Static Analysis on Numeric and Structural Properties of Array Contents

Jiangchao Liu

THÈSE DE DOCTORAT

de l'Université de recherche Paris Sciences et Lettres
PSL Research University

Préparée à L'École normale supérieure

Static Analysis on Numeric and Structural Properties of Array Contents

Ecole doctorale n°386

École Doctorale de Sciences Mathématiques de Paris Centre

Spécialité Informatique

COMPOSITION DU JURY :

M. GIACOBAZZI Roberto
Università di Verona, Rapporteur

M. PICHARDIE David
ENS Rennes, Rapporteur

M. BOURKE Timothy
CNRS/ENS/INRIA/PSL*, Membre du jury

M. CHEN Liqian
NUDT, Membre du jury

M. MULLER Gilles
LIP6/INRIA, Membre du jury

Mme. PUTOT Sylvie
Ecole Polytechnique, Membre du jury

M. RIVAL Xavier
CNRS/ENS/INRIA/PSL*, Membre du jury

Soutenue par Jiangchao LIU
le 20 Février 2018

Dirigée par **Xavier RIVAL**

Logo établissement PSL RESEARCH UNIVERSITY PARIS

Résumé

Dans cette thèse, nous étudions l'analyse statique par interprétation abstraites de programmes manipulant des tableaux, afin d'inférer des propriétés sur les valeurs numériques et les structures de données qui y sont stockées.

Les tableaux sont omniprésents dans de nombreux programmes, et les erreurs liées à leur manipulation sont difficile à éviter en pratique. De nombreux travaux de recherche ont été consacrés à la vérification de tels programmes. Les travaux existants s'intéressent plus particulièrement aux propriétés concernant les valeurs numériques stockées dans les tableaux. Toutefois, les programmes bas-niveau (comme les systèmes embarqués ou les systèmes d'exploitation temps-réel) utilisent souvent des tableaux afin d'y stocker des structures de données telles que des listes, de manière à éviter d'avoir recours à l'allocation de mémoire dynamique. Dans cette thèse, nous présentons des techniques permettant de vérifer par interprétation abstraite des propriétés concernant à la fois les données numériques ainsi que les structures composites stockées dans des tableaux.

Notre première contribution est une abstraction qui permet de décrire des stores à valeurs numériques et avec valeurs optionnelles (i.e., lorsqu'une variable peut soit avoir une valeur numérique, soit ne pas avoir de valeur du tout), ou bien avec valeurs ensemblistes (i.e., lorsqu'une variable est associée à un ensemble de valeurs qui peut être vide ou non). Cette abstraction peut être utilisée pour décrire des stores où certaines variables ont un type option, ou bien un type ensembliste. Elle peut aussi servir à la construction de domaines abstraits pour décrire des propriétés complexes à l'aide de variables symboliques, par exemple, pour résumer le contenu de zones dans des tableaux.

Notre seconde contribution est un domaine abstrait pour la description de tableaux, qui utilise des propriétés sémantiques des valeurs contenues afin de partitioner les cellules de tableaux en groupes homogènes. Ainsi, des cellules contenant des valeurs similaires sont décrites par les mêmes prédicats abstraits. De plus, au contraire des analyses de tableaux conventionnelles, les groupes ainsi formés ne sont pas nécessairement contigüs, ce qui contribue à la généralité de l'analyse. Notre analyse peut regrouper des cellules non-congitües, lorsque celles-ci ont des propriétés similaires. Ce domaine abstrait permet de construire des analyses complètement automatiques et capables d'inférer des invariants complexes sur les tableaux.

Notre troisième contribution repose sur une combinaison de cette abstraction des tableaux avec différents domaines abstraits issus de l'analyse de forme des structures de données et reposant sur la logique de séparation. Cette combinaison appelée coalescence opère localement, et relie des résumés pour des structures dynamiques à des groupes de cellules du tableau. La coalescence permet de définir de manière locale des algorithmes d'analyse statique dans le domaine combiné. Nous l'utilisons pour relier notre domaine abstrait pour tableaux et une analyse de forme générique, dont la tâche est de décrire des structures chaînées. L'analyse ainsi obtenue peut vérifier à la fois des propriétés de sûreté et des propriétés de correction fonctionnelle.

De nombreux programmes bas-niveau stockent des structures dynamiques chaînées

dans des tableaux afin de n'utiliser que des zones mémoire allouées statiquement. La vérification de tels programmes est difficile, puisqu'elle nécessite à la fois de raisonner sur les tableaux et sur les structures chaînées. Nous construisons une analyse statique reposant sur ces trois contributions, et permettant d'analyser avec succés de tels programmes. Nous présentons des résultats d'analyse permettant la vérification de composants de systèmes d'exploitation et pilotes de périphériques.

Abstract

We study the static analysis on both numeric and structural properties of array contents in the framework of abstract interpretation.

Since arrays are ubiquitous in most software systems, and software defects related to mis-uses of arrays are hard to avoid in practice, a lot of efforts have been devoted to ensuring the correctness of programs manipulating arrays. Current verification of these programs by static analysis focuses on numeric content properties. However, in some low-level programs (like embedded systems or real-time operating systems), arrays often contain structural data (e.g., lists) without using dynamic allocation. In this manuscript, we present a series of techniques to verify both numeric and structural properties of array contents.

Our first technique is used to describe properties of numerical stores with optional values (i.e., where some variables may have no value) or sets of values (i.e., where some variables may store a possibly empty set of values). Our approach lifts numerical abstract domains based on common linear inequality into abstract domains describing stores with optional values and sets of values. This abstraction can be used in order to analyze languages with some form of *option* scalar type. It can also be applied to the construction of abstract domains to describe complex memory properties that introduce symbolic variables, e.g., in order to summarize unbounded memory blocks like in arrays.

Our second technique is 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. Compared to conventional array partitioning analyses that split arrays into contiguous partitions to infer properties of sets of array cells. Our analysis can group together non-contiguous cells when they have similar properties. Our abstract domain can infer complex array invariants in a fully automatic way.

The third technique is used to combine different shape domains. This combination locally ties summaries in both abstract domains and is called a coalesced abstraction. Coalescing allows to define efficient and precise static analysis algorithms in the combined domain. We utilise it to combine our array abstraction (i.e., our second technique) and a shape abstraction which captures linked structures with separation logic-based inductive predicates. The product domain can verify both safety and functional properties of programs manipulating arrays storing dynamically linked structures, such as lists.

Storing dynamic structures in arrays is a programming pattern commonly used in low-level systems, so as to avoid relying on dynamic allocation. The verification of such programs is very challenging as it requires reasoning both about the array structure with numeric indexes and about the linked structures stored in the array. Combining the three techniques that we have proposed, we can build an automatic static analysis for the verification of programs manipulating arrays storing linked structures. We report on the successful verification of several operating system kernel components and drivers.

Résumé pour le Grand Public

Les erreurs logicielles peuvent avoir de graves conséquences humaines ou matérielles, en particulier dans le cas de systèmes critiques. De nombreux telles erreurs sont liées aux manipulations de tableaux. Un tableau peut être utilisé soit pour décrire des données numériques de base (par exemple la taille de chaque personne dans un groupe), ou bien pour stocker des structures dynamiques (comme une liste chaînée). De telles structures imbriquées sont fréquentes dans les systèmes d'exploitation, afin d'éviter un recours à l'allocation dynamique de mémoire. La correction de tels programmes est difficile à vérifier, en raison du grand nombre de cas possibles. Dans cette thèse, nous proposons plusieurs techniques permettant de vérifier la sûreté et la correction fonctionnelle de programme manipulant tous ces types de tableaux.

Notre première contribution permet de décrire des propriétés numériques d'ensembles vides ou non vides de valeurs, pouvant être de taille non bornée. Notre seconde contribution fournit un mécanisme de partition des tableaux en groupes contigüs ou non de cellules ayant des propriétés similaires. Enfin, notre troisième contribution permet de combiner une analyse de tableaux et une analyse de forme des structures de données, afin de décrire les structures imbriquées stockées dans les tableaux.

Nous avons mené une évaluation expérimentale sur des composants de systèmes d'exploitation, incluant le gestionnaire de tâches de TinyOS, et le gestionnaire de mémoire de Minix, ainsi que des pilotes de périphériques. Les résultats de cette évaluation démontrent que notre analyse est capable de traiter ce type de programmes avec succés.

Popular Science Summary

Software defects can cause huge economic losses, and even human deaths, particularly in safety-critical systems. Many of these defects are caused by the misuse of arrays. Array is one of the earliest aggregate data type. It can be used to collect plain data (e.g., the heights of all students in a class), or store dynamic linked structures (e.g., a linked list). The second case is often encountered in embedded systems, where dynamic memory allocation is not always allowed. The correctness of programs manipulating arrays can be hardly guaranteed, since the cases of array contents are enormous. In this thesis, we propose several techniques that can work together to verify some safety and functional properties of programs manipulating either kind of arrays.

Our first work can over-approximate the numeric properties of an unbounded set of array cells. Compared with existing work, it allows the set to be empty. Our second work provides a mechanism to partition an array into different groups. Compared with existing partition, ours can be more precise on arrays where cells with similar properties are not contiguous (e.g., dynamic structures in arrays often occupy non-contiguous regions). Our third work is a new way of combining different analyses. For instance, if we combine our analysis on arrays and an analysis on dynamic structures in this way, we can get an analysis on dynamic structures in arrays. Compared with existing work, this way is more precise on describing inter-wined data structures. We have conducted some experiments on operating system components, including the task scheduler in TinyOS, the memory management in Minix, etc. These experiments demonstrate the effectiveness of our analysis.

Acknowledgments

Many people to acknowledge. This part will come later.

Contents

Chapter 1

Correctness of Programs Using Arrays

As software is becoming ever-increasingly important in our society, software defects are also becoming a growing concern, particularly in safety-critical systems. Some accidents due to defects in software have caused huge economic losses, and even human deaths. In 1994 in Scotland, a Chinook helicopter crashed, killing all 29 of its passengers. Evidence [s:i02] showed that a software error had caused the crash. In 1996, a European Ariane 5 rocket veered off its path and exploded 37 seconds after launch, due to a defect in its software system. A 2002 study [Tas02] commissioned by the National Institute of Standards and Technology in the USA found that every year software defects cost the US economy 59.5 billion dollars.

Many of these defects are caused by the misuse of arrays. This chapter identifies defects that may be caused by the misuse of arrays and discusses to what extent existing techniques can help to avoid them.

1.1 Arrays and Their Usage

An array is an aggregate of data elements of the same type, where an individual element is identified by its position relative to the first element [Seb12]. The time complexity for accessing a random element in an array is only $\mathcal{O}(1)$. Another advantage of arrays regarding performance is low memory occupation, since basic array implementations do not need to store links to chain the structure, unlike linked lists or trees.

Arrays are one of the earliest and most important data structures in computer science. Their first appearance in high-level programming languages (like FORTRAN) dates back to the 1950s [All81]. Nowadays, arrays are still widely used in most software systems.

There are various types of arrays in programs. In this thesis, we classify them into three categories: arrays of scalar data, composite data and structured data.

Arrays of scalar data. This category collects arrays of primitive data types (e.g., integers), storing plain data. For example, a one-dimensional array storing the heights of all students in a class would fall into this category. The following C code declares such an array.

```
1 int height[10]={177,176,176,172,168,178,171,199,190,163,171};
```

Arrays of composite data. The second category comprises arrays that store plain data of composite data types. In the programming language C, a composite data type is declared with the keyword `struct` and is composed of a fixed set of labeled fields. The following code collects both the height and weight information of five students.

```
1 struct student {
2   int height;
3   int weight;
4 };
5 struct student five_students[] = {{177, 66},{176, 88}, {176,
     75}, {172, 77}, {168, 67} };
```

Arrays of structured data. The elements of arrays in this category could be of primitive data types or composite data types. The difference with the last two categories is that the stored data contains links (indexes or pointers) to other array elements or memory locations outside the array. These links may form dynamically chained structures (such as linked lists). For instance, in Hash tables, an imperfect hash function could generate the same index for different keys, which is called a hash collision. To resolve such collisions, programmers may use linked lists inside or outside the array recording data with the same hash indexes. In this thesis, we focus on the case where links are within arrays. Structured data is also often encountered in programs that avoid making use of dynamic memory allocation primitives like **malloc** in C. For instance, these include basic operating system services that maintain statically allocated structures and low-level embedded systems where the use of dynamic allocation is restricted for the sake of certification and more predictable execution time and memory usage.

Example 1.1 (A list in one array)**.** *Figure 1.1 shows a list (array cells in red) allocated in an integer array* `a`*. Array cells used as list nodes store the indexes of the next links of the list (e.g.,* $\mathtt{a}[1] = 3$*), and the tail node of the list stores -1 (*$\mathtt{a}[4] = -1$*).*

1.2 Correctness of Programs Using Arrays

In this section, we summarize the correctness properties that should be satisfied by programs, so as to avoid defects from misuse of arrays.

Safety properties express the absence of runtime-errors or undefined behaviors. In programs using arrays, the most important safety property is that "all array accesses must

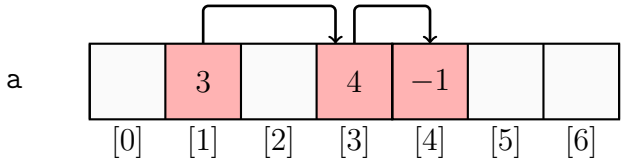


Figure 1.1 – A list in one array

be within the index range". Once the size of an array is fixed, the range of its indexes (i.e., subscripts) is also fixed. Array accesses beyond the index range may crash the program, or cause unexpected behaviors. Programming languages like Java or Pascal, perform dynamic bound checks on every array access. An out-of-bounds array access would raise an exception and abort the program. For example, the following code in Java would raise "ArrayIndexOutOfBoundsException".

```
1 int[] anArray = new int[26];
2 anArray[26] = 0;
```

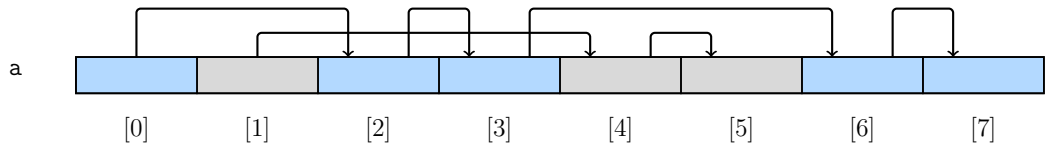
Out-of-bounds array accesses in programs using arrays of structured data are harder to avoid. This is because, in addition to scalar variables, array contents can also be used as indexes. Figure 1.1 shows a list inside an array, where -1 marks the tail of the list. All values stored in the list nodes could be used as valid indexes except the tail node. The following code pops one element from the head of the list, and before reading, it checks whether the index is -1.

```
1 if( head != -1 )
2   {
3     element = head;
4     head = a[head];
5   }
```

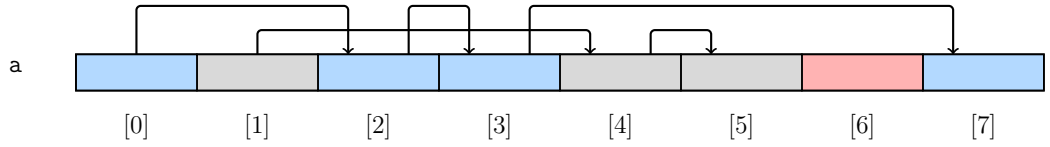
Besides out-of-bounds array accesses, defects that may occur on arrays of structured data include index leak and dangling nodes.

An index leak is a kind of resource leak. In some implementations, all the cells in an array are supposed to be linked by the structures inside the array. Index leak defect arises when some cells are leaked out of the linked structures, and could not be used again. Figure 1.2 shows two lists inside one array: one list stores user data, and the other manages free slots. If the program manipulating this array can not keep all array elements in the two lists, then the number of available array elements shrinks, which is also a form of memory leak. In Figure 1.2(b), a[6] is not reachable by the two lists and is thus leaked.

A dangling node defect arises when the chained structure in an array is broken in the middle. For instance, in the array in Figure 1.1, a[3] stores the index of the next list element (i.e., 4). If it stores any random value but not 4, then the list is broken and list node a[3] is dangling.



(a) A state without an index leak



(b) A state with an index leak

Figure 1.2 – A static memory pool

Security properties assert that non-authorized users are not able to corrupt any critical process or fetch confidential information. A well-known security exploit is buffer overflow. This means when a program writes data to a buffer, it exceeds the bounds of the buffer and overwrites the adjacent memory locations. This security exploit is often found on arrays of scalar and composite data. The following C code causes a buffer overflow, since the string to be copied exceeds the size of the buffer.

```
1 char buffer[10] = {0};
2 strcpy(buffer, "This string will overflow the buffer");
```

In many systems, the data area and executable code are arranged together as a whole. Thus a buffer overflow on the data area could possibly rewrite the executable code. Attackers could utilize this to send data designed to cause a buffer overflow and embed malicious code into the running software. If an operating system is attacked in this way, attackers could perform privilege escalation and gain full control of the computer. Morris, which was one of the first computer worms distributed via the Internet, is an instance of buffer overflow exploit.

Functional properties state that given a certain input, a program should produce an expected output. On programs using arrays of scalar and composite data, functional properties often boil down to universally quantified predicates on array contents. For instance, the functional property for an initialization program could be that all values stored in the output array should be 0. Violation of functional properties could cause incorrect results. As an example, let us consider the case where an array records the height information of all the students in a class, and stores -1 if the data for the corresponding student is absent. A program calculating the average height should take the possible incompleteness of data into account, otherwise the results would be invalid. The following code does not satisfy the following correctness property: variable `average` should store the average positive values in the array `height`.

```
1 int height[10]={177,176,176,172,168,178,171,-1,190,163,171};
```

```
int i, sum = 0, average;
for (i = 0; i < 10; i ++) sum += height[i];
average = sum / 10;
```

Some languages, like Python, provide libraries to deal with corner cases when processing data. The following code replaces any missing value with the average of the values in the column.

```
import pandas as pd
from sklearn.preprocessing import Imputer
from io import StringIO
csv_data = '''A,B,C,D
...1.0,2.0,3.0,4.0
...5.0,6.0,,8.0
...0.0,11.0,12.9,'''
df = pd.read_scv(StringIO(scv_data))
imr = Imputer(missing_values = 'NaN', strategy = 'mean', axis =
    0)
imr = imr.fit(df)
imputed_data = imr.transfrom(df.values)
```

Functional properties of programs that manipulate arrays of structured data are more complex, and may require inductively defined predicates to express them. For instance, the following code defines a task table, where the field `flag` indicates whether an array cell corresponds to a running task or a free slot, and the field `parent` stores the index of the parent task. One functional property of programs manipulating this array is that the `init` task is the direct or indirect ancestor of all the other tasks. This property is illustrated in Figure 1.3. Violation of this property could cause unexpected behavior. For instance, in some operating systems, a task could exit only when its parent task calls wait. If the parent of one task incorrectly points to itself, then it would never exit.

```
struct task{
  int flag;
  int parent;
}
struct task task_table[100];
```

The violation of this functional property could also cause safety issues. For instance, when the `parent` field of one task node stores -1, then the operating system would crash when a program tried to visit its parent task. This implies that checking basic safety properties may require checking more complex functional invariants.

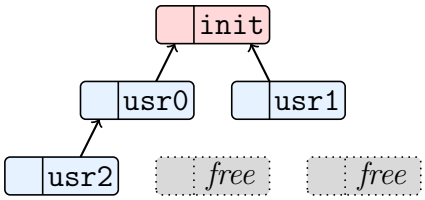


Figure 1.3 – Topology of the parent relations in a task table

1.3 Quality Control on Software Systems Using Arrays

Because of the severity of software defects, the quality of a software system greatly depends on whether it satisfies correctness properties (i.e., safety, security and functional properties). In this section, we go through the main methods that help improve the quality of software systems, particularly those using arrays.

Development guidelines. One way to improve software quality is to follow strict development guidelines. These guidelines avoid programming styles that are prone to defects. One popular set of guidelines is MISRA-C, which was developed by MISRA (Motor Industry Software Reliability Association) for the C programming language, and has been widely accepted in the industry.

MISRA-C includes several rules restricting the use of arrays, e.g., Rule 8.12 requires that when an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization. The following code shows both compliant and non-complicant cases.

```
1 extern int array1[10]; /* Compliant */
2 extern int array2[]; /* Not compliant */
3 int array3[] = {0, 1, 2, 3}; /* Compliant */
```

In safety-critical software systems, stricter guidelines are often adopted, like the prohibition of dynamic memory allocations (thus dynamically chained structures are implemented inside arrays in their software). Guidelines could be checked manually or automatically by tools like PC-lint. However, even if these rules are fully respected, it does not guarantee that a software system satisfies any correctness property.

Testing. Software testing is a classic and important way to find defects in software. It checks whether the correctness properties are satisfied by running the program with a certain set of inputs. Testing is, however, time consuming since it needs to run the program for each individual input.

Testing can verify any safety, security or functional properties of programs using arrays, on the tested inputs. However, because possible inputs are usually enormous, testing does

not guarantee that a software system satisfies correctness properties for all inputs. For instance, to fully verify the sortedness property on the following C program by testing, we need to run the program $2^{32\times100}$ times (supposing the size of an integer is 4 bytes).

```
1  int main(){
2    int array[100];
3    int n = 100;
4    int i, j, swap;
5    for( i = 0; i < n; i++) scanf("%d", &array[i]);
6    for( i = 0; i < n -1; i++)
7      for( j = 0; j < n - 1 - i; j++)
8        if(array[j] > array[j+1]){
9          swap = array[j];
10         array[j] = array[j + 1];
11         array[j + 1] = swap;
12       }
13 }
```

Formal verification. The previous two methods could effectively reduce the rate of potential defects in a software system, but provide no guarantee. Formal verification could prove that a program satisfies given correctness properties, and is free of certain types of defects. Formal verification was proposed in the late 1960s [BL68, Flo67, Hoa69] and various techniques have been developed for different applications. In industry, Microsoft has been utilizing SLAM [BCLR04] to verify its drivers for Windows; Airbus has successfully verified its avionics software on the A380. Moreover, many companies like AbsInt have been founded to provide formal verification services.

Most formal verification methods include two steps: formalizing the property of interest and carrying out the proof.

Properties are often expressed by languages of logical formulas. Some famous languages include Hoare Logic [Hoa69] which uses a triple to describe how the execution of a piece of code changes the state of the computation; Computation Tree Logic (CTL) [CE82] which is a branching-time temporal logic describing safety and liveness properties; Communicating Sequential Processes (CSP) [BHR84] which is based on process algebras, describing patterns of interaction in concurrent systems.

On programs using arrays of scalar data and composite data, the safety property "all array accesses must be within the index range" is often expressed by simple inequality relations on program variables. Functional properties on such arrays are more complex, since they require quantification over values stored in the array. Separation logic is an extension of Hoare logic, which is based on the separating conjunction "$*$" ($P * Q$ means P and Q hold for separate portions of the memory) and provides a modular description of memory states. In [BDES12], the authors combine separation logic and first-order logic to describe functional properties on arrays of scalar and composite data.

Proving by hand is time consuming since proofs are typically huge. Thus, interactive

proof assistants like Coq and Isabelle have been developed to ease the proving process. However, even with these tools, the time spent on the proof of a program is usually much longer than that spent on its actual development.

Automatic formal verification. One idea to make formal verification more practical is to automate the proving process. However, most mainstream programing languages are Turing complete. Depending on the underlying logic, the validity of the property of interest on a Turing complete language could be *undecidable* or NP complete. Thus it is impossible to design a fully automatic verification algorithm for all classes of properties on these languages. However, it is possible to design an automatic algorithm that achieves sub-goals of formal verification. For instance, some algorithms only have the *soundness* property (if the verification terminates and returns "true", then the program satisfies the property), or the *completeness* property (if the program satisfies the property, then the verification terminates and returns "true").

Automated theorem proving attempts to produce a formal proof automatically, with a description of the system, a set of logical axioms and a set of inference rules. This method is sound and complete with respect to the specification. However, it can only be applied a small class of programs.

Tools presented in [AGS13, AGS14, BMS06] identify a class of programs manipulating arrays where the transitive closures are definable, and can verify safety and functional properties that could be expressed in first-order logic.

Model checking verifies a given set of state machines with respect to a set of temporal formula with SAT-solving methods. Difficulties in this approach include synthesizing the model from the program and avoiding the "state explosion" problem.

The work by [MG16, MA15] can transform any array program into an array free program or a system of Hoare clauses. But the transformation only keeps universally quantified properties and scalability is limited by the back-end SMT solvers. Fluid updates in [DDA10, DDA11] seek for a unified way to reason about pointers, scalars and arrays. They represent a group of array cells by both under- and over-approximation. However, they cannot represent the contents of array elements as an interval or other numeric relations by abstraction, which could possibly lead to state explosion. The properties that can be verified by these methods are also limited to safety and functional properties of programs using arrays of scalar and composite data.

Static analysis by abstract interpretation represents a possibly infinite set of concrete states at a program point by a finite set of *abstract states*, and over-approximates concrete semantics of the given program soundly by *abstract semantics*. This kind of static analysis computes properties that describe all reachable states of the program automatically without performing all executions. A class of abstract states with corresponding abstract semantics is called an *abstract domain*.

Numeric abstract domains [CH78b, CLM$^+$14] can discover complex numeric relations on program variables, compute precise invariants on array subscripts, and verify the absence of out-of-bounds accesses on arrays of scalar and composite data. Array partitioning

approaches [BCC+03b, GRS05, HP08, CCL11] assume that array cells with similar properties are contiguous and partition arrays into segments according to numeric properties. Non-structured functional properties like value ranges, sortedness inside each segment can be verified automatically.

1.4 Goal of the Thesis

Existing work can verify correctness properties on a class of programs that use arrays of scalar and composite data automatically. However, several difficulties prevent them from addressing arrays of structured data.

- Dynamically chained structures (or for short, dynamic structures) usually occupy non-contiguous regions of arrays, making it impossible to extract dynamic structures out from arrays using a partition based on contiguous segments;
- The numerical properties of array contents is non-trival to describe, especially when dynamically chained structures are embedded in arrays;
- Since the dynamic structures and arrays are intertwined, verification algorithms need to reason about the accesses into dynamic structures via both their next links and array indexes at the same time.

This thesis presents a series of techniques in the framework of static analysis by abstract interpretation, to addree these three challenges. Combining these techniques can verify safety and functional properties on programs using arrays of structured data automatically.

In Chapter 2, we give an overview of these techniques and an outline of this thesis.

Chapter 2

Contributions of the Thesis

This chapter overviews the contributions of the thesis. To give an intuitive idea of our techniques, we first consider an industrial example that manipulates an array of structured data. Then, we show the challenges that must be overcomed to verify its safety and functional properties and how our static analysis addresses them.

2.1 Towards Verifying Safety and Functional Properties of Arrays of Structured Data

2.1.1 An Example: the Task Scheduler in TinyOS

TinyOS [LMP+04] is an embedded, application-specific operating system designed for sensor networks. In TinyOS 2.x, the task scheduler is configurable, and the default configuration follows the FIFO strategy. In this scheduler, the queue of tasks is maintained by a list stored in an array, as shown in Figure 2.1. The main data structure is an array `m_next` of 256 cells of type **unit8_t**. An array cell either is unused (in which case it stores value 255), or it stores the index of the next link in the list. Value 255 is also used as a special value to indicate the end of the list. The indexes of the head and the tail of the list are stored in variables `m_head` and `m_tail` respectively. When the list is empty, both variables store 255.

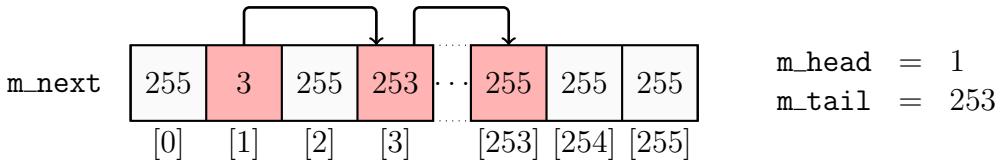
Figure 2.1(b) shows a segment of a concrete state, in which cells 1, 3 and 253 correspond to running tasks, and all the other cells are free slots. Functions that manipulate this structure mainly include `push_Task` (push a task to the tail of the list), `pop_Task` (pop a task from the head of the list) and `tinit` (initialize the array).

2.1.2 Safety and Functional Properties

In this subsection, we investigate the safety and functional properties of the `pop_Task` function. Such properties of the other two functions are similar. Figure 2.2 shows a code

```
unit8_t  m_head;
unit8_t  m_tail;
unit8_t  m_next[255];
```

(a) Definition of the data structure



(b) A concrete state

Figure 2.1 – A list in one array

```
1 inline uint8_t popTask(){
2  if (m_head != 255){
3     unit_t id = m_head;
4     m_head = m_next[m_head];
5  // The rest of this function is ignored
```

Figure 2.2 – A code segment from `pop_Task`

segment extracted from the `pop_Task` function. It checks whether the task list is empty and if not, it stores the head index in variable `id` and the next link of the head node in `m_head`.

The safety property that should be satisfied by this function is that "there is no out-of-bounds array access". If this property is violated, the program may crash. Moreover, we are interested in the following functional properties: all running tasks must be chained by the list led by `m_head`. This property prevents unexpected scheduler behaviors, such as tasks becoming dangling.

The verification of both properties is non-trivial. For instance, before the array access `m_next[m_head]` at line 4, the bounds of the `m_head` variable are not checked explicitly. To prove the safety property on this statement, we need a global description of the value of `m_head`. This is not simple because the value of the `m_head` variable could be updated by array contents (e.g., at line 4). Thus a description of the global invariant (i.e., invariant between system calls) on the contents of the `m_next` array is necessary in order to verify the safety property. As for functional properties, the global invariant is also needed and should be precise enough to describe dynamically chained structures.

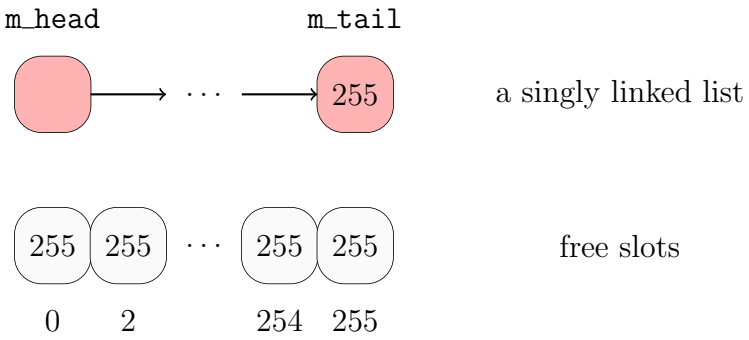


Figure 2.3 – Topology of the structural properties

2.1.3 Structural Invariant

From the previous subsection, we see that global invariant plays an essential role in proving safety and functional properties of the three system calls. In the TinyOS 2.x scheduler, the global invariant is informally described as follows.

- Variables `m_head` and `m_tail` should store indexes of the head and the tail of a well-formed acyclic list in the `m_next` array, where the value in each list node is the index of the next element, and the end-of-list is encoded by index 255. The list could be empty when both `m_head` and `m_tail` are equal to 255;
- Each cell that denotes a free slot stores the value 255.

We denote this invariant as $\mathcal{R}^{tinyos}$. Its topology is shown in Figure 2.3. The property that $\mathcal{R}^{tinyos}$ always holds between system calls, actually entails both safety and functional properties. Thus, we would like our static analysis to verify the preservation of $\mathcal{R}^{tinyos}$ by each system call. That involves verifying that:

- The `tinit` function establishes invariant $\mathcal{R}^{tinyos}$, which means that the assertion shown below should hold.

$$\texttt{tinit}();\ \textbf{assert}(\mathcal{R}^{tinyos});$$

- The `push_Task` and `pop_Task` functions preserve $\mathcal{R}^{tinyos}$. This means that, if the pre-condition satisfies $\mathcal{R}^{tinyos}$, then the post-condition should also satisfy it. If we take `push_Task`, for example, the assertion shown below should hold.

$$\textbf{assume}(\mathcal{R}^{tinyos});\ \texttt{push\_Task}(\texttt{id});\ \textbf{assert}(\mathcal{R}^{tinyos});$$

Remark 2.1. *In our static analysis, the global invariants should be provided by the users. This is a compromise to the fact that it is hard to infer global invariants from programs, especially when the global invariants are as complex as $\mathcal{R}^{tinyos}$, and it is even harder to guarantee that the inferred global invariants match the users' intention. However, the*

global invariants provided by users are only used as assumptions on preconditions and assertions on post-conditions of system calls. All invariants in the programs (including loop invariants) are inferred automatically by our analysis.

2.1.4 Challenges for Verifying the Invariant and Solutions from the Thesis

To verify $\mathcal{R}^{tinyos}$ with static analysis by abstract interpretation, we need to construct a lattice of abstract states, each of which is an abstraction of the set of concrete states satisfying a certain property, e.g., $\mathcal{R}^{tinyos}$. Our target structural invariant $\mathcal{R}^{tinyos}$ includes both shape conditions (e.g., the acyclic list is embedded in the array) and numeric conditions (e.g., all free slots store 255). Abstracting both conditions is non-trivial.

Abstraction of the shape conditions. In the `m_next` array, the cells used as list nodes and free slots interleave with each other, thus before abstracting the dynamically chained list, the abstraction has to first partition list nodes and free slots into disjoint groups.

- *Abstraction by non-contiguous partitioning.* As shown in Figure 2.1(b), the list occupies a non-contiguous region of the `m_next` array. Existing partitioning techniques only split arrays into contiguous segments, thus they could not group together non-contiguous cells, even when they have similar properties (e.g., they are all list nodes or free slots). Our *non-contiguous partitioning* utilizes semantic information to split the array into groups of cells that are not necessarily contiguous, thus list nodes and free slots can be packed into different groups. Additionally, these groups could possibly be empty, which is consistent with the fact that invariant $\mathcal{R}^{tinyos}$ allows the list to be empty.

- *Coalescing with a shape abstraction.* Our non-contiguous partitioning can split all list nodes in the array out to form a group. The dynamically chained structure (i.e., list) is still not expressed. Since the list and the array are intertwined, we propose to combine our non-contiguous partitioning with a shape abstraction which can capture linked structures. As existing combination techniques are not precise enough to express the shape conditions of $\mathcal{R}^{tinyos}$, we propose a combination method called *coalescing*. This combination locally ties predicates from both abstractions, and is precise enough to express all shape conditions in $\mathcal{R}^{tinyos}$.

Abstraction of the numeric conditions. Conventional numeric abstractions describe sets of points in a multi-dimensional Euclidean space. In one concrete state (i.e., a point in the space), each dimension is assigned one value. These numeric abstractions are able to express numeric properties of scalar program variables, with one dimension corresponding to one scalar program variable. However, the property $\mathcal{R}^{tinyos}$ requires that all array cells

that are used as free slots store the value 255. If we use one dimension to represent all values stored in a group, then this dimension maps to a possibly empty set of values. This is beyond the expressiveness of existing numeric abstractions. We propose a functor called Maya+ that can lift conventional numeric abstractions to those that are able to describe all numeric conditions in $\mathcal{R}^{tinyos}$.

Analysis. After we have designed the abstraction of both shape and numeric conditions, we have to implement the abstract predicates with computer representation and design algorithms to enable automatic reasoning about the abstract predicates. In the following chapters, we will give the formal definition of the algorithms to compute *sound* invariants at each program point, which are precise enough to prove the preservation of $\mathcal{R}^{tinyos}$.

The following sections present the basic description of our main contributions: non-contiguous partitioning, coalescing, Maya+ functor and a dynamic packing method.

2.2 Non-contiguous Partitioning

The first step in our abstraction is to distinguish list nodes and free slots in the array. Array expansion and array partitioning are the main existing methods that can split arrays.

2.2.1 Limitations of Array Expansion and Array Partitioning

Array expansion and its limitations. Given an array of scalar type, *array expansion* represents each array cell with an individual scalar variable. Consider the following array of size 8:

```
1 int a[8];
```

Program analysis using array expansion would create 8 integer variables to represent all the cells in the array `a`. It has the same effect as rewriting the program as follows.

```
1 int a0, a1, a2, a3, a4, a5, a6, a7;
```

Array expansion suffers from severe performance loss since it introduces many variables and each variable should correspond to a unique dimension in the numeric abstract domains. Another disadvantage of this method is that, when dealing with structural invariants like $\mathcal{R}^{tinyos}$, the disjunctions that these introduced scalar variables form a chained structure are enormous, thus a huge amount of disjuncts have to be created, just to express $\mathcal{R}^{tinyos}$.

Contiguous partitioning and its limitations. *Contiguous array partitioning* was first proposed by Gopan, Reps and Sagiv [GRS05] in 2005. It splits an array into contiguous and disjoint *groups*, and summarizes array contents in each group separately. The boundaries between groups are scalar variables that are used as indexes and are selected

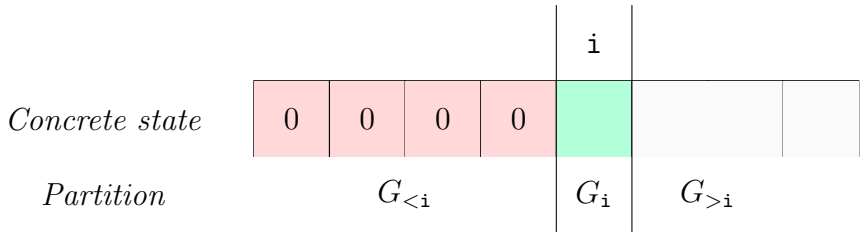


Figure 2.4 – Contiguous array partitioning

by a syntactic pre-analysis. Figure 2.4 illustrates a concrete state with an integer array `a` and a possible partition. Since `i` is the only variable used as an index of that array, array `a` is partitioned into groups $G_{<\mathtt{i}}$, $G_{\mathtt{i}}$ and $G_{>\mathtt{i}}$. Group $G_{<\mathtt{i}}$ (resp. $G_{>\mathtt{i}}$) collects all the array cells, whose indexes are less (resp. greater) than `i`; group $G_{\mathtt{i}}$ contains only one array cell `a[i]`.

Dynamic contiguous partitioning [CCL11] relies on semantic information to decide the bounds of each group and does not need a syntactic pre-analysis. A bound is described by a simple expression of program variables and constants. Another difference with [GRS05] is that dynamic contiguous partitioning allows empty groups, which reduces the number of disjuncts, when the upper/lower bounds of a group are possibly equal. The concrete state in Figure 2.4 could by partitioned by [CCL11] to $\{0\}0\{\mathtt{i}, 4\}\top\{8\}$. This means $\mathtt{i} = 4$, and the values stored in array cells of indexes $[0, \mathtt{i})$ and $[\mathtt{i}, 8)$ are 0 and unknown respectively.

