Verified Code Generation for the Polyhedral Model

NATHANAËL COURANT, Inria, France
XAVIER LEROY, Collège de France, PSL Research University, France

The polyhedral model is a high-level intermediate representation for loop nests that supports elegantly a great many loop optimizations. In a compiler, after polyhedral loop optimizations have been performed, it is necessary and difficult to regenerate sequential or parallel loop nests before continuing compilation. This paper reports on the formalization and proof of semantic preservation of such a code generator that produces sequential code from a polyhedral representation. The formalization and proofs are mechanized using the Coq proof assistant.

CCS Concepts: • **Software and its engineering** → **Compilers**; Context specific languages; **Software verification**; • **Theory of computation** → *Program verification.*

Additional Key Words and Phrases: Compiler verification, Polyhedral code generation, Polyhedral model

ACM Reference Format:
Nathanaël Courant and Xavier Leroy. 2021. Verified Code Generation for the Polyhedral Model. *Proc. ACM Program. Lang.* 1, POPL, Article 1 (January 2021), 24 pages.

1 INTRODUCTION

Numerical code often consists in nested loops operating over multidimensional arrays. Such *loop nests* are the target of many compiler optimizations. For example, memory locality can be improved by permuting two nested loops, fusing two consecutive loops, or tiling the iteration space [Muchnick 1997]. Likewise, parallelism can be increased by vectorization or loop scheduling.

The polyhedral model, also known as the polytope model, is a high-level, declarative intermediate representation for loop nests [Feautrier and Lengauer 2011]. In the polyhedral model, many loop optimizations can be expressed and performed in a uniform manner as transformations over the polytopes describing the iteration space. (Section 2 illustrates the approach on a simple example.)

Optimizers based on the polyhedral model have been integrated in compilers for conventional languages: for example, Graphite [Trifunović et al. 2010] adds polyhedral optimizations to GCC, and Polly [Grosser et al. 2012] to LLVM.

Another use for the polyhedral model is to synthesize efficient software or hardware implementations of matrix and tensor computations. Domain-specific languages such as Halide [Ragan-Kelley et al. 2017], Tensor Comprehensions [Vasilache et al. 2019] or VOBLA [Beaugnon et al. 2014] make it easy to write high-level specifications of such computations, which can, then, be automatically translated to polyhedral models and compiled to efficient low-level code.

This paper reports on the formal specification of a polyhedral model and the formal verification of one part of a loop optimizer based on the polyhedral model. Compiler verification applies program proof and other verification techniques to the compiler in order to rule miscompilation out: the generated code is guaranteed to execute as prescribed by the semantics of the source program.

Authors' addresses: Nathanaël Courant, Inria, 2 rue Simone Iff, Paris, France, nathanael.courant@inria.fr; Xavier Leroy, Collège de France, PSL Research University, 3 rue d'Ulm, Paris, France, xavier.leroy@college-de-france.fr.

2475-1421/2021/1-ART1
https://doi.org/

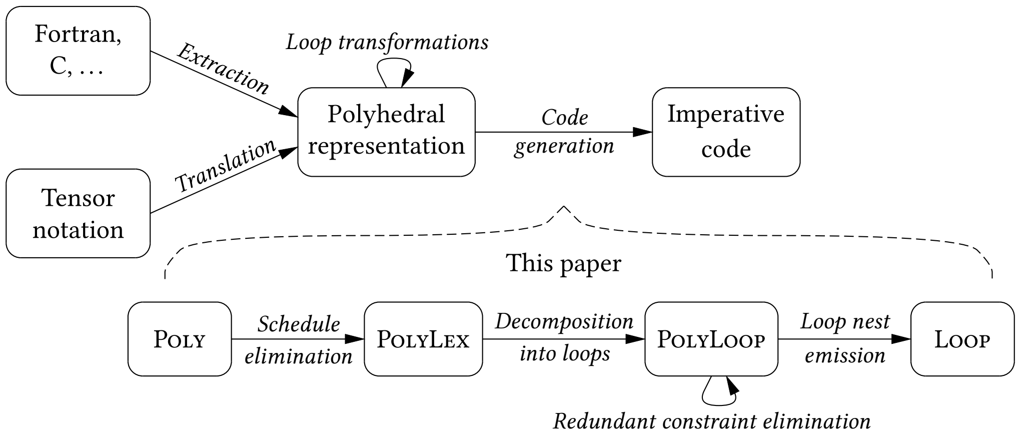


Fig. 1. General shape of a polyhedral optimizer (top) and contribution of this paper (bottom)

Recent examples of formally-verified compilers include CompCert [Leroy 2009] for the C language and CakeML [Kumar et al. 2014] for a functional language. A verified polyhedral optimizer could be integrated in CompCert, equipping this compiler with loop optimizations that it currently lacks. Such an optimizer could also be the basis for a verified code generator from a domain-specific language of tensor computations such as Halide. The formal verification would increase assurance in this kind of code synthesis.

Traditionally, a polyhedral optimizer comprises three parts, as depicted in Figure 1:

(1) detection of loop nests in Fortran or C source code, and construction of the corresponding polyhedral representations;
(2) optimization via transformations over the poyhedral representation;
(3) generation of an equivalent, sequential or parallelized, loop nest that can be given to a conventional compiler.

Preliminary investigations by Pilkiewicz [2013] suggest that part (2), that is, polyhedral optimizations proper, lends itself well to *translation validation* [Pnueli et al. 1998]: the polyhedral models before and after each program transformation are checked for semantic equivalence by generating formulas in Presburger arithmetic, which are then verified with the help of Farkas certificates.

In this paper, we focus on part (3) of a polyhedral optimizer. We describe the formal verification, using the Coq proof assistant, of a code generator that produces efficient sequential code from a polyhedral representation. This code generator follows the polyhedra scanning approach described by Bastoul [2004] and used in GCC. The verification is more difficult than that of purely polyhedral transformations, as it combines algebraic reasoning over polyhedral operations, using certified linear algebra, with CompCert-style semantic reasoning over the executions of the generated code.

The contributions of this paper are as follows. After a short tutorial on the polyhedral model (section 2), we formalize the syntax and semantics of Poly, a simple polyhedral language (section 3). We then describe a code generator for this language, along with its correctness arguments. As shown in Figure 1, the code generator comprises four passes: schedule elimination, replacing the input schedule by the identity schedule (section 4); decomposition into abstract loops expressed in the intermediate language PolyLoop (section 5); elimination of redundant constraints produced by the previous pass (section 6); and generation of concrete loops expressed in a simple sequential imperative language, Loop (section 7). We formalized the code generator, its source, intermediate

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

(a) Matrix multiplication

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

(b) Binomial coefficients computation

Fig. 2. Familiar numerical computations as loop nests

and target languages, and its correctness proofs using the Coq proof assistant and the VPL [Boulmé et al. 2018] verified library of polyhedral operations (section 8). The Coq development is available at https://github.com/Ekdohibs/PolyGen. Finally, we describe related work in more details (section 9), then discuss the next steps in developing this code generator further (section 10).

2 AN OVERVIEW OF THE POLYHEDRAL MODEL

Many numerical computing problems can be formulated as loop nests. Figure 2 shows two well-known examples:matrix multiplication and binomial coefficients computation using Pascal's triangle. More precisely, these are *imperfect* loop nests: each loop can contain a sequence of statements or other loops. These imperfect loop nests are thus described by the grammar:

$$s ::= \texttt{for (...) \{ } s \texttt{ \}} \mid s;s \mid \texttt{I}$$

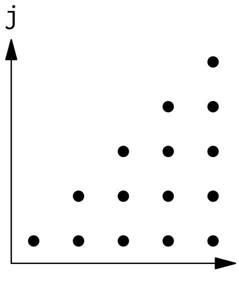
where I stands for base instructions such as assignments. Moreover, we impose the additional restriction that all loop bounds and array indices are *affine* in the existing variables: they are of the form a_1*x1 + ··· + a_n*xn + c, where x1, ..., xn are the existing variables. Even with these restrictions, a great many numerical computing problems can be expressed in this way.

These programs are amenable to many optimizations such as loop splitting, loop fusion, changing the order of two loops, or even more complicated transformations that completely change the iteration order. All these transformations can be understood in a common framework, the *polyhedral model*, which we now present. This framework is sufficiently general to handle both simple, classical optimizations and more advanced ones, making it possible to factor out the formalization effort.

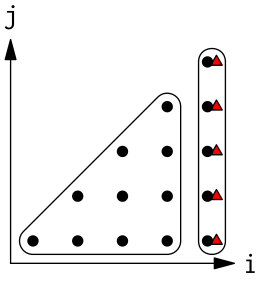
The polyhedral model works by considering each base instruction in an imperfect loop nest as an instruction that must be executed at every integer point from a given polyhedron. Moreover, the model specifies an order in which these instructions must be executed. For instance, in the binomial coefficients computation above, we have three base instructions, $\mathtt{I}_0$, $\mathtt{I}_1$ and $\mathtt{I}_2$, given by:

$$\begin{aligned}
\mathtt{I}_0(\mathtt{m}) &= \texttt{binom[m][0] = 1;}\\
\mathtt{I}_1(\mathtt{m}) &= \texttt{binom[m][m] = 1;}\\
\mathtt{I}_2(\mathtt{m},\mathtt{k}) &= \texttt{binom[m][k] = binom[m-1][k-1] + binom[m-1][k];}
\end{aligned}$$

These base instructions must be executed for every point of the polyhedron $\{\mathtt{m} \in \mathbb{Z} \mid 0 \le \mathtt{m} < \mathtt{n}\}$ for $\mathtt{I}_0$ and $\mathtt{I}_1$, and $\{(\mathtt{m},\mathtt{k}) \in \mathbb{Z}^2 \mid 1 \le \mathtt{k} < \mathtt{m} < \mathtt{n}\}$ for $\mathtt{I}_2$. However, these polyhedra alone are not enough to specify the behavior of this program: we also need to specify an execution order. In our case, it is given by executing the instructions in lexicographically increasing order of the $(\mathtt{m}, 0, 0)$ for $\mathtt{I}_0$, $(\mathtt{m}, 1, 0)$ for $\mathtt{I}_1$, and $(\mathtt{m}, 2, \mathtt{k})$ for $\mathtt{I}_2$. The constants $0, 1, 2$ are used to represent the order of



(a) One polyhedron



(b) Several polyhedra

Fig. 3. Example polyhedra for code generation

the instructions when using the ; operator. Such a specification of the execution order is called a *schedule*. The formal definition of this polyhedral language is presented in section 3.

