Form Methods Syst Des (2018) 53:221–258
https://doi.org/10.1007/s10703-017-0311-x

Inferring functional properties of matrix manipulating programs by abstract interpretation

Matthieu Journault[1] · **Antoine Miné**[1]

Published online: 12 February 2018

Abstract We present a new static analysis by abstract interpretation to prove automatically the functional correctness of algorithms implementing matrix operations, such as matrix addition, multiplication, general matrix multiplication, inversion, or more generally Basic Linear Algebra Subprograms. In order to do so, we introduce a family of abstract domains parameterized by a set of matrix predicates as well as a numerical domain. We show that our analysis is robust enough to prove the functional correctness of several versions of the same matrix operations, resulting from loop reordering, loop tiling, inverting the iteration order, line swapping, and expression decomposition. 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.

Keywords Abstract interpretation · Static analysis · Matrix manipulating programs · Functional correctness

1 Introduction

Static analysis by abstract interpretation [1] 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,

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

✉ Matthieu Journault
matthieu.journault@lip6.com

Antoine Miné
antoine.mine@lip6.com

[1] Laboratoire d'informatique de Paris 6 (LIP6), Sorbonnes Universités, UPMC Univ Paris 6, 4, place Jussieu, 75252 Paris Cedex 05, France

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 [2]. More recently, it has been extended to proving functional properties, including array properties [3–5], such as proving that a sorting algorithm indeed outputs a sorted array. In this work, we consider functional properties of a different kind, rarely tackled before (see related works in Sect. 8): properties on matrices. Consider as example Program 1 (see Sect. 4.4 for a presentation of the results of our analyzer on this program) starting with the assumption $N \geq 5$. Our analyzer will automatically infer the following postcondition for the program: $\forall u, v \in [0, N-1]^2,\ C[u][v] = \sum_{w=0}^{N-1} A[u][w] \times B[w][v]$, i.e., the program indeed computes the product of matrices A and B into C. Consider now Program 2 where `multiplication(C,A,B)` is Program 1. It computes the sequence $A = B^m$ using function calls to a matrix manipulating function. Iterative constructions such as this one are useful in the field of matrix analysis (eigenvalues, eigenvectors, matrix inversion computation), therefore it is important to infer and prove automatically that such programs are indeed computing the expected sequence.

```
1  /* N >= 5 */
2  for (i=0; i<N; i++)
3   for (j=0; j<N; j++) {
4    C[i][j] = 0;
5    for (k=0; k<N; k++)
6     C[i][j] += A[i][k] * B[k][j];
```

Program 1 Matrix multiplication $C = A \times B$

```
1  /* m >= 1 */
2  l=0;
3  while (l<m) {
4   multiplication(A,A,B);
5   i++
6  }
```

Program 2 Computing the sequence $A = B^m$

1.1 Introductory example

To explain how our method works in more details, we focus on a simpler introductory example, the matrix addition from Program 3 (see Sect. 4.4 for a presentation of the results of our analyzer on this program). Note that we will show later that our method is also successful on the multiplication from Program 1, as well as more complex, optimized variants of these algorithms, such as the tiled addition in Program 12.

We wish to design a sound analyzer capable of inferring that, at the end of 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 $\phi \stackrel{\Delta}{=} \forall a,\ \forall b,\ (0 \leq a < N \wedge 0 \leq b < N) \Rightarrow C[a][b] = A[a][b] + B[a][b]$ holds for all the memory states reachable at line 10.

In order to do so, the static analyzer needs to infer loop invariants for each of the `while` loops. For the outermost loop (line 3), we would infer: $\phi_i \stackrel{\Delta}{=} \forall a,\ \forall b,\ (0 \leq a < i \wedge 0 \leq b < N) \Rightarrow C[a][b] = A[a][b] + B[a][b]$, 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 numerical domains, such as polyhedra [6], to reason on numeric relations. We will design a family of abstract domains combining existing numerical 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 [3–5], 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 deep levels of loop nesting, which may pose scalability issues.

1.2 Contribution

The article presents new abstract domains for the static analysis of matrix-manipulating programs. Those abstract domains are based on a combination of parametric predicates and numerical domains and can be used on small programs (e.g.computing a matrix addition) as well as on more complicated programs calling matrix-manipulating functions. The analysis has been proved sound and implemented in a prototype (supporting a small pseudo-code language enabling matrix operations). 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 [7,8]. 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

```
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 3 Matrix addition $C = A + B$

optimizer; therefore, our method could be used in combination with certified compilers, such as CompCert [9], while avoiding the need to certify optimization passes in Coq. Indeed, at the time of writing it is not yet possible to perform loop tiling optimizations using CompCert. 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 numerical domain. Moreover we added to our analyzer the ability to use some helper domains, making the process of defining new predicates and adding them to the analyzer more developer-friendly. Finally, the performance of our analyzer is quite encouraging.

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 and Sect. 6 exposes a small abstract domain that enables us to analyze iterative processes on matrices. This section concludes with Sect. 6.2 that presents the analysis of an example program that exploits the combination of all the abstract domains presented in this article. Sect. 7 presents our implementation and our experimental results. Section 8 discusses related works, while Sect. 9 concludes. The appendix contains proofs of the soundness of our analyzer ("Appendix B") and the source code of some programs mentioned in the article ("Appendix C").

This article is an extended version of a paper published at SAS '16 (see [10]). Sections 3 and 4 have been extended with examples and explanations. Section 6 presents new materials. The soundness proof (in "Appendix B") is also a new addition.

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 $\mathbb{S} = \{A.\mathtt{n} \mid A \in \mathbb{A}\} \cup \{A.\mathtt{m} \mid A \in \mathbb{A}\}$, 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; there-

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

of a matrix A of size $E_1 \times E_2$.

Fig. 1 Syntax of the language

fore, in the rest of the document we will try to propose a computable approximation. The soundness property of Sect. 4.1 holds with respect to this concrete semantic.

2.2.1 Definitions

$\mathcal{D}$ 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 $\mathcal{M} = \mathcal{M}_\mathcal{V} \times \mathcal{M}_\mathcal{A}$ containing a scalar environment $\mathcal{M}_\mathcal{V} \triangleq \mathcal{V} \to \mathcal{D}$ and a matrix contents environment $\mathcal{M}_\mathcal{A} \triangleq \mathcal{A} \to (\mathbb{N}\times\mathbb{N}) \to \mathcal{D}$ (here we are not interested in proving that there is no out of bounds access, therefore $A[X_1][X_2]$ is defined as soon as $X_1, X_2 \in \mathbb{N}$), 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, 3 and 4 describe the behavior of expressions, booleans and the state reachability of the program.

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

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

- $\mathcal{B}[\![E_1 < E_2]\!] \triangleq \{m \in \mathcal{M} \mid \mathcal{E}[\![E_1]\!]m < \mathcal{E}[\![E_2]\!]m\}$
- $\mathcal{B}[\![E_1 \leq E_2]\!] \triangleq \{m \in \mathcal{M} \mid \mathcal{E}[\![E_1]\!]m \leq \mathcal{E}[\![E_2]\!]m\}$
- $\mathcal{B}[\![E_1 = E_2]\!] \triangleq \{m \in \mathcal{M} \mid \mathcal{E}[\![E_1]\!]m = \mathcal{E}[\![E_2]\!]m\}$
- $\mathcal{B}[\![\neg E_1]\!] \triangleq \mathcal{M} \setminus \mathcal{B}[\![E_1]\!]$

Fig. 3 Evaluation of booleans

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{Post}[\![C]\!](S) \triangleq \text{match } C \text{ with:}} \\
\quad | & \mathtt{skip} & \to S \\
\quad | & X := E & \to \{(m_\mathcal{V}[X := \mathcal{E}[\![E]\!]m], m_\mathcal{A}) \mid m \in S\} \\
\quad | & A[X_1][X_2] \leftarrow E & \to \{(m_\mathcal{V}, m_\mathcal{A}[A := \lambda(i,j).\ \begin{cases} \mathcal{E}[\![E]\!]m & \text{if } (i,j) = (u,v) \\ m_\mathcal{A}(A)(i,j) & \text{otherwise} \end{cases}]) \\
 & & \quad \mid m \in S \wedge (m_\mathcal{V}(X_1), m_\mathcal{V}(X_2)) = (u,v) \in \mathbb{N}^2\} \\
\quad | & \mathtt{If}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2 & \to \mathbf{Post}[\![C_1]\!](S \cap \mathcal{B}[\![B]\!]) \cup \mathbf{Post}[\![C_2]\!](S \cap \mathcal{B}[\![\neg B]\!]) \\
\quad | & \mathtt{While}\ B\ \mathtt{do}\ C_1\ \mathtt{done} & \to \mathbf{lfp}^{\subseteq}(\lambda S_0.S \cup \mathbf{Post}[\![C_1]\!](S_0 \cap \mathcal{B}[\![B]\!])) \cap \mathcal{B}[\![\neg B]\!] \\
\quad | & \mathtt{Array}\ A\ \mathtt{of}\ E_1\ E_2 & \to \{(m_\mathcal{V}[A.\mathtt{m} := \mathcal{E}[\![E_1]\!]m, A.\mathtt{n} := \mathcal{E}[\![E_2]\!]m], \\
 & & \quad m_\mathcal{A}[A := \lambda(i,j).0]) \quad \mid m \in S\} \\
\quad | & C_1\ ;\ C_2 & \to \mathbf{Post}[\![C_2]\!](\mathbf{Post}[\![C_1]\!](S))
\end{array}
$$

Fig. 4 Reachability semantics, m designates the pair $(m_\mathcal{V}, m_\mathcal{A})$

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

Fig. 5 Terms, atomic predicates, and predicates

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 programming 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. 5. 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 here. $\mathbf{var}(P)$ denotes the free variables of P that are in $\mathbb{Y} \cup \mathbb{X} \cup \mathbb{S}$. In the rest of the article, whenever we mention fresh variables, those will be fresh variables in $\mathbb{Y}$. The $\sum_{x=y}^{z} t$ term is not useful yet, however Sect. 4.2 will show how this construction enables us to link together multiple sub-predicates.

Example 1 Modulo some syntactic sugar, we can define a predicate such as: $P_+(A, B, C, x, y, z, t) \triangleq \forall u \in [x, z-1], \forall v \in [y, t-1], A[u][v] = B[u][v] + C[u][v]$. 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., $\{Q(A, B, C, x, y, z, t), P(A, u, v)\} =_S \{Q(A, B, C, x', y', z', t'), P(A, u', v')\}$).

In the following we will only consider sets of predicates $\mathfrak{P}$ wich are *well-named* in the sense that $\forall P \neq Q \in \mathfrak{P}, \mathbf{var}(P) \cap \mathbf{var}(Q) = \emptyset$ in that case $\mathfrak{P} =_S \mathfrak{Q}$ amounts to the existence of a renaming σ and a one-to-one relation b between $\mathfrak{P}$ and $\mathfrak{Q}$ such that $\forall P \in \mathfrak{P}, \exists Q \in \mathfrak{Q},\ P = b(Q)\sigma$. It is to be noted that there might exist more that one such one-to-one relation, this point will be detailed in the following Sect. 3.2.

3.2 Abstract states

We now assume we have an abstraction of the scalar environment $\mathcal{M}_{\mathcal{V}}$ (e.g.the interval or polyhedra [6] domain) that we call $\mathcal{M}^{\sharp}_{\mathcal{V}}$, with concretization function $\gamma_{\mathcal{V}}$, join $\cup_{\mathcal{V}}$, widening $\nabla_{\mathcal{V}}$, 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 numerical 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 numerical domain to relate predicate variables with program variables, if needed.

Let us revisit the introductory example of Program 3. At program point 5 we would like to infer the invariant $(\forall a,\ \forall b,\ (0 \leq a < i \wedge 0 \leq b < N) \Rightarrow C[a][b] = A[a][b] + B[a][b]) \wedge (\forall b,\ 0 \leq b < j \Rightarrow C[i][b] = A[i][b] + B[i][b])$, using the predicate from Example 1 we can rewrite the invariant as $P_+(C, A, B, x, y, z, t) \wedge P_+(C, A, B, x', y', z', t')$ with $\{x = y = 0 \wedge z = i \wedge t = N \wedge x' = i \wedge y' = 0 \wedge z' = i + 1 \wedge t' = j\}$. This last set of constraints can be represented by a numerical abstract domain. Such a set of predicates and the corresponding numerical abstract domain (that links predicate variables and program variables) will be called a monomial. In order to express disjunctions of states that require different numbers of predicates we will use as abstract state disjunctions of monomials.

Remark 1 The previous remarks amount to say that we want abstract states of the form:

$$
\begin{array}{lllllll}
(\mathrm{Mon}_1: & \texttt{numerical\_constraints\_1} & \wedge & \texttt{predicate\_1\_1} & \wedge & \texttt{predicate\_1\_2} & \wedge \ \ldots) \\
 & & & & \vee & & \\
(\mathrm{Mon}_2: & \texttt{numerical\_constraints\_2} & \wedge & \texttt{predicate\_2\_1} & \wedge & \texttt{predicate\_2\_2} & \wedge \ \ldots) \\
 & & & & \vee & & \\
 & & & & \vdots & & \\
(\mathrm{Mon}_n: & \texttt{numerical\_constraints\_n} & \wedge & \texttt{predicate\_n\_1} & \wedge & \texttt{predicate\_n\_2} & \wedge \ \ldots)
\end{array}
$$

where variables appearing in `predicate_i_j` are bound by the `numerical_constraints_i`

Hence, we will introduce a monomial as a conjunction of some predicates and a numerical abstract domain; 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 numerical 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 $(\mathfrak{P}, \mathfrak{a}) \in \mathfrak{M} = \wp(\mathcal{P}) \times \mathcal{M}^\sharp_{\mathcal{V}}$ 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 $\mathcal{M}^\sharp = \wp(\mathfrak{M})$. 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 $S^\sharp \in \mathcal{M}^\sharp$ as:

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

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: $\gamma(\{(\mathfrak{P}, \mathfrak{a})\})$ 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.

The previous definition ensures that well-formed abstract states only have a bounded number of monomials. Indeed there is only a finite number of available predicates, and we want to keep unbounded numerical domains (so as to be expressive enough), therefore we need to ensure that abstract elements are bounded. The **wf** operator ensures that a given set of predicate is only bound to a unique numerical domain.

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 $\exists f : S_1^\sharp \to S_2^\sharp$, a bijection, $\forall(\mathfrak{P}_1, \mathfrak{a}_1) \in S_1^\sharp, (\mathfrak{P}_2, \mathfrak{a}_2) = f((\mathfrak{P}_1, \mathfrak{a}_1)) \Rightarrow \mathfrak{P}_1 =_S \mathfrak{P}_2$, we also say that $S_1^\sharp$ and $S_2^\sharp$ have the same *shape* according to f.

Some of our operations on abstract states may not output well-formed states. We thus propose the following algorithm in order to construct a well-formed state. This algorithm may lose some precision, but is always sound.

Definition 5 (Algorithm $\mathbf{wf}(S^\sharp)$): while there is still a pair of monomials $(\mathfrak{P}_1, \mathfrak{a}_1) \neq (\mathfrak{P}_2, \mathfrak{a}_2) \in S^\sharp$ such that $\mathfrak{P}_1 =_S \mathfrak{P}_2$:

- remove $(\mathfrak{P}_1, \mathfrak{a}_1), (\mathfrak{P}_2, \mathfrak{a}_2)$ from $S^\sharp$
- find σ such that $\mathfrak{P}_1\sigma = \mathfrak{P}_2$
- add $(\mathfrak{P}_2, \mathfrak{a}_1\sigma \cup_{\mathcal{V}} \mathfrak{a}_2)$ to $S^\sharp$

Remark 2 We can note that the algorithm from Definition 5 is non deterministic. The precision of the analysis was satisfactory on the examples we wanted to analyze, therefore no particular merging heuristic was needed.

Proposition 1 $\forall S^\sharp$ *well-named* , $\gamma(S^\sharp) \subset \gamma(\mathbf{wf}(S^\sharp))$

The proof can be found in "Appendix B". It is to be noted that computing **wf** on some $S^\sharp$ does not require iterations until a fixpoint is reached, one can only iterate on every pair $(\mathfrak{P}_1, \mathfrak{a}_1), (\mathfrak{P}_2, \mathfrak{a}_2) \in S^\sharp$. Indeed $(\mathfrak{P}_2, \mathfrak{a}_1\sigma \cup_{\mathcal{V}} \mathfrak{a}_2)$ has the same shape as $(\mathfrak{P}_1, \mathfrak{a}_1)$ or $(\mathfrak{P}_2, \mathfrak{a}_2)$. **wf** transforms any well-named abstract state into a well-named, well-formed abstract state. However, we only have $\gamma(S^\sharp) \subset \gamma(\mathbf{wf}(S^\sharp))$ in general and not the equality. Indeed, we transform a symbolic disjunction into a disjunction $\cup_{\mathcal{V}}$ on the numerical domain, therefore this transformation might not be exact.

Figure 6 gives the basic operations on the sets of abstract states $\mathcal{M}^\sharp$: a join ($\sqcup$), a widening ($\triangledown$),[1] a meet with booleans ($\sqcap_{\mathcal{B}}$), and an inclusion test with booleans ($\sqsubseteq_{\mathcal{B},\mathcal{V}}$).

Remark 3 We note that both the widening and the join operators rely on an alignment of the predicates of two monomials, however there might be different such alignments as noted in Sect. 3.1. Indeed the **wf** operator transforms a disjunction of the form $(A \wedge B) \vee_1 (C \wedge D)$ (where $(A \wedge B)$ and $(C \wedge D)$ represent monomials and $\vee_1$ is a symbolic disjunction) into a monomial of the form $(A \vee_2 C) \wedge (B \vee_2 D)$ (where $\vee_2$ is a disjunction performed in the underlying numerical domain). The choice of matching predicates one-by-one to perform the

[1] $\triangledown$ 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, (see proof in "Appendix B") thus allowing the analyzer to perform a widening. This widening technique is similar to the one proposed by [11] on cofibred domains.

$$\sqcup\colon\ S_1^\sharp \sqcup S_2^\sharp \triangleq \mathbf{wf}(S_1^\sharp \cup S_2^\sharp)$$

$$\triangledown\colon\ S_1^\sharp \triangledown S_2^\sharp \triangleq \begin{cases} \{\ (\mathfrak{P}_1, \mathfrak{a}_1 \triangledown_{\mathcal{V}} \mathfrak{a}_2\sigma) \mid \mathfrak{m} = (\mathfrak{P}_1, \mathfrak{a}_1) \in S_1^\sharp\} & \text{if } S_1^\sharp, S_2^\sharp \text{ have the same} \\ \quad \text{where } (\mathfrak{P}_2, \mathfrak{a}_2) = f(\mathfrak{m}),\ \mathfrak{P}_1 = \mathfrak{P}_2\sigma & \text{shape according to } f \\ \texttt{undefined} & \text{otherwise} \end{cases}$$

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

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

Fig. 6 Operations on abstract states

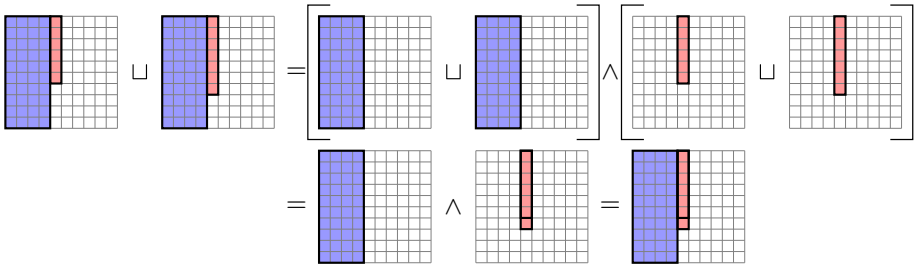

Fig. 7 The blue predicate is the first one to have appeared in the abstract state and the red predicate is the second one. (Color figure online)

disjunction ensures that the number of predicates in the monomial does not grow too large after the operation is performed. Different heuristics can be used to choose which predicates should be paired when the operation is performed. In our analyzer we decided to remember the order of apparition of predicates in each monomial, that way we could pair together predicates that appeared at the same time during the analysis. Figure 7 illustrates this choice.

3.3 Abstract transfer functions

We now describe 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 ($\mathbf{Post}^\sharp_{\mathcal{V}}[\![C]\!]$) for all instructions involving only scalar variables. In Fig. 8, the call to $\mathbf{fp}(f)(x)$ computes an over-approximation of 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: $\mathbf{Assign}^\sharp_{\mathfrak{M}}$ and **Merge**. Indeed, these functions will depend on our choice of predicates. For instance, Sect. 4.1 will introduce one version of $\mathbf{Assign}^\sharp_{\mathfrak{M}}$, named $\mathbf{Assign}^\sharp_{+,\mathfrak{M}}$, 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 matrix 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, $\mathbf{Assign}^\sharp_{\mathfrak{M}}(\ldots, E, \ldots)$ will either modify variables appearing in predicates, produce new predicates, or remove predicates. In the worst case, when $\mathbf{Assign}^\sharp_{\mathfrak{M}}$ cannot handle some expression E, it yields the approximation $\top$, thus ensuring the correctness of the analyzer.

$$\mathbf{Post}^\sharp[\![\texttt{skip}]\!](S_1^\sharp) \triangleq S_1^\sharp$$
$$\mathbf{Post}^\sharp[\![X := E]\!](S_1^\sharp) \triangleq \bigcup_{(\mathfrak{P},\mathfrak{a})\in S_1^\sharp} \{\mathfrak{P}, \mathbf{Post}^\sharp_{\mathcal{V}}[\![X := E]\!](\mathfrak{a})\}$$
$$\mathbf{Post}^\sharp[\![A[X_1][X_2] \leftarrow E]\!](S_1^\sharp) \triangleq \mathbf{wf}(\bigcup_{(\mathfrak{P},\mathfrak{a})\in S_1^\sharp} \{\mathbf{Assign}^\sharp_{\mathfrak{M}}(\mathfrak{P}, \mathfrak{a}, A, E, X_1, X_2)\})$$
$$\mathbf{Post}^\sharp[\![\texttt{If } B \texttt{ then } C_1 \texttt{ else } C_2]\!](S_1^\sharp) \triangleq$$
$$\quad \mathbf{Post}^\sharp[\![C_1]\!](S_1^\sharp \sqcap_{\mathcal{B}} B) \sqcup \mathbf{Post}^\sharp[\![C_2]\!](S_1^\sharp \sqcap_{\mathcal{B}} (\neg B))$$
$$\mathbf{Post}^\sharp[\![\texttt{While } B \texttt{ do } C \texttt{ done}]\!](S_1^\sharp) \triangleq$$
$$\quad \mathbf{Merge}(\mathbf{fp}(\lambda S^\sharp \rightarrow \texttt{ let } S_P^\sharp = S^\sharp \sqcup \mathbf{Post}^\sharp[\![C]\!](S^\sharp \sqcap_{\mathcal{B}} B) \texttt{ in}$$
$$\quad\quad \texttt{if } S_P^\sharp =_F S^\sharp \texttt{ then } S^\sharp \triangledown S_P^\sharp \texttt{ else } S_P^\sharp)(S_1^\sharp) \sqcap_{\mathcal{B}} \neg B)$$
$$\mathbf{Post}^\sharp[\![C_1 \ ; \ C_2]\!](S_1^\sharp) \triangleq \mathbf{Post}^\sharp[\![C_2]\!](\mathbf{Post}^\sharp[\![C_1]\!](S_1^\sharp))$$

Fig. 8 Abstract postconditions

The function **Merge** geometrically merges 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 **Merge** 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 [4,5]. New rewriting rules are added to **Merge** when new families of predicates are introduced.

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

3.3.1 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 **Merge** 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, experimentally setting M_{pred} to the number of nested loops is enough to successfully analyze the programs proposed in this article.

4 Abstraction instances

4.1 Matrix addition

We now consider the analysis of the assignment $E = A[X_1][X_2] \leftarrow B[X_1][X_2] + C[X_1][X_2]$, 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 $\mathbf{Assign}^\sharp_{+,\mathfrak{M}}$ and $\mathbf{Merge}_+$ of $\mathbf{Assign}^\sharp_{\mathfrak{M}}$ and **Merge** to compute postconditions and possible merges 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).

Assign$^\sharp_{+,\mathfrak{M}}$. The computation of **Assign**$^\sharp_{+,\mathfrak{M}}(\mathfrak{P}, \mathfrak{a}, A, B, C, X_1, X_2)$ is described in Algorithm 1. It starts by looking at 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 (case (a) of Fig. 9). This helper function is detailed in Algorithm 6: 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 **Assign**$^\sharp_{+,\mathfrak{M}}$ gives back $(\mathfrak{P}, \mathfrak{a})$ (case (c) of Fig. 9). If no such predicate is found but $\mathfrak{P}$ contains less than M_{pred} predicates, then a new predicate is added to $\mathfrak{P}$ (case (b) of Fig. 9), 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 **Assign**$^\sharp_{+,\mathfrak{M}}$ 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 store the sum of some newly encountered matrices in a matrix where some predicates held, then all the former predicates are removed.

Algorithm 1: Assign$^\sharp_{+,\mathfrak{M}}$, 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  : (𝔓, 𝔞),A,B,C,i,j
   Output: (𝔓′, 𝔞′) the postcondition
 1 if ∃P0 ∈ 𝔓, FIND_EXTENSION((P0, 𝔞),A,B,C,i,j) = res ≠ None then
 2 |  x′, y′, z′, t′ ← fresh();
 3 |  I ← switch res do
 4 |  |  case Some(Right)
 5 |  |  |  (x = x′ ∧ y = y′ ∧ z + 1 = z′ ∧ t = t′)
 6 |  |  case Some(Down)
 7 |  |  |  (x = x′ ∧ y = y′ ∧ z = z′ ∧ t + 1 = t′)
 8 |  |  case Some(Left)
 9 |  |  |  (x = x′ + 1 ∧ y = y′ ∧ z = z′ ∧ t = t′)
10 |  |  case Some(Up)
11 |  |  |  (x = x′ ∧ y = y′ + 1 ∧ z = z′ ∧ t = t′)
12 |  endsw
13 |  return ((𝔓 \ P0) ∪ {P+(A, B, C, x′, y′, z′, t′)}, 𝔞 ⊓B,V I)
14 else
15 |  if ∀P+(A′, B′, C′, _, _, _, _) ∈ 𝔓, A′ = A ∧ B′ = B ∧ C′ = C then
16 |  |  if ♯𝔓 < Mpred then
17 |  |  |  x′, y′, z′, t′ ← fresh();
18 |  |  |  I ← (x′ = i ∧ y′ = j ∧ z′ = i + 1 ∧ t′ = j + 1);
19 |  |  |  return (𝔓 ∪ {P+(A, B, C, x′, y′, z′, t′)}, 𝔞 ⊓B,V I)
20 |  |  else
21 |  |  |  return (𝔓, 𝔞)
22 |  |  end
23 |  else
24 |  |  return (∅, 𝔞)
25 |  end
26 end
```