Contiguous partitioning techniques [CCL11, GRS05, HP08] perform well on arrays where cells with similar properties are contiguous. However, when cells that have similar properties are not contiguous, these approaches cannot infer adequate array partitions. They would fail to abstract invariant $\mathcal{R}^{tinyos}$ where both groups of list nodes and free slots are non-contiguous. The reason is obvious: there is no bound on groups of either list nodes or free slots, since they interleave with each other.

2.2.2 Non-contiguous Partitioning Based on Semantics

In the thesis, we propose a dynamic and semantic non-contiguous partitioning domain. It does not fully rely on index bounds to perform partitioning, but also makes use of numeric and structural properties on array contents. To give a brief idea about this technique, let us take the concrete state in Figure 2.1 as an example. In the non-contiguous partitioning domain, an abstract state could partition this array into three groups, which are called G_0, G_1, G_2. Group G_0 collects all list nodes except the tail node; group G_1 contains only the tail node; and G_2 accounts for all free slots. This partitioning is described by two parts: memory predicates and numeric predicates, as shown in Figure 2.5.

An atomic memory predicate $G_0@0 \mapsto I_{G_0}$ describes the indexes and contents in group G_0. All the indexes in this group are abstracted by a *set variable* (i.e., symbolic variables representing sets of values), denoted as G_0 (the same as its name). We use set variables because these groups are possibly non-contiguous, and the indexes in each group could not

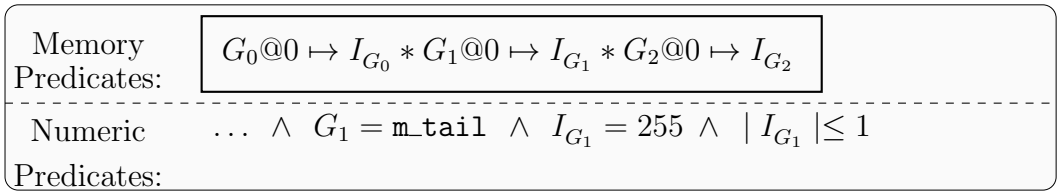


Figure 2.5 – Non-contiguous partitioning on the `m_next` array

be precisely represented by lower/upper bounds. The contents in this group are described by $0 \mapsto I_{G_0}$, where 0 indicates the offset and set variable I_{G_0} abstracts the values in this group at offset 0.

Numeric predicates on set variables are used to characterize each group. For instance, numeric predicate $I_{G_2} = 255$ means that all values in set variable I_{G_2} are equal to 255 or I_{G_2} is an empty set (in the thesis, usual value equality "=" is used to denote this meaning in numeric predicates). This characterizes the fact that free slots in the array store the value 255. Set variables may also support predicates such as cardinality (e.g., $|G| = 1$ means that the size of set G is 1). The predicates on G_1 ($G_1 = \texttt{m\_tail} \wedge I_{G_1} = 255 \wedge |I_{G_1}| \leq 1$) imply that group G_1 is either empty or contains one cell, the index and value of which are equal to `m_tail` and 255 respectively. For simplicity, we ignore some numeric predicates in Figure 2.5. Some numeric predicates in our non-contiguous partitioning domain (e.g., $I_{G_2} = 255$) could not described by conventional numeric domains and rely on the technique introduced in Section 2.4.

The formal definition of this domain including abstract semantics will be introduced in Chapter 5.

2.3 Coalescing Array and Shape Abstractions

There are various abstract domains, targeting different classes of properties. For instance, numeric domains could describe valuation of numeric variables in a system, and shape domains could express structural invariants on the memory in a system. Abstract interpretation allows different abstract domains to be combined in order to extend their expressiveness. The ASTRÉE analyzer [BCC+03a] combines different numeric domains to gain additional precision in numeric analysis. Shape domains can also be combined [LYP11, TCR13], for instance, to abstract overlaid data structures. Since the array structure and list structure are intertwined in the property $\mathcal{R}^{tinyos}$, we consider combining our non-contiguous partitioning domain with a shape domain that could describe dynamically chained structures like lists. Several kinds of combination techniques have been introduced in previous studies [CCF13, LYP11, TCR13]. In this section, we first review them and then present our coalescing domain.

2.3.1 Existing Combination Techniques

Combination of numeric domains. One way of combining numeric domains is the reduced product [CC77]. This combination expresses the logical conjunction of abstract elements from distinct domains, and refines logical statements on both sides of the conjunction. It has been successfully used the ASTRÉE analyzer [BCC+03a].

The Donut domain [GIB+12] is used to express non-convex numeric properties. It takes two elements from convex numeric domains as inputs: the first one defines an over-approximation of all the possible concrete valuations; the second one under-approximates the set of impossible valuations. The geometrical concretization of the combination is defined by a convex set minus another convex set, that is, the difference between the two convex sets.

Reduced cardinal power [CC79] expresses a conjunction of implications. It allows the analysis to track disjunctive information, like "when `x` is 3, `y` is 4 and when `x` is 2, `y` is 1".

Combination of shape domains. The combination of shape domains is more challenging than that of numeric domains, since there exists a much greater diversity in the way of abstracting memory predicates relating to concrete memory states. The works in [LYP11, TCR13, RTC14] combine shape domains with the Cartesian product, i.e., abstract elements from two distinct shape domains are connected with the logical conjunction. Their techniques can be used to abstract overlaid data structures.

Another kind of combination of shape domains is hierarchical domain [SR12]. In this combination, the shape domain that is used in order to abstract the whole memory is with contiguous "nodes" (or sub-memory) of arbitrary size, and predicates from other shape domains could be attached to these "nodes". This combination can describe structures nested into abstractions of memory blocks.

2.3.2 The Coalescing Domain and a Comparison with Existing Techniques

In this thesis, we propose a *coalescing* domain, which provides an efficient and precise method to combine different shape domains. Our coalescing domain requires the input domains to be based on separating conjunction. Separating conjunction (denoted as $*$) is introduced in Separation Logic [Rey02]. It asserts that the memory locations described by the conjuncts are disjoint. The non-contiguous partitioning domain is in this category, since the partitioned groups are disjoint. The coalescing domain describes separating conjunctions of local conjunctions of atomic predicates from two distinct domains. Applying coalescing on our non-contiguous partitioning domain and a shape domain that can abstract dynamically linked structures, the combination domain can express structural invariants on array contents like $\mathcal{R}^{tinyos}$.

To compare the main idea in our coalescing domain with other combinators, we show the structures of formulas expressed in reduced product, hierarchical domain and our

coalescing domain in Figure 2.6. We assume that the memory predicates from the two input domains are of the form $(M_0 * \ldots * M_m)$ and $(N_0 * \ldots * N_n)$, where M_i and N_i are atomic memory predicates and $*$ denotes the separating conjunction.

- In a Cartesian product, the memory predicates are of the form $(M_0 * \ldots * M_m) \wedge (N_0 * \ldots * N_n)$. A non-separating conjunction (i.e., $\wedge$) is applied on separating conjunctions of atomic predicates. Here the non-separating conjunction $\wedge$ means the conjuncts constrain the same memory locations. Note that the memory predicates $(M_0 * \ldots * M_m)$ and $(N_0 * \ldots * N_n)$ from two input domains do not necessarily partition the memory identically, thus it is hard to find relations between atomic memory predicates from two abstractions. This limits the expressiveness of the Cartesian product.

- In a hierarchical domain, the memory predicates in an abstract element are of the form $(M_0 * \ldots * M_m) * N_0 * \ldots * N_n$, where $(M_0 * \ldots * M_m)$ must describe a contiguous memory region. This limitation makes it only able to describe structures stored in a non empty and contiguous sub-memory. This is not the case discussed in Section 2.1, where a list could occupy a non-contiguous region in an array.

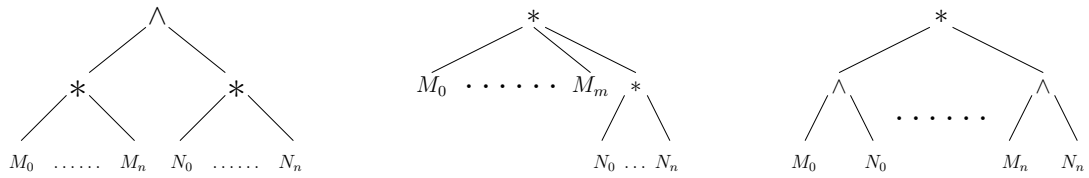
- In a coalescing domain, a non-separating conjunction is first applied on atomic memory predicates and a global separating conjunction is applied on conjunctions of atomic predicates. The memory predicates are of the form $(M_0 \wedge N_0 * \ldots * M_n \wedge N_n)$. It enforces the identical memory partition by two abstractions and builds a correspondence between atomic memory predicates from two domains. Since the non-separating conjunction is applied on the atomic level, information can be easily exchanged between atomic memory predicates, thus the coalescing domain is more expressive than the Cartesian product.

Let us suppose we coalesce the non-contiguous partitioning with a separation logic based shape domain which supports a structural predicate $\mathbf{lseg}(\alpha, \beta)$. This predicate denotes that a set of memory locations forms a list segment where the addresses of the head node and the next link stored in the tail node are abstracted by symbolic scalar variables α and β respectively. We let the predicate **true** denote all possible concrete stores. The memory predicates for structural invariant $\mathcal{R}^{tinyos}$ are shown below.

$$G_0@0 \mapsto I_{G_0} \wedge \mathbf{lseg}(\alpha, \beta) * G_1@0 \mapsto I_{G_1} \wedge \mathbf{true} * G_2@0 \mapsto I_{G_2} \wedge \mathbf{true}$$

This memory predicate indicates that the memory locations abstracted by G_0 and $\mathbf{lseg}(\alpha, \beta)$ are identical. This could not be expressed by a Cartesian product.

The coalescing domain is fully introduced in Chapter 6.



(a) Cartesian product (b) Hierarchical domain (c) Coalescing

Figure 2.6 – Comparison of Cartesian product with coalescing

2.4 Maya+ Functor

In static analysis by abstract interpretation, numeric properties are usually described by numeric abstract domains. In this section, we first review the existing numeric abstract domains and their expressiveness. Then we give the main idea of our Maya+ functor. A domain *functor* is a function that produces new abstract domains based on input domains, in order to extend their expressiveness, or to define a new abstract domain to express entirely different properties.

2.4.1 Numeric Domains and Their Expressiveness

Conventional numeric abstract domains [Kar76, CC77, Min06, CH78a, CLM+14] approximate and reason about sets of points in a multi-dimensional Euclidean space. A point is defined by a value for each dimension. Formally, the valuation over a set of dimensions is of the form $\mathbb{E} \to \mathbb{V}$, where $\mathbb{E}$ denotes the set of dimensions and $\mathbb{V}$ denotes the set of values. If we usually use one dimension to represent one scalar program variable, these numeric domains could be used to express numeric properties on concrete stores of the form $\mathbb{X} \to \mathbb{V}$, where $\mathbb{X}$ denotes the set of scalar program variables. However, when designing analysis for complex data-structures or languages like OCaml , the concrete stores may not be of the form $\mathbb{X} \to \mathbb{V}$. In the following, we show where these cases arise and to what extent the existing approachs can address them.

Abstraction of possibly empty memory locations. Many programming languages feature *possibly empty* memory locations. For instance, OCaml and Scala have an `option` type. This type can be defined by `type 'a option = None | Some of 'a`, which means a value of type `int option` may either be an integer, or undefined, represented by `None`. Similarly, spreadsheet environments feature empty cells as well as an `empty` type.

Another case where optional variables may arise is in programs with dynamic allocation. For instance, in the following code, the memory location at `*x` may be undefined or store the integer value 3.

```
1 if (random()){
2   x = malloc(4);
3   *x = 3;
```

```
4  } else {
5    x = NULL;
6  }
```

When each program variable contains either one value or no value, the concrete stores are of the form $\mathbb{X} \to \{\ominus\} \uplus \mathbb{V}$ where $\ominus$ stands for "no value". The conventional abstract domains mentioned above fail to describe such stores. Therefore, they need to be extended with dimensions which may be undefined in order to deal with optional variables.

The solution proposed in [SMS13] adds a flag $\mathtt{f_d}$ for each dimension $\mathtt{d}$, such that $\mathtt{f_d} = 1$ if $\mathtt{d}$ is defined and $\mathtt{f_d} = 0$ otherwise. It does not support relational predicates between undefined dimensions and defined dimensions, which limits its application. For instance, this domain could not infer that $\mathtt{d}$ represents no value from a set of constraints that show no value can be found for $\mathtt{d}$. This situation is commonly needed in our array analysis since our analysis sometimes needs to infer that a group is empty: for instance, when the constraints over its bounds are not satisfiable, the group is necessarily empty.

Abstraction of non-empty sets of memory locations. When designing analysis for complex data-structures, a common technique is to *summarize* sets of memory locations together. For instance, in array partitioning, the values of the cells in a group are described by a single *set variable* (i.e., variables storing sets of values). Many forms of array partitioning do not allow empty groups [GDD+04, HP08]. The concrete stores they describe are of the form $\mathbb{X} \to (\mathcal{P}(\mathbb{V})/\emptyset)$. *Numeric domains with summarized dimensions* [GDD+04] could lift conventional numeric domains to those constraining dimensions, whose valuation is of the form $\mathbb{E} \to (\mathcal{P}(\mathbb{V})/\emptyset)$.

Abstraction of possibly empty sets of memory locations. Our non-contiguous partitioning domain allows empty groups (Section 2.2.2). Thus a set variable that abstracts array indexes in a group maps to a possibly empty set of values. Thus the concrete stores are of the form $\mathbb{X} \to \mathcal{P}(\mathbb{V})$. To the best of our knowledge, no work addresses this kind of stores.

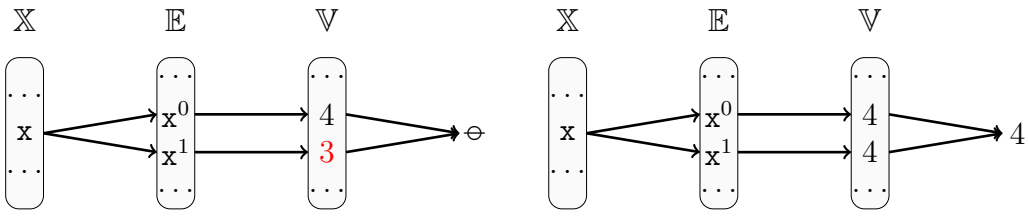
Overall, by surveying the existing work, we conclude that (1) the numeric domains that could abstract possibly empty memory locations are not precise enough for our analysis (e.g., [SMS13] could not infer that a dimension represents no value due to unsatisfiable constraints); (2) there does not exist a numeric domain that abstracts possibly empty sets of memory locations, which is needed in our analysis.

2.4.2 Maya+ Functor

In this thesis, we propose a general functor: Maya+ lifts conventional numeric abstract domains to those manipulating predicates on dimensions, each of which may map to a possibly empty set of values. Maya+ is actually a combination of two functors: the Maya

Store	Each Numeric Object	General Solutions
$\mathbb{X} \to \mathbb{V}$	one value	Numeric Abstract Domains
$\mathbb{X} \to \mathbb{V} \cup \{\ominus\}$	one value or no value	Maya
$\mathbb{X} \to (\mathcal{P}(\mathbb{V})/\emptyset)$	more than one value	Gopan et al [GDD+04]]
$\mathbb{X} \to \mathcal{P}(\mathbb{V})$	any number of values	Maya+

Figure 2.7 – Abstraction of numeric relations on different dimensions



(a) The case x maps to no value (b) The case x maps to one value

Figure 2.8 – Main idea in Maya domain

functor and the functor of summarized dimensions [GDD+04]. The first one enables the parameter numeric domain to describe numeric objects (e.g., program variables or set variables) representing one value or no value; the second one enables the parameter numeric domain to describe numeric objects representing a non-empty set of values.

The applications of Maya/Maya+ functors and existing numeric domains are shown in Figure 2.7. Now we introduce the main idea of Maya. The combination of the two functors is introduced in Chapter 4.

Main idea in Maya functor. In a Maya functor, a variable $\mathtt{x}$ representing one or zero value is represented by a set of dimensions $\mathtt{x}^0, \ldots, \mathtt{x}^k$ in the parameter numeric domain. We call these dimensions $\mathtt{x}^0, \ldots, \mathtt{x}^k$ *avatars*. The functor assumes that $\mathtt{x}$ can be defined if and only if all its avatars may map to a common value. For instance, if the constraints on the avatars of $\mathtt{x}$ are $\mathtt{x}^0 \geq 4 \wedge \mathtt{x}^1 \leq 3$. Then $\mathtt{x}$ necessarily maps to no value, because no value assignment can map $\mathtt{x}^0$ and $\mathtt{x}^1$ to the same value. This case arises when our array analysis infers that the indexes of a group is less than 3 and greater than 4, which implies that this group is empty. The principle in our Maya functor can be applied to any numeric abstract domain where abstract values are finite conjunctions of constraints (the vast majority of numeric abstract domains are of that form). The graphical illustration of this idea is shown in Figure 2.8, where $\mathbb{E}$ represents the set of dimensions in the parameter numeric domain.

2.5 Outline of the Thesis

In Chapter 3, we define a simple language, and introduce the main idea of abstract interpretation based on this language. The following three chapters present details of the techniques discussed in the last three sections, but in a different order. Maya+ functor (Chapter 4) is first presented since it provides a numeric basis for the non-contiguous partitioning domain (Chapter 5). The coalescing domain is formalized in Chapter 6. chapter 7 evaluates our work by attempting the verification of some components of several operating systems.

Chapter 3

Static Analysis by Abstract Interpretation

In this chapter, we introduce the main ideas of static analysis by abstract interpretation. To formalize this technique, we first define a simple imperative language, then present the formalization based on this language. This chapter cannot cover all aspects of this technique. More details can be found on [CC77] and [CC79].

3.1 A Simple Imperative Language

3.1.1 Syntax and Notations

Figure 3.1 shows the syntax of a simple imperative language. It could be seen as a subset of the C programming language. This language is only used to formalize the main idea of static analysis by abstract interpretation, and does not enjoy all features of our target programs. Later chapters will extend it as needed.

This language supports only one *primitive type*: machine integers **int**. Type **int** corresponds to the set of all integers in the interval $[-2^{31}, 2^{31} - 1]$ (i.e., 32 bits), which is denoted as $\mathbb{V}$. We do not have any Boolean type. Instead, in conditionals, non-zero integers and zero play the roles of "true" and "false" respectively, as in the C programming language. We let $\mathbb{X}$ denote the set of variables in a program.

At this stage, an left-value l (expression that evaluates into a memory location) can only be an integer type variable. However, it will be expanded with other forms (e.g., array accesses) in the following chapters. An r-value r (expression that evaluates into a value or a set of values) could be a constant, an l-value or unary/binary operators applied on r-values. A program is made of statements. A statement **p** could be an assignment, a skip instruction, a sequence of statements, an assertion statement, a conditional branching or an loop. All these statements are classical and defined in a standard way. Note that, we do not support dynamic memory allocation in our language, because it is not the focus of this thesis.

Types

int	::=	$[-2^{31}, 2^{31}-1]$		*primitive type: integer*

Operators

$\oplus$	::=	`+` | `-` | `*` | `/` | `-` | ...		*arithmetic operator*
$\bowtie$	::=	`!` | `||` | `&&` | `==` | `!=` | ...		*logical operator*

Left value expressions

l	::=	x	($\mathtt{x} \in \mathbb{X}$)	*primitive type variables*

Right value expressions

r	::=			
	|	v	($v \in \mathbb{V}$)	*value*
	|	l		*l-value read*
	|	$\oplus(\mathtt{r}, \ldots, \mathtt{r})$		*arithmetic operations on right values*
	|	$\bowtie(\mathtt{r}, \ldots, \mathtt{r})$		*logical operations on right values*

Statements

p	::=		
	|	$\mathtt{l} = \mathtt{r}$	*assignment*
	|	**skip**	*skip*
	|	**assert**(r)	*assert*
	|	**p** ; **p**	*sequence*
	|	**if**(r){ **p** }**else**{ **p** }	*conditional branching*
	|	**while**(r){ **p** }	*loop*

Figure 3.1 – Grammar of a simple imperative language.

3.1.2 Denotational Semantics

One common semantics for describing the behavior of transition systems is the *denotational semantics* [Sco70]. It formalizes the meaning of program statements as functions mapping initial states into final states. To define this semantics for our language, we need first to define concrete states.

Concrete state. A *concrete state* is a memory state $\sigma \in \mathbb{S}$, that maps variables to values. Thus, $\mathbb{S}$ is defined by $\mathbb{S} = \mathbb{X} \to \mathbb{V}$, or $\mathbb{S} = \mathbb{V}^m$ when the set of variables $\mathbb{X}$ is fixed, where $m = \mid \mathbb{X} \mid$ is a non-negative integer.

Concrete semantics. The denotational semantics of this language is defined in Figure 3.2. This semantics does not explicitly characterize the non-terminating executions and run-time errors, however it could trivially be extended into a semantics that collects the set of *all* reachable states and has a special "error" state. We make the choice to

Evaluation of L-values: $\mathbb{S} \to \mathbb{X}$

$$\mathbf{evalL}[\![\mathtt{x}]\!](\sigma) \;=\; \mathtt{x}$$

Evaluation of R-values: $\mathbb{S} \to \mathbb{V}$

$$\begin{array}{rclrcl} \mathbf{evalR}[\![\mathtt{x}]\!](\sigma) & = & \sigma(\mathtt{x}) & \mathbf{evalR}[\![\oplus(\mathsf{r},\ldots,\mathsf{r})]\!](\sigma) & = & \oplus([\![\mathsf{r}]\!](\sigma),[\![\mathsf{r}]\!](\sigma)) \\ \mathbf{evalR}[\![v]\!](\sigma) & = & v & \mathbf{evalR}[\![\bowtie(\mathsf{r},\ldots,\mathsf{r})]\!](\sigma) & = & \bowtie([\![\mathsf{r}]\!](\sigma),[\![\mathsf{r}]\!](\sigma)) \end{array}$$

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

$$\mathbf{guard}[\![\mathsf{r}]\!](\mathcal{S}) \;=\; \{\sigma \in \mathcal{S} \mid \mathbf{evalR}[\![\mathsf{r}]\!](\sigma) \neq 0\}$$

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

$$\begin{array}{rcl} \mathbf{stat}[\![\mathbf{skip}]\!](\mathcal{S}) & = & \mathcal{S} \\ \mathbf{stat}[\![\mathbf{assert}(\mathsf{r})]\!](\mathcal{S}) & = & \mathbf{guard}[\![\mathsf{r}]\!](\mathcal{S}) \\ \mathbf{stat}[\![\mathsf{l}=\mathsf{r}]\!](\mathcal{S}) & = & \{\sigma[[\![\mathsf{l}]\!](\sigma) \mapsto \mathbf{evalR}[\![\mathsf{r}]\!](\sigma)] \mid \sigma \in \mathcal{S}\} \\ \mathbf{stat}[\![\mathbf{p}_0;\mathbf{p}_1]\!](\mathcal{S}) & = & \mathbf{stat}[\![\mathbf{p}_1]\!] \circ \mathbf{stat}[\![\mathbf{p}_0]\!](\mathcal{S}) \\ \mathbf{stat}[\![\mathbf{if}(\mathsf{r})\{\mathbf{p}_0\}\mathbf{else}\{\mathbf{p}_1\}]\!](\mathcal{S}) & = & \mathbf{stat}[\![\mathbf{p}_0]\!] \circ \mathbf{guard}[\![\mathsf{r}]\!](\mathcal{S}) \cup \mathbf{stat}[\![\mathbf{p}_1]\!] \circ \mathbf{guard}[\![!\mathsf{r}]\!](\mathcal{S}) \\ \mathbf{stat}[\![\mathbf{while}(\mathsf{r})\{\mathbf{p}\}]\!](\mathcal{S}) & = & \mathbf{guard}[\![!\mathsf{r}]\!] \circ (\mathbf{lfp}_{\subseteq}\mathbf{F}) \\ & & \quad \textit{where} \quad \mathbf{F}(\mathcal{S}') = \mathcal{S} \cup \mathbf{stat}[\![\mathbf{p}]\!] \circ \mathbf{guard}[\![\mathsf{r}]\!](\mathcal{S}') \end{array}$$

Figure 3.2 – Denotational semantics of a simple imperative language.

use an "angelic" denotational semantics so as to make the formalization simpler. The evaluation of l-values $\mathbf{evalL}[\![\mathsf{l}]\!] : \mathbb{S} \to \mathbb{X}$ maps an l-value expression to a memory location represented by a primitive type variable. The evaluation of r-values $\mathbf{evalR}[\![\mathsf{r}]\!] : \mathbb{S} \to \mathbb{V}$ maps an r-value expression to a value. A condition test $\mathbf{guard}[\![\mathsf{r}]\!] : \mathcal{P}(\mathbb{S}) \to \mathcal{P}(\mathbb{S})$ filters out the concrete states that do not satisfy the condition expressed by r. A transformer $\mathbf{stat}[\![\mathbf{p}]\!] : \mathcal{P}(\mathbb{S}) \to \mathcal{P}(\mathbb{S})$ maps a set of initial states before the execution of $\mathbf{p}$ to the set of final states after the execution. The transformers on our language are defined in a standard way by induction over the syntax of the statements. One tricky part is the transformer for loop statements.

Intuitively, the denotational semantics of an loop statement $\mathbf{stat}[\![\mathbf{while}(\mathsf{r})\{\mathbf{p}\}]\!](\mathcal{S})$ collects all concrete states that could be obtained from finite iterations of $\mathbf{stat}[\![\mathbf{p}]\!] \circ \mathbf{guard}[\![\mathsf{r}]\!]$ on $\mathcal{S}$ and satisfy the condition $\mathbf{guard}[\![!\mathsf{r}]\!]$. The set of reachable concrete states after at most ith iterations can be computed by $\mathbf{F}^i(\emptyset)$, where

$$\mathbf{F}(\mathcal{S}') ::= \mathcal{S} \cup \mathbf{stat}[\![\mathbf{p}]\!] \circ \mathbf{guard}[\![\mathsf{r}]\!](\mathcal{S}')$$

It is easy to know $\mathbf{F}$ is monotonic (i.e., $\mathbf{F}^0() \subseteq \mathbf{F}^1() \subseteq \mathbf{F}^2() \subseteq \ldots$) and Scott-continusous (i.e., it preserves the supremum of a chain), thus according to Kleene's theo-

rem [KdBdGZ52], function $\mathbf{F}$ has a least fix-point as

$$\mathbf{lfp}_{\subseteq}\mathbf{F} = \bigcup_{\mathtt{i}\in\mathbb{N}} \mathbf{F}^{\mathtt{i}}(\emptyset)$$

Overall, the denotational semantics of the loop statement is $\mathbf{guard}[\![!\mathtt{r}]\!] \circ (\mathbf{lfp}_{\subseteq}\mathbf{F})$ as shown in Figure 3.2. In most cases, the concrete semantics is not adequate for automatic reasoning, since it is infinite and not decidable. In the following section, we will introduce a sound and decidable semantics in the framework of static analysis by abstract interpretation, to compute properties of programs.

3.2 Abstract Interpretation

Abstract interpretation [CC77] is a mathematical theory to compare program semantics. *Static analysis* is one of its applications.

We call a lattice of concrete elements as a *concrete domain*. In the framework of abstract interpretation, the behavior of a concrete domain can be over-approximated by an *abstract domain*. If the concrete domain is the power-set of concrete states $(\mathcal{P}(\mathbb{S}), \subseteq)$, then a corresponding abstract domain specifies a sound static analysis. An *abstract domain* is a class of *abstract elements* with *abstract operators* defined on them.

3.2.1 Abstract Elements

An abstract domain includes a partially ordered set of abstract elements $(\overline{\mathcal{S}}, \sqsubseteq)$. The relationship between abstract elements and concrete elements is defined by a concretization function. To simplify the formalization, we assume that the concrete domain is the power-set of concrete states in this chapter, which is not always the case.

Definition 3.1 (Concretization function)**.** *Given a partially ordered set $(\overline{\mathcal{S}}, \sqsubseteq)$ of abstract elements, and a complete lattice $(\mathcal{P}(\mathbb{S}), \subseteq)$ of concrete elements, a concretization function $\gamma : \overline{\mathcal{S}} \to \mathcal{P}(\mathbb{S})$ should satisfy the following condition.*

$$\forall \overline{\mathtt{c}}_0, \overline{\mathtt{c}}_1 \in \overline{\mathcal{S}}, \quad \overline{\mathtt{c}}_0 \sqsubseteq \overline{\mathtt{c}}_1 \Rightarrow \gamma(\overline{\mathtt{c}}_0) \subseteq \gamma(\overline{\mathtt{c}}_1)$$

Two special abstract elements are $\bot \in \overline{\mathcal{S}}$ and $\top \in \overline{\mathcal{S}}$. They correspond to the empty set of concrete elements ($\gamma(\bot) = \emptyset$) and the set of all concrete elements ($\gamma(\top) = \mathbb{S}$) respectively.

Example 3.1 (The polyhedra abstract domain)**.** *One popular abstract domain focusing on numeric properties of programs is the convex polyhedra abstract domain (or the polyhedra domain for short). An abstract element in the polyhedra domain is a conjunction of linear inequalities. There are several representations for the polyhedra domain [CH78a, SK05]. For simplicity, we use the representation in [SK05] which is based only on constraint*

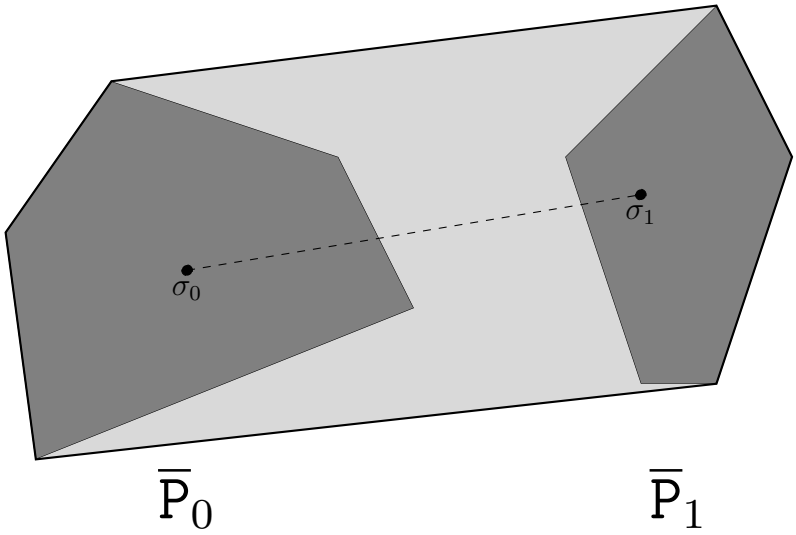


Figure 3.3 – The join of two convex polyhedra

representations. An abstract element in this domain is of the form $\overline{\mathtt{P}} = \{\tilde{\mathtt{M}}\vec{x} \leq \vec{b}\}$ *where* $\tilde{\mathtt{M}} \in \mathbb{I}^{m\times n}, \vec{b} \in \mathbb{I}^m$ *and* $\vec{x} \in \mathbb{X}^m$*. The concretization is defined below.*

$$\gamma^{Poly}(\overline{\mathtt{P}}) = \{\sigma \in \mathbb{S} \mid \tilde{\mathtt{M}}\sigma \leq \vec{b}\}$$

3.2.2 Abstract Operators

An abstract operator over-approximates a basic computation in the concrete level. Abstract operators usually include lattice operators and abstract transformers.

Abstract lattice operations. Abstract lattice operations include abstract join and inclusion checking. Abstract join $\sqcup : \overline{\mathcal{S}} \times \overline{\mathcal{S}} \Rightarrow \overline{\mathcal{S}}$ is an over-approximation of concrete union $\cup$, and computes an over-approximation of the least upper bound of two abstract elements. Its soundness is defined as follows.

Definition 3.2 (Soundness of abstract join)**.**

$$\forall \overline{\mathtt{c}}_0, \overline{\mathtt{c}}_1 \in \overline{\mathcal{S}}, \quad \gamma(\overline{\mathtt{c}}_0) \cup \gamma(\overline{\mathtt{c}}_1) \subseteq \gamma(\overline{\mathtt{c}}_0 \sqcup \overline{\mathtt{c}}_1)$$

Example 3.2 (Abstract join in the polyhedra domain)**.** *In the domain of convex polyhedra, each element* $\overline{\mathtt{P}}$ *represents a convex in a Euclidean space. The least upper bound of two elements* $\overline{\mathtt{P}}_0$ *and* $\overline{\mathtt{P}}_1$ *is their convex hull. To compute the convex hull, the basic idea is to construct the convex combination of all points in* $\overline{\mathtt{P}}_0$ *and* $\overline{\mathtt{P}}_1$*. That is, if* $\sigma_0 \in \gamma^{Poly}(\overline{\mathtt{P}}_0)$ *and* $\sigma_1 \in \gamma^{Poly}(\overline{\mathtt{P}}_1)$*, then* $\lambda_0\sigma_0 + \lambda_0\sigma_1 \in \gamma^{Poly}(\overline{\mathtt{P}}_0 \sqcup \overline{\mathtt{P}}_1)$*, where* $\lambda_0, \lambda_1 \geq 0 \wedge \lambda_0 + \lambda_1 = 1$*. This idea is illustrated in Figure 3.3. Following this idea, let* $\overline{\mathtt{P}}_0 = \{\tilde{\mathtt{M}}_0\vec{x} \leq \vec{b}_0\}$ *and* $\overline{\mathtt{P}}_1 = \{\tilde{\mathtt{M}}_1\vec{x} \leq \vec{b}_1\}$*, the join operator [SK05] is defined as follows.*

$$\overline{\mathsf{P}}_0 \sqcup \overline{\mathsf{P}}_1 = \left\{ \begin{array}{c} \exists \lambda_0, \lambda_1 \in \mathbb{V}, \vec{y}_0, \vec{y}_1 \in \mathbb{V}^m \\ \vec{x} = \vec{y}_0 + \vec{y}_1 \ \wedge \ \lambda_0 + \lambda_1 = 1 \ \wedge \ \lambda_0 \geq 0 \\ \tilde{\mathtt{M}}_0 \vec{y}_0 \leq \lambda_0 \vec{b}_0 \ \wedge \ \tilde{\mathtt{M}}_1 \vec{y}_1 \leq \lambda_1 \vec{b}_1 \ \wedge \ \lambda_1 \geq 0 \end{array} \right.$$

However, this operator introduces redundant variables $\lambda_0, \lambda_1, \vec{y}_0$ and $\vec{y}_1$. They can be projected out by Fourier-Motzkin elimination [Sch98], which is a mathematical algorithm for eliminating variables from a sysstem of linear inequalities.

The inclusion checking operator $\sqsubseteq: \overline{\mathcal{S}} \times \overline{\mathcal{S}} \Rightarrow \{\mathbf{true}, \mathbf{false}\}$ over-approximates $\subseteq$, and checks the ordering of two abstract elements. Its soundness is defined as follows.

Definition 3.3 (Soundness of abstract inclusion checking)**.**

$$\forall \overline{\mathsf{c}}_0, \overline{\mathsf{c}}_1 \in \overline{\mathcal{S}}, \quad \overline{\mathsf{c}}_0 \sqsubseteq \overline{\mathsf{c}}_1 \Rightarrow \gamma(\overline{\mathsf{c}}_0) \subseteq \gamma(\overline{\mathsf{c}}_1)$$

Example 3.3 (Abstract inclusion checking in the polyhedra domain)**.** *In the polyhedra domain, the inclusion checking $\overline{\mathsf{P}}_0 \sqsubseteq_{Poly} \overline{\mathsf{P}}_1$ is implemented by checking whether all inequalities in $\overline{\mathsf{P}}_1$ are implied by $\overline{\mathsf{P}}_0$: $\forall \varphi \in \overline{\mathsf{P}}_1, \overline{\mathsf{P}}_0 \models \varphi$. The implication checking is performed by Linear Programming. Given an inequality $\sum_i c_i \mathtt{x}_i \leq b$, Linear Programming can compute the maximal value μ of $\sum_i c_i \mathtt{x}_i$ subject to $\overline{\mathsf{P}}_0$. If $\mu \leq b$, then the implication relation holds.*

Abstract transfer functions. Abstract transfer functions over-approximates basic transitions in the concrete domain. These operators include guard and assignment.

Transfer function $\overline{\mathbf{guard}}[\mathsf{r}] : \overline{\mathcal{S}} \Rightarrow \overline{\mathcal{S}}$ over-approximates $\mathbf{guard}[\![\mathsf{r}]\!]$ in the concrete domain, which filters out all concrete states that do not satisfy condition r. Its soundness is defined as follows.

Definition 3.4 (Soundness of abstract guard)**.**

$$\forall \overline{\mathsf{c}} \in \overline{\mathcal{S}}, \quad \mathbf{guard}[\![\mathsf{r}]\!](\gamma(\overline{\mathsf{c}})) \subseteq \gamma(\overline{\mathbf{guard}}[\mathsf{r}](\overline{\mathsf{c}}))$$

Example 3.4 (Abstract guard in the polyhedra domain)**.** *In the polyhedra domain, if the condition to be tested r is of the form $\sum_i c_i \mathtt{x}_i \leq b$, then guard function $\overline{\mathbf{guard}}[\mathsf{r}]^{Poly}(\overline{\mathsf{P}})$ just adds r into $\overline{\mathsf{P}}$. Note that the added inequality r may be redundant, thus an inclusion checking $\overline{\mathsf{P}} \sqsubseteq \{\mathsf{r}\}$ is first performed to justify whether $\{\mathsf{r}\}$ is implied by $\overline{\mathsf{P}}$. If $\overline{\mathsf{P}} \sqsubseteq \{\mathsf{r}\}$ holds, then $\overline{\mathbf{guard}}[\mathsf{r}]_{Poly}(\overline{\mathsf{P}}) = \overline{\mathsf{P}}$. When the condition r is not a linear inequality, the method in [Min04a] could transfer it into the linear inequality form, and ensures that soundness still holds.*

Abstract transfer function $\overline{\mathbf{assign}}[\mathsf{l} = \mathsf{r}] : \overline{\mathcal{S}} \Rightarrow \overline{\mathcal{S}}$ over-approximates the assignment $\mathbf{stat}[\![\mathsf{l} = \mathsf{r}]\!]$ in the concrete domain. Its soundness is defined as follows.

