HAL
open science

Static Analysis by Abstract Interpretation of the Functional Correctness of Matrix Manipulating Programs

Matthieu Journault, Antoine Miné

To cite this version:

Matthieu Journault, Antoine Miné. Static Analysis by Abstract Interpretation of the Functional Correctness of Matrix Manipulating Programs. 23rd Static Analysis Symposium (SAS), Sep 2016, Edimbourg, United Kingdom. pp.257-277, 10.1007/978-3-662-53413-7_13 . hal-01360556

HAL Id: hal-01360556

https://hal.sorbonne-universite.fr/hal-01360556v1

Submitted on 6 Sep 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Static Analysis by Abstract Interpretation of the Functional Correctness of Matrix Manipulating Programs*

Matthieu Journault[1] and Antoine Miné[2]

[1] Computer Science Department, ENS Cachan, France
matthieu.journault@ens-cachan.fr
[2] Sorbonnes Universités, UPMC Univ Paris 6,
Laboratoire d'informatique de Paris 6 (LIP6)
4, pl. Jussieu, 75005 Paris, France
antoine.mine@lip6.fr

Abstract. We present new abstract domains to prove automatically the functional correctness of algorithms implementing matrix operations, such as matrix addition, multiplication, GEMM (general matrix multiplication), or more generally BLAS (Basic Linear Algebra Subprograms). In order to do so, we introduce a family of abstract domains parameterized by a set of matrix predicates and by a numeric domain. We show that our analysis is robust enough to prove the functional correctness of several versions of matrix addition and multiplication codes resulting from loop reordering, loop tiling, inverting the iteration order, line swapping, and expression decomposition. Finally, we extend our method to enable modular analysis on code fragments manipulating matrices by reference, and show that it results in a significant analysis speedup.

1 Introduction

Static analysis by abstract interpretation [7] allows discovering automatically properties about program behaviors. In order to scale up, it employs abstractions, which induce approximations. However, the approximation is sound, as it considers a super-set of all program behaviors; hence, any property proved in the abstract (such as the absence of run-time error, or the validation of some specification) also holds in all actual program executions. Static analysis by abstract interpretation has been applied with some success to the analysis of run-time errors [3]. More recently, it has been extended to proving functional properties, including array properties [6,8,1], such as proving that a sorting algorithm indeed outputs a sorted array. In this work, we consider functional properties of a different kind, not tackled before: properties on matrices. Consider as example Program 1.1 starting with the assumption $N > 0$. Our analyzer will automatically infer the following postcondition for the program:

* This work is partially supported by the European Research Council under Consolidator Grant Agreement 681393 – MOPSA.

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

Program 1.1: Matrix multiplication $C = A \times B$.

```
1   /* N >= 5 */
2   i=0;
3   while (i<N) {
4     j=0;
5     while (j<N) {
6       C[i][j] = A[i][j] + B[i][j];
7       j++;
8     };
9     i++;
10  }
```

Program 1.2: Matrix addition $C \leftarrow A + B$.

$\forall u, v \in [0, N-1],\ C[u][v] = \sum_{w=0}^{N-1} A[u][w] \times B[w][v]$, i.e., the program indeed computes the product of matrices A and B into C.

Introductory example. To explain how our method works in more details, we focus on a simpler introductory example, the matrix addition from Program 1.2. Note that we will show later that our method is also successful on the multiplication from Program 1.1, as well as more complex, optimized variants of these algorithms, such as the tiled addition in Program 1.6.

We wish to design a sound analyzer capable of inferring that, at line 10, the addition of `B` and `A` has been stored into `C`. Note that the program is parametric in the size N of the matrix, and we want our analysis to be able to prove that it is correct independently from the precise value of N. Therefore, we would like to infer that the formula $\phi \triangleq \forall a,\ \forall b,\ (0 \leq a < N \wedge 0 \leq b < N) \Rightarrow C[a][b] = A[a][b] + B[a][b]$ holds for all the memory states reachable at line 10.

In order to do so, the static analyzer needs to first infer loop invariants for each of the `while` loops. For the outermost loop (line 3), we would infer: $\phi_i \triangleq \forall a,\ \forall b,\ (0 \leq a < i \wedge 0 \leq b < N) \Rightarrow C[a][b] = A[a][b] + B[a][b]$, and, for the innermost loop (line 5): $\phi_j \triangleq (\forall a,\ \forall b,\ (0 \leq a < i \wedge 0 \leq b < N) \Rightarrow C[a][b] = A[a][b] + B[a][b]) \wedge (\forall b,\ 0 \leq b < j \Rightarrow C[i][b] = A[i][b] + B[i][b])$. Note that the loop invariants are far more complex than the formula we expect at the end of the program. In particular, they depend on the local variables i and j. They express

the fact that the addition has been performed only for some lines (up to i) and, for ϕ_j, only on one part of the last line (up to j). Every formula we need can be seen as a conjunction of one or several sub-formulas, where each sub-formula expresses that the addition has been performed on some rectangular sub-part of the matrix. Therefore, we introduce the following formula $Add(A, B, C, x, y, z, t)$ with which we will describe our matrices in the rest of the introductory example:

$$Add(A, B, C, x, y, z, t) \triangleq \forall a,\ \forall b,\ (x \leq a < z \wedge y \leq b < t) \\ \Rightarrow C[a][b] = A[a][b] + B[a][b]$$

The formulas ϕ_i, ϕ_j and ϕ can all be described as a conjunction of one or more instances of the fixed predicate *Add*, as well as numeric constraints relating only scalar variables, including program variables (i, j) and predicate variables (x, y, z, t). Abstract interpretation provides numeric domains, such as polyhedra [9], to reason on numeric relations. We will design a family of abstract domains combining existing numeric domains with predicates such as *Add* and show how, ultimately, abstract operations modeling assignments, tests, joins, etc. can be expressed as numeric operations to soundly reason about matrix contents. While the technique is similar to existing analyses of array operations [6,8,1], the application to matrices poses specific challenges: firstly, as matrices are bi-dimensional, it is less obvious how to update matrix predicates after assignments; secondly, matrix programs feature large levels of loop nesting, which may pose scalability issues.

Contribution. The article presents a new static analysis for matrix-manipulating programs, based on a combination of parametric predicates and numeric domains. The analysis has been proved sound and implemented in a prototype. We provide experimental results proving the functional correctness of a few basic matrix-manipulating programs, including more complex variants obtained by the Pluto source to source loop optimizer [5,4]. We show that our analysis is robust against code modifications that leave the semantic of the program unchanged, such as loop tiling performed by Pluto. In the context of full source to binary program certification, analyzing programs after optimization can free us from having to verify the soundness of the optimizer; therefore, our method could be used in combination with certified compilers, such as CompCert [14], while avoiding the need to certify optimization passes in Coq. The analysis we propose is modular: it is defined over matrix predicates that can be combined, and is furthermore parameterized by the choice of a numeric domain. Finally, the performance of our analyzer is reasonable.

The rest of the paper is organized as follows: Sect. 2 describes the programming language we aim to analyze; Sect. 3 formally defines the family of abstractions we are constructing. In Sect. 4, we introduce specific instances to handle matrix addition and multiplication. In Sect. 5, we briefly present a modular inter-procedural version of our analysis. Some technical details about the implementation of our analyzer prototype and our experimental results can be found in Sect. 6. Section 7 discusses related works, while Sect. 8 concludes.

$$
\begin{aligned}
E &::= \mathtt{v} \in \mathbb{R} \mid X \in \mathbb{X} \mid E_1 + E_2 \mid E_1 \times E_2 \mid A[X_1][X_2] \mid A.\mathtt{n} \mid A.\mathtt{m} \\
B &::= E_1 < E_2 \mid E_1 \leq E_2 \mid E_1 = E_2 \mid \neg B \mid B_1 \vee B_2 \mid B_1 \wedge B_2 \\
C &::= \mathtt{skip} \mid X := E \mid A[X_1][X_2] \leftarrow E \mid C_1 \ ; \ C_2 \mid \\
&\quad \mathtt{If}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2 \mid \mathtt{While}\ B\ \mathtt{do}\ C\ \mathtt{done} \mid \mathtt{Array}\ A\ \mathtt{of}\ E_1\ E_2
\end{aligned}
$$

Fig. 1: Syntax of the language.