Next, we wish to optimize the program in this polyhedral representation. This is done by changing the schedule, which corresponds to changing the iteration order. For instance, suppose we want to change our binomial program to perform initialization (instructions $\mathtt{I}_0$ and $\mathtt{I}_1$) before the rest of the code, and we also want to change the nested loops to iterate first over k and then over m. We can change our schedule to $(0, \mathtt{m}, 0)$ for $\mathtt{I}_0$, $(0, \mathtt{m}, 1)$ for $\mathtt{I}_1$, and $(1, \mathtt{k}, \mathtt{m})$ for $\mathtt{I}_2$. This transformation is allowed, because each time we get two instructions with a data dependency between one and the other, they are in the same order in the initial polyhedral program and the new polyhedral program. These data dependencies can be determined by the Bernstein conditions [Bernstein 1966], and as this step is heuristic, it is very amenable to translation validation, with a verifier checking that the new polyhedral program respects the data dependencies. At any rate, as data dependencies play no part in our generator, we will safely ignore these for the rest of the paper.

Last, we need to generate code that corresponds to this new schedule. The first step is schedule elimination: we prefix the schedule to the points, modifying the polyhedra to express that change. This replaces the explicit schedule, based on an explicit ordering on the different points, by an implicit schedule based on a lexicographic ordering on the points. In our case, the new polyhedra would be:

- $\{(x_1, x_2, x_3, \mathtt{m}) \in \mathbb{Z}^4 \mid x_1 = x_3 = 0 \land 0 \le x_2 = \mathtt{m} < \mathtt{n}\}$ for $\mathtt{I}_0$,
- $\{(x_1, x_2, x_3, \mathtt{m}) \in \mathbb{Z}^4 \mid x_1 = 0 \land x_3 = 1 \land 0 \le x_2 = \mathtt{m} < \mathtt{n}\}$ for $\mathtt{I}_1$,
- $\{(x_1, x_2, x_3, \mathtt{m}, \mathtt{k}) \in \mathbb{Z}^5 \mid x_1 = 1 \le x_2 = \mathtt{k} < x_3 = \mathtt{m} < \mathtt{n}\}$ for $\mathtt{I}_2$.

Once this is done, we generate code that scans the given polyhedra in lexicographic order. The case of only one polyhedron is simple: we generate one loop per dimension, computing the projection on the current dimensions, and setting the bounds of the loop according to the projection. For instance, to generate loops corresponding to the black points in Figure 3a (in lexicographic order of i then j), we first generate an external loop `for (int i = 0; i < 5; i++)`, since the projection of the polyhedron on the first variable i gives $\{\mathtt{i} \mid 0 \le \mathtt{i} < 5\}$. Then, inside that loop, we generate the loop `for (int j = 0; j <= i; j++)`, since the projection on the plane generated by both i and j is equal to the whole polyhedron, $\{(\mathtt{i}, \mathtt{j}) \mid 0 \le \mathtt{j} \le \mathtt{i} < 5\}$. Finally, that second loop contains the instruction `I(i, j)`.

The case of several polyhedra is more involved, as illustrated in Figure 3b. The naïve way of generating code in that case is to generate scanning code as outlined above for the union of the polyhedra, and, in the body of the generated loops, add a test to ensure that we are inside the polyhedron corresponding to the instruction. For instance, associating black points with instruction $\mathtt{I}_1$ and red triangles with instruction $\mathtt{I}_2$, the body of the loop nest executes

```
I₁(i, j); if (i == 4) I₂(i, j); .
```

However, we can produce more efficient code that does not contain tests that are useless for the most part of the loop. To this end, we project each polyhedron independently and then separate the projections into unions of polyhedra so that, in each of these polyhedra, the projection of each input polyhedron either contains that polyhedron or is disjoint from it. In our case, we split the projections in two different parts, $\{\mathtt{i} \mid 0 \le \mathtt{i} < 4\}$ and $\{\mathtt{i} \mid \mathtt{i} = 4\}$. As previously, we repeat the generating operation under each loop node. In the end, we obtain the following code:

```
for (int i = 0; i < 4; i++) {
    for (int j = 0; j <= i; j++) {
        I₁(i, j);
    }
}
for (int j = 0; j < 4; j++) {
    I₁(4, j);
    I₂(4, j);
}
```

3 THE SOURCE LANGUAGE

The source language for our code generator is a polyhedral notation for loop nests. It is parameterized by a language of base instructions I, typically assignments such as

$$\mathtt{A}[x_1][x_3] \mathrel{+}= \mathtt{B}[x_1][x_2] * \mathtt{C}[x_2][x_3].$$

As this example shows, base instructions are parameterized by a vector $x \in \mathbb{Z}^n$ of integer indices. The semantics of base instructions is given in operational style, as a relation $(\mathtt{I}(x), \mathcal{M}_1) \Downarrow_{\mathtt{I}} \mathcal{M}_2$ between the memory states before ($\mathcal{M}_1$) and after ($\mathcal{M}_2$) the execution of I with indices x. Since we do not need to perform dependence analysis, we keep instructions, their semantics, and memory states completely abstract.

A polyhedral program is a multiset of polyhedral instructions, each being a quadruplet $(\mathtt{I}, \mathcal{D}, \theta, \mathcal{T})$. Such a polyhedral instruction describes the iteration of a base instruction I over a sequence of parameters x. More precisely, the four components of a polyhedral instruction are:

- I is the base instruction to iterate.
- $\mathcal{D}$ is the iteration domain. It is a polyhedron: the set of integer tuples that satisfy a set of linear inequalities. It is thus described by a matrix A and a vector a, with

$$\mathcal{D} = \{x \in \mathbb{Z}^n \mid Ax \le a\}.$$

- θ is the scheduling function. It is an affine function[1] from elements of $\mathcal{D}$ to tuples of integers that act as time stamps. It is described by a matrix K and a vector k:

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

[1]Some polyhedral code generators, such as [Ancourt and Irigoin 1991], require that the scheduling function be *unimodular* so that points in the polyhedron are always integer affine functions of the schedule. Our generator accepts arbitrary affine scheduling functions. Non-unimodular scheduling functions such as $i \mapsto 2i$ will produce correct but possibly inefficient code involving tests such as `if (j mod 2 = 0)`.

- $\mathcal{T}$ is the transformation function that produces the arguments given to the base instruction $\mathtt{I}$. Like θ, it is an affine function, described by a matrix T and a vector t:

$$\mathcal{T} : x \mapsto Tx + t.$$

Most polyhedral models in the literature have no transformation function ($\mathcal{T}$ is the identity). We find these functions useful to express simplifications and transformations without rewriting the instructions $\mathtt{I}$ themselves. For example, assume that, during tiling, index i becomes $5i_1 + i_2$. Traditionally, instructions such as $A[i] = 0$ are rewritten into $A[5i_1 + i_2] = 0$. We prefer to treat instructions as an opaque type, without any substitution operation, and use the transformation function to record the substitution explicitly:

$$\begin{pmatrix} i \end{pmatrix} = T \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} + t \quad \text{where } T = \begin{pmatrix} 5 & 1 \end{pmatrix} \text{ and } t = \begin{pmatrix} 0 \end{pmatrix}.$$

The standard formulation of the polyhedral model, without transformation functions, can be recovered by considering each base instruction to be the composition of $\mathtt{I}$ and $\mathcal{T}$.

A simplified semantics for a single polyhedral instruction $(\mathtt{I}, \mathcal{D}, \theta, \mathcal{T})$ is as follows. First, enumerate the elements of $\mathcal{D}$ in *lexicographic nondecreasing* order of the time stamps $\theta(x)$:

$$\mathcal{D} = \{x_1, \dots, x_n\} \text{ with } \theta(x_i) \preccurlyeq \theta(x_j) \text{ if } i \leq j$$

Then, repeatedly execute instruction $\mathtt{I}$ with parameters $\mathcal{T}(x_i)$, going from state $\mathcal{M}_0$ to state $\mathcal{M}_n$:

$$(\mathtt{I}(\mathcal{T}(x_1)), \mathcal{M}_0) \Downarrow_{\mathtt{I}} \mathcal{M}_1 \quad \dots \quad (\mathtt{I}(\mathcal{T}(x_n)), \mathcal{M}_{n-1}) \Downarrow_{\mathtt{I}} \mathcal{M}_n$$

Note that the semantics is non-deterministic, even if the base instruction I has deterministic semantics: if $\theta(x) = \theta(x')$, then $\mathtt{I}(\mathcal{T}(x))$ and $\mathtt{I}(\mathcal{T}(x'))$ can be executed in any order.

The full semantics for a polyhedral program $\mathcal{P}$ is similar, with two extensions. First, it is useful to parameterize executions of polyhedral programs over some quantities, typically matrix dimensions, that remain constant during execution. These constant quantities, usually named program parameters, are given as an environment $\mathcal{E}$: a vector of length k that fixes the values of the first k dimensions of the iteration space. Hence, the effective domain for the execution of $(\mathtt{I}, \mathcal{D}, \theta, \mathcal{T})$ is

$$\mathcal{D}_{\mathcal{E}} = \left\{ \begin{pmatrix} \mathcal{E} \\ y \end{pmatrix} \middle| \, y \in \mathbb{Z}^{n-k}, \begin{pmatrix} \mathcal{E} \\ y \end{pmatrix} \in \mathcal{D} \right\}$$

Second extension: a polyhedral program comprises several polyhedral instructions. These instructions do not need to be executed one after the other; instead, their iterations can be interleaved if the schedules allow. To formalize this idea, we need to assume that in a polyhedral program $\mathcal{P} = (\mathtt{I}_k, \mathcal{D}_k, \theta_k, \mathcal{T}_k)_{k \in K}$ all the domains $\mathcal{D}_k$ have the same dimension, and all the images of the scheduling functions $\mathrm{Im}(\theta_k)$ have the same dimension. This can easily be ensured by adding extra dimensions with value 0.

