Semantic Polyhedral Model for Arrays and Lists

Paul Iannetta
Supervisors: Laure Gonnord and Lionel Morel

June 1st, 2018

Abstract

The polyhedral model is a powerful reasonning framework that permits to optimize intensive computation kernels (piece of imperative code). However in this report we propose to define a "Semantic Polyhedral Model", a formalisation of the main concepts on which the polyhedral model is built, from the notion of *operation* to *data dependencies*. This formulation, which is no longer based on syntactic computations but rather on semantic definitions, enables us to characterize the domain on which the polyhedral domain optimisations will apply. We also propose some extensions on programs with lists and memory sharing.

Résumé

Le modèle polyèdrique est un modèle puissant qui permet d'optimiser des noyaux de calculs intensifs (des bouts de code impératif). Cependant, dans ce rapport on se propose de définir un « Modèle Polyèdrique Sémantique » qui se veut une formalisation des concepts présent dans le modèle original en repartant de la notion d'*opération* jusqu'au *concept de dépendance en données*. Cette reformulation qui repose non plus sur des concepts syntaxiques mais sémantiques permet alors de caractériser le spectre d'application du modèle polyèdrique. Nous proposons aussi des extensions pour des programmes utilisant des listes et qui font du partage mémoire.

Contents

1 Introduction

The work presented in this report is the first step of the CODAS ANR project [1], for which we describe the general context (Section 1.1) and related work in the context of the polyhedral model (Section 1.2). Section 1.3 describes the motivation of this internship and its objective.

1.1 Context

This section as well as the next one are directly taken from the ANR proposal itself.

The rise of embedded systems and high performance computers generated new problems in high-level code optimization, especially for loops, both for optimizing embedded applications and for transforming programs for high-level synthesis (HLS). Moreover, everything involving data storage is of prime importance as it impacts power consumption, performance, and for hardware, chip area. Thus, there is an increasing need for better scheduling techniques for all kinds of programs, especially those manipulating a huge amount of data.

So far, the polyhedral model [FL11], a framework introduced in the late eighties, has been successfully applied to a range of these compilation problems, such as (semi-)automatic parallelization and code generation [Fea92a] or optimization of data movement [DI15]. However, this powerful model hits its limit as soon as we are faced with irregular programs (general `while` loops, unpredictable conditions). As a consequence, these powerful techniques have, for the moment, only seen their use in limited (but still important) niches, because of these intrinsic restrictions.

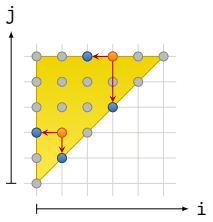
The long term objective of the CODAS ANR project is to give a general way to reason about and manipulate programs with general control flow and complex data structures. The project proposes to start with the current state of the polyhedral model, and then to enhance this framework both theoretically and algorithmically in order to be able to deal with more general programs.

The objective of this internship is to make a first step toward this ambitious objective.

1.2 The polyhedral model framework

The polyhedral model is a collection of techniques developed around a common intermediate representation of programs: integer polyhedra. Such a representation of programs inherits nice properties from the underlying mathematical structure. For instance, when loop transformations are represented as affine functions, compositions of transformations are also affine functions due to closure.

The polyhedral representation was linked to loop programs in an analysis proposed by Feautrier [Fea91] that provides exact dependency analysis information where statement instances[2] (i.e., statements executed at different loop iterations) and array elements are distinguished. The exact dependence information obtained through this analysis together with the use of linear programming techniques to explore the space of legal schedules [Fea92b] is what constitutes the basis of the polyhedral model for loop transformations.




```
for (i = 0; i < N; i++)
    for (j = i; j < N; j++)
S:      A[j] = f(A[i], A[j]);
```

$$\text{Domain}(S) = \{i, j \mid 0 \leq i \leq j < N\}$$
$$\text{Read}(S \mapsto A) = (i, j \to i); (i, j \to j)$$
$$\text{Write}(S \mapsto A) = (i, j \to i)$$

$$\text{Dep}(S \mapsto S) = \begin{cases} (i, j \to i, i) & : i < j \\ (i, j \to i - 1, j) & : 0 < i \\ \textit{corner case} & : \cdots \end{cases}$$

Figure 1: An example of polyhedral representation. Loop nests that fit the polyhedral model can be viewed as mathematical (constraint-based) objects, which can also be visualized geometrically.

Figure 1 illustrates the polyhedral representation with an example. The statement `S` is executed approximately $\frac{N^2}{2}$ times during the execution of this loop. The triangular region expressed as a set of constraints, called the *domain* of `S`, represents this set of dynamic execution instances. Accesses to array `A` from each of these statement instances can be succinctly captured through affine functions of the loop iterators. The dependencies

[1] http://codas.ens-lyon.fr/

[2] later called *operations*.

are also expressed as a function between two statement instances. [3] The key insight is that although the specific dependencies at a statement instance may differ, they are all captured by a function due to the regularity in the control flow. The figure illustrates dependencies for two instances, where one of the dependencies $(i, j \rightarrow i, i)$ has different lengths depending on the instance.

The "traditional" use of polyhedral techniques in optimizing compilers focuses on loop transformations. PLuTo [BHRS08] is a now widely used push-button tool for automatically parallelizing polyhedral loop nests. PLuTo tries to optimize locality in addition to parallelization. There is also significant work in data layout optimization for polyhedral programs where analyses are performed to minimize the memory requirement [DSV05]. Polyhedral techniques for loop transformations are now adopted by many production level compilers, such as GCC, IBM XL, and LLVM.

Recently, polyhedral techniques have been applied to many different areas beside loop transformations. One natural application of automatic parallelization techniques is in verification of given parallelizations where the tools take parallelized programs as inputs, and use polyhedral analysis to guarantee the absence of parallel bugs [BYR+11, YFRS13]. Another application of scheduling techniques is in the synthesis of ranking functions for proving program termination [ADFG10].

1.3 Limits of the polyhedral model, motivations of this internship

Although the polyhedral model provides strong analysis capabilities with many different applications, its main limitation is its applicability. The program must have regular control flow, and in addition, it has to be fully affine. Specifically, loop bounds, array accesses, and `if` conditions must be affine functions of the surrounding loop iterators and runtime constants. As an example, FFT (Fast Fourier Transform) has a regular control flow, but cannot be represented with the polyhedral model because it is not fully affine.

More precisely, we want to propose a formalization of the polyhedral model that does not rely on the syntax. The approach should be generic enough so that it can be applied to programs where loops are not explicit and where only the control flow graph is available. This would typically be used for scheduling programs in the form of LLVM intermediate representations from which we can hope to retrieve a representation with `while` loops.

This formalization should provide a generalization of [Fea91]. It should redefine the concept of dependency in a more general setting while preserving as much as possible a compatibility with the work presented in [Fea91]. The formalization should make the definition of dependencies as natural as possible.

Another goal of this formalization is to make the points that render the polyhedral decidable explicit. By doing so, we could pin-point where room is available for approximations.

1.4 Overview

The report is organized as follows: in Section 2 we recall the main concepts introduced in the seminal paper of Paul Feautrier ([Fea91]). Later works in the area are based on this paper, that is quite self-contained. From this paper, we make the notion of *dependency*, which is a fundamental concept for the rest of our work, explicit. We explain how it is defined and computed. In Section 3 we introduce the language we want to work on, its syntax and semantics, which is a non trivial extension of the small-step operational semantics that includes the notion of *iteration vectors* to keep track of dates where *operations* are computed. This allows the formal definition of dependencies, that is proven to be the same semantic concept as the one effectively computed by the polyhedral model framework (Section 4). Finally, in Section 5 we informally explain the next steps toward an effective computation of dependencies of general programs with lists. We conclude with future work.

2 Presentation of [Fea91]: contribution and limits

This section presents the results exposed in [Fea91]. Everything presented here is present in the original paper. However for the sake of ease, some definitions have been reworded and explanations added for more intuition.

