Semantic Array Dataflow Analysis

Paul Iannetta, Laure Gonnord, Lionel Morel, Tomofumi Yuki

Semantic Array Dataflow Analysis

Paul IANNETTA, Laure GONNORD, Lionel MOREL, Tomofumi Yuki

RESEARCH REPORT
N° 9232
December 2018
Project-Team CASH

ISSN 0249-6399 ISRN INRIA/RR--9232--FR+ENG

Semantic Array Dataflow Analysis

Paul IANNETTA[*], Laure GONNORD[†], Lionel MOREL [‡],

Tomofumi Yuki [§]

Project-Team CASH

Research Report n° 9232 — December 2018 — 22 pages

Abstract: This paper revisits the polyhedral model's key analysis, dependency analysis. The semantic formulation we propose allows a new definition of the notion of dependency and the computation of the dependency set.
We argue that this new formalization will later allow for a new vision of the polyhedral model in terms of semantics, which will help us fully characterize its expressivity and applicability. We also believe that abstract semantics will be the key for designing an approximate abstract model in order to enhance the applicability of the polyhedral model.

Key-words: polyhedral model, operational semantics

[*] University of Lyon, LIP (UMR CNRS/ENS Lyon/UCB Lyon1/INRIA), F-69000 Lyon, France
[†] University of Lyon, LIP (UMR CNRS/ENS Lyon/UCB Lyon1/INRIA), F-69000 Lyon, France
[‡] Univ Grenoble Alpes, CEA, F-38000 Grenoble, France
[§] Inria, Univ Rennes, CNRS, IRISA, F-35000 Rennes, France

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Semantic Array Dataflow Analysis

Résumé : Dans ce rapport de recherche nous réexprimons un des calculs-clefs du modèle polyédrique: le calcul des dépendances, en termes d'opérations sur une sémantique concrète impérative des programmes.

Mots-clés : modèle polyédrique, sémantique opérationnelle

1 Introduction

Multi-core processors, and parallel processing in general, are now broadly used. Their horizon of applications ranges from mobile platforms to high-performance computing. Allowing non-expert programmers to harness the parallelism in recent hardware require significant advances in the entire compilation chain. It also means that the general forms of sequential programs, *e.g.*, with `while` loops and data-dependent control structures, should be amenable to parallelization.

The polyhedral model [FL11] is a powerful algebraic framework that is at the core of many advances in optimization and code generation of numerical kernels. One reason of its success is the practicality of the operations that are expressed in terms of algebraic computations on affine sets.

One of the major limitations of this model is that it only applies to programs with regular control and loops with static bounds. The main issue is that the polyhedral model's algorithms are defined with strong assumptions on the *shape* of programs under analysis, which make the underlying problems decidable. However, checking those requirements is sometimes not trivial:

- A programmer may have written an algorithm that is inherently polyhedral in a way that is not compliant with the syntactic restrictions of the polyhedral model;
- The compiler may have transformed the polyhedral input program in such a way that the polyhedral structure does not appear syntactically any more.

We argue that this is an important limitation for further extending the polyhedral model. In this paper, we propose to redefine the dataflow analysis based on the operational semantics of programs. By doing so we claim that we leverage the constraints of the previous definitions and increase the range of applicability of the polyhedral model.

From a general viewpoint, our work lies in the semantic consolidation of the polyhedral model to allow its extension with static analysis techniques that opens the door to the parallelization of irregular programs (*i.e.*, with `while` loops and more general data structures such as trees or maps).

A strategic prerequisite for this long-term goal is to propose a unified formal setting that describes the semantics of general programs. As a first step for this work, the present paper proposes a semantic-based description of the array dataflow analysis [Fea91]. This paper also proposes a precise definition of *covertly regular programs* for which the classical algorithms of the polyhedral community are exactly applicable, and a notion of polyhedral approximation for more general programs.

Overview

This work is a first step towards a clear semantic of the polyhedral model. The contributions of the paper are:

- A semantic definition of the notion of dependency *à la polyhedral model*, on a general imperative language;

- A rephrasing of the classical array dataflow analysis [Fea91] in our setting, which enables us to recover classical results from the community;
- The definition of the notion of *covertly-regular* programs for which the polyhedral model algorithms and tools can be applied as-is;
- A notion of *approximated polyhedral model* for programs with non polyhedral control, on which we can compute an over-approximated set of dependence.

The rest of the paper is organized as follows: section 2 recalls the notion of data dependency set and its classical computation in the polyhedral model framework; section 3 describes our model of program and its semantics enriched with an extended notion of *iteration vector*; section 4 gives a semantic-based notion of dependencies for our general class of programs. Then, section 5 proves the equivalence of our definition with the initial one on *regular polyhedral* programs, enabling us to define the notion of *covertly regular polyhedral* programs. section 6 gives an algorithm to compute an over-approximation of the dependency set for programs with non-polyhedral control, opening perspectives to a more general *approximate polyhedral model*; section 7 compares our work to existing works. Finally, we conclude in section 8 with some directions for future work.

2 Background: Array Dataflow Analysis

The seminal paper *Array Dataflow Analysis* [Fea91] proposed exact dependency analysis for loops with static and affine control. In this section, we present an overview of the results and algorithms of this paper, rephrased with our semantic-based formalization in view.

Informally, a *data dependency* between two operations exists when two operations access to the same memory location, with at least one of them being a write. These include benign dependencies as well as *true dependencies*, which are defined based on the notion of *most recent write*. The paper shows that for affine loops the above can be formulated with Integer Linear Programming, hence providing an exact solution, which we recall below.

2.1 Tracking Operations

In order to optimize the operations of a given program, a typical polyhedral compilation flow computes the instance-wise and element-wise dependencies. The analysis identifies dependencies between all operations operating on array as long as they are within the affine restrictions. Two operations that do not depend on each other can be parallelized, or at least, rescheduled in an order different than the lexical order of the original program. The notion of dependency between operations is thus central to program transformations.

