a vu précédemment. Bien qu'il soit possible d'écrire et de formaliser ces optimisations au cas par cas, le modèle polyédrique permet d'exprimer des optimisations plus complexes, adaptées à chaque programme, et donne un cadre unifié dans lequel les formaliser.

Cette représentation intermédiaire considère un nid de boucles comme le fait de devoir exécuter un certain nombre d'instructions élémentaires en tous les points entiers d'un polyèdre (possiblement différent pour chaque instruction), ainsi que des dépendances entre certains de ces points, et un ordre d'exécution ou ordonnancement (*schedule* dans la littérature anglo-

saxonne). Le polyèdre considéré est donc l'espace des itérations. Tout réordonnancement de ces points préservant les dépendances possède alors la même sémantique que le programme initial.

Par exemple, dans le cas de la multiplication de matrices, on avait le cube $[\![0, n[\![^3$ comme espace d'itérations, exécutées dans l'ordre lexicographique sur les $(\mathtt{i}, \mathtt{j}, \mathtt{k})$. De même, pour le triangle de Pascal, on avait un triangle comme espace d'itérations, exécutées dans l'ordre lexicographique sur les $(\mathtt{m}, \mathtt{k})$.

La compilation polyédrique est donc décomposée en trois étapes : tout d'abord, la transformation d'un programme usuel sous forme d'arbre de syntaxe abstraite en une représentation polyédrique (on utilise les conditions de Bernstein [Ber66] pour déterminer les dépendances) ; ensuite, un changement de l'ordre d'exécution sur cette représentation abstraite qui préserve les dépendances (il s'agit d'une phase heuristique pour laquelle de nombreuses approches sont possibles [BRS07 ; Fea92a ; Fea92b]) ; et enfin, la génération de code sous forme de d'arbre de syntaxe abstraite, qui exécute le programme dont l'ordre d'exécution a été modifié. Par exemple, on aurait pu transformer l'ordonnancement de la multiplication de matrices en l'ordre lexicographique sur les $(\mathtt{i}, \mathtt{k}, \mathtt{j})$ pour obtenir un programme plus efficace.

C'est plus particulièrement à cette dernière phase de génération de code que je me suis intéressé, et qui est présentée par la suite. Comme cette phase ne contient pas de réordonnancement, on peut complètement oublier les dépendances entre les différents points, qu'on ne mentionnera donc pas dans le reste de ce rapport.

1.3 Vue d'ensemble

Pour rendre les phases de générations de code plus simples, et donc plus simples à prouver, la génération de code a été découpée en plusieurs étapes intermédiaires : tout d'abord, on élimine les informations d'ordonnancement, en les remplaçant par un ordonnancement lexicographique. Cela permet de faire correspondre l'ordre d'exécution à celui d'un nid de boucles imbriquées. Cette transformation est présentée section 3. Ensuite, on construit l'arbre de syntaxe abstrait correspondant au programme polyédrique obtenu, mais dans lequel on garde les conditions et expressions abstraites pour correspondre à des opérations naturelles polyédriques. La cible de cette transformation est le langage PolyLoop, qui est présenté avec cette transformation, en section 4. Le coût d'avoir effectué une génération la plus simple possible à l'étape précédente est que le programme PolyLoop résultant contient un nombre élevé de contraintes redondantes : celles-ci sont éliminées par une transformation sur les programmes PolyLoop présentée section 5. Enfin, la dernière étape de la génération de code transforme ce programme PolyLoop en programme Loop en concrétisant les contraintes et expressions de PolyLoop ; cette étape est présentée section 6. La combinaison de ces différentes étapes entre elles est résumée figure 2.

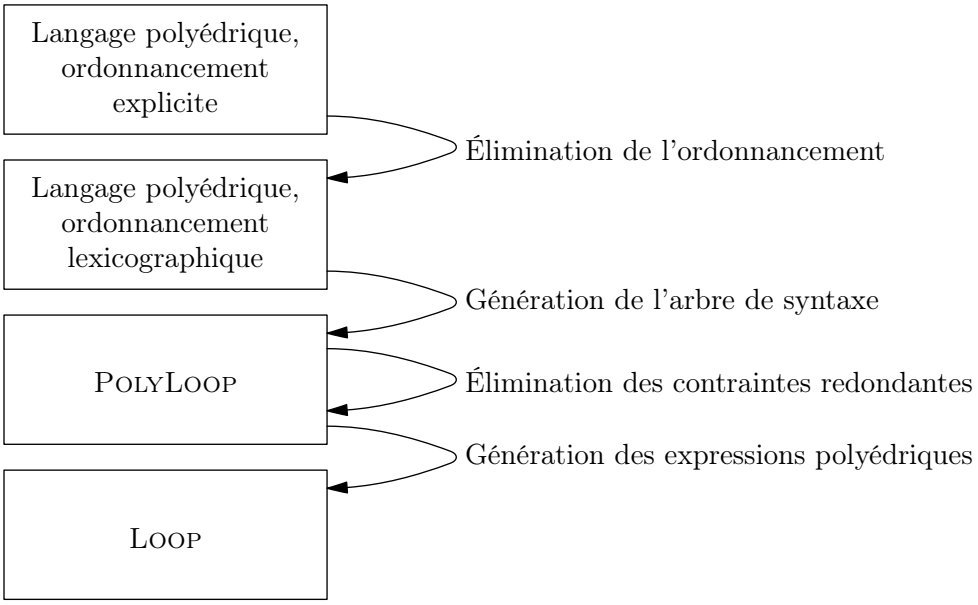


FIGURE 2 – Étapes de la génération de code

2 Le langage initial

Un programme polyédrique est un (multi-)ensemble d'instructions polyédriques, chacune étant un quadruplet $(\mathtt{I}, \mathcal{D}, \theta, \mathcal{T})$. La signification de chacune de ces composantes est la suivante :

— `I` est une instruction,

— $\mathcal{D}$ est le domaine de l'instruction polyédrique : c'est un polyèdre défini par une matrice A et un vecteur a, et donc égal à :

$$\{x \in \mathbb{Z}^n / Ax \leq a\}.$$

— θ est la fonction d'ordonnancement, qui est une fonction affine. Elle est donc définie par une matrice K et un vecteur k, et est donc :

$$\theta : x \mapsto Kx + k.$$

— Enfin, $\mathcal{T}$ est la fonction de transformation, qui est absente des définitions habituelles du modèle polyédrique ; elle sera utile par la suite. Tout comme la fonction d'ordonnancement, c'est une fonction affine, définie par une matrice T et un vecteur t.

On suppose de plus que dans un programme donné, les propriétés suivantes sont vérifiées :

— Quitte à rajouter des dimensions supplémentaires fixées à 0, la dimension de tous les domaines $\mathcal{D}$ est identique,

$$(\mathtt{I}_1, \{(n,x,y)/0 \leq x \leq y \leq n\}, (n,x,y) \mapsto (x,x), (n,x,y) \mapsto (x,y)),$$
$$(\mathtt{I}_2, \{(n,x,y)/0 \leq y \leq x \leq n\}, (n,x,y) \mapsto (x+y-1,1), (n,x,y) \mapsto (x))$$

(a) Programme polyédrique à exécuter

$\mathtt{I}_1(0,0) \rightsquigarrow (0,0)$
$\mathtt{I}_1(0,1) \rightsquigarrow (0,0)$
$\mathtt{I}_1(1,1) \rightsquigarrow (1,1)$
$\mathtt{I}_2(0) \rightsquigarrow (-1,1)$
$\mathtt{I}_2(1) \rightsquigarrow (0,1)$
$\mathtt{I}_2(1) \rightsquigarrow (1,1)$

(b) Instructions à exécuter, et *time-stamps* associés pour l'environnement ($n = 1$)

$\mathtt{I}_2(0); \mathtt{I}_1(0,0); \mathtt{I}_1(0,1); \mathtt{I}_2(1); \mathtt{I}_1(1,1); \mathtt{I}_2(1)$
$\mathtt{I}_2(0); \mathtt{I}_1(0,0); \mathtt{I}_1(0,1); \mathtt{I}_2(1); \mathtt{I}_2(1); \mathtt{I}_1(1,1)$
$\mathtt{I}_2(0); \mathtt{I}_1(0,1); \mathtt{I}_1(0,0); \mathtt{I}_2(1); \mathtt{I}_1(1,1); \mathtt{I}_2(1)$
$\mathtt{I}_2(0); \mathtt{I}_1(0,1); \mathtt{I}_1(0,0); \mathtt{I}_2(1); \mathtt{I}_2(1); \mathtt{I}_1(1,1)$

(c) Ordres possibles d'exécution, pour l'environnement ($n = 1$)

FIGURE 3 – Exemple de sémantique de programme polyédrique

— Quitte à rajouter des dimensions supplémentaires aux fonctions d'ordonnancement, la dimension de l'image de toutes les fonctions d'ordonnancement θ est identique.