The semantics of the polyhedral program $\mathcal{P}$ in environment $\mathcal{E}$ is, then, defined as follows: iterate over $(\mathtt{I}, \mathcal{D}, \theta, \mathcal{T}) \in \mathcal{P}$ and over $x \in \mathcal{D}_{\mathcal{E}}$, in lexicographic nondecreasing order of $\theta(x)$, and execute the instructions $\mathtt{I}(\mathcal{T}(x))$ sequentially in this order.

An example of a polyhedral program, and its possible execution orders are given in Figure 4.

4 ELIMINATING THE SCHEDULING FUNCTION

The scheduling function in polyhedral programs make it easy to express optimizations that change the order of loops, such as loop interchange. However, polyhedral programs that have identity scheduling functions are in close correspondence with loop nests: those programs execute by enumerating tuples of variables in lexicographic order, hence the first variable naturally corresponds to outermost loops, the second variable to the second-outer loops, and so on. This correspondence

$$\begin{aligned}&(\mathtt{I}_1, \{(n,x,y) \mid 0 \le x \le y \le n\},\\ &\qquad (n,x,y) \mapsto (x,x), (n,x,y) \mapsto (x,y)),\\ &(\mathtt{I}_2, \{(n,x,y) \mid 0 \le y \le x \le n\},\\ &\qquad (n,x,y) \mapsto (x+y-1,1), (n,x,y) \mapsto (x))\end{aligned}$$

(a) Polyhedral program to be executed

$$\begin{array}{ll} \mathtt{I}_1(0,0) \rightsquigarrow (0,0) & \mathtt{I}_2(0) \rightsquigarrow (-1,1) \\ \mathtt{I}_1(0,1) \rightsquigarrow (0,0) & \mathtt{I}_2(1) \rightsquigarrow (0,1) \\ \mathtt{I}_1(1,1) \rightsquigarrow (1,1) & \mathtt{I}_2(1) \rightsquigarrow (1,1) \end{array}$$

(b) Instructions to be executed, and associated timestamps for the environment ($n = 1$)

$$\begin{array}{l} \mathtt{I}_2(0); \mathtt{I}_1(0,0); \mathtt{I}_1(0,1); \mathtt{I}_2(1); \mathtt{I}_1(1,1); \mathtt{I}_2(1) \\ \mathtt{I}_2(0); \mathtt{I}_1(0,0); \mathtt{I}_1(0,1); \mathtt{I}_2(1); \mathtt{I}_2(1); \mathtt{I}_1(1,1) \\ \mathtt{I}_2(0); \mathtt{I}_1(0,1); \mathtt{I}_1(0,0); \mathtt{I}_2(1); \mathtt{I}_1(1,1); \mathtt{I}_2(1) \\ \mathtt{I}_2(0); \mathtt{I}_1(0,1); \mathtt{I}_1(0,0); \mathtt{I}_2(1); \mathtt{I}_2(1); \mathtt{I}_1(1,1) \end{array}$$

(c) Possible execution orders for the environment ($n = 1$)

Fig. 4. Example of a semantics for a polyhedral program

holds naturally when we transform a loop nest into a polyhedral program: we obtain a lexicographic scheduling on the variables we used for iteration. Symmetrically, the first natural step of code generation from a polyhedral program to a loop nest is to remove the explicit schedule and replace it with an implicit schedule, which is lexicographic on the iteration variables.

It must be noted that such a transformation cannot preserve *all* possible executions of a polyhedral program. Indeed, a general polyhedral program can have executions of the same instruction in several different points, all corresponding to the same timestamp, thus leaving the execution order of these instructions unspecified. (Some polyhedral generators use this to express that the loop can be run in parallel, but we only consider sequential programs.) On the other hand, this is not possible for a lexicographic schedule, since the lexicographic order is a total order, therefore two different executions of a given instruction are necessarily ordered. This transformation, therefore, does a partial determinization. Consequently, the correctness theorem for this transformation is that every valid execution of the transformed program is a valid execution of the initial program. On the other hand, some nondeterminism may remain after transformation: there is no constraint on the execution order of two *distinct* instructions executed at the same point.

The schedule elimination method presented in this section is the one introduced by Bastoul [2004]. It has numerous advantages for verification: it does not use complex operations over matrices, and it works for all inputs, not just unimodular schedules. Its only disadvantage is that it increases the number of variables that must be considered later during code generation, making code generation slower.

The transformation is simple: it adds new variables as a prefix of the list of variables, one for each dimension of the schedule, and sets the new variables to be equal to the value specified by the scheduling function at the given point in the polyhedral constraints. Since the semantics of polyhedral programs is specified by setting an environment as a prefix of the list of variables, we need to know the size of the environments for the transformation, and keep the environments as a prefix of the list of variables, inserting our new variables after the environments. It is an instance of the transformation usually called *change of basis* in the polyhedral literature.

Thus, assuming that the dimensions of the image of all scheduling functions θ are all the same, we perform the following transformation, which for each polyhedral instruction $(\mathtt{I}, \mathcal{D}, \theta, \mathcal{T})$ of $\mathcal{P}$

generates $(\mathrm{I}, \mathcal{D}', \theta', \mathcal{T}')$, where:

$$\mathcal{D} = \left\{ \begin{pmatrix} \mathcal{E} \\ y \end{pmatrix} \middle| \begin{pmatrix} \mathcal{E} \\ y \end{pmatrix} \in \mathbb{Z}^n, \begin{pmatrix} A_e & A_v \end{pmatrix} \begin{pmatrix} \mathcal{E} \\ y \end{pmatrix} \leq a \right\},$$
$$\theta : \begin{pmatrix} \mathcal{E} \\ y \end{pmatrix} \mapsto \begin{pmatrix} K_e & K_v \end{pmatrix} \begin{pmatrix} \mathcal{E} \\ y \end{pmatrix} + k,$$
$$\mathcal{T} : \begin{pmatrix} \mathcal{E} \\ y \end{pmatrix} \mapsto \begin{pmatrix} T_e & T_v \end{pmatrix} \begin{pmatrix} \mathcal{E} \\ y \end{pmatrix} + t$$

and:

$$\mathcal{D}' = \left\{ \begin{pmatrix} \mathcal{E} \\ u \\ y \end{pmatrix} \middle| \begin{pmatrix} \mathcal{E} \\ u \\ y \end{pmatrix} \in \mathbb{Z}^{n'}, \left(\begin{array}{ccc} A_e & 0 & A_v \\ \hline -K_e & Id & -K_v \end{array}\right) \begin{pmatrix} \mathcal{E} \\ u \\ y \end{pmatrix} \begin{array}{c} \leq \\ \hline = \end{array} \begin{pmatrix} a \\ k \end{pmatrix} \right\},$$
$$\theta' : \begin{pmatrix} \mathcal{E} \\ u \\ y \end{pmatrix} \mapsto \begin{pmatrix} \mathcal{E} \\ u \\ y \end{pmatrix},$$
$$\mathcal{T}' : \begin{pmatrix} \mathcal{E} \\ u \\ y \end{pmatrix} \mapsto \begin{pmatrix} T_e & 0 & T_v \end{pmatrix} \begin{pmatrix} \mathcal{E} \\ u \\ y \end{pmatrix} + t$$

Indeed, the definition of $\mathcal{D}'$ gives:

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

Then, the function $\Gamma : \mathcal{D} \rightarrow \mathcal{D}'$ defined by:

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

is a bijection from $\mathcal{D}$ to $\mathcal{D}'$, with inverse $\begin{pmatrix} \mathcal{E} \\ u \\ y \end{pmatrix} \mapsto \begin{pmatrix} \mathcal{E} \\ y \end{pmatrix}$.

The new transformation function $\mathcal{T}'$ has moreover been chosen so that for all $x \in \mathcal{D}$, $\mathcal{T}(x) = \mathcal{T}'(\Gamma(x))$.

Besides, if $\begin{pmatrix} \mathcal{E} \\ u_1 \\ y_1 \end{pmatrix} \in \mathcal{D}'_1$ and $\begin{pmatrix} \mathcal{E} \\ u_2 \\ y_2 \end{pmatrix} \in \mathcal{D}'_2$, then we have (where $\prec$ is lexicographic order):

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

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

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

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

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

Fig. 5. Big-step semantics for PolyLoop

which can be rephrased as: if $x_1 \in \mathcal{D}_1$ and $x_2 \in \mathcal{D}_2$ correspond to the same environment $\mathcal{E}$, we have:

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

that is, by contraposition:

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

As a consequence, for a fixed environment $\mathcal{E}$, every valid ordering of instructions during the execution of $\mathcal{P}'$ is a valid ordering of instructions during the execution of $\mathcal{P}$. The $\mathcal{T} = \mathcal{T}' \circ \Gamma$ property guarantees that the same instructions are executed. Thus, every execution of $\mathcal{P}'$ corresponds to an execution of $\mathcal{P}$, and this transformation preserves semantics. We also reach our goal: the θ' are now the identity function, and the schedule is thus a lexicographic enumeration of the points of the domains $\mathcal{D}'$ themselves.

