An array content static analysis based on non-contiguous partitions

Jiangchao Liu, Xavier Rival

An Array Content Static Analysis Based on Non Contiguous Partitions ☆,☆☆

Jiangchao Liu, Xavier Rival

*ENS,CNRS,INRIA,PSL**
45, rue d'Ulm 75230 Paris Cedex 05 - France

Abstract

Conventional array partitioning analyses split arrays into contiguous partitions to infer properties of sets of cells. Such analyses cannot group together non contiguous cells, even when they have similar properties. In this paper, we propose an abstract domain which utilizes semantic properties to split array cells into groups. Cells with similar properties will be packed into groups and abstracted together. Additionally, groups are not necessarily contiguous. This abstract domain allows to infer complex array invariants in a fully automatic way. Experiments on examples from the Minix 1.1 memory management and a tiny industrial operating system demonstrate the effectiveness of the analysis.

1. Introduction

Arrays are ubiquitous, yet their mis-use often causes software defects. Therefore, a large number of works address the automatic verification of array manipulating programs. In particular, partitioning abstractions [12, 18, 20] split arrays into sets of contiguous groups of cells (also called *segments*), in order to, hopefully, infer that they enjoy similar properties. A traditional example is that of an initialization loop, with the usual invariant that splits the array into an initialized zone (the segment from index 0 to the current index) and an uninitialized region (the segment from the current index to the end of the array).

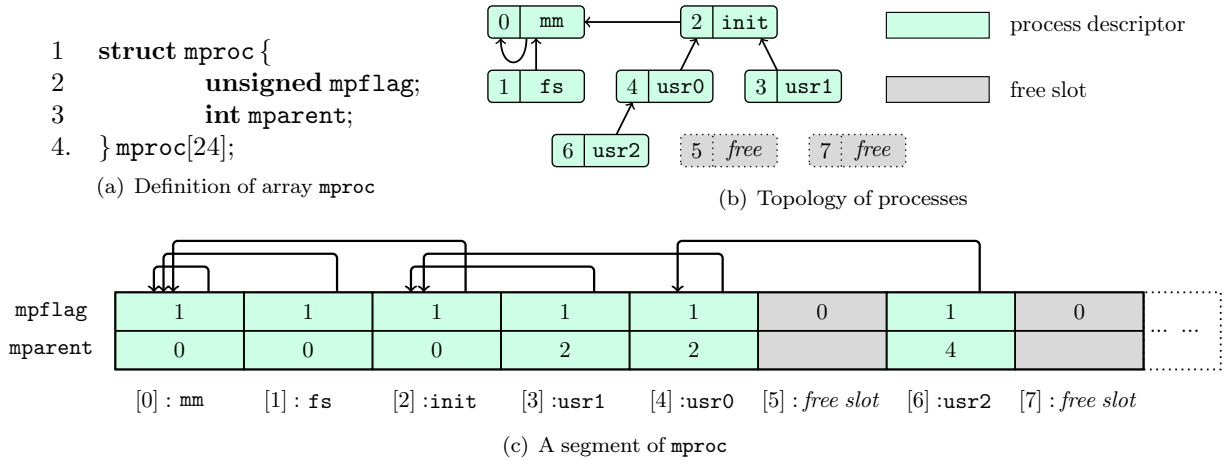
However, when cells that have similar properties are not contiguous, these approaches cannot infer adequate array partitions. This happens for unsorted arrays of structures, when there is no relation between indexes and cell fields. Sometimes the partitioning of array elements relies on relations among cell fields. This phenomenon can be observed in low-level software, such as operating system services and critical embedded systems drivers, which rely on static array zones instead of dynamically allocated blocks [30]. When cells with similar properties are not contiguous, traditional partition based techniques are unlikely to infer relevant partitions / precise array invariants.

Figure 1 illustrates the Minix 1.1 Memory Management Process Table (MMPT) main structure. The array of structures `mproc` defined in Figure 1(a) stores the process descriptors. Each descriptor comprises a field `mparent` that stores the index of the parent process in `mproc`, and a field `mpflag` that stores the process status. Figure 1(c) depicts the concrete values stored in `mproc` to describe the process topology shown in Figure 1(b) (the whole `mproc` table consists of 24 slots, here we show only 8, for the sake of space). An element of `mproc` is a process descriptor when its field `mpflag` is strictly positive and a free slot if it is null. Minix 1.1 uses the three initial elements of `mproc` to store the descriptors of the memory management service, the file system service and the init process. Descriptors of other processes appear in a random order. In the example of Figure 1, `init` has two children whose descriptors are in `mproc`[3] and `mproc`[4]; similarly, the process corresponding

☆This document is a collaborative effort.

☆☆The research leading to these results has received funding from the European Research Council under the European Union's seventh framework programme (FP7/2007-2013), grant agreement 278673, Project MemCAD, and from the ARTEMIS Joint Undertaking no 269335 (see Article II.9 of the JU Grant Agreement).

Email addresses: `jliu@di.ens.fr` (Jiangchao Liu), `rival@di.ens.fr` (Xavier Rival)



(a) Definition of array `mproc`

(b) Topology of processes

(c) A segment of `mproc`

Figure 1: Minix 1.1 Memory Management Process Table (MMPT) structure

to `mproc`[4] has a single child the descriptor of which is in `mproc`[6]. Moreover, Minix assumes a parent-child relation between `mm` and `fs`, as `mm` has index 0 and the parent field of `fs` stores 0. To abstract the process table state, valid process descriptors and free slots should be partitioned into *different groups*.

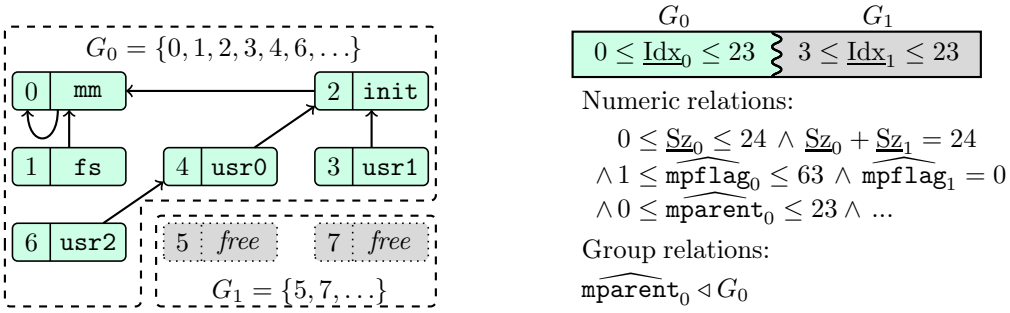
Traditional, contiguous partitioning cannot achieve this for two reasons: (1) the order of process descriptors in `mproc` cannot be predicted, hence is random in practice, and (2) there is no simple description of the boundaries between these regions (or even their sizes) in the program state. The symbolic abstract domain by Dillig, Dillig and Aiken [15] also fails here as it cannot attach arbitrary abstract properties to summarized cells.

In this paper, we set up an abstract domain to partition the array into non contiguous groups for process descriptors and free slots so as to infer this partitioning and precise invariants (Section 2) automatically. Our contributions are:

- An abstract domain that partitions array elements according to semantic properties, and can represent non contiguous partitions (Section 4);
- Static analysis algorithms for the computation of abstract post-conditions (Sections 5 and 6), widening and inclusion check (Section 7);
- The implementation and the evaluation of the analysis on the inference of tricky invariants in excerpts of some operating systems (e.g. Minix 1.1) and other challenging array examples (Section 8 and 9).

2. Overview

Minix is a Unix-like multitasking computer operating system [31]. It is a very small OS (with fewer than 10 000 lines of kernel), yet it greatly influenced the design of other kernels, including Linux. It is based on a micro-kernel architecture, with separate, lightweight *services* respectively in charge of *task scheduling* (in kernel), *memory management* and *file system*. Each service maintains a *process table* that describes the processes currently running. The tables of distinct services are consistent with each other. In the following, we consider the process table of the memory management service, which is very similar to that of the other services (and is quite representative of process table structures in operating systems kernels). This process table consists of an array `mproc` that stores the memory management information for each process in a distinct slot. As in all Unix operating systems, processes form a reversed tree, where each process has a reference to its parent (which created it) and is referred to by its children (which it created).



(a) A concrete state of `mproc`

(b) Abstract state $\mathcal{C}$ with partition

Figure 2: A partitioning of `mproc` based on non contiguous groups

New processes can be created by the system call `fork` from a parent process. A process exits after it calls `exit` and its parent calls `wait`. These two system calls form a synchronization barrier and the process and its parent are set to be "hanging" and "waiting" respectively when they reach the barrier first. In Figure 1(b), the process described by `mproc`[4] would be "hanging" after it calls `exit` if `mproc`[2] is not "waiting", and after `mproc`[2] calls `wait`, `mproc`[4] will exit. System calls `fork`, `wait` and `exit` are first handled by *memory management* and then passed on to *task scheduling* and *file system* if needed. One function and three system calls perform atomic changes on the process table structure:

- function `mm_init` is called when the operating system is initialized and constructs slots in `mproc` for the first three system level processes;
- system call `fork` creates a process descriptor in `mproc` if there exists at least one free slot, and returns an error code otherwise;
- system call `wait` frees the *hanging* descendants of the parameter process and sets itself to *waiting* if it has decendants that are not *hanging*;
- system call `exit` frees the parameter process and its *hanging* descendants if its parent is *waiting*, or sets itself to *hanging*.

Invariants. The global invariant of the process table is very complex, as it describes the reversed tree shown in Figure 1(b). In this paper, we consider the verification of the *memory safety* of the operations on the Minix memory management process table (and of other similar process tables, as we remarked they are overall quite similar), which actually does not require the verification of that strong invariant, but can be established by verifying a weaker set of properties instead:

- Each valid process descriptor has an `mparent` field, that should store a value in $[0, 23]$ (since the length of array `mproc` is 24), hence represents a valid index in `mproc`: this entails the absence of out-of-bound accesses in process table management functions;
- The `mparent` field of any valid process descriptor should be the index of a valid process descriptor: as a process can only complete its exit phase when its parent calls wait, failure to maintain a parent for each process could cause a terminating process to become dangling and never be eliminated.

We call the conjunction of these properties $\mathcal{C}$ (we will formalize it later), and to verify that the system preserve them, we propose to check that they are invariants, that is that (1) initialization establishes $\mathcal{C}$ and that (2) system calls `fork`, `wait` and `exit` preserve $\mathcal{C}$. To achieve this, we design a fully automatic, abstract interpretation-based static analysis.

Abstraction. To capture the properties expressed by $\mathcal{C}$, the analysis should utilize an abstraction that *splits* the set of cells in the array `mproc` into two *groups* of cells: the first group consists of valid process descriptors whereas the second group collects the free slots. However, we notice that each of these groups of cells is not contiguous in general. Indeed, while the slots representing valid processes are allocated from the beginning of the array at the start-up, as soon as some processes terminate, the group of slots representing valid processes is not contiguous anymore. For instance, the process state shown in Figure 2(a) corresponds to the table of Figure 1(c), where both the group of valid processes and the group of free slots are non contiguous. In this picture, group 0 consists of all the valid process descriptors whereas group 1 collects all free slots.

Once the cells of the array are partitioned into these two groups, the values of the individual fields of the slots can be abstracted in a rather precise manner as shown in Figure 2(b).

We let G_i denote the set of indexes of all the elements in group i. The abstract state shown in Figure 2(b) ties each group to properties of its elements. These will be formally defined in Section 4. By the Minix specification, the elements of group 0 satisfy the following correctness conditions in $\mathcal{C}$:

- their indexes are in $[0, 23]$, which we note $0 \leq \underline{\mathtt{Idx}}_0 \leq 23$ in Figure 2(b);
- the size of group 0 is between 0 and 24, which we note $0 \leq \underline{\mathtt{Sz}}_0 \leq 24$;
- their flag fields are in $[1, 63]$, which we note $1 \leq \widehat{\mathtt{mpflag}}_0 \leq 63$; field `mpflag` uses 6 bits to indicate the state of that cell (including in_use, waiting and hanging), and valid process descriptors have a strictly positive flag;
- their parents are valid indexes, which we note $0 \leq \widehat{\mathtt{mparent}}_0 \leq 23$;
- their parent fields are indexes of valid process descriptors, hence are also in group 0, which we note $\widehat{\mathtt{mparent}}_0 \triangleleft G_0$.

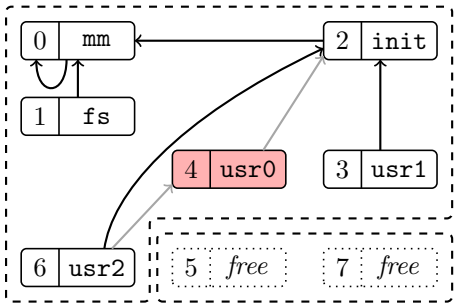
This abstraction relies on *disjoint* groups as other array partitioning abstractions [18, 20]. However, our abstraction *does not* assume each group consists of a contiguous set of cells. The *non-contiguousness* of groups is represented by a winding separation line in Figure 2(b). To characterize groups, our abstraction relies not only on constraints on indexes, but also on semantic properties of the cell contents: while groups 0 and 1 correspond to a similar range, the `mpflag` values of their elements are different (any value in $[1, 63]$ in group 0 and 0 in group 1). Therefore our abstraction can abstract both contiguous and non contiguous partitions. In this example, we believe the abstract state of Figure 2(b) is close to the programmer's intent, where the array is a collection of unsorted elements.

Analysis. We now consider the verification of the functions that operate on the Minix memory management process table. In the remainder of this section, we focus on auxiliary function `cleanup`, which is called by `wait` and `exit`, and that turns elements of `mproc` that describe *hanging* processes into free slots. This function provides a representative view of the challenges that arise when analyzing the other functions manipulating this process table. It consists of a case split, depending of the nature of the process to cleanup. Figure 3(a) displays an excerpt of a de-recursified version of `cleanup`, which handles the case where the process being cleaned-up is a child of `init`. This situation arises if we consider calling `cleanup`(4) in the state shown in Figure 2(a): indeed, this will cause the removal of user process `usr0` which has `init` as parent; this means that process `usr2` should become a child of `init`, while the record formerly associated to `usr0` turns into a free slot, the result is shown in Figure 3(b).

The correctness of the whole process table management relies on the fact that system calls `wait` and `exit` will always call `cleanup` in a state where the process table is correct, and where the process that will be cleaned up is a currently valid user process. This means that function `cleanup` should always be called in a state that satisfies the pre-condition defined by the correctness conditions listed in the previous paragraph, and where argument `child` is in group 0 (i.e., is valid), and greater than 2 (user process), which we note $\mathtt{child} \triangleleft G_0 \wedge \mathtt{child} > 2$. Figure 4 overviews the main local invariants, that can be obtained on the excerpt of `cleanup`, and starting from that pre-condition. To compute this automatically, our analysis shall proceed by performing a forward abstract interpretation of the procedure, computing sound post-conditions and loop

```
void cleanup(int child){
  ...
  int parent = mproc[child].mparent;
  if(parent == 2){
    mproc[child].mpflag = 0;
    i = 0;
    while(i < 24){
        if(mproc[i].mpflag > 0)
          if(mproc[i].mparent == child)
            mproc[i].mparent = 2;
        i = i + 1;
    }
  }
  else{
    \\cleanup child and its descendants
  }
}
```

(a) A simplified excerpt of `cleanup`



(b) Effect of `cleanup`

Figure 3: Minix 1.1 process table management, system function `cleanup`

invariants [10]. In this section, we discuss the main original aspects of this analysis, namely (1) cell materialization (to allow strong updates), (2) termination of the loop analysis and (3) pruning of unnecessary groups.

As remarked above, function `cleanup` should be called only in states that satisfy $\mathcal{C}$, and where predicate $\texttt{child} \triangleleft G_0 \wedge \texttt{child} > 2$ holds (which means `mproc[child]` may be any element of group 0 the index of which is greater than 2) as shown in ⓪ of Figure 4. After the assignment in line 2, the analysis infers that `parent` is also an index in group 0, since $\mathcal{C}$ entails that $\widehat{\texttt{mparent}}_0 \triangleleft G_0$ (the parent of any valid process is also a valid process). The resulting abstract state is shown in ①. The analysis of an update to the array is more complex. Indeed, at line 4, array cell `mproc[child]` is modified, and while this cell is known to belong to group G_0, this group may have several elements (it has at least one element since $\texttt{child} \triangleleft G_0$, thus $\underline{\text{Sz}}_0 \geq 1$). Therefore, and in order to perform a *strong update*, our analysis first *materializes* the array element that is being modified, by splitting group 0 into two groups, labeled 0 and 2, where group 2 has *exactly one element*, corresponding to `mproc[child]` (which is also expressed by $\texttt{child} \triangleleft G_2$). Both groups inherit predicates from former group 0. Since group 2 has a single element ($\underline{\text{Sz}}_2 = 1$) which corresponds exactly to the modified cell, the analysis can perform a strong update at this stage, and it generates the abstract state ②. *Materialization* is a common technique in static analysis. However, in our case, this technique does not require the analysis to make case splits and to use a disjunction abstract domain, as is often the case, e.g., in shape analysis [8, 6]. This operation will be described in detail in Section 6.