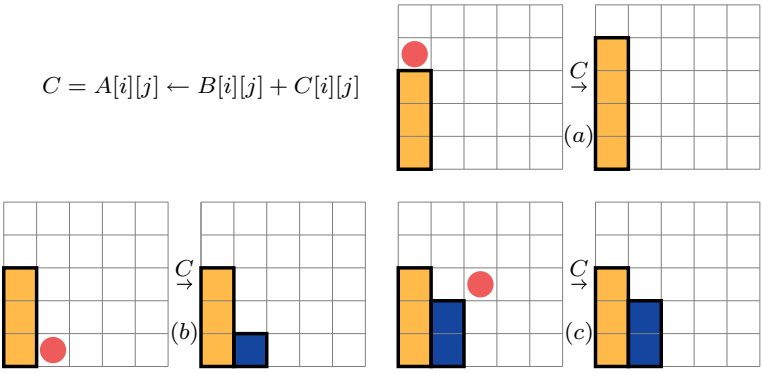

Fig. 9 Illustration of the behavior of the $\mathbf{Assign}^{\sharp}_{+,\mathfrak{M}}$ function when $M_{pred} = 2$. Yellow and blue rectangles represent predicates, the red circle represents the value (i, j). (Color figure online)

Merge$_+$. The function **Merge**$_+$ 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 merge conditions are also given by linear relations). A description of a function $\mathbf{Merge}_{+,\mathfrak{M}}$ can be found in Algorithm 2, with the help of Algorithm 4, and **Merge**$_+$ is the application of $\mathbf{Merge}_{+,\mathfrak{M}}$ to every monomial in the abstract state considered. The soundness of **Merge**$_+$ comes from the soundness of the underlying numerical domain and the tests performed by FIND_MERGE. Figure 10 gives a visual representation of the behavior of **Merge**$_+$.

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:*

$$\forall C \in \mathcal{C},\ \forall S,\ \forall S^{\sharp},\ S \subseteq \gamma(S^{\sharp}) \Rightarrow \mathbf{Post}[\![C]\!](S) \subseteq \gamma(\mathbf{Post}^{\sharp}[\![C]\!](S^{\sharp}))$$

The proof can be found in "Appendix B". The idea of the proof is to show that the proposed functions $\mathbf{Assign}^{\sharp}_{+,\mathfrak{M}}$, $\mathbf{Merge}_{+,\mathfrak{M}}$ 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 4 that implements a matrix multiplication. It employs two kinds of assignments $E_1 = A[X_1][X_2] \leftarrow c \in \mathbb{R}$ and $E_2 = A[X_1][X_2] \leftarrow A[X_1][X_2]+$

Algorithm 2: $\mathbf{Merge}_{+,\mathfrak{M}}$, merges possible predicates of a monomial

Input : $(\mathfrak{P}, \mathfrak{a})$
Output: $(\mathfrak{P}', \mathfrak{a}')$ merged state
$(\mathfrak{P}_o, \mathfrak{a}_o) \leftarrow (\mathfrak{P}, \mathfrak{a})$;
while $\exists (P_0, P_1) \in \mathfrak{P}_o$, FIND_MERGE$(P_0, P_1, \mathfrak{a}_o) \neq None$ **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 extensions 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 $A = A' \wedge B = B' \wedge C = C'$ **then**
  if $\mathfrak{a} \sqsubseteq_{\mathcal{B},\mathcal{V}} ((z = i) \wedge (y = j) \wedge (t = j + 1))$ **then**
    return *Some(Right)*
  else if $\mathfrak{a} \sqsubseteq_{\mathcal{B},\mathcal{V}} ((x = i) \wedge (z = i + 1) \wedge (t = j))$ **then**
    return *Some(Down)*
  else if $\mathfrak{a} \sqsubseteq_{\mathcal{B},\mathcal{V}} ((x = i + 1) \wedge (y = j) \wedge (t = j + 1))$ **then**
    return *Some(Left)*
  else if $\mathfrak{a} \sqsubseteq_{\mathcal{B},\mathcal{V}} ((x = i) \wedge (z = j) \wedge (y = j + 1))$ **then**
    return *Some(Up)*
  else
    return *None*
else
  return *None*
end

Algorithm 4: FIND_MERGE, finds possible merges 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 $A'' = A' \wedge B'' = B' \wedge C'' = C'$ **then**
  if $\mathfrak{a} \sqsubseteq_{\mathcal{B},\mathcal{V}} ((x_0 = x_1) \wedge (z_0 = z_1) \wedge (t_0 = y_1))$ **then**
    return *Some(Right)*
  else
    if $\mathfrak{a} \sqsubseteq_{\mathcal{B},\mathcal{V}} ((y_0 = y_1) \wedge (t_0 = t_1) \wedge (z_0 = x_1))$ **then**
      return *Some(Down)*
    else
      return *None*
    end
  end
else
  None
end

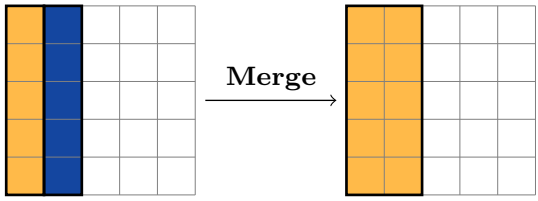

Fig. 10 Illustration of the behavior of the **Merge** function. Yellow and blue rectangles represent predicates. (Color figure online)

$B[X_1][X_3] \times C[X_3][X_2]$, 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

```
1   /*n >= 1; */
2   i := 0;
3   while (i < n) do
4    j := 0;
5    while (j < n) do
6     A[i][j] <- 0;
7     j := j +1
8    done;
9    i := i +1
10  done;
11  i := 0;
12  while (i < n) do
13   j := 0;
14   while (j < n) do
15     k := 0;
16     while (k < n) do
17      A[i][j]<-A[i][j]+
18      B[i][k]*C[k][j];
19      k := k + 1;
20     done
21     j := j + 1;
22   done
23   i := i + 1;
24  done
```

Program 4 Multiplication with inner loop on k

precisely, in our case, we consider the following two predicates:

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

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

which state respectively that the matrix A has been initialized 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 on 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 initialized to 0 as a prerequisite to receiving the product of two matrices, by summation over k of the partial products $B[i][k]C[k][j]$. How to handle these different cases is explained in details in the following.