Definition 3.5 (Soundness of abstract assignment)**.**

$$\forall \overline{\mathsf{c}} \in \overline{\mathcal{S}}, \quad \mathbf{stat}[\![\mathsf{l} = \mathsf{r}]\!](\gamma(\overline{\mathsf{c}})) \subseteq \gamma(\overline{\mathbf{assign}}[\mathsf{l} = \mathsf{r}](\overline{\mathsf{c}}))$$

Example 3.5 (Abstract assignment in the polyhedra domain)**.** *In the polyhedra domain, we only consider assignments of the form* $\mathtt{v}_0 = \sum_{i\in[1,n]} c_i\mathtt{x}_i + c_0$*, since assignments of other forms could be transferred into this form with the method in [Min04a] by over-approximation. The transfer function for assignment in the polyhedra domain* $\overline{\mathbf{assign}}[\mathtt{x}_0 = \sum_{i\in[1,n]} c_i\mathtt{x}_i + c_0]^{Poly}$ *just creates a fresh variable* $\mathtt{x}$*, and performs guard function* $\overline{\mathbf{guard}}[\mathtt{x} = \sum_{i\in[1,n]} c_i\mathtt{x}_i + c_0]^{Poly}$*, then project* $\mathtt{x}_0$ *by Fourier-Motzkin elimination and rename* $\mathtt{x}$ *to* $\mathtt{x}_0$*.*

Abstract least fix-point. Static analysis should not only be sound but also terminating. Especially for loop statements, which involve the computation of the abstract least fix-points.

We can get the abstract least fix-point operator in abstract domain by induction on it definition $\mathbf{lfp}_{\subseteq}\mathbf{F} = \bigcup_{\mathtt{i}\in\mathbb{N}} \mathbf{F}^{\mathtt{i}}(\mathcal{S})$ in the concrete domain, as follows.

$$\begin{cases} \overline{\mathtt{c}}^0 = \overline{\mathtt{c}}, \textit{where } \overline{\mathtt{c}} \textit{ is the initial state} \\ \overline{\mathtt{c}}^{i+1} = \overline{\mathtt{c}}^i \sqcup \overline{\mathbf{F}}(\overline{\mathtt{c}}^i), \textit{where } \overline{\mathbf{F}} \textit{ is an over-approximation of } \mathbf{F} \end{cases}$$

The computation for abstract least fix-points terminates when $\overline{\mathtt{c}}^{i+1} \sqsubseteq \overline{\mathtt{c}}^i$.

However, this computation does not guarantee termination or may be too slow to terminate. In static analysis by abstract interpretation, the abstract join is usually replaced by widening $\triangledown : \overline{\mathcal{S}} \times \overline{\mathcal{S}} \to \overline{\mathcal{S}}$ for the computation of abstract least fix-points. It bears the soundness of abstract join and guarantees fast termination.

Example 3.6 (Widening in the polyhedra domain)**.** *In the polyhedra domain, one definition of the widening operator* $\triangledown_{Poly}$ *is as follows.*

$$\overline{\mathtt{P}}_0 \triangledown^{Poly} \overline{\mathtt{P}}_1 = \{\varphi \in \overline{\mathtt{P}}_0 \mid \overline{\mathtt{P}}_1 \models \varphi\}$$

This definition aggressively removes all inequalities that are not implied (i.e., $\models$*, as defined in Example 3.4) by* $\overline{\mathtt{P}}_1$*. Its termination is guaranteed by the fact that the number of inequalities in the initial state is finite.*

3.2.3 Abstract Semantics

Abstract semantics is an over-approximation of concrete semantics in the sense of reachable states. It assigns denotations to the program in an abstract domain. In static analysis by abstract interpretation, abstract semantics is defined with abstract operators in abstract domains. For our target language in Figure 3.1, its abstract semantics $\overline{\mathbf{stat}}[.] : \overline{\mathcal{S}} \to \overline{\mathcal{S}}$ could be defined as Figure 3.4.

Example 3.7 (Abstract semantics in the polyhedra domain)**.** *If we instantiate the abstract semantics in Figure 3.4 with the polyhedra domain, then we get a static analysis that can automatically find linear inequality relations in our target program. Now we show how static analysis works on the following example.*

$$
\begin{array}{rcl}
\overline{\mathbf{stat}}[\mathbf{skip}](\overline{\mathsf{c}}) & = & \overline{\mathsf{c}} \\
\overline{\mathbf{stat}}[\mathbf{assert}(\mathsf{r})](\overline{\mathsf{c}}) & = & \overline{\mathbf{guard}}[\mathsf{r}](\overline{\mathsf{c}}) \\
\overline{\mathbf{stat}}[\mathsf{l} = \mathsf{r}](\overline{\mathsf{c}}) & = & \overline{\mathbf{assign}}[\mathsf{l} = \mathsf{r}](\overline{\mathsf{c}}) \\
\overline{\mathbf{stat}}[\mathbf{p}_0; \mathbf{p}_1](\overline{\mathsf{c}}) & = & \overline{\mathbf{stat}}[\mathbf{p}_1] \circ \overline{\mathbf{stat}}[\mathbf{p}_0](\overline{\mathsf{c}}) \\
\overline{\mathbf{stat}}[\mathbf{if}(\mathsf{r})\{\mathbf{p}_0\}\mathbf{else}\{\mathbf{p}_1\}](\overline{\mathsf{c}}) & = & \overline{\mathbf{stat}}[\mathbf{p}_0] \circ \overline{\mathbf{guard}}[\mathsf{r}](\overline{\mathsf{c}}) \cup \overline{\mathbf{stat}}[\mathbf{p}_1] \circ \overline{\mathbf{guard}}[!\mathsf{r}](\overline{\mathsf{c}}) \\
\overline{\mathbf{stat}}[\mathbf{while}(\mathsf{r})\{\mathbf{p}\}](\overline{\mathsf{c}}) & = & \overline{\mathbf{guard}}[!\mathsf{r}]\overline{\mathbf{lfp}}_{\subseteq}\overline{\mathbf{F}}(\overline{\mathsf{c}}) \\
 & & \text{where} \overline{\mathbf{F}} = \overline{\mathbf{stat}}[\mathbf{p}] \circ \overline{\mathbf{guard}}[\mathsf{r}]\}
\end{array}
$$

Figure 3.4 – An abstract semantics of the target language

```
1 int x = 0;
2 int y = 0;
3 while(x < 10){
4   x = x + 1;
5   y = y + 2;
6  }
```

The initial abstract state is $\overline{\mathsf{c}}_1 = \top$ *(the subscript indicates that it is the state before the statement at line 1), which represents all possible concrete states. After the following two assignments, the abstract state is* $\overline{\mathsf{c}}_3^0 = \{\mathtt{y} = \mathtt{x} \wedge \mathtt{x} = 0\}$ *(we write* $\mathtt{x} = \mathtt{y}$ *for short of* $\mathtt{y} \leq \mathtt{x} \wedge \mathtt{x} \leq \mathtt{y}$*, which is the real representation in the polyhedra domain). Here the subscript (i.e., 3) and superscript (i.e., 0) of* $\overline{\mathsf{c}}_3^0$ *indicate that this state is before the statement at line 3 after 0 iteration. The two statements in the loop body produce abstract state* $\overline{\mathsf{c}}_6^0 = \{\mathtt{y} = 2\mathtt{x} \wedge \mathtt{x} = 1\}$*. A widening operation is needed to compute the pre-condition for next iteration as* $\overline{\mathsf{c}}_3^1 = \overline{\mathsf{c}}_3^0 \nabla \overline{\mathsf{c}}_6^0$*. The implementation of widening introduced in this Chapter will produce* $\overline{\mathsf{c}}_3^1 = \{\mathtt{y} = 2\mathtt{x}\}$*.*

Chapter 4

Maya and Maya+ Functors

In this thesis, we call numeric domains for Euclidean space as *conventional numeric domains*. We have shown in Chapter 2 the limitations of conventional numeric domains to describe optional variables and summarized memory locations. In this chapter, we formalize Maya and Maya+ functors. Maya functor lifts conventional numeric domains to those abstracting numeric objects storing optional values, and Maya+ functor lifts conventional numeric domains to those abstracting possibly empty sets of values. In the following sections, we first illustrate Maya functor and then combine it with summarized dimensions functor [GDD+04] to produce Maya+ functor.

4.1 Extension of Our Language

To give a context for our Maya and Maya+ functors, we extend the syntax and semantics of the language defined in Chapter 3. The features of the this language are not found in common programming languages. Instead, they are meant to support the demonstraction of Maya and Maya+. The syntax and semantics are mostly similar to those in Chapter 3. The key difference is shown in Figure 4.1.

Types and notations. Machine integer type is replaced by the four following types (these types are also collected in Figure 4.2).

- A variable of *standard* integer type $\mathbf{int}^\bullet$ stores a machine integer (same as machine integer type defined in Chapter 3), here we add a $\bullet$ on the superscript, just to make it consistent with other types. Variables of this type are denoted as $\mathtt{y}^\bullet \in \mathbb{Y}^\bullet$.
- A variable of *optional* integer type $\mathbf{int}^?$ represents one integer or no value, and is denoted as $\mathtt{y}^? \in \mathbb{Y}^?$.
- A variable of *non-empty summary* integer type $\mathbf{int}^+$ represents a non-empty set of integers, and is denoted as $\mathtt{y}^+ \in \mathbb{Y}^+$.

Syntax

$\mathbf{eint}$	$::=$	$\mathbf{int}^{\bullet}\vert\mathbf{int}^{?}\vert\mathbf{int}^{+}\vert\mathbf{int}^{*}$		*Extended integer types*
l	$::=$	$\mathsf{y}^{\bullet}$	$(\mathsf{y}^{\bullet} \in \mathbb{Y}^{\bullet})$	*Standard variable*
	$\vert$	$\mathsf{y}^{?}$	$(\mathsf{y}^{?} \in \mathbb{Y}^{?})$	*Optional variable*
	$\vert$	y^{+}	$(\mathsf{y}^{+} \in \mathbb{Y}^{+})$	*Non-empty summary variable*
	$\vert$	y^{*}	$(\mathsf{y}^{*} \in \mathbb{Y}^{*})$	*Possible-empty summary variable*
r	$::=$	v	$(v \in \mathbb{V})$	*Constant*
	$\vert$	l		*l-value read*
	$\vert$	$\oplus(\mathsf{r},\ldots,\mathsf{r})$		*Arithmetic Expression*
	$\vert$	$\bowtie(\mathsf{r},\ldots,\mathsf{r})$		*Logical expression*
	$\vert$	$\mathbf{is\_empty}(\mathsf{y}^{?*})$	$(\mathsf{y}^{?*} \in \mathbb{Y}^{?} \cup \mathbb{Y}^{*})$	*Empty test*

Semantics

$$
\begin{aligned}
\mathbf{evalR}[\![\mathsf{y}]\!](\sigma) &= \sigma(\mathsf{y}) \\
\mathbf{evalR}[\![\mathbf{is\_empty}(\mathsf{y})]\!](\sigma) &= \begin{cases} 1 & \mathbf{evalR}[\![\mathsf{y}]\!](\sigma) = \emptyset \\ 0 & \mathbf{evalR}[\![\mathsf{y}]\!](\sigma) \neq \emptyset \end{cases} \\
\mathbf{evalR}[\![\oplus(\mathsf{r}_0,\ldots,\mathsf{r}_k)]\!](\sigma) &= \begin{cases} \oplus([\![\mathsf{r}]\!](\sigma),[\![\mathsf{r}]\!](\sigma)) & \forall i \in [0,k],\ [\![\mathsf{r}_i]\!](\sigma) \in \mathbb{V} \\ \emptyset & \exists i \in [0,k],\ [\![\mathsf{r}_i]\!](\sigma) = \emptyset \end{cases} \\
\mathbf{evalR}[\![\bowtie(\mathsf{r}_0,\ldots,\mathsf{r}_k)]\!](\sigma) &= \begin{cases} \{1\} & \exists c_i \in [\![\mathsf{r}_i]\!](\sigma), i \in [0,k], \bowtie(c_0,\ldots,c_k) = 1 \\ \emptyset & \exists i \in [0,k],\ [\![\mathsf{r}_i]\!](\sigma) = \emptyset \\ \{0\} & \textit{otherwise} \end{cases} \\
\mathbf{guard}[\![\mathsf{r}]\!](\mathcal{S}) &= \{\sigma \in \mathcal{S} \mid \mathbf{evalR}[\![\mathsf{r}]\!](\sigma) \neq \{0\} \vee \mathbf{evalR}[\![\mathsf{r}]\!] = \emptyset\} \\
\mathbf{stat}[\![\mathsf{y}^{\bullet?} = \mathsf{r}]\!](\mathcal{S}) &= \{\sigma[\mathsf{y}^{\bullet?} \mapsto [\![\mathsf{r}]\!](\sigma)] \mid \sigma \in \mathcal{S}\} \\
\mathbf{stat}[\![\mathsf{y}^{+*} = \mathsf{r}]\!](\mathcal{S}) &= \{\sigma[\mathsf{y}^{+*} \mapsto [\![\mathsf{r}]\!](\sigma) \cup \sigma(\mathsf{y}^{+*})] \mid \sigma \in \mathcal{S}\}
\end{aligned}
$$

Figure 4.1 – Extension of the language

- A variable of *possibly-empty summary* integer type $\mathbf{int}^{*}$ represents a possibly empty set of integers, and is denoted as $\mathsf{y}^{*} \in \mathbb{Y}^{*}$.

We denote the set of all types of variables as $\mathbb{Y} = \mathbb{Y}^{\bullet} \cup \mathbb{Y}^{?} \cup \mathbb{Y}^{+} \cup \mathbb{Y}^{*}$. We assume that the set of variables $\mathbb{Y}$ is fixed throughout this chapter and $\mathbb{Y}^{\bullet} \cap \mathbb{Y}^{?} \cap \mathbb{Y}^{+} \cap \mathbb{Y}^{*} = \emptyset$. We denote $\mathsf{y}^{\bullet?}$ as a variable of either standard integer type or optional integer type, i.e., $\mathsf{y}^{\bullet?} \in \mathbb{Y}^{\bullet} \cup \mathbb{Y}^{?}$. A variable of any type $\mathsf{y}^{\bullet?+*}$ is denoted as y for short.

Because of the extension on types, we have to redefine the concrete states. Since a non/possibly-empty summary variable stores a set of integers, to make the concrete stores consistent, we evaluate a standard variable (resp. an optional variable) to a set of exactly one integer (resp. a set of one integer or an empty set).

Type	Variables	Stores
int$^\bullet$	$\mathtt{y}^\bullet \in \mathbb{Y}^\bullet$	1 value
int$^?$	$\mathtt{y}^? \in \mathbb{Y}^?$	0 or 1 value
int$^+$	$\mathtt{y}^+ \in \mathbb{Y}^+$	more than 1 value
int*	$\mathtt{y}^* \in \mathbb{Y}^*$	any number of values

Figure 4.2 – Four types of variables in the language

Definition 4.1 (Concrete States)**.** *We define* $\mathbb{V}^\bullet = \{\{v\} \mid v \in \mathbb{V}\}$*, then the set of concrete stores is redefined as:*

$$\sigma \in \mathbb{S} ::\overset{def.}{=} (\mathbb{Y}^\bullet \to \mathbb{V}^\bullet) \uplus (\mathbb{Y}^? \to (\mathbb{V}^\bullet \cup \emptyset)) \uplus (\mathbb{Y}^+ \to \mathcal{P}(\mathbb{V})\backslash\emptyset) \uplus (\mathbb{Y}^* \to \mathcal{P}(\mathbb{V}))$$

Expressions and statements. An l-value can be a variable of any integer type. An r-value expression can be a constant, a left-value, unary/binary operators applied on r-values and emptiness test. The evaluation $\mathbf{evalR}[\![\mathsf{r}]\!] : \mathbb{S} \to \mathcal{P}(\mathbb{V})$ of expression r produces a possibly empty set of values. For an arithmetic expression, the values in the output set is obtained by applying the arithmetic operator on the Cartesian product of the operand sets. A strong logical expression evaluates to $\{1\}$ when the logical relation is satisfied by all tuples in the Cartesian product of the operand sets. For instance, $\{3,4\} \geq \{2\}$ evaluates to $\{1\}$, but $\{3,1\} \geq \{2\}$ evaluates to $\{0\}$. One tricky part is that $\mathbf{evalR}[\![\mathsf{r}]\!]$ produces $\emptyset$ whenever it reads an empty variable in r: all operators are $\emptyset$-strict, i.e., they return $\emptyset$ whenever one of their arguments is equal to $\emptyset$, thus $\emptyset$ always propagates. Emptiness test **is_empty** takes a variable and outputs $\{1\}$ (resp. $\{0\}$) if the variable stores an empty (resp. non-empty) set of values.

The syntax and semantics of statements is quite similar to those in Chapter 3, but we redefine the semantics for condition tests and assignements. The new semantics $\mathbf{guard}[\![\mathsf{r}]\!]$ of condition r filters out the stores in which r evaluates to $\{0\}$, thus, it will also include stores where the evaluation encounters $\emptyset$. Note that the semantics of an assignment $\mathbf{stat}[\![\mathtt{y}^{\bullet+} = \mathsf{r}]\!]$ will produce no output store when r evaluates to $\emptyset$. Intuitively, we consider only executions where the empty value is never assigned to a standard and non-empty summary integer variable. Assignments to summary variables are weak updates.

Example 4.1 (Concrete semantics of condition tests)**.** *Suppose* $\mathbb{Y}^\bullet = \{\mathtt{y}^\bullet\}, \mathbb{Y}^* = \{\mathtt{y}^*\}$*, and given two concrete states* $\sigma_0 = \{\mathtt{y}^\bullet \mapsto \{3\}, \mathtt{y}^* \mapsto \{2,4\}\}$ *and* $\sigma_1 = \{\mathtt{y}^\bullet \mapsto \{3\}, \mathtt{y}^* \mapsto \emptyset\}$*,*

- *the concrete semantics for condition test* $\mathtt{y}^\bullet \leq \mathtt{y}^*$ *is* $\mathbf{guard}[\![\mathtt{y}^\bullet \leq \mathtt{y}^*]\!]\{\sigma_0, \sigma_1\} = \sigma_1$*, which filters out* σ_0*, because in* σ_0 *not all values in* $\mathtt{y}^*$ *is greater than or equal to the value stored in* $\mathtt{y}^\bullet$*;*
- *the concrete semantics for assignment* $\mathtt{y}^* = \mathtt{y}^\bullet$ *is* $\mathbf{stat}[\![\mathtt{y}^* = \mathtt{y}^\bullet]\!]\{\sigma_0\} = \{\mathtt{y}^\bullet \mapsto \{3\}, \mathtt{y}^* \mapsto \{2,3,4\}\}$*, which weakly updates* $\mathtt{y}^*$ *with the value stored in* $\mathtt{y}^\bullet$*.*

4.2 Maya Functor

In this section, we only consider programs with variables of standard integer type and optional type. Example 4.2 shows one such program. Programming languages like Ocaml or Scala are with such variable types.

Example 4.2 (A program with optional variables)**.** *We consider the program below, where* $\mathbb{Y}^? = \{\mathtt{y}_0^?, \mathtt{y}_1^?\}$ *and* $\mathbb{Y}^\bullet = \{\mathtt{y}_2^\bullet\}$*:*

$$
\begin{array}{l}
\textbf{if}(\mathtt{y}_2^\bullet \leq \mathtt{y}_0^?)\{ \\
\quad \textbf{if}(\mathtt{y}_0^? \leq 6)\{ \\
\quad\quad ① \ \mathtt{y}_1^? = \mathtt{y}_0^? + 2; \\
\quad\quad ② \ \ldots;
\end{array}
$$

Assuming that the variables may store any value (including $\emptyset$ *for optional variables) at the beginning of the execution:*

- *at point ①, we can observe exactly the stores such that* $\sigma(\mathtt{y}_2^\bullet) \leq \sigma(\mathtt{y}_0^?) \leq 6$ *where* $\mathtt{y}_0^?$ *contains a value, and the stores defined by* $\sigma(\mathtt{y}_0^?) = \emptyset$*;*
- *at point ②, we can observe exactly the stores such that* $\sigma(\mathtt{y}_2^\bullet) \leq \sigma(\mathtt{y}_0^?) \leq 6 \wedge \sigma(\mathtt{y}_1^?) = \sigma(\mathtt{y}_0^?) + 2$ *where neither* $\mathtt{y}_0^?$ *nor* $\mathtt{y}_1^?$ *is empty and the stores where* $\sigma(\mathtt{y}_0^?) = \emptyset$ *or* $\sigma(\mathtt{y}_1^?) = \emptyset$*.*

4.2.1 Numeric Domains for Euclidean Space

In this section, we assume a numerical domain $\overline{\mathcal{N}}$ is fixed, where abstract values correspond to conjunctions of constraints. For instance, linear equalities [Kar76], intervals [CC77], octagons [Min06] and polyhedra [CH78a] fit into this category. An abstract element $\overline{\mathtt{n}} \in \overline{\mathcal{N}}$ describes a possibly infinite set of points in a multi-dimensional Euclidean space. Dimensions range over a countable set $\mathbb{E}$, and we write $\mathbf{Dim}(\overline{\mathtt{n}})$ for the dimensions of abstract value $\overline{\mathtt{n}}$ ($\mathbf{Dim}(\overline{\mathtt{n}}) \subseteq \mathbb{E}$). Each point could be seen as an assignment of a value to each dimension $\nu \in \mathbb{E} \to \mathbb{V}$. We let $\gamma^{\overline{\mathtt{n}}} : \overline{\mathcal{N}} \to \mathcal{P}(\mathbb{E} \to \mathbb{V})$ denote the concretization function of domain $\overline{\mathcal{N}}$.

For a program containing only standard variables, the concrete states are of the form $\mathbb{Y}^\bullet \to \mathbb{V}^\bullet$. In conventional numeric analyses, each scalar program variable is bound to a dimension. Thus an abstract element $\overline{\mathtt{n}} \in \overline{\mathcal{N}}$ corresponds to a set of such concrete states.

4.2.2 Abstraction in Presence of Optional Numerical Values

A Maya abstract domain is the result domain after applying Maya functor on a conventional numeric domain. The main idea of Maya functor is to represent an optional variable $\mathtt{y}^?$ by a set of dimensions in the parameter numeric domain. It assumes that $\mathtt{y}^?$ can be defined if and only if all these dimensions may map to a common value.

An abstract state of the *Maya functor* over $\overline{\mathcal{N}}$ is defined by an abstract value $\overline{\mathtt{n}} \in \overline{\mathcal{N}}$ describing constraints over a set of dimensions defined as follows:

- each standard integer variable $\mathtt{y}^\bullet$ corresponds to exactly one dimension, noted as $\mathtt{d}$;
- each optional variable $\mathtt{y}^?$ corresponds to a finite set of dimensions (for clarity, we call these dimensions *avatars* and always mark them with superscripts such as: $\mathtt{d}^0, \mathtt{d}^1, \ldots$).

Therefore, we attach a function $\mathcal{A} : \mathbb{Y}^? \to \mathcal{P}(\mathbb{E}) \cup \mathbb{Y}^\bullet \to \mathbb{E}$ which describes the mapping of program variables into dimensions in numerical abstract value $\overline{\mathtt{n}}$.

Definition 4.2 (Abstract states in the Maya domain)**.** *An abstract state of the* Maya abstract functor *over* $\overline{\mathcal{N}}$ *is a pair* $\overline{\mathtt{o}} = (\overline{\mathtt{n}}, \mathcal{A})$ *such that:*

$$\mathbf{Dim}(\overline{\mathtt{n}}) = \left(\biguplus\{\mathcal{A}(\mathtt{y}^{\bullet?}) \mid \mathtt{y}^{\bullet?} \in \mathbb{Y}^\bullet \cup \mathbb{Y}^?\}\right)$$

We let $\overline{\mathcal{O}}$ *denote the set of such states.*

Note that the above definition implicitly asserts that distinct variables are represented by disjoint sets of dimensions. We also make the convention that a program variable and the dimensions representing it have the same subscript.

Example 4.3 (An abstract state in the Maya domain)**.** *In this example, we assume* $\overline{\mathcal{N}}$ *is the Polyhedra domain, and that* $\mathbb{Y}^\bullet = \{\mathtt{y}_0^\bullet\}$, $\mathbb{Y}^? = \{\mathtt{y}_1^?\}$. *Furthermore, we let each optional variable be described by two avatars. Thus,* $\mathbb{E} = \{\mathtt{d}_0, \mathtt{d}_1^\uparrow, \mathtt{d}_1^\downarrow\}$. *Moreover, an example abstract state is* $\overline{\mathtt{o}} = (\overline{\mathtt{n}}, \mathcal{A})$*, with:*

$$\overline{\mathtt{n}} = \left\{0 \leq \mathtt{d}_0 \wedge \mathtt{d}_0 \leq 10 \wedge 5 \leq \mathtt{d}_1^\downarrow \wedge \mathtt{d}_1^\uparrow \leq \mathtt{d}_0\right\} \qquad \mathcal{A} : \mathtt{y}_0^\bullet \longmapsto \mathtt{d}_0 \wedge \mathtt{y}_1^? \longmapsto \{\mathtt{d}_1^\downarrow, \mathtt{d}_1^\uparrow\}$$

Concretization. An abstract element $\overline{\mathtt{n}}$ describes a set of valuations ν that maps all dimensions to a value. The concretization of an abstract state $\overline{\mathtt{o}} = (\overline{\mathtt{n}}, \mathcal{A})$ is a set of concrete stores, that can be obtained by collapsing *all avatars of each optional variable* to a unique value. This is described by a pair of *consistency predicates*, which state when a store σ is compatible with ν.

Definition 4.3 (Concretization function in the Maya domain)**.** *Given abstract state* $\overline{\mathtt{o}} = (\overline{\mathtt{n}}, \mathcal{A})$*, we define the following consistency predicates:*

$$\begin{array}{lcl} P_{\mathbb{Y}^\bullet}(\sigma, \overline{\mathtt{o}}, \nu) & \overset{def.}{\iff} & \forall \mathtt{y}^\bullet \in \mathbb{Y}^\bullet,\ \sigma(\mathtt{y}^\bullet) = \nu(\mathcal{A}(\mathtt{y}^\bullet)) \\ P_{\mathbb{Y}^?}(\sigma, \overline{\mathtt{o}}, \nu) & \overset{def.}{\iff} & \forall \mathtt{y}^? \in \mathbb{Y}^?,\ \left(\forall \mathtt{d} \in \mathcal{A}(\mathtt{y}^?),\ \nu(\mathtt{d}) = \sigma(\mathtt{y}^?)\right) \vee \sigma(\mathtt{y}^?) = \emptyset \end{array}$$

Then, the concretization of $\overline{\mathtt{o}} = (\overline{\mathtt{n}}, \mathcal{A})$ *is defined by:*

$$\gamma^{\overline{\mathcal{O}}}(\overline{\mathtt{o}}) \overset{def.}{::=} \left\{\sigma \in \mathbb{S} \mid \exists \nu \in \gamma^{\overline{\mathtt{n}}}(\overline{\mathtt{n}}),\ P_{\mathbb{Y}^\bullet}(\sigma, \overline{\mathtt{o}}, \nu) \wedge P_{\mathbb{Y}^?}(\sigma, \overline{\mathtt{o}}, \nu)\right\}$$

Intuitively, consistency predicate $P_{\mathbb{Y}^\bullet}$ asserts that the valuation and the concrete store agree on the mapping of the standard variables, whereas consistency predicate $P_{\mathbb{Y}^?}$ asserts that the valuation assigns all avatars of each optional variable to the value of that variable in the store.

Example 4.4 (The concretization of an abstract state in the Maya domain)**.** *We consider the abstract state shown in Example 4.3. Its concretization consists of:*

- *the stores defined by* $5 \leq \sigma(\mathtt{y}_0^\bullet) \leq 10 \wedge 5 \leq \sigma(\mathtt{y}_1^?) \leq \sigma(\mathtt{y}_0^\bullet)$ *(the valuation is then fully defined by the store since no variable stores* $\emptyset$*);*
- *the stores defined by* $0 \leq \sigma(\mathtt{y}_0^\bullet) \leq 10 \wedge \sigma(\mathtt{y}_1^?) = \emptyset$ *(a possible valuation is defined by* $\nu(\mathtt{d}_0) = \sigma(\mathtt{y}_0^\bullet), \nu(\mathtt{d}_1^\downarrow) = 15, \nu(\mathtt{d}_1^\uparrow) = \nu(\mathtt{d}_0)$*).*

This example shows how our domain can distribute the constraints on an optional variable $\mathtt{y}_1^?$ over several dimensions, so as to express the fact that $\mathtt{y}_1^?$ must store $\emptyset$.

Remark 4.1. *In this example, we also observe that, given* $\sigma \in \gamma^{\overline{\mathcal{O}}}(\overline{\mathtt{o}})$*, and if* σ' *is such that, for any standard variable* $\mathtt{y}^\bullet$*,* $\sigma'(\mathtt{y}^\bullet) = \sigma(\mathtt{y}^\bullet)$*, and for all optional variable* $\mathtt{y}^?$*, either* $\sigma'(\mathtt{y}^?) = \sigma(\mathtt{y}^?)$ *or* $\sigma'(\mathtt{y}^?) = \emptyset$*, then* $\sigma' \in \gamma^{\overline{\mathcal{O}}}(\overline{\mathtt{o}})$*. In other words, our functor cannot express that an optional variable* must not *store* $\emptyset$*. However, our abstraction allows to derive emptiness of a group via constraints over multiple avatars of variables denoting its contents or indexes.*

Choice of avatar dimensions. The definition of abstract elements assumes nothing about the number of avatar dimensions, and about the way the constraints over an optional variable are distributed over its avatars. However, in practice, the way avatar dimensions are managed has a great impact on the efficiency and precision of the analysis. In practice, we have to set some principles to help the transfer functions and abstract lattice operations implement an efficient principle to manage these dimensions. In particular at certain stages, new avatars have to be introduced so as to avoid a loss of precision.

Example 4.5 (Choice of avatar dimensions)**.** *We discuss possible abstract invariants for the program shown in Example 4.2, starting with the set of all stores as a pre-condition, described by abstract state* $\top$*. After test* $\mathtt{y}_2^\bullet \leq \mathtt{y}_0^?$*, the analysis should compute an abstraction of the stores where, either* $\mathtt{y}_0^?$ *is mapped only to* $\emptyset$ *or where the numerical constraint is satisfied. Using the octagon abstract domain, and a single avatar* $\mathtt{d}_0$ *for* $\mathtt{y}_0^?$*, this boils down to abstract state*

$$(\mathtt{d}_2 - \mathtt{d}_0 \leq 0, \mathtt{y}_0^? \mapsto \{\mathtt{d}_0\} \wedge \mathtt{y}_2^\bullet \mapsto \{\mathtt{d}_2\})$$

After the second test, we get the set of stores observed at point ①, that is such that, either $\sigma(\mathtt{y}_2^\bullet) \leq \sigma(\mathtt{y}_0^?) \leq 6$ *or* $\sigma(\mathtt{y}_0^?) = \emptyset$*. Note that this set of stores cannot be described exactly with octagons using a single avatar. Indeed, this set contains stores such that* $\sigma(\mathtt{y}_2^\bullet) > 6$ *(when* $\sigma(\mathtt{y}_0^?) = \emptyset$*). Thus, using a single avatar to describe constraints over* $\mathtt{y}_0^?$ *would force the analysis to drop either constraint* $\mathtt{y}_0^? \leq 6$ *or constraint* $\mathtt{y}_2^\bullet \leq \mathtt{y}_0^?$*. Thus, adding a second avatar for* $\mathtt{y}_0^?$ *at this point is necessary in order to maintain maximal precision. In particular, the abstract state below describes exactly the stores that can be observed at point ①:*

$$(\mathtt{d}_2 \leq \mathtt{d}_0^0 \wedge \mathtt{d}_0^1 \leq 6, \mathtt{y}_2^\bullet \mapsto \mathtt{d}_2 \wedge \mathtt{y}_0^? \mapsto \{\mathtt{d}_0^0, \mathtt{d}_0^1\})$$

The above example demonstrates the need to introduce enough avatars so that all constraints on optional variables can be maintained, without "over-constraining" standard variables (which would result in an unsound analysis). Intuitively, each avatar should not carry too much information: the base numerical domain cannot express emptiness of a specific avatar; instead, only the conjunction of all avatars of an optional variable $\mathtt{y}^?$ may express that $\mathtt{y}^?$ is empty. We formalize this as a *sufficient condition*, that we call the *independence* property, and that should be maintained by all abstract operators in the Maya domain. This property states that dropping the constraints over an avatar dimension $\mathtt{d}$ associated to variable $\mathtt{y}^?$ should have no impact on the variables other than $\mathtt{y}^?$. To maintain this property, transfer functions and abstract operators may either pay the cost of adding new avatar dimensions or will have to drop constraints that cannot be represented without adding more avatars. To formalize the independence property, and given abstract value $\overline{\mathtt{n}} \in \overline{\mathcal{N}}$ and dimension $\mathtt{d}$, we note $\mathbf{drop}(\overline{\mathtt{n}}, \mathtt{d})$ for the abstract value obtained by removing from $\overline{\mathtt{n}}$ all the constraints that involve $\mathtt{d}$ (this operation is well defined since we assumed elements of abstract domain $\overline{\mathcal{N}}$ correspond to the finite conjunctions of all the constraints of a certain form). Moreover, if ν is a valuation, we write $\nu_{|\mathbb{E}\setminus\mathtt{d}}$ for the restriction of ν to $\mathbb{E} \setminus \{\mathtt{d}\}$.

Definition 4.4 (Independence property)**.** *Let $\overline{\mathtt{o}} = (\overline{\mathtt{n}}, \mathcal{A})$ be an abstract state. We say $\overline{\mathtt{o}}$ satisfies the independence property if and only if*

$$\forall \mathtt{y}^? \in \mathbb{Y}^?,\ \forall \mathtt{d} \in \mathcal{A}(\mathtt{y}^?),\ \{\nu_{|\mathbb{E}\setminus\mathtt{d}} \mid \nu \in \gamma^{\overline{\mathtt{n}}}(\overline{\mathtt{n}})\} = \{\nu_{|\mathbb{E}\setminus\mathtt{d}} \mid \nu \in \gamma^{\overline{\mathtt{n}}}(\mathbf{drop}(\overline{\mathtt{n}}, \mathtt{d}))\}$$

Example 4.6 (Independence property)**.** *The abstract state given at the end of Example 4.5 satisfies the independence property, using two avatars, that respectively carry the lower and upper bound constraints over $\mathtt{y}^?$. Section 4.2.3 generalizes this approach to lift any domain based on linear inequalities.*

Example 4.7 (Multiple avatar dimensions for one variable)**.** *Intuitively, the independence property is likely to break when an avatar dimension carries several constraints, the conjunction of which may be unsatisfiable. Therefore, an alternate technique to achieve it consists in using one avatar per constraint over each optional variable. As an example, we consider the set of concrete states defined by $\mathbb{Y}^\bullet = \{\mathtt{y}_0^\bullet\}$ and $\mathbb{Y}^? = \{\mathtt{y}_1^?, \mathtt{y}_2^?\}$ and where the optional variables are either undefined or satisfy the following conditions: $\mathtt{y}_0^\bullet \leq \mathtt{y}_1^? \wedge \mathtt{y}_1^? \leq 2\mathtt{y}_0^\bullet \wedge \mathtt{y}_1^? = \mathtt{y}_2^? + 2$. Then, assuming $\overline{\mathcal{N}}$ is the polyhedra abstract domain, this* multi-avatar *principle will construct the following abstract state:*

$$\begin{array}{rcl} \overline{\mathtt{n}} & = & \{\mathtt{d}_0 \leq \mathtt{d}_1^0 \wedge \mathtt{d}_1^1 \leq 2\mathtt{d}_0 \wedge \mathtt{d}_1^2 \leq \mathtt{d}_2^0 + 2 \wedge \mathtt{d}_2^1 + 2 \leq \mathtt{d}_1^3\} \\ \mathcal{A} & : & \mathtt{y}_0^\bullet \mapsto \mathtt{d}_0, \mathtt{y}_1^? \mapsto \{\mathtt{d}_1^0, \mathtt{d}_1^1, \mathtt{d}_1^2, \mathtt{d}_1^3\}, \mathtt{y}_2^? \mapsto \{\mathtt{d}_2^0, \mathtt{d}_2^1\}, \end{array}$$

This principle is general (it can be applied to, e.g., linear equalities [Kar76]) but costly.

4.2.3 The Bi-avatar Principle

We now propose a principle to manage avatar dimensions and design abstract operations under the hypothesis that the predicates expressed in the base abstract domain $\overline{\mathcal{N}}$ are

conjunctions of linear inequality (which includes intervals, octagons, polyhedra, and their variants).

4.2.3.1 Abstraction

Numeric constraints in the base domain are all of the form $a_0\mathtt{d}_0 + \ldots + a_n\mathtt{d}_n \leq c$ (where $a_0, \ldots, a_n, c$ are constants), thus a constraint involving $\mathtt{d}_i$ (i.e., where $a_i \neq 0$) is either an *upper* bound constraint for $\mathtt{d}_i$ (if $a_i > 0$) or a *lower* bound constraint (if $a_i < 0$). The bi-avatar principle treats those two sets of constraints separately, using two avatar dimensions per optional variable.

Definition 4.5 (The bi-avatar principle)**.** *Abstract state $\overline{\mathtt{o}} = (\overline{\mathtt{n}}, \mathcal{A})$ follows the* bi-avatar principle *if and only if $\mathcal{A}$ maps each optional variable $\mathtt{y}^?$ to a pair of dimensions $\{\mathtt{d}^\uparrow, \mathtt{d}^\downarrow\}$, called upper avatar and lower avatar respectively. Each upper avatar $\mathtt{d}^\uparrow$ (resp., lower avatar $\mathtt{d}^\downarrow$) carries only "upper bound constraints" (resp., "lower bound constraints") on $\mathtt{y}^?$.*

In other words, the bi-avatar principle fully determines $\mathcal{A}$. In order to implement this principle, we need to ensure that all abstract operators preserve $\mathcal{A}$, and the property of lower and upper avatars. We define such abstract operations in the next subsections. Interestingly, whenever an abstract state follows this principle, and if we drop all constraints over an (upper or lower) avatar of $\mathtt{y}$, the concretization restricted to the dimensions other than that avatar do not change. This entails:

Theorem 4.1 (The bi-avatar principle satisfying the independence property)**.** *All abstract values that follow the bi-avatar principle satisfy the independence property (Definition 4.4).*

To express the emptiness of an optional variable, we simply need to let its avatars carry a pair of constraints that would be unsatisfiable, if carried by a unique dimension, such as $1 \leq \mathtt{d}^\downarrow \wedge \mathtt{d}^\uparrow \leq 0$.