The analysis of all the atomic statements in the program follows similar principles. We now discuss the termination of the loop analysis. Since our abstract domain has infinite chains (the number of groups is not bounded), and to ensure the termination of the analysis of loops, we need to use a terminating widening operator [10]. The widening operator of our array domain associates groups with similar properties from its two input abstract states (even if they disagree on the number of groups) by a heuristic group matching scheme, and it over-approximates the predicates attached to them. Termination is guaranteed by ensuring that the number of groups can only decrease. In this example, the post-fixpoint ③ is obtained after two widening iterations. In the post-fixpoint groups 0 (resp., 3) represent valid process descriptors with indexes greater (resp., lower) than `i`, group 1 describes the free slots, and group 2 consists of `mproc[child]` (the process that was just cleaned up).

Compared to other array analyses, indexes of array cells play a less important role in our analysis, since localization and materialization of cells are based on the groups, and thus on all the relations over group fields and group membership. However, information about the relations between loop indexes and groups matters greatly. For instance, the loop abstract post-fixpoint ③ shows that indexes of elements in group 0 are

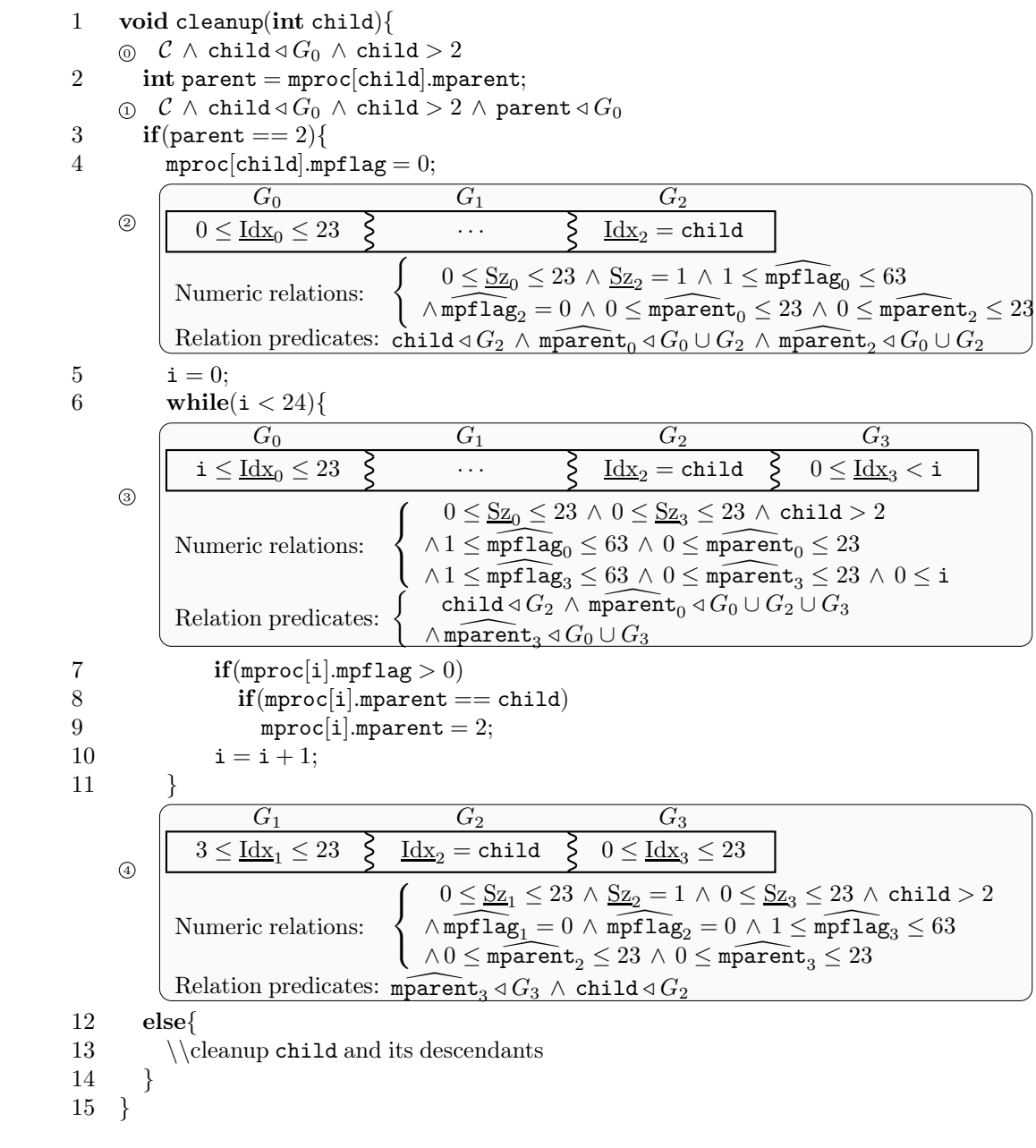


Figure 4: Overview of the analysis of `cleanup`

greater than `i`. Thus, after the loop exit, any element of group 0 should have an index greater than 24, which implies this group is empty at that point. Hence, this group can be *pruned out*, and the analysis produces post-condition ④ after the loop. This example shows that our analysis can reduce the number of groups, when some become redundant (e.g., when it proves a group empty).

Post-condition ④ entails that correctness condition $\mathcal{C}$ holds at the end of the execution of `cleanup` (note that group 2, corresponding to `child` now describes a free slot). Combining the fact that the post-condition of the **else** branch (line 12) also entails correctness condition $\mathcal{C}$ (the analysis process is not shown here, as it is mostly similar), our analysis verifies that correctness condition $\mathcal{C}$ always holds when function `cleanup` returns.

$\mathbb{V}$: base type variables ($\mathtt{v} \in \mathbb{V}$) $\quad$ $\mathbb{F}$: fields ($\mathtt{f} \in \mathbb{F}$)
$\mathbb{A}$: structural type variables ($\mathtt{a} \in \mathbb{A}$) $\quad$ $\mathbb{V}\mathbb{o}\mathbb{l}$: values ($\mathtt{c} \in \mathbb{V}\mathbb{o}\mathbb{l}$)
$\mathbb{N}$: Non-negative integers ($k \in \mathbb{N}$)

(a) Notations

$\oplus$	::=	$+ \mid - \mid * \mid \ldots$	
$\otimes$	::=	$< \mid \leq \mid > \mid \geq \mid == \mid != \mid \ldots$	
B	::=	**Bool** | **Int** | **Float**	base types
T	::=	**struct**{B f;...;B f}[k]	structural types
lv	::=	a[v].f | v	left value expressions
ex	::=	lv | c | ex $\oplus$ ex | ex $\otimes$ ex	right value expressions
cond	::=	TRUE | FALSE | ex $\otimes$ ex	conditions
s	::=	skip | B v | T a | lv = ex | s; s	statements
		| **if**(cond){s}**else**{s} | **while**(cond){s}	

(b) Syntax

Evaluation of L-values: $\mathbb{S} \to (\mathbb{A} \times \mathbb{N} \times \mathbb{F} \cup \mathbb{V})$

$$[\![\mathtt{v}]\!](\sigma) = \mathtt{v} \qquad [\![\mathtt{a}[\mathtt{v}].\mathtt{f}]\!](\sigma) = (\mathtt{a}, \sigma(\mathtt{v}), \mathtt{f})$$

Evaluation of Expressions: $\mathbb{S} \to \mathbb{V}\mathbb{o}\mathbb{l}$

$$[\![\mathtt{v}]\!](\sigma) = \sigma(\mathtt{v}) \qquad [\![\mathtt{a}[\mathtt{v}].\mathtt{f}]\!](\sigma) = \sigma(\mathtt{a}, [\![\mathtt{v}]\!](\sigma), \mathtt{f}) \qquad [\![\mathtt{c}]\!](\sigma) = \mathtt{c}$$
$$[\![\mathtt{ex} \oplus \mathtt{ex}]\!](\sigma) = [\![\mathtt{ex}]\!](\sigma) \oplus [\![\mathtt{ex}]\!](\sigma) \qquad [\![\mathtt{ex} \otimes \mathtt{ex}]\!](\sigma) = [\![\mathtt{ex}]\!](\sigma) \otimes [\![\mathtt{ex}]\!](\sigma)$$

Condition tests: $\mathcal{P}(\mathbb{S}) \to \mathcal{P}(\mathbb{S})$

$$[\![\mathtt{TRUE}]\!](S) = S \qquad [\![\mathtt{FALSE}]\!](S) = \emptyset \qquad [\![\mathtt{ex} \otimes \mathtt{ex}]\!](S) = \{\sigma \in S \mid [\![\mathtt{ex} \otimes \mathtt{ex}]\!](\sigma) = \mathtt{TRUE}\}$$

Transformers: $\mathcal{P}(\mathbb{S}) \to \mathcal{P}(\mathbb{S})$

$$\begin{aligned}
[\![\mathtt{skip}]\!](S) &= S \\
[\![\mathtt{lv} = \mathtt{ex}]\!](S) &= \{\sigma[[\![\mathtt{lv}]\!](\sigma) \mapsto [\![\mathtt{ex}]\!](\sigma)] \mid \sigma \in S\} \\
[\![\mathtt{s}_1; \mathtt{s}_2]\!](S) &= [\![\mathtt{s}_2]\!] \circ [\![\mathtt{s}_1]\!](S) \\
[\![\mathbf{if}(\mathtt{cond})\{\mathtt{s}_1\}\mathbf{else}\{\mathtt{s}_2\}]\!](S) &= [\![\mathtt{s}_1]\!] \circ [\![\mathtt{cond}]\!](S) \cup [\![\mathtt{s}_2]\!] \circ [\![\mathtt{cond} == \mathtt{FALSE}]\!](S) \\
[\![\mathbf{while}(\mathtt{cond})\{\mathtt{s}\}]\!](S) &= [\![\mathbf{while}(\mathtt{cond})\{\mathtt{s}\}]\!] \circ [\![\mathtt{s}]\!] \circ [\![\mathtt{cond}]\!](S) \cup [\![\mathtt{cond} == \mathtt{FALSE}]\!](S)
\end{aligned}$$

(c) Semantics

Figure 5: A basic imperative language with arrays of structures

3. Language and concrete states

In this section, we fix notations and set up a simple array imperative language that we are going to use in order to formalize the analysis. The syntax of this language is shown in Figure 5(b).

Syntax. We let $\mathbb{N}$ denote the set of non-negative integers and $\mathbb{F}$ denote the set of fields. Our language allows two kinds of types. Base types B include boolean, floating point and integer. Structural types T describe arrays of structures. Variables of types B and T are denoted by $\mathbb{V}$ and $\mathbb{A}$ respectively. This language also allows variables of structure type (they are considered arrays of length 1), and arrays of base type values (they are arrays of structures made of a single field).

We restrict the form of array cell accesses (to read or write a value) to expressions of the form a[v], where v is a variable, which will simplify both the semantics and the definition of the analysis (more complex array accesses can be decomposed into expressions of this form using auxiliary variables). However, we do not consider array

accesses through pointer dereference (analyzing such expressions would merely require extending our analysis by taking a product with a pointer domain). These restrictions allow to streamline the language under consideration around the purpose of our analysis, namely, to deal with arrays of complex data structures. Statements `s` comprise skip, variables declarations, assignments and control structures (condition tests, loops, sequences).

Concrete states and semantics. A concrete state σ is a partial function mapping basic cells (base variables and fields of array cells) into values (which are denoted by $\mathbb{Val}$). The set $\mathbb{S}$ of concrete states is defined by $\sigma \in \mathbb{S} = (\mathbb{A} \times \mathbb{N} \times \mathbb{F} \cup \mathbb{V}) \to \mathbb{Val}$. More specifically, the set of all fields of cells of array `a` is denoted by $\mathbb{F}_{\mathtt{a}}$, and the set of valid indexes in `a` is denoted by $\mathbb{N}_{\mathtt{a}}$.

Figure 5(c) defines the concrete collecting semantics of expressions, l-values, condition tests and statements. The semantics $[\![\mathtt{ex}]\!]$ of expression `ex` maps a state into the value `ex` evaluates to. The semantics $[\![\mathtt{lv}]\!]$ of l-value `lv` maps a state into the cell `lv` evaluates to. The semantics $[\![\mathtt{cond}]\!]$ of condition `cond` filters out the states in which `cond` does not evaluate to `TRUE`. The semantics $[\![\mathtt{s}]\!]$ of statement `s` is a denotational semantics [28] that maps a set of pre-states S into the set of states that can be reached by executing `s` from a state in S. This semantics does not explicitly characterize the non-terminating executions, however it could trivially be extended into a semantics that collects the set of *all* reachable states. (we make the choice to use an "angelic" denotational semantics so as to make the formalization of our analysis simpler).

4. Abstract domain and abstraction relation

In this section, we formalize abstract elements and their concretization. We describe the abstraction of the contents of arrays, using numeric constraints in Section 4.1. Then, we extend it with relation predicates between groups in Section 4.2.

4.1. The non-contiguous array partition

Non-contiguous array partition. Our analysis partitions each array into *groups* of cells which are not necessarily contiguous. A *group* denotes an abstraction for the set of cells it is composed of. It is denoted by a name G_i, where subscript i identifies the group. We let $\mathbb{G}$ denote the set of group names $\{G_i \mid i \geq 0\}$.

Definition 1 (Array partition). An *array partition* is a function which maps each array variable to a set of groups.

$$p : \mathbb{A} \to \mathcal{P}(\mathbb{G}) \quad \text{where} \quad \forall a_0, a_1 \in \mathbb{A},\ a_0 \neq a_1 \Longrightarrow p(a_0) \cap p(a_1) = \emptyset$$

Moreover, our abstract domain will always give different names to groups denoting cells of distinct arrays, so as to avoid confusion. The denotation of groups is defined through mappings of group names into the set of concrete indexes they denote. Such mappings are called *valuations*:

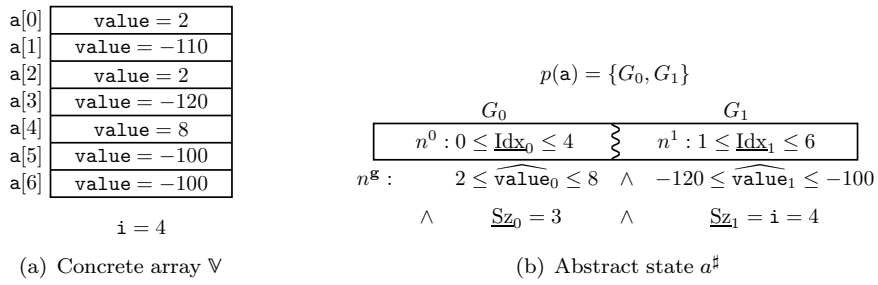
Definition 2 (Valuation). A *valuation* is a function $\psi \in \Psi = \mathbb{G} \to \mathcal{P}(\mathbb{N})$, and interprets each group into the set of indexes it represents in a given concrete state.

Figure 6(a) displays a concrete state, with an array of integers `a` of length 7 (each cell is viewed as a structure with a single field `value`). Figure 6(b) shows an abstraction with partition $p(\mathtt{a}) = \{G_0, G_1\}$. A possible valuation is ψ, defined by $\psi(G_0) = \{0, 2, 4\}$ and $\psi(G_1) = \{1, 3, 5, 6\}$.

Numerical properties. To express numerical properties of group contents, sizes, and indexes, we adjoin numeric abstract elements to partition p. The numeric abstract elements are from numeric domains with summarized dimensions [17], which hold two kinds of dimensions: non-summary dimensions are dimensions that account for exactly one concrete memory cell whereas summary dimensions represent for potentially unbounded collection of concrete memory cells.

The numerical information is a conjunction of (1) *global constraints* and of (2) *per group constraints*.