The results in [Fea91] address the (automatic) parallelization of `for` loops. Extracting the full dependency graph between each operation performed by a general program is unpredictable and the only way to extract the dependencies correctly would be to execute the program. Hence, [Fea91] focuses on the analysis of programs with

[3] There are a number of ways to represent dependencies in the polyhedral model. Here, we represent them as a function from consumer instances to producer instances.

static-control and affine indices where this analysis is decidable.

2.1 Hypotheses and Restrictions

The notion of *static control* here means that all loops are `for` loops whose bounds are affine functions of structure parameters (that is, a set of integer variables that will be set once and for all by the means of an input statement and computations involving previously computed structure parameters), numerical constants and in-scope loop counters.

Moreover, the results presented here assume that `for` loops have a constant step of one, that there is no aliasing [4] and that the program does not try to access illegal memory cells. That said, as long as the step of the loop remains a constant known at compile time, the results still hold.

2.2 Notations

2.2.1 Operations and Statements

Let us first, clarify the distinction that will be made between *operation* and *statement*. The listing in Figure 2 will help illustrate the difference.

```
1 res := 1 (* s1 *)
2 for i from 1 to n:
3   res := res * i (* s2 *)
```

Figure 2: The factorial with a for loop.

In this example there are two statements (on lines 1 and 3) but $n+1$ operations (line 1 produces one operation whereas line 3 produces n) where n is a structure parameter. In other words a *statement* is a syntactic unit, whereas an *operation* is a temporal unit. When there are neither loops nor conditional statements *operations* and *statements* coincide.

2.2.2 Tracking operations

To be able to optimize the operations of a given program, the polyhedral model suggests computing the dependencies between all operations as long as they are within the restrictions presented in 2.1. Indeed, two operations that do not depend on each other can be parallelized, or at least, rescheduled in an order different than their lexical order in the original program. The notion of dependency between operations is thus central for the polyhedral community and that is why we first focus on its definition and computation.

In order to express dependencies between operations, we need a way to number each operation with a unique identifier – an iteration vector. This *iteration vector* is a vector whose coordinates are loop counters. The first coordinate is the counter of the most outer loop and the last coordinate is the counter of the most inner loop. For example, in the listing of figure 2, on line 3 the iteration vector is (i) and in figure 3 the iteration vector on line 3 is (i, j) while the iteration vector on line 5 is (i, j, k).

```
(* a and b are n-n matrices and c = ab *)
1 for i from 1 to n
2   for j from 1 to n
3     c(i, j) = 0 (* s1 *)
4     for k from 1 to n
5       c(i, j) = c(i, j) + a(i, k)*b(k, j) (* s2 *)
```

Figure 3: Product of polynomials with a for loop.

Remark (loop counter). With `for` loops, the concept of iteration variable is crystal clear since it coincides with loop counters. However, this concept is less clear when dealing with `while` loops, which will be addressed later.

From now on, *statements* will be denoted by s_i, *instantiated iteration vectors* by t_j (because an instantiated iteration vector can be seen as a *timestamp*) and *operations* by the pair (s_i, t_j). Note that two operations can have the same iteration vector, typically if they are at the same loop level. In order to know which is before the other a Boolean T_{s_1,s_2} is set to true if s_1 is before s_2 in the text source program. Intuitively, we define $Q_{s_1,s_2}(t)$ as the set of all the operations involving s_1 that have an influence on the computation of s_2 at time t. And we define $K_{s_1,s_2}(t)$ as the last operation having an influence on the computation of s_2 at time t.

Remark. The original paper defines Q and K by the way they are computed rather than giving a high level interpretation definition. The intent behind Q and K is as presented above, their formal definition will be exposed in the next subsection.

2.3 Computation of dependencies

We are now ready to formally define K and Q and explain how they are computed in the context of the polyhedral model (and the tool [Fea88]). Let's assume that we are computing values for a matrix M, and that we want to compute the operations on which $o_2 = (s_2, t_2)$ (an operation that needs to read values in M) depends. Let's as-

[4] We will relax this assumption later in this report.

sume that o_2 needs to read $M[g(t_2)]$ where g is an affine function of the iteration vector t_2.

However, before we can compute $Q_{s_1,s_2}(t_2)$ we need to gather candidates for s_1. We will thus take into account all operations whose statement is of the form $M[f(t_1)] := \ldots$ where f is an affine access function of the iteration vector. The operations on which s_2 depends will then be the union of the operations found with s_1 as their statement.

In order to explicitly define and compute the Q quantity, [Fea91] imposes the following conditions:

C1 the cells that s_1 and s_2 try to access should match: $f(t_1) = g(t_2)$;

C2 (s_1, t_1) should happen before (s_2, t_2) (ie. $t_2 \lhd t_1$, or $t_2 = t_1 \wedge T_{s_1,s_2}$ where $\lhd$ is the lexicographic ordering on vectors). This condition is denoted by $(s_1, t_1) \prec (s_2, t_2)$

C3 t_1 must be a valid iteration (denoted as $e(t_1) \geq 0$, this notation will become clear in the following example.)

Hence, the following definition of $Q_{s_1,s_2}(t)$ as:

$$Q_{s_1,s_2}(t) = \{t' | f(t') = g(t), (s_1, t') \prec (s_2, t), e(t') \geq 0\}$$

and

$$K_{s_1,s_2}(t_2) = \max_{\lhd} Q_{s_1,s_2}(t_2)$$

.

Theorem 1 ([Fea91] Dependencies are computable in the polyhedral model)**.** *The 3 conditions above lead to a system of affine constraints that is then computable by a Linear Programming solver (such as PIP [Fea88]).*

Proof. The proof can be found in the paper. All three conditions above lead to a finite set of affine constraints. The lexicographic maximum of such a set can be computed by solving a Linear Programming instance. □

Example (Computations of dependencies for the matrix product, shown in Fig. 3)**.** *This program is made of two statements: s_1 on line 3 and s_2 on line 5, that both write values for the array c. In order to compute the dependencies we need to compute Q_{s_1,s_1}, Q_{s_1,s_2} and Q_{s_2,s_2}. The respective Ks will be computed by taking the lexicographic maximum on the Qs.*

Let's start by computing Q_{s_1,s_1}. We can see that s_1 does not need to read any variable. Hence:

$$Q_{s_1,s_1} = \emptyset$$

Now, let us compute Q_{s_1,s_2}. Let (i_1, j_1) be the iteration vector of statement s_1 and (i_2, j_2, k_2) the iteration vector of statement s_2. We can then express C1, C2 and C3 as affine conditions. C1 is $(i_1, j_1) = (i_2, j_2)$. C2 is $(i_1, 1) < (i_2, j_2)$. And C3 is $1 \leq i, j \leq n$. This leads to:

$$Q(s_1, s_2)((i_2, j_2, k_2)) = \{(i_1, j_1) \mid i_1 < i_2 \wedge j_1 < j_2\}$$

Lastly, let us compute Q_{s_2,s_2}. Let (i_2, j_2, k_2) and (i'_2, j'_2, k'_2) the iteration vectors of statement s_2 at two distinct instants. We can then express C1, C2 and C3 as affine conditions. C1 is $(i_2, j_2, k_2) = (i'_2, j'_2, k'_2)$. C2 is $(i_2, j_2, k_2) < (i'_2, j'_2, k'_2)$. And C3 is $1 \leq i_2, j_2, k_2 \leq n$. This leads to:

$$Q(s_2, s_2)((i'_2, j'_2, k'_2)) = \{(i_2, j_2, k_2) \mid i_2 < i'_2 \wedge j_2 < j'_2 \wedge k_2 < k'_2\}$$

2.4 Toward a more semantic polyhedral model

The polyhedral model suffers from two main drawbacks that we want to address in this work:

- Firstly, its syntactic restrictions limit its usage in practice: developers usually want powerful languages and analyses and complain if the compiler rejects their program while parsing. Moreover, the polyhedral tools should work on abstract syntax trees where these restrictions are easily checkable, which limits in practice the development of polyhedral tools inside production compilers like gcc or `LLVM` because the AST is not always available when performing optimizations. [5]
- Secondly, it is limited to static (with constant or parametric size) arrays without any aliasing. In some cases, the developer might have used pointer arithmetic, or lists or trees, whose behavior is "covertly regular" but the polyhedral model doesn't apply at all.