Exécuter un programme polyédrique $\mathcal{P}$ dans un environnement $\mathcal{E}$ se fait de la manière suivante :

1. Pour chaque instruction polyédrique $(\mathtt{I}, \mathcal{D}, \theta, \mathcal{T}) \in \mathcal{P}$, calculer son domaine effectif :
$$\mathcal{D}_\mathcal{E} = \left\{ \begin{pmatrix} \mathcal{E} \\ y \end{pmatrix} \middle/ y \in \mathbb{Z}^n, \begin{pmatrix} \mathcal{E} \\ y \end{pmatrix} \in \mathcal{D} \right\}$$
2. Itérer sur les $(\mathtt{I}, \mathcal{D}, \theta, \mathcal{T}) \in \mathcal{P}$ et $x \in \mathcal{D}_\mathcal{E}$, dans l'ordre lexicographique des $\theta(x)$ croissants, les instructions $\mathtt{I}(\mathcal{T}(x))$. On a ici une sémantique non déterministe, même si les $\mathtt{I}(\mathcal{T}(x))$ sont toutes déterministes : en effet, si $\theta(x) = \theta'(x')$, alors $\mathtt{I}(\mathcal{T}(x))$ et $\mathtt{I}'(\mathcal{T}'(x'))$ peuvent être exécutées dans un ordre arbitraire.

Un exemple de programme polyédrique, et ses ordres d'exécution possibles, sont donnés en figure 3.

3 Élimination de l'ordonnancement

Si on considère un nid de boucles, et qu'on le transforme en programme polyédrique, on obtiendra un ordonnancement lexicographique sur les variables utilisées pour itérer. Dès lors, une première étape naturelle de la

génération de code d'un programme polyédrique vers un nid de boucles est d'éliminer l'ordonnancement explicite, et de le remplacer par un ordonnancement implicite, qui est lexicographique sur les variables.

Du côté de la formalisation, bien qu'un ordonnancement lexicographique ne soit qu'un cas particulier d'ordonnancement explicite, il est plus pratique d'avoir deux sémantiques distinctes pour les deux : en effet, cela permet d'éviter une hypothèse supplémentaire pour tous les théorèmes sur la sémantique avec ordonnancement implicite.

Il convient de noter qu'une telle transformation ne peut pas préserver *toutes* les sémantiques d'un programme polyédrique. En effet, un programme polyédrique générique peut avoir plusieurs exécutions d'une instruction en des points différents correspondant toutes au même *timestamp*, et laisse l'ordre d'évaluation de ces instructions non décidé (il y a donc présence de non-déterminisme). D'un autre côté, cela n'est pas possible dans le cas d'un ordonnancement lexicographique, puisque deux exécutions différentes d'une même instruction sont nécessairement ordonnées : il y a donc une déterminisation partielle faite à la traduction. Le théorème qu'on obtiendra sera donc que toute exécution valide du programme transformé est une exécution valide du programme initial.

De plus, même si la sémantique des instructions de base est déterministe, la sémantique avec ordonnancement implicite lexicographique ne l'est toujours pas : il n'y a aucune contrainte sur l'ordre d'exécution de deux instructions *distinctes* exécutées au même point.

La méthode présentée dans cette section est celle introduite par Bastoul dans [Bas04]. Elle présente en effet de nombreux avantages pour la vérification, dont en particulier le fait qu'elle ne nécessite pas d'opérations complexes sur des matrices, et qu'elle fonctionne pour tout ordonnancement d'entrée. Son désavantage majeur est qu'elle augmente significativement le nombre de variables qui doivent être traitées dans la suite de la génération de code et donc ralentit celle-ci. Comme l'objectif de notre générateur de code n'est pas d'être rapide mais d'être correct, cela n'est pas un problème réellement important.

L'idée de cette transformation est simple : elle ajoute en tête de la liste de variables de nouvelles variables, une pour chaque dimension de l'ordonnancement, et les fixe égales à la valeur spécifiée par la fonction d'ordonnancement dans les contraintes du polyèdre.

Ainsi, en supposant que les dimensions d'arrivée de toutes les fonctions d'ordonnancement θ soient identiques, on effectue la transformation suivante, qui pour chaque instruction polyédrique $(\mathtt{I}, \mathcal{D}, \theta, \mathcal{T})$ de $\mathcal{P}$ génère

$(\mathtt{I}, \mathcal{D}', \theta', \mathcal{T}')$, où :

$$\mathcal{D} = \left\{ \begin{pmatrix} \mathcal{E} \\ y \end{pmatrix} \middle/ \begin{pmatrix} \mathcal{E} \\ y \end{pmatrix} \in \mathbb{Z}^n, \begin{pmatrix} A_e & A_v \end{pmatrix} \begin{pmatrix} \mathcal{E} \\ y \end{pmatrix} \leq a \right\},$$

$$\theta : \begin{pmatrix} \mathcal{E} \\ y \end{pmatrix} \mapsto \begin{pmatrix} K_e & K_v \end{pmatrix} \begin{pmatrix} \mathcal{E} \\ y \end{pmatrix} + k,$$

$$\mathcal{T} : \begin{pmatrix} \mathcal{E} \\ y \end{pmatrix} \mapsto \begin{pmatrix} T_e & T_v \end{pmatrix} \begin{pmatrix} \mathcal{E} \\ y \end{pmatrix} + t$$

et :

$$\mathcal{D}' = \left\{ \begin{pmatrix} \mathcal{E} \\ u \\ y \end{pmatrix} \middle/ \begin{pmatrix} \mathcal{E} \\ u \\ y \end{pmatrix} \in \mathbb{Z}^{n'}, \left(\begin{array}{ccc} A_e & 0 & A_v \\ \hline -K_e & Id & -K_v \end{array} \right) \begin{pmatrix} \mathcal{E} \\ u \\ y \end{pmatrix} \begin{matrix} \leq \\ = \end{matrix} \begin{pmatrix} a \\ k \end{pmatrix} \right\},$$

$$\theta' : \begin{pmatrix} \mathcal{E} \\ u \\ y \end{pmatrix} \mapsto \begin{pmatrix} \mathcal{E} \\ u \\ y \end{pmatrix},$$

$$\mathcal{T}' : \begin{pmatrix} \mathcal{E} \\ u \\ y \end{pmatrix} \mapsto \begin{pmatrix} T_e & 0 & T_v \end{pmatrix} \begin{pmatrix} \mathcal{E} \\ u \\ y \end{pmatrix} + t$$

En effet, par définition de $\mathcal{D}'$, on a :

$$\begin{pmatrix} \mathcal{E} \\ u \\ y \end{pmatrix} \in \mathcal{D}' \Leftrightarrow \begin{pmatrix} \mathcal{E} \\ y \end{pmatrix} \in \mathcal{D} \wedge u = \theta \begin{pmatrix} \mathcal{E} \\ y \end{pmatrix}.$$

La fonction $\Gamma : \mathcal{D} \to \mathcal{D}'$ définie par :

$$\Gamma : \begin{pmatrix} \mathcal{E} \\ y \end{pmatrix} \mapsto \begin{pmatrix} \mathcal{E} \\ \theta \begin{pmatrix} \mathcal{E} \\ y \end{pmatrix} \\ y \end{pmatrix}$$

est donc une bijection de $\mathcal{D}$ dans $\mathcal{D}'$, d'inverse $\begin{pmatrix} \mathcal{E} \\ u \\ y \end{pmatrix} \mapsto \begin{pmatrix} \mathcal{E} \\ y \end{pmatrix}$.

On remarque de plus que la nouvelle fonction de transformation $\mathcal{T}'$ a été choisie de manière à ce que pour tous $x \in \mathcal{D}$, $\mathcal{T}(x) = \mathcal{T}'(\Gamma(x))$.