First, the *global component* $n^{\mathtt{g}}$ constrains base type variables group sizes and group fields. It consists of a numeric abstract element with the following dimensions:

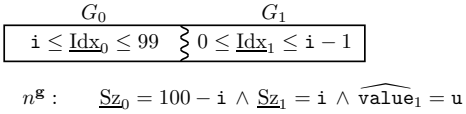


(a) Concrete array $\mathbb{V}$ (b) Abstract state $a^\sharp$

Figure 6: An abstraction in our domain

```
1  int a[100], u;
2  for(int i = 0; i < 100; i++){
3        a[i] = u;
   }
```

(a) Code



(b) Loop Invariant

Figure 7: An Array Initialization Example

- $\mathtt{v} \in \mathbb{V}$: each base type variable has a corresponding non-summary dimension in $n^{\mathbf{g}}$ (e.g., integer variable `i` in the example of Figure 6(b));
- $\underline{\mathrm{Sz}}_i$: the number of elements in group G_i is constrained by a non-summary dimension $\underline{\mathrm{Sz}}_i$ (e.g., $\underline{\mathrm{Sz}}_0$ in Figure 6(b));
- $\widehat{\mathtt{f}}_i$: for group G_i, its contents in filed `f` are summarized into a dimension $\widehat{\mathtt{f}}_i$ (e.g., $\widehat{\mathtt{value}}_0$ in Figure 6(b)).

Remember that all the dimensions listed above are constrained in a single numeric abstract element $n^{\mathbf{g}}$, this allows our analysis to capture numeric relations among indices, sizes and contents of different array-element groups and scalar variables. For instance, $\widehat{\mathtt{f}}_i \leq \widehat{\mathtt{f}}_j$ expresses that for each cell in group G_i, the value in field `f` is less or equal to that of any cell in G_j. Similarly, $\underline{\mathrm{Sz}}_i = \mathtt{v}$ states that the number of elements in group G_i is equal to the value of `v`. Figure 7(a) shows an array initialization example and figure 7(b) shows its loop invariant inferred by our analysis. All cells are partitioned into two groups: group G_0 for all uninitialized cells and group G_0 for those initialized. In $n^{\mathbf{g}}$, our analysis could express that the size of G_1 equals to `i` and the contents of all cells in G_1 equal to `u`.

Second, for each group G_i, the summarized index dimension $\underline{\mathrm{Idx}}_i$ and some base type variables (e.g., variables used to index array cells) are constrained by a *group specific* numeric abstract value n^i (e.g., n^0 in Figure 6(b)). Since group indexes are constrained in separate abstract numeric elements, the relational numeric constraints on group indexes are only those with base type variables.

For instance, in Figure 6(b), group G_0 (resp., G_1) comprises all the cells that store positive (resp., negative) values, the group specific numeric constraints reveal that all three positive values are stored in the first five cells of the array. In this example, we show explicitly n^0 and n^1. In the following, and unless a confusion is possible, we will represent numerical constraints all together, and not distinguish group specific and global constraints.

At this stage, we can introduce an abstract element as a partition together with a set of global and group specific numerical constraints:

Definition 3 (Abstract state). An abstract element $a^\sharp$ is a pair $(p, \overrightarrow{n})$ where $\overrightarrow{n}$ is a tuple $(n^{\mathbf{g}}, n^0, \ldots, n^{k-1})$, and p defines k array partitions.

Our domain needs to distinguish numeric constraints on group indices from those on group contents, because our domain allows empty groups and our algorithm reasons precisely on group indexes which may lead index dimensions to bottom. This is detailed in Example 1:

Example 1. *Let* $\mathtt{a}$ *be an array of length 4 and our abstraction partitions it into two groups* $p(\mathtt{a}) = \{G_0, G_1\}$. *When* $\psi(G_0) = \{0,1,2,3\}$ *and* $\psi(G_1) = \emptyset$, *the most precise numeric constraint on indexes of* G_1 *is* $\underline{\mathrm{Idx}}_1 = \bot$. *In any numerical domain where each dimension describes at least one concrete cell, and when a dimension describes an empty set of values, the whole abstract element can be reduced to* bottom, *meaning the whole abstract value describes the empty set of values. This would obviously not be acceptable in our case, which is why abstract predicates about group indexes have to be kept separate from the other predicates, and we have to exclude each* $\underline{\mathrm{Idx}}_i$ *from* $n^{\mathbf{g}}$. *Moreover, an empty group will be characterized by an empty set of indexes (so that the group specific information is bottom), as expected. Actually, this is a reduced cardinal power [11] operation on the global component and per group components.*

Group fields also correspond to summarized dimensions, but they do not need to be excluded from the global numeric component $n^{\mathbf{g}}$. The reason will be presented at the end of this section.

Concretization. A *concrete numeric mapping* is a function ν, mapping each base type variable to one value, each structure field to a non empty set of values and each index to a possibly empty set of values. We write $\gamma_{\mathbf{Num}}$ for the concretization of numeric elements, which maps a set of numeric constraints $\overrightarrow{n}$ into a set of functions ν as defined above. The concretization $\gamma_{\mathbf{Num}}(n^i)$ of constraints over group G_i is such that, when $n^i = \bot$ and $\nu \in \gamma_{\mathbf{Num}}(n^i)$, then $\nu(\underline{\mathrm{Idx}}_i) = \emptyset$. Then, $\gamma_{\mathbf{Num}}(n^{\mathbf{g}}, n^0, \ldots, n^{k-1}) = \gamma_{\mathbf{Num}}(n^{\mathbf{g}}) \cap \gamma_{\mathbf{Num}}(n^0) \ldots \gamma_{\mathbf{Num}}(n^{k-1})$.

Additionally, we use the following four predicates to break up the definition of concretization.

- Predicate $\mathrm{P_v}(\psi)$ states that each array element belongs to *exactly one group* (equivalently, groups form a partition of the array indexes);

$$\mathrm{P_v}(\psi) \overset{\text{def.}}{\iff} \forall \mathtt{a} \in \mathbb{A},\ \mathbb{N}_{\mathtt{a}} = \bigcup_{G_i \in p(\mathtt{a})} \psi(G_i) \wedge (\forall G_i, G_j \in p(\mathtt{a}), i \neq j \Rightarrow \psi(G_i) \cap \psi(G_j) = \emptyset)$$

- Predicate $\mathrm{P_b}(\sigma, \nu)$ expresses that ν and σ consistently abstract base type variables;

$$\mathrm{P_b}(\sigma, \nu) \overset{\text{def.}}{\iff} \forall \mathtt{v} \in \mathbb{V},\ \nu(\mathtt{v}) = \sigma(\mathtt{v})$$

- Predicate $\mathrm{P_i}(\nu, \psi)$ expresses that ν and ψ consistently abstract group indexes;

$$\mathrm{P_i}(\nu, \psi) \overset{\text{def.}}{\iff} \forall \mathtt{a} \in \mathbb{A},\ \forall G_i \in p(\mathtt{a}),\ \psi(G_i) = \nu(\underline{\mathrm{Idx}}_i) \wedge |\psi(G_i)| = \nu(\underline{\mathrm{Sz}}_i)$$

- Predicate $\mathrm{P_c}(\sigma, \psi, \nu)$ states σ and ν define compatible abstractions of groups contents.

$$\mathrm{P_c}(\sigma, \psi, \nu) \overset{\text{def.}}{\iff} \forall \mathtt{a} \in \mathbb{A},\ \forall \mathtt{f} \in \mathbb{F}_{\mathtt{a}},\ \forall G_i \in p(\mathtt{a}),\ \forall j \in \psi(G_i),\ \sigma(\mathtt{a}, j, \mathtt{f}) \in \nu(\widehat{\mathtt{f}}_i)$$

The concretization result is a set of tuples of (σ, ψ, ν) each of which respects all the four predicates above;

Definition 4 (Concretization). Concretization $\gamma_{\mathbf{Part}}$ is defined by:

$$\gamma_{\mathbf{Part}}(p, \overrightarrow{n}) \overset{\text{def.}}{::=} \{(\sigma, \psi, \nu) \mid \nu \in \gamma_{\mathbf{Num}}(\overrightarrow{n}) \wedge \mathrm{P_v}(\psi) \wedge \mathrm{P_b}(\sigma, \nu) \wedge \mathrm{P_i}(\nu, \psi) \wedge \mathrm{P_c}(\sigma, \psi, \nu)\}$$

Example 2. *The tuple defined below respects all the four predicates parameterized by* $(p, \overrightarrow{n})$ *in Fig 6(b), and implies the concrete state in Fig 6(a).*

$$\sigma: \begin{array}{lcl} (\mathtt{a}, 0, \mathtt{value}) & \mapsto & 2 \\ (\mathtt{a}, 1, \mathtt{value}) & \mapsto & -110 \\ (\mathtt{a}, 2, \mathtt{value}) & \mapsto & 2 \\ (\mathtt{a}, 3, \mathtt{value}) & \mapsto & -120 \\ (\mathtt{a}, 4, \mathtt{value}) & \mapsto & 8 \\ (\mathtt{a}, 5, \mathtt{value}) & \mapsto & -100 \\ (\mathtt{a}, 6, \mathtt{value}) & \mapsto & -100 \end{array} \qquad \psi: \begin{array}{lcl} G_0 & \mapsto & \{0,2,4\} \\ G_1 & \mapsto & \{1,3,5,6\} \end{array} \qquad \nu: \begin{array}{rcl} \underline{\mathrm{Idx}}_0 & \mapsto & \{0,2,4\} \\ \underline{\mathrm{Idx}}_1 & \mapsto & \{1,3,5,6\} \\ \underline{\mathrm{Sz}}_0 & \mapsto & 3 \\ \underline{\mathrm{Sz}}_1 & \mapsto & 4 \\ \widehat{\mathtt{value}}_0 & \mapsto & \{2,8\} \\ \widehat{\mathtt{value}}_1 & \mapsto & \{-100,-110,-120\} \end{array}$$

Our abstract domain is parameterized by the choice of a numeric abstract domain $\mathbb{N}^\sharp$, so as to tune the analysis precision and cost. In this paper, we use the octagons abstract domain [26] and polyhedra domain [9].

4.2. Relation predicates

The abstraction we have defined so far can describe non-contiguous groups of cells, yet lacks important predicates, that are necessary to ensure the analysis can be precise enough. Let us consider assignment `parent` = `mproc`[`child`].`mparent` in `cleanup` (Figure 3(a)). To deduce that `parent` is the index of a valid process descriptor, our analysis should track the facts that (1) `child` is the index of a valid process descriptor; (2) for each valid process descriptor, its parent is the index of a valid process descriptor. The numerical predicates that are available to the analysis do not entail these two facts. Instead, they only imply that `child` and $\widehat{\texttt{mparent}}_0$ are in range $[0, 23]$, hence are valid array cells. These numeric properties do not characterize the group that cells belong to, hence, that the new value of `parent` is an index of a valid process descriptor. To avoid such an imprecision, we extend abstract states with *relation predicates*, that express membership properties:

Definition 5 (Relation predicates).

r	::=	$r \wedge r$		a conjunction of predicates
	|	**true**		empty
	|	$\texttt{v} \triangleleft G^{\texttt{a}}$	where $\texttt{v} \in \mathbb{V}$	var-index predicate
	|	$\widehat{\texttt{f}}_i \triangleleft G^{\texttt{a}}$	where $\texttt{f} \in \mathbb{F}_{\texttt{a}}, G_i \in p(\texttt{a})$	content-index predicate
$G^{\texttt{a}}$	::=	$G^{\texttt{a}} \cup G_i$	where $G_i \in p(\texttt{a})$	a disjunction of groups in `a`
	|	G_i	where $G_i \in p(\texttt{a})$	

A relation predicate r is a conjunction of atomic predicates. Predicate $\texttt{v} \triangleleft G^{\texttt{a}}$ means the value of variable `v` is an index in G^a, where G^a is a disjunction of a set of groups of array `a`. Similarly, predicate $\widehat{\texttt{f}}_i \triangleleft G^{\texttt{a}}$ means that all fields of cells in group i are indexes of elements of $G^{\texttt{a}}$. As an example, if $G^{\texttt{a}} = G_1 \cup G_3$, then $\texttt{v} \triangleleft G^{\texttt{a}}$ expresses that the value of `v` is either the index of a cell in G_1 or the index of a cell in G_3.

Example 3. *We consider function* `cleanup` *of Figure 3(a). The pre-condition for the analysis of Figure 4 is based on correctness property $\mathcal{C}$, hence partitions* `mproc` *in two groups, thus* $p(\texttt{mproc}) = \{G_0, G_1\}$*, in which G_0 accounts for all valid process descriptors and G_1 for all free slots. Additionally,* `cleanup` *should be called on a valid process descriptor, hence* `child` *should be in group G_0, which corresponds to predicate* $\texttt{child} \triangleleft G_0$*. Combining this with predicate* $\widehat{\texttt{mparent}}_0 \triangleleft G_0$ *and the fact that* `parent` *is initialized as the parent of* `child`*, the analysis derives* $\texttt{parent} \triangleleft G_0$ *(i.e.,* `parent` *is a valid process descriptor index). Hence, at point* ①*, the analysis will derive relations* $r = \texttt{child} \triangleleft G_0 \wedge \texttt{parent} \triangleleft G_0 \wedge \ldots$.

We extend the abstract states set up in Definition 3 with a third component that describes group relations:

Definition 6 (Abstract state). An abstract state consists of a partition, a set of numeric constraints and a set of group relation predicates:

$$a^\sharp \in \mathbb{D}^\sharp = \{(p, \overrightarrow{n}, r)\}$$

Concretization. We now extend the concretization to account for relation predicates. As a slight abuse of notation, we extend ψ on disjunctions of groups, and let $\psi(G_0 \cup \ldots \cup G_i) = \psi(G_0) \cup \ldots \cup \psi(G_i)$.

Definition 7 (Concretization). The concretization function $\gamma_{\mathbf{Gr}}$ for relation predicates maps an abstract state into a set of triples (σ, ψ, ν) made of a concrete state, a valuation and a concrete numeric mapping; it is defined by induction on group relation predicates:

$$\begin{array}{rcl}
\gamma_{\mathbf{Gr}}(p, \overrightarrow{n}, \textbf{true}) & ::= & \gamma_{\mathbf{Part}}(p, \overrightarrow{n}) \\
\gamma_{\mathbf{Gr}}(p, \overrightarrow{n}, \texttt{v} \triangleleft G^{\texttt{a}}) & ::= & \{(\sigma, \psi, \nu) \in \gamma_{\mathbf{Part}}(p, \overrightarrow{n}) \mid \sigma(\texttt{v}) \in \psi(G^{\texttt{a}})\} \\
\gamma_{\mathbf{Gr}}(p, \overrightarrow{n}, \widehat{\texttt{f}}_i \triangleleft G^{\texttt{a}}) & ::= & \{(\sigma, \psi, \nu) \in \gamma_{\mathbf{Part}}(p, \overrightarrow{n}) \mid \forall k \in \psi(G_i),\ \sigma(\texttt{a}, k, \texttt{f}) \in \psi(G^{\texttt{a}})\} \\
\gamma_{\mathbf{Gr}}(p, \overrightarrow{n}, r_0 \wedge r_1) & ::= & \gamma_{\mathbf{Gr}}(p, \overrightarrow{n}, r_0) \cap \gamma_{\mathbf{Gr}}(p, \overrightarrow{n}, r_1)
\end{array}$$

The *concretization* $\gamma_{\mathbf{St}}$ maps an abstract state into a set of concrete states and is defined by:

$$\gamma_{\mathbf{St}}(p, \overrightarrow{n}, r) ::= \{\sigma \mid \exists \psi, \nu,\ (\sigma, \psi, \nu) \in \gamma_{\mathbf{Gr}}(p, \overrightarrow{n}, r)\}$$

Intuitively, when the relation predicate is **true**, the abstract state has the same meaning as in Definition 4. When the relation predicate is $\mathtt{v} \triangleleft G^{\mathtt{a}}$, concretization $\gamma_{\mathbf{Gr}}$ filters out 3-tuples (σ, ψ, ν) where the value of $\mathtt{v}$ is not an index of a group in $G^{\mathtt{a}}$. When the group predicate is $\widehat{\mathtt{f}_i} \triangleleft G^{\mathtt{a}}$, it filters out the 3-tuples (σ, ψ, ν) where the value of the field $\mathtt{f}$ of each cell of G_i is not an index of a group in $G^{\mathtt{a}}$. Finally, conjunctions of group predicates are mapped into the intersection of their concretizations.

The concretization of an abstract state collects all the concrete states that can be described by the abstract state for some valuation and concrete numeric mapping.

Example 4. *We propose to consider the abstract state obtained by adding relation predicates* $r = \mathtt{i} \triangleleft G_0 \wedge \widehat{\mathtt{value}_0} \triangleleft G_0$ *to the abstract state of Figure 6(b). Then, the concrete state shown in Figure 6(a) is not in the concretization* $\gamma_{\mathbf{St}}(a^\sharp)$ *anymore. Indeed,* $\mathtt{a}[4] = 8$ *violates the group predicate* $\widehat{\mathtt{value}_0} \triangleleft G_0$.

This example demonstrates how relation predicates r refine abstract states, and helps reasoning precisely on statements such as $\mathtt{a}[\mathtt{i}]$, even $\mathtt{a}[\mathtt{a}[\mathtt{i}]]$.