Those dependencies are expressed by giving a unique identifier to each operation – an *iteration vector* – whose coordinates are loop counters. The first coordinate is the outermost loop's counter while the last coordinate is the innermost loop's counter. For example, in the listing of Figure 1 the iteration vector on line 3 is $\langle i, j \rangle$ while the iteration vector on line 5 is $\langle i, j, k \rangle$.

Remark (loop counter). In the context of `for` loops, the concept of iteration variable is crystal clear since it is the same as loop counters. However, when dealing with `while` loops, the definition is not as clear and is addressed later.

```
(* a and b are n-n matrices and c = ab *)
for i from 1 to n
 for j from 1 to n
  c[i, j] := 0 (* s1 *)
  for k from 1 to n
   c[i, j] := c[i, j] + a[i, k] * b[k, j] (* s2 *)
```

Figure 1: Product of matrices with a for loop.

From now on, *statements* are denoted by s_i, *instantiated iteration vectors* by t_j (because such a vector can be seen as a *timestamp*) and *operations* by a pair $\langle s_i, t_j \rangle$. Note that two operations can have the same iteration vector, typically when 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.

2.2 Computation of Dependencies

We now formally define K and Q, and explain how they are computed within the context of the polyhedral model. Let us assume that we are computing values for a matrix M, and that we want to compute the operations on which $o_2 = \langle s_2, t_2 \rangle$ (an operation that needs to read values in M) depends. Moreover, let us assume 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 $M[f(t_1)] :=$ ... where f is an affine access function of the iteration vector. The research can be restricted to operations that *happen before* s_2 in the program flow. The operations on which s_2 depend will then be the union of the operations found with s_1 as their statement.

In order to explicitly define and compute Q, the following conditions have to be fulfilled:

C_1: the arrays/matrices cells that s_1 and s_2 try to access should match: $f(t_1) = g(t_2)$;

C_2: (s_1, t_1) should happen before (s_2, t_2) (*i.e.*, $t_1 \lhd t_2$, or $((t_1 = t_2) \wedge T_{s_1,s_2})$ where $\lhd$ is the lexicographic ordering on vectors and T the textual ordering). This condition is denoted by $\langle s_1, t_1 \rangle \prec \langle s_2, t_2 \rangle$

C_3 t_1 must be a valid iteration (denoted as $e(t_1) \geq 0$, this notation will become clear in Theorem 2.2.)

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

$$\{ t' \mid f(t') = g(t),\ \langle s_1,\ t' \rangle \prec \langle s_2, t \rangle,\ e(t') \geq 0 \}$$

and $K_{s_1,s_2}(t_2)$ as:

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

Theorem 1 (Dependencies are computable in the polyhedral model)**.** *In the polyhedral model setting (static control), the set of dependencies of a given operation is computable.*

Proof. The proof can be found in the original paper [Fea91]. The 3 conditions above lead to a system of affine constraints whose lexicographic maximum is then computable by a Parametrized Integer Linear Programming solver (such as PIP [Fea88]). □

Example (Computations of dependencies for the matrix product, shown in Figure 1)**.** *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 us 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} is empty.

Now, let us compute Q_{s_1,s_2}. Let $\langle i_1, j_1\rangle$ be the iteration vector of statement s_1 and $\langle i_2, j_2, k_2\rangle$ the iteration vector of statement s_2. We can then express C_1, C_2 and C_3 as affine conditions. C_1 is $\langle i_1, j_1\rangle = \langle i_2, j_2\rangle$. C_2 is $\langle i_1, j_1\rangle \lhd \langle i_2, j_2, k_2\rangle$. And C_3 is $1 \leq i, j \leq n$. This leads to $Q(s_1, s_2)(\langle i_2, j_2, k_2\rangle)$ being equal to:

$$\{ \langle i_1, j_1\rangle \mid i_1 = i_2 \wedge j_1 = j_2 \}.$$

Lastly, let us compute Q_{s_2,s_2}. Let $\langle i_2, j_2, k_2\rangle$ and $\langle i_2', j_2', k_2'\rangle$ be the iteration vectors of statement s_2 at two distinct instants. All conditions can be expressed as affine conditions: C_1 is $\langle i_2, j_2, k_2\rangle = \langle i_2', j_2', k_2'\rangle$, C_2 is $\langle i_2, j_2, k_2\rangle \lhd \langle i_2', j_2', k_2'\rangle$, and C_3 is $1 \leq i_2, j_2, k_2 \leq n$. This gives $Q(s_2, s_2)(\langle i_2', j_2', k_2'\rangle)$ as:

$$\{ \langle i_2, j_2, k_2\rangle \mid i_2 = i_2' \wedge j_2 = j_2' \wedge k_2 < k_2' \}.$$

2.3 Discussion

The analysis led to an efficient algorithm to store and compute (most recent) dependencies when:

- Loop iterators are easily definable and their domain is easily exactly computable (condition C3);
- Memory accesses are affine functions of loop iterators (condition C1);
- The happens-before relation is a function of syntax elements (condition C2).

What we propose in this paper is to relax these assumptions to rely less on syntactic elements, and to re-formulate the analysis based on an operational semantics of the language.

3 General Imperative Programs With Iteration Vectors

For the formalization, we use a variant of the classical (small-steps) operational semantics of a general imperative language with scalars and arrays, where we exhibit the notion of iteration vector. The syntax of the mini-language is depicted in subsection 3.1, our extension for iteration vectors in subsection 3.2. The semantics described in subsection 3.3 then enables us to properly define the notion of trace in subsection 3.4.

3.1 A Mini Language

The language we propose is a pointer-less imperative language with native support for `while` loops, `if` statements as well as arrays of integers (scalars are degenerated arrays with one cell).