4.2.1 Removing predicates

The analysis suggested for the addition in terms of postconditions can be extended to the initialization predicate the same way. Indeed, we add a $\mathbf{Assign}^\sharp_{s,\mathfrak{M}}$ abstract operator that enlarges the predicates $P_s(A, x, y, z, t, c)$ and a function $\mathbf{Merge}_{s,\mathfrak{M}}$, that merges them when possible (it is done the same way as for the addition predicate). However, for the analyzer to be sound, we need to modify the $\mathbf{Assign}^\sharp_{\times,\mathfrak{M}}$ and $\mathbf{Assign}^\sharp_{s,\mathfrak{M}}$ functions to 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. $\mathbf{Assign}^\sharp_{\times,\mathfrak{M}}$ and $\mathbf{Assign}^\sharp_{s,\mathfrak{M}}$ can both be found in "Appendix A", alongside some helper functions.

4.2.2 Splitting predicates

In Program 4, matrix A is initialized 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 $A[i][j] \leftarrow A[i][j] + B[i][k] \times C[k][j]$ 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

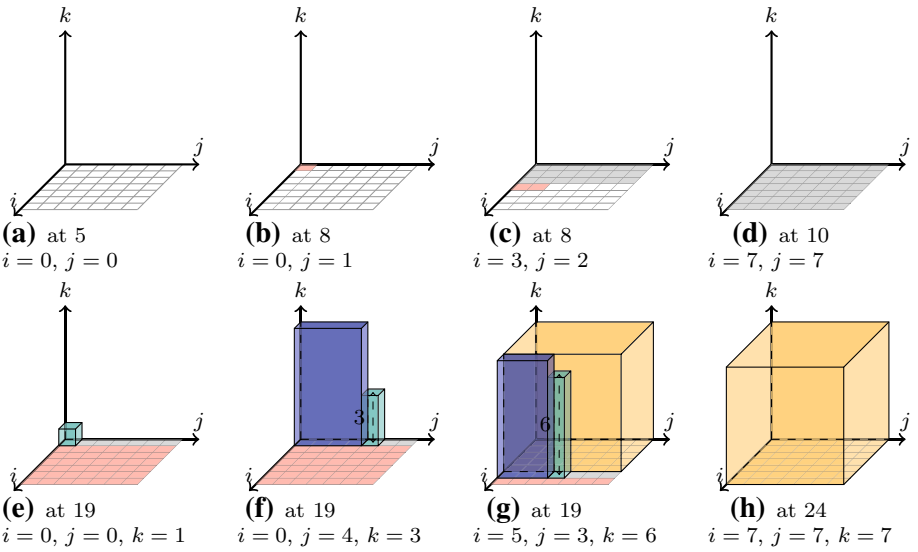

Fig. 11 Evolution of the predicates for Program 4, the predicates are the following: $P_\times(A, B, C, \dots)$, $P_\times(A, B, C, \dots)$, $P_\times(A, B, C, \dots)$, $P_s(A, B, C, \dots)$, $P_s(A, B, C, \dots)$

a multiplication predicate. If it exists, then we can, but we have to punch a hole in the zero predicates, to keep the information that, around the modified matrix element, the rest of the matrix stays initialized to 0 while the modified element is no longer guaranteed do be 0. Punching a hole is implemented by splitting the predicates, as we are only able to express matrix properties over rectangular areas. In order to illustrate this case, Fig. 11 depicts the evolution of the different predicates during the analysis of Program 4. 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 in according to the abstract state, Fig. 11 only depicts the most advanced one, i.e., the one that is the most precise about the contents of the matrix). An abstract state is the superposition of all possible states, hence not only those shown in Fig. 11.

4.2.3 Iterating on k first

Let us consider Program 5, this program also computes the matrix product of B and C and stores the result in A. However this version of the matrix product does not iterate on variables in the same order as Program 4. Indeed it iterates first on the k variable. Moreover, we notice that the variables occurring 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 and u, v variables depict the evolution of the multiplication. Therefore if we analyze Program 5 with the $P_\times$ predicate, the first analysis of the body of the loop (at line 15) will yield an abstract state of the form:[2] $P_\times(A, B, C, 0, 0, n, n, 1)$ where $n \times n$ is the size of the matrix. During the next analysis of the body of the loop we would like the analyzer to produce a second predicate so that the abstract state is of the form: $S_1^\sharp = (P_\times(A, B, C, 0, 0, n, n, 0, 1) \wedge P_\times(A, B, C, 0, 0, n, n, 1, 2)) \cup \dots$,

[2] For clarity reasons, we use here predicates with numerical value as variable instantiation instead of the conjunction of a predicate and a numerical domain.

```
1  /*n >= 1;*/
2  Array A of n n;
3  Array B of n n;
4  Array C of n n;
5  i := 0;
6  while (i < n) do
7   j := 0;
8   while (j < n) do
9    A[i][j] <- 0;
10   j := j +1
11  done;
12  i := i +1
13 done;
14 k := 0;
15 while (k < n) do
16  i := 0;
17  while (i < n) do
18   j := 0;
19   while (j < n) do
20    A[i][j]<-
          A[i][j]+B[i][k]*C[k][j];
21    j := j + 1;
22   done
23   i := i + 1;
24  done
25  k := k + 1;
26 done
```

Program 5 Multiplication with outer-loop on k

giving us after a merge a predicate of the form: $P_\times(A, B, C, 0, 0, n, n, 0, 2)$. However $S_1^\sharp$ contains a contradiction, because it is stating two different things about the cells in $[0, n[^2$, therefore we can not add a $P_\times$ predicate describing a cell already described by another $P_\times$ predicate. In order to be able to analyze matrix multiplicating programs where loops on k and i have been interchanged we have to introduce new predicates including existential quantification over matrices that were able to successfully analyze both Programs 5 and 4.

$$
\begin{aligned}
&P_{l,\times}(A, B, C, \{(x_i, y_i, z_i, t_i, u_i, v_i)\}_{i\in[0,l]}) \\
&\quad = \exists m_0, \ldots, \exists m_l, \bigwedge_{i\in[0,l]} p_\times(m_i, B, C, x_i, y_i, z_i, t_i, u_i, v_i) \\
&\qquad \wedge \forall a, b \in \bigcup_{i=0}^{l} [x_i, z_i - 1] \times \bigcup_{i=0}^{l} [y_i, t_i - 1],\ A[a][b] = m_0[a][b] + \cdots + m_l[a][b]
\end{aligned}
$$

where $p_\times(m_i, B, C, x_i, y_i, z_i, t_i, u_i, v_i)$

$$
\begin{aligned}
&= \forall a, b \in [x_i, z_i - 1] \times [y_i, t_i - 1],\ m_i[a][b] = \sum_{c=u_i}^{v_i-1} B[a][c]C[c][b] \\
&\quad \wedge \forall a, b \notin [x_i, z_i - 1] \times [y_i, t_i - 1],\ m_i[a][b] = 0
\end{aligned}
$$

Using this new family of predicates, we are able to express disjunctions on the k direction the same way we were able for i and j. A predicate $P_{l,\times}(A, B, C, \{(x_i, y_i, z_i, t_i, u_i, v_i)\}_{i\in\{0,l\}})$ can be seen as a conjunction of $l + 1$ predicates stating that the multiplication was effectively computed on $l + 1$ zones, however we need a relation combining all those predicates into one to state our goal. In this sense for Program 5 we only allow the analyzer to use some initialization predicates and one predicate at a time among $P_{0,\times}, P_{1,\times}, \ldots, P_{M_{pred},\times}$.

4.3 Trace partitioning

In this subsection we tackle some source of imprecisions in our analyzer. As an abstract state in our analyzer is a symbolic disjunction of some monomials, changing the maximum number of monomials allowed enables us to change the precision of the analyzer (more monomials make for a more precise analyzer). Figure 12 is a tiled version of the addition. In this version of the addition, the program iterates on blocks of size 32 by 32, however in order to deal with the case where n is not a multiple of 32, we need the last iterations to be on rectangles of size

```
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    endi := i < a ? 32 : b
13    while (ii < endi) do
14     jj := 0;
15     endj := j < a ? 32 : b
16     while (jj < endj) do
17      x = 32 * i + ii;
18      y = 32 * j + jj;
19      A[x][y] <- B[x][y] + C[x][y];
20      jj := jj + 1;
21     done;
22     ii := ii + 1;
23    done;
24    j := j + 1;
25   done;
26   i := i + 1;
27  done
```

Fig. 12 Addition tiled with reminder using if conditions

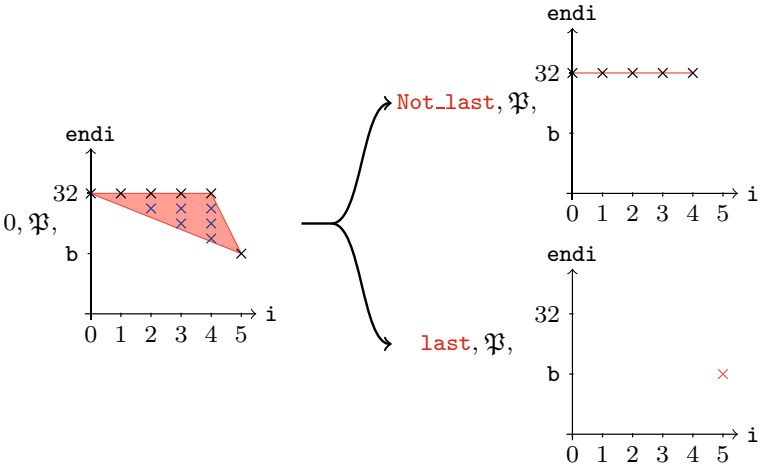

Fig. 13 Trace partitioning: black dots are concrete values, red zones are the numerical abstraction of the set of black dots, blue dots are spurious values appearing in the concretization of the red zone. (Color figure online)

$32 \times \mathtt{c}$,$\mathtt{b} \times 32$ and finally $\mathtt{b} \times \mathtt{c}$ where $\mathtt{b}$ (resp. $\mathtt{c}$) is the reminder of $\mathtt{A.n}$ (resp $\mathtt{A.m}$) modulo 32 and is therefore symbolic. This explains the tests in lines 12 and 15 of Program 12.

Most common numerical domains (and the one we chose, which is the polyhedron domain) store linear inequalities between variables. Figure 13 underlines the fact that there are no linear relations describing precisely all the concrete states, meaning that to ensure soundness the polyhedron domain over approximates the set of possible states, leading to an imprecision preventing our analyzer from concluding that the addition was indeed performed. By keeping a record of the history of the monomials we are able to increase precision by relying on a symbolic disjunction for monomials with different history alongside the program, as shown by Fig. 13. Our analyzer relies on a symbolic disjunction of monomials, this disjunction depends on the notion of well-formedness and thus of shape of a monomial; therefore by changing the definition of shape (in this new definition there is still a finite number of possible shapes, therefore ensuring the termination of the analysis) to take into account the history of the monomial we ensure that monomials with different history will not be merged. Using

this trace partitioning [12], our analyzer was able to prove the functional correctness of Program 12 (where the value of `a` and `b` are parameters of the analysis[3]).

In this subsection we used the example of Fig. 12 to underline why we had to use trace partitioning to improve precision (here the imprecision was coming from a `if` branching, therefore we partitioned monomials coming from different branches). However in order to be able to successfully analyze all the programs presented in this article, we also had to add trace partitioning on `while` loops (meaning that the analyzer keeps apart monomials that underwent at least one analysis of the body a loop, during the computation of the over-approximation of the fixpoint of the `while` loop, from those which did not).

4.4 Example of application

Now that we have given an in-depth presentation of our analyzer, let us give some examples of invariants automatically inferred on some programs we have presented.