2 Syntax and concrete semantics

2.1 Programming language syntax

We consider a small imperative language, in Fig. 1, based on variables $X \in \mathbb{X}$, (bi-dimensional) array names $A \in \mathbb{A}$, and size variables denoting the width and height of arrays $\mathbb{S} \triangleq \{A.\mathtt{n} \mid A \in \mathbb{A}\} \cup \{A.\mathtt{m} \mid A \in \mathbb{A}\}$, with arithmetic expressions $E \in \mathbb{E}$, boolean expressions $B \in \mathbb{B}$, and commands $C \in \mathbb{C}$. The command $\mathtt{Array}\ A\ \mathtt{of}\ E_1\ E_2$ denotes the definition of a matrix A of size $E_1 \times E_2$.

2.2 Concrete reachability

We define the concrete semantic of our programming language. It is the most precise mathematical expression describing the possible executions of the program and it will be given in terms of postconditions for commands. This concrete semantic is not computable; therefore, the rest of the article will propose computable approximations. The soundness property of Thm. 1 (Sect. 4.1) will hold with respect to this concrete semantic.

$\mathcal{D}$ is the domain of scalar values, and we assume from now on that $\mathcal{D} = \mathbb{R}$. $\mathcal{M}$ is the set of memory states, which are pairs in $\mathcal{M} \triangleq \mathcal{M}_{\mathcal{V}} \times \mathcal{M}_{\mathcal{A}}$ containing a scalar environment in $\mathcal{M}_{\mathcal{V}} \triangleq \mathcal{V} \to \mathcal{D}$ and a matrix contents environment in $\mathcal{M}_{\mathcal{A}} \triangleq \mathcal{A} \to (\mathbb{N} \times \mathbb{N}) \to \mathcal{D}$, where $\mathcal{V}$ (resp. $\mathcal{A}$) is any subset of $\mathbb{X} \cup \mathbb{S}$ (resp. $\mathbb{A}$). $\mathcal{E}[\![E]\!] \in \mathcal{M} \to \mathcal{D}$ is the semantic of an expression E (it is well-defined only on memory states that bind the variables and array names appearing in E). $\mathcal{B}[\![B]\!] \in \wp(\mathcal{M})$ defines the set of memory states in which B is true. $\mathbf{Post}[\![C]\!](S)$ denotes the set of states reachable from the set of states S after a command C. Finally, $\mathbf{var}(E)$, $\mathbf{var}(B)$, $\mathbf{var}(C)$ denote the variables appearing respectively in an arithmetic expression, a boolean expression, a command. Figures 2 and 3 describe the behavior of expressions and the state reachability of the program. The evaluation of boolean expressions was left out as it is standard.

3 Generic abstract semantics

3.1 Predicates

As suggested in Sect. 1, we define predicates to specify relations between matrices. The language of predicates is based on the expressions from our program-

- $\forall v \in \mathbb{R}, \mathcal{E}[\![\mathtt{v}]\!]m \triangleq v$
- $\mathcal{E}[\![X]\!]m \triangleq m_{\mathcal{V}}(X)$
- $\mathcal{E}[\![E_1 + E_2]\!]m \triangleq \mathcal{E}[\![E_1]\!]m + \mathcal{E}[\![E_2]\!]m$
- $\mathcal{E}[\![E_1 \times E_2]\!]m \triangleq \mathcal{E}[\![E_1]\!]m \times \mathcal{E}[\![E_2]\!]m$
- $\mathcal{E}[\![A.\mathtt{n}]\!]m \triangleq m_{\mathcal{V}}(A.\mathtt{n})$
- $\mathcal{E}[\![A.\mathtt{m}]\!]m \triangleq m_{\mathcal{V}}(A.\mathtt{m})$
- $\mathcal{E}[\![A[X_1][X_2]]\!]m \triangleq m_{\mathcal{A}}(A)(m_{\mathcal{V}}(X_1), m_{\mathcal{V}}(X_2))$ when $m_{\mathcal{V}}(X_1), m_{\mathcal{V}}(X_2) \in \mathbb{N}$

Fig. 2: Evaluation of expressions. m designates the pair $(m_{\mathcal{V}}, m_{\mathcal{A}})$.

$$
\begin{array}{lll}
\mathbf{Post}[\![C]\!](S) \triangleq \text{match } C \text{ with :} & & \\
\quad | & \mathtt{skip} & \to S \\
\quad | & X := E & \to \{(m_{\mathcal{V}}[X \mapsto \mathcal{E}[\![E]\!]m], m_{\mathcal{A}}) \mid m \in S\} \\
\quad | & A[X_1][X_2] \leftarrow E & \to \{(m_{\mathcal{V}}, m_{\mathcal{A}}[A \mapsto (m_{\mathcal{V}}(X_1), m_{\mathcal{V}}(X_2)) \mapsto \mathcal{E}[\![E]\!]m]) \\
 & & \quad \mid m \in S \wedge m_{\mathcal{V}}(X_1), m_{\mathcal{V}}(X_2) \in \mathbb{N}\} \\
\quad | & \mathtt{If}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2 & \to \mathbf{Post}[\![C_1]\!](S \cap \mathcal{B}[\![B]\!]) \cup \mathbf{Post}[\![C_2]\!](S \cap \mathcal{B}[\![\neg B]\!]) \\
\quad | & \mathtt{While}\ B\ \mathtt{do}\ C_1\ \mathtt{done} & \to \mathbf{lfp}^{\subseteq}(\lambda S_0.S \cup \mathbf{Post}[\![C_1]\!](S_0 \cap \mathcal{B}[\![B]\!])) \cap \mathcal{B}[\![\neg B]\!] \\
\quad | & \mathtt{Array}\ A\ \mathtt{of}\ E_1\ E_2 & \to \{(m_{\mathcal{V}}[A.\mathtt{m} \mapsto \mathcal{E}[\![E_1]\!]m, A.\mathtt{n} \mapsto \mathcal{E}[\![E_2]\!]m], m_{\mathcal{A}}) \\
 & & \quad \mid m \in S\} \\
\quad | & C_1\ ;\ C_2 & \to \mathbf{Post}[\![C_2]\!](\mathbf{Post}[\![C_1]\!](S))
\end{array}
$$

Fig. 3: Reachability semantics. m designates the pair $(m_{\mathcal{V}}, m_{\mathcal{A}})$.

ming language, with added quantifiers. The predicate language is very general but we will only be using one fragment at a time to describe the behavior of our programs. In order to do so, we define terms $t \in \mathcal{T}$, using dedicated predicate variables $y \in \mathbb{Y}$ (such that $\mathbb{X} \cap \mathbb{Y} = \mathbb{Y} \cap \mathbb{S} = \emptyset$), atomic predicates $g \in \mathcal{G}$, and predicates $P \in \mathcal{P}$, as shown in Fig. 4. Their interpretation will be respectively: $\mathcal{I}_{\mathcal{T}}[\![t]\!]$, $\mathcal{I}_{\mathcal{G}}[\![g]\!]$, and $\mathcal{I}_{\mathcal{P}}[\![P]\!]$. Those interpretations are defined the same way as interpretations of expressions and booleans; therefore, they are not detailed.

Example 1. Modulo some syntactic sugar, we can define a predicate such as: $P_+(A, B, C, x, y, z, t) \triangleq \forall u \in [x, z-1], \forall v \in [y, t-1], A[u][v] = B[u][v] + C[u][v]$.

$$
\begin{array}{l}
t ::= A[x][y] \mid x \in \mathbb{Y} \mid t_1 + t_2 \mid t_1 \times t_2 \mid \mathtt{v} \in \mathbb{R} \mid \displaystyle\sum_{x=y}^{z} t \\
g ::= t_1 = t_2 \mid t_1 \leq t_2 \mid t_1 < t_2 \\
P ::= \mathtt{tt} \mid \mathtt{ff} \mid g \in \mathcal{G} \mid \neg P_1 \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \Rightarrow P_2 \mid \forall x \in \mathbb{N},\ P_1
\end{array}
$$

Fig. 4: Terms, atomic predicates, and predicates.

The interpretation of such a predicate is the set of memory states in which the matrix A is the sum of two matrices B and C on some rectangle.