In the grammar depicted in Figure 2, capital letters (X, Y, Z) are used as placeholders for variable names. n represents an element of $\mathbb{N}$ and terms in lowercase are an instance of the expression rule which shares the same first letter: *e.g.*, $a_.$ is an instance of Aexp, $b_.$ is an instance of Bexp, and so on. The "κ_n :" notation is explained in subsection 3.2 and can be safely ignored at this point.

$$
\begin{aligned}
\langle Aexp \rangle &::= n \mid a_0 \ \langle Aop \rangle \ a_1 \mid v_0 \\
\langle Aop \rangle &::= \text{`+'} \mid \text{`*'} \mid \text{`-'} \mid \text{`/'} \mid \text{`mod'} \\
\langle Bexp \rangle &::= \text{`true'} \mid \text{`false'} \mid !(b_0) \\
&\mid \ b_0 \ \langle Bop \rangle \ b_1 \mid a_0 \ \langle Cop \rangle \ a_1 \\
\langle Bop \rangle &::= \text{`or'} \mid \text{`and'} \mid \text{`='} \\
\langle Cop \rangle &::= \text{`<='} \mid \text{`=>'} \mid \text{`<>'} \mid \text{`=='} \\
\langle Vexp \rangle &::= X \mid X\text{`['}a_0\text{`]'} \\
\langle Sexp \rangle &::= \kappa_n \ \text{`:begin'} \mid \text{`skip'} \mid s_0 \ \text{`;'} \ s_1 \\
&\mid \ \kappa_n \ \text{`:if'} \ b_0 \ \text{`then'} \ s_0 \ \text{`else'} \ s_1 \ \text{`fi'} \\
&\mid \ \kappa_n \ \text{`:while'} \ b_0 \ \text{`do'} \ s_0 \ \text{`done'} \\
&\mid \ v_0 \ \text{`:='} \ a_0
\end{aligned}
$$

Figure 2: Our Mini-Language: syntax

The grammar itself is permissive and recognizes programs that are syntactically outside of the scope addressed by the polyhedral model.

3.2 Semantic Extension: Iteration Variables and Iteration Vectors for our Language

In the classical polyhedral model, `for` loops naturally introduce counter variables. These are convenient to number the operations and use those as labels when investigating their dependencies. For general programs with tests and `while` loops, there is no canonical way to implicitly define iteration variables. We thus explicitly introduce them in our language and semantics. This kind of instrumentation is classic in other static program analyses such as Worst-Case Execution Time (WCET) analysis [MRPV+17] or complexity estimation [GMC09].

Fresh iteration variables $\kappa_i \in \texttt{Name}$ 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 those variables. The leftmost coordinate is the iteration variable of the outermost loop and the rightmost coordinate is the iteration variable of the innermost loop. This allows sorting operations by their iteration vector, with respect to the lexicographic order. We illustrate this process with an example in the

following.
Example. *Figure 3 depicts a simple array filling procedure with a* `while` *loop. We annotated each control statement with a* κ_i. *These iteration variables are introduced so as to number the operations hierarchically.*

```
c[0] := 0;
i := 1;
while i <= n do
   c[i] := c[i-1]
+ 1;
   i := i + 1
done
```

a) Before annotation

```
κ0:begin
c[0] := 0;
i := 1;
κ1:while i <= n do
 c[i] := c[i-1] +
1;
 i := i + 1;
done
```

b) After annotation

Figure 3: Array filling with increasing values

As for `if` statements, we have to do some extra work in order to make them compatible with the lexicographic order. In the annotation step, we only annotate the test itself, the actual numbering of the sub-statements will be performed in the semantic rules, as we will later see in Figure 7.
Example. *Figure 4 shows an example of an* `if` *branch annotation. Only the test itself is annotated.*

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

a) Before annotation

```
1 κ0:begin
2 i := 5;
3 κ1:while i <> 1
4   κ2:if i mod 2 ==
 0:
5      i := i / 2;
6  else:
7     i := 3 * i+1
8 done
```

b) After annotation

Figure 4: The Syracuse algorithm

For example, on line 4 *of* 4b *the (uninstantiated) iteration vector is* $(\kappa_0\ \kappa_1\ \kappa_2)$. *When the program is run, on line* 4*, the iteration vector will contain the current values of* κ_0, κ_1 *and* κ_2. *At the end the iteration vector is* $[\langle\kappa_0, 3\rangle]$, *because* κ_2 *has been dropped at the end of the* `if` *and* κ_1 *as been dropped at the end of the* `while`.

The annotation process is straightforward as it appends κ_n just before the construct that goes on one step deeper, and adds a `begin` annotation with label κ_0 at the beginning of the program. The semantics described in subsection 3.3 takes such an annotated program as input.
Remark. Our annotation system is different form the usual notation used in the polyhedral model (the $2n+1$ *notation* [Bas04]), which may use two dimensions to represent one loop: one dimension that corresponds to the number of iteration of the loop and one that would number the internal statements.

However, the reason behind the fact that we use only one dimension is that we want to be able to map each level of the loop nest to a coordinate of the iteration vector.

3.3 Execution Environment, Final Semantics of our Mini Language

We now present the semantic rules of our annotated program where the initial statement, and each `if` and `while` statements have been prefixed with new fresh variables that constitute our *iteration vector*.

In our semantics, states σ are composed of:

- An environment μ that maps variables to values as well as the last iteration vector (instance) that wrote this variable: $\mu : \texttt{Vars} \to \mathbb{Z} \times (\texttt{Name} \times \mathbb{Z})^n$;
- The current value of the iteration vector $\vec{\kappa} \in (\texttt{Name} \times \mathbb{Z})^n$.

Remark. The `Name` part of the iteration vector is here to handle imperfect loop nests and in particular it is used to tell $[\langle\kappa_0, 3\rangle, \langle\kappa_1, 1\rangle]$ and $[\langle\kappa_0, 3\rangle, \langle\kappa_2, 1\rangle]$ apart.

The effect of each statement (in Sexp) is to update the current μ according to classical small-steps operational semantics while storing the current value of the iteration vector ; and to update the current iteration vector. We use two auxiliary functions upd and inc. Let $\sigma = (\mu, \vec{\kappa})$, then:

- $\mathrm{upd}(\sigma, \kappa_i, n)$ returns a copy of σ where κ_i is appended to $\vec{\kappa}$ and set to n if κ_i is not already a component of $\vec{\kappa}$, otherwise does nothing.
- $\mathrm{incr}(\sigma)$ returns a copy of σ where the current iteration vector $\vec{\kappa}$'s rightmost component of the iteration vector has been incremented by one.