Example 4.8 (The bi-avatar pirnciple)**.** *Let $\mathbb{Y}^\bullet = \{\mathtt{y}_0^\bullet\}, \mathbb{Y}^? = \{\mathtt{y}_1^?\}$, and let $\mathcal{A}$ specify the avatars defined by the bi-avatar principle. Then, the following numerical abstract values specify the sets of concrete states below:*

abstract numerical state $\overline{\mathtt{n}}$	*concretization* $\gamma^{\overline{\mathcal{O}}}(\overline{\mathtt{n}}, \mathcal{A})$
$1 \leq \mathtt{d}_0 \wedge \mathtt{d}_0 \leq 1 \wedge \mathtt{d}_0 \leq \mathtt{d}_1^\downarrow \wedge \mathtt{d}_1^\uparrow \leq \mathtt{d}_0$	$\{\mathtt{y}_0^\bullet \mapsto \{1\}, \mathtt{y}_1^? \mapsto \{1\}\}, \{\mathtt{y}_0^\bullet \mapsto \{1\}, \mathtt{y}_1^? \mapsto \emptyset\}$
$1 \leq \mathtt{d}_0 \wedge \mathtt{d}_0 \leq 1 \wedge \mathtt{d}_0 \leq \mathtt{d}_1^\downarrow \wedge \mathtt{d}_1^\uparrow \leq \mathtt{d}_0 - 1$	$\{\mathtt{y}_0^\bullet \mapsto \{1\}, \mathtt{y}_1^? \mapsto \emptyset\}$

Preservation. The abstract operators described in the remainder of this section either discard constraints violating the bi-avatar principle (such as assignment, in Section 4.2.3.4), or never apply operations of $\overline{\mathcal{N}}$ that would cause them to bound a $\mathtt{d}^\uparrow$ (resp., $\mathtt{d}^\downarrow$) avatar below (resp., above). This implies straightforwardly that, in the resulting domain, all abstract elements with a non empty concretization follow the bi-avatar principle

(all $\mathtt{d}^{\downarrow}$ dimensions are not bounded by above and all $\mathtt{d}^{\uparrow}$ dimensions are not bounded by below).

Expressiveness. Under the bi-avatar principle, we can compare the expressiveness of Maya domain $\overline{\mathcal{O}}$ with that of its base domain $\overline{\mathcal{N}}$: if a set of stores S with no optional variable containing $\emptyset$ can be described exactly by $\overline{\mathtt{n}} \in \overline{\mathcal{N}}$, we can still describe S in $\overline{\mathcal{O}}$, up-to the change of any set of optional variable to $\emptyset$. Indeed, if we let $\overline{\mathbb{S}}_{\text{def}} = (\mathbb{Y}^{\bullet} \uplus \mathbb{Y}^{?}) \to \mathbb{V}^{\bullet}$, we have:

Theorem 4.2 (The expressivenss of abstract states that follow the bi-avatar principle)**.** *If $\mathcal{A}$ follows the bi-avatar principle, then:*

$$\forall \overline{\mathtt{n}}_0 \in \overline{\mathcal{N}},\ \mathbf{Dim}(\overline{\mathtt{n}}_0) = \mathbb{Y}^{\bullet} \uplus \mathbb{Y}^{?} \Longrightarrow \exists \overline{\mathtt{n}}_1 \in \overline{\mathcal{O}},\ \gamma^{\overline{\mathtt{n}}}(\overline{\mathtt{n}}_0) = \gamma^{\overline{\mathcal{O}}}(\overline{\mathtt{n}}_1, \mathcal{A}) \cap \overline{\mathbb{S}}_{\text{def}}$$

Proof. We assume $\mathbf{Dim}(\overline{\mathtt{n}}_0) = \mathbb{Y}^{\bullet} \uplus \mathbb{Y}^{*}$. To construct $\overline{\mathtt{n}}_1$ from $\overline{\mathtt{n}}_0$, we simply need to replace any occurrence of $\mathtt{y}^{?} \in \mathbb{Y}^{*}$ by $\mathtt{d}^{\downarrow}$ or $\mathtt{d}^{\uparrow}$ depending on the constraint (we replace it by $\mathtt{d}^{\downarrow}$ if it is a lower constraint, and by $\mathtt{d}^{\uparrow}$ if it is an upper constraint) and $\mathtt{y}^{\bullet} \in \mathbb{Y}^{\bullet}$ by $\mathtt{d}$. Then, we prove the equality by double inclusion:

- We first prove $\gamma^{\overline{\mathtt{n}}}(\overline{\mathtt{n}}_0) \subseteq \gamma^{\overline{\mathcal{O}}}(\overline{\mathtt{n}}_1, \mathcal{A}) \cap \overline{\mathbb{S}}_{\text{def}}$. Let $\sigma \in \gamma^{\overline{\mathtt{n}}}(\overline{\mathtt{n}}_0)$. Then, we can construct valuation ν_1 over $\mathbf{Dim}(\overline{\mathtt{n}}_1)$ by $\forall \mathtt{y}^{\bullet} \in \mathbb{Y}^{\bullet}$, $\nu_1(\mathtt{d}) = \sigma(\mathtt{d})$ and $\forall \mathtt{y}^{?} \in \mathbb{Y}^{*}$, $\nu_1(\mathtt{d}^{\downarrow}) = \nu_1(\mathtt{d}^{\uparrow}) = \sigma(\mathtt{y}^{?})$. Then, clearly we have both $P_{\mathbb{Y}^{\bullet}}(\sigma, (\overline{\mathtt{n}}_1, \mathcal{A}), \nu_1)$ and $P_{\mathbb{Y}^{?}}(\sigma, (\overline{\mathtt{n}}_1, \mathcal{A}), \nu_1)$ thus, $\sigma \in \gamma^{\overline{\mathcal{O}}}(\overline{\mathtt{n}}_1)$. Moreover, $\sigma \in \overline{\mathbb{S}}_{\text{def}}$ (by definition of $\gamma^{\overline{\mathtt{n}}}$).
- Second, we prove $\gamma^{\overline{\mathcal{O}}}(\overline{\mathtt{n}}_1, \mathcal{A}) \cap \overline{\mathbb{S}}_{\text{def}} \subseteq \gamma^{\overline{\mathtt{n}}}(\overline{\mathtt{n}}_0)$. Let $\sigma \in \gamma^{\overline{\mathcal{O}}}(\overline{\mathtt{n}}_1, \mathcal{A}) \cap \overline{\mathbb{S}}_{\text{def}}$. Then, there exists a valuation ν_1 such that properties $P_{\mathbb{Y}^{\bullet}}(\sigma, (\overline{\mathtt{n}}_1, \mathcal{A}), \nu_1)$ and $P_{\mathbb{Y}^{?}}(\sigma, (\overline{\mathtt{n}}_1, \mathcal{A}), \nu_1)$ hold. By the definition of $\overline{\mathtt{n}}_1$, this entails that $\sigma \in \gamma^{\overline{\mathtt{n}}}(\overline{\mathtt{n}}_0)$.

□

Example 4.9 (The expressivenss of the bi-avatar principle)**.** *Let $\mathbb{Y}^{\bullet} = \{\mathtt{y}_0^{\bullet}\}$ and $\mathbb{Y}^{?} = \{\mathtt{y}_1^{?}\}$, given an abstract element in the octagons domain as $\overline{\mathtt{n}} = 1 \leq \mathtt{y}_0^{\bullet} \leq 1 \wedge 1 \leq \mathtt{y}_1^{?} \leq 1$, it is easy to know that its concretization contains only one concrete state $\sigma = \{\mathtt{y}_0^{\bullet} \mapsto \{1\}, \mathtt{y}_1^{?} \mapsto \{1\}\}$. The corresponding abstract element in the Maya domain is*

$$\overline{\mathtt{o}} = (1 \leq \mathtt{d}_0 \leq 1 \wedge 1 \leq \mathtt{d}_1^{\downarrow} \wedge \mathtt{d}_1^{\uparrow} \leq 1, \mathcal{A}),\ \textit{where}\ \mathcal{A} : \mathtt{y}_0^{\bullet} \mapsto \{\mathtt{d}_0\}, \mathtt{y}_1^{?} \mapsto \{\mathtt{d}_1^{\downarrow}, \mathtt{d}_1^{\uparrow}\}$$

The concretization of $\overline{\mathtt{o}}$ is $\sigma \wedge \{\mathtt{y}_0^{\bullet} \mapsto 1 \wedge \mathtt{y}_1^{\bullet} \mapsto \emptyset\}$. Since $\overline{\mathbb{S}}_{\text{def}}$ asserts that no optional varaible constains $\emptyset$, $\gamma^{\overline{\mathcal{O}}}(\overline{\mathtt{o}}) \cap \overline{\mathbb{S}}_{\text{def}} = \{\sigma\} = \gamma^{\overline{\mathtt{n}}}(\overline{\mathtt{n}}_0)$.

4.2.3.2 Condition Test

The concrete semantics of a condition test $\mathtt{r}$ filters out stores for which $\mathtt{r}$ evaluates to 0. We assume $\overline{\mathcal{N}}$ provides a sound abstract function $\overline{\mathbf{guard}}^{\overline{\mathcal{N}}}[\mathtt{r}] : \overline{\mathcal{N}} \to \overline{\mathcal{N}}$ (where $\mathtt{r}$ contains only variables of standard integer type), and build an abstract operator $\overline{\mathbf{guard}}^{\overline{\mathcal{O}}}[\mathtt{r}] : \overline{\mathcal{O}} \to \overline{\mathcal{O}}$.

Optional variable emptiness test. To evaluate condition $\overline{\textbf{guard}}^{\overline{\mathcal{O}}}[\textbf{is\_empty}(\mathtt{y}^?)]$, and filter out stores that do not map $\mathtt{y}^?$ into $\emptyset$, we can simply add two constraints on $\mathtt{d}^\downarrow$ and $\mathtt{d}^\uparrow$ that would be unsatisfiable, if added for a same dimension, such as $1 \leq \mathtt{d}^\downarrow$ and $\mathtt{d}^\uparrow \leq 0$.

Numerical tests. We consider only conditions that are linear inequalities, as non-linear conditions are often handled by linearization techniques [Min04b], and a linear equality is equivalent to a pair of inequalities.

Intuitively, $\overline{\textbf{guard}}^{\overline{\mathcal{O}}}[.]$ should simply add a linear constraint to some abstract state $\overline{\mathtt{o}}$ (with some approximation, as this constraint is in general not representable exactly in $\overline{\mathcal{N}}$). Given condition test $a_0\mathtt{y}_0^\bullet + \ldots + a_n\mathtt{y}_n^\bullet + a_{n+1}\mathtt{y}_{n+1}^? + \ldots + a_{n+m}\mathtt{y}_{n+m}^? \leq c$ (where $\mathtt{y}_i^\bullet \in \mathbb{Y}^\bullet$ and $\mathtt{y}_i^? \in \mathbb{Y}^?$), we can produce another constraint that involves only standard variables and avatar dimensions by replacing $\mathtt{y}_i$ either by $\mathtt{d}_i^\downarrow$ or by $\mathtt{d}_i^\uparrow$ depending on the sign of a_i. This constraint is compatible with the bi-avatar principle (Section 4.2.3), hence it can be represented precisely in the numerical domain, even if it indirectly entails emptiness of some optional variables (in other words, not using the bi-avatar property would cause a severe precision loss here). Thus, numerical condition test can be applied to this constraint. In turn, the absence of constraints violating the bi-avatar principle needs to be verified on the output of $\overline{\textbf{guard}}^{\overline{\mathcal{N}}}[.]$. Moreover, this constraint is equivalent to the initial constraint up-to the $\gamma^{\overline{\mathcal{O}}}$ concretization function. Thus, this principle defines a sound abstract transfer function for condition tests.

Definition 4.6 (Analysis of condition tests in the Maya domain)**.** *The full algorithm of the condition test transfer function* $\overline{\textbf{guard}}^{\overline{\mathcal{O}}}[.]$ *is shown in Figure 4.3. This function uses* $\overline{\textbf{replace}}$ *to perform variable substitutions in conditions and the sound test function of the underlying domain* $\overline{\textbf{guard}}^{\overline{\mathcal{N}}}[.]$.

Theorem 4.3 (Soundness of the transfer function for condition tests)**.** *The abstract transfer function* $\overline{\textbf{guard}}^{\overline{\mathcal{O}}}[.]$ *is* sound *in the sense that, for all linear inequality constraint* $\mathtt{r}$ *and for all abstract state* $\overline{\mathtt{o}}$ *satisfying the bi-avatar principle:*

$$\textbf{guard}[\![\mathtt{r}]\!](\gamma^{\overline{\mathcal{O}}}(\overline{\mathtt{o}})) \subseteq \gamma^{\overline{\mathcal{O}}}(\overline{\textbf{guard}}^{\overline{\mathcal{O}}}[\mathtt{r}](\overline{\mathtt{o}}))$$

Theorem 4.4 (Perservation of bi-avatar principle by condition test)**.** *The image by abstract transfer function* $\overline{\textbf{guard}}^{\overline{\mathcal{O}}}[.]$ *still satisfies the bi-avatar principle.*

Example 4.10 (Transfer functions for condition tests)**.** *In this example, we assume that* $\overline{\mathcal{N}}$ *is the Polyhedra domain, and that* $\mathbb{Y}^\bullet = \{\mathtt{y}_0^\bullet\}$*, and* $\mathbb{Y}^? = \{\mathtt{y}_1^?\}$ *(thus,* $\mathcal{A} : \mathtt{y}_0^\bullet \mapsto \{\mathtt{d}_0\}, \mathtt{y}^? \mapsto \{\mathtt{d}_1^\downarrow, \mathtt{d}_1^\uparrow\}$*). We consider an abstract pre-condition*

$$\overline{\mathtt{o}} = (\overline{\mathtt{n}}_0, \mathcal{A}), \text{ where } \overline{\mathtt{n}}_0 = (5 \leq \mathtt{d}_0 \wedge \mathtt{d}_0 \leq 5)$$

$\overline{\textbf{guard}}^{\overline{\mathcal{O}}}[\mathsf{r}](\overline{\mathtt{n}}, \mathcal{A})$
  if $\mathsf{r} = \textbf{is\_empty}(\mathtt{y}^?)$
    result $:= (\overline{\textbf{guard}}^{\overline{\mathcal{O}}}[1 \leq \mathtt{d}^{\downarrow}](\overline{\textbf{guard}}^{\overline{\mathcal{N}}}[\mathtt{d}^{\uparrow} \leq 0](\overline{\mathtt{n}}), \mathcal{A})$
  if $\mathsf{r} = \sum_{i=0}^{n} a_i \mathtt{y}_i^{\bullet} + \sum_{i=n+1}^{n+m} a_i \mathtt{y}_i^{?} \leq c$
    foreach $i \in [0, n]$)
      $\mathsf{r} := \overline{\textbf{replace}}(\mathsf{r}, \mathtt{y}_i^{\bullet}, \mathtt{d}_i)$
    foreach $i \in [n+1, n+m]$)
      if $a_i < 0$ **then** $\mathsf{r} := \overline{\textbf{replace}}(\mathsf{r}, \mathtt{y}_i^{?}, \mathtt{d}_i^{\downarrow})$
      if $a_i > 0$ **then** $\mathsf{r} := \overline{\textbf{replace}}(\mathsf{r}, \mathtt{y}_i^{?}, \mathtt{d}_i^{\uparrow})$
    result $:= (\overline{\textbf{guard}}^{\overline{\mathcal{N}}}[\mathsf{r}](\overline{\mathtt{n}}), \mathcal{A})$

Figure 4.3 – Condition test abstract transfer function

and a condition test $\mathtt{y}_1^? - \mathtt{y}_0^{\bullet} \leq 3$. *Abstract test* $\overline{\textbf{guard}}^{\overline{\mathcal{O}}}[\mathtt{y}_1^? - \mathtt{y}_0^{\bullet} \leq 3](\overline{\mathtt{o}})$ *first substitutes* $\mathtt{d}_1^{\uparrow}$ *for* $\mathtt{y}_1^?$ *and* $\mathtt{d}_0$ *for* $\mathtt{y}_0^{\bullet}$ *in* $(\mathtt{y}_1^? - \mathtt{y}_0^{\bullet} \leq 3)$, *which generates condition* $\mathtt{d}_1^{\uparrow} - \mathtt{d}_0 \leq 3$. *Then, it computes* $\overline{\textbf{guard}}^{\overline{\mathcal{N}}}[\mathtt{d}_1^{\uparrow} - \mathtt{d}_0 \leq 3](\overline{\mathtt{n}}_0)$. *Thus, we obtain the abstract post-condition*

$$(\overline{\mathtt{n}}_1, \mathcal{A}), \text{ where } \overline{\mathtt{n}}_1 = (5 \leq \mathtt{d}_0 \wedge \mathtt{d}_0 \leq 5 \wedge \mathtt{d}_1^{\uparrow} - \mathtt{d}_0 \leq 3)$$

4.2.3.3 Verifying the Satisfaction of A Constraint

To verify assertions, we need an operator $\overline{\textbf{sat}}^{\overline{\mathcal{O}}}[\mathsf{r}] : \overline{\mathcal{O}} \to \{\{1\}, \{0\}\}$ such that, if $\sigma \in \gamma^{\overline{\mathcal{O}}}(\overline{\mathtt{o}})$ and $\overline{\textbf{sat}}^{\overline{\mathcal{O}}}[\mathsf{r}](\overline{\mathtt{o}}) = \{1\}$, then $\textbf{guard}[\![\mathsf{r}]\!](\sigma) = \sigma$. The case of numerical assertions is very similar to the case of numeric tests.

To test whether $\mathtt{y}^?$ can store only $\emptyset$ in any store described by $(\overline{\mathtt{n}}, \mathcal{A})$, we simply need to check whether constraint $\mathtt{d}^{\downarrow} = \mathtt{d}^{\uparrow}$ is unsatisfiable. This suggests $\overline{\textbf{sat}}^{\overline{\mathcal{O}}}[\textbf{is\_empty}(\mathtt{y}^?)](\overline{\mathtt{n}}, \mathcal{A}) = \overline{\textbf{is\_bot}}^{\overline{\mathcal{N}}}(\overline{\textbf{guard}}^{\overline{\mathcal{N}}}[\mathtt{d}^{\downarrow} = \mathtt{d}^{\uparrow}](\overline{\mathtt{n}}))$, where $\overline{\textbf{is\_bot}}^{\overline{\mathcal{N}}} : \overline{\mathcal{N}} \to \{\{1\}, \{0\}\}$ is a sound emptiness test (if $\overline{\textbf{is\_bot}}^{\overline{\mathcal{N}}}(\overline{\mathtt{n}}) = \{1\}$, then $\gamma^{\overline{\mathtt{n}}}(\overline{\mathtt{n}}) = \emptyset$). The full algorithm of satisfaction is shown in Figure 4.4.

4.2.3.4 Assignment

We now describe a transfer function $\overline{\textbf{assign}}^{\overline{\mathcal{O}}}[.]$ that over-approximates the effect of an assignment. We consider assignments with a linear right hand side expression (non linear assignment can be implemented using linearization [Min04b]).

Emptiness test. If the left-hand side $\mathtt{y}^{\bullet}$ is a standard type variable and an optional variable $\mathtt{y}^?$ appears in the right hand side, according to the concrete semantics, no state satisfies the condtition that $\mathtt{y}^?$ takes an empty set. Therefore, given an abstract pre-condition

$\overline{\textbf{sat}}^{\overline{\mathcal{O}}}[\mathsf{r}](\overline{\mathtt{n}}, \mathcal{A})$
  if $\mathsf{r} = \textbf{is\_empty}(\mathtt{y}^?)$
    $\textbf{result} := \overline{\textbf{is\_bot}}^{\overline{\mathcal{N}}}(\overline{\textbf{guard}}^{\overline{\mathcal{N}}}[\mathtt{d}^{\downarrow} = \mathtt{d}^{\uparrow}](\overline{\mathtt{n}}))$
  if $\mathsf{r} = \sum_{i=0}^{n} a_i \mathtt{y}_i^{\bullet} + \sum_{i=n+1}^{n+m} a_i \mathtt{y}_i^? \leq c$
    foreach $i \in [0, n]$)
      $\mathsf{r} := \overline{\textbf{replace}}(\mathsf{r}, \mathtt{y}_i^{\bullet}, \mathtt{d}_i)$
    foreach $i \in [n+1, n+m]$)
      if $a_i < 0$ **then** $\mathsf{r} := \overline{\textbf{replace}}(\mathsf{r}, \mathtt{y}_i^?, \mathtt{d}_i^{\downarrow})$
      if $a_i > 0$ **then** $\mathsf{r} := \overline{\textbf{replace}}(\mathsf{r}, \mathtt{y}_i^?, \mathtt{d}_i^{\uparrow})$
    $\textbf{result} := \overline{\textbf{sat}}^{\overline{\mathcal{N}}}[\mathsf{r}](\overline{\mathtt{n}})$

Figure 4.4 – Full algorithm for the verification of a constraint

$\overline{\mathtt{o}}$ and an optional variable $\mathtt{y}^?$ that appears in the right hand side, if $\overline{\textbf{sat}}^{\overline{\mathcal{O}}}[\textbf{is\_empty}(\mathtt{y}^?)]\overline{\mathtt{o}}$ (Section 4.2.3.3), $\overline{\textbf{assign}}^{\overline{\mathcal{O}}}[.]$ can safely return $\bot$. The computation of the abstract assignment starts with this check for all optional variables in the right hand side.

Numerical assignment. We first consider a simple assignment $\mathtt{y}_0^? = \mathtt{y}_0^? + \mathtt{y}_1^?$, where $\mathbb{Y}^? = \{\mathtt{y}_0^?, \mathtt{y}_1^?\}$, in order to give some intuition. If $\overline{\mathtt{o}} = (\overline{\mathtt{n}}, \mathcal{A})$ is an abstract pre-condition and $\sigma \in \gamma^{\overline{\mathcal{O}}}(\overline{\mathtt{o}})$ is such that $\sigma(\mathtt{y}_0^?) \neq \emptyset$ and $\sigma(\mathtt{y}_1^?) \neq \emptyset$, there exists a valuation $\nu \in \gamma^{\overline{\mathtt{n}}}(\overline{\mathtt{n}})$ such that $\nu(\mathtt{d}_0^{\downarrow}) = \nu(\mathtt{d}_0^{\uparrow}) = \sigma(\mathtt{y}_0^?)$ and the same for $\mathtt{y}_1^?$. After the assignment evaluates, we obtain a store σ' such that $\sigma'(\mathtt{y}_0^?) = \sigma(\mathtt{y}_0^?) + \sigma(\mathtt{y}_1^?)$ (and is unchanged for all other variables). Therefore, we need to make sure that the abstract post-condition describe a valuation ν' such that $\nu'(\mathtt{d}_0^{\downarrow}) = \nu(\mathtt{d}_0^{\uparrow}) = \sigma(\mathtt{y}_0^?) + \sigma(\mathtt{y}_1^?)$. We can achieve that by performing a pair of assignments to $\mathtt{d}_0^{\downarrow}, \mathtt{d}_1^{\uparrow}$ using *any* combination of avatars to represent $\mathtt{y}_0^?, \mathtt{y}_1^?$ in the right hand side. For instance, the following choices are sound:

$$\begin{cases} \mathtt{d}_0^{\downarrow} = \mathtt{d}_0^{\downarrow} + \mathtt{d}_1^{\downarrow}; \\ \mathtt{d}_0^{\uparrow} = \mathtt{d}_0^{\downarrow} + \mathtt{d}_1^{\downarrow}; \end{cases} \qquad \begin{cases} \mathtt{d}_0^{\downarrow} = \mathtt{d}_0^{\downarrow} + \mathtt{d}_1^{\uparrow}; \\ \mathtt{d}_0^{\uparrow} = \mathtt{d}_0^{\uparrow} + \mathtt{d}_1^{\downarrow}; \end{cases} \qquad \cdots$$

Yet, not all choices are of optimal precision. To show this, we assume that the pre-condition bounds both $\mathtt{y}_0^?$ and $\mathtt{y}_1^?$ from the above, for example with Polyhedra $\overline{\mathtt{n}} = \{\mathtt{d}_0^{\uparrow} \leq 0 \wedge \mathtt{d}_1^{\uparrow} \leq 0\}$. Then, only the left choice will produce a precise upper bound on $\mathtt{d}_0^{\uparrow}$. However, this approach may also produce constraints that violate the bi-avatar principle, such as $\mathtt{d}_0^{\uparrow} - \mathtt{d}_1^{\uparrow} \leq 0$, where $\mathtt{d}_1^{\uparrow}$ gets assigned a lower bound. Such a lower bound can be removed by adding a temporary dimension $\mathtt{d}_t$, assuming that it is positive (using $\overline{\textbf{guard}}^{\overline{\mathcal{O}}}[\mathtt{d}_t \geq 0]$), and performing assignment $\mathtt{d}_1^{\uparrow} = \mathtt{d}_1^{\uparrow} - \mathtt{d}_t$. To conclude, the analysis of assignment $\mathtt{y}_0^? = \sum_{i=0}^{n} a_i \mathtt{y}_i^{\bullet} + \sum_{i=0}^{m} a_{n+i} \mathtt{y}_{n+i}^? + c$ proceeds as follows:

1. $\overline{\textbf{assign}}^{\overline{\mathcal{O}}}[.]$ performs *in parallel* [JM09] the two assignments $\mathtt{d}^{\downarrow} = \mathsf{r}^{\downarrow} \,||\, \mathtt{d}^{\uparrow} = \mathsf{r}^{\uparrow}$, where

$\textbf{make\_exprs}(\sum_{i=0}^{n} a_i \mathtt{y}_i^{\bullet} + \sum_{i=n}^{n+m} a_i \mathtt{y}_i^{?} + c, e)$
  $\mathsf{r} := \sum_{i=0}^{n} a_i \mathtt{y}_i^{\bullet} + \sum_{i=n+1}^{n+m} a_i \mathtt{y}_i^{?} + c;$
  foreach $i \in [0, n]$
    $\mathsf{r} := \overline{\textbf{replace}}(\mathsf{r}, \mathtt{y}_i^{\bullet}, \mathtt{d}_i)$
  $\mathsf{r}' := \mathsf{r}$
  foreach $i \in [n+1, n+m]$
    if $e \bullet a_i > 0$ **then**
      $\mathsf{r} := \overline{\textbf{replace}}(\mathsf{r}, \mathtt{y}_i^{?}, \mathtt{d}_i^{\uparrow})$
      $\mathsf{r}' := \overline{\textbf{replace}}(\mathsf{r}', \mathtt{y}_i^{?}, \mathtt{d}_i^{\downarrow})$
    else
      $\mathsf{r} := \overline{\textbf{replace}}(\mathsf{r}, \mathtt{y}_i^{?}, \mathtt{d}_i^{\downarrow})$
      $\mathsf{r}' := \overline{\textbf{replace}}(\mathsf{r}', \mathtt{y}_i^{?}, \mathtt{d}_i^{\uparrow})$

$\overline{\textbf{assign}}^{\overline{\mathcal{O}}}[\mathtt{y}^{\bullet ?} = \sum_{i=0}^{n} a_i \mathtt{y}_i^{\bullet} + \sum_{i=n+1}^{n+m} a_i \mathtt{y}_i^{?} + c](\overline{\mathtt{n}}, \mathcal{A}) :$
    $\overline{\mathtt{n}} := add\_dims(\overline{\mathtt{n}}, \{\mathtt{d}_i', \mathtt{d}_t\})$
  if $\mathtt{y}^{\bullet ?} \in \mathbb{Y}^{\bullet}$ **then**
    foreach $i \in [n+1, n+m]$
      if $\overline{\textbf{sat}}^{\overline{\mathcal{O}}}[\textbf{is\_empty}(\mathtt{y}_i^{?})](\overline{\mathtt{o}})$
        return $(\bot, \mathcal{A})$;
    $\textbf{make\_exprs}(\sum_{i=0}^{n} a_i \mathtt{y}_i^{\bullet} + \sum_{i=n+1}^{n+m} a_i \mathtt{y}_i^{?} + c, 1);$
    $\overline{\mathtt{n}} := \overline{\textbf{assign}}^{\overline{\mathcal{N}}}[\mathtt{d}_i = \mathsf{r} \,||\, \mathtt{d}' = \mathsf{r}'](\overline{\mathtt{n}})$
    $\overline{\mathtt{n}} := \overline{\textbf{guard}}^{\overline{\mathcal{N}}}[\mathtt{d}_i == \mathtt{d}_i'](\overline{\mathtt{n}})$
  else
    $\textbf{make\_exprs}(\sum_{i=0}^{n} a_i \mathtt{y}_i^{\bullet} + \sum_{i=n+1}^{n+m} a_i \mathtt{y}_i^{?} + c,$
      if $\exists j \in [n+1, n+m], \mathtt{y}_j^{?} = \mathtt{y}^{\bullet ?}$ **then** a_j **else** $1);$
    $\overline{\mathtt{n}} := \overline{\textbf{assign}}^{\overline{\mathcal{N}}}[\mathtt{d}_i^{\uparrow} = \mathsf{r} \,||\, \mathtt{d}_i^{\downarrow} = \mathsf{r}'](\overline{\mathtt{n}})$
    foreach $i \in [n+1, n+m]$
      $\overline{\mathtt{n}} := \overline{\textbf{assign}}^{\overline{\mathcal{N}}}[\mathtt{d}_i^{\uparrow} = \mathtt{d}_i^{\uparrow} - \mathtt{d}_t](\overline{\mathtt{n}})$
      $\overline{\mathtt{n}} := \overline{\textbf{assign}}^{\overline{\mathcal{N}}}[\mathtt{d}_i^{\downarrow} = \mathtt{d}_i^{\downarrow} + \mathtt{d}_t](\overline{\mathtt{n}})$
  result $:= (rem\_dims(\overline{\mathtt{n}}, \{\mathtt{d}_i', \mathtt{d}_t\}), \mathcal{A})$

Figure 4.5 – Assignment transfer function

$\mathtt{r}^{\downarrow}, \mathtt{r}^{\uparrow}$ are obtained from the assignment right hand by substituting $\mathtt{y}_i^?$ with $\mathtt{d}_i^{\downarrow}$ or $\mathtt{d}_i^{\uparrow}$ depending on the sign of the a_is (see below);

2. then it forces the removal of constraints violating the bi-avatar property, using the aforementioned method.

Expression $\mathtt{r}^{\uparrow}$ is defined as $\sum_{i=0}^{n} a_i \mathtt{d}_i + \sum_{i=1}^{m} a_{n+i} \mathtt{d}_{n+i}^{\epsilon_{n+i}} + c$ where avatar signs are determined as follows ($\mathtt{r}^{\downarrow}$ uses the opposite avatar dimensions as $\mathtt{r}^{\uparrow}$):

- if the assignment is not invertible ($\mathtt{y}^?$ does not appear in the right hand side), then ϵ_i is the sign of a_i;
- if the assignment is invertible and $\mathtt{y}^?$ is $\mathtt{y}_{n+1}^?$, then ϵ_i is the sign of the product $a_{n+1}a_i$.

Finally, an assignment with a standard variable $\mathtt{y}^\bullet$ as a left hand side can be handled in a similar manner (after the emptiness test described earlier): it boils down to the introduction of a temporary dimension $\mathtt{d}'$, the analysis of two assignments $\mathtt{d}' = \mathtt{r}^{\uparrow}$ and $\mathtt{d}' = \mathtt{r}^{\downarrow}$ with the above notations, the application of $\overline{\mathbf{guard}}^{\overline{\mathcal{O}}}[\mathtt{d} = \mathtt{d}']$, and finally the removal of $\mathtt{d}'$. By contrast, doing a single assignment would possibly cause relations between $\mathtt{d}'$ and avatars be discarded.

Definition 4.7 (Transfer functions for assignments)**.** *The algorithm for the analysis of assignments* $\overline{\mathbf{assign}}^{\overline{\mathcal{O}}}[.]$ *is shown in Figure 4.5.*

Theorem 4.5 (Soundness of transfer functions for assignments)**.** *If* $\mathtt{y}^{\bullet ?} \in \mathbb{Y}^\bullet \uplus \mathbb{Y}^?$ *and* $\mathtt{r}$ *is a linear expression, then:*

$$\forall \overline{\mathtt{o}} \in \overline{\mathcal{O}},\ [\![\mathtt{y}^{\bullet ?} = \mathtt{r}]\!](\gamma^{\overline{\mathcal{O}}}(\overline{\mathtt{o}})) \subseteq \gamma^{\overline{\mathcal{O}}}(\overline{\mathbf{assign}}^{\overline{\mathcal{O}}}[\mathtt{y}^{\bullet ?} = \mathtt{r}]\overline{\mathtt{o}})$$

Example 4.11 (Transfer functions for assignments)**.** *We assume* $\mathbb{Y}^\bullet = \{\mathtt{y}_0^\bullet\}$, $\mathbb{Y}^? = \{\mathtt{y}_1^?, \mathtt{y}_2^?\}$ *and consider the abstract pre-condition defined by octagon* $\overline{\mathtt{n}} = \{0 \leq \mathtt{d}_1^{\downarrow} \wedge \mathtt{d}_1^{\uparrow} \leq 10 \wedge 0 \leq \mathtt{d}_2^{\downarrow} \wedge \mathtt{d}_2^{\uparrow} \leq 1 + \mathtt{d}_0\}$.

- *non invertible assignment* $\mathtt{y}_1^? = 1 - \mathtt{y}_2^?$ *boils down to parallel assignments* $\mathtt{d}_1^{\uparrow} = 1 - \mathtt{d}_2^{\downarrow} \,||\, \mathtt{d}_1^{\downarrow} = 1 - \mathtt{d}_2^{\uparrow}$ *in Octagons [Min06] and produces numerical post-condition* $\{-\mathtt{d}_0 \leq \mathtt{d}_1^{\downarrow} \wedge \mathtt{d}_1^{\uparrow} \leq 1 \wedge 0 \leq \mathtt{d}_2^{\downarrow} \wedge \mathtt{d}_2^{\uparrow} \leq 1 + \mathtt{d}_0\}$*;*
- *invertible assignment* $\mathtt{y}_1^? = \mathtt{y}_1^? + \mathtt{y}_2^?$ *boils down to parallel assignments* $\mathtt{d}_1^{\uparrow} = \mathtt{d}_1^{\uparrow} + \mathtt{d}_2^{\uparrow} \,||\, \mathtt{d}_1^{\downarrow} = \mathtt{d}_1^{\downarrow} + \mathtt{d}_2^{\downarrow}$*, and produces numerical post-condition* $\{0 \leq \mathtt{d}_1^{\downarrow} \wedge \mathtt{d}_1^{\uparrow} \leq 11 + \mathtt{d}_0 \wedge 0 \leq \mathtt{d}_2^{\downarrow} \wedge \mathtt{d}_2^{\uparrow} \leq 1 + \mathtt{d}_0\}$.

4.2.3.5 Inclusion Checking, Join and Widening

To analyze condition tests and loops, we also need abstract operations for join, widening and inclusion test. Using the bi-avatar principle, these operations can be implemented in a straightforward manner, using the operations of the underlying domain, since avatars are the same for all abstract values. We write $\mathcal{A}$ for the set of avatars defined by the bi-avatar principle in $\mathbb{Y}^\bullet \uplus \mathbb{Y}^?$. We let $\overline{\mathbf{isle}}^{\overline{\mathcal{N}}}$, $\overline{\mathbf{join}}^{\overline{\mathcal{N}}}$, $\overline{\mathbf{widen}}^{\overline{\mathcal{N}}}$ denote the abstract inclusion

check, abstract join and abstract widening of abstract domain $\overline{\mathcal{N}}$, satisfying the following soundness conditions:

$$\forall \overline{\mathtt{n}}_0, \overline{\mathtt{n}}_1 \in \overline{\mathcal{N}},\ \overline{\mathbf{isle}}^{\overline{\mathcal{N}}}(\overline{\mathtt{n}}_0, \overline{\mathtt{n}}_1) = 1 \Longrightarrow \gamma^{\overline{\mathtt{n}}}(\overline{\mathtt{n}}_0) \subseteq \gamma^{\overline{\mathtt{n}}}(\overline{\mathtt{n}}_1)$$
$$\forall \overline{\mathtt{n}}_0, \overline{\mathtt{n}}_1 \in \overline{\mathcal{N}},\ \gamma^{\overline{\mathtt{n}}}(\overline{\mathtt{n}}_0) \cup \gamma^{\overline{\mathtt{n}}}(\overline{\mathtt{n}}_1) \subseteq \gamma^{\overline{\mathtt{n}}}(\overline{\mathbf{join}}^{\overline{\mathcal{N}}}(\overline{\mathtt{n}}_0, \overline{\mathtt{n}}_1))$$
$$\forall \overline{\mathtt{n}}_0, \overline{\mathtt{n}}_1 \in \overline{\mathcal{N}},\ \gamma^{\overline{\mathtt{n}}}(\overline{\mathtt{n}}_0) \cup \gamma^{\overline{\mathtt{n}}}(\overline{\mathtt{n}}_1) \subseteq \gamma^{\overline{\mathtt{n}}}(\overline{\mathbf{widen}}^{\overline{\mathcal{N}}}(\overline{\mathtt{n}}_0, \overline{\mathtt{n}}_1))$$

Furthermore, we assume that $\overline{\mathbf{widen}}^{\overline{\mathcal{N}}}$ ensures convergence of any sequence of abstract iterates [CC77].

Definition 4.8 (Algorithms of inclusion checking, join and widening)**.** *We let the operators over $\overline{\mathcal{O}}$ be defined by:*

$$\begin{array}{rcl}
\overline{\mathbf{isle}}^{\overline{\mathcal{O}}}((\overline{\mathtt{n}}_0, \mathcal{A}), (\overline{\mathtt{n}}_1, \mathcal{A})) & = & \overline{\mathbf{isle}}^{\overline{\mathcal{N}}}(\overline{\mathtt{n}}_0, \overline{\mathtt{n}}_1) \\
\overline{\mathbf{join}}^{\overline{\mathcal{O}}}((\overline{\mathtt{n}}_0, \mathcal{A}), (\overline{\mathtt{n}}_1, \mathcal{A})) & = & (\overline{\mathbf{join}}^{\overline{\mathcal{N}}}(\overline{\mathtt{n}}_0, \overline{\mathtt{n}}_1), \mathcal{A}) \\
\overline{\mathbf{widen}}^{\overline{\mathcal{O}}}((\overline{\mathtt{n}}_0, \mathcal{A}), (\overline{\mathtt{n}}_1, \mathcal{A})) & = & (\overline{\mathbf{widen}}^{\overline{\mathcal{N}}}(\overline{\mathtt{n}}_0, \overline{\mathtt{n}}_1), \mathcal{A})
\end{array}$$

These operators trivially inherit the properties of the operators of $\overline{\mathcal{N}}$:

Theorem 4.6 (Soundness of lattice operators)**.** *Operations $\overline{\mathbf{isle}}^{\overline{\mathcal{O}}}$., $\overline{\mathbf{join}}^{\overline{\mathcal{O}}}$. and $\overline{\mathbf{widen}}^{\overline{\mathcal{O}}}$. satisfy soundness condition of the same form as their underlying counterpart. In particular:*

$$\forall \overline{\mathtt{n}}_0, \overline{\mathtt{n}}_1 \in \overline{\mathcal{N}},\ \gamma^{\overline{\mathtt{n}}}(\overline{\mathtt{n}}_0) \cup \gamma^{\overline{\mathtt{n}}}(\overline{\mathtt{n}}_1) \subseteq \gamma^{\overline{\mathtt{n}}}(\overline{\mathbf{join}}^{\overline{\mathcal{N}}}(\overline{\mathtt{n}}_0, \overline{\mathtt{n}}_1))$$

Moreover, $\overline{\mathbf{widen}}^{\overline{\mathcal{O}}}$. also ensures termination.