De plus, si $\begin{pmatrix} \mathcal{E} \\ u_1 \\ y_1 \end{pmatrix} \in \mathcal{D}'_1$ et $\begin{pmatrix} \mathcal{E} \\ u_2 \\ y_2 \end{pmatrix} \in \mathcal{D}'_2$, alors on a pour l'ordonnancement

(où on note $\prec$ l'ordre lexicographique) :

$$\begin{aligned}\begin{pmatrix}\mathcal{E}\\u_1\\y_1\end{pmatrix} \prec \begin{pmatrix}\mathcal{E}\\u_2\\y_2\end{pmatrix} &\Leftrightarrow (u_1 \prec u_2) \vee (u_1 = u_2 \wedge y_1 \prec y_2)\\ &\Leftrightarrow \theta_1\begin{pmatrix}\mathcal{E}\\y_1\end{pmatrix} \prec \theta_2\begin{pmatrix}\mathcal{E}\\y_2\end{pmatrix} \vee (u_1 = u_2 \wedge y_1 \prec y_2)\\ &\Leftarrow \theta_1\begin{pmatrix}\mathcal{E}\\y_1\end{pmatrix} \prec \theta_2\begin{pmatrix}\mathcal{E}\\y_2\end{pmatrix},\end{aligned}$$

ce qu'on peut reformuler en : si $x_1 \in \mathcal{D}_1$ et $x_2 \in \mathcal{D}_2$ correspondent au même environnement $\mathcal{E}$, on a :

$$\theta_1(x_1) \prec \theta_2(x_2) \Rightarrow \theta'_1(\Gamma_1(x_1)) \prec \theta'_2(\Gamma_2(x_2)),$$

ou encore :

$$\theta'_1(\Gamma_1(x_1)) \preccurlyeq \theta'_2(\Gamma_2(x_2)) \Rightarrow \theta_1(x_1) \preccurlyeq \theta_2(x_2).$$

En particulier, pour un environnement $\mathcal{E}$ fixé, tout ordre valide utilisé pour ordonner les instructions lors de l'exécution de $\mathcal{P}'$ est un ordre valide pour les ordonner lors de l'exécution de $\mathcal{P}$, la propriété $\mathcal{T} = \mathcal{T}' \circ \Gamma$ garantissant que les mêmes instructions sont exécutées. Ainsi, toute exécution de $\mathcal{P}'$ correspond à une exécution de $\mathcal{P}$, et cette transformation est correcte. On a également atteint le but recherché, puisque les θ' sont désormais l'identité, et l'ordre est donc un ordre lexicographique sur les points des $\mathcal{D}'$ eux-mêmes.

4 Génération de l'arbre de syntaxe

La deuxième étape de la génération de code est de calculer la structure du programme en termes de nids de boucles imparfaitement imbriquées. Le flot de contrôle devient explicite, mais les conditions et bornes de boucles restent explicitement affines et sont représentées par des polyèdres. Le langage intermédiaire PolyLoop, qui est la cible de cette étape de la génération est présenté figure 4. On peut remarquer certaines ressemblances entre PolyLoop et les *schedule trees* de Verdoolaege et al. [Ver+14 ; GVC15], mais leurs objectifs sont différents : PolyLoop est une représentation intermédiaire, tandis que les *schedule trees* sont bien plus complexes car ils servent de représentation intermédiaire tout au long de la génération de code, sur laquelle on effectue des transformations successives.

L'algorithme utilisé pour faire la transformation d'un programme polyédrique avec ordonnancement implicite vers PolyLoop est basé sur celui présenté par Quilleré et al. dans [QRW00], et est présenté figure 5. Une différence à noter est qu'il n'y a pas de contexte dans lequel on génère les boucles, et que l'on effectue donc pas la simplification des bornes de boucles

$$\text{Expressions :} \quad e ::= (x_1, \ldots, x_k) \mapsto \left\lfloor \left(\sum_{i=1}^{k} a_i x_i + c \right) \Big/ d \right\rfloor \quad \text{avec} \begin{cases} a_i, c, d \in \mathbb{Z}, \\ d > 0 \end{cases}$$

$$\text{Polyèdres :} \quad \mathcal{P} ::= \left\{ (x_1, \ldots, x_k) \in \mathbb{Z}^k \Big/ A \begin{pmatrix} x_1 \\ \vdots \\ x_k \end{pmatrix} \leq a \right\} \quad \text{avec} \begin{cases} A \in \mathcal{M}_{p,k}(\mathbb{Z}), \\ a \in \mathbb{Z}^p \end{cases}$$

$$\text{Commandes :} \quad s ::= \mid \texttt{skip} \mid s_1; s_2 \mid \texttt{I}(e_1, \ldots, e_k) \mid \texttt{guard } \mathcal{P}\ s \mid \texttt{loop } \mathcal{P}\ s$$

(a) Syntaxe de PolyLoop

$$\textsc{Skip}\ \frac{}{\mathcal{E} \vdash (\texttt{skip}, \mathcal{M}) \Downarrow \mathcal{M}} \qquad \textsc{Seq}\ \frac{\mathcal{E} \vdash (s_1, \mathcal{M}_1) \Downarrow \mathcal{M}_2 \qquad \mathcal{E} \vdash (s_2, \mathcal{M}_2) \Downarrow \mathcal{M}_3}{\mathcal{E} \vdash (s_1; s_2, \mathcal{M}_1) \Downarrow \mathcal{M}_3}$$

$$\textsc{Instr}\ \frac{(\texttt{I}(e_1(\mathcal{E}), \ldots, e_k(\mathcal{E})), \mathcal{M}_1) \Downarrow_{\texttt{I}} \mathcal{M}_2}{\mathcal{E} \vdash (\texttt{I}(e_1, \ldots, e_k), \mathcal{M}_1) \Downarrow \mathcal{M}_2}$$

$$\textsc{GuardTrue}\ \frac{\mathcal{E} \in \mathcal{P} \qquad \mathcal{E} \vdash (s, \mathcal{M}_1) \Downarrow \mathcal{M}_2}{\mathcal{E} \vdash (\texttt{guard } \mathcal{P}\ s, \mathcal{M}_1) \Downarrow \mathcal{M}_2} \qquad \textsc{GuardFalse}\ \frac{\mathcal{E} \notin \mathcal{P}}{\mathcal{E} \vdash (\texttt{guard } \mathcal{P}\ s, \mathcal{M}) \Downarrow \mathcal{M}}$$

$$\textsc{Loop}\ \frac{\forall x, \mathcal{E} :: x \in \mathcal{P} \Leftrightarrow a \leq x < b \qquad \forall x \in [\![a, b[\![, (\mathcal{E} :: x \vdash (s, \mathcal{M}_x) \Downarrow \mathcal{M}_{x+1})}{\mathcal{E} \vdash (\texttt{loop } \mathcal{P}\ s, \mathcal{M}_a) \Downarrow \mathcal{M}_b}$$

(b) Sémantique à grands pas de PolyLoop

Figure 4 – Syntaxe et sémantique de PolyLoop

dans ce contexte. À la place, cette simplification se produit dans une phase séparée, qui élimine les contraintes redondantes d'un programme PolyLoop sans changer sa sémantique.

L'algorithme présenté figure 5 est utilisé pour générer un arbre de syntaxe abstraite correspondant à un programme polyédrique donné. Pour cela, il procède dimension par dimension, en commençant par la dimension la plus externe à générer : ainsi, les appels récursifs traitent la variable qui vient d'être générée comme un paramètre de plus.

La première étape de l'algorithme traite du cas où on a plus aucune variable sur lesquelles générer du code. Il s'agit alors de générer une instruction pour chaque instruction existante, avec un nœud `guard` pour s'assurer que l'instruction ne sera exécutée que si on est dans un environnement dans le-

Entrée : une liste d'instructions polyédriques à ordonnancement implicite $(\mathtt{I}_1, \mathcal{D}_1, \mathcal{T}_1), \ldots, (\mathtt{I}_n, \mathcal{D}_n, \mathcal{T}_n)$, la dimension actuelle d.
Sortie : un programme PolyLoop.