- Program 3
  - at line 6:

$$\left\{ \left(\left\{ \begin{array}{l} P_+(C, A, B, \mathtt{a}_0, \mathtt{b}_0, \mathtt{c}_0, \mathtt{d}_0) \\ P_+(C, A, B, \mathtt{a}_1, \mathtt{b}_1, \mathtt{c}_1, \mathtt{d}_1) \end{array} \right\}, \left\{ \begin{array}{c} \mathtt{-a_1+i=0;\ -a_1+c_1-1=0;\ -a_1+c_0=0;} \\ \mathtt{-d_0+n=0;\ -d_1+j+1=0;\ b_1=0;\ b_0=0;} \\ \mathtt{a_0=0;\ -a_1+d_0-1>=0;\ d_1-1>=0;} \\ \mathtt{d_0-d_1>=0;\ d_0-5>=0;\ a_1-1>=0} \end{array} \right\} \right) \right.$$
$$\left. \left(\left\{ P_+(C, A, B, \mathtt{a}_2, \mathtt{b}_2, \mathtt{c}_2, \mathtt{d}_2) \right\}, \left\{ \begin{array}{c} \mathtt{-d_2+j+1=0;\ i=0;\ c_2-1=0;} \\ \mathtt{b_2=0;\ a_2=0;\ -d_2+n>=0;} \\ \mathtt{n-5>=0;\ d_2-1>=0} \end{array} \right\} \right) \right\}$$

  - at line 10:

$$\left\{ \left(\left\{ P_+(C, A, B, \mathtt{a}_0, \mathtt{b}_0, \mathtt{c}_0, \mathtt{d}_0) \right\}, \left\{ \begin{array}{c} \mathtt{-c_0+n=0;\ -c_0+j=0;\ -c_0+i=0;} \\ \mathtt{-c_0+d_0=0;\ b_0=0;\ a_0=0;} \\ \mathtt{c_0-5>=0} \end{array} \right\} (1) \right) \right\}$$

- Program 1
  - at line 5: (The abstract state contains 7 monomials, therefore for clarity reasons only one will be given here)

$$\left\{ \left(\left\{ \begin{array}{l} P_s(C, \mathtt{a}_0, \mathtt{b}_0, \mathtt{c}_0, \mathtt{d}_0, \mathtt{z}_0) \\ P_\times(C, A, B, \mathtt{a}_1, \mathtt{b}_1, \mathtt{c}_1, \mathtt{d}_1, \mathtt{e}_1, \mathtt{f}_1) \\ P_\times(C, A, B, \mathtt{a}_2, \mathtt{b}_2, \mathtt{c}_2, \mathtt{d}_2, \mathtt{e}_2, \mathtt{f}_2) \end{array} \right\}, \left\{ \begin{array}{c} \mathtt{z_0 = 0;\ -a_0+i=0;\ -a_0+c_2-1=0;} \\ \mathtt{-a_0+c_1=0;\ -a_0+c_0-1=0;\ -a_0+a_2=0;} \\ \mathtt{-b_0+j=0;\ -b_0+d_2=0;\ -b_0+d_0-1=0;} \\ \mathtt{-d_1+n=0;\ -d_1+f_2=0;\ -d_1+f_1=0;} \\ \mathtt{k=0;\ e_2=0;\ e_1=0;\ b_2=0;} \\ \mathtt{b_1=0;\ a_1=0;\ -a_0+d_1-1>=0;} \\ \mathtt{-b_0+d_1-1>=0;\ d_1-5>=0;\ b_0-1>=0;} \\ \mathtt{a_0-1>=0} \\ \ldots \end{array} \right\} \right) \right\}$$

  - at line 7:

$$\left\{ \left(\left\{ P_\times(C, A, B, \mathtt{a}_0, \mathtt{b}_0, \mathtt{c}_0, \mathtt{d}_0, \mathtt{e}_0, \mathtt{f}_0) \right\}, \left\{ \begin{array}{c} \mathtt{-c_0+n=0;\ -c_0+k=0;\ -c_0+j=0;} \\ \mathtt{-c_0+i=0;\ -c_0+f_0=0;\ -c_0+d_0=0;} \\ \mathtt{e_0=0;\ b_0=0;\ a_0=0;} \\ \mathtt{c_0-5>=0} \end{array} \right\} (2) \right) \right\}$$

Relations in the numerical abstract element of (1) (resp. (2)) and predicates prove that Program 3 (resp. Program 1) is functionally correct in the sense that at the end of the program C contains the addition (resp. the product) of A and B.

[3] With some conditions indicated in Program 12.

4.5 Adding further predicates

We presented in details three predicates so far (Addition, Set, Multiplication) but added more to our analyzer (Scale, Gemm). If we want to analyze programs not in the scope of the before-mentioned predicates (e.g.proving that a program performs a scale operation $A \leftarrow \alpha B$), we need to verify that loop invariants can be expressed using the predicate language we showed in Sect. 3.1. Some features (independent from the actual choice of predicate) are automatically provided (gestion of conjunctions, predicate disjunction, trace partitioning, numerical domains, widening, …); however as mentioned, the following work still needs to be performed when adding a new predicate:

1. Define the predicate (what are the free variables? what is its semantic?).
2. Define the way the predicate can be extended (the **Assign**$^\sharp$ function). When encountering a command where matrix names occur, our analyzer tries to pattern match the command to a set of known commands (e.g.$A[X][Y] \leftarrow B[X][Y]+C[X][Y]$,…). This step extends the set of patterns that can be matched by the analyzer.
3. Define the way predicates can be merged (the **Merge** function).

Example 4 Let us present what should be done for the scale operation:

1. Define the scale predicate $P_{sc}(A, B, x, y, z, t, \alpha) = \forall i, j \in [x, z[\times[y, t[,\ A[i][j] = \alpha B[i][j]$
2. A scale predicate $P_{sc}(A, B, x, y, z, t, \alpha)$ can be extended if the analyzer encounters a $A[i][j] \leftarrow \beta \times B[i][j]$ where $i, j, x, y, z, t, \alpha, \beta$ satisfy some relations that can be tested in the underlying numerical domain (e.g.: $\alpha = \beta$, …)
3. In the case of the scale predicate, predicates can be merged in the same way as the addition predicate.

5 Modular analysis of functions

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 by avoiding the constant reanalysis of the same functions when the calling context only slightly changes, which would be the case with the classic inlining approach to inter-procedural static analysis. 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 handling of scalar function arguments as this is standard).

5.1 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 another 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 $\mathbf{Post}(\gamma(S^\sharp)) \subseteq \gamma(S_o^\sharp)$ (see for instance

```
function add(A,B,C) {
 /* n >= 10 */;
 /* n = 32 * a*/
 i := 0;
 while (i <= a) do
  j := 0;
  while (j <= a) do
   add32(A,B,C,i,j);
   j := j + 1;
  done;
  i := i + 1;
 done
}
```

```
function add32(A,B,C,i,j) {
 ii := 0;
 while (ii < 32) do
  jj := 0;
  while (jj < 32) do
   x = 32 * i + ii;
   y = 32 * j + jj;
   A[x][y] <- B[x][y] + C[x][y];
   jj := jj + 1;
  done;
  ii := ii + 1;
 done;
}
```

Program 6 Memoisation of the two inner-most loops of a tiled addition

[13]). Therefore, when $S^\sharp \sqsubseteq S_i^\sharp$ holds, a possible postcondition is $S_o^\sharp$, which enables us to avoid reanalyzing the function. However when $S^\sharp \sqsubseteq S_i^\sharp$ does not hold, a new analysis of the callee needs to be performed, starting from a state $S_n^\sharp$ such that $S^\sharp \sqsubseteq S_n^\sharp$ holds. 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 (therefore increasing the reuse possibility of the stored relations), but small enough, so that the evaluation of the function produces an interesting state $S_o^\sharp$. In the implementation of our analyzer, we chose to relax the following constraints in $S_i^\sharp$:

- Conditions of the form $A.\mathtt{n} = \mathtt{v} \in \mathbb{N}^*$ 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 content of matrices are forgotten (from a monomial $(\mathfrak{P}, \mathfrak{a})$ we only keep $\mathfrak{a}$).

These choices can prevent us from analyzing precisely programs we could analyze using the non-modular analysis of Sect. 4 by inlining functions; however, they make 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.

Example 5 As a motivating example, consider Program 6. This program is a tiled version of the addition, where the two inner-most loops are placed in another function and replaced by a call to this function. 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. Being able to successfully perform a modular analysis of the function `add32` provides a way to speed up the analysis of function `add` by not having to reanalyze the two inner-most loops at each analysis of the `j` loop thanks to stored input/output relations on `add32`. We implemented this speed-up in our analyzer, on this example 4 input/ouput relations are stored for `add32` and those relations are used 21 times therefore $\frac{21}{21+4} = 84\%$ of the two inner-most loops analyses are table lookups. Section 7 provides experimental results on the speed up of the analysis using this method.

```
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 7 Addition with aliasing

5.2 The aliasing problem

In Sect. 4 we presented an analysis that was able to prove that some matrix-manipulating programs were performing additions. However we worked in a context where matrices with different names were different matrices. Consider now Program 7, as matrices are passed by reference to the function `add`, we can not perform a precise analysis of the body of the function without considering every possible way matrices can be related (e.g.whether A and B point to the same matrix or not, this equivalence relation on formal arguments, indicating whether they refer to the same concrete matrix or not is called an aliasing). However some of those analyses might not be useful for the analysis of the whole program, therefore we will only perform analysis when needed (i.e. when we encounter a new kind of aliasing). This analysis is performed on formal matrices, verifying some aliasing conditions. For every different aliasing the function is called with the analyzer keeps a record of the input/output relation. Therefore when the analyzer encounters another function call with the same aliasing, it can soundly reuse the previous analysis. Note that there is only a finite number of possible aliasings.

5.3 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 (by punching in a hole, the same way we removed the set predicate for the analysis of matrix multiplication programs in Sect. 4.2). This method gives us:

– Input/Output relations between matrices (expressed as symbolic or predicate relations between matrices and their auxiliary versions).
– Which 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 6 Let us consider Program 7. The function `add` is analyzed twice, because of the two different aliasing relationships. A first analysis of `add` is performed on line 13 (with aliasing `G,H,I` independent from each other), 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 (with aliasing `G,H,I` where `G = I` and `H` independent from the others) 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`. At the end of the analysis the store contains the following relations:

$$\left(\texttt{add}(D, E, F), (\emptyset, \mathfrak{a}), \left(\left\{\begin{array}{c} P_+(D', E, F, x, y, z, t) \\ Eq(E', E, x', y', z', t') \\ Eq(F, F', x'', y'', z'', t'') \end{array}\right\}, \mathfrak{a}'\right)\right)$$

and

$$\left(\texttt{add}(A, B, A), (\emptyset, \mathfrak{a}), \left(\left\{\begin{array}{c} P_+(A', B, A, x, y, z, t) \\ Eq(B', B, x', y', z', t') \end{array}\right\}, \mathfrak{a}'\right)\right)$$

6 Matrix inversion program

6.1 Iterator abstraction

In this section, we discuss the analysis of more complex matrix-manipulating programs. These programs are higher-level, and use the matrix additions and multiplications seen in the previous sections as a base operation they use multiple times to perform complex computations. On the one hand, these programs will benefit from the technique from Sect. 4 to prove, in a modular way, that matrix additions and multiplications are indeed correct. On the other hand, we will require novel predicates to prove that the overall algorithm obeys its specification. We consider in particular iterative algorithms, such as iterative matrix inversion. Let us consider as an example Program 8, which is computing the sequence: $v_{n+1} = 2v_n - v_n A v_n$. If A is an invertible matrix and if v_0 is well-chosen then the sequence v_n converges toward A^{-1}. Therefore we want our analyzer to infer and prove that Program 8 is indeed computing the sequence (v_n). It is important to note that the analyzer will not infer that v_n is converging towards A^{-1}, however it will infer that v_n is computed by induction and follows $v_{n+1} = 2v_n - v_n A v_n$, which will help proving the functional correctness of this program.

Remark 4 The work of this section is focused around the iterative matrix inversion algorithm for presentation reasons, however iterative algorithms using only simple operation over matrices are fairly common and the techniques proposed in this section can be reused for other iterative processes. Let us mention as an example that an eigenvector associated to the biggest eigenvalue of a matrix A can be computed (under suitable conditions) using the sequence : $b_k = \frac{Ab_{k-1}}{\|Ab_{k-1}\|}$.