Finally, we introduce the following relation $P_1 =_S P_2$, meaning that $\exists \sigma \in \mathbf{var}(P_1) \to \mathbf{var}(P_2)$ a bijection such that $P_1\sigma = P_2$, i.e., P_1 and P_2 are syntactically equivalent modulo a renaming of the variables (e.g., $P_+(A, B, C, x, y, z, t) =_S P_+(A, B, C, x', y', z', t')$). This definition is extended to sets of predicates, $\mathfrak{P}_1 =_S \mathfrak{P}_2$, meaning that there is a one-to-one relation between the sets and every pair of elements in the relation are syntactically equivalent (e.g., $\{Q(A, B, C, x, y, z, t), P(A, u, v)\} =_S \{Q(A, B, C, x', y', z', t'), P(A, u', v')\}$).

3.2 Abstract states

We now assume we have an abstraction of the state $\mathcal{M}_\mathcal{V}$ of the scalar, numeric variables (e.g., the interval or polyhedra [9] domain) that we call $\mathcal{M}^\sharp_\mathcal{V}$, with concretization function $\gamma_\mathcal{V}$, join $\cup_\mathcal{V}$, widening $\triangledown_\mathcal{V}$, meet with a boolean constraint $\cap_{\mathcal{B},\mathcal{V}}$ or a family of boolean constraints $\sqcap_{\mathcal{B},\mathcal{V}}$, and a partial order relation $\sqsubset_\mathcal{V}$. The set of variables bound by an abstract variable memory state $\mathfrak{a} \in \mathcal{M}^\sharp_\mathcal{V}$ is noted $\mathbf{var}(\mathfrak{a}) \subset \mathbb{X} \cup \mathbb{Y} \cup \mathbb{S}$. In the following, we define an abstraction for the analysis of our programming language, this abstraction being built upon a set of predicates describing relations between matrices and a numeric domain. To simplify, without loss of generality, we forbid predicates from referencing program variables or array dimensions, hence $\mathbb{Y} \cap (\mathbb{X} \cup \mathbb{S}) = \emptyset$. We rely instead on the numeric domain to relate predicate variables with program variables, if needed.

We need abstract states to be disjunctions (expressing the different possible states a program can be in at some point) of conjunctions of predicates (that describe a possible state of the program, as a set of constraints and predicates that hold). The conjunctions are necessary as a predicate may hold on several rectangles; the disjunctions are necessary because, within the same loop, the number of predicates may vary from one loop iteration to another (a graphical illustration is given in Fig. 7). Hence, we will introduce a monomial as a conjunction of some predicates and a numeric abstract domain element; an abstract state will be a disjunction of such monomials. Moreover, we want our state to be a set of monomials as small as possible (and in all cases, bounded, to ensure termination); therefore, we would rather use the disjunctive features provided by the numeric domain, if any, than predicate-level disjunctions. We enforce this rule through a notion of well-formedness, explained below.

Definition 1 (*Monomial*). *We call a well-named monomial an element* $(\mathfrak{P}, \mathfrak{a}) \in \mathfrak{M} \triangleq \wp(\mathcal{P}) \times \mathcal{M}^\sharp_\mathcal{V}$ *such that* $\bigcup_{P \in \mathfrak{P}} \mathbf{var}(P) \subset \mathbf{var}(\mathfrak{a})$ *and* $\forall P_1 \neq P_2 \in \mathfrak{P}$, $\mathbf{var}(P_1) \cap \mathbf{var}(P_2) = \emptyset$.

Definition 2 (*Abstract memory state*). *We build an abstraction* $\mathcal{M}^\sharp$ *for memory states* $\mathcal{M}$ *that is* $\mathcal{M}^\sharp \triangleq \wp(\mathfrak{M})$. *We will say that an abstract memory state is well-named if every one of its monomials is well-named. The following*

concretization function is defined on every well-named abstract state as:

$$\gamma_p(S^\sharp) = \bigcup_{(\mathfrak{P},\mathfrak{a})\in S^\sharp} \{(m_{\mathcal{V}|\mathbb{X}\cup\mathbb{S}}, m_{\mathcal{A}}) \mid m_{\mathcal{V}} \in \gamma_{\mathcal{V}}(\mathfrak{a}) \wedge \forall P \in \mathfrak{P},\ (m_{\mathcal{V}}, m_{\mathcal{A}}) \in \mathcal{I}_{\mathcal{P}}(P)\}$$

where $m_{\mathcal{V}|\mathbb{X}\cup\mathbb{S}}$ is the restriction of $m_{\mathcal{V}}$ to $\mathbb{X}\cup\mathbb{S}$.

Example 2. (P_+ is the predicate defined in Example 1). Let us consider an abstract state $\{(\mathfrak{P},\mathfrak{a})\}$ where $\mathfrak{P} = \{P_+(A,B,C,x,y,z,t)\}$ and $\mathfrak{a}$ is a polyhedron defined by the set of equations $\{x=0, y=0, z=n, t=1, i=1, j=n, A.\mathtt{n}=n, B.\mathtt{n}=A.\mathtt{n}, C.\mathtt{n}=A.\mathtt{n}, C.\mathtt{m}=B.\mathtt{m}, B.\mathtt{m}=A.\mathtt{m}, A.\mathtt{m}=n\}$. In this case: $\gamma_p(\{(\mathfrak{P},\mathfrak{a})\})$ is the set of memory states in which the first column of the matrix `A` is the sum of the first columns of `B` and `C`.

Definition 3 (*Well-formed*). *We will say that an abstract state $S^\sharp$ is well-formed if $\forall(\mathfrak{P}_1,\mathfrak{a}_1),(\mathfrak{P}_2,\mathfrak{a}_2) \in S^\sharp, (\mathfrak{P}_1 =_S \mathfrak{P}_2) \Rightarrow (\mathfrak{P}_1,\mathfrak{a}_1) = (\mathfrak{P}_2,\mathfrak{a}_2)$, i.e., no two monomials in $S^\sharp$ can be made equal through variable renaming.*

Definition 4 (*Shape*). *We will say that two abstract states $S_1^\sharp, S_2^\sharp$ have the same shape, noted $S_1^\sharp =_F S_2^\sharp$, when $S_1^\sharp \subset_F S_2^\sharp \wedge S_2^\sharp \subset_F S_1^\sharp$ where $S_1^\sharp \subset_F S_2^\sharp \triangleq \forall(\mathfrak{P}_1,\mathfrak{a}_1) \in S_1^\sharp, \exists(\mathfrak{P}_2,\mathfrak{a}_2) \in S_2^\sharp,\ \mathfrak{P}_1 =_S \mathfrak{P}_2$*

Remark 1. All the previous definitions amount to say that a well-named, well-formed abstract state is of the form :

$$\begin{array}{l}
(\mathrm{Mon}_1 : \mathtt{numeric_constraints_1} \wedge \mathtt{predicate_1_1} \wedge \mathtt{predicate_1_2} \wedge \ldots) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee \\
(\mathrm{Mon}_2 : \mathtt{numeric_constraints_2} \wedge \mathtt{predicate_2_1} \wedge \mathtt{predicate_2_2} \wedge \ldots) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
(\mathrm{Mon}_n : \mathtt{numeric_constraints_n} \wedge \mathtt{predicate_n_1} \wedge \mathtt{predicate_n_2} \wedge \ldots)
\end{array}$$

where no two lines have the same multi-set of predicates.

Let us now define an algorithm $\mathbf{wf}_p$ that transforms any well-named abstract state into a well-named, well-formed abstract state, by transforming every pair of monomials $(\mathfrak{P}_1,\mathfrak{a}_1),(\mathfrak{P}_2,\mathfrak{a}_2) \in S_1^\sharp$ such that $\mathfrak{P}_1 =_S \mathfrak{P}_2$ into $(\mathfrak{P}_2, \mathfrak{a}_1\sigma \cup_{\mathcal{V}} \mathfrak{a}_2)$ where $\mathfrak{P}_1\sigma = \mathfrak{P}_2$ for some permutation σ. However, we only have $\gamma_p(S^\sharp) \subset \gamma_p(\mathbf{wf}_p(S^\sharp))$ in general and not the equality. Indeed, we transform a symbolic disjunction into a disjunction $\cup_{\mathcal{V}}$ on the numeric domain, therefore this transformation might not be exact. Figure 5 gives the basic operations on the sets of abstract states $\mathcal{M}^\sharp$: a join ($\sqcup_p$), a widening ($\triangledown_p$),[3] a meet with booleans ($\sqcap_{p,\mathcal{B}}$), and an inclusion test of the numeric component with boolean expressions ($\sqsubseteq_{\mathcal{B},\mathcal{V}}$).