5 DECOMPOSITION INTO LOOP NESTS

The second step of code generation is to compute the structure of the program in terms of imperfectly nested loops. The control flow becomes explicit, but the conditions and loop bounds remain affine and are represented by polyhedra. The target of this generation step is an intermediate language called PolyLoop. The syntax of PolyLoop is as follows:

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

$$\text{Polyhedra:} \quad \mathcal{P} ::= \{(x_1, \dots, x_k) \in \mathbb{Z}^k \mid A(x_1 \dots x_k)^\top \le a\} \qquad \text{where } A \in \mathcal{M}_{p,k}(\mathbb{Z}), a \in \mathbb{Z}^p$$

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

Expressions e compute linear functions of their arguments, up to the floor function. Polyhedra $\mathcal{P}$ are defined as in Poly by a matrix A, a vector a, and the constraint $Ax \le a$. Statements comprise base instructions, sequences, conditionals and loops. The conditional construct guard $\mathcal{P}$ s executes s if the current environment, mapping variables to their current values, is in the polyhedron $\mathcal{P}$.

Input: a list of lexicographically scheduled polyhedral instructions $(\mathtt{I}_1, \mathcal{D}_1, \mathcal{T}_1), \ldots, (\mathtt{I}_n, \mathcal{D}_n, \mathcal{T}_n)$, and the current dimension d.
Output: a PolyLoop program.

(1) If d is the dimension of the $\mathcal{D}_i$, return the program:

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

(2) For each polyhedron $\mathcal{D}_i$, compute its projection $\mathcal{P}_i$ on the d first dimensions, and compute the list of the $\mathcal{P}_i \to (\mathtt{I}_i, \mathcal{D}_i, \mathcal{T}_i)$.
(3) Separate the projections $\mathcal{P}_i$ into a list of disjoint polyhedra and their associated instructions: from $\mathcal{P}_1 \to \mathcal{I}_1$ and $\mathcal{P}_2 \to \mathcal{I}_2$, we get $(\mathcal{P}_1 \cap \mathcal{P}_2) \to (\mathcal{I}_1; \mathcal{I}_2)$, $(\mathcal{P}_1 \setminus \mathcal{P}_2) \to \mathcal{I}_1$, and $(\mathcal{P}_2 \setminus \mathcal{P}_1) \to \mathcal{I}_2$, and then repeat with the other polyhedra that need to be separated. The differences $\mathcal{P}_1 \setminus \mathcal{P}_2$ and $\mathcal{P}_2 \setminus \mathcal{P}_1$ are not polyhedra, but unions of polyhedra: the list we produce is then potentially very long.
(4) Compute the lexicographic ordering graph, which contains an edge $\mathcal{P} \to \mathcal{P}'$ if the loop over $\mathcal{P}$ must be placed before the loop over $\mathcal{P}'$ in the result to respect the lexicographic schedule, then sort the graph topologically. If the graph contains a cycle, the code generation fails.
(5) For each $\mathcal{P} \to ((\mathtt{I}_1, \mathcal{D}_1, \mathcal{T}_1); \ldots ; (\mathtt{I}_k, \mathcal{D}_k, \mathcal{T}_k))$, let s be the result of a recursive call with current dimension $d + 1$ and instruction list $(\mathtt{I}_1, \mathcal{D}_1 \cap \mathcal{P}, \mathcal{T}_1), \ldots, (\mathtt{I}_k, \mathcal{D}_k \cap \mathcal{P}, \mathcal{T}_k)$; build the instruction $\mathtt{loop}\ \mathcal{P}\ s$.
(6) Return the program formed by the sequence of the results from the previous step.

Fig. 6. Transformation of a polyhedral program into a PolyLoop program

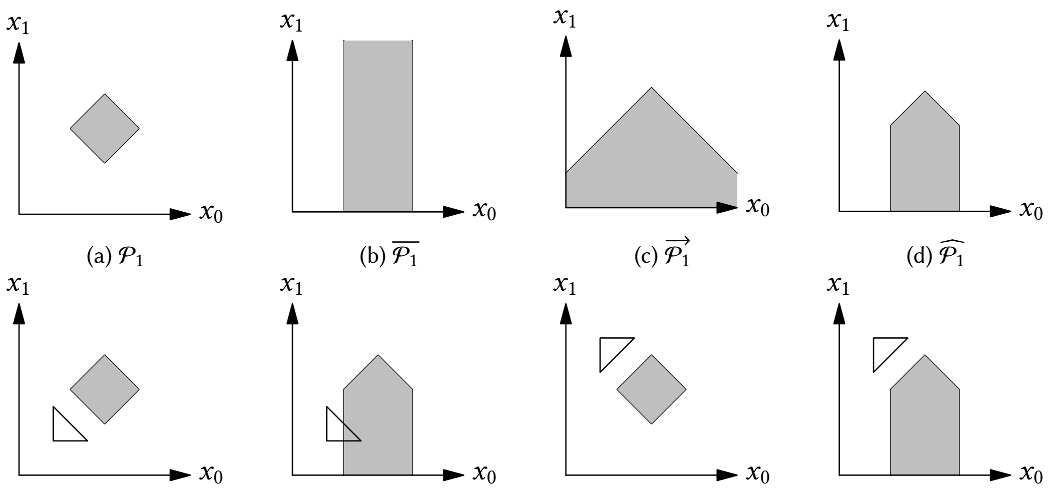
The iteration construct, $\mathtt{loop}\ \mathcal{P}\ s$ executes s for increasing values of x such that the current environment with x added is in $\mathcal{P}$. The semantics of PolyLoop are detailed in Figure 5. In rule Loop, the precondition $\forall x, (\mathcal{E} :: x \in \mathcal{P} \Leftrightarrow a \le x < b)$ ensures that the iteration domain is finite: the rule does not apply if x has no upper or no lower bound implied by $\mathcal{P}$.

There are some similarities between PolyLoop and the schedule trees of Verdoolaege et al. [2014] and Grosser et al. [2015], but their objectives are different: PolyLoop is a simple intermediate representation, while schedule trees are more complex, as they are used as an intermediate representation throughout code generation and subject to many transformations.

The algorithm used to transform a polyhedral program with lexicographic schedule into a PolyLoop program is shown in Figure 6. It is based on the algorithm by Quilleré et al. [2000]. An important difference is that there is no context in which we generate the syntax tree, and that we do not simplify loop bounds in this context. Instead, this simplification is done in a separate phase (described in section 6) that removes redundant constraints from a PolyLoop program without changing its semantics.

The algorithm presented in Figure 6 generates a syntax tree corresponding to a given polyhedral program. It proceeds dimension by dimension, starting with the outermost dimension that must be generated: then, the recursive calls treat the iteration variable that has just been generated as an additional parameter.

The first step of the algorithm considers the case where there is no variable left for which there is code to generate. We then have to generate an instruction for each polyhedral instruction, with a `guard` node to ensure that the instruction will only be executed if it is inside an environment where it must be executed. These guards are redundant most of the time, and most of them will be removed by the simplification pass described in section 6. We note an arbitrary scheduling decision here: instructions are executed in the input order of the algorithm.



(a) $\mathcal{P}_1$ (b) $\overline{\mathcal{P}_1}$ (c) $\overrightarrow{\mathcal{P}_1}$ (d) $\widehat{\mathcal{P}_1}$

(e) $\mathcal{P}_1$ and $\mathcal{P}_2$ (f) $\mathcal{P}_1$ cannot be ordered before $\mathcal{P}_2$ (g) $\mathcal{P}_1$ and $\mathcal{P}_2$ (h) $\mathcal{P}_1$ can be ordered before $\mathcal{P}_2$

Fig. 7. Ordering two disjoint polyhedra with respect to execution order

The rest of the algorithm considers the case where we add another dimension. We first compute the projections over the current dimensions and the new dimension, and we make a case disjunction (steps 2 and 3) depending on which input polyhedra the current environment is in. Thus, each instruction will only be in the internal loops for which is it necessary for it to appear.

Step 4 sorts the resulting list of polyhedra. Indeed, in the PolyLoop program, the loops that will be generated need to be ordered; however, this means that in every environment, one of these loops will always be executed before the other. Quilleré et al. [2000] show that it is always possible to order two disjoint polyhedra to respect the desired execution order. The property "$\mathcal{P}_1$ can be ordered before $\mathcal{P}_2$" is expressed as:

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

This is the same as saying that in every environment, all iterations of $\mathcal{P}_1$ will be executed before all iterations of $\mathcal{P}_2$. This can be determined by the criterion depicted in Figure 7. Define $\overline{\mathcal{P}_1} = \{\mathcal{E} :: x_1 \mid \exists x_2, \mathcal{E} :: x_2 \in \mathcal{P}_1\}$, that is, the projection of $\mathcal{P}_1$ with respect to the variable x_1, seen as a polyhedron having the same dimension as $\mathcal{P}_1$. Further define $\overrightarrow{\mathcal{P}_1}$ as the polyhedron obtained by only keeping the constraints of $\mathcal{P}_1$ for which the coefficient of x_1 is positive, and $\widehat{\mathcal{P}_1} = \overline{\mathcal{P}_1} \cap \overrightarrow{\mathcal{P}_1}$. We then have that $\mathcal{P}_2$ must be ordered before $\mathcal{P}_1$ if and only if:

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

We then have, as desired:

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

However, in the case of three or more polyhedra, it can happen that there exist no compatible order because the lexicographic ordering graph contains a cycle. An example involving three polyhedra is given in appendix A. In this case, our algorithm fails, aborting code generation. A more sophisticated algorithm could, instead, divide some of the polyhedra in several fragments that can be ordered independently.

Steps 5 and 6 make a recursive call on each polyhedron for which there is a loop to generate, compute the corresponding PolyLoop subprograms, and combine them in sequence. The result is a PolyLoop program whose semantics respect the semantics of the initial polyhedral program.

The correctness of this code generation algorithm is proved by showing, by induction on the number of remaining dimensions, that in every environment, every execution of the generated program corresponds to an execution of the initial polyhedral program. The base case is simple. To handle the case where we generate the code for one more dimension, we have to show the following results on the polyhedra obtained as result of step 4:

- If $\mathcal{P}$ is before $\mathcal{P}'$ in the list of obtained polyhedra, then $\mathcal{P}$ can be ordered before $\mathcal{P}'$ in every environment;
- All polyhedra obtained are disjoint;
- If $x \in \mathcal{P}_i$ at the step 2, then x is in one of the $\mathcal{P}$ of the obtained list, and the corresponding instruction $\mathcal{I}_i$ appears in the list of instructions of $\mathcal{P}$.