Our work aims at formalizing an extension of the polyhedral model to more general programs than the ones historically addressed. This report proposes an original *semantic* definition of dependencies on general flowchart programs with arrays (and simple lists). We show that this definition:

- Has the same expressivity than the polyhedral model when a "syntactic polyhedral program" has been modified by a preprocessing phase like in production

[5] Actually, there exists polyhedral frameworks such that Graphite https://gcc.gnu.org/wiki/Graphite or Polly https://polly.llvm.org/ that try to recover high level information from low-level intermediate representations, however they did not formalize their applicability in a semantic fashion.

compilers where `for` loops (ie. static loops with an explicit loop iteration variable) have been replaced by `while` loops and where conditional statements are allowed.

- Also captures list dependencies, even in the case of aliasing.

To achieve our objective, we propose the following agenda:

- First, define a proper language expressive enough to express all the features of an imperative languages with arrays and lists. It should be able to express general computations, general array accesses, non static loops and finally some kind of aliasing on lists ;
- On this language, propose an operational semantics and the counterparts of all the key concepts of the polyhedral model ;
- Show that all these concepts are equivalent on the sub-case addressed by the polyhedral model ;
- Propose extensions handling more general loops and lists.

3 General imperative programs with iteration vectors

In order to properly define all the polyhedral model concepts in a semantic fashion, we first need to define a proper language and its semantics. The language we define is representative of general flowcharts programs without pointers, but with arrays and lists.

3.1 A mini language

We have to be able to deal with the fact that programs can process whatever type of data as well as using arrays and lists. However, we want to keep apart integer variables because we need them as iteration variables.

Hence, the language formalized in this paper is a pointer-less imperative language with native support for `while` loops, `if` statements as well as arrays and lists. This language is not designed to do any actual computations (since the only value it can work on is ♣) however it is designed to provide a precise definition of dependencies between operations.

In the grammar depicted in Figure 4, capital letters (X, Y, Z) are used as placeholders for variable names. n represents an element of $\mathbb{N}$ and term in lowercase represent an instance of the rule which shares the same first letter: *i.e* $a_{\cdot}$ is an instance of Aexp, $l_{\cdot}$ is an instance of Lexp, etc.

$\langle Aexp \rangle$ ::= n | intvar(X) | a_0 $\langle Aop \rangle$ a_1

$\langle Aop \rangle$ ::= '+' | '*' | '−' | '/'

$\langle Bexp \rangle$ ::= 'true' | 'false' | !(b_0)
| b_0 $\langle Bop \rangle$ b_1 | a_0 '<' a_1 | l_0 '<>' 'nil'

$\langle Bop \rangle$ ::= 'or' | 'and' | '='

$\langle Lexp \rangle$ ::= 'nil' | list(X)

$\langle Vexp \rangle$ ::= var(X) | var(X, a_0)

$\langle Cexp \rangle$ ::= 'skip' | c_0 ';' c_1
| 'if' b_0 'then' c_0 'else' c_1 'fi'
| 'while' b_0 'do' c_0 'done'
| v_0 := g_0
| intvar(X) := a_0
| 'cons' '(' g_0 ',' list(X_1) ')'
| 'nxt' '(' list(X_1) ')' | list(X_0) := list(X_1)
| listval(X) := g_0

$\langle Gexp \rangle$ ::= ♣ | l_0 | a_0 | v_0 | listval(l_0) | $g_0 \star g_1$

Figure 4: Our Mini-Language: syntax

The language itself is really permissive and can be used to write programs that are syntactically more general that those addressed by the polyhedral model. Semantically, as this language can express arbitrary behavior of more than two numerical variables, it is Turing complete. The language has not been restricted because we want to keep the capability to analyze all programs even though we will not be able to say something significant about some programs.

3.1.1 Informal semantics

Type annotations such as `intvar`, `var` and `list` are here to guarantee that the program is sufficiently correct to be handled by the grammar. `var` is a special case of array, namely array with only one cell.

Those annotations have the following role:

- `intvar(Var)` guarantees that `Var` is an integer and that we can perform standard (Presburger) arithmetic on it. Only integer variables will matter when we will compute dependencies. We assume that all required annotations will be present in the program, *i.e* after a *typing* phase even if they do not syntactically appear in the original program.

- `var(Var), var(Var, i) : var(Var)` is a shortcut for `var(Var, 0)`. `var(Var, i)` refers to the i-th cell of the array `Var`. The annotation `var` guarantees that `Var` is either an array or a scalar variable (ie. an array of size 1). This annotation is not appended during the annotation phase and it should be provided in the source program.
- `list(Var), nil, listval(Var) : listval` is an annotation used to express that we are dealing with the head of list `Var`. `nil` being the empty list, `listvar(nil)` is undefined. This type annotation is not appended during the annotation phase and it should be provided by the source program.
- `cons, nxt : nxt(nil)` is undefined, `nxt(list)` discards the head of `list`. list becomes its head. In `cons(a, list(b))`, `a` is consed to `list(b)` and the reference `list(b)` is updated to the address of the cell containing `a`. That means that after `cons(a, list(b)), list(b)` is the list that starts with `a` followed by `list(b)` as it was before the `cons` operation. Those operators provide the standard operations on lists (however, those operations are done in-place).

General expressions (`Gexp`) are here to allow the source program to make computations on all types without restrictions.

In the code listings that will appear in this report, we will most often not write `var(Var)` and `var(Var), idx`, but simply `Var` and `Var(idx)`.

The formal semantics is presented in the next section. We first introduce the notion of iteration vectors that will enable us to compute *operation dates* inside a rather classical operational semantics. Then we define an execution model (in particular the representation of states and the annotation programs before their execution) and then we define our semantics. Finally some classical definitions of the polyhedral model framework are rephrased on the traces induced by the operational semantics.

3.1.2 Semantic extension: iteration variables and iteration vectors for our language

`for` loops naturally introduce counter variables. These are very convenient to number the operations and allow to label them when investigating the dependencies between them. Unfortunately, `if` and `while` do not introduce such variables.

Hence, we have to artificially introduce variables that will serve this purpose. This is a classical activity in static analysis, for instance, this is widely used as a preliminary step to invariant generation in order to compute the worst-case execution time of a program [HAMM14].

Iteration variables are created so that operations are numbered hierarchically, the first level counts the number of operations at level zero, the second level those at level one, and so on. The iteration vector is the concatenation of these variables. The leftmost coordinate is the outermost iteration variable and the rightmost coordinate is the innermost iteration variable. This allows sorting of operations by their iteration vector, with respect to the lexicographic order.

However, since `if` statements have two branches we have to do some extra work in order to make them compatible with the lexicographic order. This is done by numbering the operation in the `then` branch by the opposite of the number of statements of that branch. As can be seen in the listing in Figure 5 the `then` branch of the `if` starts at -1 because this branch contains one statement. This works only because, unlike `while`, `if` are only executed once.

```
1 i := 5;
2 while i <> 1 do
3  if i mod 2 == 0
4    i := i / 2
5  else
6    i := 3*i + 1
```

a) Before annotation

```
01 k0 := 0;
02 i  := 5;
03 k0 := intvar(k0)+1;
04 k1 := 0;
05 while (var(i) <> 1)
06   if (intvar(i) mod 2 == 0)
07     k2 := -1;
08     i := var(i)/2;
09     k2 := var(k2)+1
10  else:
11    k2 := 0
12    i := 3*var(i)+1
13    k2 := intvar(k2)+1
14  k1 := intvar(k1)+1
15 k0 := intvar(k0)+1
```

b) After annotation

Figure 5: The Syracuse algorithm

For example, the statement on line 8 in the Figure 5b has $(k_0\ k_1\ k_2)$ as iteration vector and when the program is run the iteration vector is instantiated with the current values of k_0, k_1 and k_2.