Abstract order. We let relation $\leq$ be defined by:

$$a_1^\sharp \leq a_2^\sharp \iff \gamma_{\mathbf{St}}(a_1^\sharp) \subseteq \gamma_{\mathbf{St}}(a_2^\sharp)$$

We also define that $a_1^\sharp \leq a_2^\sharp \wedge a_2^\sharp \leq a_1^\sharp \iff a_1^\sharp = a_2^\sharp$. It is easy to observe that this relation defines a partial order over abstract states.

Group fields and indexes both correspond to summarized dimensions, but only group indexes need to be excluded from the global numeric component $n^{\mathtt{g}}$. That is because our static analysis algorithms (Section 5, 6 and 7) treat them differently. When a group is empty, the set of values for its index dimension could be described in the abstract either as the empty set or by a more conservative over-approximation. However, our analysis will always over-approximate the values its other fields may take (which implies the corresponding dimensions will not carry precise information). This is necessary since our numerical abstract domain allows one abstract dimension to stand for either one concrete cell (in the case of a non summary dimension) or for a strictly positive number of concrete cells (in the case of a summary dimension), henceforth, it is not able to describe precisely states where one abstract dimension corresponds to *zero* concrete cell. As a consequence, when a group becomes empty, our analysis will always consider each of its fields still corresponds to a non-empty set of concrete cells, and will attach to the fields of such a group the same abstract predicates as it would if the group were not empty.

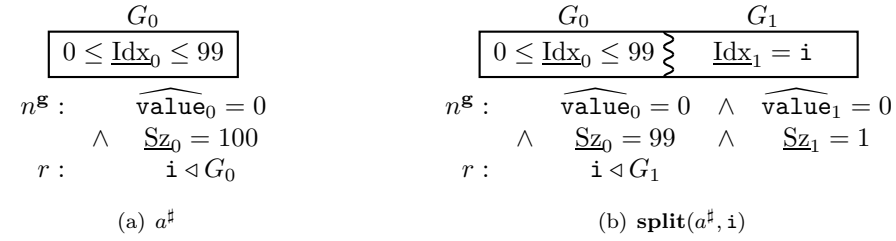
5. Basic operators on partitions

In this section, we define a set of basic operations on partitions, that abstract transfer functions and lattice operators will call to modify the structure of partitions.

Splitting. Unless it is provided with a pre-condition that specifies otherwise, our analysis initially partitions each array into a single group, with unconstrained contents. Additional groups can get introduced during the analysis, by a basic operator **split**.

Operator **split** applies to an abstract state $a^\sharp$, an array $\mathtt{t}$ and a group G_i corresponding to array $\mathtt{t}$ and replaces it with two groups G_i, G_j (where G_j is a fresh group name). The two new groups inherit the properties of the group they replace (membership in the old group turns into membership in the union of the two new groups). Assuming that $a^\sharp = (p, \overrightarrow{n}, r)$, and with the above notations, **split** performs the following actions:

- It extends partition p with the fresh group name G_j;
- The numeric constraints on indexes and fields of group G_j are inherited from those of G_i, and every occurrence of $\underline{\mathrm{Sz}}_i$ is replaced by $\underline{\mathrm{Sz}}_i + \underline{\mathrm{Sz}}_j$;
- The relation predicates on G_j are inherited from those on G_i.



(a) $a^\sharp$ (b) $\mathbf{split}(a^\sharp, \mathtt{i})$

Figure 8: Partition splitting in array `a` from abstract state $a^\sharp$

In practice, the analysis often needs to use **split** in order to precisely handle an update into an array, or the reading of a value in an array. Therefore, we extend **split** so that it can also be applied to an abstract state $a^\sharp$, an array `t` and a variable `v` known to store a valid index in `t`, and splits `t` so as to materialize the cell pointed to by `v`. This can only be done when the value of `v` can be tracked as an element of a specific group of `t`; operator **split** then splits this group into a group of one element, of index `v` and another group. This scheme will allow *strong updates* into the array.

Example 5. *Figure 8(a) defines an abstract state $(p, \overrightarrow{n}, r)$ with a single array, fully initialized to 0, and represented by a single group. Applying operator* **split** *to that abstract state and to index* `i` *produces the abstract state of Figure 8(b), where G_1 is a group with exactly one element, with the same constraints on $\widehat{\mathtt{value}}$ as in the previous state.*

Theorem 1 (Soundness of split). *Suppose $a^\sharp$ is an abstract state, G_i a group and* `t` *an array, the operator* **split** *is sound in the sense that*

$$\gamma_{\mathbf{St}}(a^\sharp) \subseteq \gamma_{\mathbf{St}}(\mathbf{split}(a^\sharp, \mathtt{t}, G_i))$$

Proof. Let $a^\sharp = (p, \overrightarrow{n}, r)$ be an abstract state, and G_i be a group of `a` in $a^\sharp$. We assume that splitting G_i in $a^\sharp$ produces groups $G_{i'}, G_k$ ($G_{i'}$ is actually G_i in the output, we add a superscript to distinguish it from the G_i in the input) in $a^{\sharp\prime}$ ($\mathbf{split}(a^\sharp, \mathtt{a}, G_i) = a^{\sharp\prime}$). Let $\sigma \in \gamma_{\mathbf{St}}(a^\sharp)$. We write ψ, ν for the witnesses of $\sigma \in \gamma_{\mathbf{St}}(a^\sharp)$ in Definition 3.

Then, we define ψ', ν' from ψ, ν by:

- fixing $\psi'(G_{i'})$ and $\psi'(G_k)$ so that $\psi'(G_{i'}) \cup \psi'(G_k) = \psi(G_i)$;
- adding new dimensions (fields, size, index) for groups $G_{i'}, G_k$, that inherit from the values of the dimensions corresponding to G_i:
  - ν' maps $\underline{\mathrm{Idx}}_{i'}$ and $\underline{\mathrm{Idx}}_k$ to the respective sets of elements of $G_{i'}$ and G_k;
  - ν' maps $\underline{\mathrm{Sz}}_{i'}$ and $\underline{\mathrm{Sz}}_k$ to the respective sizes of $G_{i'}$ and G_k;
  - other dimensions take the same value as in ν.

Then, since the relation predicates on $G_{i'}$ and G_k inherit those on G_i, to prove that ψ', ν' are witnesses of $\sigma \in \gamma_{\mathbf{St}}(a^{\sharp\prime})$, we simply need to establish one by one conditions $\mathrm{P_v}, \mathrm{P_b}, \mathrm{P_i}$ and $\mathrm{P_c}$.

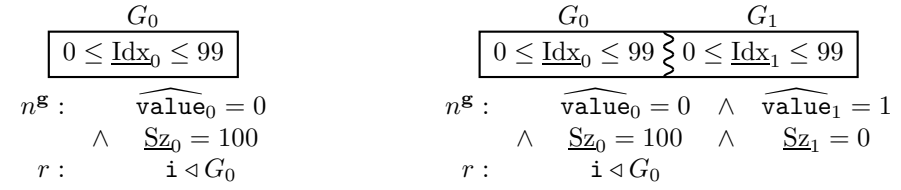
This proves the soundness of **split**.

This operator may lose a little precision on the sizes of the generated groups when the underlying numeric domain is not complete on linear assignments.

Creation. Operator **create** extends the partition of an existing array with a new, empty group. It is used by join and widening, so as to generalize abstract states. By nature, an empty group satisfies any field property, thus the analysis may assign any numeric property to the fields of the new group, depending on the context.

When applied to abstract state $a^\sharp = (p, \overrightarrow{n}, r)$ and to array variable `a`, operator **create** performs the following operations:

- It introduces a fresh group name G_j to the partition $p(\mathtt{a})$ of array `a`;



(a) $a^\sharp$ (b) $\mathbf{create}(a^\sharp)$

Figure 9: Partition creation in array `a` from abstract state $a^\sharp$

- The size constraint $\underline{\text{Sz}}_j = 0$ is added to $\overrightarrow{n}$;
- Additional constraints on the index and the fields of group G_j are added to $\overrightarrow{n}$;
- For each group $G_i \in p(\texttt{a})$ and each $\texttt{f} \in \mathbb{F}_{\texttt{a}}$, content-index predicate $\widehat{\texttt{f}}_j \triangleleft G_i$ is added to r.

Example 6. *Figure 9(a) defines an abstract state $(p, \overrightarrow{n}, r)$ with a single array, fully initialized to 0, and represented by a single group. Similarly, Figure 9(b) shows a possible result for* **create**.

Theorem 2 (Soundness of create). *Operator* **create** *is sound in the sense that, for all abstract state $a^\sharp$, for all array variable* `t` *and for all G_i,*

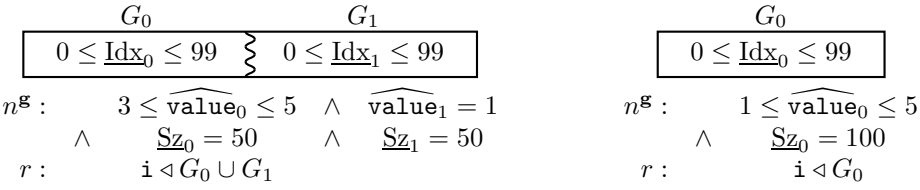
$$\gamma_{\mathbf{St}}(\mathbf{create}(a^\sharp, \texttt{t})) = \gamma_{\mathbf{St}}(a^\sharp)$$

Proof. In the new group G_i created by operator **create**, the predicate $\underline{\text{Sz}}_i = 0$ guarantees that $|\psi(G_i)| = \nu(\underline{\text{Sz}}_i) = 0$ which indicates that the addition of the new group does not affect the concretization.

Merging groups. Fine-grained abstract states, with many groups can express precisely complex properties, yet may incur increased analysis cost. In fact, the basic operators shown so far only add new groups, and removing groups may be required, at least for the sake of termination. Therefore, the analysis needs to *merge* distinct groups. This merge operator occurs as part of join, widening or when other transfer functions detect distinct groups of a same array enjoy similar properties. Operator **merge** takes an abstract state $a^\sharp = (p, \overrightarrow{n}, r)$, an array `t` and a set of groups S of array `t` as arguments and replaces all the groups of that set by a single group. For the sake of simplicity, we describe the operations performed when S has two elements G_j, G_k (the case of a set of more than two elements is similar):

- It creates a fresh group name G_i and adds it to partition p;
- The numeric constraints on indexes and fields of G_i over-approximate those on G_k and G_j; group size $\underline{\text{Sz}}_i$ is assigned with $\underline{\text{Sz}}_k + \underline{\text{Sz}}_j$ in $\overrightarrow{n}$;
- The relation predicates on G_i over-approximate those on G_k and G_j in r (namely any field that is known to be an element of G_j or G_k is then known to be an element of G_i);
- It removes group names G_j and G_k from partition p;
- It renames G_i to G_j or G_k, the choice depends on which group may originally consist of more cells.

Example 7. *Figure 10(a) defines an abstract state $a^\sharp$ which describes an array with two groups. Applying* **merge** *to $a^\sharp$ and set $\{0, 1\}$ produces the state shown in Figure 10(b), with a single group and coarser predicates, obtained by joining the constraints over the contents of the initial groups.*



(a) $a^\sharp$ (b) $\mathbf{merge}(a^\sharp, \{0, 1\})$

Figure 10: Merging in abstract state $a^\sharp$

Theorem 3 (Soundness of merge). *Operator* **merge** *is sound in the following sense: For all abstract state* $a^\sharp$*, array variable* $\mathtt{t}$*, and two groups* G_k *and* G_j *in array* $\mathtt{t}$*,*

$$\gamma_{\mathbf{St}}(a^\sharp) \subseteq \gamma_{\mathbf{St}}(\mathbf{merge}(a^\sharp, \mathtt{t}, \{G_k, G_j\}))$$

Proof. Let $a^\sharp = (p, \overrightarrow{n}, r)$ be an abstract state, and two groups G_k and G_j in p. We assume that applying **merge** on G_k and G_j in $a^\sharp$ produces group G_i (in the algorithm of **merge**, G_i will be renamed to G_j or G_k finally, but the renaming does not affect the concretization) in abstract state $a^{\sharp\prime} = (p', \overrightarrow{n}', r')$.

Let $\sigma \in \gamma_{\mathbf{St}}(a^\sharp)$. We write ψ, ν for the witnesses of $\sigma \in \gamma_{\mathbf{St}}(a^\sharp)$ in Definition 3. We now show that σ is also in the concretization of $a^{\sharp\prime}$, by constructing a witnesses ψ', ν':

1. fixing $\psi'(G_i)$ so that $\psi'(G_i) = \psi(G_j) \cup \psi(G_k)$;
2. adding new dimensions (fields, size, index) for group G_i, that inherit from the values of the dimensions corresponding to G_j, G_k (we recall some of these dimensions are *summary dimensions* in the numerical domain):
   (a) $\nu(\underline{\mathrm{Idx}}_i) = \nu(\underline{\mathrm{Idx}}_j) \cup \nu(\underline{\mathrm{Idx}}_k)$ (summary dimension);
   (b) $\nu(\underline{\mathrm{Sz}}_i) = \nu(\underline{\mathrm{Sz}}_j) + \nu(\underline{\mathrm{Sz}}_k)$;
   (c) similarly to $\underline{\mathrm{Idx}}$, fields are summary dimensions and defined by $\nu(\widehat{\mathtt{f}}_i) = \nu(\widehat{\mathtt{f}}_j) \cup \nu(\widehat{\mathtt{f}}_k)$;
3. removing all dimensions corresponding to G_j, G_k.

Then, since the relation predicates on G_i over-approximate those on G_k and G_j in r, to prove that ψ', ν' are witnesses of $\sigma \in \gamma_{\mathbf{St}}(a^{\sharp\prime})$, we simply need to establish one by one conditions $\mathrm{P_v}, \mathrm{P_b}, \mathrm{P_i}$ and $\mathrm{P_c}$.

This proves the soundness of **merge**.

The precision loss in merging depends on the similarity of the groups being merged. Our analysis loses no precision when the merged groups are exactly the same.

Reduction. Our abstract domain can be viewed as a product abstraction and can benefit from *reduction* [11]. If we consider abstract state $a^\sharp = (p, \overrightarrow{n}, r)$, components $\overrightarrow{n}$ and r may allow to refine each other. Such steps are performed by a *partial reduction* operator **reduce**, which strengthens the numeric and relation predicates, without changing the global concretization [11]. The operations of **reduce** are based on the numeric implications of relation predicates. It consists of two directions:

- from r to $\overrightarrow{n}$: relation predicates always imply numeric constraints over the size and indexes of array groups, e.g., if $\mathtt{v} \triangleleft G_i$, then group G_i has at least one element ($\underline{\mathrm{Sz}}_i \ge 1$), and if $\underline{\mathrm{Idx}}_i < 5$, then $\mathtt{v} < 5$;
- from $\overrightarrow{n}$ to r: more precise relation predicates can be inferred from the numeric relations between variables and summarized group indexes, e.g., if $\mathtt{v} < \underline{\mathrm{Idx}}_i$, then **reduce** removes G_i from $\mathtt{v} \triangleleft G_i \cup G_j$ in r.

Note that the optimal reduction would be overly costly to compute in general for an arbitrary numerical abstract domain. To avoid that, reduction is done lazily: for instance, the analysis will attempt to generate relations between $\mathtt{v}$ and $\underline{\mathrm{Idx}}_i$ only when $\mathtt{v}$ is used as an index to access the array G_i corresponds to.

Theorem 4 (Soundness of reduce). *Suppose $a^\sharp$ is an abstract state, operator* **reduce** *does not change concretization.*

$$\gamma_{\mathbf{St}}(\mathbf{reduce}(a^\sharp)) = \gamma_{\mathbf{St}}(a^\sharp)$$

PROOF. To establish the soundness of **reduce**, we simply need to consider each of the reduction cases mentioned above. We discuss only the first case, as the proof of the other cases is similar. We let $(p, \overrightarrow{n}, r)$ be an abstract state, such that $\mathtt{v} \triangleleft G_i$ appears in r. Then $\forall(\sigma, \psi, \nu) \in \gamma_{\mathbf{Gr}}(p, \overrightarrow{n}, r)$, we have $\sigma(\mathtt{v}) \in \psi(G_i)$, which implies $|\psi(G_i)| \geq 1$. Since $\nu(\underline{\mathrm{Sz}}_1) = |\psi(G_i)|$, it is sound to add constraint $\underline{\mathrm{Sz}}_i \geq 1$ to $\overrightarrow{n}$.

Principles of Partitioning. The basic operators on partitions are utilized by transfer functions and lattice operators to manipulate groups. The group modifications follow the principles listed below:

- No disjunctions are introduced: our analysis does not produce disjunctions even if it has to lose some precision;
- Groups with similar properties get merged: in most cases, only groups with similar properties are merged. Especially in join and widening, our analysis computes the similarities between groups and decides which groups to be merged;
- Assigments are based on strong updates: our analysis generates a group which contains only the cell being in assigned to allow strong update;
- The analysis strives to limit the number of groups: the analysis cost increases dramatically with the the number of groups. Therefore our analysis merges groups whenever merging is an option (e.g., in an assignment and when the group an array cell belongs to is not known, our analysis merges all possible groups instead of generating a disjunction; this helps keeping the number of groups reasonable).