1. Si d est la dimension des $\mathcal{D}_i$, renvoyer le programme :

$$\texttt{guard } \mathcal{D}_1 \ \mathtt{I}_1(\mathcal{T}_1); \ldots; \texttt{guard } \mathcal{D}_n \ \mathtt{I}_n(\mathcal{T}_n)$$

2. Pour chaque polyèdre $\mathcal{D}_i$, calculer sa projection $\mathcal{P}_i$ sur les d premières dimensions, et construire la liste des $\mathcal{P}_i \to (\mathtt{I}_i, \mathcal{D}_i, \mathcal{T}_i)$.
3. Séparer les projections $\mathcal{P}_i$ en une liste de polyèdres disjoints et leurs instructions associées : de $\mathcal{P}_1 \to \mathcal{I}_1$ et $\mathcal{P}_2 \to \mathcal{I}_2$, on obtient $(\mathcal{P}_1 \cap \mathcal{P}_2) \to (\mathcal{I}_1; \mathcal{I}_2)$, $(\mathcal{P}_1 - \mathcal{P}_2) \to \mathcal{I}_1$, et $(\mathcal{P}_2 - \mathcal{P}_1) \to \mathcal{I}_2$, puis on répète avec les autres polyèdres à séparer. Les différences $\mathcal{P}_1 - \mathcal{P}_2$ et $\mathcal{P}_2 - \mathcal{P}_1$ ne sont pas des polyèdres, mais des unions de polyèdres disjoints : la liste obtenue est alors d'une longueur potentiellement très élevée.
4. Calculer le graphe de dépendances, qui possède une arête $\mathcal{P} \to \mathcal{P}'$ si la boucle sur $\mathcal{P}$ doit être placée avant la boucle sur $\mathcal{P}'$ dans le programme obtenu pour respecter l'ordre lexicographique, puis effectuer un tri topologique. Si ce graphe possède un cycle, la génération échoue.
5. Pour chaque $\mathcal{P} \to ((\mathtt{I}_1, \mathcal{D}_1, \mathcal{T}_1); \ldots; (\mathtt{I}_k, \mathcal{D}_k, \mathcal{T}_k))$, soit s le résultat d'un appel récursif avec la dimension actuelle $d+1$ et la liste d'instructions $(\mathtt{I}_1, \mathcal{D}_1 \cap \mathcal{P}, \mathcal{T}_1), \ldots, (\mathtt{I}_k, \mathcal{D}_k \cap \mathcal{P}, \mathcal{T}_k)$; créer l'instruction $\texttt{loop } \mathcal{P}\ s$.
6. Renvoyer le programme formé par la séquence des résultats de l'étape précédente.

Figure 5 – Transformation d'un programme polyédrique en un programme PolyLoop

quel elle doit être exécutée (ces nœuds sont la plupart du temps redondants et seront éliminés plus tard). On note ici une seconde résolution arbitraire d'ordonnancement : les instructions sont exécutées dans l'ordre d'entrée de l'algorithme.

Le reste de l'algorithme traite du cas d'une dimension supplémentaire. On calcule tout d'abord les projections sur les dimensions actuelles plus la nouvelle dimension, et on effectue une disjonction de cas (étapes 2 et 3) selon dans quels polyèdres d'entrée l'environnement actuel pourra se trouver. Ainsi, chaque instruction ne sera que dans les boucles internes pour lesquelles il est nécessaire qu'elle soit présente.

L'étape 4 se charge de trier les polyèdres résultants : en effet, on souhaite ordonner dans le programme PolyLoop obtenu les différentes boucles qui seront générées dans un ordre ou un autre ; or, cela veut dire que dans tout environnement, une de ces deux boucles sera toujours exécutée avant l'autre.

Quilleré et al. montrent [QRW00] qu'il est toujours possible d'ordonner deux polyèdres disjoints pour respecter l'ordre d'exécution voulu. On peut énoncer la propriété « $\mathcal{P}_1$ peut être ordonné avant $\mathcal{P}_2$ » comme :

$$\forall \mathcal{E}, x_1, x_2, \mathcal{E} :: x_1 \in \mathcal{P}_1 \wedge \mathcal{E} :: x_2 \in \mathcal{P}_2 \Rightarrow x_1 < x_2.$$

Cela revient en effet à dire que dans tout environnement, toutes les itérations de $\mathcal{P}_1$ seront avant toutes celles de $\mathcal{P}_2$. Ainsi, $\mathcal{P}_2$ doit être ordonné avant $\mathcal{P}_1$ si et seulement si :

$$\begin{aligned} & \exists \mathcal{E}, x_1, x_2, \mathcal{E} :: x_1 \in \mathcal{P}_1 \wedge \mathcal{E} :: x_2 \in \mathcal{P}_2 \wedge x_2 \leq x_1 \\ \Leftrightarrow\ & \exists \mathcal{E}, x_2, \mathcal{E} :: x_2 \in \mathcal{P}_2 \wedge \mathcal{E} :: x_2 \in \widehat{\mathcal{P}_1} \\ \Leftrightarrow\ & \widehat{\mathcal{P}_1} \cap \mathcal{P}_2 \neq \emptyset, \end{aligned}$$

avec $\widehat{\mathcal{P}_1} = \overline{\mathcal{P}_1} \cap \overrightarrow{\mathcal{P}_1}$, $\overline{\mathcal{P}_1}$ la projection de $\mathcal{P}_1$ par rapport à x_1, et $\overrightarrow{\mathcal{P}_1}$ le polyèdre obtenu en ne gardant que les contraintes de $\mathcal{P}_1$ pour lesquelles le coefficient devant x_1 est positif.

On a alors en effet :

$$\mathcal{E} :: x_2 \in \widehat{\mathcal{P}_1} \Leftrightarrow \exists x_1, \mathcal{E} :: x_1 \in \mathcal{P}_1 \wedge x_2 \leq x_1.$$

Cependant, dans le cas de plus de deux polyèdres, il se peut qu'il n'existe pas d'ordre qui correspond, le graphe de dépendances généré contenant un cycle : en ce cas, notre algorithme échoue [1].

Les étapes 5 et 6 se chargent d'effectuer un appel récursif sur chaque polyèdre sur lequel on doit générer une boucle, créer le nœud de boucle, et créer la séquence des résultats obtenus, pour obtenir un programme PolyLoop dont la sémantique respecte bien la sémantique du programme polyédrique initial.

La correction de cet algorithme de génération de code se prouve en montrant, par récurrence sur le nombre de dimensions restant à traiter, que dans tout environnement, toute exécution d'un programme généré correspond à une exécution du programme polyédrique initial. Pour traiter le cas d'une dimension supplémentaire (le cas de base étant immédiat), il faut montrer les résultats suivants sur les polyèdres obtenus à l'issue de l'étape 4 :

- Si $\mathcal{P}$ est avant $\mathcal{P}'$ dans la liste de polyèdres obtenus, alors $\mathcal{P}$ peut être ordonné avant $\mathcal{P}'$ dans tout environnement,
- Tous les polyèdres obtenus sont disjoints,
- Si $x \in \mathcal{P}_i$ à l'étape 2, alors x est dans l'un des $\mathcal{P}$ de la liste obtenue, et l'instruction $\mathcal{I}_i$ correspondante apparaît dans la liste d'instructions de $\mathcal{P}$.

Ces résultats proviennent des algorithmes sur les polyèdres gérant les étapes 3 et 4 de l'algorithme.

1. Un contre-exemple est fourni en annexe B. Il se trouve que des propriétés supplémentaires de l'algorithme de génération font que ce cas ne se produit pas en pratique, mais cela n'a pas été formalisé

$$
\begin{aligned}
\texttt{psimpl}(\texttt{skip}, \mathcal{C}) &= \texttt{skip} \\
\texttt{psimpl}(s1; s2, \mathcal{C}) &= \texttt{psimpl}(s1, C); \texttt{psimpl}(s2, C) \\
\texttt{psimpl}(\texttt{I}(e_1, \ldots, e_k), \mathcal{C}) &= \texttt{I}(e_1, \ldots, e_k) \\
\texttt{psimpl}(\texttt{guard}\ \mathcal{P}\ s, \mathcal{C}) &= \texttt{guard}\ (\texttt{simplify}(\mathcal{P}, \mathcal{C}))\ \texttt{psimpl}(s, \mathcal{P} \cap \mathcal{C}) \\
\texttt{psimpl}(\texttt{loop}\ \mathcal{P}\ s, \mathcal{C}) &= \texttt{loop}\ (\texttt{simplify}(\mathcal{P}, \mathcal{C}))\ \texttt{psimpl}(s, \mathcal{P} \cap \mathcal{C})
\end{aligned}
$$

FIGURE 6 – Simplification d'un programme PolyLoop

5 Simplification des contraintes redondantes

L'algorithme de la section 4 génère de nombreuses contraintes redondantes, ce qui facilite la preuve de sa correction. En particulier, il insère un nœud `guard` avant chaque instruction, qui n'est la plupart du temps pas nécessaire car il est une conséquence directe des nœuds `loop` englobants. Cependant, l'algorithme est fait de manière à ce qu'il soit facile d'éliminer une importante fraction de ces contraintes a posteriori, sans devoir prouver qu'il est possible de les enlever : si jamais il s'avérait que leur élimination n'est pas possible, ces contraintes sont simplement conservées, et le code généré est alors un peu moins efficace.