Here again, we delegate the creation of the iteration variables to a *preprocessing phase*, as we explain in the next section.

3.1.3 Annotation of a general program

As can be seen in the listing in Figure 5, once annotated the program is much less readable. Hence, the annotation is done by performed by the algorithm in Figure

5b. The annotation phase inserts the iteration variables and append the type annotation 'itervar' to all variables with an occurrence in `if` and `while` conditions.

```
annot :: [Cmd] -> [Cmd]
annot prog = fst (annot_ 1 [0] (intvar('k0') := 0 : prog))

annot_ :: Integer -> [Integer] -> [Cmd] -> ([Cmd], Integer)
annot_ nxt vec (If cond pos neg : tl) =
  (If cond' pos' neg' : incr : tl', nxt')
    where
      cond' = annot_cond cond
      len  = length(pos)
      nxt0 = nxt + 1
      pos'', nxt1 = annot_ nxt0 vec pos
      neg'', nxt2 = annot_ nxt1 vec neg
      pos' = intvar('k' ++ nxt') := -len : pos''
      neg' = intvar('k' ++ nxt') :=    0 : neg''
      incr = intvar('k' ++ nxt) := intvar('k' ++ nxt) + 1
      tl', nxt' = annot_ nxt2 vec tl
annot_ nxt vec (While cond cs : tl) =
  (While cond' cs' : incr : tl', nxt')
    where
     cond' = annot_cond cond
     nxt0 = nxt + 1
     cs'', nxt1 = annot_ nxt0  vec cs
     cs' = intvar('k' ++ nxt') := 0 : cs''
     incr = intvar('k' ++ nxt) := intvar('k' ++ nxt) + 1
     tl', nxt' = annot_ nxt1 vec tl
annot_ nxt vec (cmd : tl) = (cmd : incr : tl', nxt)
    where
      incr = intvar('k' ++ nxt) := intvar('k' ++ nxt) + 1

annot_cond (cond1 and cond2) = (annot_cond cond1) and (
annot_cond cond2)
annot_cond (cond1 or cond2) = (annot_cond cond1) or (
annot_cond cond2)
annot_cond (not cond) = not (annot_cond cond)
annot_cond var(X) = intvar(X)
annot_cond cond = cond
```

Figure 6: Annotation Algorithm

From now on, we will only work on programs that are not annotated.

3.2 Execution Environment, final semantics of our mini-language

This section presents the language in more depth; in particular, the grammar and semantic will be fully defined. Since we need to store information during the execution in order to have a clear definition of dependencies (since everything here is defined dynamically there is no restriction on the class of dependencies we are able to define), a memory model has to be designed and integrated into the intermediate representation of the program we want to analyze.

3.2.1 Memory Model

The main hypothesis is that arrays as well as every thing contained in a list cell are contiguous. The details concerning memory management are not important as long as we can get the addresses of each object. A valid memory model is presented in the appendix A which describes the implementation of an interpreter for our mini-language in `PROLOG`.

```
1 c(0) := 0;
2 i := 1;
3 while i <= n do
4   c(i) = c(i-1) + 1;
5   i := i + 1
6 done
```

a) Before annotation

```
  k0 := 0;
1 c(0) := 0;
  k0 := k0 + 1;
2 i := 1;
  k0 := k0 + 1;
  k1 := 0;
3 while i <= n do
4   c(i) = c(i-1) + 1;
    k1 := k1 + 1;
5   i := i + 1;
    k1 := k1 + 1
6 done;
  k0 := k0 + 1
```

b) After annotation

Figure 7: Array filling with increasing values

```
 1 i := 0;
 2 list(vals) := nil;
 3 while i <= n do
 4   cons(i, list(vals));
 5   i := i + 1
 6 done;
 7 save(list(vals'), list(vals'));
 8 while list(vals) <> nil do
 9   res := listval(vals);
10   nxt(list(vals))
11 done
```

Figure 8: Creation and Reduction of a list

In order to guarantee those hypotheses, ie. we need to know the size of all the variables appearing in the program, we need to declare those variables as well as their size beforehand. However, in this paper, we assume the declaration has been done somewhere, and thus it is not included in our code examples (nevertheless, that is explicitly done in the `PROLOG` code.

3.2.2 States

A state should represent the name of the variables currently allocated as well as the state of the memory and the timestamp of the current operation. It is defined as a triple (Mem, Loc, Vec). More precisely, the types of Loc, Mem and Vec are as follows.

- Loc : $\texttt{Vars} \mapsto \mathbb{N}$
- Mem : $\mathbb{N} \mapsto \{\clubsuit\} \times (\texttt{String}^{\mathbb{N}} \times \mathbb{N}^{\mathbb{N}})$
- Vec : $\texttt{String}^{\mathbb{N}} \times \mathbb{N}^{\mathbb{N}}$

Loc is a map from the set of variables to the set of

Loc		Mem	
Variable names	Addresses	Addresses	Values
$\texttt{intvar}(k_0)$	0	0	3
$\texttt{var}(c, 0)$	1	1	1
$\texttt{var}(c, 1)$	2	2	2
$\texttt{var}(c, 2)$	3	3	3
$\texttt{intvar}(i)$	4	4	2
$\texttt{intvar}(k_1)$	5	5	4

Figure 9: End state of the annotated program of fig. 7b.

memory addresses; Mem is a map from the set of memory addresses to pairs made up of the value stored at this address as well as the instantiated iteration vector of the operation that wrote this cell; Vec is the instantiated iteration vector of the current operation.

As an illustration, Figure 9 gives the shape of the end state of the annotated program in Figure 7b.

And the end iteration vector is $\{(k_0), (3)\}$. Because, as we will see later, k_2 has been dropped at the end of the `while`.

In order to easily refer to coordinates of a `state` we define the following functions.

- $\texttt{Loc} : \text{state} \mapsto (\texttt{Vars} \mapsto \mathbb{N})$
- $\texttt{Mem} : \text{state} \mapsto \mathbb{N} \mapsto \{\clubsuit\} \times (\texttt{String}^{\mathbb{N}} \times \mathbb{N}^{\mathbb{N}})$
- $\texttt{Vec} : \text{state} \mapsto (\texttt{String}^{\mathbb{N}} \times \mathbb{N}^{\mathbb{N}})$

The parameter of type `state` will most often be written in subscript (eg. $\texttt{Loc}_\sigma$ is function of type $(\texttt{Vars} \mapsto \mathbb{N})$).

3.2.3 Semantic

We will now expose the (big-step) semantic rules of that mini-language. Those rules will define the execution relation $\rightarrow$ (this is a transition relation over $(\texttt{state} \times \texttt{Cexp}) \times \texttt{state}$).

Second, let's explain what means:

- $\sigma[\text{Var}(X, a_0) := g_0]$;
- $\sigma[\text{cons}(g_0, \texttt{list}(X_1))]$;
- $\sigma[\texttt{nxt}(\texttt{list}(X_1))]$;
- $\sigma[\texttt{list}(X_0) := \texttt{list}(X_1)]$.

- $\sigma' = \sigma[\text{Var}(X, a_0) := g_0]$ is identical to σ except that $\texttt{Mem}_{\sigma'}(\texttt{Loc}_{\sigma'}(\text{Var}(X, a_0))) = (g_0, \texttt{Vec}_\sigma)$.

$\sigma' = \sigma[\text{cons}(g_0, \texttt{list}(X_1))]$ is identical to σ except that $\texttt{Loc}_{\sigma'}(\texttt{list}(X_1)) = \texttt{new\_addr}$. Where `new_addr` is an address that has never been used up to this point and $\texttt{Mem}_{\sigma'}(\texttt{new\_addr}) = (g_0, \texttt{Vec}_\sigma)$.