6. Transfer functions

Our array static analysis performs a *forward abstract interpretation* [10]. In this section, we study the abstract transfer functions for tests (Section 6.1) and assignments (Section 6.2). Each transfer function should over-approximate the concrete effect of the corresponding program construction in the abstract domain.

6.1. Analysis of conditions

The concrete semantics of a condition `cond` is a function that inputs a set of states S and returns the subset of S in which `cond` evaluates to `TRUE`. Therefore, the abstract interpretation of a test from abstract state $a^\sharp = (p, \overrightarrow{n}, r)$ should narrow the set of concrete states described by $a^\sharp$ by filtering out states in which `cond` does not evaluate to `TRUE`. Intuitively, it proceeds by strengthening constraints in the numeric component $\overrightarrow{n}$, and propagating them into r thanks to **reduce**.

However, the application of test `cond` to numeric constraints $\overrightarrow{n}$ is not immediate, since the array cells that occur in `cond` do not necessarily correspond directly to dimensions in $\overrightarrow{n}$. As an example, let us consider condition test `a[i].f` $== 0$ in an abstract state where `a` is partitioned into two groups G_0, G_1 and where the only constraint available is $\mathtt{i} \triangleleft G_0 \cup G_1$: then, the group array cell `a[i]` belongs to cannot be identified without ambiguity. Moreover, each group may contain several elements, and its `f` field may be described by a summary variable. Therefore, our analysis cannot refine `a[i].f`.

To derive a precise post-condition, our analysis relies on a *local disjunction* such that each case covers a group the index may belong to, and allows for a more precise test. In the above example, the analysis will analyze test `a[i].f` $== 0$ in a disjunction of *two* abstract states where the relation predicate is replaced by $\mathtt{i} \triangleleft G_0$ (resp., $\mathtt{i} \triangleleft G_1$). This process is called **enumerate**. The analysis then applies the numerical domain condition test operator to each disjunct. In this case, it will apply $\widehat{\mathtt{f}}_0 == 0$ to disjunct 0 and $\widehat{\mathtt{f}}_1 == 0$ to disjunct 1. Note that $\overrightarrow{n}$ may have summary dimensions (whenever a group describes more than a single array cell, its fields are summary dimensions), and that the actual condition test may not strengthen the constraints: for instance, if the size of group G_0 is not known to be exactly one, condition test $\widehat{\mathtt{f}}_0 == 0$ *will not* strengthen

$\mathbf{test}(\mathtt{cond}, (p, \overrightarrow{n}, r))\{$
$\quad \mathtt{cond}_{0\sim k}, r_{0\sim k} = \mathbf{enumerate}(\mathtt{cond}, r);$
$\quad \mathbf{foreach}(i \in \{0, 1, \ldots, k\})\{$
$\quad\quad \overrightarrow{n}_i = \mathbf{test_{Num}}(\mathtt{cond}_i, \overrightarrow{n}, r_i);$
$\quad\quad p, \overrightarrow{n}_i, r_i = \mathbf{reduce}(p, \overrightarrow{n}_i, r_i);$
$\quad \}$
$\quad (p, \overrightarrow{n}', r') = \mathbf{join}_{\equiv}((p, \overrightarrow{n}_0, r_0), (p, \overrightarrow{n}_1, r_1), \ldots, (p, \overrightarrow{n}_k, r_k));$
$\quad \mathbf{return}\ (p, \overrightarrow{n}', r');$
$\}$

Figure 11: The algorithm of the condition test transfer function

the constraints in $\overrightarrow{n}$. After the numerical condition test operator has been applied to all disjuncts, the analysis applies operator **reduce**, and merges all resulting disjuncts.

Note that the abstract test operator does not change the shape of the array partitions, thus, all the disjuncts generated by the above process can be merged by a trivial join operator, which simply over-approximates the properties for each group (a more general join operator, able to deal with abstract states with incompatible partitions will be presented in Section 7):

Definition 8 (Local disjunction join). Utilizing the join operator $\mathbf{join_{Num}}$ of the numeric domain with summarized domains [17], we define the *local disjunction join* operator $\mathbf{join}_{\equiv}$ as

$$\mathbf{join}_{\equiv}((p, \overrightarrow{n}_0, r_0), (p, \overrightarrow{n}_1, r_1)) = (p, \mathbf{join_{Num}}(\overrightarrow{n}_0, \overrightarrow{n}_1), r_0 \cap r_1)$$

Theorem 5 (Soundness of local disjunction join). *The local disjunction join operator* $\mathbf{join}_{\equiv}$ *is sound.*

$$\forall i \in \{0, 1\},\ \gamma_{\mathbf{St}}(p, \overrightarrow{n}_i, r_i) \subseteq \gamma_{\mathbf{St}}(\mathbf{join}_{\equiv}((p, \overrightarrow{n}_0, r_0), (p, \overrightarrow{n}_1, r_1)))$$

Algorithm. The algorithm of the abstract transfer function for condition tests is fully described in Figure 11. Operator $\mathbf{enumerate} : \mathbb{D}^\sharp \to \mathcal{P}(\mathbb{D}^\sharp)$ generates the set of every possible state in which each array cell in `cond` belongs to exactly one group. Namely, $\gamma_{\mathbf{St}}(a^\sharp) = \cup\{\gamma_{\mathbf{St}}(a_i^\sharp) \mid a_i^\sharp \in \mathbf{enumerate}(a^\sharp)\}$, and in any $a_i^\sharp$, and for any array cell `a`[`v`] in `cond`, the group that `a`[`v`] belongs to is deterministic. Then, condition test $\mathbf{test_{Num}}$ of the numeric domain with summarized domains [17] and the reduction operator are applied in each disjunctive state. All states are eventually joined together by the local disjunction join operator $\mathbf{join}_{\equiv}$.

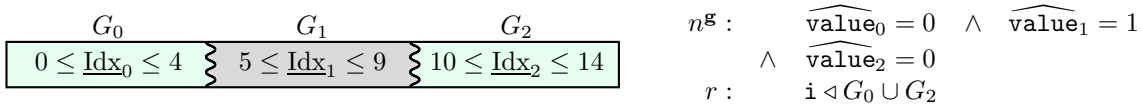
Example 8. *We consider the analysis of the code studied in Section 2. At the beginning of the iteration on fixpoint of the loop, from numeric constraints over* `i` *and group indexes,* `mproc`[`i`] *may be in* G_0, G_1 *or* G_2. *Then, the analysis of test* `mproc`[`i`].`mpflag` > 0 *will locally create three disjuncts corresponding to each of these groups. However, in the case of* G_1, $\widehat{\mathtt{mpflag}}_1 = 0$, *thus the numeric test* $\widehat{\mathtt{mpflag}}_1 > 0$ *will produce abstract value* $\bot$ *denoting the empty set of states. The same happens in the case of* G_2. *Therefore, only the third disjunct (the case corresponding to* G_0*) contributes to the abstract post-condition. Thus, the analysis derives* $\mathtt{i} \lhd G_0$.

Theorem 6 (Soundness). *The abstract transfer function* **test** *is* sound *in the sense that:*

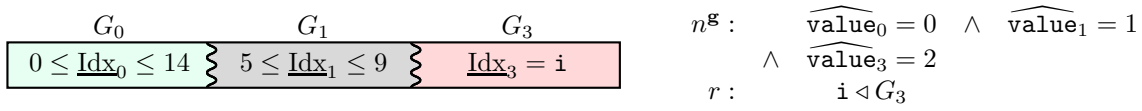
$$[\![\mathtt{cond}]\!]\gamma_{\mathbf{St}}(a^\sharp) \subseteq \gamma_{\mathbf{St}}(\mathbf{test}(\mathtt{cond}, a^\sharp))$$

Proof. The soundness of the condition test operator **test** follows from the fact that operators **enumerate** and **reduce** do not change the concretization, and from the soundness of the condition test operator $\mathbf{test_{Num}}$ of the numeric domain, and of the local disjunction join operator $\mathbf{join}_{\equiv}$.

The precision loss in condition test mainly comes from that in the condition test of numeric domains with summarized dimensions.



(a) Pre-condition



(b) Post-condition

Figure 12: The pre-and post-condition of assignment $\texttt{a}[\texttt{i}] = 2$

6.2. Assignment

Given l-value `lv` and expression `ex`, the concrete semantics of assignment $\texttt{lv} = \texttt{ex}$ writes the value of `ex` into the cell that `lv` evaluates to. On the abstract level, given abstract pre-condition $a^\sharp = (p, \overrightarrow{n}, r)$, an abstract post-condition for $\texttt{lv} = \texttt{ex}$ can be computed in three steps:

1. materialization of the memory cell that gets updated,
2. update of the numeric constraints in $\mathbb{N}^\sharp$ using $\mathbf{assign}_{\mathbf{Num}}$ [21], and update of the relation predicates, and
3. application of the reduction operator to the resulting abstract state.

In the following, and unless specified otherwise, we mainly focus on assignment that write on an array cells.

Materialization. When `lv` denotes an array cell, the analysis first *materializes* it into a group consisting of a single cell, so that strong updates can be carried out on $\overrightarrow{n}$ and r. To achieve this, the analysis computes which group(s) `lv` may evaluate into in abstract state $a^\sharp$. If there is a single such group G_i, that contains a single cell (i.e., $\underline{\text{Sz}}_i = 1$), then materialization is already achieved. If there is a single such group G_i, and $\underline{\text{Sz}}_i$ is greater than 1, then the analysis calls **split** in order to divide G_i into a group of size 1 and a group containing the other elements. Last, when there are several such groups (e.g., when `lv` is $\texttt{a}[\texttt{i}]$ and $\texttt{i} \lhd G_0 \cup G_1$), the analysis first calls **merge** to merge all such groups and then falls back to the case where `lv` can only evaluate into a single group. This process is formalized as operator $\mathbf{materialize} : \{\texttt{lv}\} \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp$.

Note that in the last case, the merge of several groups may incur a loss in precision since the properties of several groups get merged before the abstract assignment takes place. We believe this loss in precision is acceptable here. The other option would be to produce a *disjunction* of abstract states, yet it would increase the analysis cost and the gain in precision would be unclear, as programmers typically view those disjunctions of groups of cells as having similar roles. Our experiments (Section 9) did confirm this intuition.

Materialization is not a novel technique in array analysis. Nevertheless, materialization gains extra precision due to the non-contiguous partition in our domain. To illustrate this, we compare the way our analysis performs with array analyses that are based solely on contiguous partitions. In Figure 12(a), array `a` is partitioned into three contiguous groups. The relation predicate $\texttt{i} \lhd G_0 \cup G_2$ indicates that variable `i` stores an index that belongs to group 0 or group 2. To materialize array cell $\texttt{a}[\texttt{i}]$, there are in general two methods: we could either let the analysis create two disjuncts such that, in each case the group $\texttt{a}[\texttt{i}]$ belongs to is fully known, or we could let it merge groups G_0 and G_2. The first solution obviously requires additional work in all cases. The second solution requires either a non-contiguous partition be created (since groups G_0 and G_2 are not adjacent), or a non trivial merge of groups throughout the array: in an array analysis that is based solely on contiguous partitions, this implies that *all groups between* G_0 *and* G_2 have to be merged, which incurs a significant, and unnecessary precision loss. In our domain, non-contiguous partitions are allowed, so that our analysis is able to merge only G_0 and G_2. Therefore, we can rely on the latter solution, so as to avoid (even local) disjunctions, and without losing much precision.

$$
\begin{array}{l}
\textbf{assign}(\texttt{lv} = \texttt{ex}, (p, \overrightarrow{n}, r))\{ \\
\quad (p, \overrightarrow{n}, r) = \textbf{materialize}(\texttt{lv}, (p, \overrightarrow{n}, r)); \\
\quad \texttt{ex}_{0\sim k}, r_{0\sim k} = \textbf{enumerate}(\texttt{ex}, r); \\
\quad \textbf{foreach}(i \in \{0, 1, \ldots, k\})\{ \\
\quad\quad \overrightarrow{n}_i = \textbf{assign}_{\textbf{Num}}(\texttt{lv} = \texttt{ex}_i, \overrightarrow{n}); \\
\quad\quad r_i = \textbf{propagate}(\texttt{lv} = \texttt{ex}_i, r_i); \\
\quad\quad p, \overrightarrow{n}_i, r_i = \textbf{reduce}(p, \overrightarrow{n}_i, r_i); \\
\quad \} \\
\quad (p, \overrightarrow{n}, r) = \textbf{join}_{\equiv}((p, \overrightarrow{n}_0, r_0), (p, \overrightarrow{n}_1, r_1), \ldots, (p, \overrightarrow{n}_k, r_k)); \\
\quad \textbf{return } (p, \overrightarrow{n}, r); \\
\}
\end{array}
$$

Figure 13: The algorithm of the assignment transfer function

Constraints. New relation predicates can be inferred by operator **propagate** : $\{\texttt{lv} = \texttt{ex}\} \times \{r\} \to \{r\}$. It propagates relation predicates in two ways: (1.) if both sides of the assignment are base variables, e.g. $\texttt{v} = \texttt{u}$, and we have $\texttt{u} \triangleleft G_i$, then after assignment, we get $\texttt{v} \triangleleft G_i$; (2) if the right hand side is an array cell as in $\texttt{parent} = \texttt{mproc}[\texttt{child}].\texttt{mparent}$ in the example of Section 2, since child stores an index in group G_0 ($\texttt{child} \triangleleft G_0$), the operator first looks for relations between fields and indexes such as $\widehat{\texttt{mparent}}_0 \triangleleft G_0$, and propagates them to the l-value as $\texttt{parent} \triangleleft G_0$.

In this phase, the numeric assignment relies on local disjuncts that are merged right after the abstract assignment, as we have shown in the case of condition tests (Section 6.1).

Algorithm. The algorithm of assignment is formalized in Figure 13. Operator **materialize** is used to materialize the array cell to be updated (when lv is a base type variable, it does not modify the abstract state). It utilizes **split** (sometimes also **merge**) to separate lv to compose a group by itself. Operator **enumerate** generates every possible state in which each array cell in ex belongs to exactly one group as in Section 6.1. Then, assignment $\textbf{assign}_{\textbf{Num}}$ of numeric domain with summary dimensions [17] is applied on each disjunctive state. Operator **propagate** infers new relation predicates without numerical information. The reduction operator is then applied on each disjuncts after both $\overrightarrow{n}$ and r are updated by $\textbf{assign}_{\textbf{Num}}$ and **propagate** respectively. Finally, all states are eventually joined together by the local disjunction join operator $\textbf{join}_{\equiv}$.

Example 9. *We consider* $\texttt{a}[\texttt{i}] = 2$ *and abstract pre-condition shown in Figure 12(a). The l-value evaluates into an index in* G_0 *or* G_2*, The result of materialization is shown in Figure 12(b), we can see that groups* G_0 *and* G_2 *are merged into group* G_0 *and* $\texttt{a}[\texttt{i}]$ *is split as the sole element of group* G_3*. Then, the assignment boils down to a strong update on numeric dimension* $\widehat{\texttt{value}}_3$*.*

Theorem 7 (Soundness). *Abstract transfer function* **assign** *is* sound *in the sense that:*

$$[\![\texttt{lv} = \texttt{ex}]\!]\gamma_{\textbf{St}}(a^\sharp) \subseteq \gamma_{\textbf{St}}(\textbf{assign}(a^\sharp, \texttt{lv}, \texttt{ex}))$$

Proof. The soundness of the assignment operator **assign** follows from the fact that operators **enumerate** and **reduce** do not change the concretization, from the soundness of the assignment operator $\textbf{assign}_{\textbf{Num}}$ of the numeric domain, from the soundness of **propagate** and **materialize** (since operators **split** and **merge** are sound), and of the local disjunction join operator $\textbf{join}_{\equiv}$.

Our analysis performs strong update in assignments, which captures the precise information on the concrete memory cells being modified. However, the merging phase that occurs before strong update might lead to a precision loss. Without such a merge, the analysis would have to enumerate all the groups an index may belong to, and to carry out a case analysis over this set of groups (each case would require a splitting of a group), which could turn out overly costly. This motivates the decision to perform the merge before the update. Additionally, and without a merging phase, the number of groups would be increased by one for each assignment, which could significantly impact the analysis performance.