4.3 Maya+ Functor

In this section, we first introduce *summarizing abstract numeric domains* [GDD+04]. It is a functor that lifts conventional numeric domains to those describing concrete stores with standard variable (i.e., $\mathbb{Y}^{\bullet}$) and non-empty summary variable (i.e., $\mathbb{Y}^{+}$). Then we show how to compose the Maya functor and the functor of summarizing numeric domains, to produce Maya+ functor. The Maya+ functor extends conventional numeric domains with the ability to constrain possibly-empty summary variables (i.e., $\mathbb{Y}^{*}$).

To simplify the formalization, we make the following conventions.

- The right-value r in a condition test contains only one kind of logical operators. A right-value that contains weak (resp. strong) logical operators are denoted as r_w (resp. r_s).
- The right-value in an assignment $\mathsf{l} = \mathsf{r}$ does not contain logical operators.

$$\mathbf{guard}[\![\bowtie(\mathsf{r}_0,\ldots,\mathsf{r}_k)]\!]_w(\sigma) = \begin{cases} \{1\} & \exists c_i \in [\![\mathsf{r}_i]\!](\sigma), i \in [0,k], \bowtie(c_0,\ldots,c_k) = 1 \\ \emptyset & \exists i \in [0,k],\ [\![\mathsf{r}_i]\!](\mathcal{S}) = \emptyset \\ \{0\} & \textit{otherwise} \end{cases}$$

Figure 4.6 – Weak concrete semantics of condition tests

4.3.1 Summarizing Numeric Domains

The functor of summarizing numeric domains [GDD+04] extends conventional numeric domains with *summarized dimensions*. One summarized dimension $\overline{\mathtt{d}} \in \overline{\mathbb{E}}$ represents a non-empty set of values, and could be seen the summary of several dimensions ($\mathtt{d} \in \mathbb{E}$).

An abstract element $\overline{\mathtt{n}^{\mathtt{s}}}$ in a summarizing numeric domain is a conventional numeric element $\overline{\mathtt{n}}$ where $\mathbf{Dim}(\overline{\mathtt{n}}) \subseteq \mathbb{E} \cup \overline{\mathbb{E}}$. A standard variable is represented by a standard dimension $\mathtt{d}$, and a non-empty summary variable is represented by a summary dimension $\overline{\mathtt{d}}$. We ignore the mapping function from variables to dimensions, and just use the same subscript to indicate a variable and the dimension represents it.

The concretization of $\overline{\mathtt{n}}$ is a set of $\nu^s \in (\mathbb{E} \to \mathbb{V}) \cup (\overline{\mathbb{E}} \to \mathcal{P}(\mathbb{V})\backslash\emptyset)$, which evaluates a standard dimension to a value and a summary dimension to a set of values.

Definition 4.9 (Concretization in summarizing numeric domains)**.** *The concretization of* $\overline{\mathtt{n}^{\mathtt{s}}}$ *is defined by:*

$$\gamma^{\overline{\mathcal{N}}^s}(\overline{\mathtt{n}^{\mathtt{s}}}) \overset{def.}{::=} \left\{ \nu^s \;\middle|\; \begin{array}{l} \forall \nu \in \mathbf{Dim}(\overline{\mathtt{n}}) \to \mathbb{V},\ \textit{if}\ (\forall \mathtt{d} \in \mathbb{E}, \nu^s(\mathtt{d}) = \nu(\mathtt{d})) \\ \wedge(\forall \overline{\mathtt{d}} \in \overline{\mathbb{E}}, \nu^s(\overline{\mathtt{d}}) \in \nu^s(\overline{\mathtt{d}})), \nu \in \gamma^{\overline{\mathtt{n}}}(\overline{\mathtt{n}^{\mathtt{s}}}) \end{array} \right\}$$

Lattice operators in the underlying numeric domains can be safely re-used in the summarizing numeric domain. But the transfer functions for condition tests and assignments need to be redefined. In [GDD+04], the abstract guard operator $\overline{\mathbf{guard}}^{\overline{\mathcal{N}}^s}[.]$ does not correspond to the concrete semantcis of condition test defined in Figure 4.1. In contrast, $\overline{\mathbf{guard}}^{\overline{\mathcal{N}}^s}[.]$ over-approximats the concrete semantics in Figure 4.6.

We call this version of concrete semantics for condition tests *weak tests* (the version in Figure 4.1 is *strong* tests). A weak test evaluates to $\{1\}$ when the logical relation is satisfied by at least one tuple in the Cartesian product of the operand sets. For instance, $\mathbf{guard}[\![\{3,1\} \leq \{2\}]\!]_w$ evaluates to $\{1\}$, but $\mathbf{guard}[\![\{3,1\} \geq \{2\}]\!]$ evaluates to $\{0\}$.

Theorem 4.7 (Soundness of the transfer function for condition tests)**.** *If condition* r *only contains standard and non-empty summary variables, the abstract transfer function* $\overline{\mathbf{guard}}^{\overline{\mathcal{N}}^s}[.]$ *is* sound *in the sense that:*

$$\mathbf{guard}[\![\mathsf{r}]\!]_w(\gamma^{\overline{\mathcal{N}}^s}(\overline{\mathtt{n}^{\mathtt{s}}})) \subseteq \gamma^{\overline{\mathcal{N}}^s}(\overline{\mathbf{guard}}^{\overline{\mathcal{N}}^s}[\mathsf{r}](\overline{\mathtt{n}^{\mathtt{s}}}))$$

In [GDD+04], the transfer function for assignments $\overline{\mathbf{assign}}^{\overline{\mathcal{N}}^s}[.]$ performs weak updates on the left-value when it is a non-empty summary variable.

Theorem 4.8 (Soundness of the transfer function for assignments)**.** *The transfer function for assignments* $\overline{\textbf{assign}}^{\overline{\mathcal{N}}^s}[.]$ *is sound in the sense that:*

$$\forall \overline{\mathtt{n}^{\mathtt{s}}} \in \overline{\mathcal{N}}^s,\ [\![\mathtt{y}^{\bullet+} = \mathtt{r}]\!](\gamma^{\overline{\mathcal{N}}^s}(\overline{\mathtt{n}^{\mathtt{s}}})) \subseteq \gamma^{\overline{\mathcal{N}}^s}(\overline{\textbf{assign}}^{\overline{\mathcal{N}}^s}[\mathtt{y}^{\bullet+} = \mathtt{r}]\overline{\mathtt{n}^{\mathtt{s}}})$$

4.3.2 Composition of Maya Functor and Summarizing Numeric Domains

Our Maya+ functor is obtained by composing the Maya functor and the functor of summarizing numeric domains. The resulting Maya+ domains can abstract stores with standard variables $\mathbb{Y}^\bullet$ and possibly-empty summary variables $\mathbb{Y}^*$. An element in the Maya+ domain is a tuple $(\overline{\mathtt{n}^{\mathtt{s}}}, \mathcal{A})$, where $\overline{\mathtt{n}^{\mathtt{s}}} \in \overline{\mathcal{N}}^s$ and $\mathcal{A} \in (\mathbb{Y}^\bullet \to \mathbb{E}) \cup (\mathbb{Y}^* \to \mathcal{P}(\overline{\mathbb{E}}))$ maps a standard variable into a standard dimension and a possibly-empty summary variable into a set of summary dimensions.

Definition 4.10 (Concretization in the Maya+ domain)**.** *Given an abstract state* $\overline{\mathtt{u}} = (\overline{\mathtt{n}^{\mathtt{s}}}, \mathcal{A})$*, we define the following consistency predicates:*

$$\begin{array}{lll} P_{\mathbb{Y}^\bullet}(\sigma, \overline{\mathtt{u}}, \nu^s) & \overset{def.}{\iff} & \forall \mathtt{y}^\bullet \in \mathbb{Y}^\bullet,\ \sigma(\mathtt{y}^\bullet) = \nu^s(\mathtt{y}^\bullet) \\ P_{\mathbb{Y}^*}(\sigma, \overline{\mathtt{u}}, \nu^s) & \overset{def.}{\iff} & \forall \mathtt{y}^* \in \mathbb{Y}^*,\ \sigma(\mathtt{y}^*) \subseteq \bigcap_{\overline{\mathtt{d}} \in \mathcal{A}(\mathtt{y}^*)} \nu^s(\overline{\mathtt{d}}) \end{array}$$

Then, the concretization of $\overline{\mathtt{u}} = (\overline{\mathtt{n}^{\mathtt{s}}}, \mathcal{A})$ *is defined by:*

$$\gamma^{\overline{\mathcal{U}}}(\overline{\mathtt{u}}) \overset{def.}{::=} \left\{ \sigma \in \mathbb{S} \mid \exists \nu^s \in \gamma^{\overline{\mathcal{N}}^s}(\overline{\mathtt{n}^{\mathtt{s}}}),\ P_{\mathbb{Y}^\bullet}(\sigma, \overline{\mathtt{u}}, \nu^s) \wedge P_{\mathbb{Y}^*}(\sigma, \overline{\mathtt{u}}, \nu^s) \right\}$$

Example 4.12 (Concretization in the Maya+ domain)**.** *We assume* $\mathbb{Y}^\bullet = \{\mathtt{y}_0^\bullet\}$*,* $\mathbb{Y}^* = \{\mathtt{y}_1^{?*}\}$*, and consider the abstract element* $(3 \leq \mathtt{d}_0 \wedge \mathtt{d}_0 \leq 4 \wedge 0 \leq \overline{\mathtt{d}}_1^{\downarrow} \wedge \overline{\mathtt{d}}_1^{\uparrow} \leq \mathtt{x} - 3, \mathcal{A})$ *where* $\mathcal{A}$ *follows the bi-avatar principle. These constraints define valid elements of both Maya and Maya+ domains. However, the concretizations of this abstract element in both domains are different as shown below:*

Maya :

①	$\mathtt{y}_0^\bullet$	$\mapsto$	$\{3\}$	$\mathtt{y}_1^{?*}$	$\mapsto$	$\{0\}$
②	$\mathtt{y}_0^\bullet$	$\mapsto$	$\{3\}$	$\mathtt{y}_1^{?*}$	$\mapsto$	$\emptyset$
③	$\mathtt{y}_0^\bullet$	$\mapsto$	$\{4\}$	$\mathtt{y}_1^{?*}$	$\mapsto$	$\{1\}$
④	$\mathtt{y}_0^\bullet$	$\mapsto$	$\{4\}$	$\mathtt{y}_1^{?*}$	$\mapsto$	$\{0\}$
⑤	$\mathtt{y}_0^\bullet$	$\mapsto$	$\{4\}$	$\mathtt{y}_1^{?*}$	$\mapsto$	$\emptyset$

Maya+ :

①	$\mathtt{y}_0^\bullet$	$\mapsto$	$\{3\}$	$\mathtt{y}_1^{?*}$	$\mapsto$	$\{0\}$
②	$\mathtt{y}_0^\bullet$	$\mapsto$	$\{3\}$	$\mathtt{y}_1^{?*}$	$\mapsto$	$\emptyset$
③	$\mathtt{y}_0^\bullet$	$\mapsto$	$\{4\}$	$\mathtt{y}_1^{?*}$	$\mapsto$	$\{1\}$
④	$\mathtt{y}_0^\bullet$	$\mapsto$	$\{4\}$	$\mathtt{y}_1^{?*}$	$\mapsto$	$\{0\}$
⑤	$\mathtt{y}_0^\bullet$	$\mapsto$	$\{4\}$	$\mathtt{y}_1^{?*}$	$\mapsto$	$\emptyset$
⑥	$\mathtt{y}_0^\bullet$	$\mapsto$	$\{4\}$	$\mathtt{y}_1^{?*}$	$\mapsto$	$\{0,1\}$

The lattice operators in the Maya+ domain are from those in $\overline{\mathcal{N}}^s$. But the transfer functions need to be redefined.

Transfer functions. The transfer functions in the Maya+ domain can be obtained by calling those in the Maya functor and the functor of summarizing numeric domains.

Definition 4.11 (The transfer function for assignments)**.** *The transfer function for assignments* $\overline{\textbf{assign}}^{\overline{\mathcal{U}}}[.]$ *is constructed by replacing all the occurrences of* $\overline{\textbf{assign}}^{\overline{\mathcal{N}}}[.]$ *and* $\overline{\textbf{guard}}^{\overline{\mathcal{N}}}[.]$ *by* $\overline{\textbf{assign}}^{\overline{\mathcal{N}}^s}[.]$ *and* $\overline{\textbf{guard}}^{\overline{\mathcal{N}}^s}[.]$ *in* $\overline{\textbf{assign}}^{\overline{\mathcal{O}}}[.]$.

The soundness of the transfer function for assignments $\overline{\textbf{assign}}^{\overline{\mathcal{U}}}[.]$ follows the soundness of the operators from the two functors.

Theorem 4.9 (Soundness of the transfer function for assignments)**.** *The transfer function for assignments* $\overline{\textbf{assign}}^{\overline{\mathcal{U}}}[.]$ *is sound in the sense that:*

$$\forall \overline{\mathsf{u}} \in \overline{\mathcal{U}},\ [\![\mathsf{y}^{\bullet *} = \mathsf{r}]\!](\gamma^{\overline{\mathcal{U}}}(\overline{\mathsf{u}})) \subseteq \gamma^{\overline{\mathcal{N}}^s}(\overline{\textbf{assign}}^{\overline{\mathcal{U}}}[\mathsf{y}^{\bullet *} = \mathsf{r}]\overline{\mathsf{u}})$$

Definition 4.12 (The transfer function for condition tests)**.** *We define two transfer functions for condition tests in Maya+ domain: one accounts for weak (Figure 4.6) semantics and the other for strong (Figure 4.1) semantics. Indeed, the transfer function for weak condition test in Maya+ domain* $\overline{\textbf{guard}}^{\overline{\mathcal{U}}}[.]_w$ *is obtained by replacing all the occurrences of* $\overline{\textbf{guard}}^{\overline{\mathcal{N}}}[.]$ *by* $\overline{\textbf{guard}}^{\overline{\mathcal{N}}^s}[.]$ *in* $\overline{\textbf{guard}}^{\overline{\mathcal{O}}}[.]$. *The transfer function* $\overline{\textbf{guard}}^{\overline{\mathcal{U}}}[.]$ *for strong tests is the same as* $\overline{\textbf{guard}}^{\overline{\mathcal{O}}}[.]$.

Theorem 4.10 (Soundness of the transfer function for condition tests)**.** *The abstract transfer function* $\overline{\textbf{guard}}^{\overline{\mathcal{U}}}[.]$ *is* sound *in the sense that:*

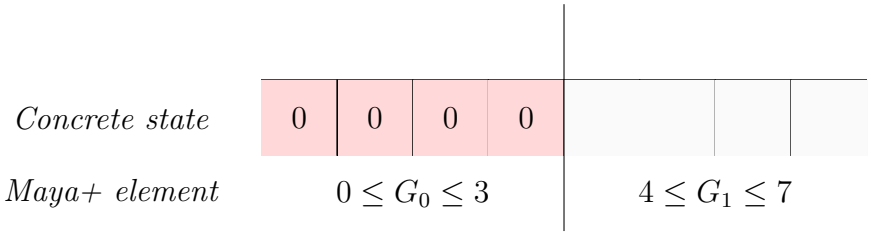
$$\textbf{guard}[\![\mathsf{r}]\!]_w(\gamma^{\overline{\mathcal{U}}}(\overline{\mathsf{u}})) \subseteq \gamma^{\overline{\mathcal{U}}}(\overline{\textbf{guard}}^{\overline{\mathcal{U}}}[\mathsf{r}]_w(\overline{\mathsf{u}}))$$
$$\textbf{guard}[\![\mathsf{r}]\!](\gamma^{\overline{\mathcal{U}}}(\overline{\mathsf{u}})) \subseteq \gamma^{\overline{\mathcal{U}}}(\overline{\textbf{guard}}^{\overline{\mathcal{U}}}[\mathsf{r}](\overline{\mathsf{u}}))$$

4.3.3 Case Study: Application of The Maya+ Functor to A Simple Array Analysis

We have implemented abstract domain functors Maya and Maya+ with the bi-avatar principle (so that they can be applied to numerical abstract domains representing linear inequalities), as well as the analysis of the language of Figure 4.1. To assess its precision, we encode an array analysis on an array initialization example to a program of the language defined in Figure 4.1. Figure 4.7(a) shows a C code program that initializes an array. We consider an array analysis inspired by array partitioning, which proceeds by forward abstract interpretation [CC77]. An observation is that during the iteration of the loop, the array can be divided into two sets of cells, namely initialized cells and uninitialized cells. We consider an abstraction of the array, that partitions it into two groups of cells called: G_0 and G_1 (where all cells in group G_0 are initialized to zero and cells in group G_1 may

```
int i = 0; int a[8];
while(i < 8){
    a[i] = 0;
    i = i + 1;
}
```

(a) An array initialization example



(b) A concrete state and the corresponding abstract states

$\mathbf{int}^{\bullet}$ i $= 0$;

$\mathbf{int}^{*}$ G_0, G_1;

assert(**is_empty**$(G_0) = \{1\}$);

assert$(0 \leq G_1 \leq 7)$;

⓪ $0 \leq \mathtt{i} \wedge \mathtt{i} \leq 0 \wedge 1 \leq G_0^{\downarrow} \wedge G_0^{\uparrow} \leq 0 \wedge 0 \leq G_1^{\downarrow} \wedge G_1^{\uparrow} \leq 7$

while(i < 8){

① $0 \leq \mathtt{i} \wedge \mathtt{i} \leq 7 \wedge 0 \leq G_0^{\downarrow} \wedge G_0^{\uparrow} \leq \mathtt{i} - 1 \wedge \mathtt{i} \leq G_1^{\downarrow} \wedge G_1^{\uparrow} \leq 7$

$G_0 = \mathtt{i}$;

② $0 \leq \mathtt{i} \wedge \mathtt{i} \leq 7 \wedge 0 \leq G_0^{\downarrow} \wedge G_0^{\uparrow} \leq \mathtt{i} \wedge \mathtt{i} \leq G_1^{\downarrow} \wedge G_1^{\uparrow} \leq 7$

assert$(G_1 ! = \mathtt{i})$;

③ $0 \leq \mathtt{i} \wedge \mathtt{i} \leq 7 \wedge 0 \leq G_0^{\downarrow} \wedge G_0^{\uparrow} \leq \mathtt{i} \wedge \mathtt{i} + 1 \leq G_1^{\downarrow} \wedge G_1^{\uparrow} \leq 7$

i $= \mathtt{i} + 1$;

④ $0 \leq \mathtt{i} \wedge \mathtt{i} \leq 7 \wedge 0 \leq G_0^{\downarrow} \wedge G_0^{\uparrow} \leq \mathtt{i} - 1 \wedge \mathtt{i} \leq G_1^{\downarrow} \wedge G_1^{\uparrow} \leq 7$

}

⑤ $8 \leq \mathtt{i} \wedge 0 \leq G_0^{\downarrow} \wedge G_0^{\uparrow} \leq \mathtt{i} - 1 \wedge \mathtt{i} \leq G_1^{\downarrow} \wedge G_1^{\uparrow} \leq 7$

(c) Analysis of the array initialization example: invariants over group indexes

Figure 4.7 – Applicaiton of Maya+ functor on A Simple Array Analysis

Program	LOCs	#Standard	#Summary	Time (ms)	#Assertions	#Verified
array-init	9	1	2	4.7	1	1
array-random-access	30	3	6	36.5	3	3
array-traverse	6	1	1	6.6	1	1
array-compare	10	3	2	14.1	1	1

Figure 4.8 – Analysis results

hold any value), and we also use the group names G_0, G_1 to denote two possibly-empty summary variables that over-approximate the sets of indexes corresponding to the cells of the two groups. Figure 4.7(b) shows a concrete state on the array and the corresponding abstract state in Maya+ domain. The array analysis on the array initialization example could be encoded into the numeric analysis on the program in Figure 4.7(c).

In Figure 4.7(c), $\mathbb{Y}^* = \{G_0, G_1\}$ and $\mathbb{Y}^\bullet = \{\mathtt{i}\}$. In the analysis, the polyhedra domain is used as parameter domain and $\mathcal{A}$ is defined according to the bi-avatar principle ($\mathcal{A}(G_j) = \{G_j^\downarrow, G_j^\uparrow\}$ —note these are all *summary* dimensions, since a group of cells may span several indexes).

At point ⓪, group G_1 contains all the elements of the array (uninitialized elements) and G_0 is empty (initialized elements). The weak update $G_0 = \mathtt{i}$ and statement **assert**($G_1 \neq \mathtt{i}$) stem from the assignment $\mathtt{a}[\mathtt{i}] = 0$ in the array program (Figure 4.7(a)). Note that weak update $G_0 = \mathtt{i}$ just adds the value represented by $\mathtt{i}$ to set G_0. They are analyzed by $\overline{\textbf{assign}}^{\overline{\mathcal{U}}}[.]$ and $\overline{\textbf{guard}}^{\overline{\mathcal{U}}}[.]$, and effectively extend group G_0 and shrink group G_1 by one cell. The loop exit invariant shown at point ⑤ defines stores where G_1 is mapped to no value, which indeed means that the group of uninitialized cells is empty. This actually means that the analysis proves the whole array is initialized to 0. The resulting invariants are shown in Figure 4.7(c).

The analysis was run on a few similar programs encoding the steps that [LR15] needs to achieve to verify array programs, and the results are shown in Figure 4.8. The columns show numbers of lines of codes, standard variables, possibly-empty summary variables, run-time, total numbers of assertions and numbers of verified assertions. Test case "array-init" is shown in Fig 4.7(c). Test cases "array-random-access", "array-traverse" and "array-compare" simulate the array analysis on programs of corresponding algorithm. The analyses are performed with Polyhedra as underlying domain and succeed in computing all invariants (the value range of array contents) required for the verification of these program. Last, the invariants produced express relations between groups, even when those could be empty.

4.4 Related Work and Conclusion

Abstractions based on summary dimensions [GDD+04, SS12] extend basic numerical domains to abstract vectors of non empty sets, so that one dimension may describe an unbounded family of variables. Summaries are also used in shape analysis [SRW99a], with a similar semantics. Empty summaries can be dealt with using disjunctions.

Siegel and Simon [SMS13] abstract dynamic stores, where the set of memory cells is dynamic, and also utilize summary dimensions. In this work, a summary variable may also denote an empty set of values. To abstract precisely which dimension may be empty, a flag is associated to each summary variable, and it is true if and only if the variable is defined to at least one value. This approach allows to express relations between the emptiness of distinct variables. However, it does not allow to infer that a variable is undefined from conflicting constraints over its value (as needed in, e.g., [LR15]). This approach is thus orthogonal to ours, and both techniques could actually be combined. Another technique [CCS15, LR15] uses a conjunction of numerical abstract elements $\overline{\mathtt{n}}_0, \ldots, \overline{\mathtt{n}}_p$ such that a group of variables that should either all be empty or all be defined to a non empty value are constrained together in a same $\overline{\mathtt{n}}_i$. While this approach tracks emptiness precisely and without disjunctions, it is fairly *ad hoc* and expresses no relational constraints across groups.

Last, we note that other works on numerical abstract domains use several dimensions in the abstract domain so as to constrain a single variable. For instance, the implementation of octagons on top of DBMs lets a variable $\mathtt{x}$ be described in a DBM by dimensions $\mathtt{x}^+ = \mathtt{x}$ and $\mathtt{x}^- = -\mathtt{x}$ (so that $\mathtt{x} = \frac{1}{2}(\mathtt{x}^+ - \mathtt{x}^-)$) [Min06].

We have proposed the Maya functor to lift numerical abstract domains into abstractions for sets of stores where some variables may be undefined, and a functor Maya+ that performs the same task in presence of possibly empty summary dimensions. We have fully described the design of abstract operations using a "bi-avatar" principle, that allows to cope with abstract domains based on linear inequalities. These two functors help achieve the goal set in Section 4.1.

Our construction can be applied either to analyze languages that allow optional values, or as a back-end for static analyses that rely on groups of locations to describe complex memories (such as array and shape analyses). In the next Chapter, we will use Maya+ functor in an array analysis in a way that pretty much like that in Section 4.3.3.

Future work should focus on additional strategies, for instance, based on the multi-avatar principle (Example 4.7), to accommodate other kinds of numerical abstract domains.

Chapter 5

Non-contiguous Partitioning

Conventional array partitioning analyses split arrays into contiguous segments to infer properties of sets of array cells. Such analyses cannot group together non contiguous cells, even when they have similar properties. In this chapter, we propose an abstract domain which utilizes semantic properties to split array cells into groups. Cells with similar properties will be packed into a same group and abstracted together. Additionally, a group denotes a set of cells that 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 academic test cases demonstrate the effectiveness of the analysis.

5.1 Context of Non-contiguous Partitioning

In this section, we set the context for our non-contiguous partitioning domain. We first extend the language defined in Figure 3.1 to support array types, and then recall the motivation of our non-contiguous partitioning.

5.1.1 Extension of the Language

The syntax and semantics of the extension is shown in Figure 5.1.

Syntax. We let $\mathbb{I}$ denote the set of non-negative integers and $\mathbb{F}$ denote the set of fields. We extend the language in Figure 3.1 with a composite type $\textbf{struct}\{\textbf{int}\ \ \mathtt{f}; \ldots; \textbf{int}\ \ \mathtt{f}\}[k]$ which describes arrays of structures. Variables of this type are denoted by $\mathbb{A} \ni \mathtt{a}$. This language also allows variables of structure type (they are considered arrays of length 1), and arrays of primitive 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 $\mathtt{a}[\mathtt{x}]$, where $\mathtt{x}$ is an integer variable, which simplifies 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

$$\mathbb{F} : \text{fields}(\mathtt{f} \in \mathbb{F}) \qquad \mathbb{A} : \text{structural type variables}(\mathtt{a} \in \mathbb{A})$$
$$\mathbb{I} : \text{Non-negative integers}(k \in \mathbb{I})$$

$$\begin{array}{rcll} \mathbf{T} & ::= & \mathbf{int} & \text{primitive type} \\ & | & \mathbf{struct}\{\mathbf{int}\ \mathtt{f}; \ldots; \mathbf{int}\ \mathtt{f}\}[k] & \text{structural type} \\ \mathsf{l} & ::= & \mathtt{a}[\mathtt{x}].\mathtt{f} \mid \mathtt{x} & \text{left value expressions} \end{array}$$

(a) Syntax

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

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

Evaluation of R-values: $\mathbb{S} \to \mathbb{V}$

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

(b) Semantics

Figure 5.1 – Extension of the language with composite type

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.

Concrete states and semantics. Because of the extension in the syntax, we need to redefine the concrete states and semantics.

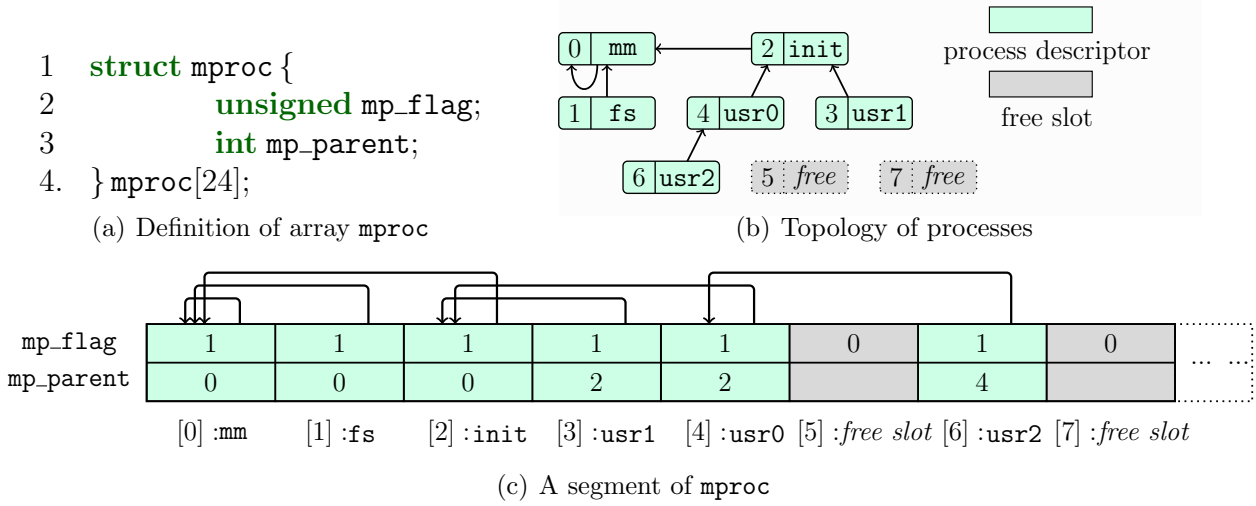
Definition 5.1 (Concrete states)**.** *In this chapter, a concrete state σ is a partial function mapping basic cells (base variables and fields of array cells) into values (which are denoted by $\mathbb{V}$). The set $\mathbb{S}$ of concrete states is defined by*

$$\sigma \in \mathbb{S} = (\mathbb{A} \times \mathbb{I} \times \mathbb{F} \cup \mathbb{X}) \to \mathbb{V}$$

Specifically, the set of all fields of cells of array $\mathtt{a}$ is denoted by $\mathbb{F}_{\mathtt{a}}$, and the set of valid indexes in $\mathtt{a}$ is denoted by $\mathbb{I}_{\mathtt{a}}$. The semantics of l-values and r-values are also extended to account for array accesses.

5.1.2 An Example from Minix.

To recall the motivation for our non-contiguous partitioning domain, we show an example from Minix 1.1, which will be used throughout this paper.



(a) Definition of array `mproc`

(b) Topology of processes

(c) A segment of `mproc`

Figure 5.2 – Minix 1.1 Memory Management Process Table (MMPT) structure

Memory management in Minix. In Minix 1.1, the component of memory management maintains a *process table* that describes the processes currently running. Figure 5.2 illustrates the Memory Management Process Table (MMPT) main structure. The array of structures `mproc` defined in Figure 5.2(a) stores the process descriptors. Each descriptor comprises a field `mp_parent` that stores the index of the parent process in `mproc`, and a field `mp_flag` that stores the process status. An element of `mproc` is a process descriptor when its field `mp_flag` is strictly positive and a free slot if it is null. As in all Unix operating systems, processes form a reversed tree, where each process has a reference to its parent (namely, the process that created it) and is referred to by its children (the process that it has created). Figure 5.2(c) depicts the concrete values stored in `mproc` to describe the process topology shown in Figure 6.1(b) (the whole `mproc` table consists of 24 slots, here we show only 8, for the sake of space). 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 concrete state of Figure 5.2(c), `init` has two children whose descriptors are in `mproc`[3] and `mproc`[4]; similarly, the process corresponding 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 [GRS05, CCL11] 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 in the program state. The symbolic abstract domain by Dillig, Dillig and Aiken [DDA10] also fails here as it cannot attach arbitrary abstract

```
1  void cleanup (int child){
2              int parent = mproc[child].mp_parent;
3              if( parent == 2 ){
4                 mproc[child].mpflag = 0;
5                 i = 0;
6                 while(i < 24){
7                   if(mproc[i].mpflag > 0 )
8                     if( mproc[i].mp_parent == child )
9                       mproc[i].mp_parent = 2;
10                  i = i + 1;
11                }
12             } else {
13               \\ cleanup child and its descendants
14                 }
15 }
```

Figure 5.3 – A simplified excerpt of `cleanup`

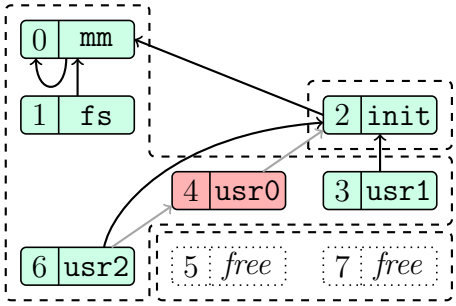


Figure 5.4 – Effect of `cleanup`

properties to summarized cells.

Invariants. Our non-contiguous partitioning domain can abstract the process table state by partitioning valid process descriptors and free slots into *different groups*. To be more specific, it can abstract the following global invariants.

- Each valid process descriptor (resp. free slot) has an `mp_flag` which stores a strictly positive value (resp. null).
- Each valid process descriptor has an `mp_parent` field, that should store a value in $[0, 23]$ (since the length of array `mproc` is 24), It represents a valid index in `mproc`. This entails the absence of out-of-bound accesses in process table management functions.

- The `mp_parent` 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 denote the conjunction of these properties as $\mathcal{R}^{minix}$. It is necessary for the verification of the *memory safety* of the operations on the Minix memory management process table (and of other similar process tables).

System calls. 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 such that the process and its parent are set to be "hanging" and "waiting" respectively when they reach the barrier first. In Figure 6.1(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. Function `mm_init` is called when the operating system is initialized and constructs slots in `mproc` for the first three system level processes.

Our non-contiguous partitioning domain needs to verify that (1) initialization function `mm_init` establishes $\mathcal{R}^{minix}$ and that (2) system calls `fork`, `wait` and `exit` preserve $\mathcal{R}^{minix}$. To achieve this, we design a fully automatic, abstract interpretation-based static analysis.

As an example, in the remainder of this chapter, we focus on an 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 5.3 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 5.2(c): indeed, this will cause the removal of user process `usr0` the parent of which is `init`; 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 5.4.

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. This means that function `cleanup` should always be called in a state that satisfies the precondition defined by the correctness condition $\mathcal{R}^{minix}$. To verify this automatically, our analysis shall abstract $\mathcal{R}^{minix}$ and perform a forward abstract interpretation of `cleanup`, computing sound post-conditions and loop invariants [CC77].

5.2 Abstraction

In this section, we formalize the abstract elements in our domain and their concretization. An abstract element consists of a combination of memory predicates and numeric

predicates.

5.2.1 Memory Predicates

To highlight the main idea of our domain, we assume that there is only one array `a` in the concrete states. In an abstract element, the memory predicate $\overline{\mathtt{P}}$ partitions the array `a` into *groups* of cells, which may be non-contiguous, unlike array partitioning based on segments such as [GRS05, CCL11].

Definition 5.2 (Memory predicates)**.** *We use* `f` *to denote fields in* $\mathbb{F}$*, a memory predicate* $\overline{\mathtt{P}}$ *may be an atomic memory predicate* $G_i@\vec{\mathtt{f}} \mapsto I^{\vec{\mathtt{f}}}_{G_i}$ *or separating conjunctions of atomic memory predicates, as defined below.*

$$\begin{array}{rcll} \overline{\mathtt{P}} & := & G_i@\vec{\mathtt{f}} \mapsto I^{\vec{\mathtt{f}}}_{G_i} & \textit{Predicates on a possibly empty group of cells} \\ & | & \overline{\mathtt{P}} * \overline{\mathtt{P}} & \textit{Separating conjunction} \end{array}$$

An atomic memory predicate $G_i@\vec{\mathtt{f}} \mapsto I^{\vec{\mathtt{f}}}_{G_i}$ describes a possibly empty group of array cells. In this predicate, the group name is denoted by G_i (where i is a sequential number), which is also a *set variable* (representing a set of values) that abstracts the indexes in the group. A vector of set variables $I^{\vec{\mathtt{f}}}_{G_i}$ abstracts the values stored in each field in the group. Since all atomic memory predicates in our domain follow the same style, we usually use $\mathcal{G}_i$ to abbreviate $G_i@\vec{\mathtt{f}} \mapsto I^{\vec{\mathtt{f}}}_{G_i}$. We write $\mathbb{G}$ for the set of group names. The memory predicates heavily utilize set variables to denote group indexes as well as contents. That is because array cells in a given group are not necessarily contiguous, thus upper / lower bounds could not precisely describe the set of indexes in a group.

The separating conjunction of two memory predicates $\overline{\mathtt{P}}_0 * \overline{\mathtt{P}}_1$ means that $\overline{\mathtt{P}}_0$ and $\overline{\mathtt{P}}_1$ describe disjoint memory locations, and that $\overline{\mathtt{P}}_0 * \overline{\mathtt{P}}_1$ describes the union of the sets of locations described by either of them. Therefore, a conjunction of memory predicates that constrain the whole array `a` represents a possibly non-contiguous partition of it.

Example 5.1 (Memory predicates)**.** *The array in the concrete state in Figure 5.2(c) can be partitioned into two groups: valid process descriptors (group* G_0*) and free slots (group* G_1*). The memory predicate in that example can be described as follows.*

$$\begin{array}{rl} & G_0@\{\mathtt{mp\_flag} \mapsto I^{\mathtt{mp\_flag}}_{G_0}, \mathtt{mp\_parent} \mapsto I^{\mathtt{mp\_parent}}_{G_0}\} \\ * & G_1@\{\mathtt{mp\_parent} \mapsto I^{\mathtt{mp\_parent}}_{G_1}, \mathtt{mp\_parent} \mapsto I^{\mathtt{mp\_parent}}_{G_1}\} \end{array}$$

If we use abbreviation, this memory predicate can be denoted as $\mathcal{G}_0 * \mathcal{G}_1$*.*

5.2.2 Numeric Predicates

To characterize each group, our domain utilizes numeric predicates to describe numeric properties of each group.

Definition 5.3 (Numeric predicates)**.** *In this chapter, we let* $c \in \mathbb{V}$ *denote values. Atandard variables from programs and set variables from memory predicates* $\overline{\mathtt{P}}$ *are denoted by* $\mathtt{x} \in \mathbb{X}$ *and* $\mathtt{y} \in \mathbb{Y}$ *respectively.*

Numeric predicates are composed of two parts: set constraints *and* pure-numeric predicates*, where set constraints are defined as*

$$\overline{\mathtt{g}} ::= \mathtt{x} \in G_{i_0} \uplus \ldots \uplus G_{i_n} \mid G_{i_0} \subseteq G_{i_1} \uplus \ldots \uplus G_{i_n} \mid \overline{\mathtt{g}} \wedge \overline{\mathtt{g}}$$

and pure-numeric predicates are defined as

$$\overline{\mathtt{u}} ::= c_0\mathtt{x}_0 + \ldots + c_{m-1}\mathtt{x}_{m-1} + c'_0\mathtt{y}_0 + \ldots + c'_{n-1}\mathtt{y}_{n-1} \leq c'' \mid \mathtt{x} \neq c \mid \mathtt{x}_0 \neq \mathtt{x}_1 \mid \overline{\mathtt{u}} \wedge \overline{\mathtt{u}}$$

Thus numeric predicates $\overline{\mathtt{Q}}$ *can be defined as* $\overline{\mathtt{g}} \wedge \overline{\mathtt{u}}$

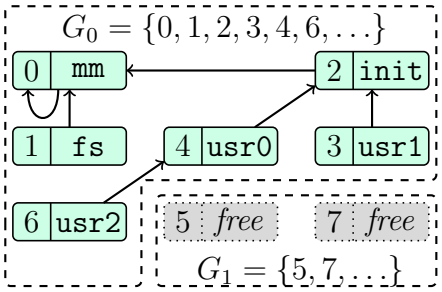
A set constraint $\overline{\mathtt{g}}$ can be a basic var-set constraint $\mathtt{x} \in G_{i_0} \uplus \ldots \uplus G_{i_n}$, which states that the value of variable $\mathtt{x}$ lies in one of the groups $G_{i_0}, \ldots, G_{i_n}$, or a basic set-set constraint $G_{i_0} \subseteq G_{i_1} \uplus \ldots \uplus G_{i_n}$, which states that the cells in group G_{i_0} is in the disjoint union of groups $G_{i_1}, \ldots, G_{i_n}$, or a conjunction of basic set constraints.