$\sigma' = \sigma[\texttt{nxt}(\texttt{list}(X_1))]$ is identical to σ save for, assuming that $\texttt{Mem}_\sigma(\texttt{Loc}_\sigma(\texttt{list}(X_1)) = (g_0, \texttt{nxt\_val}))$, then $\texttt{Loc}_{\sigma'}(\texttt{list}(X_1)) = \texttt{nxt\_val}$.

$\sigma[\texttt{list}(X_0) := \texttt{list}(X_1)]$ is identical to σ save for, assuming that $\texttt{Loc}_\sigma(\texttt{list}(X_1)) = \texttt{addr}$, then $\texttt{Loc}_{\sigma'}(\texttt{list}(X_1)) = \texttt{addr}$.

$$\textsc{skip}\ \frac{}{\langle \sigma, \text{skip} \rangle \rightarrow \sigma'}$$

$$\textsc{compose}\ \frac{\langle \sigma, c_0 \rangle \rightarrow \sigma_1 \qquad \langle \sigma_1, c_1 \rangle \rightarrow \sigma'}{\langle \sigma, c_0; c_1 \rangle \rightarrow \sigma'}$$

The two rules above do not need much explanation but are those which allow considering programs as lists of statements.

$$\textsc{Assign tab}\ \frac{}{\langle \sigma, \text{Var}(X, a_0) := g_0 \rangle \rightarrow \sigma[\text{Var}(X, a_0) := g_0]^1}$$

$$\textsc{Cons}\ \frac{}{\langle \sigma, \text{cons}(g_0, \texttt{list}(X_1)) \rangle \rightarrow \sigma[\text{cons}(g_0, \texttt{list}(X_1))]^1}$$

Those two rules are the most important ones since they create dependencies between operations. What is important to notice is that the assignment operator, in some way, overloaded. Indeed, it does not only store the value of g_0 but it stores at the same time the value of the iteration vector.

$$\textsc{Nxt}\ \frac{}{\langle \sigma, \texttt{nxt}(\texttt{list}(X_0)) \rangle \rightarrow \sigma[\texttt{nxt}(\texttt{list}(X_1))]}$$

The rule 'Nxt' works in place. The head of the list $\texttt{list}(X_0)$ is discarded (it still continues to live in memory) and the reference $\texttt{list}(X_0)$ is updated and is now referring to the tail of $\texttt{list}(X_0)$. That will justify the rule 'Save'.

$$\textsc{Save}\ \frac{}{\langle \sigma, \texttt{list}(X_0) := \texttt{list}(X_1) \rangle \rightarrow \sigma[\texttt{list}(X_0) := \texttt{list}(X_1)]}$$

This rule 'Save' creates a new reference called $\texttt{list}(X_0)$ to the head of the list $\texttt{list}(X_1)$. The purpose of the 'Save' instruction is mainly to be able to rewind to the start of a list after traversing it. This is, for example, done in the example in Figure 8. Note that it can be used for other purposes which are not obvious or desirable. For example, let us assume that we have a list l and that we save it to list l', let us now assume that we cons a value to

[1] Also stores the iteration vector with the value

l and another value to l', we now have two lists l and l' whose heads are different but who share exactly (ie. the same memory cells) the same tail.

$$\text{WHF}\ \frac{\langle \sigma, b_0\rangle \to \text{false}}{\langle \sigma, \text{while } b_0 \text{ do c done}\rangle \to \text{drop}(\sigma)}$$

$$\text{WHT}\ \frac{\langle \sigma, b_0\rangle \to \text{true} \quad \langle \sigma, c_0\rangle \to \sigma_1 \quad \langle \sigma_1, \text{while } b_0 \text{ do c done}\rangle \to \sigma'}{\langle \sigma, \text{while } b_0 \text{ do c done}\rangle \to \text{drop}(\sigma')}$$

$$\text{IT}\ \frac{\langle \sigma, b_0\rangle \to \text{true} \quad \langle \sigma, c_1\rangle \to \sigma'}{\langle \sigma, \text{if } b_0 \text{ then } c_1 \text{ else } c_2 \text{ fi}\rangle \to \text{drop}(\sigma')}$$

$$\text{IF}\ \frac{\langle \sigma, b_0\rangle \to \text{false} \quad \langle \sigma, c_2\rangle \to \sigma'}{\langle \sigma, \text{if } b_0 \text{ then } c_1 \text{ else } c_2 \text{ fi}\rangle \to \text{drop}(\sigma')}$$

In addition to the `while` and `if` classic rules we need to add specials rules to handle the management of the iteration vector. The 'push' function add its parameter as a coordinate of the iteration vector while the 'drop' function discard the last added coordinate of the iteration vector.

$$\text{WH}'\ \frac{\langle \sigma, b_0\rangle \to \text{true} \quad \langle \text{push}(X, \sigma[\texttt{intvar}(X) := 0]), c\rangle \to \sigma_1 \quad \langle \sigma_1, \text{while } b_0 \text{ do c done}\rangle \to \sigma'}{\langle \sigma, \text{intvar}(X) := 0; \text{while } b_0 \text{ do c done}\rangle \to \text{drop}(\sigma')}$$

$$\text{IT}'\ \frac{\langle \sigma, b_0\rangle \to \text{true} \quad \langle \text{push}(X, \sigma[\texttt{intvar}(X) := 0]), c_1\rangle \to \sigma'}{\langle \sigma, \text{if } b_0 \text{ then intvar}(X) := a_0; c_1 \text{ else } c_2 \text{ fi}\rangle \to \text{drop}(\sigma')}$$

$$\text{IF}'\ \frac{\langle \sigma, b_0\rangle \to \text{false} \quad \langle \text{push}(X, \sigma[\texttt{intvar}(X) := 0]), c_2\rangle \to \sigma'}{\langle \sigma, \text{if } b_0 \text{ then } c_1 \text{ else intvar}(X) := 0; c_2 \text{ fi}\rangle \to \text{drop}(\sigma')}$$

Those three rules are applied as soon as a `while` or `if` is detected after special variables added by the annotation phase. This way they bypass the normal rules 'IF' and 'WH' and add a coordinate to our iteration vector. And once the rule is done, then the iteration vector is put back to its previous state.

For example at line 3 in the Figure 7b the variable monitored by the iteration vector is (k_0), at line 4 the variables monitored are (k_0, k_1). It should be noted that the internal representation handled by 'push' and 'drop' uses a FIFO, therefore, the operation that happens on the iteration vector when entering the `while` is that k_1 is pushed on top of k_0 and once the `while` is done k_1 is discarded and k_0 becomes again the head of the FIFO.

3.3 Traces

For ease of presentation, the relation $\to$ introduced in Section 3.2.3 is big-step, however when defining traces we will use the small-step counter part of $\to$ that we will also note $\to$.

Definition 3.1 (trace on states)**.** A *trace on states* Σ is a sequence of pairs of the form (state, command) $(\sigma_0, c_0) \to (\sigma_1, c_1) \to \ldots$, and an *initial trace* is a trace which starts from the empty state.

In the polyhedral model, the computations that are actually performed are not of prime importance. This is why we decided to consider that all those computations happen on a singleton set $\{\clubsuit\}$. Moreover, all the memory accesses are completely deterministic, hence there exists one unique initial trace. This leads to the following remark.

Remark. There is a one-to-one mapping between iteration vectors and states.

Therefore, we will from now on work directly on operations rather than states. Indeed, since an operation o is the pair (s, t) we can retrieve the corresponding state from t if necessary according to the previous remark.

Definition 3.2 (trace on operations)**.** A *trace on operations* O is a sequence of operations $o_1 \to o_2 \to \ldots$; an *initial trace* is a trace that with the trivial iteration vector (the iteration vector filled with zeroes).