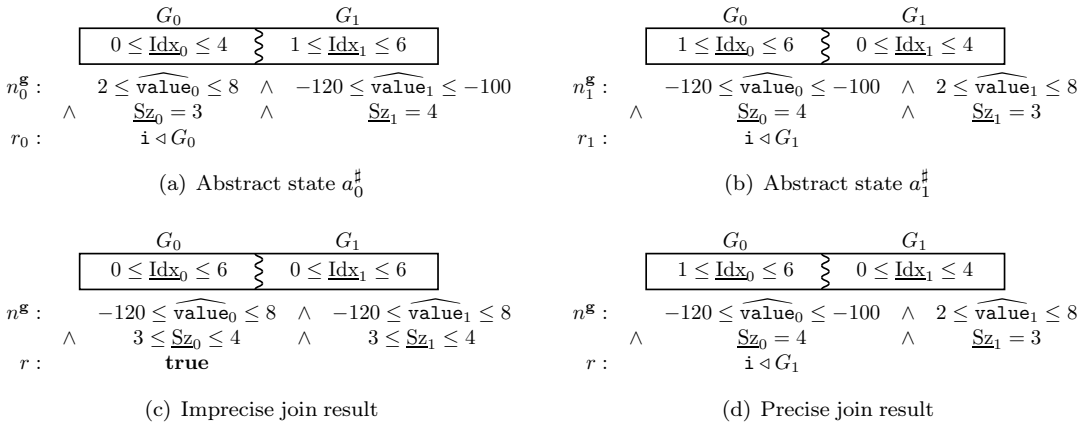


Figure 14: Impact of the group matching on the abstract join

Our analysis does not materialize the array elements that participate in condition tests (e.g., $\mathtt{a}[\mathtt{i}] == 0$). The other solution would be to split the array element (e.g., $\mathtt{a}[\mathtt{i}]$) out as a new separate group and to constrain its value to be zero. The reason is that compared to assignments, the precision our analysis gains from materialization in condition test does not seem worth the increased cost it would entail. Indeed, if there is no read / write operation in $\mathtt{a}[\mathtt{i}]$ after it has been materialized in condition test, there would be no precision gain.

7. Join, widening and inclusion check

Our analysis proceeds by standard abstract interpretation, and uses widening and inclusion test to compute abstract post-fixpoints for loops and abstract join for control flow union (e.g., after an **if** statement). All these operators face the same difficulties: they may be applied to a pair of abstract states that do not have compatible array partitions (either the numbers of partitions are different, or the groups that appear in both arguments have radically different meanings), thus, they may need to "re-partition" their arguments before they can compute any precise information. We discuss this issue in detail in the case of join.

7.1. Join and the group matching problem

Abstract join should compute an abstract state whose concretization over-approximates that of both of its arguments.

The partition compatibility problem. The local join operator $\mathbf{join}_{\equiv}$ shown in Definition 8 cannot be applied to pairs of abstract states that do not have the same number of groups. In fact, in the context of control flow joins (and not basic abstract post-conditions as in Section 6.1), this operator would not be adequate even when both inputs have the same number of groups.

Let us assume two abstract states $a_0^\sharp, a_1^\sharp$ with the same number of groups for each array, that we assume to have the same names. Then, the operator $\mathbf{join}_{\equiv}$ can be applied to these states, and computes an over-approximation for $a_0^\sharp, a_1^\sharp$, by joining predicates for each group name, the global numeric invariants and the side relations. However, this simple operator may produce very imprecise results if applied directly. As an example, we show two abstract states $a_0^\sharp$ and $a_1^\sharp$ in Figure 14(a) and Figure 14(b), that are similar up to a group name permutation. The direct join is shown in Figure 14(c). We note that the exact size of groups and the tight constraints over `value` were lost. Conversely, if the same operation is done after a permutation of group names, an optimal result is found, as shown in Figure 14(d).

Obviously, this *group matching problem* is actually even more complicated in general as $a_0^\sharp, a_1^\sharp$ usually do not have the same number of groups. To address this, we need to define a join operator that modifies partitions, and uses constraints over partition group fields into account to decide what partition modification is most adequate.

Ranking function. To properly associate G_0 in Figure 14(a) with G_1 in Figure 14(b), the analysis should take into account the *group field properties*. This is achieved with the help of a ranking function $\mathbf{rank} : \mathbb{G} \times \mathbb{G} \to \mathbb{N}$, which computes a distance between groups of cells of the same array in different abstract states by comparing their numerical and relation predicates. A high value of $\mathbf{rank}(G_i, G_j)$ indicates G_i of $a_0^\sharp$ and G_j of $a_1^\sharp$ are likely to describe sets of cells with similar properties.

The value of $\mathbf{rank}(G_i, G_j)$ is determined by three factors:

- the number of common constraints on the dimensions associated to fields and indexes in $\overrightarrow{n}$ (including their ranges and, when a relational abstract domain is used, relations with program variables);
- the number of variables that have the same value ranges in both states, and that have var-index relations with both groups;
- the "group origin", determined by group names in the representation of the abstract values (this allows the analysis to detect that two groups were computed from a single group instance, in a predecessor abstract state).

Re-partitioning. Using the set of $\mathbf{rank}(G_i, G_j)$ values, the analysis computes a *pairing* $\leftrightarrow \in \mathcal{P}(\mathbb{G} \times \mathbb{G})$, that is a set of relations between groups of $a_0^\sharp$ and groups of $a_1^\sharp$. The pairing is defined by the rules below:

1. we sort all pairs of groups decreasingly according to their ranking values, and then select the first k pairs (k is a constant number, usually the analysis lets k be the maximal number of groups in $a_0^\sharp$ and $a_1^\sharp$). That is, if the value of $\mathbf{rank}(G_i, G_j)$ is among the highest k ranking values of all group pairs, a relation $G_i \leftrightarrow G_j$ is added to the pairing;
2. if three relations of the form $G_i \leftrightarrow G_k$, $G_i \leftrightarrow G_j$ and $G_t \leftrightarrow G_j$ have been added to the pairing, then the "middle" relation $G_i \leftrightarrow G_j$ is removed.

The algorithm we choose to filter pairs is based on heuristics, yet a non optimal pairing will impact only precision, but not soundness. After the two steps above, our analysis applies a *partition transforming* which transforms both arguments into "compatible" abstract states using the following (symmetric) principles:

- if there is no G_j such that $G_i \leftrightarrow G_j$, then an empty such group is created with **create**;
- if $G_i \leftrightarrow G_j$ and $G_i \leftrightarrow G_k$, then G_j and G_k are merged by operator **merge** and the resulting group is paired with G_i;
- if G_i is mapped only to G_j, G_j is mapped only to G_i, and $i \neq j$, then one of them is renamed accordingly (so that they carry the same name).

The process of pairing and partition transforming is formalized as operator **repartition**. It takes two abstract states, together with a set of ranking values, and outputs two "compatible" abstract states. After this process has completed, a pair of abstract states is produced that have the same number of groups, and such that groups of the same name carry similar abstract predicates, and $\mathbf{join}_{\equiv}$ can be applied. This defines the abstract join operator **join**.

The algorithm of **join** is shown in Figure 15. It first computes the ranking values of all groups from two abstract states by operator **rank**, and then repartition the two states by **repartition** (with **create** and **merge**) according to the ranking values. Finally, it applies $\mathbf{join}_{\equiv}$ on two compatible states.

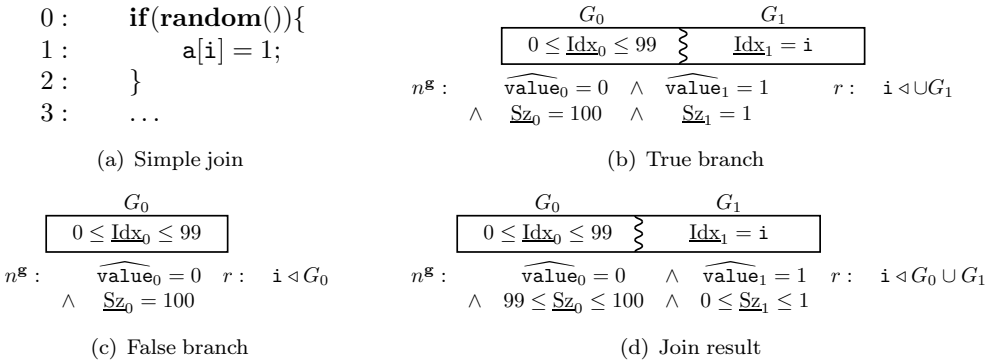
Theorem 8 (Soundness). *Join operator* **join** *is* sound *in the sense that:*

$$\forall a_0^\sharp, a_1^\sharp,\ \gamma_{\mathbf{St}}(a_0^\sharp) \subseteq \gamma_{\mathbf{St}}(\mathbf{join}(a_0^\sharp, a_1^\sharp)) \wedge \gamma_{\mathbf{St}}(a_1^\sharp) \subseteq \gamma_{\mathbf{St}}(\mathbf{join}(a_0^\sharp, a_1^\sharp))$$

Proof. The soundness of the join operator **join** follows from the fact that the operator **repartition** is sound (since **create** and **merge** are sound), and from the soundness of $\mathbf{join}_{\equiv}$.

$\mathbf{join}(a_0^\sharp, a_1^\sharp)\{$
  $\mathbf{foreach}(G_i \mathbf{\ in\ } a_0^\sharp)$
    $\mathbf{foreach}(G_j \mathbf{\ in\ } a_1^\sharp)$
      $W_{ij} = \mathbf{rank}(G_i, G_j);$
  $a_0^\sharp, a_1^\sharp = \mathbf{repartition}(W, a_0^\sharp, a_1^\sharp);$
  $\mathbf{return\ join}_{\equiv}(a_0^\sharp, a_1^\sharp);$
$\}$

Figure 15: The algorithm of the join operator



(a) Simple join (b) True branch

(c) False branch (d) Join result

Figure 16: Join of a one group state with a two groups state

Example 10. *We assume* `a` *is an integer array of length* 100 *and* `i` *is an integer variable storing a value in* $[0, 99]$*, and consider the program of Figure 16(a). At the exit of the* **if** *statement, the analysis needs to join the pre-condition (also the state in false branch) shown in Figure 16(c) (that has a single group) and the state in true branch shown in Figure 12(b) (that has two groups). We note that* G_0 *in Figure 16(c) has similar properties as* G_0 *in Figure 16(b), thus they get paired. Moreover,* G_1 *in Figure 16(b) is paired to no group, so a new group is created, and paired to it. At that stage* $\mathbf{join}_{\equiv}$ *applies, and returns the abstract state shown in Figure 16(d).*

Our join operator is not optimal. That is due to the fact the algorithm for pairing is heuristic and thus may not find the best pairing relations. Join may also lose precision due to the **merge** operator.

7.2. Widening

The widening algorithm is similar to that of join, but with a different re-partitioning strategy that ensures termination.

Case of compatible partitions. We first define a restriction of widening to *compatible* abstract states (that is, abstract states, whose partitions have the same numbers of groups, with the same names):

Definition 9 (Widening for compatible states). The *widening for compatible states* is defined by

$$\mathbf{widen}_{\equiv}((p, \overrightarrow{n}_0, r_0), (p, \overrightarrow{n}_1, r_1)) = (p, \mathbf{widen}_{\mathbf{Num}}(\overrightarrow{n}_0, \overrightarrow{n}_1), r_0 \cap r_1)$$

This operator clearly defines a widening operator. Indeed the widening operator $\mathbf{widen}_{\mathbf{Num}}$ of the numeric domain ensures convergence, when the number of numeric dimensions in $\overrightarrow{n}_0, \overrightarrow{n}_1$ is bounded, and in the case of $\mathbf{widen}_{\equiv}$, it is constant (the set of groups is fixed here). Similarly, the resulting set of relation predicates is included in the set of relation predicates r_0, r_1 of the arguments, which are finite sets, thus this component is well-founded and will also eventually converge. However, the termination property of $\mathbf{widen}_{\equiv}$ relies on the assumption that partitions never change. As this assumptions is obviously not satisfied in general, we define a widening operator that can be applied to any sequence of abstract states, with no assumption on the partitions.

$$
\begin{array}{l}
\textbf{widen}(a_0^\sharp, a_1^\sharp)\{ \\
\quad \textbf{foreach}(G_i \textbf{ in } a_0^\sharp) \\
\quad\quad \textbf{foreach}(G_j \textbf{ in } a_1^\sharp) \\
\quad\quad\quad W_{ij} = \textbf{rank}(G_i, G_j); \\
\quad a_0^\sharp, a_1^\sharp = \textbf{repartition}_\triangledown(W, a_0^\sharp, a_1^\sharp); \\
\quad \textbf{return } \textbf{widen}_\equiv(a_0^\sharp, a_1^\sharp); \\
\}
\end{array}
$$

Figure 17: The algorithm of the widening operator

Re-partitioning for widening. To achieve termination, **widen** needs to ensure that the partition component eventually converges: after the partition component has converged, $\textbf{widen}_\equiv$ can be applied, and will ensure both soundness and termination. This convergence property is *not guaranteed* by the group matching algorithm of Section 7.1. Therefore the widening operator **widen** relies on a slightly different group re-partitioning operator $\textbf{repartition}_\triangledown$. The only difference between it with **repartition** lies in the pairing phase:

1. operator $\textbf{repartition}_\triangledown$ pairs each group with the group with which it has the highest ranking value;
2. if three relations of the form $G_i \leftrightarrow G_k$, $G_i \leftrightarrow G_j$ and $G_t \leftrightarrow G_j$ have been added to the pairing, the "middle" relation $G_i \leftrightarrow G_j$ gets removed.

The new pairing scheme pairs every group with at least one other group, which has the effect that no **create** is needed in the partition transforming phase. Actually, only operator **merge** is used. This group matching ensures termination.

The algorithm of **widen** is shown in Figure 17. It just replaces **repartition** and $\textbf{join}_\equiv$ in the algorithm of join with $\textbf{repartition}_\triangledown$ and $\textbf{widen}_\equiv$ respectively.

The resulting **widen** operator is a sound and terminating widening operator [10]. For better precision, the analysis always uses **join** for the first abstract iteration for a loop, and uses widening afterwards.

Theorem 9 (Soundness and termination). *The operator* **widen** *is a widening operator: it over-approximates its arguments and ensures the termination of abstract iterates.*

Proof. As in the case of **join**, the fact that **widen** returns an over-approximation of its inputs follows from the soundness of the basic operators on groups. Thus, we only have to establish the convergence of any sequence of abstract iterates of the form $a_{n+1}^\sharp = \textbf{widen}(a_n^\sharp, a_n^{\sharp\prime})$.

Since **widen** never calls **create** and **split**, and changes the number of groups only by calling **merge**, the number of groups in its result decreases in any sequence of widened iterates, and eventually stabilizes after finitely many steps. From that point, groups are stable. Also, the height of the set of relation constraints over these groups is finite, thus the r component will also stabilize after finitely many iterates. Therefore, since **widen** applies $\textbf{widen}_{\textbf{Num}}$ on the numeric constraints component, it ensures the termination of any sequence of abstract iterates.

Therefore, **widen** is a widening operator.

Example 11. *We consider the abstract states depicted in Figure 16(b) and in Figure 16(c) and show how* **widen** *applies to these abstract states. The group matching algorithm will merge the two groups in Figure 16(b) and pair the resulting group to the only group in Figure 16(c). The output state after applying* $\textbf{widen}_\equiv$ *is shown in Figure 18.*

7.3. Inclusion check

To check the termination of sequences of abstract iterates over loops, and the entailment of post-conditions, the analysis uses a sound inclusion checking operator **isle**: when $\textbf{isle}(a_0^\sharp, a_1^\sharp)$ returns `TRUE`, then $\gamma_{\mathbf{St}}(a_0^\sharp) \subseteq \gamma_{\mathbf{St}}(a_1^\sharp)$.

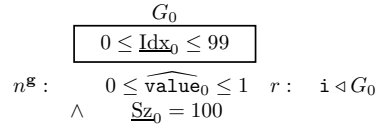


Figure 18: Widening result of two abstracts with different partitions

$$
\begin{array}{l}
\textbf{isle}(a_0^\sharp, a_1^\sharp)\{ \\
\quad \textbf{foreach}(G_i \textbf{ in } a_0^\sharp) \\
\quad\quad \textbf{foreach}(G_j \textbf{ in } a_1^\sharp) \\
\quad\quad\quad W_{ij} = \textbf{rank}_<(G_i, G_j); \\
\quad \textbf{if}(\exists G_i \textbf{ in } a_0^\sharp, \forall G_j \textbf{ in } a_1^\sharp, W_{ij} \leq 0) \\
\quad\quad \textbf{return } \texttt{FALSE}; \\
\quad a_0^\sharp = \textbf{repartition}_<(W, a_0^\sharp); \\
\quad \textbf{return isle}_\equiv(a_0^\sharp, a_1^\sharp); \\
\}
\end{array}
$$

Figure 19: The algorithm of the inclusion check operator

As in the case of join, a restricted inclusion checking operator $\textbf{isle}_\equiv$ can be defined in a straightforward manner, that checks inclusion on "compatible" abstract states, that is abstract states with matching partitions: if $\textbf{is\_le}_{\textbf{Num}}(\overrightarrow{n}_0, \overrightarrow{n}_1) = \texttt{TRUE}$ and r_1 is included in r_0 (as a set of constraints), then $\gamma_{\mathbf{St}}(p, \overrightarrow{n}_0, r_0) \subseteq \gamma_{\mathbf{St}}(p, \overrightarrow{n}_1, r_1)$, hence we let $\textbf{isle}_\equiv$ return `TRUE` in that case.