[3] $\triangledown_p$ can not be used when the two abstract states do not have the same shape, in which case the analyzer will perform a join. However, ultimately, the shape will stabilize, thus allowing the analyzer to perform a widening. This widening technique is similar to the one proposed by [18] on cofibred domains.

$\sqcup_p$: $S_1^\sharp \sqcup_p S_2^\sharp \triangleq \mathbf{wf}_p(S_1^\sharp \cup S_2^\sharp)$

$\triangledown_p$: When two abstract states $S_1^\sharp, S_2^\sharp$ have the same shape, there is a one-to-one function f from the monomials of $S_1^\sharp$ onto the monomials of $S_2^\sharp$ such that $\forall \mathfrak{m} = (\mathfrak{P}_1, \mathfrak{a}_1) \in S_1^\sharp$, if $(\mathfrak{P}_2, \mathfrak{a}_2) = f(\mathfrak{m}) \in S_2^\sharp$, then $\mathfrak{P}_1 = \mathfrak{P}_2\sigma$. Thus allowing us to define $S_1^\sharp \triangledown_p S_2^\sharp \triangleq \{(\mathfrak{P}_1, \mathfrak{a}_1 \triangledown_\mathcal{V} \mathfrak{a}_2\sigma) \mid \mathfrak{m} = (\mathfrak{P}_1, \mathfrak{a}_1) \in S_1^\sharp \wedge (\mathfrak{P}_2, \mathfrak{a}_2) = f(\mathfrak{m})\}$.

$\sqcap_{p,\mathcal{B}}$: $S_1^\sharp \sqcap_{p,\mathcal{B}} B \triangleq \bigcup_{(\mathfrak{P},\mathfrak{a}) \in S_1^\sharp} \{(\mathfrak{P}, \mathfrak{a} \cap_{\mathcal{B},\mathcal{V}} B)\}$

$\sqsubseteq_{\mathcal{B},\mathcal{V}}$: $\mathfrak{a} \in \mathcal{M}_\mathcal{V}^\sharp, \sqsubseteq_{\mathcal{B},\mathcal{V}} \bigwedge_{i \in I} B_i \in B \triangleq \forall i \in I,\ \gamma_\mathcal{V}(\mathfrak{a}) \subseteq \mathcal{B}[\![B_i]\!]$

Fig. 5: Operations on abstract states.

$$
\begin{aligned}
&\mathbf{Post}^\sharp[\![\texttt{skip}]\!](S_1^\sharp) \triangleq S_1^\sharp \\
&\mathbf{Post}^\sharp[\![X := E]\!](S_1^\sharp) \triangleq \textstyle\bigcup_{(\mathfrak{P},\mathfrak{a}) \in S_1^\sharp} \{(\mathfrak{P}, \mathbf{Post}_\mathcal{V}^\sharp[\![X := E]\!](\mathfrak{a}))\} \\
&\mathbf{Post}^\sharp[\![A[X_1][X_2] \leftarrow E]\!](S_1^\sharp) \triangleq \mathbf{wf}_p(\textstyle\bigcup_{(\mathfrak{P},\mathfrak{a}) \in S_1^\sharp} \{\mathbf{Post}_\mathfrak{M}^\sharp(\mathfrak{P}, \mathfrak{a}, A, E, X_1, X_2)\}) \\
&\mathbf{Post}^\sharp[\![\texttt{If}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!](S_1^\sharp) \triangleq \\
&\quad \mathbf{Post}^\sharp[\![C_1]\!](S_1^\sharp \sqcap_{p,\mathcal{B}} B) \sqcup_p \mathbf{Post}^\sharp[\![C_2]\!](S_1^\sharp \sqcap_{p,\mathcal{B}} (\neg B)) \\
&\mathbf{Post}^\sharp[\![\texttt{While}\ B\ \texttt{do}\ C\ \texttt{done}]\!](S_1^\sharp) \triangleq \\
&\quad \mathbf{Res}(\mathbf{fp}(\lambda S^\sharp \rightarrow\ \texttt{let}\ S_P^\sharp = S^\sharp \sqcup_p \mathbf{Post}^\sharp[\![C]\!](S^\sharp \sqcap_{p,\mathcal{B}} B)\ \texttt{in} \\
&\qquad \texttt{if}\ S_P^\sharp =_F S^\sharp\ \texttt{then}\ S^\sharp \triangledown_p S_P^\sharp\ \texttt{else}\ S_P^\sharp)(S_1^\sharp) \sqcap_{p,\mathcal{B}} (\neg B)) \\
&\mathbf{Post}^\sharp[\![C_1\ ;\ C_2]\!](S_1^\sharp) \triangleq \mathbf{Post}^\sharp[\![C_2]\!](\mathbf{Post}^\sharp[\![C_1]\!](S_1^\sharp))
\end{aligned}
$$

Fig. 6: Abstract postconditions.

3.3 Abstract transfer functions

We describe, in Fig. 6, how program commands are interpreted using the memory abstract domain to yield a computable over-approximation of the set of reachable states. We assume that the numeric domain provides the necessary transfer functions ($\mathbf{Post}_\mathcal{V}^\sharp[\![C]\!]$) for all instructions involving only scalar variables. When interpreting loops, the call to $\mathbf{fp}(f)(x)$ computes the fixpoint of f reached by successive iterations of f starting at x (terminating in finite time through the use of a widening).

Two functions are yet to be defined: $\mathbf{Post}_\mathfrak{M}^\sharp$ and $\mathbf{Res}$. Indeed, these functions will depend on our choice of predicates. For instance, Sect. 4.1 will introduce one version of $\mathbf{Post}_\mathfrak{M}^\sharp$, named $\mathbf{Post}_{+,\mathfrak{M}}^\sharp$, able to handle expressions of the form $A[X_1][X_2] \leftarrow B[X_1][X_2] + C[X_1][X_2]$ to analyze additions, while Sect. 4.2 will discuss multiplications. In practice, the developer will enrich the functions when adding new predicates to analyze new kinds of matrix algorithms in order to design a flexible, general-purpose analyzer. Depending on E and the predicates already existing in the abstract state, $\mathbf{Post}_\mathfrak{M}^\sharp(\ldots, E, \ldots)$ will either modify variables appearing in predicates, produce new predicates, or remove predicates. In

the worst case, when $\mathbf{Post}^{\sharp}_{\mathfrak{M}}$ cannot handle some expression E, it yields the approximation $\top$, thus ensuring the correctness of the analyzer.

The function **Res** geometrically resizes the abstract state: it coalesces predicates describing adjacent matrix parts into a single part, as long as their join can be exactly represented in our domain (over-approximating the join would be unsound as our predicates employ universal quantifiers). An algorithm for **Res** for the case of the addition is proposed in Sect. 4.1. As expected, the problem of extending a bi-dimensional rectangle is slightly more complex than that of extending a segment, as done in traditional array analysis [6,8]. New rewriting rules are added to **Res** when new families of predicates are introduced.

Example 3. For example, we set $\mathbf{Post}^{\sharp}_{\mathfrak{M}}[\![A[0][0] \leftarrow B[0][0] + C[0][0]]\!](\emptyset, \emptyset) = (\{P_+(A, B, C, x, y, z, t)\}, \{x = y = 0, z = t = 1\})$ and $\mathbf{Res}(\{P_+(A, B, C, x, y, z, t), P_+(A, B, C, a, b, c, d)\}, \{x = y = 0, z = 1, t = 10, a = 1, c = 2, b = 0, d = 10\}) = (\{P_+(A, B, C, x, y, z, t)\}, \{x = 0, z = 2, y = 0, t = 10\})$.

Maximum number of predicates. As widenings are used on pairs of abstract states with the same shape, we need to ensure that the shape will stabilize; therefore, we need to bound the number of possible shapes an abstract state can have. In order to do so, we only authorize a certain amount of each kind of predicates in a given abstract state. This number will be denoted as M_{pred}. This bound is necessary to ensure the termination of the analysis, but it can be set to an arbitrary value by the user. Note, however, that **Res** will naturally reduce the number of predicates without loss of precision, whenever possible. In practice, M_{pred} depends on the complexity of the loop invariant, which experimentally depends on the number of nested loops and is usually small ($M_{pred} = 4$ was enough to prove the correctness of all the programs considered here).