We also define $\sigma \setminus n$ that removes the component n of the current iteration vector, if it exists.

Figure 5 depicts the semantic rules for basic statements.

$$\textsc{Skip}\ \frac{}{\langle\sigma, \mathrm{skip}\rangle}$$

$$\textsc{Begin}\ \frac{}{\langle\sigma, \kappa_0 : \mathrm{begin}; s\rangle \to \langle\mathrm{upd}(\sigma, \kappa_0, 0), s\rangle}$$

$$\textsc{Assign}\ \frac{}{\langle\sigma, v := e\ ;\ s\rangle \to \langle\mathrm{incr}(\sigma[v := e]), s\rangle}$$

Figure 5: Our semantics 1/3

Example. *After the first statement of the program of 4b (begin), the state is* $([], [\langle\kappa_0, 0\rangle])$. *The initialization of* i *gives the new state:* $([i \mapsto (5, [\langle\kappa_0, 0\rangle])], [\langle\kappa_0, 1\rangle])$.

For `while` loops, the semantics also mimics the initialization of its counter to 0 the first time we enter the loop, its incrementation at the end of one body execution; and also the removal of this counter at the end of the loop ($\sigma \setminus \kappa_n$ *removes* κ_n). Figure 6 depicts these two rules. Let us recall that in a small-steps semantics, $\to^+$ depicts the execution of the body of the loop.

For `if`s, we consider that we have a built-in length function that tells us the number of sub-statements contained in a statement s. The "true" part of the test rule is constructed so that the outermost component of the iteration vector grows from $-\mathrm{length}(c_1)$ to -1; the "false" part

$$\text{WHT}\ \frac{\langle \sigma, b_0 \rangle \to \text{true} \qquad \langle \text{upd}(\sigma, \kappa_n, 0), s_1 \rangle \to^+ \langle \sigma', \text{skip} \rangle}{\langle \sigma, \kappa_n : \text{while } b_0 \text{ do } s_1 \text{ done ; } s \rangle \to \quad \langle \text{incr}(\sigma'), \kappa_n : \text{while } b_0 \text{ do } s_1 \text{ done ; } s \rangle}$$

$$\text{WHF}\ \frac{\langle \sigma, b_0 \rangle \to \text{false}}{\langle \sigma, \kappa_n : \text{while } b_0 \text{ do } s_1 \text{ done ; } s \rangle \to \langle \text{incr}(\sigma \setminus \kappa_n), s \rangle}$$

Figure 6: Our semantics 2/3 (while)

makes it grow from 0 to $\text{length}(c_2) - 1$. Our operations continue to be uniquely numbered. Figure 7 depicts these two rules.

$$\text{IT}\ \frac{\langle \sigma, b_0 \rangle \to \text{true} \qquad \langle \text{upd}(\sigma, \kappa_n, -\text{length}(s_1)), s_1 \rangle \to^+ \langle \sigma', \text{skip} \rangle}{\langle \sigma, \kappa_n : \text{if } b_0 \text{ then } s_1 \text{ else } s_2 \text{ fi}; s \rangle \to \langle \text{incr}(\sigma' \setminus \kappa_n); s \rangle}$$

$$\text{IF}\ \frac{\langle \sigma, b_0 \rangle \to \text{false} \qquad \langle \text{upd}(\sigma, \kappa_n, 0), s_2 \rangle \to^+ \langle \sigma', \text{skip} \rangle}{\langle \sigma, \kappa_n : \text{if } b_0 \text{ then } s_1 \text{ else } s_2 \text{ fi}; s \rangle \to \langle \text{incr}(\sigma' \setminus \kappa_n), s \rangle}$$

Figure 7: Our semantics 3/3 (if)

3.4 Traces

We define traces of general programs based on our semantics.

Definition 3.1 (trace on states)**.** A *trace on states* Σ is a sequence of pairs of the form (state, statement) $\langle \sigma_0, c_0 \rangle \to \langle \sigma_1, c_1 \rangle \to \ldots$. An *initial trace* is a trace which begins from the empty state.

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 $\langle s, t \rangle$ 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 $o_1 \to o_2 \to \cdots$. An *initial trace* is a trace which begins from the trivial (empty) iteration vector.

Remark. From now on, the term *trace* will always refer to *trace on operations* unless stated otherwise.

Definition 3.3 (reachability/validity)**.** An operation $\langle s_1, t \rangle$ is *valid* if and only if there exists an initial trace $O = \{o_i\}_{i \in \mathbb{N}}$ such that there exists o_0 such that $o_0 = \langle s_1, t \rangle$.

Definition 3.4 (happens-before: $<$)**.** There exists a natural order $<$ on operations, called *happens-before*. $\langle s_1, t_1 \rangle < \langle s_2, t_2 \rangle$ if and only if there is one trace such that there exists i_1 and i_2 such that $o_{i_1} \to^+ o_{i_2}$ and $t_1 = \texttt{Vec}(o_{i_1})$ and $t_2 = \texttt{Vec}(o_{i_2})$ where $\texttt{Vec}(o_i)$ denotes the second component of the pair $o_i = \langle s_i, t_i \rangle$.

Prop 1 (strict order)**.** Happens-before *as defined above is a strict order.*

4 Dependencies for General Programs

The semantics allows defining the key notion of dependency in a general context. Our program traces now contain all elements to define the dependencies: a notion of *ordered time*, which is induced by the succession of states in a trace (including our iteration vectors), and all information to define a *last write* notion with respect to a given state or operation.