The inclusion checking algorithm is quite similar to that of join, but uses a modified ranking operator $\textbf{rank}_<$ and a modified re-partition operator $\textbf{repartition}_<$. Operator $\textbf{rank}_<$ is the same as **rank** except that it evaluates $\textbf{rank}_<(G_0, G_0')$ into a negative integer when the ranges of indexes, field contents and size of group G_0 (from the left argument) are not included into those of G_0'. The difference of $\textbf{repartition}_<$ with **repartition** lies in two aspects: firstly, in the pairing phase $\textbf{repartition}_<$ guarantees that each group from the left argument is paired with at least one group in the right argument; secondly, the partition transforming phase only modifies the groups in the left argument so as to construct an abstract state with the same groups as the right argument. This means that, when two groups G_j and G_k from the right argument are paired with a single group G_i in the left argument, the inclusion checking algorithm will apply **split** to G_i and pair the two resulting groups with G_j and G_k respectively.

The inclusion checking algorithm is shown in Figure 19. It first computes the ranking values of all groups from two abstract states using operator **rank**. If there is a group from the left argument that ranks negatively with all groups from the right argument, **isle** conservatively returns false. Otherwise, it re-partitions the two states by $\textbf{repartition}_<$ (with **create**, **split** and **merge**) according to the ranking values. Finally, it applies $\textbf{isle}_\equiv$ to two compatible states.

Theorem 10 (Soundness). *The inclusion check operator* **isle** *is* sound *in the sense that*

$$\textbf{isle}(a_0^\sharp, a_1^\sharp) = \mathit{TRUE} \Longrightarrow \gamma_{\mathbf{St}}(a_0^\sharp) \subseteq \gamma_{\mathbf{St}}(a_1^\sharp)$$

Proof. First, we note that $\textbf{repartition}_<$ does not modify the right hand side argument, and performs an *over*-approximation of the left hand side argument (through **create**,**split** and **merge**). Second, $\textbf{isle}_\equiv$ is sound. Therefore, when **isle** returns true, all elements in the concretization of the left argument also belong to the concretization of the right argument. Hence, it is sound.

Our inclusion check operator is not complete because of the heuristics of pairing algorithm and the precision loss in operators on partitions.

8. Static analysis on programs involving arrays

In this section, we formalize an abstract interpreter for the language of Section 3, and we discuss in detail the full analysis of the `cleanup` example.

8.1. Abstract semantics

The abstract semantics of a program (a sequence of statements) $[\![s]\!]^\sharp : \mathbb{D}^\sharp \to \mathbb{D}^\sharp$ is a function that maps an abstract pre-condition into an abstract post-condition. To define it, we let $\mathbf{lfp}^\#$ denote an abstract post-fixpoint operator, that applies to an abstract transformer, and computes an abstract post-fixpoint: if $\mathbf{F} : \mathcal{P}(\mathbb{S}) \to \mathcal{P}(\mathbb{S})$ is a continuous concrete function and $\mathbf{F}^\# : \mathbb{D}^\sharp \to \mathbb{D}^\sharp$ is such that $\mathbf{F} \circ \gamma_{\mathbf{St}} \subseteq \gamma_{\mathbf{St}} \circ \mathbf{F}^\#$, then:

$$\mathbf{lfp}_{\gamma_{\mathbf{St}}(a^\sharp)} \subseteq \gamma_{\mathbf{St}}(\mathbf{lfp}^\#_{a^\sharp} \mathbf{F}^\#)$$

In practice, as usual, $\mathbf{lfp}^\#$ computes a sequence of the abstract iterates, enforces its converges using widening, uses inclusion check to check when convergence has occurred, and may perform unroll iterations (our analysis unrolls all loops exactly once) or apply other techniques to enhance the precision of abstract post-fixpoint.

Definition 10 (Abstract semantics). The abstract semantics $[\![\mathtt{s}]\!]^\sharp$ of a statement $\mathtt{s}$ is defined by:

$$\begin{array}{rcl}
[\![\mathtt{skip}]\!]^\#(a^\sharp) & = & a^\sharp \\
[\![\mathtt{lv} = \mathtt{ex}]\!]^\#(a^\sharp) & = & \mathbf{assign}(a^\sharp, \mathtt{lv}, \mathtt{ex}) \\
[\![\mathtt{s}_1; \mathtt{s}_2]\!]^\#(a^\sharp) & = & [\![\mathtt{s}_2]\!]^\# \circ [\![\mathtt{s}_1]\!]^\#(a^\sharp) \\
[\![\mathbf{if}(\mathtt{cond})\{\mathtt{s}_1\}\mathbf{else}\{\mathtt{s}_2\}]\!]^\#(a^\sharp) & = & \mathbf{join}([\![\mathtt{s}_1]\!]^\# \circ \mathbf{test}(\mathtt{cond}, a^\sharp), [\![\mathtt{s}_2]\!]^\# \circ \mathbf{test}(\mathtt{cond} == \mathtt{FALSE}, a^\sharp)) \\
[\![\mathbf{while}(\mathtt{cond})\{\mathtt{s}\}]\!]^\#(a^\sharp) & = & \mathbf{test}(\mathtt{cond} == \mathtt{FALSE}, \mathbf{lfp}^\#_{a^\sharp} \mathbf{F}^\#) \\
 & & \text{where } \mathbf{F}^\# := a^\sharp \mapsto [\![\mathtt{s}]\!]^\#(\mathbf{test}(\mathtt{cond}, a^\sharp))
\end{array}$$

Theorem 11 (Soundness). *Given a program* $\mathtt{s}$ *and an abstract pre-condition* $a^\sharp$*, the post-condition derived by the analysis is sound:*

$$[\![\mathtt{s}]\!](\gamma_{\mathbf{St}}(a^\sharp)) \subseteq \gamma_{\mathbf{St}}([\![s]\!]^\#(a^\sharp))$$

Proof. By induction over the syntax, and using the soundness of all the primitives called by the analysis.

8.2. Example `cleanup` *revisited*

In Section 2, we overviewed some key parts of the analysis of an excerpt of function `cleanup`, which is part of the Minix memory management process table operations. We now provide more details about this analysis.

The function `cleanup` should always be called in a state where the Minix Memory Management Process Table satisfies global correctness property $\mathcal{C}$ described in Figure 2(b), and where argument `child` is the identifier of a valid user process descriptor. Therefore, the analysis starts with pre-condition $\mathcal{C} \wedge \mathtt{child} \triangleleft G_0 \wedge \mathtt{child} > 2$. It then proves that, under this pre-condition, and after executing the body of `cleanup`, $\mathcal{C}$ always holds: to achieve this, it verifies that $\mathbf{isle}([\![\mathtt{cleanup}(\mathtt{child})]\!]^\#(\mathcal{C} \wedge \mathtt{child} \triangleleft G_0 \wedge \mathtt{child} > 2), \mathcal{C}) = \mathtt{TRUE}$.

In Figure 20, we describe with more details the analysis of the excerpt of `cleanup`. At start-up, we get pre-condition $\mathcal{C} \wedge \mathtt{child} \triangleleft G_0 \wedge \mathtt{child} > 2$ (before line 1 in the figure). Because of the property of field `mparent` in group G_0 according to $\mathcal{C}$, we obtain $\mathtt{parent} \triangleleft G_0$ at the beginning of line 2. At line 4, since `child` could be any cell whose index is largter or queal to 2 in group G_0. The analysis performs a materialization during the analysis of that update, which splits group G_0 into groups G_0, G_2, as shown after line 4.

Then, the analysis enters the **while** loop that starts at line 6. For the sake of clarity, we show only the abstract states computed after the convergence of the sequence of widening iterates (note those are the final invariants, and not the abstract states for the unroll iteration). The loop head invariant is shown right after line 6. For the sake of space and readability, we elide some groups and the properties of their fields:

- group G_1 always describes the slots that were free *before* the call to `cleanup` (note that excludes the process descriptor of index `child` that is being freed).

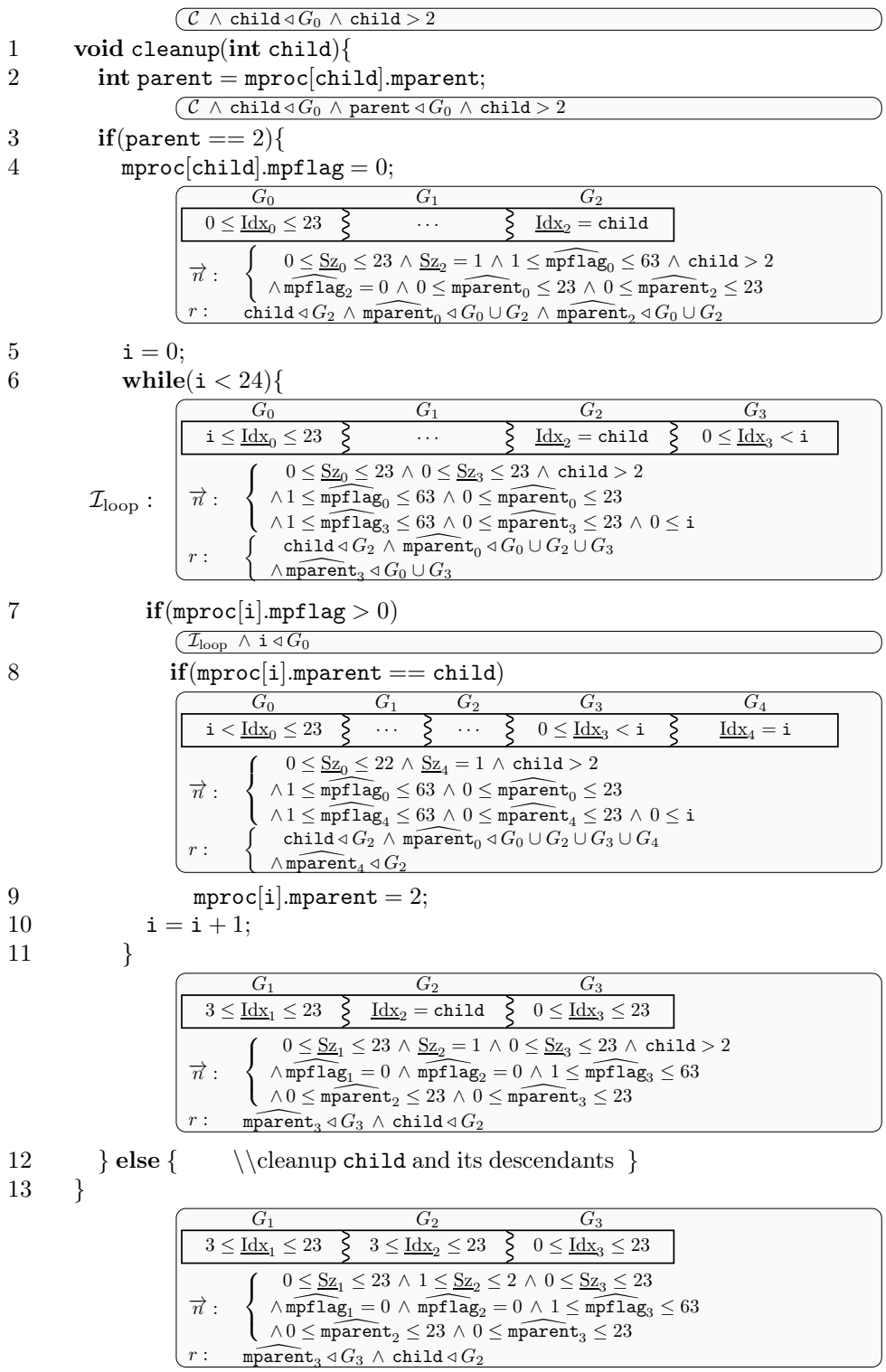


Figure 20: Analysis of the `cleanup` excerpt

In addition to these, and at loop head, we have the following groups:

- group G_0 describes the valid process descriptors that have not yet been visited by the loop (i.e., with an index greater or equal than `i`);
- group G_2 describes a group that consists of exactly one cell, corresponding to the process descriptor that is being cleaned up (cell `mproc[child]`);
- group G_3 describes the valid process descriptors that were already examined during the loop (i.e., with an index strictly lower than `i`).

The test at line 7 entails that `i` cannot be in groups G_1 and G_2 (all those processes have a null flag), thus, $\texttt{i} \triangleleft G_0$. The test at line 8 keeps only the states where `i` is the index of a child of the process being cleaned up. This test leads to the splitting of that group, which enables a strong update at line 9.

We now briefly discuss the abstract iterates that lead to this invariant. During the first iteration, a new group is created so that, during the loop, the analysis always distinguishes the valid process descriptors with an index strictly lower than `i` from those with an index that is greater or equal than `i`. Not applying **widen** at the end of the first iteration, and delaying it to the second iteration allows to preserve this group. At the end of the subsequent widening iterations, the groups corresponding to index `i` and to indexes lower than or equal to `i` are merged together.

Last, when exiting the loop, $\texttt{i} \geq 24$. Since, the loop head invariant contains constraints $\texttt{i} \leq \underline{\text{Idx}}_0$ and $\underline{\text{Idx}}_0 \leq 23$, this group is necessarily empty, and can be removed. After removal of that group, the analysis produces the abstract post-condition shown at line 11.

The post-condition of function `cleanup` is presented right after line 13. Actually the only difference with the state after line 11 is that group 2 may contains more elements (more slots might be cleaned up in the **else** branch). It is easy to prove that it implies $\mathcal{C}$ by comparison operator **isle**.

9. Experimental evaluation

We have implemented our analysis and evaluated how it copes with two classes of programs: (1) process tables as found in operating systems, including the Minix Memory Management Process Table and the task scheduling table of a proprietary Embedded Operating System currently under development, and (2) academic examples introduced in related works, and where we demonstrate that partitions in contiguous groups are not strictly necessary for the verification. Our abstract domain has been integrated into the MemCAD static analyzer [30, 32, 7]. It uses the Apron library of numerical abstract domains [21]. In practice, our analysis uses octagons [26] for all test cases except one that is analyzed using convex polyhedra [9].

9.1. Verification of codes from operating systems

Verification of memory management part in Minix. The main data-structure of the Memory Management operating system service of Minix 1.1 is the MMPT `mproc`, which contains memory management information for each process. At start up, it is initialized by function `mm_init`, which creates process descriptors for `mm`, `fs` and `init`. After that, `mproc` should satisfy property $\mathcal{C}$ (Section 2). Then, it gets updated by system calls `fork`, `wait` and `exit`, which respectively create a process, wait for terminated children process descriptors to be removed, and terminate a process. Each of these functions should be called only in a state that satisfies $\mathcal{C}$, and should return a state that also satisfies $\mathcal{C}$ (we recall $\mathcal{C}$ was defined in Figure 2(b), and splits the indexes in the process table into two groups: group G_0 contains all the indexes of the valid processes whereas group G_1 contains all the indexes of the "free cells" in the table). If property $\mathcal{C}$ was violated, several critical issues could occur. First, system calls could crash due to out-of-bound accesses, e.g., when accessing `mproc` through field `mparent`. Moreover, higher level, hard to debug issues could occur, such as the persistence of dangling processes, that would never be eliminated.