4 Abstraction instances

4.1 Matrix addition

We now consider the analysis of the assignment $E \triangleq A[X_1][X_2] \leftarrow B[X_1][X_2] + C[X_1][X_2]$, as part of proving that a matrix addition is correct. As suggested by the introductory example in Sect. 1, let us define the following predicate:

$$P_+(A, B, C, x, y, z, t) \triangleq \forall a, b \in [x, z-1] \times [y, t-1],\ A[a][b] = B[a][b] + C[a][b]$$

We define now versions $\mathbf{Post}^{\sharp}_{+,\mathfrak{M}}$ and $\mathbf{Res}_+$ to compute postconditions and possible resize over P_+. Even though the analyzer we implemented can handle predicates on arbitrary many matrices, we will make the description of the algorithms and the proof of correctness simpler by only allowing our analyzer to use predicates of the form P_+ such that $\exists A, B, C, \forall P_+(A', B', C', \dots) \in \mathfrak{P}$, $A' = A \wedge B' = B \wedge C' = C$ (i.e., the source and destination matrices are the same for all the addition predicates used in an abstract state).

Algorithm 1: $\mathbf{Post}^{\sharp}_{+,\mathfrak{M}}$, computes the image by the transfer functions (associated to the expression $E = A[X_1][X_2] \leftarrow B[X_1][X_2] + C[X_1][X_2]$)

Input : $(\mathfrak{P}, \mathfrak{a})$,$A$,$B$,$C$,$i$,$j$
Output: $(\mathfrak{P}', \mathfrak{a}')$ the postcondition

if $\exists P_0 \in \mathfrak{P}$, FIND_EXTENSION$((P_0, \mathfrak{a})$,$A$,$B$,$C$,$i$,$j) = res \neq$ *None* **then**
  $x', y', z', t' \leftarrow$ **fresh**();
  $I \leftarrow$ **switch** res **do**
    case *Some(Right)*
      $(x = x' \wedge y = y' \wedge z + 1 = z' \wedge t = t')$
    case *Some(Down)*
      $(x = x' \wedge y = y' \wedge z = z' \wedge t + 1 = t')$
    case *Some(Left)*
      $(x = x' + 1 \wedge y = y' \wedge z = z' \wedge t = t')$
    case *Some(Up)*
      $(x = x' \wedge y = y' + 1 \wedge z = z' \wedge t = t')$
  endsw
  return $((\mathfrak{P} \setminus P_0) \cup \{P_+(A, B, C, x', y', z', t')\}, \mathfrak{a} \sqcap_{\mathcal{B},\mathcal{V}} I)$
else
  if $\forall P_+(A', B', C', _, _, _, _) \in \mathfrak{P},\ A' = A \wedge B' = B \wedge C' = C$ **then**
    if $\sharp\mathfrak{P} < M_{pred}$ **then**
      $x', y', z', t' \leftarrow$ **fresh**();
      $I \leftarrow (x' = i \wedge y' = j \wedge z' = i + 1 \wedge t' = j + 1)$;
      return $(\mathfrak{P} \cup \{P_+(A, B, C, x', y', z', t')\}, \mathfrak{a} \sqcap_{\mathcal{B},\mathcal{V}} I)$
    else
      return $(\mathfrak{P}, \mathfrak{a})$
    end
  else
    return $(\emptyset, \mathfrak{a})$
  end
end

Algorithm 2: $\mathbf{Res}_{+,\mathfrak{M}}$, resizes possible predicate of a monomial

Input : $(\mathfrak{P}, \mathfrak{a})$
Output: $(\mathfrak{P}', \mathfrak{a}')$ resized state

$(\mathfrak{P}_o, \mathfrak{a}_o) \leftarrow (\mathfrak{P}, \mathfrak{a})$;
while $\exists (P_0, P_1) \in \mathfrak{P}_o$, FIND_RESIZE$(P_0, P_1, \mathfrak{a}_o) \neq$ *None* **do**
  $P_+(A, B, C, x_0, y_0, z_0, t_0) = P_0$;
  $P_+(A, B, C, x_1, y_1, z_1, t_1) = P_1$;
  $x', y', z', t' \leftarrow$ **fresh**();
  $I \leftarrow ((x' = x_0) \wedge (y' = y_0) \wedge (z' = z_1) \wedge (t' = t_1))$;
  $(\mathfrak{P}_o, \mathfrak{a}_o) \leftarrow ((\mathfrak{P}_o \setminus \{P_0, P_1\}) \cup P_+(A, B, C, x', y', z', t'), \mathfrak{a}_o \sqcap_{\mathcal{B},\mathcal{V}} I)$
end
return $(\mathfrak{P}_o, \mathfrak{a}_o)$

Algorithm 3: FIND_EXTENSION, finds possible extension of a monomial

Input : $(P, \mathfrak{a})$,A,B,C,i,j
Output: `None` ‖ `Some(dir)`: the direction in which the rectangle can be extended

```
P_+(A', B', C', x, y, z, t) = P;
if A = A' ∧ B = B' ∧ C = C' then
    if a ⊑_{B,V} ((z = i) ∧ (y = j) ∧ (t = j + 1)) then
        Some(Right)
    else if a ⊑_{B,V} ((x = i) ∧ (z = i + 1) ∧ (t = j)) then
        Some(Down)
    else if a ⊑_{B,V} ((x = i + 1) ∧ (y = j) ∧ (t = j + 1)) then
        Some(Left)
    else if a ⊑_{B,V} ((x = i) ∧ (z = j) ∧ (y = j + 1)) then
        Some(Up)
    else
        None
else
    None
end
```

Algorithm 4: FIND_RESIZE, finds possible resize among two predicates in a monomial

Input : $P_0, P_1, \mathfrak{a}$
Output: `None` ‖ `Some(dir)`: the direction in which the two rectangles can be merged

```
P_+(A', B', C', x_0, y_0, z_0, t_0) = P_0;
P_+(A'', B'', C'', x_1, y_1, z_1, t_1) = P_1;
if A'' = A' ∧ B'' = B' ∧ C'' = C' then
    if a ⊑_{B,V} ((x_0 = x_1) ∧ (z_0 = z_1) ∧ (t_0 = y_1)) then
        Some(Right)
    else
        if a ⊑_{B,V} ((y_0 = y_1) ∧ (t_0 = t_1) ∧ (z_0 = x_1)) then
            Some(Down)
        else
            None
        end
    end
else
    None
end
```

$\mathbf{Post}^\sharp_{+,\mathfrak{M}}$. The computation of $\mathbf{Post}^\sharp_{+,\mathfrak{M}}(\mathfrak{P}, \mathfrak{a}, A, B, C, X_1, X_2)$ is described in Algorithm 1. It starts by looking whether one of the predicates stored in $\mathfrak{P}$ (the variables of which are bound by $\mathfrak{a}$) can be geometrically extended using the fact that the cell (X_1, X_2) (also bound by $\mathfrak{a}$) now also contains the addition of B and C. This helper function is detailed in Algorithm 3: we only have to test a linear relation among variables in the numerical domain. In this case, the variables in $\mathfrak{a}$ are rebound to fit the new rectangle. If no such predicate is found and $\mathfrak{P}$ already contains M_{pred} predicates then $\mathbf{Post}^\sharp_{+,\mathfrak{M}}$ gives back $(\mathfrak{P}, \mathfrak{a})$. If no such predicate is found but $\mathfrak{P}$ contains less than M_{pred} predicates, then a new predicate is added to $\mathfrak{P}$, stating that the square $(X_1, X_2, X_1 + 1, X_2 + 1)$ contains the addition of B and C. Finally, the other cases in the algorithm ensure that all predicates of the abstract state are describing the same matrices. The soundness of $\mathbf{Post}^\sharp_{+,\mathfrak{M}}$ comes from the fact that we extend a predicate only in the cell where an addition has just been performed and from the test on line 15 in Algorithm 1 that ensures that if we start to store the sum of some newly encountered matrices in a matrix where some predicates held, then all the former predicates are removed.

$\mathbf{Res}_+$. The function $\mathbf{Res}_+$ tries to merge two predicates when they correspond to two adjacent rectangles, the union of which can be exactly represented as a larger rectangle (the resize conditions are also given by linear relations). A description of a function $\mathbf{Res}_{+,\mathfrak{M}}$ can be found in Algorithm 2, with the help of Algorithm 4, and $\mathbf{Res}_+$ is the application of $\mathbf{Res}_{+,\mathfrak{M}}$ to every monomial in the abstract state considered. The soundness of $\mathbf{Res}_+$ comes from the soundness of the underlying numerical domain and the tests performed by FIND_RESIZE.