Remark. From now on, trace will always refer to *trace on operations* unless stated otherwise. Moreover, we will only worked on traced were commands about annotations have all been trimmed. For example we do not work on the trace $(s_1, t_1) \to (k_0 := k_0 + 1, t') \to (s_2, t_2)$ but on the trace where there is no more references to command about annotations, hence, we would work with $(s_1, t_1) \to (s_2, t_2)$.

Definition 3.3 (reachability/validity)**.** An operation (s_1, t) is valid if and only if there exists an initial trace $O = \{o_i\}_{i\in\mathbb{N}}$ such that there exists o_{i_0} such that $o_{i_0} = (s_1, t)$.

Definition 3.4 (happens-before: $<$)**.** There is a natural order $<$ on operations: *happens-before*. $(s_1, t_1) < (s_2, t_2)$ iff forall traces Σ there exists i and i' such that $\sigma_{i_1} \to^+ \sigma_{i_2}$ and $t_1 = \texttt{Vec}(\sigma_{i_1})$ and $t_2 = \texttt{Vec}(\sigma_{i_2})$.

4 Dependencies for general programs

This section starts by defining dependencies in the setting permitted by our mini-language. Those definitions are very general and do not suffer from restriction due to staticness. The second subsection shows how to convert this dynamic definition to something statically computable when we are dealing with regular (polyhedral) programs.

4.1 Different types of dependencies

Definition 4.1 (rvars)**.** Let $o = (s, t)$ be an operation, the set of variables that s needs to read at time t is called $\texttt{rvars}(o)$.

Definition 4.2 (wvars)**.** Let $o = (s, t)$ be an operation, the set of variables that s will write at time t is called $\texttt{wvars}(o)$. $\texttt{wvars}(o)$ is either a singleton or the empty set.

Example. *Let us consider the operation defined by* $o = (var(a, i) := var(a, i-1) + var(a, i) + 1, t)$*. Let us assume that at time* t *the variable* i *is equal to* 1 *(This information is accessible since for* t *we can recover the whole state corresponding to* t *and therefore access the access the content of the memory at this state).*

In that case, $\texttt{rvars}(o) = \{var(a, 0), var(a, 1)\}$ *and* $\texttt{wvars}(o) = \{var(a, 1)\}$.

Definition 4.3 (Last write)**.** Given an initial trace O and an operation o_2 which belongs to O, the function `last` returns the operation o_{i_1} (which belongs to O) that last wrote the cell containing the variable v before o_2 reads it.

The function `last` satisfies the following formula:

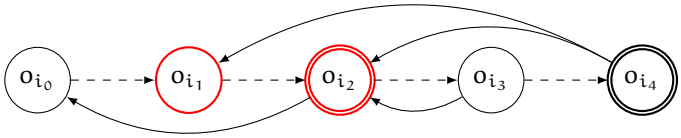
$$\begin{aligned} &\exists i_1, o_{i_1} = \texttt{last}_{O,o_2}(\texttt{wvars}(v)) \in O \\ \wedge\, &\forall i, i_1 < i < i_2, \texttt{wvars}(o_{i_1}) \neq \{v\} \end{aligned}$$

Example. *In Figure 7, and the sequential initial trace, the statement* s_4 *is the statement on line 4, the operation* $(s_4, (\{k_0, k_1\}, \{(3, 0)\}))$ *writes in cell* $c(1)$ *and needs to read* $c(0)$ *which was last wrote by* $(s_1, (\{k_0\}, \{0\}))$.

Definition 4.4 (Direct Data Dependencies)**.** Let $o_2 = (s_2, t_2)$ be an operation, o_2 directly depends on operation $o_1 = (s_1, t_1)$ if there exists $v \in \texttt{rvars}(o_2) \cup \texttt{wvars}(o_2)$ such that $o_1 \in \texttt{last}_o(v)$. It is denoted by $o_1 \rightsquigarrow o_2$.

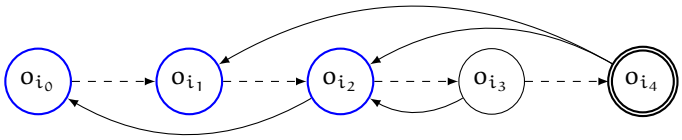
Definition 4.5 (Most Recent Direct Data Dependencies)**.** Let $o_2 = (s_2, t_2)$ be an operation, and D the set of operations on which o_2 directly depends. The most recent operation on which o_2 depends is $o_1 = \max_{<} D$. It is denoted by $o_1 \twoheadrightarrow o_2$.

Example (An operation with two direct dependencies)**.** *In the following case, the direct dependencies of* o_{i_4} *are* o_{i_1} *and* o_{i_2}*. And the most recent dependency is* o_{i_2}.



Definition 4.6 (Data Dependencies)**.** Operation o_2 depends on operation o_1 iff $o_1 \rightsquigarrow^+ o_2$ where $\rightsquigarrow^+$ is the transitive closure of $\rightsquigarrow$.

Example (An operation with three dependencies)**.** *In the following case, the direct dependencies of* o_{i_4} *are* o_{i_1} *and* o_{i_2}*. And* o_1 *is an indirect dependency.*



4.2 Equivalence with [Fea91]

In this subsection we will prove that, when considering regular programs with respect to the polyhedral model, our approach is strictly equivalent to the one presented in [Fea91]. Let P be a regular program and P' the same program rewritten with `while` loops in the most straightforward fashion. That is,

```
for i from start to finish
  (* ... *)
done
```

is rewritten to the following "pseudo polyhedral" program:

```
i := start;
while i <= finish
  (* ... *)
  i = i + 1
done
```

Prop 1. *Let* $o_1 = (s_1, t_1)$ *and* $o_2 = (s_2, t_2)$ *be two operations in an initial trace* O *then,* $o_1 \twoheadrightarrow o_2 \Leftrightarrow K_{s_1,s_2}(\texttt{convert}(t_2)) = \texttt{convert}(t_1)$ *where* `convert` *is the function that converts our iteration vector into the iteration vector introduced in [Fea91].*

Proof. The existence of the `convert` function will be assured by the proposition 3. In this proof, we will show that if $o_1 \rightsquigarrow o_2$ then the conditions C1, C2 and C3 are satisfied.

C1 : From the construction of $\twoheadrightarrow$, we have the guarantee that o_1 produces a value for o_2 or wrote the same cell as o_2. Hence, the cells that are accessed match ;

C2 : The definition of the function `last` guarantees that o_1 happens before o_1 ;

C3 : o_1 belongs to the initial trace, therefore, the statement s_1 happens during a valid iteration.

Moreover, the definition of `last` guarantees that o_1 is the last operation before o_2 that produces a value for o_2 or writes the same cell as o_2. Therefore, o_1 is the last operation on which o_2 depends. Hence, $o_1 \twoheadrightarrow o_2 \Leftrightarrow K_{s_1,s_2}(\texttt{convert}(t_2)) = \texttt{convert}(t_1)$ □

Prop 2. *Let $o_1 = (s_1, t_1)$ and $o_2 = (s_2, t_2)$ be two operations then, $o_1 \twoheadrightarrow^+ o_2 \Leftrightarrow \texttt{convert}(t_1) \in Q_{s_1,s_2}(\texttt{convert}(t_2))$ where `convert` is the function that converts our iteration vector into the iteration vector introduced in [Fea91].*

Proof. The same proof as for proposition 1 holds. The only difference is that since we take all direct dependences and the transitive closure we indeed get all the dependencies. □

This equivalence proves that our formalization includes the polyhedral model and in this case (for loops rewritten as while loops) our system can harness the classical polyhedral computations. We thus reached our first goal, which is to be able to *semantically* capture the key notion of *dependency* and being able to compute it.