```
1  function main () {
2  /*n >= 10*/
3  /*n=m*/
4  /*en >= 100*/
5  matrix V of n of m;
6  matrix A of n of m;
7  matrix T1 of n of m;
8  matrix T2 of n of m;
9  matrix T3 of n of m;
10 matrix T4 of n of m;
11 i:=0;
12 while (i < en) do
13  multiplication(T1,A,V);
14  multiplication(T2,V,T1);
15  scale1(T3,T2);
16  addition(T4,V,V);
17  addition(T,T4,T3);
18  i := i + 1;
19 done
20 }
```

Program 8 Main function of the inversion program

$$\mathcal{U} \triangleq \mathcal{U} + \mathcal{U} \mid \mathcal{U}\mathcal{U} \mid x.\mathcal{U} \mid 0 \mid A \in \mathbb{A}$$

$$\begin{aligned}
&\mathcal{I}_{\mathcal{U}}[\![U]\!](m) = \text{ match } U \text{ with:}\\
&\quad |U_1 + U_2 \rightarrow \quad \{f \mid \forall (i,j) \in [0, m_{\mathcal{V}}(U_1.\mathtt{n})[\times[0, m_{\mathcal{V}}(U_1.\mathtt{m})[,\\
&\qquad\qquad f(i,j) = \mathcal{I}_{\mathcal{U}}[\![U_1]\!](m)(i,j) + \mathcal{I}_{\mathcal{U}}[\![U_2]\!](m)(i,j)\}\\
&\quad |U_1 U_2 \rightarrow \quad \{f \mid \forall (i,j) \in [0, m_{\mathcal{V}}(U_1.\mathtt{n})[\times[0, m_{\mathcal{V}}(U_2.\mathtt{m})[,\\
&\qquad\qquad f(i,j) = \sum_{k=0}^{m_{\mathcal{V}}(U_1.\mathtt{m})} \mathcal{I}_{\mathcal{U}}[\![U_1]\!](m)(i,k)\mathcal{I}_{\mathcal{U}}[\![U_2]\!](m)(k,j)\}\\
&\quad |x.U_1 \rightarrow \quad \{f \mid \forall (i,j) \in [0, m_{\mathcal{V}}(U_1.\mathtt{n})[\times[0, m_{\mathcal{V}}(U_1.\mathtt{m})[,\\
&\qquad\qquad f(i,j) = m_{\mathcal{V}}(x)\mathcal{I}_{\mathcal{U}}[\![U_1]\!](m)(i,j)\}\\
&\quad |0 \rightarrow \quad \{f \mid \forall (i,j) \in [0, m_{\mathcal{V}}(U_1.\mathtt{n})[\times[0, m_{\mathcal{V}}(U_1.\mathtt{m})[,\\
&\qquad\qquad f(i,j) = 0\}\\
&\quad |A \rightarrow \quad \{f \mid \forall (i,j) \in [0, m_{\mathcal{V}}(U_1.\mathtt{n})[\times[0, m_{\mathcal{V}}(U_1.\mathtt{m})[,\\
&\qquad\qquad f(i,j) = m_{\mathcal{A}}(A)(i,j)\}
\end{aligned}$$

$$\mathcal{I}_{\mathfrak{u}}[\![\mathfrak{u}]\!] = \{(m_{\mathcal{V}}, m_{\mathcal{A}}) \mid \bigwedge_{A \in \mathbf{def}(\mathfrak{u})} m_{\mathcal{A}}(A) \in \mathcal{I}_{\mathcal{U}}[\![\mathfrak{u}(A)]\!](m)\}$$

Fig. 14 Terms over matrices and stores

Predicates over the matrix ring As these algorithms are higher-level, we no longer need to express arithmetic predicates over real-valued scalar variables and individual matrix elements, but rather predicates over the matrix ring. We thus introduce a new predicate language and its interpretation, we also define a *store* $\mathfrak{u}$ as a partial function from $\mathbb{A}$ to $\mathcal{U}$ with finite set of definitions denoted $\mathbf{def}(\mathfrak{u})$. All those definitions can be found in figure 14. The subset of $\mathbb{A}$ of matrix names appearing in a term $t \in \mathcal{U}$ is denoted by $\mathbf{mat}(t)$.

Example 7 $\mathfrak{u} = (A \mapsto BC) \circ (C \mapsto xB)$ is a store, whose interpretation is the set of states where matrix A is the matrix product of B and C and matrix C is the scalar product of x and B.

Example 8 At the end of the previous section (Sect. 5) we considered the analysis of Program 7. Using the symbolic store defined here, the analyzer concludes that `E,F,B,C` were not modified, that `A=B+B+C`, and `D=E+F`.

In order to prove that Program 8 is indeed computing the sequence v_n we introduce a new abstract domain. We call an *iterator item* an element $j = (V, f_V, V_i) \in (\mathbb{A} \times \mathcal{U} \times \mathbb{A}) = \mathcal{J}$ (e.g.(A, AB, A_0) is an iterator item). An *iterator* is an element $\mathfrak{i} = (\mathfrak{j}, i) \in (\wp(\mathcal{J}) \times \mathbb{Y}) = \mathcal{D}$,

$$\begin{aligned}\mathcal{I}_{\mathfrak{I}}[\![(\mathfrak{j},l)]\!] = \{m \mid & \exists IV_0^{(0)},\ldots,IV_{n-1}^{(0)},\ldots,IV_0^{(i)},\ldots,IV_{n-1}^{(i)}, \forall k \in [1,i[, \forall j \in [0,n-1],\\ & IV_j^{(k)} \in \mathcal{I}_{\mathcal{U}}[\![f_{V_j}[V_0 \leftarrow V_0^{(k-1)},\ldots,V_{n-1} \leftarrow V_{n-1}^{(k-1)}]]\!](m_e)\} \wedge\\ & IV_0^{(0)} = m_{\mathcal{A}}(V_0^{(0)}) \wedge \cdots \wedge IV_{n-1}^{(0)} = m_{\mathcal{A}}(V_{n-1}^{(0)}) \wedge\\ & IV_0^{(i)} = m_{\mathcal{A}}(V_0^{(i)}) \wedge \cdots \wedge IV_{n-1}^{(i)} = m_{\mathcal{A}}(V_{n-1}^{(i)}) \wedge\\ & i = m_{\mathcal{V}}(l)\}\end{aligned}$$

where:

$$\begin{aligned}\mathfrak{j} &= \{(V_0, f_{V_0}, V_0^{(0)}),\ldots,(V_{n-1}, f_{V_{n-1}}, V_{n-1}^{(0)})\}\\ m_e &= (m_{\mathcal{V}}, m_{\mathcal{A}}[V_0^{(0)} := IV_0^{(0)},\ldots,V_{n-1}^{(0)} := IV_{n-1}^{(0)},\ldots,V_{n-1}^{(i)} := IV_{n-1}^{(i)}])\end{aligned}$$

$$\gamma_i(S^\sharp) = \bigcup_{(\mathfrak{I},\mathfrak{a})\in S^\sharp} \{(m_{\mathcal{V}|\mathbb{X}\cup\mathbb{S}}, m_{\mathcal{A}}) \mid m_{\mathcal{V}} \in \gamma_{\mathcal{V}}(\mathfrak{a}) \qquad \mathfrak{i} \in \mathfrak{I},\ (m_{\mathcal{V}}, m_{\mathcal{A}}) \in \mathcal{I}_{\mathfrak{I}}[\![(\mathfrak{i})]\!]\}$$

Fig. 15 Interpretation of an iterator and concretisation function

that is, a set of iterator items (understood conjunctively) and an element of $\mathbb{Y}$ (the set of numerical variables not appearing in the program). It is said to be *closed* if $\forall(V, f_V, V_i) \in \mathfrak{j}, \forall A \in \mathbf{Mat}(f_V), \exists(U, f_U, U_i) \in \mathfrak{j},\ A = U$. As in Sect. 3.2 the numerical variable $i \in \mathbb{Y}$ appearing in iterators is bound by a numerical abstract domain. The set of definition of an iterator is: $\mathbf{Mat}((\mathfrak{j}, i)) = \{V \in \mathbb{A} \mid \exists(U, f_U, U_i) \in \mathfrak{j},\ V = U\}$, in the following if $V \in \mathbf{Mat}((\mathfrak{j}, i))$ we note f_V and V_i to denote the elements such that $(V, f_V, V_i) \in \mathfrak{j}$.

Example 9
- $(\{(A, AB, A_0)\}, i)$ is an iterator, however it is not closed, indeed $B \in \mathbf{Mat}(AB)$.
- $(\{(A, AB, A_0), (B, B, B_0)\}, i)$ is a closed iterator, whose set of definitions is $\{A, B\}$.

Figure 15 gives the formal interpretation of an iterator, the idea is the following: the interpretation of a iterator $(\mathfrak{j}, i) \in (\wp(\mathcal{J}) \times \mathbb{Y})$ is the set of memory states where the matrices V appearing in $\mathbf{Mat}((\mathfrak{j}, i))$ are obtained from iterating $\{f_U \mid U \in \mathbf{Mat}((\mathfrak{j}, i))\}$ ith time, starting on the set $\{U_i \mid U \in \mathbf{Mat}((\mathfrak{j}, i))\}$.

As previously, our predicates (here the iterators) have free variables, that will be bound in a numerical abstract domain. Therefore as in Sect. 3.2 we define a notion of monomial, shape and well-formedness, a monomial is said to be well-formed when it contains at most one iterator, thus ensuring that we have a finite number of possible shapes in our abstract state. This enables us to define a join as the application of some **wf** operator (as in Definition 5 in Sect. 3.2) on the union of the monomials of the two abstract states, a widening as the application of the widening in the underlying numerical domain. Figure 15 gives the concretisation function for such an abstract state.

Example 10 The concretisation of $S_1^\sharp = \{(\{\{(A, AB, B), (B, B, B)\}, i\}, \{i = 9\})\}$ is such that $A = B^{10}$, indeed $f_A = AB$ and $A_0 = B$ moreover $f_B = B$ and $B_0 = B$ therefore $A_1 = A_0B_0 = B^2$ and $B_1 = B_0 = B$ …. Moreover if $S_2^\sharp = \{(\{\{(A, AB, B), (B, B, B)\}, i\}, \{i = 10\})\}$ then $S_1^\sharp \sqcup S_2^\sharp = \{(\{\{(A, AB, B), (B, B, B)\}, i\}, \{i = 9 \vee i = 10\})\}$.

From now on we will suppose that monomials are of the form $(\mathfrak{u}, \mathfrak{I}, \mathfrak{P}, \mathfrak{a})$ where:

- $\mathfrak{u}$ is a symbolic store over matrix variables
- $\mathfrak{I}$ is an iterator

- $\mathfrak{P}$ is a set of predicates
- $\mathfrak{a}$ is the underlying numerical domain

and enable some rewriting from a monomial into another as shown by the following example:

Example 11 A monomial $(\emptyset, \emptyset, \{P_+(A, B, C, x, y, z, t)\}, \mathfrak{a})$, where $\mathfrak{a} \sqsubseteq_{\mathcal{B},\mathcal{V}} \{x = y = 0 \wedge y = A.\mathtt{n} \wedge z = A.\mathtt{m}\}$ is turned into $(\{A \mapsto B + C\}, \emptyset, \emptyset, \mathfrak{a}))$.

Analyzer We want our analyzer to infer and prove that Program 8 is computing the sequence $v_{n+1} = 2v_n - v_n A v_n$. In order to do so we want the iterator of the main loop to be of the form $(\ldots, (\{(V, V + V + xVAV, V_0), (A, A, A)\}, i), \ldots, \{x = -1 \wedge 0 \leq i \leq en - 1\})$, where V_0 denotes the initial value of V. However V is a matrix that is modified by the content of the loop, therefore in order to have an iterator that holds along the analysis of the body of the loop, the analyzer describes an iterator on a copy of matrices V and A (resp. V_{beg} and A_{beg}) and uses the matrix store to relate all matrices to those two. We note that at the end of the body of the loop the matrix store will therefore contain: $\mathtt{T1} \mapsto \mathtt{A\_beg.V\_beg}$, $\mathtt{T2} \mapsto \mathtt{V\_beg.A\_beg.V\_beg}$, $\mathtt{T3} \mapsto x\mathtt{V\_beg.A\_beg.V\_beg}$, $\mathtt{T4} \mapsto y\mathtt{V\_beg}, \mathtt{V} \mapsto y\mathtt{V\_beg} + x\mathtt{V\_beg.A\_beg.V\_beg}$, Therefore at the end of the first analysis of the loop, by looking up the contents of the store, we can infer $(\{(V_{beg}, V_{beg} + V_{beg} + xV_{beg}A_{beg}V_{beg}, V_0), (A_{beg}, A_{beg}, A_0)\}, 1)$. In the following iterations of the analysis of the loop, a widening is performed in the underlying numerical abstract domain, allowing our analyzer to infer the iterator shown in Sect. 6.2.

6.2 Full example

In this subsection we will not give the entire analysis of the entire Program 9, however we will present how the different abstractions presented in this article all come together in this analysis. $\mathtt{i}$ designates the abstracted states associated with control point i of the program and $\mathtt{i}$-i_1-$\cdots$-i_n designates the partial abstracted state constructed by the analyzer at control point i after i_1 iterations of the outer-most while loop, i_2 iterations of the second outer-most while loop up until i_n iterations of the inner-most loop.

$\mathtt{44}$ $\{\mathfrak{u} = \emptyset, \mathfrak{J} = \emptyset, \mathfrak{P} = \emptyset, \mathfrak{a} = \top\}$ Initial abstract states

$\mathtt{54}$ $\{\mathfrak{u} = \emptyset, \mathfrak{J} = \emptyset, \mathfrak{P} = \emptyset, \mathfrak{a} = (\mathfrak{a}_0 \triangleq \{\mathtt{V.n} = \mathtt{A.n} = \mathtt{T1.n} = \mathtt{T2.n} = \mathtt{T3.n} = \mathtt{T4.n} = \mathtt{n} = \mathtt{m} = \mathtt{V.m} = \mathtt{A.m} = \mathtt{T1.m} = \mathtt{T2.m} = \mathtt{T3.m} = \mathtt{T4.m} \geq 10 \wedge \mathtt{en} \geq 100\}) \wedge \mathtt{i} = 0\}$ Initialization

$\mathtt{58\text{-}0}$ $\{\mathfrak{u} = \{\mathtt{T3} \mapsto x_1\mathtt{V}_i\mathtt{A}_i\mathtt{V}_i, \mathtt{T2} \mapsto \mathtt{V}_i\mathtt{A}_i\mathtt{V}_i, \mathtt{T1} \mapsto \mathtt{A}_i\mathtt{V}_i, \mathtt{V} \mapsto \mathtt{V}_i, \mathtt{A} \mapsto \mathtt{A}_i\}, \mathfrak{J} = \emptyset, \mathfrak{P} = \emptyset, \mathfrak{a} = \mathfrak{a}_0 \wedge \mathtt{i} = 0 \wedge x_1 = -1\}$

> $\mathtt{33}$ $\{\mathfrak{u} = \emptyset, \mathfrak{J} = \emptyset, \mathfrak{P} = \{Eq(\mathtt{V}, \mathtt{V}', x_1, y_1, z_1, t_1), Eq(\mathtt{T4}, \mathtt{T4}', x_0, y_0, z_0, t_0)\}, \mathfrak{a} = (\mathfrak{a}_1 \triangleq \mathtt{T4.n} = \mathtt{T4}'\mathtt{.n} = \mathtt{V.n} = \mathtt{V}'\mathtt{.n} = \mathtt{T4.m} = \mathtt{T4}'\mathtt{.m} = \mathtt{V.m} = \mathtt{V}'\mathtt{.m} = z_1 = t_1 \geq 1 \wedge x_1 = y_1 = 0) \wedge x_0 = y_0 = 0 \wedge z_0 = y_0 = \mathtt{A.n}\}$ Entering addition call
>
> $\mathtt{38\text{-}2\text{-}2}$ $\{\mathfrak{u} = \emptyset, \mathfrak{J} = \emptyset, \mathfrak{P} = \{Eq(\mathtt{V}, \mathtt{V}', x_1, y_1, z_1, t_1), Eq(\mathtt{T4}, \mathtt{T4}', x_0, y_0, z_0, t_0), Eq(\mathtt{T4}, \mathtt{T4}', x_2, y_2, z_2, t_2), P_+(\mathtt{T4}', \mathtt{V}, \mathtt{V}, x_3, y_3, z_3, t_3)\}, \mathfrak{a} = \mathfrak{a}_1 \wedge \ldots, \mathfrak{u} = \emptyset, \mathfrak{J} = \emptyset, \mathfrak{P} = \{Eq(\mathtt{V}, \mathtt{V}', x_1, y_1, z_1, t_1), Eq(\mathtt{T4}, \mathtt{T4}', x_0, y_0, z_0, t_0), Eq(\mathtt{T4}, \mathtt{T4}', x_2, y_2, z_2, t_2), P_+(\mathtt{T4}', \mathtt{V}, \mathtt{V}, x_3, y_3, z_3, t_3), P_+(\mathtt{T4}', \mathtt{V}, \mathtt{V}, x_4, y_4, z_4, t_4)\}, \mathfrak{a} = \mathfrak{a}_1 \wedge \ldots, \ldots\}$ Analysis of addition program (see Sect. 4)
>
> > $\mathtt{42}$ $\{\mathfrak{u} = \emptyset, \mathfrak{J} = \emptyset, \mathfrak{P} = \{Eq(\mathtt{V}, \mathtt{V}', x_1, y_1, z_1, t_1), P_+(\mathtt{T4}', \mathtt{V}, \mathtt{V}, x_1, y_1, z_1, t_1)\}\}, \mathfrak{a} = \mathfrak{a}_1 \wedge x_0 = y_0 = 0 \wedge z_0 = y_0 = \mathtt{A.n}\}$ End of analysis of addition, input/output relations is added to a store (see Sect. 5)