Theorem 1. *The analyzer defined by the* $\mathbf{Post}^\sharp$ *function is sound, in the sense that it over-approximates the reachable states of the program:*

$$\forall C \in \mathbb{C},\ \forall S^\sharp,\ \mathbf{Post}[\![C]\!](\gamma_p(S^\sharp)) \subseteq \gamma_p(\mathbf{Post}^\sharp[\![C]\!](S^\sharp))$$

The idea of the proof is to show that the proposed functions $\mathbf{Post}^\sharp_{+,\mathfrak{M}}$ and $\mathbf{Res}_{+,\mathfrak{M}}$ are sound, and to underline that the termination of the analysis of the `while` loop is ensured by a convergence of the shape of the abstract states.

4.2 Matrix multiplication

Consider Program 1.3 that implements a matrix multiplication. It employs two kinds of matrix assignments: $E_1 \triangleq A[X_1][X_2] \leftarrow c \in \mathbb{R}$ and $E_2 \triangleq A[X_1][X_2] \leftarrow A[X_1][X_2] + B[X_1][X_3] \times C[X_3][X_2]$, the first one being used as an initialization, and the other one to accumulate partial products. To achieve a modular design, we naturally associate to each kind of assignments a kind of predicates, and show how these predicates interact. More precisely, in our case, we consider the two predicates:

$$P_s(A, x, y, z, t, c) \triangleq \forall i, j \in [x, z-1] \times [y, t-1], A[i][j] = c$$

$$P_\times(A, B, C, x, y, z, t, u, v) \triangleq \forall i, j \in [x, z-1] \times [y, t-1],\ A[i][j] = \sum_{k=u}^{v} B[i][k] \times C[k][j]$$

```
n >= 1;
i := 0;
while (i < n) do
  j := 0;
  while (j < n) do
    A[i][j] <- 0;
    j := j +1
  done;
  i := i +1
done;
i := 0;
while (i < n) do
  j := 0;
  while (j < n) do
    k := 0;
    while (k < n) do
      A[i][j]<-A[i][j]+
        B[i][k]*C[k][j];
      k := k + 1;
    done
    j := j + 1;
  done
i := i + 1;
done
```

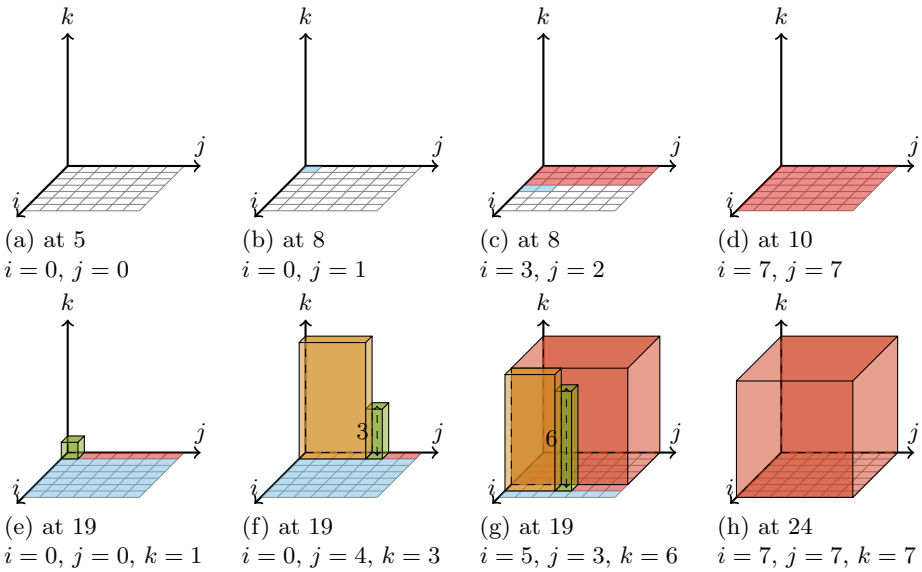
Program 1.3: Multiplication with inner loop on k.

which state respectively that the matrix A has been set to c on some rectangle, and that the matrix A received a partial product of B and C.

Predicates can interact together in two ways. Firstly, in a non productive way, for example an addition is performed in a matrix A and then the matrix A is reset to 0. In order to ensure soundness, we need to remove the predicate stating that A received an addition. Secondly, in a productive way, meaning that a matrix A is set to 0 as a prerequisite to receiving the product of two matrices, by summation over k of the partial products $B[i][k] \times C[k][j]$.

Removing predicates. The analysis suggested for the addition in terms of postconditions can be extended to the set predicate P_s the same way. Indeed, we add a function $\mathbf{Post}^\sharp_{s,\mathfrak{M}}$, that enlarges the predicates $P_s(A, x, y, z, t, c)$ and a function $\mathbf{Res}_{s,\mathfrak{M}}$, that resizes them when possible, and likewise $\mathbf{Post}^\sharp_{\times,\mathfrak{M}}$ and $\mathbf{Res}_{\times,\mathfrak{M}}$ for predicate $P_\times$ (it is done the same way as the addition predicate). However, for the analyzer to be sound, we need to ensure that $\mathbf{Post}^\sharp_{\times,\mathfrak{M}}$ and $\mathbf{Post}^\sharp_{s,\mathfrak{M}}$ check whether the matrix that was modified (A) by the evaluated command (resp. $A[X_1][X_2] \leftarrow B[X_1][X_2] + C[X_1][X_2]$ and $A[X_1][X_2] \leftarrow c$) appears in some other predicate P. If it is the case, then P is removed from the state, thus loosing information but ensuring soundness.

Splitting predicates. In Program 1.3, matrix `A` is set to 0 before the main loop. This is necessary if we want to compute the product of `B` and `C` into `A`. Therefore, we can only assert $P_\times(A, B, C, i, j, i+1, j+1, 0, 1)$ after a command `A[i][j] <- A[i][j] + B[i][k]*C[k][j]` if some precondition states that $A[i][j] = 0$ (e.g., a P_s predicate). Therefore the postcondition $\mathbf{Post}^\sharp_{\times,\mathcal{M}}$ checks whether a predicate states that $A[i][j] = 0$. If no such predicate exists, we can not produce a multiplication predicate. If it exists, then we can but we have to split the zero predicates. In order to illustrate this case, Fig. 7 depicts the evolution of the different predicates during the analysis of Program 1.3. Notice that we have only drawn the evolution of predicates that will lead to a successful result (meaning that, among all the possible states the matrix can be according to the abstract state, Fig. 7 only depicts the most advanced one, i.e., the one that is the most



(a) at 5
$i = 0,\ j = 0$

(b) at 8
$i = 0,\ j = 1$

(c) at 8
$i = 3,\ j = 2$

(d) at 10
$i = 7,\ j = 7$

(e) at 19
$i = 0,\ j = 0,\ k = 1$

(f) at 19
$i = 0,\ j = 4,\ k = 3$

(g) at 19
$i = 5,\ j = 3,\ k = 6$

(h) at 24
$i = 7,\ j = 7,\ k = 7$

Fig. 7: Evolution of the predicates for Program 1.3. The predicates are, in order of apparition: $P_s(A, B, C, \dots)$ (in b, c, e, f, g), $P_s(A, B, C, \dots)$ (in c, d, e, f, g), $P_\times(A, B, C, \dots)$ (in e, f, g), $P_\times(A, B, C, \dots)$ (in f, g), $P_\times(A, B, C, \dots)$ (in g, h).

precise about the contents of the matrix). An abstract state is the superposition of all possible states, hence not only those shown in Fig. 7.

Finally, we notice that the variables occuring in the $P_\times$ predicate mentioned above do not have the same role. Indeed, x, y, z, t variables denote a localization in the matrix while u, v variables depict the evolution of the multiplication. To be able to analyze matrix multiplicating programs where loops on k and i have been interchanged we had to introduce new predicates including existential quantification over matrices, which we do not detail here for lack of space.

5 Function calls

In this section, we extend our analysis to support analyzing function calls in a modular way. Our goal is to be able to efficiently analyze larger programs, where different functions need different predicates to prove the correctness of the whole program. In order to do so, we do not want to have to inline and prove again the correctness of each function when called. Therefore, we propose a method to compute pre and postconditions of functions and achieve a modular inter-procedural analysis. For the sake of presentation, function arguments are limited to matrices (we omit the handing of scalar function arguments as this is standard). Moreover, we assume that we are given a domain that can express symbolic equalities between matrices, which is useful to bind formal and actual

matrix arguments during function calls, but the presentation of which we also omit for the sake of concision (and as it does not present additional difficulties).