A pure-numeric constraint $\overline{\mathtt{u}}$ can be a disequality relation, an inequality relation or a conjunction of such relations. Pure-numeric constraints can be applied on both set variables $\mathtt{y}$ (i.e., set variables introduced by memory predicates) and standard variables $\mathtt{x}$ (i.e., variables that represent one value, like program variables in $\mathbb{X}$). Set variables $\mathtt{y}$ like G_i in a memory predicate, denote sets of values. An inequality on set variables means that, any value in the sets represented by these variables should satisfy the constraint. For example, $0 \leq G_0 \leq 99$ means that G_0 only contains indexes between 0 and 99 (i.e., the semantics of arithmetic/logical operators is the same with that in Chapter 4).

The representation of such constraints requires a numeric domain that can abstract both standard variables and set variables. The *Maya+ domain functor* of Chapter 4 can describe such numerical constraints. However it cannot express several properties that are required in our domain, such as the fact that a set variable describes a non empty set, or a disequality between a set variable and a non-set variable. Therefore, our domain uses an extension of Maya+ in Chapter 4 that can express such constraints: first, the domain attaches a *cardinality variable* $|G_i|$ to group G_i so as to represent its number of elements; second, it relies on a reduced product [CC77] with a domain expressing only disequality constraints. We still use the original notations (e.g., a pure-numeric constraint is denoted with a Maya+ abstract element $\overline{\mathtt{u}}$) of *Maya+ domain functor* in this chapter to denote this extension of Maya+.

Example 5.2 (Numeric predicates)**.** *Let us assume set variable* G *corresponds exactly to* $\{0, 1\}$*. This can be represented exactly using pure-numeric constraints* $0 \leq G \leq 1$ *and* $|G| = 2$*.*

Definition 5.4 (Abstract states in the array domain)**.** *An abstract state* $\overline{\mathtt{a}} \in \overline{\mathcal{H}}$ *is a tuple* $(\overline{\mathtt{P}}, \overline{\mathtt{Q}})$ *where* $\overline{\mathtt{Q}}$ *constrains set variables from* $\overline{\mathtt{P}}$ *and program variables.*



(a) The topology of the partition on `mproc`

Memory: $\mathcal{G}_0$ | $\mathcal{G}_1$

Numeric:
$$0 \leq |G_0| \leq 24 \wedge |G_0| + |G_1| = 24 \wedge 0 \leq G_0 \leq 23$$
$$\wedge 1 \leq I_{G_0}^{\texttt{mp\_flag}} \leq 63 \wedge I_{G_1}^{\texttt{mp\_flag}} = 0$$
$$\wedge 0 \leq I_{G_0}^{\texttt{mp\_parent}} \leq 23 \wedge I_{G_0}^{\texttt{mp\_parent}} \subseteq G_0$$

(b) Abstract state $\mathcal{R}^{minix}$ with partitioning

Figure 5.5 – A partitioning of `mproc` based on non contiguous groups

Abstraction of Invariant $\mathcal{R}^{minix}$ in MMPT. To capture the properties expressed by $\mathcal{R}^{minix}$, our domain 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. The topology of this partition is shown in Figure 5.5(a).

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 5.5(b). According to $\mathcal{R}^{minix}$, the elements of group 0 satisfy the following correctness conditions.

- Their indexes are in $[0, 23]$, which we note $0 \leq G_0 \leq 23$.
- The size of group 0 is between 0 and 24, which we note $0 \leq |G_0| \leq 24$.
- Their flag fields are in $[1, 63]$, which we note $1 \leq I_{G_0}^{\texttt{mp\_flag}} \leq 63$ (Field `mp_flag` uses 6 bits to indicate the state of that cell, and valid process descriptors have a strictly positive flag).
- Their parents are valid indexes, which we note $0 \leq I_{G_0}^{\texttt{mp\_parent}} \leq 23$.
- Their parent fields are indexes of valid process descriptors, hence are also in group 0, which we note $I_{G_0}^{\texttt{mp\_parent}} \subseteq G_0$.

This 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 5.5(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 `mp_flag` 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 5.5(b) is close to the programmer's intent, where the array is a collection of unsorted elements.

Concretization. To define the concretization of abstract states, we first define the concretization of numeric predicates, which maps $\overline{\mathtt{Q}}$ to a set of valuations. A valuation is a function $\nu \in \mathbb{X} \to \mathbb{V} \cup \mathbb{Y} \to \mathcal{P}(\mathbb{V})$, that maps standard variables to values and set variables to sets of values. The concretization of $\overline{\mathtt{Q}}$ is a set of valuations that satisfy all constraints in $\overline{\mathtt{Q}}$.

Definition 5.5 (Concretization of numeric predicates)**.** *This concretization $\gamma^{\overline{\mathtt{Q}}}$ of numeric predicates is expressed using a relation $\models$ as follows.*

$$\gamma^{\overline{\mathtt{Q}}}(\overline{\mathtt{Q}}) = \{\nu \mid \nu \models \overline{\mathtt{Q}} \wedge (\forall G_i \in \mathbb{G}, |\nu(G_i)| = \nu(|G_i|))\}$$

$$\begin{array}{lcl}
\nu & \models & \mathtt{x} \in G_{i_0} \uplus \ldots \uplus G_{i_n} \quad \mathtt{iff} \quad \exists k \in \{0, \ldots, n\},\ \nu(\mathtt{x}) \in \nu(G_{i_k}) \\
\nu & \models & G_{i_0} \subseteq G_{i_1} \uplus \ldots \uplus G_{i_n} \quad \mathtt{iff} \quad \nu(G_0) \subseteq \biguplus_{1 \le k \le n} \nu(G_{i_k}) \\
\nu & \models & c_0\mathtt{x}_0 + \ldots + c_{m-1}\mathtt{x}_{m-1} + c'_0\mathtt{y}_0 + \ldots + c'_{n-1}\mathtt{y}_{n-1} \le c'' \\
 & & \quad \mathtt{iff} \quad \forall v_0 \in \nu(\mathtt{y}_0), \ldots, v_{n-1} \in \nu(\mathtt{y}_{n-1}), \\
 & & \qquad c_0\nu(\mathtt{x}_0) + \ldots + c_{m-1}\nu(\mathtt{x}_{m-1}) + c'_0 v_0 + \ldots + c'_{n-1} v_{n-1} \le c'' \\
\nu & \models & \mathtt{x}_0 \neq \mathtt{x}_1 \quad \mathtt{iff} \quad \nu(\mathtt{x}_0) \neq \nu(\mathtt{x}_1) \\
\nu & \models & \mathtt{x} \neq c \quad \mathtt{iff} \quad \nu(\mathtt{x}) \neq c \\
\nu & \models & \overline{\mathtt{u}}_0 \wedge \overline{\mathtt{u}}_1 \quad \mathtt{iff} \quad \nu \models \overline{\mathtt{u}}_0 \text{ and } \nu \models \overline{\mathtt{u}}_1 \\
\nu & \models & \overline{\mathtt{g}}_0 \wedge \overline{\mathtt{g}}_1 \quad \mathtt{iff} \quad \nu \models \overline{\mathtt{g}}_0 \text{ and } \nu \models \overline{\mathtt{g}}_1 \\
\nu & \models & \overline{\mathtt{g}} \wedge \overline{\mathtt{u}} \quad \mathtt{iff} \quad \nu \models \overline{\mathtt{g}} \text{ and } \nu \models \overline{\mathtt{u}}
\end{array}$$

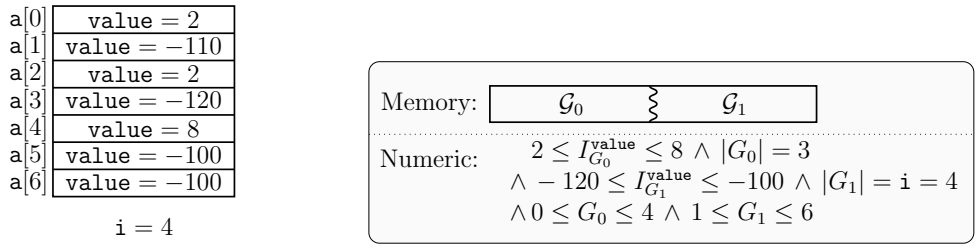
Definition 5.6 (Concretization of abstract states in the array domain)**.** *This concretization γ^a of abstract states maps an abstract state in the domain to a set of concrete states and valuations that satisfy both the memory and numeric predicates, as shown below.*

$$\gamma^a(\overline{\mathtt{P}}, \overline{\mathtt{Q}}) = \{\sigma \in \mathbb{S} \mid \exists \nu \in \gamma^{\overline{\mathtt{Q}}}(\overline{\mathtt{Q}}),\ \sigma \models (\overline{\mathtt{P}}, \nu) \wedge \forall \mathtt{x} \in \mathbb{X},\ \nu(\mathtt{x}) = \sigma(\mathtt{x})\}$$

$$\sigma \models (G_i @ \vec{\mathtt{f}} \mapsto I^{\vec{\mathtt{f}}}_{G_i}, \nu) \quad \mathtt{iff} \quad \forall \mathtt{f} \in \mathbb{F}, \forall j \in \nu(G_i),\ \sigma(\mathtt{a}[j] \cdot \mathtt{f}) \in \nu(I^{\mathtt{f}}_i)$$

$$\sigma_0 * \sigma_1 \models (\overline{\mathtt{P}}_0 * \overline{\mathtt{P}}_1, \nu) \quad \mathtt{iff} \quad \sigma_0 \models (\overline{\mathtt{P}}_0, \nu) \wedge \sigma_1 \models (\overline{\mathtt{P}}_1, \nu)$$

Example 5.3 (Concretization of abstract states in the array domain)**.** *Given a concrete state of array* `a` *in Figure 5.6(a), an abstract element in our domain that over-approximates the concrete state is shown in Figure 5.6(b). Group G_0 (resp., G_1) comprises all the cells that store positive (resp., negative) values, the numeric predicates reveal that all three positive values are stored in the first five cells of the array.*



(a) Concrete array a (b) Abstract state $\overline{\texttt{a}}$

Figure 5.6 – An concrete state and a corresponding abstact state

If we concretize the abstract element in Fig 5.6(b), then one tuple in the concretization result is shown as below.

$$\nu: \begin{array}{rcl} G_0 & \mapsto & \{0,2,4\} \\ G_1 & \mapsto & \{1,3,5,6\} \\ |G_0| & \mapsto & 3 \\ |G_1| & \mapsto & 4 \\ I_{G_0}^{\texttt{value}} & \mapsto & \{2,8\} \\ I_{G_1}^{\texttt{value}} & \mapsto & \{-100,-110,-120\} \end{array} \qquad \sigma: \begin{array}{rcl} (\texttt{a},0,\texttt{value}) & \mapsto & 2 \\ (\texttt{a},1,\texttt{value}) & \mapsto & -110 \\ (\texttt{a},2,\texttt{value}) & \mapsto & 2 \\ (\texttt{a},3,\texttt{value}) & \mapsto & -120 \\ (\texttt{a},4,\texttt{value}) & \mapsto & 8 \\ (\texttt{a},5,\texttt{value}) & \mapsto & -100 \\ (\texttt{a},6,\texttt{value}) & \mapsto & -100 \end{array}$$

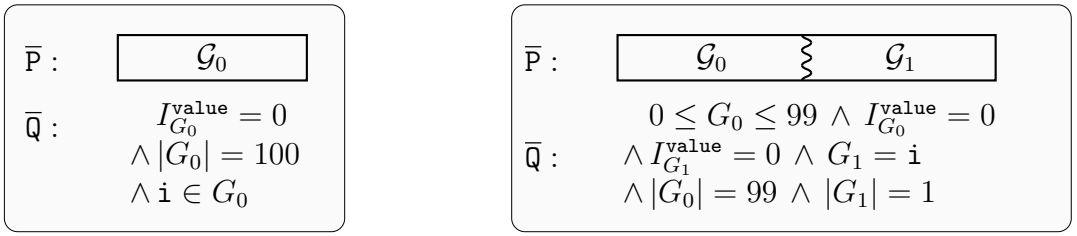
5.3 Basic Operators on Partitions

In this section, we define a set of basic operations on partitions, that abstract transfer functions and lattice operators will use in order 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 $\overline{\textbf{split}}$.

Operator $\overline{\textbf{split}}$ applies to an abstract state $\overline{\texttt{a}}$, an array a and a group G_i corresponding to array a 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 $\overline{\texttt{a}} = (\overline{\texttt{P}}, \overline{\texttt{Q}})$, and with the above notations, $\overline{\textbf{split}}$ performs the following actions:

- It extends $\overline{\texttt{P}}$ with memory predicate $\mathcal{G}_j$ on the fresh group G_j.
- The pure-numeric predicates on indexes and fields of group G_j are inherited from those of G_i, and every occurrence of $|G_i|$ is replaced by $|G_i| + |G_j|$.
- The set relation predicates on G_j are inherited from those on G_i.



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

Figure 5.7 – Partition splitting in array a from abstract state $\overline{\mathtt{a}}$

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

Example 5.4 (The splitting operator)**.** *Figure 5.7(a) defines an abstract state* $(\overline{\mathtt{P}}, \overline{\mathtt{Q}})$ *with a single array, fully initialized to 0, and represented by a single group. Applying operator* $\overline{\mathbf{split}}$ *to that abstract state and to index* i *produces the abstract state of Figure 5.7(b), where* G_1 *is a group with exactly one element, with the same constraints on field* value *as in the previous state.*

Theorem 5.1 (Soundness of the splitting operator)**.** *Suppose* $\overline{\mathtt{a}}$ *is an abstract state,* G_i *a group and* a *an array, the operator* $\overline{\mathbf{split}}$ *is sound in the sense that*

$$\gamma^a(\overline{\mathtt{a}}) \subseteq \gamma^a(\overline{\mathbf{split}}(\overline{\mathtt{a}}, \mathtt{a}, G_i))$$

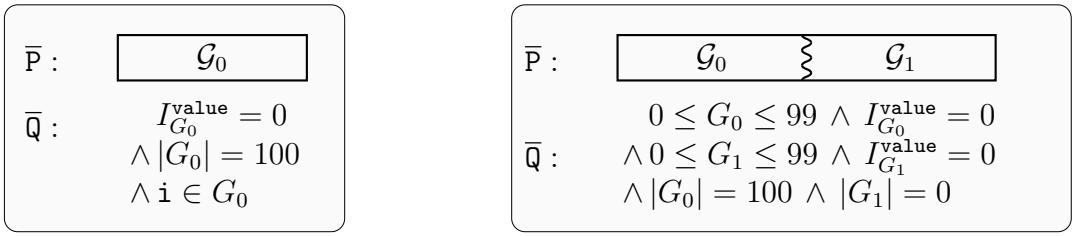
Proof. Let $\overline{\mathtt{a}} = (\overline{\mathtt{P}}, \overline{\mathtt{Q}})$ be an abstract state, and G_i be a group of a in $\overline{\mathtt{a}}$. We assume that splitting G_i in $\overline{\mathtt{a}}$ produces groups $G_{i'}, G_k$ ($G_{i'}$ is actually G_i in the output, we add a superscript to distinguish it from the G_i in the input) in $\overline{\mathtt{a}}' = \overline{\mathbf{split}}(\overline{\mathtt{a}}, \mathtt{a}, G_i)$. Let $\sigma \in \gamma^a(\overline{\mathtt{a}})$. We write ν for the witnesses of $\sigma \in \gamma^a(\overline{\mathtt{a}})$ in the definition of Figure **??**.

Then, we define ν' from ν by:

- fixing $\nu'(G_{i'})$ and $\nu'(G_k)$ so that $\nu'(G_{i'}) \cup \nu'(G_k) = \nu(G_i)$;
- adding set variables (fields, size, index) for groups $G_{i'}, G_k$, that inherit from the values of the variables corresponding to G_i:
  - ν' maps $|G_{i'}|$ and $|G_k|$ to the respective sizes of $\nu'(G_{i'})$ and $\nu'(G_k)$;
  - other variables 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 ν' are witnesses of $\sigma \in \gamma^a(\overline{\mathtt{a}}')$, we simply need to check the implication relations in the concretization function.

This proves the soundness of $\overline{\mathbf{split}}$. □



(a) $\overline{\mathtt{a}}$ (b) $\overline{\mathbf{create}}(\overline{\mathtt{a}})$

Figure 5.8 – Partition creation in array a from abstract state $\overline{\mathtt{a}}$

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 of an empty group. Operator $\overline{\mathbf{create}}$ 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 $\overline{\mathtt{a}} = (\overline{\mathtt{P}}, \overline{\mathtt{Q}})$ and to array variable a, operator $\overline{\mathbf{create}}$ performs the following operations:

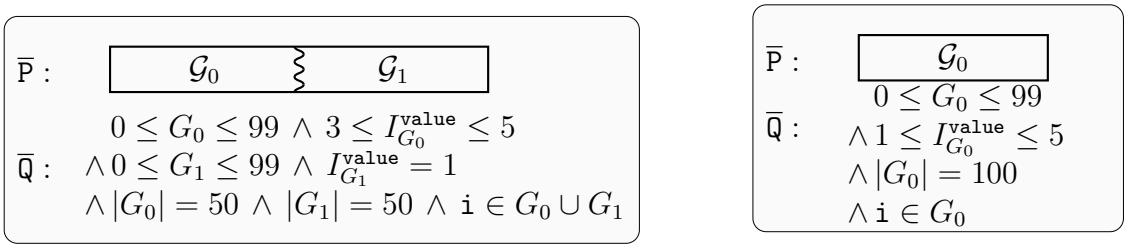
- It introduces a fresh group G_j to the memory predicate $\overline{\mathtt{P}}$ on array a.
- The size constraint $|G_j| = 0$ is added to $\overline{\mathtt{Q}}$.
- Additional constraints on the index and the fields of group G_j are added to $\overline{\mathtt{Q}}$.
- For each group G_i in $\overline{\mathtt{P}}$ and each $\mathtt{f} \in \mathbb{F}_{\mathtt{a}}$, the set relations $I^{\mathtt{f}}_{G_j} \subseteq G_i$ are added to $\overline{\mathtt{Q}}$.

Example 5.5 (The creation operator)**.** *Figure 5.8(a) defines an abstract state* $(\overline{\mathtt{P}}, \overline{\mathtt{Q}})$ *with a single array, fully initialized to 0, and represented by a single group. Similarly, Figure 5.8(b) shows a possible result for* $\overline{\mathbf{create}}$*.*

Theorem 5.2 (Soundness of the creation operator)**.** *Operator* $\overline{\mathbf{create}}$ *is sound in the sense that, for all abstract state* $\overline{\mathtt{a}}$*, for all array variable* a*,*

$$\gamma^a(\overline{\mathbf{create}}(\overline{\mathtt{a}}, \mathtt{a})) = \gamma^a(\overline{\mathtt{a}})$$

Proof. In the new group G_j created by operator $\overline{\mathbf{create}}$, the predicate $|G_j| = 0$ indicates that the addition of the new group does not affect the concretization. □



(a) $\overline{\mathtt{a}}$ (b) $\overline{\mathbf{merge}}(\overline{\mathtt{a}}, \{0, 1\})$

Figure 5.9 – Merging in abstract state $\overline{\mathtt{a}}$

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 $\overline{\mathbf{merge}}$ takes an abstract state $\overline{\mathtt{a}} = (\overline{\mathtt{P}}, \overline{\mathtt{Q}})$, an array $\mathtt{a}$ and a set of groups of array $\mathtt{a}$ 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 input set of groups 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 corresponding memory predicate to $\overline{\mathtt{P}}$;
- The numeric constraints on indexes and fields of G_i over-approximate those on G_k and G_j; group size $|G_i|$ is assigned with $|G_k| + |G_j|$ in $\overline{\mathtt{Q}}$;
- The set relation predicates on G_i over-approximate those on G_k and G_j in $\overline{\mathtt{Q}}$ (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 memory predicates $\mathcal{G}_j$ and $\mathcal{G}_k$ from $\overline{\mathtt{P}}$.

Example 5.6 (The merging operator)**.** *Figure 5.9(a) defines an abstract state $\overline{\mathtt{a}}$ which describes an array with two groups. Applying $\overline{\mathbf{merge}}$ to $\overline{\mathtt{a}}$ and set $\{0, 1\}$ produces the state shown in Figure 5.9(b), with a single group and coarser predicates, obtained by joining the constraints over the contents of the initial groups.*

Theorem 5.3 (Soundness of the merging operator)**.** *Operator $\overline{\mathbf{merge}}$ is sound in the following sense: For all abstract state $\overline{\mathtt{a}}$, array variable $\mathtt{a}$, and two groups G_k and G_j in array $\mathtt{a}$,*

$$\gamma^a(\overline{\mathtt{a}}) \subseteq \gamma^a(\overline{\mathbf{merge}}(\overline{\mathtt{a}}, \mathtt{a}, \{G_k, G_j\}))$$

Proof. Let $\overline{\mathtt{a}} = (\overline{\mathtt{P}}, \overline{\mathtt{Q}})$ be an abstract state, and two groups G_k and G_j in $\overline{\mathtt{P}}$. We assume that applying $\overline{\textbf{merge}}$ on G_k and G_j in $\overline{\mathtt{a}}$ produces group G_i (in the algorithm of $\overline{\textbf{merge}}$, G_i will be renamed to G_j or G_k finally, but the renaming does not affect the concretization) in abstract state $\overline{\mathtt{a}}' = (\overline{\mathtt{P}}', \overline{\mathtt{Q}}')$.

Let $\sigma \in \gamma^a(\overline{\mathtt{a}})$. We write ν for the witnesses of $\sigma \in \gamma^a(\overline{\mathtt{a}})$ in the definition of Figure **??**. We now show that σ is also in the concretization of $\overline{\mathtt{a}}'$, by constructing a witnesses ν':

1. fixing $\nu'(G_i)$ so that $\nu'(G_i) = \nu(G_j) \cup \nu(G_k)$;
2. adding new variables (fields, size, index) for group G_i, that inherit from the values of the set variables corresponding to G_j, G_k:
   (a) $\nu(|G_i|) = \nu(|G_j|) + |\nu(G_k|)$;
   (b) similarly to $G_.$, fields are set variables and defined by $\nu(I^{\mathtt{f}}_{G_i}) = \nu(I^{\mathtt{f}}_{G_j}) \cup \nu(I^{\mathtt{f}}_{G_k})$;
3. removing all set variables corresponding to G_j, G_k.

Then, since the set relation predicates on G_i over-approximate those on G_k and G_j in $\overline{\mathtt{Q}}$, to prove that ν' are witnesses of $\sigma \in \gamma^a(\overline{\mathtt{a}}')$, we simply need to check the implication relations in the concretization function.

This proves the soundness of $\overline{\textbf{merge}}$. □

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 numeric predicates can be viewed as a product of set relations and pure-numeric predicates and can benefit from *reduction* [CC79]. That is, if we consider numeric predicate $\overline{\mathtt{Q}} = \overline{\mathtt{g}} \wedge \overline{\mathtt{u}}$, components $\overline{\mathtt{g}}$ and $\overline{\mathtt{u}}$ may allow to refine each other. Such steps are performed by a *partial reduction* operator $\overline{\textbf{reduce}}$, which strengthens the set relations and pure-numeric predicates, without changing the global concretization [CC79]. The operations of $\overline{\textbf{reduce}}$ are based on the numeric implications of set relation predicates. It consists of two directions:

- from $\overline{\mathtt{g}}$ to $\overline{\mathtt{u}}$: set relations always imply pure-numeric constraints over the size and indexes of array groups, e.g., if $\mathtt{x} \in G_i$, then group G_i has at least one element ($|G_i| \geq 1$), and if $G_i < 5$, then $\mathtt{x} < 5$;
- from $\overline{\mathtt{u}}$ to $\overline{\mathtt{g}}$: more precise set relations can be inferred from the pure-numeric relations between variables and group indexes, e.g., if $\mathtt{x} < G_i$, then $\overline{\textbf{reduce}}$ removes G_i from $\mathtt{x} \in G_i \cup G_j$ in $\overline{\mathtt{g}}$.

Note that reduction could be overly costly to compute in general. To avoid that, reduction is done lazily: for instance, the analysis will attempt to generate relations between $\mathtt{x}$ and G_i only when $\mathtt{x}$ is used as an index to access the array G_i corresponds to.

Theorem 5.4 (Soundness of the reduction operator)**.** *Suppose* $\overline{\mathtt{a}}$ *is an abstract state, operator* $\overline{\textbf{reduce}}$ *does not change concretization.*

$$\gamma^a(\overline{\textbf{reduce}}(\overline{\mathtt{a}})) = \gamma^a(\overline{\mathtt{a}})$$

Proof. To establish the soundness of $\overline{\textbf{reduce}}$, 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 $(\overline{\mathtt{P}}, \overline{\mathtt{g}} \wedge \overline{\mathtt{u}})$ be an abstract state, such that $\mathtt{x} \in G_i$ appears in $\overline{\mathtt{g}}$. Then for any $(\sigma, \nu) \in \gamma^a(\overline{\mathtt{P}}, \overline{\mathtt{g}} \wedge \overline{\mathtt{u}})$, we have $\sigma(\mathtt{x}) \in \nu(G_i)$, which implies $|\nu(G_i)| \geq 1$. Thus it is sound to add constraint $|G_i| \geq 1$ to $\overline{\mathtt{u}}$. □

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: our analysis computes the similarities between groups and decides which groups to be merged, especially in join and widening.
- Assignments are based on strong updates: our analysis generates a group which contains only the cell being 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).

5.4 Transfer Functions

Our array static analysis performs a *forward abstract interpretation* [CC77]. In this section, we study the abstract transfer functions for tests (Section 5.4.1) and assignments (Section 5.4.2). Each transfer function should over-approximate the concrete effect of the corresponding program construction in the abstract domain.

5.4.1 Analysis of Conditions

The concrete semantics of a condition $\mathtt{r}$ is a function that inputs a set of states $\mathcal{S}$ and returns the subset of $\mathcal{S}$ in which $\mathtt{r}$ evaluates to 1. Therefore, the abstract interpretation of a test from abstract state $\overline{\mathtt{a}} = (\overline{\mathtt{P}}, \overline{\mathtt{Q}})$ should narrow the set of concrete states described by $\overline{\mathtt{a}}$ by filtering out states in which $\mathtt{r}$ does not evaluate to 1. Intuitively, it proceeds by strengthening constraints in the pure-numeric component $\overline{\mathtt{u}}$, and propagating them into $\overline{\mathtt{g}}$ thanks to $\overline{\textbf{reduce}}$.

However, the application of test $\mathtt{r}$ to pure-numeric constraints $\overline{\mathtt{u}}$ is not immediate, since the array cells that occur in $\mathtt{r}$ do not necessarily correspond directly to variables

in $\overline{\mathtt{u}}$. As an example, let us consider condition test $\mathtt{a}[\mathtt{i}].\mathtt{f} == 0$ in an abstract state where $\mathtt{a}$ is partitioned into two groups G_0, G_1 and where the only constraint available is $\mathtt{i} \in G_0 \cup G_1$: then, the group array cell $\mathtt{a}[\mathtt{i}]$ belongs to cannot be identified without ambiguity. Moreover, each group may contain several elements, and its $\mathtt{f}$ field may be described by a set variable. Therefore, our analysis cannot refine $\mathtt{a}[\mathtt{i}].\mathtt{f}$.

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 $\mathtt{a}[\mathtt{i}].\mathtt{f} == 0$ in a disjunction of *two* abstract states where the set relation is replaced by $\mathtt{i} \in G_0$ (resp., $\mathtt{i} \in G_1$). This process is called $\overline{\textbf{enumerate}}$. The analysis then applies the numerical domain condition test operator to each disjunct. In this case, it will apply $I^{\mathtt{f}}_{G_0} == 0$ to disjunct 0 and $I^{\mathtt{f}}_{G_1} == 0$ to disjunct 1. Note that $\overline{\mathtt{u}}$ may have set variables (whenever a group describes more than a single array cell, its fields are set variable), 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 $I^{\mathtt{f}}_{G_0} == 0$ *will not* strengthen the constraints in $\overline{\mathtt{u}}$. This suits the weak version of guard function $\overline{\textbf{guard}}^{\overline{\mathcal{U}}}[.]_w$ in the Maya+ functor. After the numerical condition test operator has been applied to all disjuncts, the analysis applies operator $\overline{\textbf{reduce}}$, and merges all resulting disjuncts.

Note that the abstract test operator does not change the memory predicates, 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 5.5):

Definition 5.7 (Local disjunction join)**.** *Utilizing the join operator* $\overline{\textbf{join}}^{\overline{\mathcal{U}}}$ *of Maya+ domain, we define the* local disjunction join *operator* $\overline{\textbf{join}}^{a}_{\equiv}$ *as*

$$\overline{\textbf{join}}^{a}_{\equiv}((\overline{\mathtt{P}}, \overline{\mathtt{Q}}_0), (\overline{\mathtt{P}}, \overline{\mathtt{Q}}_1)) = (\overline{\mathtt{P}}, \overline{\textbf{join}}^{\overline{\mathcal{U}}}(\overline{\mathtt{u}}_0, \overline{\mathtt{u}}_1) \wedge (\overline{\mathtt{g}}_0 \cap \overline{\mathtt{g}}_1))$$

Theorem 5.5 (Soundness of local disjunction join)**.** *The local disjunction join operator* $\overline{\textbf{join}}^{a}_{\equiv}$ *is sound.*

$$\forall i \in \{0, 1\},\ \gamma^a(\overline{\mathtt{P}}, \overline{\mathtt{Q}}_0) \subseteq \gamma^a(\overline{\textbf{join}}^{a}_{\equiv}((\overline{\mathtt{P}}, \overline{\mathtt{Q}}_0), (\overline{\mathtt{P}}, \overline{\mathtt{Q}}_1)))$$

Definition 5.8 (The transfer function for condition tests)**.** *The algorithm of the abstract transfer function* $\overline{\textbf{guard}}^{a}[.]$ *for condition tests is fully described in Figure 5.10. Operator* $\overline{\textbf{enumerate}} : \overline{\mathcal{H}} \to \mathcal{P}(\overline{\mathcal{H}})$ *generates the set of every possible state in which each array cell in* $\mathtt{r}$ *belongs to exactly one group. Namely,* $\gamma^a(\overline{\mathtt{a}}) = \cup\{\gamma^a(\overline{\mathtt{a}}_i) \mid \overline{\mathtt{a}}_i \in \overline{\textbf{enumerate}}(\overline{\mathtt{a}})\}$*, and in any* $\overline{\mathtt{a}}_i$*, and for any array cell* $\mathtt{a}[\mathtt{x}]$ *in* $\mathtt{r}$*, the group that* $\mathtt{a}[\mathtt{x}]$ *belongs to is deterministic. Then, condition test* $\overline{\textbf{guard}}^{\overline{\mathcal{U}}}[.]_w$ *of the Maya+ domain and the reduction operator are applied in each disjunctive state. All states are eventually joined together by the local disjunction join operator* $\overline{\textbf{join}}^{a}_{\equiv}$*.*

$\overline{\textbf{guard}}^a[\mathsf{r}](\overline{\mathsf{P}}, \overline{\mathsf{g}} \wedge \overline{\mathsf{u}}))\{$
  $\mathsf{r}_{0\sim k}, \overline{\mathsf{g}}_{0\sim k} = \overline{\textbf{enumerate}}(\mathsf{r}, \overline{\mathsf{g}})$;
  foreach$(i \in \{0, 1, \ldots, k\})\{$
    $\overline{\mathsf{u}}_i = \overline{\textbf{guard}}^{\overline{\mathcal{U}}}[\mathsf{r}_i]_w(\overline{\mathsf{u}})$;
    $\overline{\mathsf{P}}, \overline{\mathsf{u}}_i, \overline{\mathsf{g}}_i = \overline{\textbf{reduce}}(\overline{\mathsf{P}}, \overline{\mathsf{u}}_i, \overline{\mathsf{g}}_i)$;
  $\}$
  $(\overline{\mathsf{P}}, \overline{\mathsf{u}}', \overline{\mathsf{g}}') = \overline{\textbf{join}}^a_{\equiv}((\overline{\mathsf{P}}, \overline{\mathsf{u}}_0, \overline{\mathsf{g}}_0), (\overline{\mathsf{P}}, \overline{\mathsf{u}}_1, \overline{\mathsf{g}}_1), \ldots, (\overline{\mathsf{P}}, \overline{\mathsf{u}}_k, \overline{\mathsf{g}}_k))$;
  return $(\overline{\mathsf{P}}, \overline{\mathsf{u}}' \wedge \overline{\mathsf{g}}')$;
$\}$

Figure 5.10 – The algorithm of the condition test transfer function

Theorem 5.6 (Soundness of the transfer function for condition tests)**.** *The abstract transfer function* $\overline{\textbf{guard}}^a[.]$ *is* sound *in the sense that:*

$$\textbf{guard}[\![\mathsf{r}]\!]\gamma^a(\overline{\mathsf{a}}) \subseteq \gamma^a(\overline{\textbf{guard}}^a[\mathsf{r}](\overline{\mathsf{a}})))$$

Proof. The soundness of the condition test operator $\overline{\textbf{guard}}^a[.]$ follows from the fact that operators $\overline{\textbf{enumerate}}$ and $\overline{\textbf{reduce}}$ do not change the concretization, and from the soundness of the condition test operator $\overline{\textbf{guard}}^{\overline{\mathcal{U}}}[.]_w$ of the numeric domain, and of the local disjunction join operator $\overline{\textbf{join}}^a_{\equiv}$. □

The precision loss in condition test mainly comes from that in the condition test of Maya+ domain.

Example 5.7 (The transfer function for condition tests)**.** *Now, let us look at the function* `cleanup` *in Figure 5.3. At the beginning of the first fixpoint iteration over the loop at line 6, the abstract state is shwon in Figure 5.11. From numeric constraints over* `i` *and group indexes,* `mproc[i]` *may be in* G_0 *or* G_1*. Then, the analysis of test* `mproc[i].mp_flag` > 0 *at line 7 will locally create two disjuncts corresponding to each of these groups. However, in the case of* G_1*,* $I^{\texttt{mp\_flag}}_{G_1} = 0$*, thus the numeric test* $I^{\texttt{mp\_flag}}_{G_1} > 0$ *will produce abstract value* $\bot$ *denoting the empty set of states. Therefore, only the second disjunct (the case corresponding to* G_0*) contributes to the abstract post-condition. Thus, the analysis derives* $\texttt{i} \in G_0$*.*

5.4.2 Assignment

Given an l-value l and an expression r, the concrete semantics of the assignment $\mathsf{l} = \mathsf{r}$ writes the value of r into the cell that l evaluates to. On the abstract level, given abstract pre-condition $\overline{\mathsf{a}} = (\overline{\mathsf{P}}, \overline{\mathsf{Q}})$, an abstract post-condition for $\mathsf{l} = \mathsf{r}$ can be computed in two steps:

1. materialization of the memory cell that gets updated,

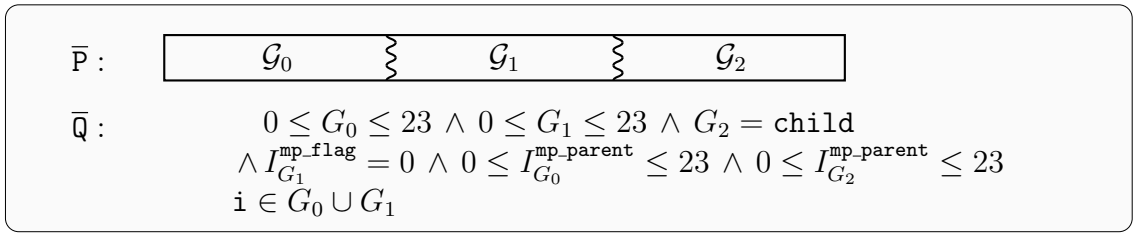


Figure 5.11 – The abstract state before the condition test at line 7

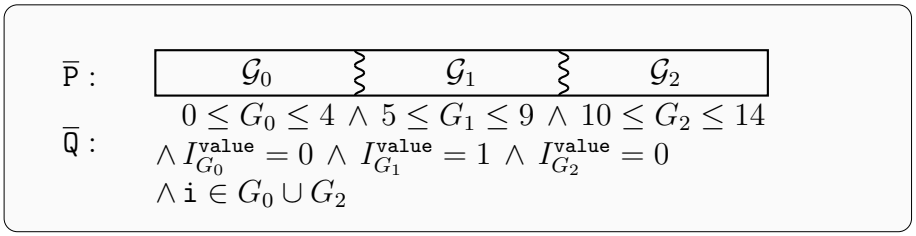
2. update of the numeric constraints on the materialized cell fields in $\overline{\mathcal{U}}$ using $\overline{\textbf{assign}}^{\overline{\mathcal{U}}}[.]$, and update of the set relations, and application of the reduction operator to the resulting abstract state.

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

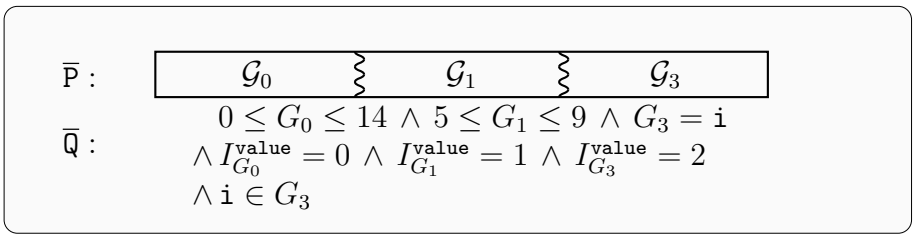
- **Step 1: Materialization.** When the l-value $\mathtt{l}$ denotes an array cell, the analysis first *materializes* it into a group consisting of a single cell, so that strong updates can be carried out on $\overline{\mathtt{Q}}$. To achieve this, the analysis computes which group(s) $\mathtt{l}$ may evaluate into in abstract state $\overline{\mathtt{a}}$. If there is a single such group G_i, that contains a single cell (i.e., $|G_i| = 1$), then materialization is already achieved. If there is a single such group G_i, and $|G_i|$ is greater than 1, then the analysis calls $\overline{\textbf{split}}$ 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 $\mathtt{l}$ is $\mathtt{a}[\mathtt{i}]$ and $\mathtt{i} \in G_0 \cup G_1$), the analysis first calls $\overline{\textbf{merge}}$ to merge all such groups and then falls back to the case where $\mathtt{l}$ can only evaluate into a single group. This process is formalized as operator $\overline{\textbf{materialize}} : \{\mathtt{l}\} \times \overline{\mathcal{H}} \to \overline{\mathcal{H}}$.