Definition 4.1 (rvars)**.** Let $o = \langle s, t\rangle$ 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 $o = \langle s, t\rangle$ 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* $o = (a[i] := a[i-1]+a[i]+1, t)$*. 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 content of the memory at this state). Then* $\texttt{rvars}(o) = \{\, a[0], a[1]\,\}$ *and* $\texttt{wvars}(o) = \{\, a[1]\,\}$*.*

Definition 4.3 (Last write)**.** Given an initial trace O and an operation o_{i_2} which belongs to O, the function $\texttt{last}$ returns the operation o_{i_1} (which belongs to O) that last wrote the cell containing the variable v before o_{i_2} reads it. The function $\texttt{last}$ satisfies the following formula:

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

Example. *In* 3b*, consider statement* s_4*, on line 4. The operation* $\langle s_4, [\langle \kappa_0, 2\rangle, \langle \kappa_1, 0\rangle]\rangle$ *writes in cell* $c[1]$ *and needs to read* $c[0]$ *which was last wrote by* $\langle s_1, [\langle \kappa_0, 0\rangle]\rangle$*.*

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

Definition 4.5 (Data Dependencies)**.** Operation o_2 depends on operation o_1 if and only if $o_1 \rightsquigarrow^+ o_2$, where $\rightsquigarrow^+$ is the transitive closure of $\rightsquigarrow$.

Definition 4.6 (Most Recent Direct Data Dependencies)**.** Let $o_2 = \langle s_2, t_2\rangle$ 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 $o_1 \twoheadrightarrow o_2$.

Prop 2. *The most recent dependency of an operation can be either computed from the set of all dependencies or from the set of direct dependencies. In other words,*

$$\max_{\lhd} \rightsquigarrow\; = \max_{\lhd} \rightsquigarrow^+$$

Proof. The paths appended by the transitive closure are all about iterations smaller (with respect to the lexicographic order) than the ones originally present in the relation $\rightsquigarrow$. Hence, the computation of the max on the transitive closure leads to the same result. □

Remark. The reason why we keep a clear distinction between direct and most recent direct data dependency is that the transitive closure of $\twoheadrightarrow$ is not the same as the transitive closure of $\rightsquigarrow$ (which is the full dependency graph), as can be seen in 4.6.

Remark. The notion of most recent dependency will parallel the definition of K (as defined in section 2) as it will be exposed in 3.

Example (Dependencies of an operation)**.** *Consider the sequence of operations* o_0 *to* o_5 *depicted in Figure 8. The sequentiality is represented with dashed arrows. The direct dependencies between these operations are represented with plain arrows.* o_5*, directly depends on* o_1 *and* o_3*, both being represented with simply dashed circles. The dotted circles denotes indirect data dependencies of* o_5*, obtained by the*

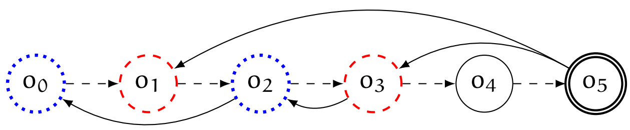

Figure 8: Direct (dashed) and indirect (dotted, obtained by transitive closure) data dependencies of operation o_5.

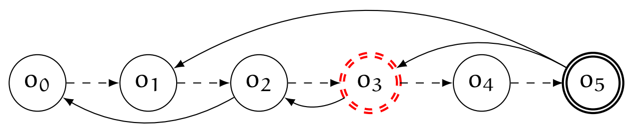

Figure 9: Most Recent Direct Data Dependency of o_5.

transitive closure $\rightsquigarrow^+$. *In Figure 9, the most recent direct dependence of* o_5 *is* o_3*, noted with double dashed red circle.*

5 Semantic-Driven Dependency Analysis

The semantic based reformulation of dependencies seamlessly extends the possibility for analysing programs that are not written as canonical affine loop nests. In this section, we first show that our formulation gives the same notion when the behavior of a program matches that of an affine loop nest, and express a new *Semantic-Driven Dependency Analysis* based on this result. Then we describe how our analysis applies to covertly-regular programs: programs that do not exactly correspond to affine loops, but still exhibit a regular behavior expressible within the polyhedral model.

5.1 Equivalence on Regular Polyhedral Programs

In this section we prove that, when considering regular programs with respect to the polyhedral model, our approach is strictly equivalent to the one presented in Feautrier's original paper [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
```

A complication at this point is that what has been presented here does not exactly match with the presentation given in the seminal paper [Fea91] on array dataflow analysis. Hence, we need to make a bridge between the loop counters used as iteration vectors in the original paper and our iteration vectors. The following example explains how this *convert* function is computed on a simple program.

Example. *Let's compute the function* `convert` *on the* Array filling example *of Figure 3, which we recall here:*