Function calls. We will store the result of the analysis of a function for further use. However, the context in which functions are called may differ from one call to another. Those differences can be either the size of the matrices or the contents of the matrices. In our analyzer, pre and postconditions are expressed as abstract states: when the analysis of a function f is made, starting from an abstract state $S_i^\sharp$ yielding a postcondition $S_o^\sharp$, we store $(f, S_i^\sharp, S_o^\sharp)$. Then, when a call to f is made under a context $S^\sharp$, we test whether $S^\sharp \sqsubseteq S_i^\sharp$, in which case $\gamma(S^\sharp) \subseteq \gamma(S_i^\sharp)$ therefore $\mathbf{Post}(\gamma(S^\sharp)) \subseteq \mathbf{Post}(\gamma(S_i^\sharp))$ and the soundness of the analyzer gives us $\mathbf{Post}(\gamma(S^\sharp)) \subseteq \gamma(\mathbf{Post}^\sharp(S_i^\sharp))$ and finally $\mathbf{Post}(\gamma(S^\sharp)) \subseteq \gamma(S_o^\sharp)$. Therefore, when $S^\sharp \sqsubseteq S_i^\sharp$ holds, a possible postcondition is $S_o^\sharp$, which enables us to avoid reanalyzing the function. In order for this method to be efficient, we need to store elements $(S_i^\sharp, S_o^\sharp)$ such that $S_i^\sharp$ is the biggest possible, so that it covers many different calling contexts for the function, but small enough, so that the evaluation of the function produces an interesting state $S_o^\sharp$. To ensure this, in our implementation, we chose to enlarge the entry context $S_i^\sharp$ found at call sites in the following way:

- Conditions of the form $A.\mathtt{n} = \mathtt{n} \in \mathbb{N}^*$ stored in the ground domain $\mathfrak{a}$ are replaced with $A.\mathtt{n} \geq 1$. But we keep (in)equalities between the sizes of matrices. That way, we generalize the precondition in order to be able to reuse the analysis on matrices of different sizes.
- Predicates on the matrices are forgotten.

These choices can prevent us from analyzing precisely programs we could analyze using the non-modular analysis of Sect. 4 by inlining functions; however, it makes the efficient and modular analysis easier to perform. Finding more clever abstractions of the calling context that are precise enough to enable the analysis and that maximize the possibility of reuse is a hard problem on which future work is required.

The aliasing problem. To present the aliasing problem, let us start by looking at Program 1.4. The analyzer proposed in Sect. 3.1 will not conclude that this program is storing in A the addition of the initial value of B and A (because when a command $A[X_1][X_2] \leftarrow B[X_1][X_2] + C[X_1][X_2]$ is encountered, it needs to check that $A \neq B \wedge A \neq C$). However, we would like our analyzer to conclude on Program 1.5 that at the end of function `main` we have $A = B + B + C$ where $+$ is the matrix addition. Firstly, let us note that the analyzer will reanalyze the code of a function each time it is called with a different aliasing. Indeed, the analysis made for a call `add(A,B,C)` is not the same as the analysis that will be made for `add(A,B,A)`. Note that each aliasing is analyzed only once and there are a finite number of possible aliasing (i.e., partitions of $\{A, B, C\}$).

```
i := 0;
while (i < n) do
  j := 0;
  while (j < n) do
    A[i][j] <- B[i][j] +
          A[i][j];
    j := j + 1;
  done
i := i + 1;
done
```

Program 1.4: Addition.

```
1  function add(A,B,C) {
2    i := 0;
3    while (i < A.n) do
4      j := 0;
5      while (j < A.n) do
6        A[i][j] <- B[i][j] + C[i][j];
7        j := j + 1;
8      done
9    i := i + 1;
10   done
11 }
12 function main () {
13   add(D,E,F)
14   add(A,B,C);
15   add(A,B,A)
16 }
```

Program 1.5: Addition with aliasing.

Callee analysis. We consider new predicates, called equality predicates: $Eq(A, B, x, y, z, t) = \forall a, b \in [x, z-1] \times [y, t-1],\ A[a][b] = B[a][b]$ that we will use in the analysis of the callee. In order to analyze a function call `function` $f(A_0, \ldots, A_{n-1}) = \{\ C\ \}$, we perform two substitutions in the code C of the callee: a first one to match the semantic of the function call ($(\lambda A.C)B \rightarrow C[B/A]$) and a second one which transforms every matrix name (B) in the body of the callee into an auxiliary name (B'). We add equality predicates stating that those two matrices are the same ($B = B'$) at the entry of the function. When a read is made in an auxiliary version (B') and the equality predicates specify that the two matrices are identical ($Eq(B, B', \ldots)$) we can state that the read was made in the original matrix (B). When a write is made to a matrix, we destroy the equality predicate in the corresponding cell. This method gives us:

- Input/Output relations between matrices (expressed as symbolic or predicate relations between matrices and their auxiliary versions).
- What matrices were unmodified by the function call (as matrices are passed by reference to the functions, knowing it was not modified is necessary if we want to ensure that relations existing in the caller before the function call are still holding).

Example 4. Let us consider Program 1.5. The function `add` is analyzed twice, because of the two different aliasing. A first analysis of `add` is performed on line 13, this concludes that `add(G,H,I)` stores the sum of `H` and `I` in `G` and leaves `H` and `I` unchanged. Therefore on line 14, no new analysis of `add` is performed as we are able to reuse the first analysis. However, on line 15, we need to perform a new analysis as no stored analysis result can be found to match the arguments. We are able to conclude at the end of the analysis of `main` that `E,F,B,C` were not modified, that `A=B+B+C`, and `D=E+F`.

The method proposed in this section can also be exploited to accelerate the analysis of nested loops. Traditional abstract interpretation by induction on the

syntax requires reanalyzing inner loops completely for each iteration of outer loops, which is very costly for deeply nested loops. By considering each loop level as a module to be analyzed separately, we can compute pre and postconditions that are expressive enough, so that each evaluation of the loop body is reduced to a table lookup. This was used to improve the performance of our analyzer (Sect. 6).

6 Implementation

```
1   /* n >= 10 */;
2   /* n = 32 * a + b */
3   /* 1 <= b < 32 */
4   Array A of n n;
5   Array B of n n;
6   Array C of n n;
7   i := 0;
8   while (i <= a) do
9     j := 0;
10    while (j <= a) do
11      ii := 0;
12      if (i = a) then
13        endi := 32
14      else
15        endi := b;
16      while (ii < endi) do
17        jj := 0;
18        if (j = b) then
19          endj := 32
20        else
21          endj := b;
22        while (jj < endj) do
23          x = 32 * i + ii;
24          y = 32 * j + jj;
25          A[x][y] <- B[x][y] +
                C[x][y];
26          jj := jj + 1;
27        done;
28        ii := ii + 1;
29      done;
30      j := j + 1;
31    done;
32  i := i + 1;
33  done
```

Program 1.6: Addition tiled with reminder.

Results. We implemented the algorithms proposed above as well as various improvements to make the analysis more robust, notably symbolic equality domains [15] able to improve the pattern matching used in the assignment transfer function. Our prototype was able to prove the functional correctness of all the programs mentioned earlier (addition and multiplication). We also analyzed tiled versions of these algorithms, which is a classic optimization (performed by hand or automatically) that increases cache efficiency, but makes the algorithm more difficult to understand. Program 1.6 gives one example of optimized matrix addition, with a tiling factor of 32. Our method successfully analyzes the tiled algorithms, as long as the tiling size is a constant, which is the case for all the optimizers we know of. Note that the tiling transformation causes a doubling in the depth of nested loops and adds some complex conditionals to handle border cases (partially filled tiles), resulting in a more challenging program to analyze. Additionally, we analyzed alternate versions where loops are interchanged, or indices run in decreasing order (from $n-1$ to 0). In all those cases, the analyzer still proves the functional correctness of the program. All of those programs were analyzed using only few predicates, thus showing that multiple versions

	2 loops	3 loops	4 loops	6 loops
Set to 0	0.022			
Addition	0.042		0.772	50.9 (3.36 using §5)
Multiplication		0.232		110.0
Addition with rest			1.389	
GEMM		0.54		

Fig. 8: Analysis time in seconds.

of a program can be analyzed using a single predicate. Hence, the analysis is robust against all the following transformations: loop tiling, switching the loops (row-major or column-major iteration), reverting the loops (iterating downward instead of upward).