```
1  function scale1 (C,A) {
2   cc := -1;
3   i := 0;
4   while (i <= (A.n - 1)) do
5    j := 0;
6    while (j <= (A.m - 1)) do
7     C[i][j] <- cc * A[i][j];
8     j := j + 1
9    done;
10   i := i + 1
11  done;
12 }
13 function multiplication
       (C,A,B) {
14  i := 0;
15  while (i <= (A.n - 1)) do
16   j := 0;
17   while (j <= (A.m - 1)) do
18    zz := 0;
19    C[i][j] <- zz;
20    k := 0;
21    while (k <= (A.m -1)) do
22     C[i][j] <- C[i][j] +
           A[i][k] * B[k][j];
23     k := k + 1
24    done;
25    j := j + 1
26   done;
27   i := i + 1
28  done
29 }
30
31
32
33 function addition (C,A,B) {
34  i := 0;
35  while (i <= (A.n - 1)) do
36   j := 0;
37   while (j <= (A.m - 1)) do
38    C[i][j] <-
          A[i][j]+B[i][j];
39    j := j + 1
40   done;
41   i := i + 1
42  done;
43 }
44 function main () {
45  /*n >= 10*/
46  /*n=m*/
47  /*en >= 100*/
48  matrix V of n of m;
49  matrix A of n of m;
50  matrix T1 of n of m;
51  matrix T2 of n of m;
52  matrix T3 of n of m;
53  matrix T4 of n of m;
54  i:=0;
55  while (i < en) do
56   multiplication(T1,A,V);
57   multiplication(T2,V,T1);
58   scale1(T3,T2);
59   addition(T4,V,V);
60   addition(T,T4,T3);
61   i := i + 1;
62  done
63 }
```

Program 9 Inversion program: complete version

59-0 $\{u = \{T3 \mapsto x_1 V_i A_i V_i, T2 \mapsto V_i A_i V_i, T1 \mapsto A_i V_i, V \mapsto V_i, A \mapsto A_i, T4 \mapsto V_i + V_i\}, \mathfrak{J} = \emptyset, \mathfrak{P} = \emptyset, \mathfrak{a} = \mathfrak{a}_0 \wedge i = 0 \wedge x_1 = -1\}$

55-1 $\{u = \{V \mapsto V_{beg}, A \mapsto A_{beg}\}, \mathfrak{J} = \{\{(V_{beg}, V_{beg} + V_{beg} - V_{beg} A_{beg} V_{beg}, V_i), (A_{beg}, A_{beg}, A_i)\}, x_2\}, \mathfrak{P} = \emptyset, \mathfrak{a} = \mathfrak{a}_0 \wedge i = x_2 = 1 \wedge x_1 = -1\}$ After finding an iterator (see Sect. 6.1)

55-2 $\{u = \{V \mapsto V_{beg}, A \mapsto A_{beg}\}, \mathfrak{J} = \{\{(V_{beg}, V_{beg} + V_{beg} - V_{beg} A_{beg} V_{beg}, V_i), (A_{beg}, A_{beg}, A_i)\}, x_2\}, \mathfrak{P} = \emptyset, \mathfrak{a} = \mathfrak{a}_0 \wedge 0 \leq i = x_2 < en \wedge x_1 = -1\}$ Proving the iterator on every loop and widening

62 $\{u = \{V \mapsto V_{beg}, A \mapsto A_{beg}\}, \mathfrak{J} = \{\{(V_{beg}, V_{beg} + V_{beg} - V_{beg} A_{beg} V_{beg}, V_i), (A_{beg}, A_{beg}, A_i)\}, x_2\}, \mathfrak{P} = \emptyset, \mathfrak{a} = \mathfrak{a}_0 \wedge i = x_2 = en \wedge x_1 = -1\}$ Final state

This example shows how to combine together the different abstractions suggested before to enable users to prove the soundness of a program containing multiple functions. We remind that each of the function of the program is analyzed in a modular way, therefore we do not have to reanalyze a function at each call site. The final state of the previous analysis underlines that the sequence V_n of matrices after the n-th loop iteration is indeed defined by the recurrence $V_{n+1} = 2V_n - V_n A V_n$, thus under good initial conditions[4] and if A is invertible a user is ensured that this program computes a sequence that is converging toward A^{-1}.

[4] Those conditions are independant from the analysis, our analyzer does not verify them and does not need them to infer the iterator of the program.

7 Implementation

Results We implemented the algorithms proposed above as well as various improvements to make the analysis more robust, notably symbolic equality domains [14] 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 (additions and multiplications). We also analyzed tiled versions of these algorithms, which is a classic optimization (performed by hand or automatically) that increases cache efficiency, but makes algorithms more difficult to understand. Program 12 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 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 analyzer 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 (which contains no pointer, no struct, no float, no dynamic allocation,…, it only manipulates matrices and real numbers) 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: $C \leftarrow \alpha AB + \beta C$, Program 11 in "Appendix C" gives the code of the program as found in a BLAS library, with minor modifications), with various optimizations and on every possible loop order. As examples, we mostly used programs optimized by PLUTO and BLAS (Basic Linear Algebra Subprograms) library programs we found in the `LAPACK` package manually rewritten in our analyzer input language. Those programs are all successfully analyzed by the final implementation. Table 1 gives the time it took the analyzer to prove the functional correctness of programs proposed in this article. The number of loops indicated is the biggest number of nested 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 requires 6 loops).

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 [6]. Note however that in simpler cases (e.g.when no tiling is involved) the octagon domain [15] provides enough precision for our analyzer to successfully conclude. As most of the operations performed by our analyzer are done on the numerical abstract domain, the cost of the analysis depends

Table 1 Analysis time in seconds

Set to 0	0.022		0.724	49.87
Addition	0.015		0.425	28.15 (1.72 using Sect.5)
Multiplication		0.233		56.59
Addition with rest			0.781	
Scale	0.020		0.866	62.71 (3.10 using Sect.5)
GEMM		0.854		

Table 2 Analyzer time in seconds with memoisation

Loop tabulated	∅	5	4	{2, 6}	{1, 5}	{1, 6}	{3, 6}
Time	28.15	1.72	2.00	2.00	2.68	2.73	3.33
Reusability	0%	97%	93%	92%	92%	97%	84%

mainly on the cost of the operations 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).

Loop tabulation In Sect. 5 we proposed a method to find postconditions for function calls that prevented re-computation of already analyzed code. We apply this same method for loop analysis, indeed we perform an analysis the first time the analyzer encounters a loop, starting from a relaxed version (to enable more reuse) of the abstract state holding when entering the loop. However as pointed earlier, we do not yet have any interesting method for inferring interesting preconditions on matrices (big enough to cover many cases and small enough to provide meaningful postconditions) therefore we tested this method only on programs computing additions as no precondition on matrices is needed. In our tests we relaxed some conditions in the underlying numerical domain on well-chosen variables (those appearing in loop conditions) so that the pre/postconditions computed by the analyzer could be reused for other entry values of the loop. Table 2 shows the analyzer computation time on an addition with 6 nested loops (a twice tiled addition), with memoisation on some loops, the first line indicates at which loops depths conditions were stored, the third line indicates the reusability factor (Number of analyses that amounts to a table lookup/Number of analyses). Table 2 only shows the computation time for the 6 fastest analyses, we notice that we are able to get a factor 15 speedup in the best case. It is to be noted that adding memoisation on a loop induces an increase in the evaluation time of the body of the loop due to the duplication of variables. Therefore memoisation can not be used on every loop otherwise the computation time would increase significantly. Empirically a good heuristic seems to be to apply memoisation on inner-most loops.

8 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 updates. As the static analysis of array properties has drawn some significant attention lately, more precise methods have been devised. Gopan et al. [16] extend this standard

method by allowing relational (but still uniform) abstractions. A class of analyses [5,17,18] dynamically partitions arrays, which allows expressing non-uniform properties of arrays as well as strong updates. Allamigeon has designed companion numerical domains specifically targeting array partition bounds [3]. String analysis for C programs can be seen as a special case of array analysis, and similar partitioning-based methods have been proposed [19]. 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. Fluid updates [20] 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. [21] 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 predicates abstractions [4], which also analyzes arrays using predicates of fixed shape conjoined with numeric properties. The large majority of these works only focus on properties of uni-dimensional 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 [5] but without details nor experiments) while the method of [21] can analyze matrix initialization (and possibly more) when parameterized with the correct predicate. Some other approaches to prove the functional correctness of matrix manipulating programs have been to automate the proof through the help of theorem provers such as ACL2 as proposed in [22]. The FLAME framework proposed in [23] enables the development of matrix manipulating programs, and the automatic generation, from invariants, of programs that are sound by constructions. Their work focuses on higher level algorithms than those presented here and the soundness proofs rely on the soundness of the underlying BLAS library, therefore our work complements theirs. Non relational properties of matrices (e.g.lower triangular, diagonal,…) could also be obtained via the use of loop transformation as proposed in [24]. 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 nor has there been experiments performed on optimized programs (e.g.PLUTO outputs).

9 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 numerical 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. 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 library implementations or scientific applications.

A Complementary Algorithms

A.1 Set operations

Algorithm 5: Assign$^{\sharp}_{S,\mathfrak{M}}$, computes the image by the transfer functions (associated to the expression $E = A[X_1][X_2] \leftarrow c$)

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

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

A.2 Product operations

Remark 5 the **sanitize** operator ensures soundness by punching holes in predicates that were holding in a cell where a write has been performed. For clarity reason we did not make it appear in the $\mathbf{Assign}^{\sharp}_{+,\mathfrak{M}}$ algorithm because its description was given under stronger hypothesis.

Algorithm 6: FIND_EXTENSION_SET, finds possible extensions of a monomial for the set operation

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

$P_+(A', x, y, z, t, c) = P$;
if $A = A' \wedge B = B' \wedge C = C'$ **then**
  if $\mathfrak{a} \sqsubseteq_{\mathcal{B},\mathcal{V}} ((z = i) \wedge (y = j) \wedge (t = j + 1) \wedge (k = c))$ **then**
    return *Some(Right)*
  else if $\mathfrak{a} \sqsubseteq_{\mathcal{B},\mathcal{V}} ((x = i) \wedge (z = i + 1) \wedge (t = j) \wedge (k = c))$ **then**
    return *Some(Down)*
  else if $\mathfrak{a} \sqsubseteq_{\mathcal{B},\mathcal{V}} ((x = i + 1) \wedge (y = j) \wedge (t = j + 1) \wedge (k = c))$ **then**
    return *Some(Left)*
  else if $\mathfrak{a} \sqsubseteq_{\mathcal{B},\mathcal{V}} ((x = i) \wedge (z = j) \wedge (y = j + 1) \wedge (k = c))$ **then**
    return *Some(Up)*
  else
    return *None*
else
  return *None*
end

Algorithm 7: $\mathbf{Assign}^{\sharp}_{\times,\mathfrak{M}}$, computes the image by the transfer functions (associated to the expression $E = A[X_1][X_2] \leftarrow A[X_1][X_3] + B[X_3][X_2] \times C[X_1][X_2]$)

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

if $\exists P_0 \in \mathfrak{P}$, FIND_EXTENSION_MULT$((P_0, \mathfrak{a}), A, B, C, i, j, k) = res \neq$ *None* **then**
  $x', y', z', t', u', v' \leftarrow$ **fresh**();
  $I \leftarrow$ **switch** *res* **do**
    case *Some(Above)*
      $(x = x' \wedge y = y' \wedge z = z' \wedge t = t' \wedge u = u' \wedge v + 1 = v')$
    case *Some(Below)*
      $(x = x' \wedge y = y' \wedge z = z' \wedge t = t' \wedge u - 1 = u' \wedge v = v')$
  endsw
  return $((\mathfrak{P} \setminus P_0) \cup \{P_\times(A, x', y', z', t', u', v')\}, \mathfrak{a} \sqcap_{\mathcal{B},\mathcal{V}} I)$
else
  if $\sharp\mathfrak{P} < M_{pred} \wedge \exists P_1 \in \mathfrak{P}$, FIND_ZERO_MULT$(P_1, a, A, i, j)$ **then**
    $x', y', z', t', u', v' \leftarrow$ **fresh**();
    $I \leftarrow (x' = i \wedge y' = j \wedge z' = i + 1 \wedge t' = j + 1 \wedge u' = k \wedge v' = k + 1)$; **return**
    $((\textbf{\textit{sanitize}}(\mathfrak{P}, A, i, j)) \cup \{P_\times(A, x', y', z', t', u', v')\}, \mathfrak{a} \sqcap_{\mathcal{B},\mathcal{V}} I)$
  else
    return $(\textbf{\textit{sanitize}}(\mathfrak{P}, A, i, j), \mathfrak{a})$
  end
end

Algorithm 8: FIND_EXTENSION_MULT, finds possible extensions of a monomial for the partial product

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

$P_+(A', B', C', x, y, z, t, u, v) = P$;
if $A = A' \wedge B = B' \wedge C = C'$ **then**
  if $\mathfrak{a} \sqsubseteq_{\mathcal{B},\mathcal{V}} ((x = i) \wedge (y = j) \wedge (z = i + 1) \wedge (t = j + 1) \wedge (v = k))$ **then**
    return *Some(Above)*
  else if $\mathfrak{a} \sqsubseteq_{\mathcal{B},\mathcal{V}} ((x = i) \wedge (y = j) \wedge (z = i + 1) \wedge (t = j + 1) \wedge (u + 1 = k))$ **then**
    return *Some(Below)*
  else
    return *None*
else
  return *None*
end

Algorithm 9: FIND_ZERO_MULT, finds wether a zero predicate holds in a cell

Input : P, a, A, i, j
Output: a boolean, whether a zero predicate holds in cell (i, j)

if $P_S(A', x, y, z, t, c) = P \wedge A' = A$ **then**
  return $\mathfrak{a} \sqsubseteq_{\mathcal{B},\mathcal{V}} ((x = i) \wedge (y = j) \wedge (z = i + 1) \wedge (t = j + 1) \wedge (c = 0))$
else
  return $\bot$
end

B Soundness proof

In the following $m[r]$ denotes the memory state (and r is called an extension):

$$m' : x \mapsto \begin{cases} r(x) & \text{when } x \in \mathbf{def}(r) \\ m(x) & \text{when } x \in \mathbf{def}(m) \wedge x \notin \mathbf{def}(r) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Proof of Proposition 1: Let us prove that $\forall S_0^\sharp,\ \gamma(S_0^\sharp) \subseteq \gamma(\mathbf{wf}(S_0^\sharp))$.

We prove the following invariant along the algorithm **wf**: $\gamma(S_0^\sharp) \subseteq \gamma(S^\sharp)$ where $S_0^\sharp$ is the value at the beginning of the algorithm.

- The invariant holds trivially initially
- Let us consider $\underline{S}^\sharp$ such that $\gamma(S_0^\sharp) \subseteq \gamma(\underline{S}^\sharp)$, and $(\mathfrak{P}_1, \mathfrak{a}_1) \in \underline{S}^\sharp$, $(\mathfrak{P}_2, \mathfrak{a}_2) \in \underline{S}^\sharp$ such that $\mathfrak{P}_1 =_s \mathfrak{P}_2$, as $\underline{S}^\sharp$ is well-named (by induction and Proposition hypothesis), there exists some $\sigma \in \mathbf{var}(\mathfrak{P}_1) \to \mathbf{var}(\mathfrak{P}_2)$ a bijection such that $\mathfrak{P}_1\sigma = \mathfrak{P}_2$, and $\overline{S}^\sharp = \underline{S}^\sharp \setminus \{(\mathfrak{P}_1, \mathfrak{a}_1), (\mathfrak{P}_2, \mathfrak{a}_2)\} \cup (\mathfrak{P}_2, \mathfrak{a}_1\sigma \cup_{\mathcal{V}} \mathfrak{a}_2)$. Let $m \in \gamma(S_0^\sharp)$, by induction hypothesis $m \in \gamma(\underline{S}^\sharp)$, therefore $m = (m_{\mathcal{V}}, m_{\mathcal{A}})$ is such that $\exists r, \mathfrak{P}, \mathfrak{a},\ \mathbf{def}(r) \subseteq \mathbb{Y} \wedge (\mathfrak{P}, \mathfrak{a}) \in \underline{S}^\sharp \wedge m_{\mathcal{V}}[r] \in \gamma_{\mathcal{V}}(\mathfrak{a}) \wedge \forall P \in \mathfrak{P}, (m_{\mathcal{V}}[r], m_{\mathcal{A}}) \in \mathcal{I}_{\mathcal{P}}(P)$
  - If $\mathfrak{P} \notin \{\mathfrak{P}_1, \mathfrak{P}_2\}$ then trivially $m \in \gamma(\underline{S}^\sharp \setminus \{(\mathfrak{P}_1, \mathfrak{a}_1), (\mathfrak{P}_2, \mathfrak{a}_2)\} \cup (\mathfrak{P}_2, \mathfrak{a}_1\sigma \cup_{\mathcal{V}} \mathfrak{a}_2))$ and therefore $m \in \gamma(\overline{S}^\sharp)$
  - If $\mathfrak{P} = \mathfrak{P}_1$ then $m_{\mathcal{V}}[r] \in \gamma_{\mathcal{V}}(\mathfrak{a}_1)$, thus $m_{\mathcal{V}}[r] \circ \sigma^{-1} \in \gamma_{\mathcal{V}}(\mathfrak{a}_1\sigma)$ and by soundness of the $\cup_{\mathcal{V}}$ operator we have: $m_{\mathcal{V}}[r] \circ \sigma^{-1} \in \gamma_{\mathcal{V}}(\mathfrak{a}_1\sigma \cup_{\mathcal{V}} \mathfrak{a}_2)$. Moreover we show by induction on terms, atomic predicates and predicates that $(m_{\mathcal{V}}[r], m_{\mathcal{A}}) \in \mathcal{I}_{\mathcal{P}}(P) \Rightarrow$

$(m_{\mathcal{V}}[r] \circ \sigma^{-1}, m_{\mathcal{A}}) \in \mathcal{I}_{\mathcal{P}}(P\sigma)$. Therefore $\forall P \in \mathfrak{P}_1\sigma,\ (m_{\mathcal{V}}[r] \circ \sigma^{-1}, m_{\mathcal{A}}[r]) \in \mathcal{I}_{\mathcal{P}}(P)$ and as $\mathbf{def}(\sigma^{-1}) \subseteq \mathbb{Y}$ we have $m_{\mathcal{V}} = m_{\mathcal{V}}[r] \circ \sigma^{-1}_{|\mathbb{X}\cup\mathbb{S}}$. Therefore $m \in \gamma(\overline{S}^\sharp)$.

- The case $\mathfrak{P} = \mathfrak{P}_2$ is proved in the same manner as the previous case.

Therefore the invariant holds at the end of the loop, and $\forall S_0^\sharp,\ \gamma(S_0^\sharp) \subseteq \gamma(\mathbf{wf}(S_0^\sharp))$. □

Proof of Theorem 1 An analyzer is said to be *sound* if it over-approximates the reachable states of the program. In order to prove such a result, we prove by induction on the commands that:

$$\forall C \in \mathcal{C},\ \forall S,\ \forall S^\sharp,\ S \subseteq \gamma(S^\sharp) \Rightarrow \mathbf{Post}[\![C]\!](S) \subseteq \gamma(\mathbf{Post}^\sharp[\![C]\!](S^\sharp))$$

In the following r will always denote an extension such that $\mathbf{var}(r) \subseteq \mathbb{Y}$, therefore $\forall m, m[r]_{|\mathbb{X}\cup\mathbb{S}} = m$.

- $C = X := E$: Let $S, S^\sharp$ be such that $S \subseteq \gamma(S^\sharp)$, let $m_a = (m_{a,\mathcal{V}}, m_{a,\mathcal{A}}) \in \mathbf{Post}[\![C]\!](S)$, there is $m_b = (m_{b,\mathcal{V}}, m_{b,\mathcal{A}}) \in S$ such that $m_a \in \mathbf{Post}[\![X := E]\!](\{m_b\})$. Hence from the definition of **Post**, we have $m_{a,\mathcal{A}} = m_{b,\mathcal{A}}$ and $m_{a,\mathcal{V}} = m_{b,\mathcal{V}}[X := v]$ where $v = \mathcal{E}[\![E]\!](m_b)$. From $S \subseteq \gamma(S^\sharp)$ and the definition of γ we have: $\exists r,\ \exists \mathfrak{a}, \mathfrak{P},\ (\mathfrak{a}, \mathfrak{P}) \in S^\sharp \wedge m_{b,\mathcal{V}}[r] \in \gamma_{\mathcal{V}}(\mathfrak{a}) \wedge \forall P \in \mathfrak{P}, (m_{b,\mathcal{V}}[r], m_{b,\mathcal{A}}) \in \mathcal{I}_{\mathcal{P}}(P)$, from the definition of $\mathbf{Post}^\sharp$, $(\mathfrak{P}, \mathbf{Post}^\sharp_{\mathcal{V}}[\![X := E]\!](\mathfrak{a})) \in \mathbf{Post}^\sharp[\![C]\!](S^\sharp)$, from the properties of the underlying abstract domain we have: $m_{a,\mathcal{V}} = m_{b,\mathcal{V}}[X := v] \in \gamma_{\mathcal{V}}(\mathbf{Post}^\sharp_{\mathcal{V}}[\![X := E]\!](\mathfrak{a}))$ as $v = \mathcal{E}[\![E]\!](m_b)$, moreover $X \in \mathbb{X}$ and $\mathbb{X} \cap \mathbb{Y} = \emptyset$ therefore $\forall P \in \mathfrak{P},\ X \notin \mathbf{var}(P)$, hence $\forall P \in \mathfrak{P}, (m_{b,\mathcal{V}}[r], m_{b,\mathcal{A}}) \in \mathcal{I}_{\mathcal{P}}(P) \Rightarrow (m_{b,\mathcal{V}}[r, X := v], m_{b,\mathcal{A}}) \in \mathcal{I}_{\mathcal{P}}(P)$. Therefore we have $m_{a,\mathcal{V}}[r] \in \gamma_{\mathcal{V}}(\mathbf{Post}^\sharp_{\mathcal{V}}[\![X := E]\!](\mathfrak{a})) \wedge (m_{a,\mathcal{V}}[r], m_{a,\mathcal{A}}) \in \mathcal{I}_{\mathcal{P}}(P)$, hence: $m_a \in \gamma(\mathbf{Post}^\sharp[\![X := E]\!](S^\sharp))$ and $\mathbf{Post}[\![C]\!](S) \subseteq \gamma(\mathbf{Post}^\sharp[\![X := E]\!](S^\sharp))$.
- $C = A[X_1][X_2] \leftarrow B[X_1][X_2] + C[X_1][X_2]$: Let $S, S^\sharp$ be such that $S \subseteq \gamma(S^\sharp)$, let $m_a = (m_{a,\mathcal{V}}, m_{a,\mathcal{A}}) \in \mathbf{Post}[\![C]\!](S)$, there is $m_b = (m_{b,\mathcal{V}}, m_{b,\mathcal{A}}) \in S$ such that $m_a \in \mathbf{Post}[\![A[X_1][X_2] \leftarrow B[X_1][X_2] + C[X_1][X_2]]\!](\{m_b\})$. Hence from the definition of **Post**, we have $m_{a,\mathcal{V}} = m_{b,\mathcal{V}}$ and $m_{a,\mathcal{A}} = m_{b,\mathcal{A}}[A[m_{b,\mathcal{V}}(X_1)][m_{b,\mathcal{V}}(X_2)] := m_{b,\mathcal{A}}(B)(m_{b,\mathcal{V}}(X_1), m_{b,\mathcal{V}}(X_2)) + m_{b,\mathcal{A}}(C)(m_{b,\mathcal{V}}(X_1), m_{b,\mathcal{V}}(X_2))]$. $m_b \in \gamma(S^\sharp)$ hence $\exists r,\ \exists \mathfrak{a}, \mathfrak{P} \in S^\sharp,\ m_{b,\mathcal{V}}[r] \in \gamma_{\mathcal{V}}(\mathfrak{a}) \wedge \forall P \in \mathfrak{P}, (m_{b,\mathcal{V}}[r], m_{b,\mathcal{A}}) \in \mathcal{I}_{\mathcal{P}}(P)$. We now consider the following different cases: (where we will reuse the notation introduced by Algorithms 6 and 1)
  - $\exists P_0 \in \mathfrak{P}$, FIND_EXTENSION$((P_0, \mathfrak{a}), A, B, C, X_1, X_2) =$ `Some(Right)`, then from the definition of FIND_EXTENSION we have $P_0 = P_+(A, B, C, x, y, z, t)$ and $\mathfrak{a} \sqsubseteq_{\mathcal{B},\mathcal{V}} ((z = X_1) \wedge (y = X_2) \wedge (t = X_2 + 1))$. As $(m_{b,\mathcal{V}}[r], m_{b,\mathcal{A}}) \in \mathcal{I}_{\mathcal{P}}(P_0)$ we have $\forall u, v \in [m_{b,\mathcal{V}}[r](x), m_{b,\mathcal{V}}[r](z) - 1] \times [m_{b,\mathcal{V}}[r](y), m_{b,\mathcal{V}}[r](t) - 1]$, $m_{b,\mathcal{A}}(A)(u, v) = m_{b,\mathcal{A}}(B)(u, v) + m_{b,\mathcal{A}}(C)(u, v)$ and as $m_{b,\mathcal{V}}[r] \in \gamma(\mathfrak{a})$ we have $m_{b,\mathcal{V}}[r](X_1) = m_{b,\mathcal{V}}[r](z) \wedge m_{b,\mathcal{V}}[r](X_2) = m_{b,\mathcal{V}}[r](y) \wedge m_{b,\mathcal{V}}[r](t) = m_{b,\mathcal{V}}[r](X_2) + 1$. From the definition of **Post**, we have $m_{b,\mathcal{A}}(A)(m_{a,\mathcal{V}}(X_1), m_{a,\mathcal{V}}(X_2)) = m_{b,\mathcal{A}}(B)(m_{b,\mathcal{V}}(X_1), m_{b,\mathcal{V}}(X_2)) + m_{b,\mathcal{A}}(C)(m_{b,\mathcal{V}}(X_1), m_{b,\mathcal{V}}(X_2))$ therefore $m_{a,\mathcal{A}}(A)(m_{a,\mathcal{V}}[r](z), m_{a,\mathcal{V}}[r](y)) = m_{a,\mathcal{A}}(B)(m_{a,\mathcal{V}}[r](z), m_{a,\mathcal{V}}[r](y)) + m_{a,\mathcal{A}}(C)(m_{a,\mathcal{V}}[r](z), m_{a,\mathcal{V}}[r](y))$, and as $m_{a,\mathcal{V}}[r](t) = m_{a,\mathcal{V}}[r](y) + 1$ we have $\forall u, v \in [m_{a,\mathcal{V}}[r](x), m_{a,\mathcal{V}}[r](z)] \times [m_{a,\mathcal{V}}[r](y), m_{a,\mathcal{V}}[r](t) - 1]$, $m_{a,\mathcal{A}}(A)(u, v) = m_{a,\mathcal{A}}(B)(u, v) + m_{a,\mathcal{A}}(C)(u, v)$. As x', y', z', t' are fresh variables and as the only available predicate states that the addition of B and C holds in some cells of A (thanks to the hypothesis of Sect. 4) then $m_{a,\mathcal{V}}[r, x' \to x, y' \to y, z' \to z + 1, t' \to t], m_{a,\mathcal{A}}) \in \mathcal{I}_{\mathcal{P}}(\mathfrak{P} \setminus P_0)$. Finally $m_{a,\mathcal{V}}[r, x' \to x, y' \to y, z' \to z + 1, t' \to t], m_{a,\mathcal{A}}) \in \mathcal{I}_{\mathcal{P}}((\mathfrak{P} \setminus P_0) \cup \{P_+(A, B, C, x', y', z', t')\})$ and as $m_{a,\mathcal{V}}[r, x' \to x,$

$y' \to y, z' \to z+1, t' \to t] \in \gamma_{\mathcal{V}}(\mathfrak{a} \sqcap_{\mathcal{B},\mathcal{V}} I)$ we have $m_a = (m_{a,\mathcal{V}}[r, x' \to x, y' \to y, z' \to z+1, t' \to t]_{|\mathbb{X} \cup \mathbb{S}}, m_{a,\mathcal{A}}) \in \gamma(\mathbf{Post}^\sharp[\![C]\!](S^\sharp))$

- The cases Down, Left, Up are very similar.
- $\forall P_0 \in \mathfrak{P}$, FIND_EXTENSION$((P_0, \mathfrak{a}), A, B, C, X_1, X_2) = \texttt{None}$
  - Case $P_+(A', B', C', \_, \_, \_, \_) \in \mathfrak{P}$ with $A' = A \wedge B' = B \wedge C' = C$.
    - Case $\sharp\mathfrak{P} < M_{pred}$: $m_{a,\mathcal{V}}[r, x' \leftarrow X_1, y' \leftarrow X_2, z' \leftarrow X_1 + 1, t' \leftarrow X_2 + 1] \in \gamma_{\mathcal{V}}(\mathfrak{a} \sqcap_{\mathcal{B},\mathcal{V}} x' = X_1 \wedge y' = X_2 \wedge z' = X_1 + 1 \wedge t' = X_2 + 1)$. As $m_{a,\mathcal{A}}(A)(m_{b,\mathcal{V}}(X_1), m_{b,\mathcal{V}}(X_2)) = m_{b,\mathcal{A}}(B)(m_{b,\mathcal{V}}(X_1), m_{b,\mathcal{V}}(X_2)) + m_{b,\mathcal{A}}(C)(m_{b,\mathcal{V}}(X_1), m_{b,\mathcal{V}}(X_2))$, that is $\forall u, v \in [m_{b,\mathcal{V}}(x'), m_{b,\mathcal{V}}(z') - 1] \times [m_{b,\mathcal{V}}(y'), m_{b,\mathcal{V}}(t') - 1]$, $m_{a,\mathcal{A}}(A)(u, v) = m_{a,\mathcal{A}}(B)(u, v) + m_{a,\mathcal{A}}(C)(u, v)$, thus proving $(m_{a,\mathcal{V}}[r, x' \leftarrow X_1, y' \leftarrow X_2, z' \leftarrow X_1 + 1, t' \leftarrow X_2 + 1]_{|\mathbb{X} \cup \mathbb{S}}, m_{a,\mathcal{A}}) \in \mathcal{I}_{\mathcal{P}}(P_+(A, B, C, x', y', z', t'))$, and as variables x', y', z', t' are fresh we still have : $\forall P \in \mathfrak{P}$, $(m_{a,\mathcal{V}}[r, x' \leftarrow X_1, y' \leftarrow X_2, z' \leftarrow X_1 + 1, t' \leftarrow X_2 + 1], m_{a,\mathcal{A}}) \in \mathcal{I}_{\mathcal{P}}(P)$ therefore proving that $m_a \in \gamma(\mathbf{Post}^\sharp[\![C]\!](S^\sharp))$.
    - Case $\sharp\mathfrak{P} \geq M_{pred}$: Trivial from $A' = A \wedge B' = B \wedge C' = C$.
  - Otherwise: $m_{a,\mathcal{V}}[r] = m_{b,\mathcal{V}}[r] \in \gamma_{\mathcal{V}}(\mathfrak{a})$, moreover $(m_{a,\mathcal{V}}, m_{a,\mathcal{A}}) \in \mathcal{I}_{\mathcal{P}}(\emptyset)$. Thus $m_a \in \gamma(\mathbf{Post}^\sharp[\![C]\!](S^\sharp))$.

– $C = C_1 \ ; \ C_2$: Let $S, S^\sharp$ be such that $S \subseteq \gamma(S^\sharp)$, by induction hypothesis applied on C_1 we have $\mathbf{Post}[\![C_1]\!](S) \subseteq \gamma(\mathbf{Post}^\sharp[\![C_1]\!](S^\sharp))$ and now applied on C_2: $\mathbf{Post}[\![C_2]\!](\mathbf{Post}[\![C_1]\!](S)) \subseteq \gamma(\mathbf{Post}^\sharp[\![C_2]\!](\mathbf{Post}[\![C_1]\!](S))) \subseteq \gamma(\mathbf{Post}^\sharp[\![C_2]\!](\mathbf{Post}^\sharp[\![C_1]\!](S^\sharp)))$ (by monotony), hence from the definitions: $\mathbf{Post}[\![C_1; C_2]\!](S) \subseteq \gamma(\mathbf{Post}^\sharp[\![C_1; C_2]\!](S^\sharp))$.

– $C = \texttt{If}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2$: The proof for this point is not stated here as the analysis we proposed for this command is classic and its soundness is ensured by the soundness of the underlying abstraction.

– $\texttt{While}\ B\ \texttt{do}\ C\ \texttt{done}$: The proof for this point is not stated here as the analysis we proposed for this command is classic and its soundness is ensured by the soundness of the underlying abstraction. However for the usual proof to work we need to show the two following lemmas. □

Lemma 1 $\forall S^\sharp, \gamma(S^\sharp) \subseteq \gamma(\mathbf{Merge}_+(S^\sharp))$.

Proof Let $m_b \in \gamma(S^\sharp)$, by definition: $\exists r, \exists(\mathfrak{P}, \mathfrak{a}) \in S^\sharp, m_{b,\mathcal{V}}[r] \in \gamma_{\mathcal{V}}(\mathfrak{a}) \wedge \forall P \in \mathfrak{P}, (m_{b,\mathcal{V}}[r], m_{b,\mathcal{A}}) \in \mathcal{I}_{\mathcal{P}}(P)$. We now show by induction on the loop in $\mathbf{Merge}_{+,\mathfrak{M}}$ (defined in Algorithm 2) the following invariant (we use notations of Algorithm 2): $m_b \in \gamma(\{\mathfrak{P}_o, \mathfrak{a}_o\})$. This is initially true thanks to the previous remark. Now let $(P_0, P_1) \in \mathfrak{P}_o$, FIND_MERGE$(P_0, P_1, \mathfrak{a}_o) = \texttt{Some(Right)}$, with $P_0 = P_+(A, B, C, x_0, y_0, z_0, t_0)$ and $P_1 = P_+(A, B, C, x_1, y_1, z_1, t_1)$, thanks to FIND_MERGE we have $\mathfrak{a}_o \sqsubseteq_{\mathcal{B},\mathcal{V}} ((x_0 = x_1) \wedge (z_0 = z_1) \wedge (t_0 = y_1))$. By induction $\exists r, m_{b,\mathcal{V}}[r] \in \gamma_{\mathcal{V}}(\mathfrak{a}_o) \wedge \forall P \in \mathfrak{P}_o\, (m_{b,\mathcal{V}}[r], m_{b,\mathcal{A}}) \in \mathcal{I}_{\mathcal{P}}(P)$ hence $m_{b,\mathcal{V}}[r, x' \to x_0, y' \to y_0, z' \to z_1, t' \to t_1] \in \gamma_{\mathcal{V}}(\mathfrak{a}_o \sqcap_{\mathcal{B},\mathcal{V}} I)$, moreover $\forall u, v \in [m_{b,\mathcal{V}}[r](x_0), m_{b,\mathcal{V}}[r](z_0) - 1] \times [m_{b,\mathcal{V}}[r](y_0), m_{b,\mathcal{V}}[r](t_0) - 1]$, $m_{a,\mathcal{A}}(A)(u, v) = m_{a,\mathcal{A}}(B)(u, v) + m_{a,\mathcal{A}}(C)(u, v)$ and $\forall u, v \in [m_{b,\mathcal{V}}[r](x_1), m_{b,\mathcal{V}}[r](z_1) - 1] \times [m_{b,\mathcal{V}}[r](y_1), m_{b,\mathcal{V}}[r](t_1) - 1]$, $m_{a,\mathcal{A}}(A)(u, v) = m_{a,\mathcal{A}}(B)(u, v) + m_{a,\mathcal{A}}(C)(u, v)$ and as $m_{b,\mathcal{V}}[r](x_0) = m_{b,\mathcal{V}}[r](x_1)$, $m_{b,\mathcal{V}}[r](z_0) = m_{b,\mathcal{V}}[r](z_1)$ and $m_{b,\mathcal{V}}[r](t_0) = m_{b,\mathcal{V}}[r](y_1)$ we get $\forall u, v \in [m_{b,\mathcal{V}}[r](x_0), m_{b,\mathcal{V}}[r](z_1) - 1] \times [m_{b,\mathcal{V}}[r](y_0), m_{b,\mathcal{V}}[r](t_1) - 1]$, $m_{a,\mathcal{A}}(A)(u, v) = m_{a,\mathcal{A}}(B)(u, v) + m_{a,\mathcal{A}}(C)(u, v)$ hence $(m_{b,\mathcal{V}}[r, x' \to x_0, y' \to y_0, z' \to z_1, t' \to t_1], m_{b,\mathcal{A}}) \in \mathcal{I}_{\mathcal{P}}(P_+(A, B, C, x', y', z', t'))$. Thus ensuring the invariant $m_b \in \gamma(\{\mathfrak{P}_o, \mathfrak{a}_o\})$. The case $(P_0, P_1) \in \mathfrak{P}_o$, FIND_MERGE$(P_0, P_1, \mathfrak{a}_o) = \texttt{Some(Down)}$ is identical. We note that $\mathbf{Merge}_{+,\mathfrak{M}}$ terminates because the number of predicates appearing in $\mathfrak{P}_o$

is strictly decreasing. The previous invariant gives us that $m_b \in \gamma(\mathbf{Merge}_{+,\mathfrak{M}}(\mathfrak{P}, \mathfrak{a}))$ hence $\gamma(S^\sharp) \subseteq \gamma(\mathbf{Merge}_+ S^\sharp))$. □

Lemma 2 (Widening ensures convergence) *For all well-formed, well-named sequences* $(V_n^\sharp)_{n\in\mathbb{N}}$ *such that* $V_{n+1}^\sharp = V_n^\sharp \triangledown f(V_n^\sharp)$, $(V_n^\sharp)_{n\in\mathbb{N}}$ *converges.*

Proof Due to the widening definition, the sequence $V_n^\sharp$ has constant shape, thus allowing us (modulo a renaming of variables) to write $\forall n \in \mathbb{N}$, $V_n^\sharp = \bigcup_{i\in I}(\mathfrak{P}_i, \mathfrak{a}_{n,i,0})$, $\forall n \in \mathbb{N}$, $f(V_n^\sharp) = \bigcup_{i\in I}(\mathfrak{P}_i, \mathfrak{a}_{n,i,1})$ (thanks to the widening definition, $V_n^\sharp$ and $f(V_n^\sharp)$ must have the same shape). Moreover, $\forall n \in \mathbb{N}$, $V_n^\sharp$ is well-formed, therefore $\forall i \neq j \in I^2$, $\mathfrak{P}_i$ and $\mathfrak{P}_j$ have different shape, hence $\forall n \in \mathbb{N}$, $V_{n+1}^\sharp = \bigcup_{i\in I}(\mathfrak{P}_i, \mathfrak{a}_{n,i,0} \triangledown_\mathcal{V} \mathfrak{a}_{n,i,1}) = \bigcup_{i\in I}(\mathfrak{P}_i, \mathfrak{a}_{n+1,i,0})$ with I a finite set. As $\forall i \in I$, $\mathfrak{a}_{n+1,i,0} = \mathfrak{a}_{n,i,0} \triangledown_\mathcal{V} \mathfrak{a}_{n,i,1}$, the properties of $\triangledown_\mathcal{V}$ ensure that $\forall i \in I$, $(\mathfrak{a}_{n,i,0})_{n\in\mathbb{N}}$ converges. Therefore (I being finite) the sequence $(V_n^\sharp)$ converges. □

Lemma 3 *The analysis terminates.*

Proof We remind that the analyzer is only allowed a finite number M_{pred} of synchronous use of the addition predicate in monomials hence ensuring that well-formed abstract states have a finite number of possible shapes, more precisely $2^{M_{pred}+1}$ different shapes, we consider the following shape function: $\mathbf{sh}(S^\sharp) = \bigcup_{(\mathfrak{P},\mathfrak{a})\in S^\sharp}\{\sharp(\mathfrak{P})\}$ as we use only an addition predicate, it is easy to show that two abstract states $S_1^\sharp$ and $S_2^\sharp$ have the same shape if and only if $\mathbf{sh}(S_1^\sharp) = \mathbf{sh}(S_2^\sharp)$. Let us now consider a sequence with $U_0^\sharp$ well-formed and $U_{n+1}^\sharp =$ `let` $S_P^\sharp = U_n^\sharp \sqcup \mathbf{Post}^\sharp[\![C]\!](U_n^\sharp \sqcap_\mathcal{B} B)$ `in if` $S_P^\sharp =_F U_n^\sharp$ `then` $U_n^\sharp \triangledown S_P^\sharp$ `else` $S_P^\sharp$.The definition of $\sqcup$ ensures that $\forall n$, $U_n^\sharp$ is well-formed. By definition of $\sqcup$ and the **wf** function we have that $\forall A^\sharp$, $\mathbf{sh}(U_n^\sharp) \subseteq \mathbf{sh}(U_n^\sharp \sqcup A^\sharp)$, by definition of the widening $\forall A^\sharp, B^\sharp$, $A^\sharp =_F B^\sharp \Rightarrow A^\sharp \triangledown B^\sharp =_F A^\sharp$ hence ensuring that $\mathbf{sh}(U_n^\sharp) \subseteq \mathbf{sh}(U_{n+1}^\sharp)$, moreover $\forall n$, $\mathbf{sh}(U_n) \subseteq \{0, \ldots, M_{pred}\}$, hence the sequence $\mathbf{sh}(U_n)$ converges. Let us consider the sub-sequence $V_n^\sharp = U_{n+m}^\sharp$ such that $\forall n$, $\mathbf{sh}(V_n^\sharp) = \mathbf{sh}(V_0^\sharp)$, therefore $V_{n+1}^\sharp = V_n^\sharp \triangledown (V_n^\sharp \sqcup \mathbf{Post}^\sharp[\![C]\!](V_n^\sharp \sqcap_\mathcal{B} B))$. Finally the properties of $\triangledown$ (Lemma 2) ensures the convergence of $(V_n^\sharp)_{n\in\mathbb{N}}$ and thus of $(U_n^\sharp)_{n\in\mathbb{N}}$. The termination proof in this case could be made by more simple consideration, however the idea of defining a shape function and showing its convergence can be reused to prove the convergence of other abstractions we have set up. □

C Programs

C.1 Addition tiled

C.2 GEMM