Pour simplifier les contraintes redondantes, on a besoin d'une unique primitive `simplify`, qui prend en entrée un polyèdre $\mathcal{P}$ et un contexte $\mathcal{C}$, et revoie un polyèdre $\mathcal{P}'$ vérifiant la propriété $\mathcal{P}' \cap \mathcal{C} = \mathcal{P} \cap \mathcal{C}$, ce qui peut se reformuler $\forall x \in \mathcal{C}, x \in \mathcal{P}' \Leftrightarrow x \in \mathcal{P}$. Le but est que `simplify` renvoie un polyèdre décrit par un minimum de contraintes, puisqu'il s'agit de ces contraintes qui seront conservées dans le code généré.

On peut alors définir une fonction `psimpl` par les règles de la figure 6. Dans ces règles, on a passé sous silence un détail concernant le fait que le contexte $\mathcal{C}$ et les polyèdres des instructions `guard` et `loop` ont uniquement le droit de se référer aux variables déjà déclarées. Modulo ce détail, on peut aisément prouver le théorème qui nous intéresse : si $\mathcal{E} \in \mathcal{C}$, alors $\mathcal{E} \vdash (\texttt{psimpl}(\mathcal{C}, s), \mathcal{M}_1) \Downarrow \mathcal{M}_2$ est dérivable si et seulement si $\mathcal{E} \vdash (s, \mathcal{M}_1) \Downarrow \mathcal{M}_2$ l'est.

6 Génération de code Loop

La dernière étape de la génération de code est de transformer les expressions ainsi que les constructions `guard` et `loop` de PolyLoop vers un langage de plus bas niveau, Loop, qui n'a plus accès aux constructions polyédriques abstraites de PolyLoop. Sa syntaxe et sa sémantiques sont définies figure 7.

Expressions : Tests : Commandes :

$$
\begin{array}{lll}
e ::= \mid c \mid x & t ::= \mid e_1 = e_2 & s ::= \mid \texttt{skip} \mid s_1\texttt{;}s_2 \\
\quad \mid e_1 + e_2 & \quad \mid e_1 \leq e_2 & \quad \mid \texttt{I}(e_1, \ldots, e_k) \\
\quad \mid c \cdot e & \quad \mid t_1 \texttt{ \&\& } t_2 & \quad \mid \texttt{if (}t\texttt{) } s \\
\quad \mid \lfloor e/c \rfloor \mid e \texttt{ mod } c & \quad \mid t_1 \texttt{ || } t_2 & \quad \mid \texttt{for (}i \texttt{ = } e_1\texttt{; } i \texttt{ < } e_2\texttt{; } i\texttt{++) } s \\
\quad \mid \texttt{min}(e_1, e_2) & \quad \mid \texttt{!}t & \\
\quad \mid \texttt{max}(e_1, e_2) & \quad \mid \texttt{true} \mid \texttt{false} &
\end{array}
$$

(a) Syntaxe de Loop

$$\textsc{Skip}\ \frac{}{\mathcal{E} \vdash (\texttt{skip}, \mathcal{M}) \Downarrow \mathcal{M}} \qquad \textsc{Seq}\ \frac{\mathcal{E} \vdash (s_1, \mathcal{M}_1) \Downarrow \mathcal{M}_2 \qquad \mathcal{E} \vdash (s_2, \mathcal{M}_2) \Downarrow \mathcal{M}_3}{\mathcal{E} \vdash (s_1\texttt{;}s_2, \mathcal{M}_1) \Downarrow \mathcal{M}_3}$$

$$\textsc{Instr}\ \frac{(\texttt{I}(\texttt{eval}(\mathcal{E}, e_1), \ldots, \texttt{eval}(\mathcal{E}, e_k)), \mathcal{M}_1) \Downarrow_{\texttt{I}} \mathcal{M}_2}{\mathcal{E} \vdash (\texttt{I}(e_1, \ldots, e_k), \mathcal{M}_1) \Downarrow \mathcal{M}_2}$$

$$\textsc{IfTrue}\ \frac{\texttt{eval}(\mathcal{E}, t) = \texttt{true} \qquad \mathcal{E} \vdash (s, \mathcal{M}_1) \Downarrow \mathcal{M}_2}{\mathcal{E} \vdash (\texttt{if (}t\texttt{) } s, \mathcal{M}_1) \Downarrow \mathcal{M}_2} \qquad \textsc{IfFalse}\ \frac{\texttt{eval}(\mathcal{E}, t) = \texttt{false}}{\mathcal{E} \vdash (\texttt{if (}t\texttt{) } s, \mathcal{M}) \Downarrow \mathcal{M}}$$

$$\textsc{For}\ \frac{\forall x \in [\![\texttt{eval}(\mathcal{E}, e_1), \texttt{eval}(\mathcal{E}, e_2)[\![, (\mathcal{E}, i : x \vdash (s, \mathcal{M}_x) \Downarrow \mathcal{M}_{x+1})}{\mathcal{E} \vdash (\texttt{for (}i \texttt{ = } e_1\texttt{; } i \texttt{ < } e_2\texttt{; } i\texttt{++) } s, \mathcal{M}_{\texttt{eval}(\mathcal{E}, e_1)}) \Downarrow \mathcal{M}_{\texttt{eval}(\mathcal{E}, e_2)}}$$

(b) Sémantique de Loop

Figure 7 – Syntaxe et sémantique de Loop

La transformation d'un programme PolyLoop en programme Loop doit donc faire trois opérations :

— Transformer les expressions de PolyLoop en expressions de Loop,
— Transformer les constructions `guard` en constructions `if`, en remplaçant le polyèdre correspondant par son test d'appartenance,
— Transformer les constructions `loop` en constructions `for`, en calculant les bornes à donner aux boucles.

Grâce au nombre élevé de constructions disponibles dans les expressions de Loop, il est simple de traduire les expressions de PolyLoop en expressions Loop, et il est également aisé de traduire le test d'appartenance à un polyèdre utilisé pour `guard` en un test de Loop en utilisant l'opérateur `&&`.

C'est le cas de `loop` qui pose le plus de problèmes. Pour traduire une instruction `guard` $\mathcal{P}$ s, après avoir traduit s en s', il faut faire les opérations

suivantes, où i est le nom de la nouvelle variable :

- On décompose les contraintes de $\mathcal{P}$ (qui sont de la forme $a \cdot x \leq c$) en trois groupes : un premier groupe, $\mathcal{P}_0$, où le coefficient de i est nul ; un deuxième groupe, $\mathcal{P}_+$, où ce coefficient est strictement positif, et un troisième groupe, $\mathcal{P}_-$ où il est strictement négatif.
- Si $\mathcal{P}_+$ ou $\mathcal{P}_-$ n'a aucune contrainte, la génération échoue.
- Pour chaque contrainte $ui + \sum_k a_k x_k \leq c$ de $\mathcal{P}_+$, on forme l'expression $\lfloor (c + u - \sum_k a_k x_k)/u \rfloor$, puis on forme l'expression correspondant à la borne supérieure e_+ comme le minimum de toutes ces expressions (cela est possible car $\mathcal{P}_+$ contient au moins une contrainte).
- De même, pour chaque contrainte $ui + \sum_k a_k x_k \leq c$ de $\mathcal{P}_-$, on forme l'expression $\lfloor (\sum_k a_k x_k - c - u - 1)/(-u) \rfloor$, puis on forme l'expression correspondant à la borne inférieure e_- comme le maximum de toutes ces expressions (cela est possible car $\mathcal{P}_-$ contient au moins une contrainte).
- On forme le test t_0 d'appartenance à $\mathcal{P}_0$.
- On renvoie le résultat `if (`t_0`) for (`i `=` e_-`; `i `<` e_+`; `i`++) `s'.

Il est relativement facile, à l'aide de résultats usuels sur la division entière, de voir que pour tout $\mathcal{E}$:

$$\mathcal{E} :: x \in \mathcal{P} \Leftrightarrow \mathcal{E} \in \mathcal{P}_0 \wedge \texttt{eval}(\mathcal{E}, e_-) \leq x < \texttt{eval}(\mathcal{E}, e_+).$$

Par conséquence, en distinguant le cas où $\mathcal{E} \in \mathcal{P}_0$ ou non, on obtient que la sémantique du programme Loop obtenu est la même que celle du programme PolyLoop initial, en supposant que les sémantiques de s et s' sont les mêmes.