  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. Another 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 5.7) confirm this intuition.

- **Step 2: Constraints update.** New relation predicates can be inferred by operator $\overline{\textbf{propagate}} : \{\mathtt{l} = \mathtt{r}\} \times \{\overline{\mathtt{g}}\} \to \{\overline{\mathtt{g}}\}$. It propagates relation predicates in two ways: (1) if both sides of the assignment are standard variables, e.g., $\mathtt{v} = \mathtt{u}$, and we have $\mathtt{u} \in G_i$, then after assignment, we get $\mathtt{v} \in G_i$; (2) if the right hand side is an array cell as in $\mathtt{parent} = \mathtt{mproc}[\mathtt{child}].\mathtt{mp\_parent}$ in the example of Figure 5.3, if $\mathtt{child}$ stores an index in group G_0 ($\mathtt{child} \in G_0$), the operator first looks for relations between fields and indexes such as $I_{G_0}^{\mathtt{mp\_parent}} \subseteq G_0$, and propagates them to the l-value as $\mathtt{parent} \in G_0$.



(a) Pre-condition



(b) Post-condition

Figure 5.12 – The pre-and post-condition of assignment $\mathtt{a}[\mathtt{i}] = 2$

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 5.4.1). The reduction operator is applied after both numeric constraints and set relations are updated.

Definition 5.9 (The transfer function for assignments)**.** *The algorithm for the analysis of assignment* $\overline{\mathbf{assign}}^a[.]$ *is formalized in Figure 5.13. Operator* $\overline{\mathbf{materialize}}$ *is used to materialize the array cell to be updated . Operator* $\overline{\mathbf{enumerate}}$ *generates every possible state in which each array cell in* l *belongs to exactly one group as in Section 5.4.1. Then, the assignment* $\overline{\mathbf{assign}}^{\mathcal{U}}[.]$ *of the Maya+ domain, operator* $\overline{\mathbf{propagate}}$ *and the reduction operator are applied on each disjunctive state. Finally, all states are eventually joined together by the local disjunction join operator* $\overline{\mathbf{join}}^a_{\equiv}$*.*

Example 5.8 (The transfer function for assignments)**.** *We consider* $\mathtt{a}[\mathtt{i}] = 2$ *and abstract pre-condition shown in Figure 5.12(a). The l-value evaluates into an index in* G_0 *or* G_2*, The result of materialization is shown in Figure 5.12(b), we can see that groups* G_0 *and* G_2 *are merged into group* G_0 *and* $\mathtt{a}[\mathtt{i}]$ *is split as the sole element of group* G_3*. Then, the assignment boils down to a strong update on set variable* $I_{G_3}^{\mathtt{value}}$*.*

Theorem 5.7 (Soundness of the transfer function for assignments)**.** *Abstract transfer function* $\overline{\mathbf{assign}}^a[.]$ *is* sound *in the sense that:*

$$\mathbf{stat}[\![\mathsf{l} = \mathsf{r}]\!](\gamma^a(\overline{\mathsf{a}})) \subseteq \gamma^a(\overline{\mathbf{assign}}^a[\mathsf{l} = \mathsf{r}](\overline{\mathsf{a}}))$$

$$
\begin{array}{l}
\overline{\textbf{assign}}^{a}[\mathsf{l} = \mathsf{r}](\overline{\mathsf{P}}, \overline{\mathsf{u}} \wedge \overline{\mathsf{g}})\{ \\
\quad (\overline{\mathsf{P}}, \overline{\mathsf{u}} \wedge \overline{\mathsf{g}}) = \overline{\textbf{materialize}}(\mathsf{l}, (\overline{\mathsf{P}}, \overline{\mathsf{u}} \wedge \overline{\mathsf{g}})); \\
\quad \mathsf{r}_{0\sim k}, \overline{\mathsf{g}}_{0\sim k} = \overline{\textbf{enumerate}}(\mathsf{r}, \overline{\mathsf{g}}); \\
\quad \textbf{foreach}(i \in \{0, 1, \ldots, k\})\{ \\
\quad\quad \overline{\mathsf{u}}_i = \overline{\textbf{assign}}^{\overline{\mathcal{U}}}[\mathsf{l} = \mathsf{r}_i](\overline{\mathsf{u}}); \\
\quad\quad \overline{\mathsf{g}}_i = \overline{\textbf{propagate}}(\mathsf{l} = \mathsf{r}_i, \overline{\mathsf{g}}_i); \\
\quad\quad \overline{\mathsf{P}}, \overline{\mathsf{u}}_i, \overline{\mathsf{g}}_i = \overline{\textbf{reduce}}(\overline{\mathsf{P}}, \overline{\mathsf{u}}_i \wedge \overline{\mathsf{g}}_i); \\
\quad \} \\
\quad (\overline{\mathsf{P}}, \overline{\mathsf{u}} \wedge \overline{\mathsf{g}}) = \overline{\textbf{join}}^{a}_{\equiv}((\overline{\mathsf{P}}, \overline{\mathsf{u}}_0 \wedge \overline{\mathsf{g}}_0), (\overline{\mathsf{P}}, \overline{\mathsf{u}}_1 \wedge \overline{\mathsf{g}}_1), \ldots, (\overline{\mathsf{P}}, \overline{\mathsf{u}}_k \wedge \overline{\mathsf{g}}_k)); \\
\quad \textbf{return } (\overline{\mathsf{P}}, \overline{\mathsf{u}} \wedge \overline{\mathsf{g}}); \\
\}
\end{array}
$$

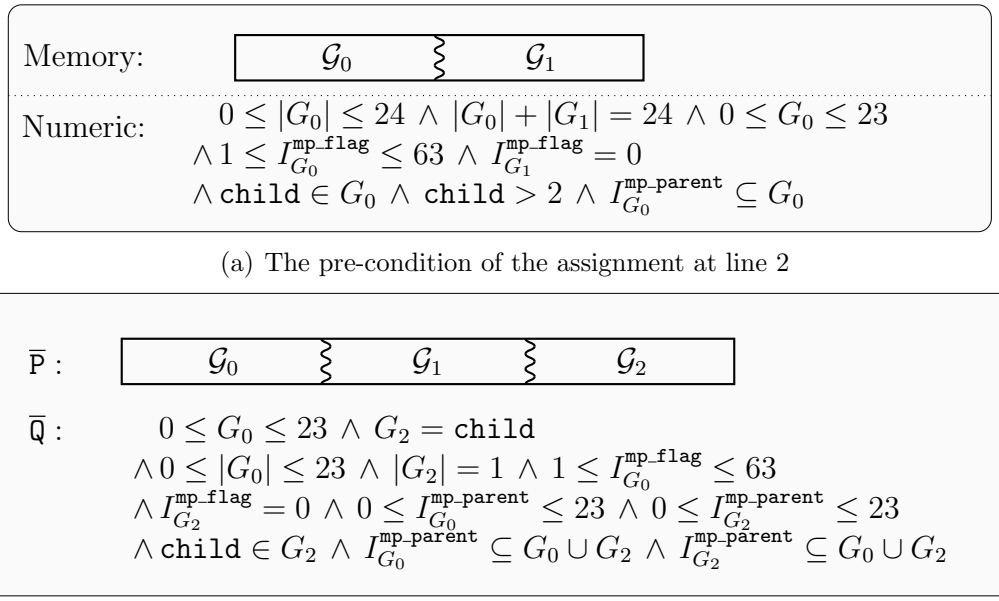
Figure 5.13 – The algorithm of the assignment transfer function

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

Remark 5.1. *Our analysis performs strong updates in assignments, which capture 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.*

Our analysis does not materialize the array elements that participate in condition tests. 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 `a[i]` *after it has been materialized in condition test, there would be no precision gain.*

Example 5.9 (The transfer function for assignments)**.** *As remarked in Section 5.1.2, function* `cleanup` *should be called only in states that satisfy* $\mathcal{R}^{minix}$*, and where predicate* $\texttt{child} \in G_0 \wedge \texttt{child} > 2$ *holds (which means* `mproc[child]` *may be any element of group 0 the index of which is greater than 2). This is shown in Figure 5.14(a). After the assignment in line 2, the transfer function infers that* `parent` *is also an index in group* G_0*, since* $\mathcal{R}^{minix}$ *entails that* $I^{\texttt{mp\_parent}}_{G_0} \subseteq G_0$ *(the parent of any valid process is also a valid*



(a) The pre-condition of the assignment at line 2

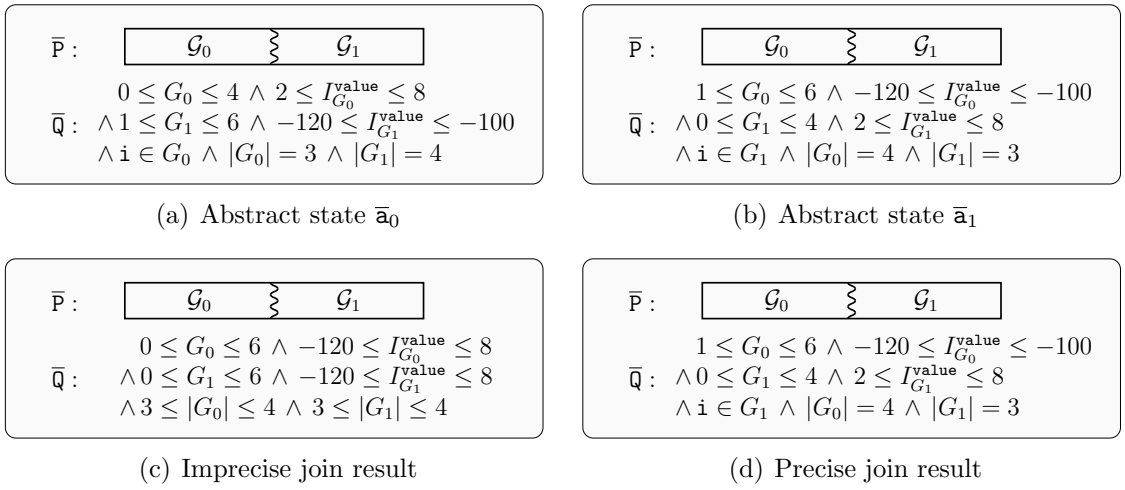
(b) The post-condition of the assignment at line 4

Figure 5.14 – Analysis on two assignments

process). The resulting abstract state is with set relation $\texttt{parent} \in G_0$*. At line 4, array cell* $\texttt{mproc}[\texttt{child}]$ *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* $\texttt{child} \in G_0$*, thus* $|G_0| \geq 1$*). Therefore, and in order to perform a* strong update*, our analysis first* materializes *the array element that is being modified, by splitting group* G_0 *into two groups, labeled* G_0 *and* G_2*, where group* G_2 *has* exactly one element*, corresponding to* $\texttt{mproc}[\texttt{child}]$ *(which is also expressed by* $\texttt{child} \in G_2$*). Both groups inherit predicates from former group* G_0*. Since group* G_2 *has a single element (*$|G_2| = 1$*) which corresponds exactly to the modified cell, the analysis can perform a strong update at this stage, and it generates the abstract state at Figure 5.14(b).*

5.5 Join, Widening and Inclusion Check

Our analysis proceeds by standard abstract interpretation, and uses widening and inclusion check to compute abstract post-fixpoints for loops and abstract join for control flow union (e.g., after an **if** statement or unrolled iterations in loops). All these operators face the same difficulties: they may be applied to a pair of abstract states that do not have compatible memory predicates (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.



(a) Abstract state $\overline{\mathtt{a}}_0$

(b) Abstract state $\overline{\mathtt{a}}_1$

(c) Imprecise join result

(d) Precise join result

Figure 5.15 – Impact of the group matching on the abstract join

5.5.1 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 $\overline{\mathbf{join}}^a_{\equiv}$ shown in Definition 5.7 simply over-approximates the properties for each group in abstract states with the same memory predicates. It 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 5.4.1), this operator would not be adequate even when both inputs have the same number of groups.

Example 5.10 (The partition compatibility problem)**.** *Let us assume two abstract states* $\overline{\mathtt{a}}_0, \overline{\mathtt{a}}_1$ *with the same number of groups for each array, that we assume to have the same names. Then, the operator* $\overline{\mathbf{join}}^a_{\equiv}$ *can be applied to these states, and computes an over-approximation for* $\overline{\mathtt{a}}_0, \overline{\mathtt{a}}_1$*, by joining predicates for each group name, the pure-numeric predicates and the set relations. However, this simple operator may produce very imprecise results if applied directly. As an example, we show two abstract states* $\overline{\mathtt{a}}_0$ *and* $\overline{\mathtt{a}}_1$ *in Figure 5.15(a) and Figure 5.15(b), that are similar up to a group name permutation. The direct join is shown in Figure 5.15(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 5.15(d). This example shows that we need to match groups with similar properties from both inputs before we can apply* $\overline{\mathbf{join}}^a_{\equiv}$*.*

Obviously, this *group matching problem* is actually even more complicated in general as $\overline{\mathtt{a}}_0, \overline{\mathtt{a}}_1$ usually do not have the same number of groups. To address this, we need to

define a join operator that modifies partitions and match groups with similar properties in both inputs to avoid precision loss. In this thesis, we use constraints over partition group fields into account to decide what partition modification is most adequate. The algorithm we choose is based on heuristics, yet a non optimal algorithm will impact only precision, but not soundness.

Ranking function. The *group field properties* are achieved with the help of a ranking function $\overline{\mathbf{rank}} : \mathbb{G}\times\mathbb{G} \to \mathbb{N}$, 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 $\overline{\mathbf{rank}}(G_i, G_j)$ indicates G_i of $\overline{\mathsf{a}}_0$ and G_j of $\overline{\mathsf{a}}_1$ are likely to describe sets of cells with similar properties.

The value of $\overline{\mathbf{rank}}(G_i, G_j)$ is positively correlated with three factors:

- the number of common constraints on the set variables associated to fields and indexes in $\overline{\mathsf{u}}$ (including their ranges and, when a relational abstract domain is used, relations with program variables);
- the number of variables that have var-index relations with both groups;
- the "group origin", determined by group names in the representation of the abstract values (the name of a group keeps unchanged if it is not split or merged, thus two groups with the same name may be from a single group in a predecessor abstract state).

Re-partitioning. Using the set of $\overline{\mathbf{rank}}(G_i, G_j)$ values, the analysis computes a *pairing* $\leftrightarrow \in \mathcal{P}(\mathbb{G} \times \mathbb{G})$, that is a set of relations between groups of $\overline{\mathsf{a}}_0$ and groups of $\overline{\mathsf{a}}_1$.

The pairing is defined by the rules below:

1. the analysis sorts all pairs of groups decreasingly according to their ranking values, and then select the first k pairs (k is parametric, usually the analysis lets k be the maximal number of groups in $\overline{\mathsf{a}}_0$ and $\overline{\mathsf{a}}_1$). That is, if the value of $\overline{\mathbf{rank}}(G_i, G_j)$ 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 (since all relations are added sequentially, this also prevents these relations to forming a circle).

After the two steps above, our analysis 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 in the right argument is created with $\overline{\mathbf{create}}$;

$\overline{\textbf{join}}^a(\overline{\mathtt{a}}_0, \overline{\mathtt{a}}_1)\{$
  foreach$(G_i$ **in** $\overline{\mathtt{a}}_0)$
    foreach$(G_j$ **in** $\overline{\mathtt{a}}_1)$
      $W_{ij} = \overline{\textbf{rank}}(G_i, G_j)$;
  $\overline{\mathtt{a}}_0, \overline{\mathtt{a}}_1 = \overline{\textbf{repartition}}(W, \overline{\mathtt{a}}_0, \overline{\mathtt{a}}_1)$;
  return $\overline{\textbf{join}}^a_{\equiv}(\overline{\mathtt{a}}_0, \overline{\mathtt{a}}_1)$;
$\}$

Figure 5.16 – The algorithm of the join operator

- if $G_i \leftrightarrow G_j$ and $G_i \leftrightarrow G_k$, then G_j and G_k are merged by operator $\overline{\textbf{merge}}$ and the resulting group is paired with G_i (when more than two groups are to be merged, merge is assoicative);
- 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 $\overline{\textbf{repartition}}$. 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 $\overline{\textbf{join}}^a_{\equiv}$ can be applied. This defines the abstract join operator $\overline{\textbf{join}}^a$.

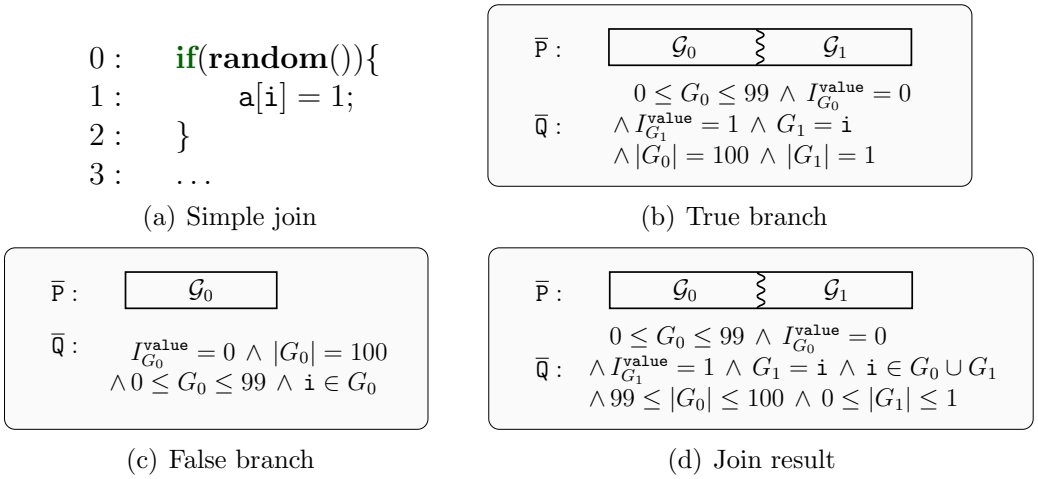
Definition 5.10 (Join algorithm)**.** *The algorithm of* $\overline{\textbf{join}}^a$ *is shown in Figure 5.16. It first computes the ranking values of all groups from two abstract states by operator* $\overline{\textbf{rank}}$*, and then repartitions the two states by* $\overline{\textbf{repartition}}$ *(with* $\overline{\textbf{create}}$ *and* $\overline{\textbf{merge}}$*) according to the ranking values. Finally, it applies* $\overline{\textbf{join}}^a_{\equiv}$ *on two compatible states.*

Theorem 5.8 (Soundness of the join algorithm)**.** *Join operator* $\overline{\textbf{join}}^a$ *is* sound *in the sense that:*

$$\forall \overline{\mathtt{a}}_0, \overline{\mathtt{a}}_1,\ \gamma^a(\overline{\mathtt{a}}_0) \subseteq \gamma^a(\overline{\textbf{join}}^a(\overline{\mathtt{a}}_0, \overline{\mathtt{a}}_1)) \wedge \gamma^a(\overline{\mathtt{a}}_1) \subseteq \gamma^a(\overline{\textbf{join}}^a(\overline{\mathtt{a}}_0, \overline{\mathtt{a}}_1))$$

Proof. The soundness of the join operator $\overline{\textbf{join}}^a$ follows from the fact that the operator $\overline{\textbf{repartition}}$ is sound (since $\overline{\textbf{create}}$ and $\overline{\textbf{merge}}$ are sound), and from the soundness of $\overline{\textbf{join}}^a_{\equiv}$. □

Example 5.11 (Join algorithm)**.** *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 5.17(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 5.17(c) (that has a single group) and the state in true*



(a) Simple join (b) True branch

(c) False branch (d) Join result

Figure 5.17 – Join of a one group state with a two groups state

branch shown in Figure 5.12(b) (that has two groups). We note that G_0 in Figure 5.17(c) has similar properties as G_0 in Figure 5.17(b), thus they get paired. Moreover, G_1 in Figure 5.17(b) is paired to no group, so a new group is created, and paired to it. At that stage $\overline{\mathbf{join}}^a_{\equiv}$ applies, and returns the abstract state shown in Figure 5.17(d). In this abstract state, group G_1 with known content is possibly empty abstracts the fact that the assignment at line 1 is possibly executed.

5.5.2 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 abstract states with *compatible* partitions (that is, partitions that have the same numbers of groups, with the same names):

Definition 5.11 (Widening for abstract states with compatible partitions)**.** *The* widening for abstract states with compatible partitions *is defined by*

$$\overline{\mathbf{widen}}^a_{\equiv}((\overline{\mathtt{P}}, \overline{\mathtt{u}}_0 \wedge \overline{\mathtt{g}}_0), (\overline{\mathtt{P}}, \overline{\mathtt{u}}_1 \wedge \overline{\mathtt{g}}_1)) = (\overline{\mathtt{P}}, \overline{\mathbf{widen}}^{\overline{\mathcal{U}}}(\overline{\mathtt{u}}_0, \overline{\mathtt{u}}_1) \wedge \overline{\mathtt{g}}_0 \cap \overline{\mathtt{g}}_1)$$

This operator clearly defines a widening operator. Indeed the widening operator $\overline{\mathbf{widen}}^{\overline{\mathcal{U}}}$ of the Maya+ domain ensures convergence, when the number of variables in $\overline{\mathtt{u}}_0, \overline{\mathtt{u}}_1$ is bounded, and in the case of $\overline{\mathbf{widen}}^a_{\equiv}$, it is constant (the set of groups is fixed here). Similarly, the resulting set of set relation predicates is included in the set of $\overline{\mathtt{g}}_0, \overline{\mathtt{g}}_1$

$$
\begin{array}{l}
\overline{\textbf{widen}}^{a}(\overline{\mathsf{a}}_0, \overline{\mathsf{a}}_1)\{ \\
\quad \textbf{foreach}(G_i \textbf{ in } \overline{\mathsf{a}}_0) \\
\quad\quad \textbf{foreach}(G_j \textbf{ in } \overline{\mathsf{a}}_1) \\
\quad\quad\quad W_{ij} = \overline{\textbf{rank}}(G_i, G_j); \\
\quad \overline{\mathsf{a}}_0, \overline{\mathsf{a}}_1 = \overline{\textbf{repartition}}_{\triangledown}(W, \overline{\mathsf{a}}_0, \overline{\mathsf{a}}_1); \\
\quad \textbf{return } \overline{\textbf{widen}}^{a}_{\equiv}(\overline{\mathsf{a}}_0, \overline{\mathsf{a}}_1); \\
\}
\end{array}
$$

Figure 5.18 – The algorithm of the widening operator

of the arguments, which are finite sets, thus this component is well-founded and will also eventually converge. However, the termination property of $\overline{\textbf{widen}}^{a}_{\equiv}$ relies on the assumption that memory predicates never change. As this assumption 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 memory predicates.

Re-partitioning for widening. To achieve termination, $\overline{\textbf{widen}}^{a}$ needs to ensure that for any sequence of abstract iterates, the memory predicates eventually converges: when memory predicates have converged, the set of groups is stable and $\overline{\textbf{widen}}^{a}_{\equiv}$ can be applied, and will ensure both soundness and termination. This convergence property is *not guaranteed* by the group matching algorithm of Section 5.5.1. Therefore the widening operator $\overline{\textbf{widen}}^{a}$ relies on a slightly different group re-partitioning operator $\overline{\textbf{repartition}}_{\triangledown}$.

1. operator $\overline{\textbf{repartition}}_{\triangledown}$ pairs each group with the group with which it has the highest ranking value, thus each group is with at least one pair relation (this is different with the re-partitioning in join);

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 $\overline{\textbf{create}}$ is needed in the partition transforming phase. Actually, only operator $\overline{\textbf{merge}}$ is used. Therefore this group matching operation ensures termination.

Definition 5.12 (Widening algorithm)**.** *The algorithm of* $\overline{\textbf{widen}}^{a}$ *is shown in Figure 5.18. It just replaces* $\overline{\textbf{repartition}}$ *and* $\overline{\textbf{join}}^{a}_{\equiv}$ *in the algorithm of join with* $\overline{\textbf{repartition}}_{\triangledown}$ *and* $\overline{\textbf{widen}}^{a}_{\equiv}$ *respectively.*

The resulting $\overline{\textbf{widen}}^{a}$ operator is a sound and terminating widening operator [CC77]. For better precision, the analysis always uses $\overline{\textbf{join}}^{a}$ for the first abstract iteration for a loop, and uses widening afterwards.

$$\overline{\mathtt{P}}: \boxed{\mathcal{G}_0}$$
$$\overline{\mathtt{Q}}: \begin{array}{l} 0 \leq G_0 \leq 99 \\ \wedge\ 0 \leq I_{G_0}^{\mathtt{value}} \leq 1 \\ \wedge\ |G_0| = 100 \\ \wedge\ \mathtt{i} \in G_0 \end{array}$$

Figure 5.19 – Widening result of two abstracts with different partitions

Theorem 5.9 (Soundness and termination of the widening algorithm)**.** *The operator* $\overline{\mathbf{widen}}^a$ *is a widening operator: it over-approximates its arguments and ensures the termination of abstract iterates.*

Proof. As in the case of $\overline{\mathbf{join}}^a$, the fact that $\overline{\mathbf{widen}}^a$ 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 $\overline{\mathtt{a}}_{n+1} = \overline{\mathbf{widen}}^a(\overline{\mathtt{a}}_n, \overline{\mathtt{a}}_n{}')$.

Since $\overline{\mathbf{widen}}^a$ never calls $\overline{\mathbf{create}}$ and $\overline{\mathbf{split}}$, and changes the number of groups only by calling $\overline{\mathbf{merge}}$, 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 component will also stabilize after finitely many iterates. Therefore, since $\overline{\mathbf{widen}}^a$ applies $\overline{\mathbf{widen}}^{\overline{\mathcal{U}}}$ on the pure-numeric predicates component, it ensures the termination of any sequence of abstract iterates.

Therefore, $\overline{\mathbf{widen}}^a$ is a widening operator. □

Example 5.12 (Widening algorithm)**.** *We consider the abstract states depicted in Figure 5.17(b) and in Figure 5.17(c) and show how* $\overline{\mathbf{widen}}^a$ *applies to these abstract states. The group matching algorithm will merge the two groups in Figure 5.17(b) and pair the resulting group to the only group in Figure 5.17(c). The output state after applying* $\overline{\mathbf{widen}}^a_{\equiv}$ *is shown in Figure 5.19.*

5.5.3 Inclusion Checking

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 $\overline{\mathbf{isle}}^a$: when $\overline{\mathbf{isle}}^a(\overline{\mathtt{a}}_0, \overline{\mathtt{a}}_1)$ returns **true**, then $\gamma^a(\overline{\mathtt{a}}_0) \subseteq \gamma^a(\overline{\mathtt{a}}_1)$.

As in the case of join, a restricted inclusion checking operator $\overline{\mathbf{isle}}^a_{\equiv}$ can be defined in a straightforward manner, that checks inclusion on "compatible" abstract states, that is abstract states with matching partitions: if $\overline{\mathbf{isle}}^{\overline{\mathcal{U}}}(\overline{\mathtt{u}}_0, \overline{\mathtt{u}}_1) = \mathbf{true}$ and $\overline{\mathtt{g}}_1$ is included in $\overline{\mathtt{g}}_0$ (as a set of constraints), then $\gamma^a(\overline{\mathtt{P}}, \overline{\mathtt{u}}_0 \wedge \overline{\mathtt{g}}_0) \subseteq \gamma^a(\overline{\mathtt{P}}, \overline{\mathtt{u}}_1 \wedge \overline{\mathtt{g}}_1)$, hence we let $\overline{\mathbf{isle}}^a_{\equiv}$ return **true** in that case.

$\overline{\textbf{isle}}^a(\bar{\mathsf{a}}_0, \bar{\mathsf{a}}_1)\{$
  foreach$(G_i$ **in** $\bar{\mathsf{a}}_0)$
    foreach$(G_j$ **in** $\bar{\mathsf{a}}_1)$
      $W_{ij} = \overline{\textbf{rank}}_<(G_i, G_j)$;
  if$(\exists G_i$ **in** $\bar{\mathsf{a}}_0, \forall G_j$ **in** $\bar{\mathsf{a}}_1, W_{ij} \leq 0)$
    return false;
  $\bar{\mathsf{a}}_0 = \overline{\textbf{repartition}}_<(W, \bar{\mathsf{a}}_0)$;
  return $\overline{\textbf{isle}}^a_\equiv(\bar{\mathsf{a}}_0, \bar{\mathsf{a}}_1)$;
$\}$

Figure 5.20 – The algorithm of the inclusion check operator

The inclusion checking algorithm is quite similar to that of join, but uses a modified ranking operator $\overline{\textbf{rank}}_<$ and a modified re-partition operator $\overline{\textbf{repartition}}_<$. Operator $\overline{\textbf{rank}}_<$ is the same as $\overline{\textbf{rank}}$ except that it evaluates $\overline{\textbf{rank}}_<(G_0, G_0')$ 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 $\overline{\textbf{repartition}}_<$ with $\overline{\textbf{repartition}}$ lies in two aspects: firstly, in the pairing phase $\overline{\textbf{repartition}}_<$ 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 $\overline{\textbf{split}}$ to G_i and pair the two resulting groups with G_j and G_k respectively.

Definition 5.13 (Includsion checking)**.** *The inclusion checking algorithm is shown in Figure 5.20. It first computes the ranking values of all groups from two abstract states using operator* $\overline{\textbf{rank}}_<$*. If there is a group from the left argument that ranks negatively with all groups from the right argument,* $\overline{\textbf{isle}}^a$ *conservatively returns false. Otherwise, it re-partitions the two states by* $\overline{\textbf{repartition}}_<$ *(with* $\overline{\textbf{create}}$*,* $\overline{\textbf{split}}$ *and* $\overline{\textbf{merge}}$*) according to the ranking values. Finally, it applies* $\overline{\textbf{isle}}^a_\equiv$ *to the two resulting compatible states.*

Theorem 5.10 (Soundness of inclusion checking)**.** *The inclusion check operator* $\overline{\textbf{isle}}^a$ *is* sound *in the sense that*

$$\overline{\textbf{isle}}^a(\bar{\mathsf{a}}_0, \bar{\mathsf{a}}_1) = \textit{true} \Longrightarrow \gamma^a(\bar{\mathsf{a}}_0) \subseteq \gamma^a(\bar{\mathsf{a}}_1)$$

Proof. First, we note that $\overline{\textbf{repartition}}_<$ does not modify the right hand side argument, and performs an *over*-approximation of the left hand side argument (through $\overline{\textbf{create}}$, $\overline{\textbf{split}}$ and $\overline{\textbf{merge}}$). Second, $\overline{\textbf{isle}}^a_\equiv$ is sound. Therefore, when $\overline{\textbf{isle}}^a$ 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 the pairing algorithm and the precision loss in operators on partitions.

5.6 Static Analysis on Programs Involving Arrays

In this section, we formalize an abstract interpreter for the language of Figure 5.1, and we discuss in detail the full analysis of the `cleanup` example.

5.6.1 Abstract Semantics

Based on the abstract operators introduced in the previous sections, we can build the abstract semantics of a program $\overline{\mathbf{stat}}[s] : \overline{\mathcal{H}} \to \overline{\mathcal{H}}$, which is a function that maps an abstract pre-condition into an abstract post-condition. The build process has been formalized in Chapter 3.

Theorem 5.11 (Soundness of abstract semantics)**.** *Given a program* $\mathbf{p}$ *and an abstract pre-condition* $\overline{\mathtt{a}}$*, the post-condition derived by the analysis is sound:*

$$[\![\mathbf{p}]\!](\gamma^a(\overline{\mathtt{a}})) \subseteq \gamma^a(\overline{\mathbf{stat}}[\mathbf{p}](\overline{\mathtt{a}}))$$

5.6.2 Example "`cleanup`" Revisited

We have shown some parts of the analysis on the function `cleanup` in Figure 5.3. In this section we provide more details about this analysis in Figure 5.21.

The function `cleanup` should always be called in a state where the Minix Memory Management Process Table satisfies global correctness property $\mathcal{R}^{minix}$ described in Figure 5.5(b), and where argument `child` is the identifier of a valid user process descriptor. Therefore, the analysis starts with pre-condition $\mathcal{R}^{minix} \wedge \mathtt{child} \in G_0 \wedge \mathtt{child} > 2$. It then proves that, under this pre-condition, and after executing the body of `cleanup`, $\mathcal{R}^{minix}$ always holds, which is checked with $\overline{\mathbf{isle}}^a$.

At start-up, we get pre-condition $\mathcal{R}^{minix} \wedge \mathtt{child} \in G_0 \wedge \mathtt{child} > 2$ (before line 1 in the figure). Because of the property of field `mp_parent` in group G_0 according to $\mathcal{R}^{minix}$, we obtain $\mathtt{parent} \in G_0$ at line 2. At line 4, since `child` could be any cell whose index is larger than or equal 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. Our analysis for loops always unroll the loop once before applying the widening operator. For the sake of clarity, we show only the abstract states computed after the convergence of the sequence of 2 widening iterates. The loop head invariant is shown right after line 6. For the sake of space and readability, we elide the properties of the fields of some groups:

- 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).

$\mathcal{R}^{minix} \wedge \texttt{child} \in G_0 \wedge \texttt{child} > 2$

```
1   void cleanup(int child){
2     int parent = mproc[child].mp_parent;
```

$\mathcal{R}^{minix} \wedge \texttt{child} \in G_0 \wedge \texttt{parent} \in G_0 \wedge \texttt{child} > 2$

```
3     if(parent == 2){
4       mproc[child].mp_flag = 0;
```

$\overline{\texttt{P}}$: $\mathcal{G}_0 \;\wr\; \mathcal{G}_1 \;\wr\; \mathcal{G}_2$

$\overline{\texttt{Q}}$: $0 \le G_0 \le 23 \wedge G_2 = \texttt{child} \wedge 0 \le |G_0| \le 23 \wedge |G_2| = 1 \wedge \texttt{child} > 2 \wedge I_{G_2}^{\texttt{mp\_flag}} = 0 \wedge \texttt{child} \in G_2 \wedge I_{G_0}^{\texttt{mp\_parent}} \in G_0 \cup G_2 \wedge I_{G_2}^{\texttt{mp\_parent}} \in G_0 \cup G_2$

```
5       i = 0;
6       while(i < 24){
```

$\mathcal{I}_{\text{loop}}$:

$\overline{\texttt{P}}$: $\mathcal{G}_0 \;\wr\; \mathcal{G}_1 \;\wr\; \mathcal{G}_2 \;\wr\; \mathcal{G}_3$

$\overline{\texttt{Q}}$: $\texttt{i} \le G_0 \le 23 \wedge 0 \le |G_0| \le 23 \wedge 0 \le |G_3| \le 23 \wedge \texttt{child} > 2 \wedge 1 \le I_{G_3}^{\texttt{mp\_flag}} \le 63 \wedge 0 \le I_{G_3}^{\texttt{mp\_parent}} \le 23 \wedge 0 \le \texttt{i} \wedge \texttt{child} \in G_2 \wedge I_{G_0}^{\texttt{mp\_parent}} \subseteq G_0 \cup G_2 \cup G_3 \wedge I_{G_3}^{\texttt{mp\_parent}} \subseteq G_0 \cup G_3$

```
7         if(mproc[i].mp_flag > 0)
```

$\mathcal{I}_{\text{loop}} \wedge \texttt{i} \in G_0$

```
8           if(mproc[i].mp_parent == child)
```

$\overline{\texttt{P}}$: $\mathcal{G}_0 \;\wr\; \mathcal{G}_1 \;\wr\; \mathcal{G}_2 \;\wr\; \mathcal{G}_3 \;\wr\; \mathcal{G}_4$

$\overline{\texttt{Q}}$: $\texttt{i} < G_0 \le 23 \wedge 0 \le G_0 \le 22 \wedge |G_4| = 1 \wedge \texttt{child} > 2 \wedge 0 \le G_3 < \texttt{i} \wedge \texttt{child} \in G_2 \wedge I_{G_4}^{\texttt{mp\_parent}} \subseteq G_2 \wedge |G_4| = \texttt{i} \wedge 1 \le I_{G_4}^{\texttt{mp\_flag}} \le 63 \wedge 0 \le I_{G_4}^{\texttt{mp\_parent}} \le 23 \wedge 0 \le \texttt{i}$

```
9             mproc[i].mp_parent = 2;
10        i = i + 1;
11      }
```

$\overline{\texttt{P}}$: $\mathcal{G}_1 \;\wr\; \mathcal{G}_2 \;\wr\; \mathcal{G}_3$

$\overline{\texttt{Q}}$: $3 \le G_1 \le 23 \wedge |G_2| = 1 \wedge 0 \le |G_3| \le 23 \wedge G_2 = \texttt{child} > 2 \wedge 0 \le G_3 \le 23 \wedge I_{G_2}^{\texttt{mp\_flag}} = 0 \wedge 1 \le I_{G_3}^{\texttt{mp\_flag}} \le 63 \wedge \texttt{child} \in G_2 \wedge 0 \le I_{G_2}^{\texttt{mp\_parent}} \le 23 \wedge 0 \le I_{G_3}^{\texttt{mp\_parent}} \le 23 \wedge I_{G_2}^{\texttt{mp\_parent}} \subseteq G_3$

```
12    } else {      \\cleanup child and its descendants  }
13  }
```

$\overline{\texttt{P}}$: $\mathcal{G}_1 \;\wr\; \mathcal{G}_2 \;\wr\; \mathcal{G}_3$

$\overline{\texttt{Q}}$: $3 \le G_1 \le 23 \wedge 0 \le |G_1| \le 23 \wedge 1 \le |G_2| \le 2 \wedge 0 \le |G_3| \le 23 \wedge 3 \le |G_2| \le 23 \wedge I_{G_1}^{\texttt{mp\_flag}} = 0 \wedge I_{G_2}^{\texttt{mp\_flag}} = 0 \wedge 1 \le I_{G_3}^{\texttt{mp\_flag}} \le 63 \; I_{G_3}^{\texttt{mp\_parent}} \subseteq G_3 \wedge \texttt{child} \in G_2$

Figure 5.21 – 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, $\texttt{i} \in G_0$. 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 $\overline{\textbf{widen}}^a$ 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, the analysis obtains $\texttt{i} \geq 24$. Since the loop head invariant contains constraints $\texttt{i} \leq G_0$ and $G_0 \leq 23$, 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 contain more elements (more slots might be cleaned up in the **else** branch). With comparison operator $\overline{\textbf{isle}}^a$, our analysis proves that it implies $\mathcal{R}^{minix}$ automatically.