```
/* n >= 10 */; /* n = 32 * a + b*/; /* 1 <= b < 32*/
Array A of n n; Array B of n n; Array C of n n;
i := 0;
while (i < a) do
 j := 0;
 while (j < a) do
  ii := 0;
  while (ii < 32) do
   jj := 0;
   while (jj < 32) do
    x = 32 * i + ii; y = 32 * j + jj;
    A[x][y] <- B[x][y] + C[x][y];
    jj := jj + 1;
   done; ii := ii + 1;
  done; j := j + 1;
 done;
 ii := 0;
 while (ii < 32) do
  jj := 0;
  while (jj < b) do
   x = 32 * i + ii; y = 32 * j + jj;
   A[x][y] <- B[x][y] + C[x][y];
   jj := jj + 1;
  done; ii := ii + 1;
 done; i := i + 1;
done;
j := 0;
while (j < a) do
 ii := 0;
 while (ii < b) do
  jj := 0;
  while (jj < 32) do
   x = 32 * i + ii; y = 32 * j + jj;
   A[x][y] <- B[x][y] + C[x][y];
   jj := jj + 1;
  done; ii := ii + 1;
 done; j := j + 1;
done;
ii := 0;
while (ii < b) do
 jj := 0;
 while (jj < b) do
  x = 32 * i + ii; y = 32 * j + jj;
  A[x][y] <- B[x][y] + C[x][y];
  jj := jj + 1;
 done; ii := ii + 1;
done;
```

Program 10 Addition tiled with reminder using unroll