Example. *Let's compute the function `convert` on the "Array filling example" of Figure 7, which we recall here:*

```
  k0 := 0;
1 c(0) := 0;
  k0 := k0 + 1;
2 i := 1;
  k0 := k0 + 1;
  k1 := 0;
3 while i <= n do
4   c(i) = c(i-1) + 1;
    k1 := k1 + 1;
5   i := i + 1;
    k1 := k1 + 1
6 done;
  k0 := k0 + 1
```

Once in the `while` on line 3 , the value of k_0 is fixed forever at 2. Indeed, the last time where k_0 is increased has no effect whatsoever since it is the last operation of the program. Then, we have to compute the relation between i and k_1. i starts at 1 whereas k_1 starts at 0. At s_1, k_0 is first, 0, then 2, then 4 and so on. The gap between two values is 2: that is the number of statements (without taking account of statements added by the annotation phase) in the loop. Thus, $k_1 = 1 + 2 * (i - 1)$. Hence,

$$\texttt{convert}(k_0, k_1) := 1 + 2 * (i - 1)) = (i)$$

The decision process of finding the set of dependencies of a given program thus relies to the ability of effectively computing this `convert` function. We are thus searching for a relation between the variables of the program which implies a *one-to-one* relation between the iteration vector and the indices of array accesses.

There is a plethoric literature on invariant generation for general imperative programs (a survey can be found in [GS14]) and the computation of transitive closures of numerical relations.

In the general case, the transitive closure of an affine relation is not computable, however, there exists subclasses that are known to be exactly computable. The paper [VCB11] propose an algorithm that compute over-approximations of transitive closures of *quasi-affine* relations (a more general family of relations that encompass affine relations). Moreover, it also returns a Boolean value that says that this transitive closure is exact.

Prop 3. *If the relation is a translation, its transitive closure is computable.*

Proof. For instance, [VCB11]. □

Thus, as long as we are dealing with regular polyhedral programs our model is decidable because our notions as well as those in [Fea91] coincide.

Remark. Proposition 1, 2 and 3 give us a decision procedure to test dependency between two given operations for our model. However, in the case where `convert` is invertible we not only have a decision procedure but the full symbolic graph of dependencies. Since, in our setting of this section, `convert` is a translation it is invertible, the symbolic graph of dependencies can also be expressed, computed and stored when we analyse a pseudo polyhedral program with while loops.

5 Extensions

From this section onward, we present the directions that we are currently investigating. Therefore, the tone will be more informal than the previous sections.

5.1 "Covertly regular" loops with scalar and arrays

In the previous section, we addressed the case of polyhedral loops that have been straightforwardly transformed into while loops. In practice, state-of-the-art production compilers like gcc or `LLVM` can perform a lot of structural transformations on a given program while parsing and optimizing. Even in the case of initial syntactic polyhedral loops, a polyhedral-based dependency analyses may struggle to find whether or not to apply, and may miss optimisation opportunities.

However, the `CFG` that is obtained is equivalent to the initial one, which means that an invariant generator like ISL [VCB11] should be able to compute a relation that contains exact ranges of the iteration vectors. Conditions C1, C2 and C3 are thus replaced by the same conditions where the validity of a given operation is replaced by is "reachability" condition (invariant at this point). The result in this case should thus be a set of constraints that correspond to the *exact set* of dependencies of the program.

5.2 General (affine or non affine) loops

General loops are loops for which the transition relation is not a simple translation, but a general (affine or non affine) relation between the program variables. For this more general case, the transitive closure of the loop is generally not computable.

In that particular case, a loop invariant generator will give us an overapproximation of the behavior of the scalar variables of the program (in a shape we have to define, we most probably will compute a polyhedron). We thus will have to take into account this approximation and define the notion of *false dependencies*. This is not a very hard issue, but it will have an impact on future formalization of the polyhedral model framework, since it as an impact on code generation. For instance, a given computation should not wait forever a data it doesn't really depend on.

5.3 Lists

This section will present how it is possible to extend the polyhedral model to a new, dynamic, data structure: lists.

By definition, lists, unlike arrays, are dynamic data structures. Hence, in addition to losing iteration variables when dealing with while loops, we lose the straightforward canonical representations of memory cells. Indeed, an array has a fixed size and all its cells are contiguous in memory which is not the case with lists.

As a first step, we will address a special case of list, those where *it is only possible to insert an element at the beginning*. Such lists present the same behavioral properties as dynamic arrays such as C++'s vectors. It is important to note that dynamic arrays are already out of the scope of the polyhedral model since their size is not fixed.

Up until now we have been dealing with arrays. Hence it may seem strange that instead of addressing the case of dynamic arrays which is an extension of standard arrays, we have decided to address the case of lists with insertion at the beginning. The reason is two-fold. First their behavior with respect to insertion (from the user's viewpoint) is the same as dynamic arrays, the only difference are that the new cell is appended at the beginning of the list rather than at the end of the dynamic array and that there is no built-in index on list cells. Second, when we will want to add insertion in the middle as an operation, lists will outperform dynamic arrays. Indeed, in order to add a value in the middle of a dynamic array there is a need to shift values, whereas adding a value in the middle to a list does not have this trade-off.

5.4 On giving numbers to list cells

Unlike arrays which have a built-in index, list cells does not have that feature and the only way to talk about a list cell is to know its address in memory. However, this is not very practical to talk about dependencies in a setting where memory addresses should not be exposed. Therefore we introduce a way to index list cells.

A good numbering on list cells should assign each cell a number that should never change. In order to do that, the cell number id is given by its distance to the `nil` list. The listing in Figure 10 creates the lists as depicted in Figure 11.

```
1 a = nil;
2 cons("", list(a));
3 cons("", list(a));
4 list(b) = list(a);
5 nxt(list(b));
6 cons("", list(b));
```

Figure 10: Two lists with a shared tail

Figure 11 does not only illustrate the numbering of cells but also the fact that we have to deal with aliasing.

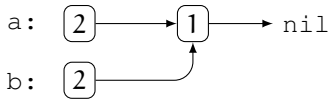


Figure 11: Two lists with a shared tail labeled with their cell number

5.5 On dealing with aliasing

Dealing with aliasing is mandatory to be able to analyze ans optimize lists properly. Indeed, even for something as simple as traversing a list twice we have to be able to save the head of the list because it is not possible to rewind a linked list. This is left for future work.

6 Conclusion

One of our prime goal was to formalize the polyhedral model so as to be able to use it without its intrinsic limitations due to the fact that it relies entirely on syntax. We addressed those limitations by identifying the hypotheses within the polyhedral framework when computing dependencies as described in [Fea91]. The hypothesis made by [Fea91] have been all taken into account when designing our mini-language, however we have relaxed some hypotheses like those concerning `for` loops and we have made room for extension because we keep optimizations on lists in sight.

This semantics of the mini-language is defined above an execution environment which makes explicit the implicit hypotheses that were assummed in the original paper. The semantics allows to define a notion of trace that is the central for all our definitions and for making the link between the framework presented in [Fea91] and ours.

We will now focus on handling correctly programs with general polyhederal loops and lists.

A An implementation in Prolog

This appendix exposes an implementation in `PROLOG` of the mini-language. This tool is used in order extract the relations of dependency between operations on programs which terminate [6]. In the case of the polyhedral model only iteration variables matter, and we can safely forget about what the program actually computes. Therefore, a tool that would analyse dependencies only need to keep track of which operations wrote which memory cell. However, since we would like to be able to handle access of the form `a[b[i]] := ...` in the future we still keep a very loose approximation of the values computed by the actual program, this approximation could be made more precise if needed in the future.

[6] and was also used to debug and validate our semantics.

A.1 Details About the Memory Model

Since we want to store the values of the computation, we need to make sure that the memory model guarantees that there is no risk whatsoever that to variables overlap in memory. Hence, we require that every object is defined beforehand with its name and size.

All variables are global, each variable is alive and accessible as soon as it has been defined. All variables are released at the end of the program, however it is possible to undeclare a name to reuse it later. In order to keep the memory model as straightforward as possible memory is never freed, that means that even if a variable is undeclared the memory used for it will not be freed. This allows ignoring problems arising as soon as the memory is fragmented.