Pour simplifier légèrement plus le code généré, on distingue également le cas où $\mathcal{P}$ contient une égalité sur i. En effet, comme dans ce cas on connaît l'unique valeur possible de i, on peut faire un simple test pour vérifier si la boucle est vide ou non (ce test contenant un test de divisibilité si le coefficient devant i est différent de 1), puis générer une boucle qui contient toujours une unique itération.

7 Développement Coq

La totalité des algorithmes présentés ci-dessus ont été formalisés et prouvés en Coq. Pour cela, il a fallu développer une petite bibliothèque d'algèbre linéaire, capable de gérer les opérations dont nous avons eu besoin. Différentes bibliothèques sont utilisées en compilation polyédrique pour fournir les opérations de base, comme Polylib [Wil93] ou isl [Ver10] ; mais il n'en existe pas de vérifiée. Nous avons donc utilisé la VPL (Verified Polyhedra Library) [Fou+], une bibliothèque vérifiée pour effectuer des calculs sur des polyèdres à des fins d'interprétation abstraite, qui nous a permis d'obtenir un opérateur pour déterminer si un polyèdre est vide, et un opérateur de

canonisation de polyèdres. Nous avons cependant dû réimplanter les autres opérateurs (en particulier la projection) nous-mêmes, la VPL ne fournissant pas de certificat de précision de ses opérations. Il a également fallu formaliser la sémantique des différents langages présentés plus haut pour pouvoir énoncer les théorèmes de correction de la génération de code.

7.1 Algèbre linéaire

Les vecteurs sont représentés par des listes d'entiers. Pour que tout vecteur soit acceptable dans tout contexte (afin de limiter le nombre d'hypothèses des différents théorèmes), on définit l'égalité de vecteur `x =v= y` comme étant l'égalité de `x` et `y` aux zéros de fin près. Cette égalité est déclarée avec `Setoid` pour pouvoir être aisément utilisée dans le reste du code.

```
Definition is_null xs := forallb (fun x ⇒ x =? 0) xs.

Fixpoint is_eq (xs ys : list Z) :=
  match xs, ys with
  | nil, nil ⇒ true
  | nil, ys ⇒ is_null ys
  | xs, nil ⇒ is_null xs
  | x :: xs, y :: ys ⇒ (x =? y) && is_eq xs ys
  end.

Definition veq xs ys := is_eq xs ys = true.
Infix "=v=" := veq (at level 70) : vector_scope.
```

On définit également les opérations standard sur les vecteurs : addition, multiplication par un scalaire, produit scalaire, mais également d'autres opérations nécessaires, comme l'ordre lexicographique. On définit une contrainte comme étant une paire d'un vecteur et d'un scalaire (la paire `(a, c)` représentant la contrainte $\mathtt{a} \cdot x \leq \mathtt{c}$). Un polyèdre est simplement une liste de contraintes.

7.2 Opérations sur des polyèdres

On a besoin d'un certain nombre d'opérations sur les polyèdres également. Tester si un polyèdre est vide, et réduire le nombre de contraintes d'un polyèdre, se font en appelant la VPL. Cependant, il faut pour cela construire un polyèdre de la VPL équivalent à notre représentation de polyèdres. Outre la représentation des contraintes différentes, la difficulté majeure provient du fait que la VPL ne fournit aucune fonction de construction de polyèdre : il faut donc ajouter les contraintes au fur et à mesure, puis vérifier que toutes les contraintes ajoutées sont satisfaites à la fin, puisque l'ajout aura pu introduire une sur-approximation. Cette conversion peut donc échouer, ce qu'il faut prendre en compte dans notre code. Par ailleurs, le test de si un polyèdre est vide de la VPL ne donne aucune garantie qu'un polyèdre est non-vide

si jamais le test est négatif ; heureusement, ce n'est pas un problème pour nous.

Munis de ces deux primitives, nous pouvons définir les autres opérations polyédriques qu'il nous faut. L'intersection est de loin la plus simple : il suffit de concaténer les listes de contraintes des deux polyèdres, puis de canoniser le résultat obtenu pour éliminer les contraintes redondantes.

```
Definition poly_inter_pure (p1 p2 : polyhedron) : polyhedron := p1 ++ p2.
Lemma poly_inter_pure_def :
  forall p pol1 pol2,
    in_poly p (poly_inter_pure pol1 pol2) =
      in_poly p pol1 && in_poly p pol2.

Definition poly_inter p1 p2 :=
  Canon.canonize (poly_inter_pure p1 p2).
Lemma poly_inter_def :
  forall p pol1 pol2,
    WHEN pol ← poly_inter pol1 pol2 THEN
    in_poly p pol = in_poly p (poly_inter_pure pol1 pol2).
```

Une autre opération nécessaire est la projection sur les d premières dimensions. Elle est implémentée à partir de projections itérées par rapport à chaque autre dimension, mais il faut donc définir la projection par rapport à une variable donnée. Pour cela, on effectue une projection en utilisant l'algorithme d'élimination de Fourier-Motzkin. On prouve ensuite que cette opération de projection est exacte sur $\mathbb{Q}$.

```
Definition isExactProjection n pol proj :=
  forall p s, 0 < s →
    in_poly p (expand_poly s proj) = true ↔
    exists t k, 0 < t ∧
      in_poly (assign n k (mult_vector t p)) (expand_poly (s * t) pol) = true.

Theorem pure_project_in_iff :
  forall n pol, isExactProjection n pol (pure_project n pol).
```

Les opérations polyédriques restantes sont celles utilisées aux étapes 3 et 4 de l'algorithme de génération d'arbre de syntaxe abstrait. Pour pouvoir définir plus simplement la fonction de traduction d'un programme polyédrique en programme PolyLoop, ces deux étapes ont été fusionnées dans la fonction `split_and_sort`. Les propriétés ci-dessous sont celles énoncées à la section 4 qui permettent de montrer la correction de l'algorithme de génération de code.

```
Lemma split_and_sort_disjoint :
  forall n pols,
    WHEN out ← split_and_sort n pols THEN
    forall p k1 k2 ppl1 ppl2,
      nth_error out k1 = Some ppl1 → nth_error out k2 = Some ppl2 →
      in_poly p (fst ppl1) = true → in_poly p (fst ppl2) = true → k1 = k2.
```

```
Lemma split_and_sort_cover :
  forall n pols,
    WHEN out ← split_and_sort n pols THEN
    forall p pol i,
      nth_error pols i = Some pol → in_poly p pol = true →
      exists ppl, In ppl out ∧ In i (snd ppl) ∧ in_poly p (fst ppl) = true.

Lemma split_and_sort_sorted :
  forall n pols,
    WHEN out ← split_and_sort n pols THEN
    forall k1 k2 ppl1 ppl2,
      nth_error out k1 = Some ppl1 → nth_error out k2 = Some ppl2 →
      (k1 < k2)%nat → canPrecede n (fst ppl1) (fst ppl2).
```

7.3 Sémantiques

On a ici deux types de sémantiques : les sémantiques polyédriques (pour notre langage polyédrique, dans le cas d'un ordonnancement explicite ou implicite lexicographique), et les sémantiques sur les programmes avec arbre de syntaxe. Ces dernières sont les plus simples à spécifier, et sont une simple traduction des règles des figures 4 et 7 sous forme d'un `Inductive` de Coq. Ci-dessous, la définition de la sémantique de PolyLoop, celle de Loop étant similaire.

```
Inductive poly_loop_semantics : poly_stmt → list Z → mem → mem → Prop :=
| PLInstr : forall i es env mem1 mem2,
    instr_semantics i (map (eval_affine_expr env) es) mem1 mem2 →
    poly_loop_semantics (PInstr i es) env mem1 mem2
| PLSkip : forall env mem, poly_loop_semantics PSkip env mem mem
| PLSeq : forall env st1 st2 mem1 mem2 mem3,
    poly_loop_semantics st1 env mem1 mem2 →
    poly_loop_semantics st2 env mem2 mem3 →
    poly_loop_semantics (PSeq st1 st2) env mem1 mem3
| PLGuardTrue : forall env t st mem1 mem2,
    poly_loop_semantics st env mem1 mem2 →
    in_poly (rev env) t = true →
    poly_loop_semantics (PGuard t st) env mem1 mem2
| PLGuardFalse : forall env t st mem,
    in_poly (rev env) t = false → poly_loop_semantics (PGuard t st) env mem mem
| PLLoop : forall env p lb ub st mem1 mem2,
    (forall x, in_poly (rev (x :: env)) p = true ↔ lb ≤ x < ub) →
    iter_semantics (fun x ⇒ poly_loop_semantics st (x :: env)) (Zrange lb ub) mem1 mem2 →
    poly_loop_semantics (PLoop p st) env mem1 mem2.
```