Therefore, we verified, using our analysis, that (1) `mm_init` properly initializes the structure, so that $\mathcal{C}$ holds afterwards (under no pre-condition), and that (2) `fork`, `wait` and `exit` preserve $\mathcal{C}$ (i.e., the analysis of

Program	LOCs	Verified property	Time(s)	Max. groups	Num. domain	Description
`mm_init`	26	establishes $\mathcal{C}$	0.12	3	Octagons	Minix MMPT: `mproc` init
`fork`	22	preserves $\mathcal{C}$	0.07	3	Octagons	Minix MMPT sys. call
`exit`	68	preserves $\mathcal{C}$	3.75	6	Octagons	Minix MMPT sys. call
`wait`	70	preserves $\mathcal{C}$	3.88	6	Octagons	Minix MMPT sys. call
`tinit`	52	establishes $\mathcal{T}$	0.17	7	Octagons	Task scheduling table init
`tcreat`	29	preserves $\mathcal{T}$	0.13	4	Octagons	AOS sys. call
`tsched`	31	preserves $\mathcal{T}$	0.41	5	Octagons	AOS sys. call
`tstop`	60	preserves $\mathcal{T}$	3.52	7	Polyhedra	AOS sys. call
`tstart`	73	preserves $\mathcal{T}$	1.25	5	Octagons	AOS sys. call
`complex`	21	$\forall i \in [0, 54], a[i] \geq -1$	0.296	4	Octagons	Example from [12]
`int_init`	8	$\forall i \in [0, N], a[i] = 0$	0.025	3	Octagons	Array initialization

Figure 21: Analysis results (timings measured on Ubuntu 12.04.4, with 16 Gb of RAM, on an Intel Xeon E3 desktop, running at 3.2 GHz)

each of these functions from pre-condition $\mathcal{C}$ returns a post-condition that also satisfies $\mathcal{C}$). This verification boils down to the following computations:

$$\begin{array}{lcl}
\textbf{isle}(\llbracket \texttt{mm_init} \rrbracket^{\#}(\top), \mathcal{C}) & = & \texttt{TRUE} \\
\textbf{isle}(\llbracket \texttt{exit(who)} \rrbracket^{\#}(\mathcal{C} \wedge \texttt{who} \lhd G_0 \wedge \texttt{who} > 2), \mathcal{C}) & = & \texttt{TRUE} \\
\textbf{isle}(\llbracket \texttt{fork(who)} \rrbracket^{\#}(\mathcal{C} \wedge \texttt{who} \lhd G_0), \mathcal{C}) & = & \texttt{TRUE} \\
\textbf{isle}(\llbracket \texttt{wait(who)} \rrbracket^{\#}(\mathcal{C} \wedge \texttt{who} \lhd G_0), \mathcal{C}) & = & \texttt{TRUE}
\end{array}$$

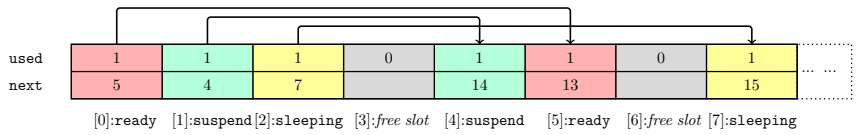
Note that function `cleanup` was inlined in `wait` and `fork` in a recursion free form (our analyzer currently does not supported recursion). Our tool achieves the verification of all these four functions. The results are shown in the first four lines of the table in Figure 21, including analysis time and peak number of groups for array `mproc`.

The analysis of `mm_init` and `fork` is very fast. The analysis of `exit` and `wait` also succeeds, although it is more complex due to the intricate structure of `cleanup` (which consists of five loops and a large number of tests) which requires 151 joins. Despite this, the maximum number of groups remains reasonable (six in the worst case).

Verification of task scheduling part in an industrial operating system: AOS. To assess the ability of our analysis to reason about process tables found in operating systems, and beyond the Minix case, we have also analyzed the task scheduling part of a more recent, industrial small embedded operating system, which we refer to as *AOS*. It uses a table `taskq` to record the status of all processes. Figure 22(a) shows the definition of this table. Field `used` indicates whether this cell is a valid process descriptor or a free slot. Since processes may get allocated and may exit at any time, the set of valid process descriptors is dynamic, and may correspond to any set of indexes of `taskq`. At a given time, a process may be either in state *ready*, in state *suspended* or in state *sleeping*.

```
1   struct taskq {
2           unsigned used;
3           unsigned next;
4.  } taskq[16];
```

(a) Definition of `taskq`



(b) An example concrete state of `taskq`

The processes that are in the same status are connected together by field `next`, which means there are three embedded lists in this table. The first three and the last three elements of array `taskq` are not used to store information of any process; instead, they respectively perform as list heads and tails. Figure 22(b) shows the first 8 elements of `taskq` in a given concrete state. We can see that the `used` cells are not contiguous and the elements of the three lists are interleaved with each other. This array is manipulated by five functions listed below:

- Function `tinit` initializes array `taskq` by setting the first and last three elements as the heads and tails of the three lists that respectively correspond to *ready*, *suspended* and *sleeping* processes;
- System call `tcreat` finds a free slot in `taskq`, changes it to `used` and adds it to the list of *sleeping* processes;
- System call `tsched` decides which state the executing process should be in after its current execution;
- System call `tstop` moves one node from the list of *ready* processes to the list of *sleeping* processes;
- System call `tstart` moves one node from the list of *sleeping* processes to the list of *ready* processes.

Instead of abstracting lists embedded in arrays (which is very complex, and essentially requires an abstract domain similar to that of [30]), we verify an invariant which is a bit weaker but still sufficient to verify memory safety. The property we defined states that the `next` field of each `used` cell points to a `used` cell (it is also true on the tails of the three lists because the value of their `next` fields are all 0 which also indexes a `used` cell). We call this invariant $\mathcal{T}$, and we have verified that function `tinit` establishes this invariant on start-up and that the other four functions preserve this invariant. To achieve this, our analysis simply carries out the computations below:

$$\begin{array}{lcl}
\textbf{isle}([\![\texttt{tinit}]\!]^{\#}(\top), \mathcal{T}) & = & \texttt{TRUE} \\
\textbf{isle}([\![\texttt{tcreat}]\!]^{\#}(\mathcal{T}), \mathcal{T}) & = & \texttt{TRUE} \\
\textbf{isle}([\![\texttt{tstart}]\!]^{\#}(\mathcal{T}), \mathcal{T}) & = & \texttt{TRUE} \\
\textbf{isle}([\![\texttt{tstop}]\!]^{\#}(\mathcal{T}), \mathcal{T}) & = & \texttt{TRUE} \\
\textbf{isle}([\![\texttt{tsched}]\!]^{\#}(\mathcal{T}), \mathcal{T}) & = & \texttt{TRUE}
\end{array}$$

Our analysis successfully verified all these properties. The analysis results are shown in Figure 21. We can see that the run-times and maximal numbers of groups are quite reasonable. The test case with the longest analysis time is `tstop`, and this longer analysis time is due to a high number of joins and due to the need to use the abstract domain of convex polyhedra, whose precision is necessary to verify that `tstop` preserves $\mathcal{T}$. The reason why polyhedra are needed here is to express properties of the form $\underline{\mathrm{Sz}}_0 + \ldots + \underline{\mathrm{Sz}}_k = N$, where at least three groups have a size that cannot be proved equal to a constant (thus, numerical abstract domains with at most two variables per constraint will incur a loss in precision).

9.2. Application to academic test cases

We now consider a couple of examples from the literature, where arrays are used as containers, i.e., where the relative order of groups does not matter for the program's correctness. The purpose of this study is to exemplify other examples of cases our abstract domain is adequate for. Program `int_init` consists of a simple initialization loop. Our analysis succeeds here, and can handle other cases relying on basic segments, although our algorithms are not specific to segments (and are geared towards the abstraction of non contiguous partitions).

Moreover, Figure 22 shows `complex`, an excerpt of an example from [12]. The second example is challenging for most existing techniques, as observed in [12] since resolving `a[index]` at line 10 is tricky. As shown in Figure 21, our analysis handles these two loops well, with respectively 4 and 3 groups.

The invariant of the first initialization loop in Figure 22 is abstract state ① (at line 4): group G_1 accounts for initialized cells, whereas cells of G_0 remain to be initialized. The analysis of `a[i] = 0;` from ① materializes a single uninitialized cell, so that a strong update produces abstract state ②. At the next iteration, and after increment operation `i++`, widening merges G_2 with G_1, which produces abstract state ① again. At loop exit, the analysis derives G_0 is empty as $56 \leq \underline{\mathrm{Idx}}_0 \leq 55$. At this stage, this group is eliminated. The analysis of the second loop converges after two widening iterations, and produces abstract state ③. We note that group G_3 is kept separate, while groups G_1 and G_2 get merged when the assignment at line 10 is analyzed (Section 6.2). This allows to prove the assertion at line 11.

10. Related work and conclusion

In this paper, we have presented a novel abstract domain that is tailored for arrays, and that relies on partitioning, without imposing the constraint that the cells of a given group be contiguous.

```
2   int a[56];
3   for(int i = 0; i < 56; i++){
      ①
4       a[i] = 0;
      ②
    }
5   a[55] = random();
6   for(int i = 0; i < 55; i++){
      ③
7       int index = 21 * i%55;
8       int num = random();
9       if(num < 0){num = −1;}
10      a[index] = num;
    }
11  assert(∀i ∈ [0, 54], a[i] ≥ −1);
```

(c) Test case `complex`

state ① G_0 G_1

$\boxed{\texttt{i} \le \underline{\text{Idx}}_0 \le 55 \;\wr\; 0 \le \underline{\text{Idx}}_1 \le \texttt{i} - 1}$

$n^{\mathbf{g}}$: $\underline{\text{Sz}}_0 = 56 - \texttt{i} \wedge \underline{\text{Sz}}_1 = \texttt{i} \wedge \widehat{\texttt{value}}_1 = 0$

r : $\texttt{i} \lhd G_0$

state ② G_0 G_1 G_2

$\boxed{\texttt{i} + 1 \le \underline{\text{Idx}}_0 \le 55 \;\wr\; 0 \le \underline{\text{Idx}}_1 \le \texttt{i} - 1 \;\wr\; \underline{\text{Idx}}_2 = \texttt{i}}$

$n^{\mathbf{g}}$: $\underline{\text{Sz}}_0 = 55 - \texttt{i} \wedge \underline{\text{Sz}}_1 = \texttt{i} \wedge \underline{\text{Sz}}_2 = 1$
$\widehat{\texttt{value}}_1 = 0 \wedge \widehat{\texttt{value}}_2 = 0$

r : $\texttt{i} \lhd G_2$

state ③ G_1 G_2 G_3

$\boxed{0 \le \underline{\text{Idx}}_1 \le 54 \;\wr\; 0 \le \underline{\text{Idx}}_2 \le 54 \;\wr\; \underline{\text{Idx}}_3 = 55}$

$n^{\mathbf{g}}$: $\underline{\text{Sz}}_1 = 54 \wedge \underline{\text{Sz}}_2 = 1 \wedge \underline{\text{Sz}}_3 = 1$
$-1 \le \widehat{\texttt{value}}_1 \wedge -1 \le \widehat{\texttt{value}}_2$

r : $\texttt{i} \lhd G_1 \cup G_2$

(d) Invariants

Figure 22: Array random accesses

Most array analyses require each group be a contiguous array *segment*. Abstract interpretation based static analysis tools [4, 18, 20] and [12] contiguously partition arrays over indexes statically and dynamically respectively. Tools based on decision procedures [1, 2, 5], and theorem provers [22, 29, 25, 23] can describe properties of array cells over a certain range of indexes. We believe that both approaches are adequate for different sets of problems: segment based approaches are adequate to verify algorithms that use arrays to order elements, such as sorting algorithms, while our segment-less approach works better to verify programs that use arrays as dictionaries.

Other works target dictionary structures and summarize non contiguous sets of cells, that are not necessarily part of arrays. In particular, [15, 16] seeks for a unified way to reason about pointers, scalars and arrays. These works are orthogonal to our approach, as we strive to use properties specific to arrays in order to reason about the structure of groups. Therefore, [15, 16] cannot express the invariants presented in Section 2 for two reasons: (1) the *access paths* cannot describe the contents of array elements as an interval or with other numeric constraints; (2) they cannot express *content-index* predicates. Similarly, HOO [13] is an effective abstract domain for containers and JavaScript open objects. As it uses a set abstract domain [14], it has a very general scope but does not exploit the structure of arrays, hence would sacrifice efficiency in such cases.

Last, template-base methods [3, 19] are very powerful invariant generation techniques, yet require user supplied templates and can be quite costly.

Our approach has several key distinguishing factors. First, it not only relies on index relations, but also exploits semantic properties of array elements, to select groups. Second, relation predicates track lightweight properties, that would not be captured in a numerical domain. Last, it allows empty groups, which eliminated the need for any global disjunction in our examples (a few assignments and tests benefit from cheap, local disjunctions). Finally, experiments show it is effective at inferring non trivial array invariants with non contiguous groups, and is able to verify the manipulation of two complex data-structures in two distinct operating systems.

Our analysis currently does not handle dynamically allocated arrays. However, that support could be added rather simply, by leaving the size of arrays abstract, to be represented by non-summarized dimensions in $n^{\mathbf{g}}$. Non-contiguous partitioning and group relation predicates enable our domain to verify memory safety invariants involving complex data structure in operating systems, which could not be achieved by previous static analyses [27, 33] which also target system code.

Finally, this paper extends [24] with a full description of the algorithms for the transfer functions and analysis operators and of their proofs of soundness, with a formalization of the analysis of a basic array language, and with a deeper experimental evaluations. In particular, the verification of the task manager table `taskq` is novel.

[1] Francesco Alberti, Silvio Ghilardi, and Natasha Sharygina. Definability of accelerated relations in a theory of arrays and its applications. In *Symposium on Frontiers of Combining Systems (FCS)*, pages 23–39, 2013.

[2] Francesco Alberti, Silvio Ghilardi, and Natasha Sharygina. Decision procedures for flat array properties. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 15–30, 2014.

[3] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Invariant synthesis for combined theories. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 378–394, 2007.

[4] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 196–207, 2003.

[5] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 427–442, 2006.

[6] Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 247–260, 2008.

[7] Bor-Yuh Evan Chang and Xavier Rival. Modular construction of shape-numeric analyzers. In *Semantics, Abstract Interpretation, and Reasoning about Programs (SAIRP)*, pages 161–185, 2013.

[8] Bor-Yuh Evan Chang, Xavier Rival, and George C. Necula. Shape analysis with structural invariant checkers. In *Static Analysis Symposium (SAS)*, pages 384–401, 2007.

[9] Liqian Chen, Antoine Miné, and Patrick Cousot. A sound floating-point polyhedra abstract domain. In *Asian Symposium on Programming Languages and Systems (APLAS)*, pages 3–18, 2008.

[10] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[11] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 269–282, 1979.

[12] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 105–118, 2011.

[13] Arlen Cox, Bor-Yuh Evan Chang, and Xavier Rival. Automatic analysis of open objects in dynamic language programs. In *Static Analysis Symposium (SAS)*, pages 134–150, 2014.

[14] Arlen Cox, Bor-Yuh Evan Chang, and Sriram Sankaranarayanan. QUIC graphs: Relational invariant generation for containers. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 401–425, 2013.

[15] Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid updates: Beyond strong vs. weak updates. In *European Symposium on Programming (ESOP)*, pages 246–266, 2010.

[16] Isil Dillig, Thomas Dillig, and Alex Aiken. Precise reasoning for programs using containers. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 187–200, 2011.

[17] Denis Gopan, Frank DiMaio, Nurit Dor, Thomas W. Reps, and Shmuel Sagiv. Numeric domains with summarized dimensions. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 512–529, 2004.

[18] Denis Gopan, Thomas W. Reps, and Shmuel Sagiv. A framework for numeric analysis of array operations. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 338–350, 2005.

[19] Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting abstract interpreters to quantified logical domains. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 235–246, 2008.

[20] Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 339–348, 2008.

[21] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Computer Aided Verification (CAV)*, pages 661–667, 2009.

[22] Ranjit Jhala and Kenneth L. McMillan. Array abstraction from proofs. In *Computer Aided Verification (CAV)*, pages 193–206, 2007.

[23] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over array using a theorem prover. In *Fundamental Approaches to Software Engineering (FASE)*, pages 470–485, 2009.

[24] Jiangchao Liu and Xavier Rival. Abstraction of arrays based on non contiguous partitions. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 282–299, 2015.

[25] Kenneth L. McMillan. Quantified invariant generation using an interpolation saturation prover. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 413–427, 2008.

[26] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19:31–100, 2006.

[27] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia L. Lawall, and Gilles Muller. Faults in linux: ten years later. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 305–318, 2011.

[28] Dana Scott. *Outline of a mathematical theory of computation.* Oxford University Computing Laboratory, Programming Research Group, 1970.

[29] Mohamed Nassim Seghir, Andreas Podelski, and Thomas Wies. Abstraction refinement for quantified array assertions. In *Static Analysis Symposium (SAS)*, pages 3–18, 2009.

[30] Pascal Sotin and Xavier Rival. Hierarchical shape abstraction of dynamic structures in static blocks. In *Asian Symposium on Programming Languages and Systems (APLAS)*, pages 131–147, 2012.

[31] Andrew S Tanenbaum, Albert S Woodhull, Andrew S Tanenbaum, and Andrew S Tanenbaum. *Operating systems: design and implementation*, volume 2. Prentice-Hall Englewood Cliffs, NJ, 1987.

[32] Antoine Toubhans, Bor-Yuh Evan Chang, and Xavier Rival. Reduced product combination of abstract domains for shapes. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 375–395, 2013.

[33] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In *Computer Aided Verification (CAV)*, pages 385–398, 2008.