```
κ0:begin
c[0] := 0;                    (* s1 *)
i := 1;
κ1:while i <= n do
  c[i] := c[i-1] + 1;         (* s2 *)
  i := i + 1;
done
end
```

The key point is to express the relation which describes the transition between two consecutive iterations of the loop: that is the state of the loop at iteration k *and the state of the loop at iteration* $k+1$. *Let us denote* $\mathcal{R}$ *this relation. One execution of the loop content can be described as:*

$$(\kappa_0, \kappa_1, i)\ \mathcal{R}\ (\kappa_0, \kappa_1 + 2, i + 1).$$

The relation describes a transition in Presburger arithmetic, which means that its transitive closure is exactly computable. Moreover, we know that when the loop is initialized the following is true:

$$(\kappa_0, \kappa_1, i) \;=\; (0, 0, 1).$$

Hence, we can derive the exact expression of the `convert` *function in this example.*

$$\kappa_1 = 1 + 2(i - 1).$$

This function is the bridge that we want to create between our two approaches.

Prop 3. *Let* $o_1 = \langle s_1, t_1 \rangle$ *and* $o_2 = \langle s_2, t_2 \rangle$ *be two operations in an initial trace* O. *Then,*

$$o_1 \twoheadrightarrow o_2 \Leftrightarrow K_{s_1,s_2}(\mathtt{convert}(t_2)) = \mathtt{convert}(t_1)$$

where `convert` *is the function that converts our iteration vector into the iteration vector introduced in the original paper [Fea91].*

Proof. The existence of the `convert` function will be assured by 5. In this proof, we will show that if $o_1 \twoheadrightarrow o_2$ then the conditions C_1, C_2 and C_3 are satisfied.

We need to prove the two directions of the equivalence. Since similar arguments can be used for both directions we only prove the left-to-right direction.

C_1 **:** The construction of $\twoheadrightarrow$ guarantees that o_1 produces a value for o_2 or wrote the same cell as o_2. This means that the accesses in s_1 and s_2 are on the same cell. The index of that cell is an affine function of the loop counters, which is independent of the use of the `convert` function.

C_2 **:** The definition of the function `last` guarantees that o_1 happens before o_2. And since the `convert` function preserves the lexicographic order, we are sure that $\texttt{convert}(t_1) \lhd \texttt{convert}(t_2)$

C_3 **:** 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.

□

Example. *On the previous example, let* s_1 : `c[0]:=0` *and* s_2 : `c[i]:=c[i-1]+1`. *We search* t_1, t_2 *instantiations of* (κ_0, κ_1) *such that* $o_1 \twoheadrightarrow o_2$. *The systems of constraints is constructed with* $C_1 : 0 = i-1$, *and* $C_3 = \texttt{true}$ *(*s_1 *is always valid). For* C_2*, we need to transform the constraint* $(\kappa_0) \prec (\kappa_0, \kappa_1)$. *Since* κ_0 *has no equivalent in Feautrier's model* $\texttt{convert}(\kappa_0) = []$ *(the empty vector), hence,* C_2 *becomes* $[] < [1 + 2(i-1)]$, *which is true for all values of* i, *thus* $C_2 = \texttt{true}$. *Finally,* Q_{s_1,s_2} *has a unique constraint* $0 = i - 1$, *equivalent to* $i = 1$. *The maximum of this set,* K, *is also this unique point. After applying the convert function* $\kappa_1 = 1 + 2(i-1)$ *we finally get the final result* $\langle s_1, [\langle \kappa_0, 0\rangle] \rangle \twoheadrightarrow \langle s_2, [\langle \kappa_0, 0\rangle, \langle \kappa_1, 1\rangle] \rangle$.

Prop 4. *Let* $o_1 = \langle s_1, t_1\rangle$ *and* $o_2 = \langle s_2, t_2\rangle$ *be two operations. Then,*

$$o_1 \rightsquigarrow^+ o_2 \Leftrightarrow \texttt{convert}(t_1) \in Q_{s_1,s_2}(\texttt{convert}(t_2))$$

where `convert` *is the function that converts our iteration vector into the iteration vector introduced in the original paper [Fea91].*

Proof. The same proof as for 3 holds. The only difference is that since we take all direct dependencies 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.

The decision process of finding the set of dependencies of a given program thus relies on 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 an abundant literature on invariant generation for general imperative programs (a survey [GS14] is available on the subject), 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 sub-classes that are known to be exactly computable. There exist an algorithm [VCB11] 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 tells whether this transitive closure is exact.

Prop 5. *If a relation is a translation (from the point of view of Presburger arithmetic), its transitive closure is computable.*

Proof. For instance, the work by Verdoolaege et al. [VCB11]. □

Thus, as long as we are dealing with regular polyhedral programs our model is decidable because our notions as well as those in the original paper [Fea91] coincide.
Remark. 3 and 5 give us a decision procedure to test dependency between two given operations for our model. However, in the case where `convert` is invertible we do not only have a decision procedure but the full symbolic graph of dependencies. Since, in our setting of this section, it is invertible (`convert` is a translation), the symbolic graph of dependencies can also be expressed, computed and stored when we analyse a pseudo polyhedral program with `while` loops.

Once we have the relations between the artificial variables that were introduced and the variables appearing in the program we can express our iteration vector as linear combinations of the variables of the program.

5.2 Covertly-Regular Polyhedral Programs

Our analysis also extends to programs with `while` loops that are not straightforward translations of `for` loops. We are also interested in capturing programs with affine control that are not necessarily written as affine `while` loops. A possible example of such case is some kind of state machine with affine transitions. The programmer may decide to write such computation in a way that does not syntactically match affine loops, or a different pass in the compiler may strip away syntactic elements that are necessary to (syntactically) view them as affine loops.

Our analysis may be directly applied to such programs, and provide exact dependency information. However, the `convert` function that connects the iteration variables to syntactic elements can be difficult to find. For instance, an affine state machine with multiple variables – which may correspond to multi-dimensional loops – would require invariants involving polynomials in general.

We propose an algorithm that reintroduces structure to the programs such that computing the convert function becomes easier. We also give a precise characterization of the class of programs "equivalent" to polyhedral programs, which we call covertly-regular programs.
Prop 6. *A polyhedral program* (i.e., *a loop nest) can be represented as*

$$A\vec{\imath} + \vec{c} \leq 0$$

where A *is a lower-triangular matrix with ones on the diagonal,* $\vec{\imath}$ *is the vector whose coordinates are the loop counters, and* $\vec{c}$ *is a vector of constants expressions that may contain structure parameters.*

Proof. A polyhedral program is a `for` loop nest. Without loss of generality we can assume that all loop counters are lower-bounded by 0, if they were not we would apply a translation on the iteration range. Moreover, the upper-bound of a loop counter cannot be a function of loop counters deeper in the loop nest. Hence, the bound on the loop counters can be written as $A\vec{\imath}+\vec{c} \leq 0$ where A is a lower-triangular matrix with ones on the diagonal, $\vec{\imath}$ is the vector whose coordinates are the loop counters in the order as they appear in the loop nest, and $\vec{c}$ is a vector of constants.

□

From now on, we will denote a polyhedral program P by a triple $\langle A, \vec{i}, \vec{c} \rangle$.

Definition 5.1 (Covertly regular polyhedral program)**.** A covertly regular program $\langle A, \vec{i}, \vec{c} \rangle$ is such that there exists an orthogonal matrix O with $\det(O) = 1$, such that $\langle OAO^{-1}, O\vec{i}, O\vec{c} \rangle$ is a regular program.

Remark. The change of basis affect the whole program including arrays accesses.

Remark. Effectively computing such a base changing is in general undecidable. However, in the case of covertly regular programs, we might expect to be in practise able to decide if a given program is covertly regular or not, because transition matrices are in practice not too complex. Of course, such an affirmation needs to be experimentally validated. Such an experimentation is left for future work.

5.3 Conclusion

This section exposed how we could cover the exact computation of dependencies both in the regular and covertly regular cases in the case where the basis change is exactly computable. This is a first step to relax the initial syntactic restrictions of the polyhedral model and this enables a new definition of (covertly) regular programs on which the exact computation of dependencies is expressible and coincides with our new semantic definition of dependencies.

6 Approximate Polyhedral Model for General Imperative Programs

Irregular programs have complex control and non-affine accesses to arrays. In this section we propose to address the problem of non polyhedral control. Non affine accesses are left for future work.

In the previous sections, the key characterization is that the relation linking the program variables (including our annotation) was (in the favorable case) exactly computable. For general programs, we will rely on an over-approximation of this relation.

6.1 Dependence Analysis for Non-Affine Control

Let us recall the result of 3: if we are able to link our iteration vectors to the scalar variables of the program with an invertible `convert` function $\kappa_i = convert_i(i, j, k)$, then we can exactly compute the dependencies of a given program. A first remark is that instead of computing the sets K or Q with initial variables, we can equivalently write the equivalent equations on κ_i variables, and add the definition of `convert` as additional constraints, as is illustrated in subsection 6.1.

Example. *On the* Array filling example *we obtained* $C_1 : 0 = i - 1$, $C_2 : (\kappa_0) \prec (\kappa_0, \kappa_1)$, *and* $C_3 = true$. *Instead of replacing* κ_i *with their image by* `convert`, *we can solve the same constraint system augmented with the constraint* $\kappa_1 = 1 + 2(i - 1)$. *Now the set* Q'_{s_1,s_2} *is a polyhedron on* i *and the* κ_i *variables whose projection on* i *gives the same result* $Q_{s_1,s_2} = \{i = 0\}$ *as in 5.1.*

Now that we have work on general programs, we do not have a `convert` function any more, we thus compute an over-approximation of the relationship between the scalar variables of the programs and the κ_i variables.

Definition 6.1. Let $P_{s_1}(\vec{\imath}, \vec{\kappa})$ (resp. $P_{s_2}(\vec{\imath}, \vec{\kappa})$) be polyhedral invariants at statement s_1 (resp. s_2). Let us denote by $\mathrm{cons}(P)$ the set of constraints of P. Let us define $Q'_{s_1,s_2} = C_1 \cup C_2 \cup \mathrm{cons}(P_{s_1}) \cup \mathrm{cons}(P_{s_2})$ the union of C_1 and C_2 constraints and these over-approximations and $Q^\sharp$ is the projection on the κ_i variables.
Prop 7. *Let $o_1 = \langle s_1, t_1 \rangle$ and $o_2 = \langle s_2, t_2 \rangle$ be two operations in an initial trace O. Then,*

$$o_1 \rightsquigarrow^+ o_2 \Rightarrow t_2 \in Q^\sharp_{s_1,s_2}(t_1)$$

Proof. As $\mathrm{cons}(P_{s_1})$ is an over-approximation of C_3 (s_1 should be a valid iteration), and $\mathrm{cons}(P_{s_2}) \cup C_2$ is an over-approximation of C_2 (happens-before), all initial dependencies satisfy $Q^\sharp$. □

Remark. An important issue here is that we do not have any result about *the most recent dependence* (K) since computing the lexicographic maximum of $Q^\sharp$ may led to picking a spurious dependency.

As we already mentioned in subsection 5.1, computing polyhedral over-approximations can be done by various methods including abstract interpretation. The precision of our analysis will thus rely on the precision of the underlying invariant generator. In the case of regular or covertly regular programs, if the invariant generator gives us the most precise invariants, then we will recover the result of 3 and 4 (equivalence).
Example. *Let us consider the following example:*