Implementation. The implementation of the different analyzers has been written in `Ocaml` using the `Apron` module (for numerical domains). The final implementation using all proposed abstract domains is about 6000 line long (not counting the parser). It enables us to parse code written in a C like language and analyze this code using the previously mentioned abstractions. This analyzer computes the abstract reachability at every code point depending on the initial abstract states. The implementation has been tested mainly on programs performing additions, products, GEMM (general matrix multiplication: $C \leftarrow \alpha AB + \beta C$), scaling, with various optimizations and on every possible loop order. As examples, we mostly used programs optimized by Pluto and programs we found in a BLAS library (Basic Linear Algebra Subprograms). Those programs are all successfully analyzed by the final implementation. Figure 8 gives the time it took for the analyzer to prove the functional correctness of programs proposed in this article. The number of loops indicated is the biggest number of interlocked while loops that can be found in the program (additions and multiplications require respectively 2 and 3 nested loops without tiling, and 4 and 6 nested loops with tiling enabled, addition tiled twice require 6 loops). For the (twice) tiled addition, we also show the benefit of the loop-modular analysis of Sect. 5.

Trace partitioning. In order to make the analysis more precise, we used trace partitioning [17] to separate some monomials according to their history. We recall that abstract states are of the form $(\mathfrak{P}, \mathfrak{a})$ and that we ensured a finite number of possible shapes for monomials by bounding the number of predicates the analyzer can use. Hence, we modified abstract states to be of the form $(\mathfrak{i}, \mathfrak{P}, \mathfrak{a})$ where $\mathfrak{i} \in \mathbb{N}$ is called a *flag*. We now say that two monomials have the same shape if they have the same flag and they have the same shape according to the previous definition. We encode in a flag $\mathfrak{i}$ the program path taken by each monomial along the analysis (whether it has been through each loop or not and in which branch of a condition it has been).

Numerical abstract domain. The abstract states defined in this paper require an underlying numerical domain. Some of the numerical variables stored in the numerical domain are used to describe "zones" in a matrix where some relations hold. As we wanted our analyzer to be robust against operations such as tiling we needed relations such as $32i - j = 0$ and $32i - j \leq 0$ to be precise, therefore we chose the polyhedra domain [9]. As most of the operations performed by our analyzer are done on the numerical abstract domain, the cost of the analysis depends mainly on the cost of the operation in the underlying numerical domain. However for a different set of predicates (e.g., not describing rectangular shapes), a less expensive numerical domain could be used (such as intervals or octagons).

7 Related work

A standard way to efficiently handle possibly unbounded arrays when a low level of precision is sufficient is to abstract arrays as a single cell containing a non-relational abstract value, using weak update. As the static analysis of array properties has drawn some significant attention lately, more precise methods have been devised. Gopan et al. [11] extend this standard method by allowing relational (but still uniform) abstractions. A class of analyses [12,13,8] dynamically partitions arrays, which allows expressing non-uniform properties of arrays as well as strong updates. Allamigeon has designed companion numeric domains specifically targeting array partition bounds [1]. String analysis for C programs can be seen as a special case of array analysis, and similar partitioning-based methods have been proposed [2]. All these methods differ on whether a partition or a covering is used, how partition bounds are expressed and inferred, the relationality between partition contents and partition bounds, etc. Fluid updates [10] address the problem of weak updates in a different way, by expressing array parts using constraints, manipulated by a SMT solver, instead of explicit partitions. Monniaux et al. [16] propose an original method by abstraction through ad-hoc Galois connections into purely scalar programs that are then analyzed with standard methods. Our work is much closer to parametric predicate abstraction [6], which also analyzes arrays using predicates of fixed shape conjoined with numeric properties. The large majority of these works only focus on properties of unidimensional arrays. Nevertheless some array abstractions are powerful enough to consider array elements of arbitrary types, and could be possibly nested to handle matrices as arrays of arrays of numbers (this is explicitly mentioned as a possibility in [8] but without details nor experiments) while the method of [16] can analyze matrix initialization (and possibly more) when parameterized with the correct predicate. As far as we know, none of these methods has been applied to prove matrix multiplication algorithms correct, nor do they address the problem of deeply nested loops in optimized matrix algorithms.

8 Conclusion

We have proposed new abstractions to prove the functional correctness of matrix-manipulating programs, such as addition and multiplication. They are parametric in a set of predicates (corresponding to the functional properties to prove) and classic relational numeric domains. Our prototype implementation provided encouraging experimental results, both in term of performance and precision. In particular, precision-wise, we could prove the correctness of several variants of additions and multiplications, including basic loop reordering but also versions generated from a source-to-source loop optimizer which introduces tiling (making the code significantly more complex, with in particular more nested loops). We also showed how our method can be embedded in a modular inter-procedural analysis, with clear benefits for the efficiency of the analysis.

Future work will include enriching our predicates in order to tackle a wider range of matrix and vector manipulations, such as Gauss elimination, LU decomposition, linear system resolution, eigenvectors, or eigenvalues computation, etc. Another direction for future work is to make the analysis robust against more varied program transformations and optimizations, such as instruction-level reordering, loop pipelining, or vectorization. Ultimately, we would like to be able to analyze the assembly or low-level representation output by a compiler, and prove that the functional correctness still holds despite compiler optimization, without the burden of certifying the optimizing part of the compiler itself. Our analysis currently assumes a real semantics, while actual implementations of matrix operations employ floating-point arithmetic. Hence, with respect to a float implementation, we prove that, e.g., a matrix addition program computes one float approximation of the matrix addition, but not that it does so while respecting the order of operations specified in the original, unoptimized program (and this may change the result due to rounding errors). While we believe that this already provides an interesting correctness criterion (especially because matrix libraries seldom guarantee an evaluation order), future work will include designing predicates reasoning on floats in order to take execution order and rounding into account. Finally, we have only demonstrated our method on a simple prototype for a toy language, but future work will consider more realistic C programs, such as actual BLAS libraries or scientific applications.

References

1. Xavier Allamigeon. Non-disjunctive numerical domain for array predicate abstraction. In *ESOP 2008*, volume 4960 of *LNCS*, pages 163–177. Springer, 2008.
2. Xavier Allamigeon, Wenceslas Godard, and Charles Hymans. Static analysis of string manipulations in critical embedded C programs. In *SAS 2006*, pages 35–51. Springer, 2006.
3. Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proc. of PLDI'03*, pages 196–207. ACM, Jun. 2003.

4. Uday Bondhugula, Muthu Manikandan Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *CC 2008*, volume 4959 of *LNCS*, pages 132–146. Springer, 2008.
5. Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proc. of PLDI 2008*, pages 101–113. ACM, 2008.
6. Patrick Cousot. Verification by abstract interpretation. In *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, volume 2772 of *LNCS*, pages 243–268. Springer, 2003.
7. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL 1977*, pages 238–252. ACM, 1977.
8. Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Proc. of POPL 2011*, pages 105–118. ACM, 2011.
9. Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. POPL 1978*, pages 84–96. ACM, 1978.
10. Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid updates: Beyond strong vs. weak updates. In *ESOP 2010*, pages 246–266. Springer, 2010.
11. Denis Gopan, Frank DiMaio, Nurit Dor, Thomas Reps, and Mooly Sagiv. Numeric domains with summarized dimensions. In *TACAS 2004*, pages 512–529. Springer, 2004.
12. Denis Gopan, Thomas W. Reps, and Shmuel Sagiv. A framework for numeric analysis of array operations. In *Proc. POPL 2005*, pages 338–350. ACM, 2005.
13. Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. *SIGPLAN Not.*, 43(6):339–348, June 2008.
14. Xavier Leroy. Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In *Proc. POPL 2006*, pages 42–54. ACM, 2006.
15. Antoine Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *VMCAI 2006*, volume 3855 of *LNCS*, pages 348–363. Springer, 2006.
16. David Monniaux and Francesco Alberti. A simple abstraction of arrays and maps by program translation. In *SAS 2015*, volume 9291 of *LNCS*, pages 217–234. Springer, 2015.
17. Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007.
18. Arnaud Venet. Abstract cofibered domains: Application to the alias analysis of untyped programs. In *SAS'96*, volume 1145 of *LNCS*, pages 366–382. Springer, 1996.