A.2 Practical Implementation in Prolog

The implementation is divided into several files each implementing a logical phase of the analysis: `state.pl`, `annotation.pl`, `eval.pl`, `exec.pl`, `dot.pl` and `utils.pl`.

`state.pl` A state is the conjunction of 4 components:

- `Reg`: The address of the next unused memory block. Since memory is never freed, this integer can only increase.
- `Vec`: The current value of the iteration vector.
- `Loc`: A map from variable names to address. That means that variables are in fact references.
- `Mem`: A map from the memory address space to the values held in memory. Alongside values is stored the value of the iteration vector which wrote the current value held in the memory cell.

The other definitions in this file are all about how commands affect state. The predicates defined by itself are already sufficient to write programs but since the state is still exposed to the user and there are no annotations at this stage hence the iteration vector is not available.

The annotation part does exactly what the Section 3.1.3 explains but it also performs the declaration/undeclaration of iteration variables when needed.

The evaluation externalises the computations happening in the program in an external file for convenience because it is used both by the dot phase and the regular execution phase.

The regular execution phase is in the `eval.pl` file and implements the mini-language completely. If a program terminates, then the last state is returned.

On the other hand, the `dot.pl` file needs to be able to construct the dependency diagram hence it has to store all the operations in order to reconstruct the diagram.

A.3 Dependencies in the Finite Case

When the program terminates it is possible to use the interface exposed by the dot module, to obtain the list of operations happening in the programs as well as extracting the dependency graph such as the following one. The snake like dependencies represents loop counters and as expected they always depend on the last time they were assigned. The full dependency graph is available in pdf format as an appendix.

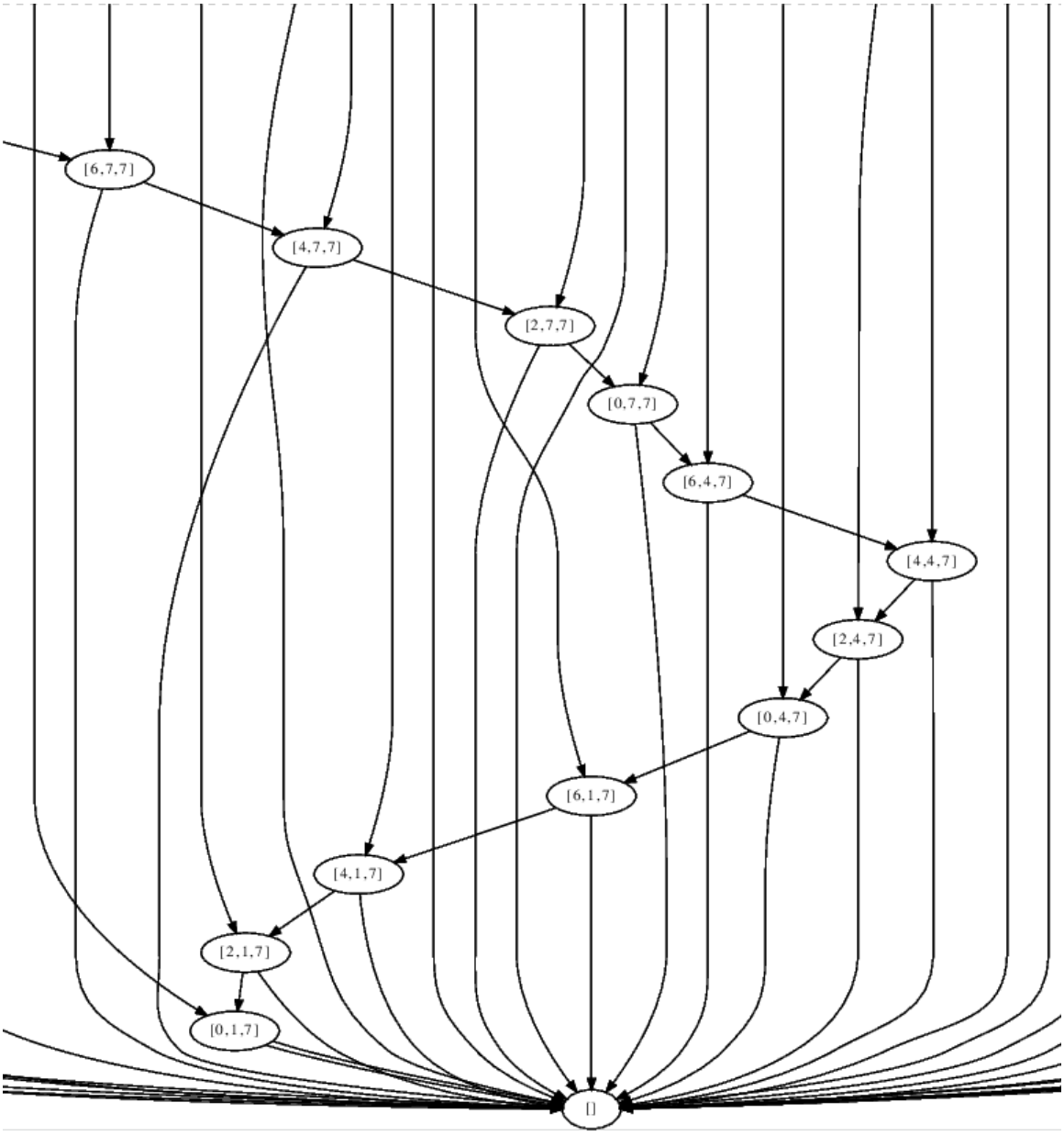


Figure 12: A part of the dependency graph of program performing a 4x4 matrices multiplication

References

[ADFG10] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *Proceedings of the 17th International Conference on Static Analysis*, SAS '10, pages 117–133, 2010.

[BHRS08] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, 2008.

[BYR+11] Vamshi Basupalli, Tomofumi Yuki, Sanjay Rajopadhye, Antoine Morvan, Steven Derrien, Patrice Quinton, and Dave Wonnacott. ompVerify: Polyhedral analysis for the OpenMP programmer. In *Proceedings of the 7th International Workshop on OpenMP*, IWOMP '11, pages 37–53, June 2011.

[DI15] Alain Darte and Alexandre Isoard. Exact and approximated data-reuse optimizations for tiling with parametric sizes. In *Proceedings of the 24th International Conference on Compiler Construction*, CC '15, pages 151–170, April 2015.

[DSV05] Alain Darte, Robert Schreiber, and Gilles Villard. Lattice-based memory allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, 2005.

[Fea88] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Operationnelle*, 22, 09 1988.

[Fea91] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[Fea92a] Paul Feautrier. Some efficient solutions to the affine scheduling problem, I, one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, October 1992.

[Fea92b] Paul Feautrier. Some efficient solutions to the affine scheduling problem, II, multi-dimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992.

[FL11] Paul Feautrier and Christian Lengauer. The polyhedron model. In David Padua, editor, *Encyclopedia of Parallel Programming*. Springer, 2011.

[GS14] Laure Gonnord and Peter Schrammel. Abstract Acceleration in Linear Relation Analysis. *Science of Computer Programming*, 93, part B(125 - 153):125 – 153, 2014. Author version : http://hal.inria.fr/hal-00787212/en.

[HAMM14] Julien Henry, Mihail Asavoae, David Monniaux, and Claire Maiza. How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics. In Youtao Zhang and Prasad Kulkarni, editors, *LCTES*, pages 43–52. ACM, 2014.

[VCB11] Sven Verdoolaege, Albert Cohen, and Anna Beletska. Transitive Closures of Affine Integer Tuple Relations and their Overapproximations. In Eran Yahav, editor, *SAS 2011 - The 18th International Static Analysis Symposium*, volume 6887 of *LNCS*, pages 216–232, Venice, Italy, September 2011. Springer.

[YFRS13] Tomofumi Yuki, Paul Feautrier, Sanjay Rajopadhye, and Vijay Saraswat. Array dataflow analysis for polyhedral X10 programs. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 23–34, February 2013.