These results are a consequence of the polyhedral computations performed at steps 3 and 4 of the algorithm.

As previously mentioned, the base case of our algorithm (step 1 in Figure 6) systematically guards the execution of instructions by a dynamic check that the current indices are within the corresponding domains $\mathcal{D}_i$. Besides simplifying the correctness proof, these checks enable the recursive case (steps 2 to 5) to safely over-approximate their intermediate results, namely the projections $\mathcal{P}_i$ and their further decomposition into unions of polyhedra $\mathcal{P}$. (Excessive over-approximation can trigger failures in step 4 or in the code generation algorithm from section 7, however.) Taking advantage of this degree of freedom, we use Fourier-Motzkin elimination [Schrijver 1998, section 12.2] to compute the projections $\mathcal{P}_i$ at step 2. Fourier-Motzkin elimination produces an exact result in the rationals ($\mathbb{Q}$) but an over-approximation in the integers ($\mathbb{Z}$). Hence, the polyhedra $\mathcal{P}_i$ may be bigger than the exact projections in $\mathbb{Z}$, but this does not endanger the correctness of the generated PolyLoop code.

6 SIMPLIFICATION OF REDUNDANT CONSTRAINTS

To keep its correctness proof simple, the algorithm of section 5 generates a large number of redundant constraints. For instance, it inserts a `guard` node before each instruction, which is most of the time unnecessary as it is a direct consequence of the surrounding `loop` nodes. However, the algorithm is written so that is it easy to remove most constraints in a separate pass, without having to care whether it is possible to remove them while generating the PolyLoop code: whenever one of these constraints could not be removed, it will simply be kept, and the generated code will be slightly less efficient.

To simplify redundant constraints, we need a single primitive `simplify`, which takes as input a polyhedron $\mathcal{P}$ and a context C, and returns a polyhedron $\mathcal{P}'$ verifying $\mathcal{P}' \cap C = \mathcal{P} \cap C$, which can be reformulated as $\forall x \in C, x \in \mathcal{P}' \Leftrightarrow x \in \mathcal{P}$. The objective is that `simplify` returns a polyhedron described by as few constraints as possible, since these are the constraints that will appear in the generated code.

The algorithm for computing $\mathcal{P}' = \texttt{simplify}(\mathcal{P}, C)$ is simple. Starting with $\mathcal{P}' = \emptyset$, we consider each constraint $ax \leq c$ in $\mathcal{P}$ one after the other. If the constraint is implied by $\mathcal{P} \cap C$, that is, if $\mathcal{P} \cap C \cap \neg(ax \leq c) = \emptyset$, it is removed; otherwise, the constraint is added to $\mathcal{P}'$.

$$
\begin{aligned}
\texttt{psimpl}(\texttt{skip}, C) &= \texttt{skip} \\
\texttt{psimpl}(s_1; s_2, C) &= \texttt{psimpl}(s_1, C); \texttt{psimpl}(s_2, C) \\
\texttt{psimpl}(\texttt{I}(e_1, \dots, e_k), C) &= \texttt{I}(e_1, \dots, e_k) \\
\texttt{psimpl}(\texttt{guard}\, \mathcal{P}\ s, C) &= \texttt{guard}\ (\texttt{simplify}(\mathcal{P}, C))\ \texttt{psimpl}(s, \mathcal{P} \cap C) \\
\texttt{psimpl}(\texttt{loop}\, \mathcal{P}\ s, C) &= \texttt{loop}\ (\texttt{simplify}(\mathcal{P}, C))\ \texttt{psimpl}(s, \mathcal{P} \cap C)
\end{aligned}
$$

Fig. 8. Simplification of a PolyLoop program

Expressions:	Tests:	Statements:
$e ::= \mid c \mid x$	$t ::= \mid e_1 = e_2$	$s ::= \mid \texttt{skip} \mid s_1 ; s_2$
$\mid e_1 + e_2$	$\mid e_1 \leq e_2$	$\mid \texttt{I}(e_1, \dots, e_k)$
$\mid c \cdot e$	$\mid t_1 \,\&\&\, t_2$	$\mid \texttt{if}\ (t)\ s$
$\mid \lfloor e/c \rfloor \mid e \ \texttt{mod}\ c$	$\mid\, !t$	$\mid \texttt{for}\ (i = e_1;\ i < e_2;\ i\texttt{++})\ s$
$\mid \texttt{min}(e_1, e_2) \mid \texttt{max}(e_1, e_2)$	$\mid \texttt{true} \mid \texttt{false}$	

Fig. 9. Syntax of Loop

As implemented and verified in the Coq development, the `simplify` operation is sound but suboptimal, owing to the fact that our emptiness test for polyhedra uses rationals ($\mathbb{Q}$) instead of integers ($\mathbb{Z}$): it guarantees that a set of constraints has no solutions with rational coordinates, while we are interested in solutions with integer coordinates. Consequently, some constraints that are redundant in $\mathbb{Z}$ are kept because they are not redundant in $\mathbb{Q}$. For example, in a context $x \leq 1$, the polyhedron $x = 2y \land x \leq 0$ is not simplified currently, but could be simplified to $x = 2y$ since $x \leq 0$ is redundant for integer points. This is suboptimal but sound.

We then define a function `psimpl` by the rules given in Figure 8. In effect, the constraints carried by `guard` and `loop` constructs are collected in the current context C, while being simplified using the `simplify` function. In these rules, we have omitted a slight detail: the dimension of the context needs to increase when we get under a `loop` node, to account for the newly declared variable. That detail aside, we can easily prove the theorem which interests us: if $\mathcal{E} \in C$, then $\mathcal{E} \vdash (\texttt{psimpl}(C, s), \mathcal{M}_1) \Downarrow \mathcal{M}_2$ is derivable if and only if $\mathcal{E} \vdash (s, \mathcal{M}_1) \Downarrow \mathcal{M}_2$ is.

7 LOOP CODE GENERATION

The last step of code generation is to translate the PolyLoop intermediate language to a lower-level language, Loop, which replaces the abstract polyhedral constructions of PolyLoop by concrete Boolean tests and “for” loops. Figure 9 defines the syntax of Loop, and Figure 10 its semantics. The language is structured in integer expressions, tests (Boolean expressions), and statements. Expressions are semi-linear, as they include multiplications and divisions by constants in addition to affine constructs. They also feature `min`, `max` and `mod` operators in order to express loop bounds and guard conditions. Statements correspond to a standard structured imperative language, with `if` conditionals and counted `for` loops. A “let” binding can be defined as syntactic sugar for a loop with one iteration:

$$\texttt{let}\ x = e\ \texttt{in}\ s \overset{\text{def}}{=} \texttt{for}\ (x = e;\ x < 1 + e;\ x\texttt{++})\ s$$

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

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

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

$$\textsc{ForEmpty}\quad \frac{\mathtt{eval}(\mathcal{E}, e_1) \geq \mathtt{eval}(\mathcal{E}, e_2)}{\mathcal{E} \vdash (\mathtt{for}\ \ (i\ =\ e_1;\ i\ <\ e_2;\ i\mathtt{++})\ \ s, \mathcal{M}) \Downarrow \mathcal{M}}$$

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

Fig. 10. Semantics for Loop

$$\begin{aligned}
\mathtt{tr}(\mathtt{skip}) &= \mathtt{skip}\\
\mathtt{tr}(s_1; s_2) &= \mathtt{tr}(s_1)\,;\mathtt{tr}(s_2)\\
\mathtt{tr}(\mathtt{guard}\ \mathcal{P}\ s) &= \mathtt{if}\ \ (\mathtt{inside}(\mathcal{P}))\ \ \mathtt{tr}(s)\\
\mathtt{tr}(\mathtt{loop}\ \mathcal{P}\ s) &= \mathtt{if}\ \ (\mathtt{inside}(\mathcal{P}_0))\ \ \mathtt{for}\ \ (i\ =\ e_-;\ i\ <\ e_+;\ i\mathtt{++})\ \ \mathtt{tr}(s) \quad \text{(general case)}\\
\mathtt{tr}(\mathtt{loop}\ \mathcal{P}\ s) &= \mathtt{if}\ \ (\mathtt{inside}(\mathcal{P}'))\ \ \mathtt{let}\ \ i\ =\ e\ \ \mathtt{in}\ \ \mathtt{tr}(s) \quad \text{(if there is an equation } i = e\text{)}\\
\mathtt{tr}(\mathtt{loop}\ \mathcal{P}\ s) &= \mathtt{if}\ \ (\mathtt{inside}(\mathcal{P}')\ \ \mathtt{\&\&}\ \ e\ \ \mathtt{mod}\ \ k = 0)\ \ \mathtt{let}\ \ i\ =\ \lfloor e/k \rfloor\ \ \mathtt{in}\ \ \mathtt{tr}(s)\\
&\qquad \text{(if there is an equation } ki = e\text{)}
\end{aligned}$$

Fig. 11. Outline of the translation from PolyLoop to Loop. See the main text for auxiliary definitions.

The transformation of a PolyLoop program into a Loop program is outlined in Figure 11. It is straightforward except for the `guard` and `loop` constructs.

A `guard` $\mathcal{P}$ s construct is translated to a statement `if (inside(`$\mathcal{P}$`))` `tr(`s`)`, where `tr(`s`)` is the translation of s and `inside(`$\mathcal{P}$`)` is a Boolean expression that tests membership in $\mathcal{P}$. The expression is simply a Boolean conjunction of the linear inequalities that define $\mathcal{P}$: if $\mathcal{P}_0$ is the conjunction of constraints $a_i x \leq c_i$ for $i = 1, \ldots, n$, the test `inside(`$\mathcal{P}$`)` is the Boolean expression $a_1 x \leq c_1$ `&&` $\cdots$ `&&` $a_n x \leq c_n$.