```
c := 0
while ( i < 10 ) do
   if i > 10 then
       c := c + 1 (* s1 *)
   else
       c := c - 1 (* s2 *)
   i : = i +1
done
b := c   (* s3 *)
```

A good invariant generator would enable us to find that s_3 does not depend on any iteration of s_1 since its corresponding invariant is empty. However, any over-approximation P_{s_1} is safe, and we would find out that s_3 depends on s_1 for some values of scalar variables satisfying $\mathrm{cons}(P_{s_1})$, thus compute spurious dependencies.

6.2 Work in Progress: Dealing with Non-Affine Accesses

To deal with non-affine accesses, we might find inspiration from the recently proposed non-polyhedral dependency analysis [Fea15], which uses a variant of the Handelman's algorithm for solving multivariate polynomials. The abundant literature on linear relaxations of polynomials constraints has been recently been put to the attention of the program verification community that now uses it to compute over-approximations [MFK+16, RVS16] that could be useful for solving our non-linear set of constraints. However we should be careful about their complexity in practice.

7 Related Work

The array dataflow analysis [Fea91] and the Omega test [Pug91] proposed exact dependency analysis for loops with affine controls and array accesses. Both of these work rely on the ability to characterize the three conditions that define dependency (recall subsection 2.2) as affine functions of syntax elements in the source program—loop iterators. The semantics of the target language are abstracted away and are assumed to provide the required properties. In contrast, our work formulates the dependency analysis based on the semantics, and the connection to syntax elements is later established through the `convert` function or its approximation. This provides additional flexibility on how the programmer can express their computation. For instance, `while` loops that can be rewritten as `for` loops are seamlessly handled as we have shown in subsection 5.1.

Polly [GGL12] performs polyhedral optimizations to LLVM-IR, which is a low-level IR without high-level information such as loop iterators. The *semantic polyhedral regions* in a program are identified by searching for a *single induction variable* (with affine lower bounds and upper bounds) for each loop. Combined with additional analyses and transformations in LLVM, Polly can recognize program regions that are syntactically far from the canonical polyhedral loops in the original high-level specification. In the context of our work, the analysis in Polly can be viewed as a low-level version of our *covertly regular* loops detection without loop instrumentation. Our specificity is to fully characterize these semantically polyhedral loops and also to leave room for handling non-polyhedral programs through approximations of the convert function.

The original exact dependency analyses were later extended to expand the scope of the analysis, including `while` loops, non-affine `if` guards, and non-affine array accesses [BCF97, PW96, BPCB10, VNS13]. In Fuzzy Array Dataflow Analysis [BCF97], the non-affine conditions are expressed as predicates encoded with additional parameters, whereas the extension to the Omega test [PW96] express them with uninterpreted function symbols. Exact analysis is possible in some cases, but these extensions require runtime checks or over-approximations in general.

These extensions treat `while` loops as unbounded `for` loops with a predicate that defines the exit condition [BCF97, PW96, BPCB10]. The unbounded `for` loop uses an iterator that is not in the original program, which is analogous to the iteration variables in our work. There are two key differences: (i) we use iteration variables uniformly to both `for` and `while` loops, and (ii) we (attempt to) compute connections to syntax elements to express dependencies in terms of integer variables in the source program. For instance, the `while` loop in subsection 3.2 is viewed as:

```
c[0] := 0;
i := 1;
for t from 0
  c[i] := c[i-1] + 1;
  i := i + 1;
```

where the variable `i` is treated as data, and all array accesses are now data-dependent. Our work identifies the relation between `i` and `t`, linking the variable `i` to the iteration count of the `while` loop.

Alphabets [RGK11] is an equational language that can be viewed as an intermediate representation for polyhedral compilers. The language supports `while` loops in a manner similar to other work [BCF97, PW96, BPCB10]: unbounded domain with exit condition. The crucial difference[1] is that the dependencies are expressed as affine functions of the domain indices,

[1] Alphabets can express such computations as data-dependent dependencies like other work, but this is not the

including the unbounded domain corresponding to `while` loops. In other words, there are only (semantic) iteration variables in the language. A potential application of our analysis is to construct Alphabets representations of programs including `while` loops.

Apollo [SRC15] is a framework for runtime optimization that detects (affine) regularity in program behavior and applies polyhedral optimizations, speculating that the regularity persists. Runtime analysis enables code regions that cannot be determined to be polyhedral at compile-time to be found and optimized. Our work shares some similarities with the dynamically polyhedral programs targeted by Apollo. The main difference is that we target statically regular programs, but with more flexibility on how the program is written.

8 Conclusion

In this paper we proposed a new semantic formalization of the key analysis of the polyhedral literature, namely *Array Dataflow Analysis*. We formalized the notion of dependency in a semantic fashion and showed the relevance of this notion by demonstrating its applicability to the traditional syntactic polyhedral programs as well as to covertly regular programs. We also proposed an approximated computation of dependencies in the general case of non-regular control flows.

Future work includes extensions of our analyses for more general programs including non-affine accesses and more complex data structures such as trees. Based on a proper definition of our general approximated dependence analysis, we will then be able to revisit and extend other classical polyhedral activities such as loop transformations and code generation to allow for optimization and parallelization of programs with while loops and loosened control structures.

default/intended use in Alphabets.

References

[Bas04] Cédric Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. In *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT'04)*, pages 7–16. IEEE Computer Society, 2004.

[BCF97] Denis Barthou, Jean-François Collard, and Paul Feautrier. Fuzzy array dataflow analysis. *Journal of Parallel and Distributed Computing*, 40(2):210 – 226, 1997.

[BPCB10] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction (CC'10)*, CC'10/ETAPS'10, pages 283–303, 2010.

[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.

[Fea15] Paul Feautrier. The power of polynomials. In *In Proceedings of 5th International Workshop on Polyhedral Compilation Techniques (IMPACT'15)*, 2015.

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

[GGL12] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010–1–1250010–28, 2012.

[GMC09] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. Speed: Precise and efficient static estimation of program computational complexity. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 127–139, New York, NY, USA, 2009. ACM.

[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.

[MFK+16] Alexandre Maréchal, Alexis Fouilhé, Tim King, David Monniaux, and Michaël Périn. Polyhedral approximation of multivariate polynomials using handelman's theorem. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *Lecture Notes in Computer Science*, pages 166–184. Springer Verlag, January 2016.

[MRPV+17] Claire Maiza, Pascal Raymond, Catherine Parent-Vigouroux, Armelle Bonenfant, Fabienne Carrier, Hugues Cassé, Philippe Cuenot, Denis Claraz, Nicolas Halbwachs, Erwan Jahier, et al. The w-sept project: Towards semantic-aware wcet estimation. In *OASIcs-OpenAccess Series in Informatics*, volume 57, 2017.

[Pug91] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *1991 ACM/IEEE Conference on Supercomputing*, SC '91, pages 4–13, 11 1991.

[PW96] William Pugh and David Wonnacott. *Non-Linear Array Dependence Analysis*, pages 1–14. 1996.

[RGK11] Sanjay Rajopadhye, Samik Gupta, and Daegon Kim. Alphabets: An extended polyhedral equational language. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW'11)*, pages 656–664, May 2011.

[RVS16] Pierre Roux, Yuen-Lam Voronin, and Sriram Sankaranarayanan. Validating numerical semidefinite programming solvers for polynomial invariants. In Xavier Rival, editor, *Static Analysis Symposium (SAS'16)*, pages 424–446, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[SRC15] Aravind Sukumaran-Rajam and Philippe Clauss. The polyhedral model of nonlinear loops. *ACM Transactions on Architecure and Code Optimization*, 12(4):48:1–48:27, December 2015.

[VCB11] Sven Verdoolaege, Albert Cohen, and Anna Beletska. Transitive Closures of Affine Integer Tuple Relations and their Overapproximations. In Eran Yahav, editor, *Proceedings of the 18th International Static Analysis Symposium (SAS'11)*, volume 6887 of *LNCS*, pages 216–232, Venice, Italy, September 2011. Springer.

[VNS13] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. On Demand Parametric Array Dataflow Analysis. In Armin Größlinger and Louis-Noël Pouchet, editors, *Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques*, pages 23–36, Berlin, Germany, January 2013.

Contents

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399