```
/*n >= 5*/;
m := n;
Array A of n of n;
Array B of n of n;
Array D of n of n;
Array E of n of n;
zz := 0;
if (alpha = 0) {
 if (beta = 0) {
  t2 := 1;
   while (t2 <= n) do
    t1 := 1;
    while (t1 <= m) do
     i := t1 - 1;
     j := t2 - 1;
     E[i][j] <- zz;
     t1 ++
    done;
    t2 ++
   done
  } else {
   t2 := 1;
   while( t2 <= n) do
    t1 := 1;
    while (t1 <= m) do
     i := t1 -1;
     j := t2 -1;
     u <- D[i][j];
     uu <- u * beta;
     E[i][j] <- uu;
     t1++
    done;
    t2 ++
   done
 }
} else {
 t2 := 1;
 while (t2 <= n) do
  if (beta = 0) {
   t1 := 1;
   while (t1 <= m) do
    i := t1 -1;
    j := t2 -1;
    c := 0;
    u <- D[i][j];
    uu <- u * c;
    E[i][j] <- uu;
    t1++
   done
  } else {
   t1 := 1;
   while (t1 <= m) do
    i := t1 -1;
    j := t2 -1;
    u <- D[i][j];
    uu <- u * beta;
    E[i][j] <- uu;
    t1 ++
   done
  };
  t3 := 1;
  while (t3 <= n) do
   k := t3 -1;
   j := t2 -1;
   u <- B[k][j];
   prev <- alpha*u;
   t1 := 1;
   while (t1 <= m) do
    i := t1 -1;
    v <- E[i][j];
    aa <- A[i][k];
    w <- prev * aa;
    zz <- v + w ;
    E[i][j] <- zz;
    t1 ++
   done;
   t3 ++
  done;
  t2++
 done
}
```

Program 11 GEMM

References

1. Cousot P, Cousot R (1977) Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the POPL 1977. ACM, pp 238–252
2. Blanchet B, Cousot P, Cousot R, Feret J, Mauborgne L, Miné A, Monniaux D, Rival X (2003) A static analyzer for large safety-critical software. In: Proceedings of the PLDI'03. ACM, pp 196–207
3. Allamigeon X (2008) Non-disjunctive numerical domain for array predicate abstraction. In: SOP 2008, volume 4960 of LNCS. Springer, pp 163–177
4. Cousot P (2003) 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. Springer, pp 243–268
5. Cousot P, Cousot R, Logozzo F (2011) A parametric segmentation functor for fully automatic and scalable array content analysis. In: Proceedings of POPL 2011. ACM, pp 105–118
6. Cousot P, Halbwachs N (1978) Automatic discovery of linear restraints among variables of a program. In: Proceeings of POPL 1978. ACM, pp 84–96
7. Bondhugula U, Baskaran M, Krishnamoorthy S, Ramanujam J, Rountev A, Sadayappan P (2008) Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In: CC 2008, volume 4959 of LNCS. Springer, pp 132–146
8. Bondhugula U, Hartono A, Ramanujam J, Sadayappan P (2008) A practical automatic polyhedral parallelizer and locality optimizer. In: Proceedings of the PLDI 2008. ACM, pp 101–113
9. Leroy X (2006) Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: Proceedings of the POPL 2006. ACM, pp 42–54
10. Journault M, Miné A (2016) Static analysis by abstract interpretation of the functional correctness of matrix manipulating programs. In Xavier R (eds) Static analysis—23rd international symposium, SAS 2016, Edinburgh, UK, September 8–10, 2016, proceedings, volume 9837 of lecture notes in computer science. Springer, pp 257–277
11. Venet A (1996) Abstract cofibered domains: application to the alias analysis of untyped programs. In: SAS'96, volume 1145 of LNCS. Springer, pp 366–382
12. Rival X, Mauborgne L (2007) The trace partitioning abstract domain. ACM Trans. Program. Lang. Syst. 29(5):26
13. Cousot P, Cousot R (2002) Modular static program analysis. In: Horspool RN (ed) Compiler construction, 11th international conference, CC 2002, held as part of the joint European conferences on theory and practice of software, ETAPS 2002, Grenoble, France, April 8–12, 2002, proceedings, volume 2304 of lecture notes in computer science. Springer, pp 159–178
14. Miné A (2006) Symbolic methods to enhance the precision of numerical abstract domains. In: VMCAI 2006, volume 3855 of LNCS. Springer, pp 348–363
15. Miné A (2006) The octagon abstract domain. Higher Order Symb Comput (HOSC) 19(1):31–100 http://www-apr.lip6.fr/~mine/publi/article-mine-HOSC06.pdf
16. Gopan D, DiMaio F, Dor N, Reps T, Sagiv M (2004) Numeric domains with summarized dimensions. In: TACAS 2004. Springer, pp 512–529
17. Gopan D, Reps TW, Sagiv S (2005) A framework for numeric analysis of array operations. In: Proceedings of POPL 2005. ACM, pp 338–350
18. Halbwachs N, Péron M (2008) Discovering properties about arrays in simple programs. SIGPLAN Not 43(6):339–348
19. Allamigeon X, Godard W, Hymans C (2006) Static analysis of string manipulations in critical embedded C programs. In: SAS 2006. Springer, pp 35–51
20. Dillig I, Dillig T, Aiken A (2010) Fluid updates: beyond strong vs. weak updates. In: ESOP 2010. Springer, pp 246–266
21. Monniaux D, Alberti F (2015) A simple abstraction of arrays and maps by program translation. In: SAS 2015, volume 9291 of LNCS. Springer, pp 217–234
22. Peng Y. Automate convergence rate proof for gradient descent on quadratic functions
23. Gunnels JA, Gustavson FG, Henry G, van de Geijn RA (2001) FLAME: formal linear algebra methods environment. ACM Trans. Math. Softw. 27(4):422–455
24. Henzinger TA, Hottelier T, Kovács L, Voronkov A (2010) Invariant and type inference for matrices. In: Barthe G, Hermenegildo MV (eds) Verification, model checking, and abstract interpretation, 11th international conference, VMCAI 2010, Madrid, Spain, January 17–19, 2010, proceedings, volume 5944 of lecture notes in computer science. Springer, pp 163–179