Loops are harder to translate. Consider an instruction `loop` $\mathcal{P}$ s. After having translated s into `tr(`s`)`, we pick a fresh name i for the iteration variable, which corresponds to the last dimension of $\mathcal{P}$, the only dimension that is not already determined by the context. We then perform the following steps, broadly similar to Fourier-Motzkin elimination:

- Decompose the constraints of $\mathcal{P}$ (which are of the form $a \cdot x \leq c$) in three groups: the first group, $\mathcal{P}_0$, where the coefficient of i is zero; the second group, $\mathcal{P}_+$, where this coefficient is positive; and the third group, $\mathcal{P}_-$, where it is negative.
- If $\mathcal{P}_+$ or $\mathcal{P}_-$ contains no constraints, code generation fails, as it corresponds to an unbounded loop.
- Each constraint $ui + \sum_k a_k x_k \leq c$ of $\mathcal{P}_+$ provides an upper bound for i: we have $i \leq (c - \sum_k a_k x_k)/u$, thus $i \leq \lfloor (c - \sum_k a_k x_k)/u \rfloor$ since i is an integer. We want a strict upper bound, which is given by $i < \lfloor (c + u - \sum_k a_k x_k)/u \rfloor$. There are several constraints, so the upper bound is in fact the minimum of all those expressions. This is possible, since $\mathcal{P}_+$ contains at least one constraint. We call the resulting expression for the upper bound e_+.
- Likewise, each constraint $ui + \sum_k a_k x_k \leq c$ in $\mathcal{P}_-$ is equivalent to $(\sum_k a_k x_k - c)/(-u) \leq i$, thus to $\lfloor (\sum_k a_k x_k - c + (-u) - 1)/(-u) \rfloor \leq i$, providing a lower bound for i. The actual lower bound e_- will then be the maximum of all these expressions, which is well-defined since $\mathcal{P}_-$ is nonempty.
- Generate the code `if (inside(`$\mathcal{P}_0$`)) for (`i `=` e_-`;` i `<` e_+`;` i`++) tr(`s`)`.

It is relatively easy, using familiar properties of integer division, to prove that for all $\mathcal{E}$:

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

Thus, by case analysis on whether $\mathcal{E} \in \mathcal{P}_0$ or not, we can prove that the semantics of the generated Loop program is the same as the semantics of the initial PolyLoop program, assuming that the semantics of s and `tr(`s`)` are the same.

To simplify the generated code, we also distinguish the case where $\mathcal{P}$ contains an equality on i, say $ki = e$. In this case, there is only one possible value of i. We can then express all other constraints on i, using this equality, as a polyhedron $\mathcal{P}'$ which does not depend on i. Then, if the coefficient k is 1, the value of i is simply e, and we produce the code `if (inside(`$\mathcal{P}'$`)) let` i `=` e `in tr(`s`)`. If the coefficient k is not 1, we need to guard the computation by a divisibility check:

$$\texttt{if (inside(}\mathcal{P}'\texttt{) \&\& } e \texttt{ mod } k = 0\texttt{) let } i \texttt{ = } \lfloor e/k \rfloor \texttt{ in tr(}s\texttt{)}$$

8 COQ DEVELOPMENT

The totality of the algorithms presented above have been formalized and proved in Coq. For this purpose, we had to develop a small library of linear algebra, define the operations we needed on polyhedra, and prove their correctness theorems. We also had to formalize the semantics of the various languages presented above to be able to express the correctness theorems of the code generation.

8.1 Linear algebra

When working with vectors and matrices in a dependently-typed framework such as Coq or Agda, it is tempting to use dependent types such as `vec` n A, the type of vectors of A of dimension n, and `mat` p q A, the type of $p \times q$ matrices of A. This is close to mathematical usage but requires many retyping functions to be used, e.g. to convert between `vec` $(n+1)$ A and `vec` $(1+n)$ A.

In our development, we chose to represent vectors as lists of integers, with the convention that each list is followed by infinitely many zeros. With this convention, a list $[x_1; \ldots; x_k]$ can be interpreted as a n-dimensional vector for any n: the vector is either $(x_1, \ldots, x_n)$ if $n \leq k$, or $(x_1, \ldots, x_k, 0, \ldots, 0)$ if $n > k$. Likewise, a matrix is just a list of vectors, with the convention that lines not represented in the list are the all-zero vector. This simply-typed representation is a good match for reasoning about the polyhedral model: sometimes the model forces us to consider that several matrices or vectors have the same dimensions by padding them with zeros if needed; this is a no-operation in our representation.

Operations over vectors and matrices are refreshingly easy to define and comprise vector addition, scalar multiplication, dot product, lexicographic ordering, and matrix application. However, vectors and matrices do not have unique representations (trailing zeros can be materialized or left implicit), hence we need to reason up to setoid equality.

A constraint (a linear inequality) is defined as a pair of a vector and a scalar, the pair `(a, c)` representing the constraint $\mathtt{a} \cdot x \leq \mathtt{c}$. A polyhedron is simply a list of constraints.

8.2 Operations on polyhedra

We also need a number of operations on polyhedra: emptiness check, intersection, canonicalization, projection on the first d dimensions, etc. To this end, we reused the Verified Polyhedron Library (VPL) [Boulmé et al. 2018], a mixed Coq-OCaml library that uses *a posteriori* certification: operations over polyhedra are implemented efficiently in OCaml and produce Farkas-style certificates [Schrijver 1998, corollary 10.1a]; the results and their certificates are then checked by Coq functions that have been proved (once and for all) to be sound (if the checker succeeds, the result is correct) [Fouilhé and Boulmé 2014]. Since certificate checking may fail, most operations over polyhedra live in a monad that supports errors and also the possibility that the OCaml implementation is not pure (e.g. maintains internal state).

A limitation of the VPL is that most of its operations are specified by one-way implications instead of equivalences. For example, consider the two VPL operations we use most: the emptiness test `isempty` and the `assume` function that adds a linear constraint $ax \leq c$ to a polyhedron. They are specified by the following implications:

$$\mathtt{isempty}(\mathcal{P}) = \mathtt{true} \Longrightarrow \mathcal{P} = \emptyset$$
$$x \in \mathcal{P} \land ax \leq c \Longrightarrow x \in \mathtt{assume}(\mathcal{P}, ax \leq c)$$

However, `isempty` is allowed to return `false` even for an empty polyhedron, and $\mathtt{assume}(\mathcal{P}, ax \leq c)$ is allowed to return a polyhedron larger than the intersection of $\mathcal{P}$ with the semi-space $ax \leq c$. This is fine for the original application for which the VPL was developed, namely as a relational domain for static analysis by abstract interpretation: the implications that are proved are sufficient to show the soundness of the static analysis; over-approximation in the result of the analysis is allowed.

For the emptiness check, we can use the VPL `isempty` predicate directly, since the implication proved by the VPL is the one we need. Other operations we need must be synthesized from what the VPL provides. For example, converting a polyhedron from our representation (a list of linear constraints $a_i x \leq c_i$ for $i = 1, \ldots, k$) to the internal VPL representation is not trivial. We compute $\mathcal{P} = \mathtt{assume}(\ldots\ \mathtt{assume}(\emptyset, a_1 x \leq c_1) \ldots, a_k x \leq c_k)$, then check that $\mathtt{isempty}(\mathtt{assume}(\mathcal{P}, \neg(a_i x \leq c_i))) = \mathtt{true}$ for every $i = 1, \ldots, k$. The VPL specifications guarantee that the exact result $Q = \bigcap_i (a_i x \leq c_i)$ is included in $\mathcal{P}$. The extra checks guarantee that $\mathcal{P} \subseteq (a_i x \leq c)$ for $i = 1, \ldots, k$, from which it follows that $\mathcal{P} \subseteq Q$, hence $\mathcal{P}$ is the exact result Q. Of course, any of the extra check can fail, causing the conversion to fail. The conversion is, therefore, presented as a monadic operation in our code.

Projection of the first d dimensions of a polyhedron is implemented by iterating the Fourier-Motzkin elimination algorithm d times. We proved that Fourier-Motzkin elimination is exact on $\mathbb{Q}$, which implies that it is sound on $\mathbb{Z}$ but can result in over-approximation, as discussed in section 5. In the following, `isExactProjection n pol proj` expresses the fact that `proj` is the $\mathbb{Q}$-projection of `pol` over the dimension `n`. For `s` a positive integer and `pol` a polyhedron, `scale_poly s pol` is the polyhedron `pol`, scaled `s` times, so that we can express membership of rational points without using rational numbers. The functions `in_poly` check membership, `mult_vector` multiply a vector by an integer, and `assign n k v` returns the vector `v` where the `n`-th component was replaced by `k`.

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

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

The remaining polyhedral operations are those used in steps 3 and 4 of the syntax tree generation algorithm. In order to simplify the translation function from a polyhedral program into a PolyLoop program, these two steps have been combined inside the `split_and_sort` function. This function takes as input a list of polyhedra, and splits and sorts the result into a list where each element is a pair of a polyhedron, and the indices of the input polyhedra that contain this polyhedron. The properties below are those that were expressed in section 5 to prove the correctness of the code generation algorithm. The `WHEN x ← e THEN` notation means “if the monadic computation `e` succeeds and returns value `x`, then …”.

```
Lemma split_and_sort_disjoint :
  forall n (pols : list polyhedron),
    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 : list polyhedron),
    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 : polyhedron * list nat),
        In ppl out ∧ In i (snd ppl) ∧ in_poly p (fst ppl) = true.

Lemma split_and_sort_sorted :
  forall n (pols : list polyhedron),
    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).
```

8.3 Correctness theorems

There is a semantics preservation theorem for each of the code generation steps presented previously. We only show the final theorem that results from the composition of these individual proofs:

```
Theorem complete_generate_many_preserve_sem :
  forall es n (pis : Poly_Program) 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.
```

The hypothesis `WHEN st ← complete_generate_many es n pis THEN` means that the code generator succeeds and produces the Loop program `st`. Several other hypotheses are mostly technical, ensuring that the initial program is well-formed. In particular, `n` being the total dimension of the input program, the two hypotheses `pis_have_dimension pis n` and `forall pi, In pi pis → ...` ensure that the input program does not refer to non-existing variables. Code generation is done in an environment of fixed size, that is, where it is already decided what will be a variable and what will be a parameter. The variable `es` is the size of that environment; we thus have the two conditions `length env = es`, which ensures that the environment that is used has the correct size, and `(es ≤ n)%nat`, which ensures that the size of this environment is not greater than the dimension of the input program. Under these hypotheses, we conclude that every execution of the generated Loop program `st` corresponds to one of the possible executions of the initial Poly program `pis`.