D'un autre côté, la formalisation des sémantiques polyédriques est plus complexe. On les paramétrise par une fonction indiquant quels points des polyèdres correspondant à une instruction devant encore être exécutée, qui

est ensuite choisie comme précisément les points des différents polyèdres du programme.

```
Inductive poly_lex_semantics : (nat → list Z → bool) →
      Poly_Program → mem → mem → Prop :=
| PolyLexDone : forall to_scan prog mem,
    (forall n p, to_scan n p = false) → poly_lex_semantics to_scan prog mem mem
| PolyLexProgress : forall to_scan prog mem1 mem2 mem3 poly_instr n p,
    to_scan n p = true → nth_error prog n = Some poly_instr →
    (forall n2 p2, lex_compare p2 p = Lt → to_scan n2 p2 = false) →
    instr_semantics poly_instr.(pi_instr)
      (affine_product poly_instr.(pi_transformation) p) mem1 mem2 →
    poly_lex_semantics (scanned to_scan n p) prog mem2 mem3 →
    poly_lex_semantics to_scan prog mem1 mem3.

Definition env_scan (prog : Poly_Program)
      (env : list Z) (dim : nat) (n : nat) (p : list Z) :=
  match nth_error prog n with
  | Some pi ⇒ is_eq env (resize (length env) p) &&
                is_eq p (resize dim p) && in_poly p pi.(pi_poly)
  | None ⇒ false
  end.

Definition env_poly_lex_semantics (env : list Z)
      (dim : nat) (prog : Poly_Program) (mem1 mem2 : mem) :=
  poly_lex_semantics (env_scan prog env dim) prog mem1 mem2.
```

7.4 Théorèmes de correction

On a un théorème de préservation de la sémantique pour chacune des étapes de la génération de code présentées précédemment. Ils sont présentés individuellement en annexe D. La composition de toutes ces transformations et de tous ces théorèmes permet de prouver le théorème de conservation global :

```
Theorem complete_generate_many_preserve_sem :
  forall es n pis env mem1 mem2,
    (es ≤ n)%nat →
    WHEN st ← complete_generate_many es n pis THEN
    loop_semantics st env mem1 mem2 →
    length env = es →
    pis_have_dimension pis n →
    (forall pi, In pi pis → (poly_nrl pi.(pi_schedule) ≤ n)%nat) →
    env_poly_semantics (rev env) n pis mem1 mem2.
```

On peut noter ici quelques hypothèses techniques qui nous assurent que le programme initial est bien formé, qui sont nécessaires à prouver la correction du générateur de code. En effet, dans ce théorème, `n` est la dimension totale du programme d'entrée. Les deux hypothèses `pis_have_dimension pis n` et

`forall pi, In pi pis → (poly_nrl pi.(pi_schedule) ≤ n)%nat` assurent que le programme d'entrée ne fait pas référence à des variables non-existantes. La génération de code s'effectue de plus dans un environnement de taille fixée, c'est à dire, où on a déjà décidé ce qui sera une variable et ce qui sera un paramètre. La variable `es` est la taille de cet environnement ; on a donc les deux conditions `length env = es`, qui assurent que l'environnement utilisé aura bien la bonne taille, et `(es ≤ n)%nat`, qui assure que la taille de cet environnement n'est pas plus grande que la dimension du programme d'entrée. Enfin, la condition restante est simplement le fait que la génération a réussi en produisant un code `st` (`WHEN st ← complete_generate_many es n pis THEN ...`). On en conclut alors que toute exécution de `st` correspond à une exécution du programme initial `pis`.

Le développement Coq complet est disponible à l'adresse `https://github.com/Ekdohibs/PolyGen`.

8 Conclusion

Ce travail est ainsi la première implantation formellement vérifiée de la partie de génération de code du modèle polyédrique, et est donc un premier pas vers l'intégration d'optimisations de nids de boucles dans des compilateurs formellement vérifiés. Elle permettrait également une compilation formellement vérifiée de langages dédiés pour faire du calcul numérique efficace.

Parmi les objectifs à court terme, il faudrait intégrer les autres optimisations présentées dans [Bas04] et dans [GVC15], comme la détection de pas de boucle, l'élimination de code mort et la réduction de la taille du code pour obtenir du code plus efficace et plus compact en sortie.

Les objectifs à long terme seraient principalement de compléter la formalisation du modèle polyédrique pour obtenir un générateur complet formellement vérifié, puis une possible intégration à CompCert, soit en compilant un langage dédié, soit en ajoutant des optimisations de nids de boucles dans la chaîne de compilation CompCert. Il faudrait également pour cela considérer le problème des dépassements de capacité arithmétiques, qui a été traité dans [Cue+12].

Annexes

A Références

[AI91] Corinne Ancourt et François Irigoin. "Scanning Polyhedra with DO Loops". In : *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP'91)*. Williamsburg, Virginia, USA, avr. 1991, p. 39–50. doi : 10.1145/109625.109631. url : http://doi.acm.org/10.1145/109625.109631.

[Bas04] Cédric Bastoul. "Code Generation in the Polyhedral Model Is Easier Than You Think". In : *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT'04)*. Washington, DC, USA : IEEE Computer Society, 2004, p. 7–16. doi : 10.1109/PACT.2004.11. url : https://hal.archives-ouvertes.fr/hal-00017260/file/bastoul2004code.pdf.

[Bea+14] Ulysse Beaugnon et al. "VOBLA : a vehicle for optimized basic linear algebra". In : *Proceedings of the 2014 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems (LCTES'14)*. Edinburgh, United Kingdom, juin 2014, p. 115–124. doi : 10.1145/2666357.2597818. url : https://hal.inria.fr/hal-01508181.

[Ber66] A. J. Bernstein. "Analysis of Programs for Parallel Processing". In : *IEEE Transactions on Electronic Computers* EC-15.5 (oct. 1966), p. 757–763. doi : 10.1109/PGEC.1966.264565.

[BF98] Pierre Boulet et Paul Feautrier. "Scanning polyhedra without DO-loops". In : *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques*. Oct. 1998, p. 4–11. doi : 10.1109/PACT.1998.727127.

[BRS07] Uday Bondhugula, J. Ramanujam et P. Sadayappan. *PLuTo : A Practical and Fully Automatic Polyhedral Program Optimization System*. 2007.

[Cue+12] Bruno Cuervo Parrino et al. "Dealing with arithmetic overflows in the polyhedral model". In : *IMPACT 2012 - 2nd International Workshop on Polyhedral Compilation Techniques*. Paris, France, jan. 2012. url : https://hal.inria.fr/hal-00655485.

[Fea92a] Paul Feautrier. "Some efficient solutions to the affine scheduling problem. I. One-dimensional time". In : *International Journal of Parallel Programming* 21.5 (oct. 1992), p. 313–347. doi : 10.1007/BF01407835. url : https://doi.org/10.1007/BF01407835.

[Fea92b] Paul FEAUTRIER. "Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time". In : *International Journal of Parallel Programming* 21.6 (déc. 1992), p. 389–420. DOI : 10.1007/BF01379404. URL : https://doi.org/10.1007/BF01379404.

[Fou+] Alexis FOUILHÉ et al. "VPL (Verified Polyhedra Library)". URL : https://github.com/VERIMAG-Polyhedra/VPL.

[GVC15] Tobias GROSSER, Sven VERDOOLAEGE et Albert COHEN. "Polyhedral AST Generation Is More Than Scanning Polyhedra". In : *ACM Trans. Program. Lang. Syst.* 37.4 (juil. 2015), 12 :1–12 :50. DOI : 10.1145/2743016. URL : http://doi.acm.org/10.1145/2743016.

[Ler09] Xavier LEROY. "Formal verification of a realistic compiler". In : *Communications of the ACM* 52.7 (2009), p. 107–115. URL : http://xavierleroy.org/publi/compcert-CACM.pdf.

[Pil] Alexandre PILKIEWICZ. "s2sLoop". URL : https://github.com/pilki/s2sLoop.