8.4 Extraction and execution

We used Coq's extraction facility to generate OCaml code from our Coq development and the Coq part of the VPL. This extracted code can be linked with the rest of the VPL and a Loop printer to produce an executable prototype of our code generator.

By lack of a concrete syntax and a parser for our input language Poly, we could only hand-code a number of polyhedral programs taken from Bastoul [2004] and visually inspect the generated code. We now show three such examples.

The first example, shown in Figure 12, has only one polyhedron and describes the scanning of a $n \times m$ rectangle. Our generator produces two nested loops corresponding to the two dimensions of the schedule, plus two `let` bindings that map the schedule to points in the input polyhedron.

The second example, shown in Figure 13, has only one polyhedron and describes the scanning of a sequence of n instructions, but with a schedule that is not unimodular. The resulting code contains a test with a modulo operation before executing the instruction.

The third example, shown in Figure 14 and taken from Bastoul [2004], illustrates the case of multiple polyhedra. To avoid an unreasonable increase in the size of the produced code (which would otherwise include numerous "let" bindings), the initial program is a polyhedral program with lexicographic scheduling. The generated code is quite good, with innermost loops that contain no conditionals and involve no min/max computations.

$$(\mathtt{I},\ \{(n,m,i,j) \in \mathbb{Z}^4 \mid 0 \le i \le n \land 0 \le j \le m\},\\ (n,m,i,j) \mapsto (j,i),\\ (n,m,i,j) \mapsto (i,j))$$

(a) Initial polyhedron

```
if (0 ≤ n)
  for (t₁ = 0; t₁ < 1 + m; t₁++)
    for (t₂ = 0; t₂ < 1 + n; t₂++)
      let i = t₂ in
      let j = t₁ in
        I(i, j)
```

(b) Generated program

Fig. 12. Code generation for a single polyhedron

$$(\mathtt{I},\ \{(n,i) \in \mathbb{Z}^2 \mid 0 \le i \le n\},\\ (n,i) \mapsto (2i),\\ (n,i) \mapsto (i))$$

(a) Initial polyhedron

```
for (t₁ = 0; t₁ < 1 + 2 * n; t₁++)
  if (t₁ mod 2 = 0)
    let i = ⌊t₁/2⌋ in
      I(i)
```

(b) Generated program

Fig. 13. Code generation for a single polyhedron with modulo

9 RELATED WORK

9.1 Verification of loop optimizations

There are two main ways to verify a compilation pass: compiler proof, where the pass is proved once and for all to preserve semantics for all inputs; and translation validation [Pnueli et al. 1998], where at every run of the pass the input and output codes are compared for semantic equivalence. We are not aware of any significant loop optimization that has been verified with the compiler proof approach. However, translation validation has been applied many times to loop optimizations.

For optimizations that mostly preserve the execution order of basic instructions, validation can use symbolic representations of the codes plus specialized, efficient equivalence checking algorithms. Examples include loop unrolling as handled by Necula [2000], software pipelining by Tristan and Leroy [2010], loop-invariant code motion by Tristan et al. [2011], and loop peeling and induction variable strength reduction by Tate et al. [2011]. In contrast, loop transformations that completely change the evaluation order, such as permutation, fusion or tiling, require more advanced and more costly validation techniques. Zuck et al. [2005] add permutation rules, specific to this purpose, to the logic of their TVOC validator. Kundu et al. [2009] integrate similar permutation rules in their parameterized equivalence checking approach. Barthe et al. [2016] use a relational program logic that they reduce to Hoare logic via product programs. Churchill et al. [2019] automate the product program approach using SMT solving and loop matching heuristics.

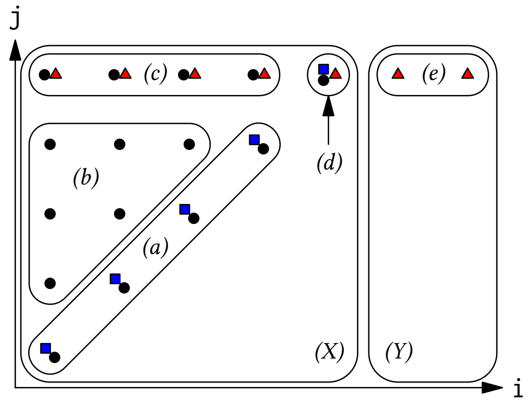
For a loop optimizer that uses the polyhedral model, it is tempting to base validation on the polyhedral model as well: build the polyhedral representation of the optimized code and compare it with the polyhedral representation of the initial code. This is the approach followed by Verdoolaege et al. [2012] and by [Namjoshi and Singhania 2016]. However, this is not always possible: the code generated by a polyhedral optimizer can contain non-affine conditional control (such as `if` x `mod` $2 = 0$) that cannot be expressed by a polyhedral representation. Schordan et al. [2014]

$$(\mathtt{I}_1, \{(n,m,i,j) \in \mathbb{Z}^4 \mid 1 \le i \le n \land j = i\},$$
$$(n,m,i,j) \mapsto (i,j)),$$

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

$$(\mathtt{I}_3, \{(n,m,i,j) \in \mathbb{Z}^4 \mid 0 \le i \le m \land j = n\},$$
$$(n,m,i,j) \mapsto (i,j))$$

(a) Initial program



(b) Polyhedra (with $n < m$). $\mathtt{I}_1$ is blue squares, $\mathtt{I}_2$ is black dots, and $\mathtt{I}_3$ is red triangles.

```
for (i = 1; i < min(1+n, 1+m); i++) (X)
  if (n ≤ i)
    let j = i in (d)
      I₁(i, j)
      I₂(i, j)
      I₃(i, j)
  if (i ≤ n − 1)
    let j = i in (a)
      I₁(i, j)
      I₂(i, j)
  for (j = i+1; j < n; j++) (b)
    I₂(i, j)
  if (i ≤ n − 1)
    let j = n in (c)
      I₂(i, j)
      I₃(i, j)
for (i = max(1, n+1); i < 1+m; i++) (Y)
  let j = n in (e)
    I₃(i, j)
for (i = max(1, m+1); i < 1+n; i++)
  let j = i in
    I₁(i, j)
    I₂(i, j)
  for (j = i+1; j < 1+n; j++)
    I₂(i, j)
```

(c) Generated code

Fig. 14. Code generation for several polyhedra, with an implicit lexicographic schedule in the initial program

investigate richer representations and equivalence checks that can account for such non-affine control.

The approach that we follow in this paper combines compiler proof and translation validation in a way that plays to their respective strengths. As explored by Pilkiewicz [2013], loop transformations are performed and validated over a polyhedral representation, using only Presburger arithmetic formulas, without a need for SMT solving or specialized decision procedures. As reported in this paper, the final code generation step is proved semantics-preserving once and for all, avoiding difficulties with translation validation caused by non-affine control in the generated code.

9.2 Code generation in the polyhedral model

Early work on automatic parallelization of loop nests introduced polyhedral techniques only for analyzing dependencies between loops: the loop nests were still represented by Abstract Syntax Trees and optimized by rewriting these ASTs [Feautrier 1991; Lamport 1974]. Pugh [1991] and Lu [1991] introduced the idea of representing loop nests by their polyhedral models and transforming directly the models. This idea gave rise to a new need: synthesizing imperative code from a polyhedral representation, an operation often called “polyhedra scanning” because it involves enumerating all the integer points in a set of polyhedra. Ancourt and Irigoin [1991] introduced the use of Fourier-Motzkin elimination to project the enumeration space on successive iteration dimensions. This technique works well for a single polyhedron but produces inefficient control for more complex spaces. Quilleré et al. [2000] showed how to avoid this inefficiency by decomposing the enumeration space as a disjoint union of polyhedra. Bastoul [2004] further developed these ideas, leading to the well-known CLooG code generator.

Our Coq formalization and correctness proof described in this paper follows the CLooG approach and covers most of the techniques described in Bastoul [2004], with the notable exception of non-unit loop strides. More recent improvements also remain to be formalized, such as those described by Grosser et al. [2015] and by Razanajato et al. [2017]. We have not investigated the significantly different approaches to polyhedra scanning proposed by Boulet and Feautrier [1998] and more recently by Chen [2012].

9.3 Verified neural networks

A new application of polyhedral-based compilation appeared recently in the context of machine learning: generating efficient low-level code implementing deep neural networks and recurrent neural networks expressed in high-level matrix and tensor notations. For instance, polyhedral optimization plays an important role in Tensor Comprehensions [Vasilache et al. 2019] and in Tiramisu [Baghdadi et al. 2019]. In parallel, the formal verification of neural networks is making progress, with new methodologies and tools such as Reluplex [Katz et al. 2017] and DeepPoly [Singh et al. 2019]. This is a strong incentive to formally verify the optimizers and code generators that produce the actual implementations of these neural networks, as a way to ensure that the properties formally verified at the level of the neural network carry over to the actual implementation.

10 CONCLUSIONS AND FUTURE WORK

The work described in this paper is the first formal verification of a relatively sophisticated polyhedral code generator, broadly similar to those used in production compilers. The verification combines linear algebra with semantic reasoning in interesting ways. It exposed one oversight in the literature: the loop generation algorithm of Quilleré et al. [2000] can fail in the case where the projection of a polyhedron produces three or more disjoint polyhedra, because it is not always possible to order these polyhedra in a way that enforces the desired execution order. It is good to know about this limitation, even though we do not know whether the problematic case can show up in practice with production optimizers such as Graphite and Polly.