[QR] Fabien QUILLERÉ et Sanjay V. RAJOPADHYE. "Alpha and the Polyhedral Model". URL : https://www.semanticscholar.org/paper/466c25e7f45b37fea6cac001b9979861b37f9996.

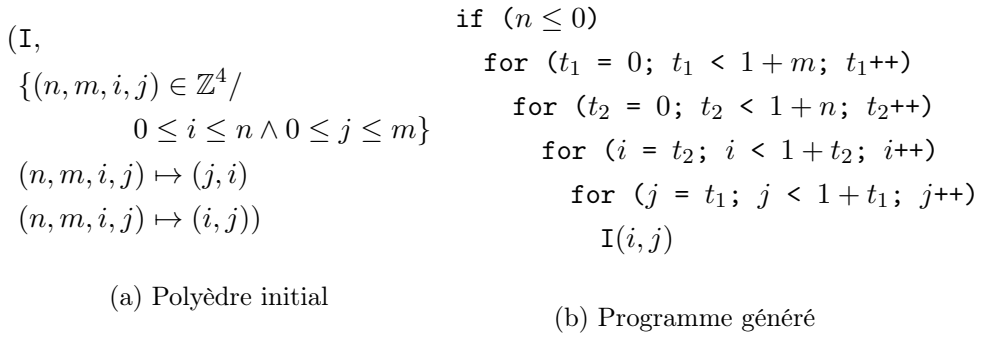
[QRW00] Fabien QUILLERÉ, Sanjay RAJOPADHYE et Doran WILDE. "Generation of efficient nested loops from polyhedra". In : *International Journal of Parallel Programming* 28.5 (2000), p. 469–498.

[Vas+18] Nicolas VASILACHE et al. *Tensor Comprehensions : Framework-Agnostic High-Performance Machine Learning Abstractions*. 2018. arXiv : 1802.04730. URL : http://arxiv.org/abs/1802.04730.

[Ver+14] Sven VERDOOLAEGE et al. "Schedule Trees". In : *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*. Vienna, Austria, jan. 2014.

[Ver10] Sven VERDOOLAEGE. "isl : An Integer Set Library for the Polyhedral Model". In : *Mathematical Software (ICMS'10)*. LNCS 6327. Springer-Verlag, 2010, p. 299–302.

[Wil93] Doran K. WILDE. *A Library for doing polyhedral operations*. Research Report RR-2157. INRIA, 1993. URL : https://hal.inria.fr/inria-00074515.



(a) Polyèdre initial

(b) Programme généré

FIGURE 8 – Génération de code pour deux boucles imbriquées

B Existence de cycles dans le graphe de dépendances

Considérons les trois polyèdres disjoints suivants :

$$\begin{aligned}
\mathcal{P}_1 &= \{(x,y,z) \in \mathbb{Z}^3 / x = 0 \wedge y = z \wedge 0 \leq z \leq 1\} \\
&= \{(0,0,0),(0,1,1)\} \\
\mathcal{P}_2 &= \{(x,y,z) \in \mathbb{Z}^3 / y = 0 \wedge x = 1 - z \wedge 0 \leq z \leq 1\} \\
&= \{(1,0,0),(0,0,1)\} \\
\mathcal{P}_3 &= \{(x,y,z) \in \mathbb{Z}^3 / x = 1 - y \wedge x = z \wedge 0 \leq z \leq 1\} \\
&= \{(0,1,0),(1,0,1)\}
\end{aligned}$$

On a $(0,0,0) \in \mathcal{P}_1$ et $(0,0,1) \in \mathcal{P}_2$, donc la boucle pour $\mathcal{P}_1$ doit précéder celle pour $\mathcal{P}_2$. Mais de même, celle pour $\mathcal{P}_2$ doit précéder celle pour $\mathcal{P}_3$, qui doit précéder celle pour $\mathcal{P}_1$: il n'y a donc pas d'ordre possible.

C Exemples de code générés

Un premier exemple, dans le cas d'un seul polyèdre, correspond à la génération de deux boucles imbriquées. On a alors un unique polyèdre pour lequel on doit générer le code. Cet exemple est présenté figure 8. On peut en particulier remarquer les boucles à un élément ayant été générées, qui peuvent être syntaxiquement détectées comme étant des simples `let`.

Un deuxième exemple, qui provient de [Bas04], se place dans le cas de plusieurs polyèdres. Pour éviter une augmentation déraisonnable de la taille du code produit (qui inclurait sinon de nombreuses boucles supplémentaires à un élément), le programme initial est un programme polyédrique avec ordonnancement lexicographique. Il est présenté figure 9. Le code produit n'est pas optimal : il pourrait être plus simplifié en utilisant les techniques présentées dans [Bas04], qui sont laissées en temps que travaux futurs.

$(\mathtt{I}_1,$
$\{(n,m,i,j) \in \mathbb{Z}^4 /$
$\quad 1 \leq i \leq n \wedge j = i\}$
$(n,m,i,j) \mapsto (i,j)),$

$(\mathtt{I}_2,$
$\{(n,m,i,j) \in \mathbb{Z}^4 /$
$\quad 0 \leq i \leq j \leq n\}$
$(n,m,i,j) \mapsto (i,j)),$

$(\mathtt{I}_3,$
$\{(n,m,i,j) \in \mathbb{Z}^4 /$
$\quad 0 \leq i \leq m \wedge j = n\}$
$(n,m,i,j) \mapsto (i,j))$

(a) Programme initial

```
for (i = 1; i < min(1+n, 1+m); i++)
  if (n ≤ i)
    for (j = i; j < i+1; j++)
      I1(i,j)
      I2(i,j)
      I3(i,j)
  if (i ≤ n-1)
    for (j = i; j < i+1; j++)
      I1(i,j)
      I2(i,j)
  for (j = i+1; j < n; j++)
    I2(i,j)
  if (i ≤ n-1)
    for (j = n; j < n+1; j++)
      I2(i,j)
      I3(i,j)
for (i = max(1, n+1); i < 1+m; i++)
  for (j = n; j < n+1; j++)
    I3(i,j)
for (i = max(1, m+1); i < 1+n; i++)
  for (j = i; j < i+1; j++)
    I1(i,j)
    I2(i,j)
  for (j = i+1; j < 1+n; j++)
    I2(i,j)
```

(b) Programme généré

FIGURE 9 – Génération de code pour plusieurs polyèdres, avec ordonnancement lexicographique dans le programme initial

D Théorèmes de préservation

Le premier théorème de préservation de la sémantique est celui pour l'élimination de l'ordonnancement :

```
Theorem poly_elim_schedule_semantics_env_preserve :
  forall d es env dim prog mem1 mem2,
    es = length env →
    (es ≤ dim)%nat →
    env_poly_lex_semantics env (dim + d) (elim_schedule d es prog) mem1 mem2 →
    (forall n pi, nth_error prog n = Some pi → (length pi.(pi_schedule) ≤ d)%nat) →
    env_poly_semantics env dim prog mem1 mem2.
```

On a ensuite celui qui donne la préservation de la sémantique par la génération de code PolyLoop :

```
Theorem generate_loop_many_preserves_sem :
  forall d n pis env mem1 mem2,
    (d ≤ n)%nat →
    WHEN st ← generate_loop_many d n pis THEN
    poly_loop_semantics st env mem1 mem2 →
    length env = (n − d)%nat →
    pis_have_dimension pis n →
    generate_invariant (n − d)%nat pis env →
    env_poly_lex_semantics (rev env) n pis mem1 mem2.
```

Il est suivi par la préservation de la sémantique de la simplification de code PolyLoop :

```
Lemma polyloop_simplify_correct :
  forall stmt n ctx,
    (poly_nrl ctx ≤ n)%nat →
    WHEN nst ← polyloop_simplify stmt n ctx THEN
    forall env mem1 mem2,
      length env = n → in_poly (rev env) ctx = true →
      poly_loop_semantics nst env mem1 mem2 →
      poly_loop_semantics stmt env mem1 mem2.
```

On a finalement le théorème justifiant la préservation de la sémantique par l'opération de génération de code Loop :

```
Lemma polyloop_to_loop_correct :
  forall pstmt n env mem1 mem2,
    WHEN st ← polyloop_to_loop n pstmt THEN
    loop_semantics st env mem1 mem2 →
    length env = n →
    poly_loop_semantics pstmt env mem1 mem2.
```

La composition de ces théorèmes permet de prouver le théorème de conservation global présenté à la fin de la section 7.