Several ideas from Bastoul [2004] remain to be implemented and verified. This includes the generation of stride loops, where, at each iteration, the loop index is incremented by a stride $s > 1$. Additional simplifications are also possible at the PolyLoop level.

The next step in code generation is to produce actual C code (or CompCert Clight code) instead of stopping at the Loop language. The main challenge is to deal with arithmetic overflow in index computations. Loop uses exact, arbitrary-precision arithmetic to compute array indices, polyhedron membership, and loop bounds, while performance demands that we use fixed-precision machine arithmetic. Machine arithmetic can overflow, causing undefined behaviors or incorrect results. Cuervo Parrino et al. [2012] propose an elegant solution to this problem: static analysis of the polyhedral program produces a Boolean condition that, when true, guarantees the absence of overflows in the whole program. They suggest to evaluate the condition at run-time, branching to the optimized, generated loop nest if true, and to the original, unoptimized loop nest if false.

Finally, to test the performance of the generated code and fine-tune the code generation algorithms, it would be nice to connect our code generator with an existing front-end that produces polyhedral models.

REFERENCES

Corinne Ancourt and François Irigoin. 1991. Scanning Polyhedra with DO Loops. In *PPoPP'91: 3rd symposium on Principles & Practice of Parallel Programming*. ACM, 39–50.

Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman P. Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *CGO 2019: International Symposium on Code Generation and Optimization*. IEEE, 193–205.

Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2016. Product programs and relational program logics. *J. Log. Algebraic Methods Program.* 85, 5 (2016), 847–859.

Cédric Bastoul. 2004. Code Generation in the Polyhedral Model Is Easier Than You Think. In *PACT'04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 7–16.

Ulysse Beaugnon, Alexey Kravets, Sven Van Haastregt, Riyadh Baghdadi, David Tweed, Javed Absar, and Anton Lokhmotov. 2014. VOBLA: a vehicle for optimized basic linear algebra. In *LCTES'14: conference on Languages, compilers and tools for embedded systems*. ACM, 115–124.

A. J. Bernstein. 1966. Analysis of Programs for Parallel Processing. *IEEE Transactions on Electronic Computers* EC-15, 5 (1966), 757–763.

Pierre Boulet and Paul Feautrier. 1998. Scanning polyhedra without DO-loops. In *PACT'98: conference on Parallel Architectures and Compilation Techniques*. IEEE, 4–11.

Sylvain Boulmé, Alexandre Maréchal, David Monniaux, Michaël Périn, and Hang Yu. 2018. The Verified Polyhedron Library: an Overview. In *SYNASC 2018: 20th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE, 9–17.

Chun Chen. 2012. Polyhedra Scanning Revisited. In *PLDI 2012: conference on Programming Language Design and Implementation*. ACM, 499–508.

Berkeley R. Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic program alignment for equivalence checking. In *PLDI 2019: conference on Programming Language Design and Implementation*. ACM, 1027–1040.

Bruno Cuervo Parrino, Julien Narboux, Eric Violard, and Nicolas Magaud. 2012. Dealing with arithmetic overflows in the polyhedral model. In *IMPACT 2012: 2nd International Workshop on Polyhedral Compilation Techniques*.

Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *Int. J. Parallel Program.* 20, 1 (1991), 23–53.

Paul Feautrier and Christian Lengauer. 2011. The Polyhedron Model. In *Encyclopedia of Parallel Programming*, David Padua (Ed.). Springer, 1581–1592.

Alexis Fouilhé and Sylvain Boulmé. 2014. A Certifying Frontend for (Sub)polyhedral Abstract Domains. In *VSTTE 2014: Verified Software: Theories, Tools and Experiments (LNCS)*, Vol. 8471. Springer, 200–215.

Tobias Grosser, Armin Größlinger, and Christian Lengauer. 2012. Polly – Performing Polyhedral Optimizations on a Low-level Intermediate Representation. *Parallel Processing Letters* 22, 04 (2012).

Tobias Grosser, Sven Verdoolaege, and Albert Cohen. 2015. Polyhedral AST Generation Is More Than Scanning Polyhedra. *ACM Trans. Program. Lang. Syst.* 37, 4, Article 12 (July 2015), 50 pages.

Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *CAV 2017: Computer Aided Verification - 29th International Conference (LNCS)*, Vol. 10426. Springer, 97–117.

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *POPL'14: 41st symposium Principles of Programming Languages*. ACM, 179–191.

Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. 2009. Proving Optimizations Correct using Parameterized Program Equivalence. In *PLDI 2009: conference on Programming Language Design and Implementation*. ACM, 327–337.

Leslie Lamport. 1974. The Parallel Execution of DO Loops. *Commun. ACM* 17, 2 (1974), 83–93.

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (2009), 107–115.

Lee-Chung Lu. 1991. A Unified Framework for Systemic Loop Transformation. In *PPOPP 1991: 3rd Symposium on Principles & Practice of Parallel Programming*. ACM, 28–38.

Steven S. Muchnick. 1997. *Advanced compiler design and implementation.* Morgan Kaufmann.

Kedar S. Namjoshi and Nimit Singhania. 2016. Loopy: Programmable and Formally Verified Loop Transformations. In *SAS 2016: Static Analysis, 23rd International Symposium (LNCS)*, Vol. 9837. Springer, 383–402.

George C. Necula. 2000. Translation validation for an optimizing compiler. In *PLDI 2000: conference on Programming Language Design and Implementation.* ACM, 83–95.

Alexandre Pilkiewicz. 2010–2013. s2sLoop: a validator for polyhedral transformations. (2010–2013). https://github.com/pilki/s2sLoop

Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *TACAS'98: Tools and Algorithms for Construction and Analysis of Systems (LNCS)*, Vol. 1384. Springer, 151–166.

William Pugh. 1991. Uniform techniques for loop optimization. In *ICS 1991: 5th international conference on Supercomputing.* ACM, 341–352.

Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. 2000. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming* 28, 5 (2000), 469–498.

Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2017. Halide: Decoupling Algorithms from Schedules for High-performance Image Processing. *Commun. ACM* 61, 1 (Dec. 2017), 106–115.

Harenome Razanajato, Vincent Loechner, and Cédric Bastoul. 2017. Splitting Polyhedra to Generate More Efficient Code. In *IMPACT 2017: 7th International Workshop on Polyhedral Compilation Techniques.*

Markus Schordan, Pei-Hung Lin, Daniel J. Quinlan, and Louis-Noël Pouchet. 2014. Verification of Polyhedral Optimizations with Constant Loop Bounds in Finite State Space Computations. In *ISoLA 2014: Leveraging Applications of Formal Methods, Verification and Validation (LNCS)*, Vol. 8803. Springer, 493–508.

Alexander Schrijver. 1998. *Theory of Linear and Integer Programming.* Wiley.

Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin T. Vechev. 2019. An abstract domain for certifying neural networks. *Proc. ACM Program. Lang.* 3, POPL (2019), 41:1–41:30.

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2011. Equality Saturation: A New Approach to Optimization. *Log. Methods Comput. Sci.* 7, 1 (2011).

Konrad Trifunović, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, and Ramakrishna Upadrasta. 2010. GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation. In *GROW'10: 2nd GCC Research Opportunities Workshop.*

Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. 2011. Evaluating value-graph translation validation for LLVM. In *PLDI 2011: conference on Programming Language Design and Implementation.* ACM, 295–305.

Jean-Baptiste Tristan and Xavier Leroy. 2010. A simple, verified validator for software pipelining. In *POPL 2010: 37th symposium Principles of Programming Languages.* ACM, 83–92.

Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2019. The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically. *ACM Trans. Archit. Code Optim.* 16, 4 (2019).

Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. 2014. Schedule Trees. In *IMPACT 2014: 4th International Workshop on Polyhedral Compilation Techniques*, Sanjay Rajopadhye and Sven Verdoolaege (Eds.).

Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. 2012. Equivalence checking of static affine programs using widening to handle recurrences. *ACM Trans. Program. Lang. Syst.* 34, 3 (2012), 11:1–11:35.

Lenore Zuck, Amir Pnueli, Benjamin Goldberg, Clark Barrett, Yi Fang, and Ying Hu. 2005. Translation and Run-Time Validation of Loop Transformations. *Formal Methods in System Design* 27 (2005), 335–360.

APPENDIX

A CYCLES IN THE LEXICOGRAPHIC ORDERING GRAPH

Consider the following three disjoint polyhedra:

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

Suppose those three polyhedra are the result of step 2 of the algorithm. Since they are disjoint, they also are the result of step 3. In step 4, the lexicographic ordering graph contains a cycle. Indeed, we have $(0,0,0) \in \mathcal{P}_1$ and $(0,0,1) \in \mathcal{P}_2$, thus the loop for $\mathcal{P}_1$ must precede the loop for $\mathcal{P}_2$. Likewise, $\mathcal{P}_2 \ni (1,0,0) \prec (1,0,1) \in \mathcal{P}_3$, hence the loop for $\mathcal{P}_2$ must precede the loop for $\mathcal{P}_3$, which itself must precede the loop for $\mathcal{P}_1$ because $\mathcal{P}_3 \ni (0,1,0) \prec (0,1,1) \in \mathcal{P}_1$. Therefore, the three loops cannot be ordered. This problem is reminescent of the failure case of the painter's algorithm in computer graphics, and the solutions used there (splitting the polygons) could also be used in our case. However, while this shows the three loops cannot be ordered, the previous steps in the generation algorithm often prevent the problem from happening. In our case, the problem would only manifest itself if we have these three polyhedra and only x and y were parameters, while z was not. We have not yet seen cases where the problem happens after the first step of the generation, and we suspect it cannot happen; however, we did not formally prove it, and did not find justification for it in the literature.