5.7 Experimental Evaluation

We have implemented our analysis and evaluated how it copes with two classes of programs:

- process tables as found in the Minix memory management component;
- 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 [SR12, TCR13, CR13]. It uses the Apron library of numerical abstract domains [JM09]. In practice, our analysis uses octagons [Min06] for all test cases except one that is analyzed using convex polyhedra [CMC08].

5.7.1 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 $\mathcal{R}^{minix}$. 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 $\mathcal{R}^{minix}$, and should return a state that also satisfies $\mathcal{R}^{minix}$ (we recall $\mathcal{R}^{minix}$ was defined in Figure 5.5(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 $\mathcal{R}^{minix}$ 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 `mp_parent`. 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 $\mathcal{R}^{minix}$ holds afterwards (under no pre-condition), and that (2) `fork`, `wait` and `exit` preserve $\mathcal{R}^{minix}$ (i.e., the analysis of each of these functions from pre-condition $\mathcal{R}^{minix}$ returns a post-condition that also satisfies $\mathcal{R}^{minix}$). This verification boils down to the following computations:

$$\begin{array}{lcl}
\overline{\mathbf{isle}}^a(\overline{\mathbf{stat}}[\texttt{mm_init}](\top), \mathcal{R}^{minix}) & = & \mathbf{true} \\
\overline{\mathbf{isle}}^a(\overline{\mathbf{stat}}[\texttt{exit(who)}](\mathcal{R}^{minix} \wedge \texttt{who} \in G_0 \wedge \texttt{who} > 2), \mathcal{R}^{minix}) & = & \mathbf{true} \\
\overline{\mathbf{isle}}^a(\overline{\mathbf{stat}}[\texttt{fork(who)}](\mathcal{R}^{minix} \wedge \texttt{who} \in G_0), \mathcal{R}^{minix}) & = & \mathbf{true} \\
\overline{\mathbf{isle}}^a(\overline{\mathbf{stat}}[\texttt{wait(who)}](\mathcal{R}^{minix} \wedge \texttt{who} \in G_0), \mathcal{R}^{minix}) & = & \mathbf{true}
\end{array}$$

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 5.22, 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).

5.7.2 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

Program	LOCs	Verified property	Time(s)	Max. groups	Num. domain	Description
`mm_init`	26	establishes $\mathcal{R}^{minix}$	0.12	3	Octagons	Minix MMPT: `mproc` init
`fork`	22	preserves $\mathcal{R}^{minix}$	0.07	3	Octagons	Minix MMPT sys. call
`exit`	68	preserves $\mathcal{R}^{minix}$	3.75	6	Octagons	Minix MMPT sys. call
`wait`	70	preserves $\mathcal{R}^{minix}$	3.88	6	Octagons	Minix MMPT sys. call
`complex`	21	$\forall i \in [0, 54], a[i] \geq -1$	0.296	4	Octagons	Example from [CCL11]
`int_init`	8	$\forall i \in [0, N], a[i] = 0$	0.025	3	Octagons	Array initialization

Figure 5.22 – 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)

our algorithms are not specific to segments (and are geared towards the abstraction of non contiguous partitions).

Moreover, Figure 5.23 shows `complex`, an excerpt of an example from [CCL11]. The second example is challenging for most existing techniques, as observed in [CCL11] since resolving `a[index]` at line 10 is tricky. As shown in Figure 5.22, our analysis handles these two loops well, with respectively 4 and 3 groups.

The invariant of the first initialization loop in Figure 5.23 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 the incremention operation `i++`, the widening merges G_2 with G_1, which produces abstract state ① again. At loop exit, the analysis infers that G_0 is empty as it establishes that $56 \leq G_0 \leq 55$. 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 5.4.2). This allows to prove the assertion at line 11.

5.8 Related Work and Conclusion

In this chapter, 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.

Most array analyses require each group be a contiguous array *segment*. Abstract interpretation based static analysis tools [BCC+03b, GRS05, HP08] and [CCL11] contiguously partition arrays over indexes statically and dynamically respectively. Tools based on decision procedures [AGS13, AGS14, BMS06], and theorem provers [JM07, SPW09, McM08, KV09] 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, [DDA10, DDA11] 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, [DDA10, DDA11] cannot express the invariants $\mathcal{R}^{minix}$ 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 *set-set* predicates. Similarly, HOO [CCR14] is an effective abstract domain for containers and JavaScript open objects. As it uses a set abstract domain [CCS13], 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 [BHMR07, GMT08] 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, set 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 a standard variable in $\overline{\mathtt{Q}}$. Our domain can verify memory safety invariants involving complex data structure in operating systems, which could not be achieved by previous static analyses [PTS+11, YLB+08] which also target system code.

```
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);
```

(a) Test case `complex`

state ①

$\overline{\mathtt{P}}$: $\boxed{\mathcal{G}_0 \;\wr\; \mathcal{G}_1}$

$\overline{\mathtt{Q}}$: $\mathtt{i} \le G_0 \le 55 \wedge |G_0| = 56 - \mathtt{i} \wedge |G_1| = \mathtt{i}$
$\wedge 0 \le G_1 \le \mathtt{i} - 1 \wedge \mathtt{i} \in G_0 \wedge I_{G_1}^{\mathtt{value}} = 0$

state ②

$\overline{\mathtt{P}}$: $\boxed{\mathcal{G}_0 \;\wr\; \mathcal{G}_1 \;\wr\; \mathcal{G}_2}$

$\overline{\mathtt{Q}}$: $\mathtt{i} + 1 \le G_0 \le 55 \wedge |G_0| = 55 - \mathtt{i} \wedge |G_1| = \mathtt{i}$
$0 \le G_1 \le \mathtt{i} - 1 \wedge I_{G_1}^{\mathtt{value}} = 0 \wedge I_{G_2}^{\mathtt{value}} = 0$
$G_2 = \mathtt{i} \wedge \mathtt{i} \in G_2 \wedge |G_2| = 1$

state ③

$\overline{\mathtt{P}}$: $\boxed{\mathcal{G}_1 \;\wr\; \mathcal{G}_2 \;\wr\; \mathcal{G}_3}$

$\overline{\mathtt{Q}}$: $0 \le G_1 \le 54 \wedge |G_1| = 54 \wedge G_2 = 1 \wedge G_3 = 1$
$\wedge 0 \le G_2 \le 54 \wedge -1 \le I_{G_1}^{\mathtt{value}} \wedge -1 \le I_{G_2}^{\mathtt{value}}$
$\wedge G_3 = 55 \wedge \mathtt{i} \in G_1 \cup G_2$

(b) Invariants

Figure 5.23 – Array random accesses

Chapter 6

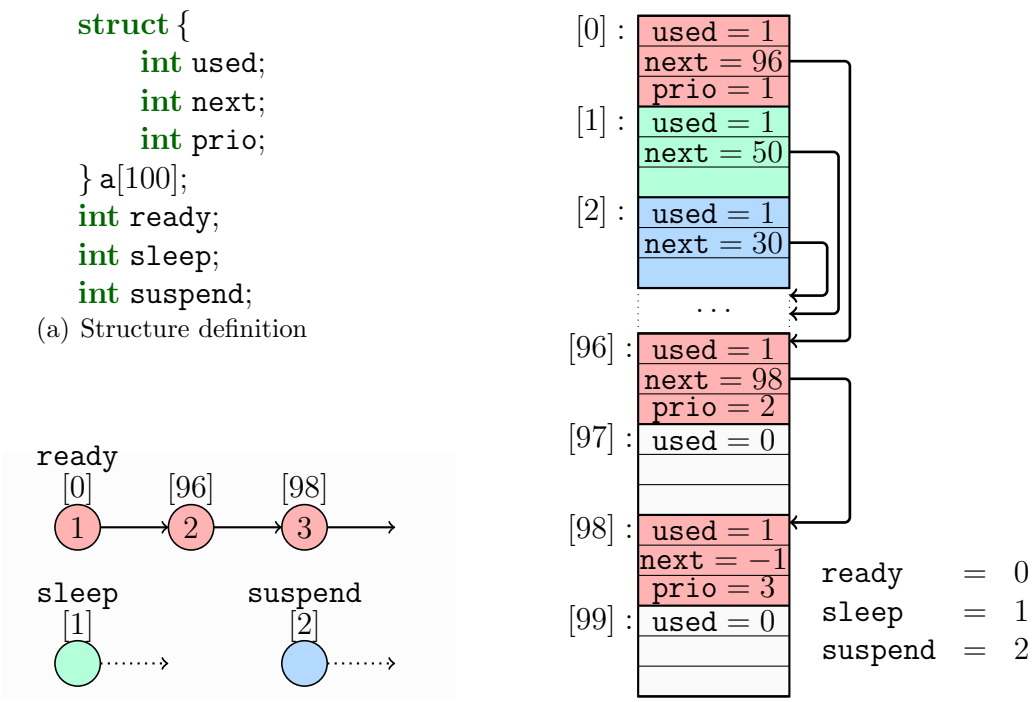
Coalescing Array and Shape Abstraction

In this chapter, we propose a technique to combine different shape abstractions. This combination locally ties summaries in both abstract domains and is called a coalescing abstraction. Coalescing allows to define efficient and precise static analysis algorithms on the combined abstraction, which can express new properties on the structure of memory. As an instance, we show the combination of the array abstraction in Chapter 5 and a shape abstraction which captures linked structures with separation logic-based inductive predicates. This leads to an automatic static analysis for the verification of programs manipulating arrays storing linked structures, such as lists in an array. This programming pattern is commonly used in low-level systems, which avoids relying on dynamic allocation. The verification of such programs is non-trivial as it requires reasoning both about the array structure with numeric indexes and about the linked structures stored in the array.

6.1 Context of The Analysis

In this section, we set the context of the analysis and recall the motivation for the coalescing domain. We use the same language as in Chapter 5.

An example taken from a real-world OS. Figure 6.1(a) shows a structure taken from an industrial real-time embedded operating system (called AOS), that stores three *task priority lists* a single array. Each cell is either unused, or it stores the information associated to a task. Moreover, the cells representing tasks are arranged into three (generally non contiguous) lists: the list of tasks that are ready (with head `a[ready]`), the list of sleeping tasks (with head `a[sleep]`), and the list of suspended tasks (with head `a[suspend]`). The next links of list elements are stored in the field `next` of array cells. Additionally, each cell has a field `used`, which indicates whether it corresponds to an actual process or a free slot (if `used` contains 0), and field `prio` indicates the priority



(a) Structure definition

(b) Topology of the lists

(c) A concrete state

Figure 6.1 – Three linked lists in one array

level of the corresponding task. The list of ready processes is sorted in increasing priority order. Figure 6.1(c) and Figure 6.1(b) respectively show an excerpt of a concrete state and its topology. In Figure 6.1(b), the number in each list node corresponds to the value of field `prio`. In this configuration, cells 0, 96, 98 describe processes that are ready, cell 1 describes a sleeping process, and cell 2 stores a suspended process; last, cells 97 and 99 are unused. System calls that manipulate this data structure include **init** (initialize the array and the three list variables), **create** (locate a free slot in the array and insert it into the ready list), **stop** (release a list node to be free), and **schedule** (move array nodes between lists).

Invariants. The following invariants should hold before and after each system call:

- variables `ready`, `sleep` and `suspend` should point to the heads of three well-formed acyclic disjoint lists, where the `next` field of each cell holds the index of the next element, and the end-of-list is encoded by index -1;
- free slots and used nodes in the array are distinguished by the values stored in their `used` field (0 for free slots and 1 for used slots); furthermore, any used slot belongs to one of the three lists;
- the list with head `ready` is sorted with respect to the values in field `prio`.

We call these invariants *global correctness condition* in AOS, denoted by $\mathcal{R}^{aos}$. These invariants are essential to the correctness of the operating system, since their violation may cause issues such as out-of-bound array accesses or tasks becoming dangling. Thus, it is necessary to verify that system calls preserve $\mathcal{R}^{aos}$.

To verify a system call, we can let a static analyzer assume pre-condition $\mathcal{R}^{aos}$, analyze the body, and attempt to verify that the post-condition also satisfies $\mathcal{R}^{aos}$. To achieve this, it is necessary to reason at the same time about (1) the accesses into dynamic structures via both their `next` link (they are chained dynamically even though the array is statically allocated) and array indexes, (2) the shapes of the linked structures, (3) the non-contiguousness of regions occupied by each dynamic structure, (4) the sortedness of list structures. Each of these points is non trivial.

Shape analysis for dynamic structures like [SRW99b, DOY06, CR08] do not cope with array-specific statements, like accesses with random indexes. The non-contiguous array partition of Chapter 5 can express (3) using numeric predicates of non-contiguous sets of cells that should be abstracted together, but cannot describe nested structures that are dynamically linked, thus would fail to meet (2) and (4).

In this chapter, we propose a *coalescing* domain, which provides an efficient and precise method of combining different shape domains to deal with intertwined data structures. As an instance, the coalescing of the array abstraction from Chapter 5 and a heap abstraction can express properties (1), (2), (3) and (4). Our analysis proceeds by abstract interpretation, and is parameterized by the structural invariant $\mathcal{R}^{aos}$.

A system call. As an example, in the remainder of this chapter, we focus on the function `create` (shown in Figure 6.2), which preserves the invariant $\mathcal{R}^{aos}$. Function `create` locates a free slot (line 3 to 10), initializes it with the given priority and then inserts it (line 13 to 28) into the sorted list with head `ready`. It takes the priority of the new task as parameter `priority`. For concision, we omit some cases and lines that are not immediately relevant. To verify the correctness of `create`, we let our analysis compute an abstract post-condition under the pre-condition that property $\mathcal{R}^{aos}$ holds and check that $\mathcal{R}^{aos}$ still holds at the exit of the function. This boils down to the verification of:

$$\textbf{assume}(\mathcal{R}^{aos});\ \texttt{create}(\texttt{priority});\ \textbf{assert}(\mathcal{R}^{aos});$$

6.2 Abstraction

This section formalizes the coalescing of two abstract domains both of which deal with summaries of memory blocks (we call them memory abstract domains). A signature of memory abstract domains is presented in Section 6.2.1 which defines a family of memory abstract domains that can be coalesced. A shape domain that we take as an instance of memory abstract domain is presented in Section 6.2.2. The principles for the coalescing of such domains are described in Section 6.2.3. The resulting coalescing domain of the

```
1  void create(int priority){
2    int i = 0;
3    while(i < 100){
4      if(a[i].used == 0){
5        a[i].used = 1;
6        a[i].prio = priority;
7        break;
8      }
9      i ++;
10   }
11   // corner cases
12     ...
13   // insert a[i] to ready list
14   // for case ready  = -1
15     ...
16   // for case a[i].prio ≤ a[ready].prio
17       ...
18   int pre, cur;
19   pre = ready;
20   cur = a[ready].next;
21   while(cur != −1){
22     if(a[cur].prio > priority)
23       break;
24     pre = cur;
25     cur = a[cur].next;
26   }
27   a[pre].next = i;
28   a[i].next = cur;
29   // other fields initialization
30     ...
31   return i;
32 }
```

Figure 6.2 – Code of function `create`

array domain in Chapter 5 and the shape domain introduced in Section 6.2.2 is shown in Section 6.2.4.

Concrete states. In this chapter, we use the same definition of concrete states as that in Chapter 5. Here, we just recall the notations.

- $\mathbb{A}$: the set of program variables of array type;
- $\mathbb{X}$: the set of program scalar variables;
- $\mathbb{I}$: the set of non-negative integers;
- $\mathbb{F}$: the set of fields;
- $\mathbb{V}$: the set of values.

Definition 6.1 (Concrete states)**.** *A concrete state is a partial function mapping basic cells (base variables and fields of array cells) into values, denoted as σ. The set $\mathbb{S}$ of concrete states is defined by*

$$\sigma \in \mathbb{S} = (\mathbb{A} \times \mathbb{I} \times \mathbb{F} \cup \mathbb{X}) \to \mathbb{V}$$

More specifically, the set of the field names of the elements of array `a` is denoted by $\mathbb{F}_{\mathtt{a}}$, and the set of valid indexes in `a` is denoted by $\mathbb{I}_{\mathtt{a}}$.

6.2.1 A Signature of Memory Abstract Domains

To define the general principles of coalescing, we first need to define the signature of underlying memory abstract domains.

Abstract domains are characterized by abstract elements and transfer functions. A family of abstract domains can be defined by an *abstract domain signature*. A signature $\mathbb{D}$ of abstract domains consists of a definition of the set of concrete elements, a description for the set of abstract elements, and a list of descriptions for the transfer functions and operations.

Our coalescing can only be applied to a class of memory abstract domains, which can be specified with a signature $\mathbb{D}^m$.

Definition 6.2 (A signature of memory abstract domains: $\mathbb{D}^m$)**.** *The signature of memory abstract domains:* $\mathbb{D}^m$ *is defined below.*

$\mathbb{S}$	$=$	$(\mathbb{A} \times \mathbb{I} \times \mathbb{F} \cup \mathbb{X}) \to \mathbb{V}$	*the set of concrete elements*
$\overline{\mathtt{M}}$	$::=$	$\overline{\mathtt{A}} * \ldots * \overline{\mathtt{A}}$	*separating conjunction*
$\overline{\mathtt{A}}$	$::=$	$\overline{\mathtt{b}}$	*non-inductive memory predicates*
	$\mid$	$\mathbf{i}$	*inductive memory predicates*
$\overline{\mathtt{N}}$			*numeric predicates*
$\overline{D}$	$=$	$\overline{\mathcal{D}}_{\overline{\mathtt{M}}} \times \overline{\mathcal{D}}_{\overline{\mathtt{N}}}$	*the set of abstract elements*
γ	$\in$	$\overline{D} \to \mathcal{P}(\mathbb{S})$	*concretization*
$\top$	$\in$	$\overline{D}$	*top element*
$\bot$	$\in$	$\overline{D}$	*bottom element*
$\overline{\mathbf{unfold}}$	$\in$	$\overline{D} \to \overline{D}$	*unfolding operator*
$\overline{\mathbf{fold}}$	$\in$	$\overline{D} \to \overline{D}$	*folding operator*
$\overline{\mathbf{guard}}[.]$	$\in$	$\overline{D} \to \overline{D}$	*transfer function for condition tests*
$\overline{\mathbf{assign}}[.]$	$\in$	$\overline{D} \to \overline{D}$	*transfer function for assignments*
$\overline{\mathbf{isle}}$	$\in$	$\overline{D} \times \overline{D} \to \overline{D}$	*inclusion check operator*
$\overline{\mathbf{join}}$	$\in$	$\overline{D} \times \overline{D} \to \overline{D}$	*join operator*
$\overline{\mathbf{widen}}$	$\in$	$\overline{D} \times \overline{D} \to \overline{D}$	*widening operator*

This signature is quite standard except that it defines some specific requirements on the form of abstract elements. It requires that all abstract elements contain two components: memory predicates $\overline{\mathtt{M}}$ (predicates on symbolic abstraction of memory states) and numeric predicates $\overline{\mathtt{N}}$ (predicates on the numeric properties of symbolic variables from memory predicates and programs). The set of all memory predicates and numeric predicates are denoted as $\overline{\mathcal{D}}_{\overline{\mathtt{M}}}$ and $\overline{\mathcal{D}}_{\overline{\mathtt{N}}}$ respectively. One requirement on the memory predicates is that different atomic memory predicates (i.e., $\overline{\mathtt{A}}$) should be compounded by separating conjunction (denoted as $*$), i.e., they constrain disjoint memory blocks. This is formalized as $\overline{\mathtt{A}} * \ldots * \overline{\mathtt{A}}$. The other requirement on memory predicates is that atomic memory predicates should include both inductive predicates (i.e., $\mathbf{i}$) and non-inductive predicates (i.e., $\overline{\mathtt{b}}$). *Inductive predicates* describe possible configurations of summarized

memory contents using a recursive property. They are instances of generic definitions supplied to the analyzer as abstract domain parameters before it is launched.

Definition 6.3 (Inductive predicates)**.** *The form of inductive predicates is defined below.*

$$\mathbf{i} := \langle \overline{\mathtt{A}}_{0,0} * \ldots * \overline{\mathtt{A}}_{0,n_0}, \overline{\mathtt{N}}_0 \rangle \vee \ldots \vee \langle \overline{\mathtt{A}}_{k,0} * \ldots * \overline{\mathtt{A}}_{k,n_k}, \overline{\mathtt{N}}_k \rangle$$

In the definition above, each element of the disjunction (called inductive case) consists of a separating conjunction [Rey02] of atomic memory predicates together with a conjunction of numeric constraints. This definition is more general than some others in the literature, where the form of $\overline{\mathtt{A}}_{i,j}$ is often restricted. For instance, in the inductive definition of list in [CR08], the non-inductive atomic memory predicates in any case must describe the head or the tail.

6.2.2 Introduction to A Shape Domain

Now we introduce a separation logic based domain $\overline{D}^s$ [CR08], which is used to track linked structures. This shape domain fits into the signature defined in Section 6.2.1. Coalescing this domain with our non-contiguous partitioning array domain, the analysis is able to describe the invariant $\mathcal{R}^{aos}$. As in the signature, an abstract element in this shape domain is composed of two parts: memory predicates and numeric predicates.

Definition 6.4 (Memory predicates of a shape domain)**.** *The syntax of memory predicates* $\overline{\mathtt{M}}^s$ *in* $\overline{D}^s$ *is defined as follows.*

$\overline{\mathtt{M}}^s$	:=	$\overline{\mathtt{A}}^s * \ldots * \overline{\mathtt{A}}^s$	*Separating conjunction*
$\overline{\mathtt{A}}^s$	:=	$\alpha@\vec{\mathtt{f}} \mapsto \vec{\beta}$	*Predicates on a single memory cell*
	|	**emp**	*Predicates on an empty region*
	|	$\mathbf{i}^s(\vec{\alpha})$	*Predicates on a possibly empty set of memory cells*
	|	**true**	*True predicates*

An atomic memory predicate $\overline{\mathtt{A}}^s$ either describes a single cell with the index and contents (denoted as $\alpha@\vec{\mathtt{f}} \mapsto \vec{\beta}$, where symbolic variables α represents the index and $\vec{\beta}$ represent the values stored in each field), or an empty region (denoted as **emp**), or a complex structure summarized by an inductive predicate (noted as $\mathbf{i}^s(\vec{\alpha})$), or no constraint (noted as **true**, which is our extension to [CR08]). The symbolic variables described by Greek letters denote values (array indexes, numeric or pointer values...). Unless specified otherwise, we use α, β to denote actual parameters, and π, τ to denote the formal parameters of inductive predicates.

The numeric predicates in $\overline{D}^s$ are a subset of the numeric predicates in the array domain in Chapter 5 and can be described by the extended Maya+ domain. Thus we still use $\overline{\mathtt{Q}}$ to denote them.

The shape domain [CR08] assumes a restriction on the inductive definition $\mathbf{i}^s(\vec{\pi})$ in $\overline{D}^s$: each inductive case should correspond to a memory region reachable from π (i.e., the first parameter) and do not contain **true**. This is for the ease of unfolding/folding algorithm [CR08].

Our extension **true** is a special memory predicate. It is a non-inductive predicate. But in some algorithms, it is often taken as an inductive predicate, where its inductive definition is the disjunction of all tuples of memory predicates and numeric predicates in $\overline{D}^s$.

Example 6.1 (Inductive predicates in a shape domain)**.** *The following inductive predicate* **lseg** *is the formal definition of list segment in* $\overline{D}^s$*. Symbolic variable* π *stores the index of the first element of the list segment, and* τ *stores the value in the* `next` *field of the last element.*

$$\begin{array}{rcl} \mathbf{lseg}(\pi,\tau) & := & \langle \mathbf{emp}, \pi = \tau \rangle \\ & \vee & \langle \pi @ (\mathtt{next} \mapsto \pi') * \mathbf{lseg}(\pi', \tau), \pi \neq \tau \rangle \end{array}$$

Definition 6.5 (Concretization function in the shape domain)**.** *The concretization function* γ^s *for* $\overline{D}^s$ *maps an abstract element* $(\overline{\mathtt{M}}^s, \overline{\mathtt{Q}})$ *to a set of concrete states* $\sigma \in \mathbb{S}$*. The concretization is defined as follows.*

$$\gamma^s(\overline{\mathtt{M}}^s, \overline{\mathtt{Q}}) = \{\sigma \in \mathbb{S} \mid \exists \nu \in \gamma_{\overline{\mathtt{Q}}}(\overline{\mathtt{Q}}),\ (\sigma, \nu) \models \overline{\mathtt{M}}^s \wedge \forall \mathtt{x} \in \mathbb{X},\ \nu(\mathtt{x}) = \sigma(\mathtt{x})\}$$

where

$$\begin{array}{rcll} (\sigma,\nu) & \models & \alpha @ \vec{\mathtt{f}} \mapsto \vec{\beta} \quad \mathtt{iff} & \forall \mathtt{f} \in \mathbb{F}, \sigma(\mathtt{a}[\nu(\alpha)] \cdot \mathtt{f}) \in \nu(\beta^{\mathtt{f}}) \\ (\sigma,\nu) & \models & \mathbf{i}^s(\vec{\alpha}) \quad \mathtt{iff} & \textit{there exists a disjunctive case } \langle \overline{\mathtt{A}}_0^s * \ldots * \overline{\mathtt{A}}_n^s, \overline{\mathtt{Q}} \rangle \textit{ of } \mathbf{i}^s(\vec{\pi}) \\ & & & \textit{such that, } (\sigma,\nu) \models \langle \overline{\mathtt{A}}_0^s * \ldots * \overline{\mathtt{A}}_n^s, \overline{\mathtt{Q}} \rangle \\ (\sigma,.) & \models & \mathbf{emp} \quad \mathtt{iff} & \sigma = [.] \textit{ is an empty store} \\ (\sigma,.) & \models & \mathbf{true} \quad \mathtt{iff} & \sigma \in \mathbb{S} \\ (\sigma_0 * \sigma_1, \nu) & \models & \overline{\mathtt{M}}_0^s * \overline{\mathtt{M}}_1^s \quad \mathtt{iff} & (\sigma_0,\nu) \models \overline{\mathtt{M}}_0^s \wedge (\sigma_1,\nu) \models \overline{\mathtt{M}}_1^s \end{array}$$

6.2.3 Principles of Coalescing Memory Abstract Domains

Given two memory abstract domains that fit the signature in Definition 6.2, we can define the coalescing domain as follows.

Definition 6.6 (Coalescing domain)**.** *Suppose that domains* $\overline{D}^\bullet$ *and* $\overline{D}^\diamond$ *fit the signature* $\mathbb{D}^m$*, then an element* $\overline{\mathtt{c}}$ *in their coalescing domain* $\mathcal{C}$ *is a tuple* $(\mathfrak{M}, \mathfrak{N})$*, where* $\mathfrak{M}$ *and* $\mathfrak{N}$ *follows the syntax defined below.*

$$\begin{array}{lcl} \mathfrak{M} & ::= & \mathfrak{A} * \ldots * \mathfrak{A} \\ \mathfrak{A} & ::= & \mathbf{i}^\bullet \,\&\, \mathbf{i}^\diamond \\ & | & \overline{\mathtt{b}}^\bullet \,\&\, \overline{\mathtt{b}}^\diamond \\ \mathfrak{N} & :: & \overline{\mathtt{N}}^\bullet \wedge \overline{\mathtt{N}}^\diamond \end{array}$$

Memory predicates $\mathfrak{M}$ in the coalescing domain are separating conjunction of atomic memory predicates $\mathfrak{A}$. The set of memory predicates $\mathfrak{M}$ is denoted as $\overline{\mathcal{D}}_{\mathfrak{M}}$.

An atomic memory predicate $\mathfrak{A}$ in the coalescing domain can be a non-separating conjunction of non-inductive predicates $\overline{\mathtt{b}}^\bullet \,\&\, \overline{\mathtt{b}}^\diamond$ or a non-separating conjunction of inductive predicates $\mathbf{i}^\bullet \,\&\, \mathbf{i}^\diamond$. Note that we restrict the use of $\&$ in the following way: it can only be applied to two atomic inductive memory predicates or two atomic non-inductive memory predicates. This restriction indeed is the essential idea behind coalescence, which brings additional precision and makes unfolding/folding in the coalescing domain significantly simpler.

A non-separating conjunction of non-inductive predicates $\overline{\mathtt{b}}^\bullet \,\&\, \overline{\mathtt{b}}^\diamond$ denotes that the addresses and contents of a possibly empty set of memory cells satisfy memory predicates $\overline{\mathtt{b}}^\bullet$ and $\overline{\mathtt{b}}^\diamond$ simultaneously. A non-separating conjunction of inductive predicates $\mathbf{i}^\bullet \,\&\, \mathbf{i}^\diamond$ is also inductive, and is called a *coalescing inductive predicate*. Coalescing inductive predicates parameterize the coalescing domain, just like $\mathbf{i}^s$ in the shape domain of Section 6.2.2.

Definition 6.7 (Coalescing Inductive Predicates)**.** *The definition of a coalescing inductive predicate* $\mathbf{i}^\bullet \,\&\, \mathbf{i}^\diamond$ *is of the form*

$$\begin{array}{rl} \mathbf{i}^\bullet \,\&\, \mathbf{i}^\diamond := & \langle \mathfrak{A}_{0,0} * \ldots * \mathfrak{A}_{0,n_0}, \mathfrak{N}_0 \rangle \\ & \vee \ldots \vee \\ & \langle \mathfrak{A}_{k,0} * \ldots * \mathfrak{A}_{k,n_k}, \mathfrak{N}_k \rangle \end{array}$$

where if $\mathfrak{A}_{i,j}$ *is denoted as* $\overline{\mathtt{A}}^\bullet_{i,j} \,\&\, \overline{\mathtt{A}}^\diamond_{i,j}$*, then*

$$\mathbf{i}^\bullet ::= \langle \overline{\mathtt{A}}^\bullet_{0,0} * \ldots * \overline{\mathtt{A}}^\bullet_{0,n_0}, \mathfrak{N}_0 \rangle \vee \ldots \vee \langle \overline{\mathtt{A}}^\bullet_{k,0} * \ldots * \overline{\mathtt{A}}^\bullet_{k,n_k}, \mathfrak{N}_k \rangle$$

and

$$\mathbf{i}^\diamond ::= \langle \overline{\mathtt{A}}^\diamond_{0,0} * \ldots * \overline{\mathtt{A}}^\diamond_{0,n_0}, \mathfrak{N}_0 \rangle \vee \ldots \vee \langle \overline{\mathtt{A}}^\diamond_{k,0} * \ldots * \overline{\mathtt{A}}^\diamond_{k,n_k}, \mathfrak{N}_k \rangle$$

are both valid inductive definitions in the domains $\overline{D}^\bullet$ *and* $\overline{D}^\diamond$ *respectively.*

This definition ensures that a coalescing inductive definition can be decomposed into two inductive definitions from the two underlying domains respectively. This allows our coalescing domain to utilize the algorithms manipulating inductive predicates in the underlying domains.

Definition 6.8 (Concretization function in the coalescing domain)**.** *Let* $\gamma^\bullet$ *(resp.* $\gamma^\diamond$*) be the concretization function in the memory abstract domain* $\overline{D}^\bullet$ *(resp.* $\overline{D}^\diamond$*). It maps an abstract element* $(\overline{\mathtt{M}}^\bullet, \overline{\mathtt{N}}^\bullet)$ *(resp.* $(\overline{\mathtt{M}}^\diamond, \overline{\mathtt{N}}^\diamond)$*) to a set of concrete states* $\mathcal{S} \subseteq \mathbb{S}$*. The*

concretization function γ^c *for coalescing domain* $\mathcal{C}$ *maps* $(\mathfrak{M}, \mathfrak{N})$ *to a set of concrete states, defined with the relation* $\models$ *as follows.*

$$\gamma^c(\mathfrak{M}, \mathfrak{N}) = \{\sigma \in \mathbb{S} \mid \sigma \models (\mathfrak{M}, \mathfrak{N})\}$$

$$\begin{array}{rcl}
\sigma & \models & (\mathbf{i}^\bullet \,\&\, \mathbf{i}^\diamond, \mathfrak{N}) \quad \texttt{iff} \quad \textit{there exists a disjunctive case } \langle \mathfrak{A}_0 * \ldots * \mathfrak{A}_n, \mathfrak{N}' \rangle \textit{ of } \mathbf{i}^\bullet \,\&\, \mathbf{i}^\diamond \\
& & \textit{such that,} \quad \sigma \models (\mathfrak{A}_0 * \ldots * \mathfrak{A}_n, \mathfrak{N} \wedge \mathfrak{N}') \\
\sigma & \models & (\overline{\mathtt{b}}^\bullet \,\&\, \overline{\mathtt{b}}^\diamond, \mathfrak{N}) \quad \texttt{iff} \quad \sigma \in \gamma^\bullet(\overline{\mathtt{b}}^\bullet, \mathfrak{N}) \cap \gamma^\diamond(\overline{\mathtt{b}}^\diamond, \mathfrak{N}) \\
\sigma_0 * \sigma_1 & \models & (\mathfrak{M}_0 * \mathfrak{M}_1, \mathfrak{N}) \quad \texttt{iff} \quad \sigma_0 \models (\mathfrak{M}_0, \mathfrak{N}) \wedge \sigma_1 \models (\mathfrak{M}_1, \mathfrak{N})
\end{array}$$

This definition of the concretization function is quite straightforward. But it is worth noting that the concretization of a coalescing inductive predicates is not the intersection of the concretizations of the decomposed inductive predicates in the two underlying domains.

6.2.4 The Array/Shape Coalescing Domain

Following the principles introduced in Section 6.2.3, we would like to define the coalescing domain of the array domain in Chapter 5 and the shape domain introduced in Section 6.2.2. However, the array domain does not fit the signature in Section 6.2.1 strictly, since it does not contain inductive predicates. Actually, the atomic memory predicates in the array domain are of the form $G_i@\vec{\mathtt{f}} \mapsto \vec{I^{\mathtt{f}}_{G_i}}$, which can be seen as both non-inductive and inductive. The following definition gives an inductive view of the atomic memory predicates in the array domain.

$$\begin{array}{rcl}
G@\vec{\mathtt{f}} \mapsto \vec{I^{\mathtt{f}}_G} & ::= & \langle \mathbf{emp}, |G| = 0 \rangle \\
& \mid & \langle G_0@\vec{\mathtt{f}} \mapsto \vec{I^{\mathtt{f}}_{G_0}} * \ldots * G_n@\vec{\mathtt{f}} \mapsto \vec{I^{\mathtt{f}}_{G_n}}, \\
& & G = \uplus_{0 \le i \le n} G_i \wedge \forall \mathtt{f} \in \mathbb{F}, I^{\mathtt{f}}_G = \uplus_{0 \le i \le n_0} I^{\mathtt{f}}_{G_{0,i}} \wedge \mid G_0 \mid = 1 \rangle
\end{array}$$

In this inductive definition, we can see that a group could be either empty or split into several sub-groups, where the first group contains only one array cell. The $\overline{\textbf{unfold}}$ and $\overline{\textbf{fold}}$ operations in the signature correspond to the $\overline{\textbf{split}}$ and $\overline{\textbf{merge}}$/$\overline{\textbf{create}}$ respectively. With this view, the array domain fits the signature $\mathbb{D}^m$, and we can coalesce it with the shape domain [CR08]. The resulting domain is called *the array/shape coalescing domain.*

Definition 6.9 (The array/shape coalescing domain)**.** *An element* $(\mathfrak{M}^{a/s}, \mathfrak{N}^{a/s})$ *in the array/shape coalescing domain is defined as follows.*

$$\begin{array}{rcl}
\mathfrak{M}^{a/s} & := & \mathfrak{A}^{a/s} * \ldots * \mathfrak{A}^{a/s} \\
\mathfrak{A}^{a/s} & := & \mathcal{G}_i \,\&\, \alpha@\vec{\mathtt{f}} \mapsto \vec{\beta} \mid \mathcal{G}_i \,\&\, \mathbf{emp} \mid \mathcal{G}_i \,\&\, \mathbf{i}^s(\vec{\alpha}) \mid \mathcal{G}_i \,\&\, \mathbf{true} \\
\mathfrak{N}^{a/s} & := & \overline{\mathtt{Q}}
\end{array}$$

Note that we still use $\mathcal{G}_i$ for short of $G_i@\vec{\mathtt{f}} \mapsto \vec{I^{\mathtt{f}}_{G_i}}$, which means $\mathcal{G}_i \,\&\, \mathbf{i}^s(\vec{\alpha})$ is a shortcut for $G_i@\vec{\mathtt{f}} \mapsto \vec{I^{\mathtt{f}}_{G_i}} \,\&\, \mathbf{i}^s(\vec{\alpha})$. A memory predicate in the coalescing domain can describe

- a single cell in the array: $\mathcal{G}_i$ & $\alpha@\vec{\mathtt{f}} \mapsto \vec{\beta}$;
- an empty group: $\mathcal{G}_i$ & **emp**;
- a possibly empty group of cells with structural predicates: $\mathcal{G}_i$ & $\mathbf{i}^s(\vec{\alpha})$ (by "structural", we mean that the data in the corresponding memory region forms a definable structure);
- a possibly empty group of cells without structural predicates: $\mathcal{G}_i$ & **true**.

The concretization of the coalescing domain follows the principles described in Section 6.2.3.

Now we use the array/shape coalescing domain to abstract the global correctness condition $\mathcal{R}^{aos}$. We first define the inductive predicates in it.

Example 6.2 (Coalescing inductive definition)**.** *The coalescing inductive definition* $\mathcal{G}$ & $\mathbf{lseg}(\pi, \tau)$ *formalizes a possibly empty linked list. The definition of it is shown below (recall that* $\mathcal{G}$ *is short for* $G@\mathtt{next} \mapsto I_G^{\mathtt{next}}$*).*

$$\begin{array}{rcl} \mathcal{G} \ \& \ \mathbf{lseg}(\pi,\tau) & ::= & \langle \mathcal{G} \ \& \ \mathbf{emp}, \pi = \tau \wedge |G| = 0\rangle \\ & \vee & \langle \mathcal{G}' \ \& \ \pi@(\mathtt{next} \mapsto \pi') * \mathcal{G}'' \ \& \ \mathbf{lseg}(\pi', \tau), \pi \neq \tau \wedge |G'| = 1\rangle \end{array}$$

In this definition, group $\mathcal{G}$ *either is an empty group, or it can be split into two groups, where the first one contains the list head, and the other consists of the rest of the list. In the following, if some numeric constraints (e.g.,* $|G| = 0$*) in the coalescing definition are implied by the memory predicates (e.g.,* $\mathcal{G}$ & **emp***), we will not write them.*

If we break down this definition into two inductive definitions in the underlying domains, they are still valid in their domains, as shown below.

$$\begin{array}{rcl} \mathcal{G} & ::= & \langle \mathcal{G}, \pi = \tau \wedge |G| = 0\rangle \\ & \vee & \langle \mathcal{G}' * \mathcal{G}'', \pi \neq \tau \wedge |G'| = 1\rangle \\ \mathbf{lseg}(\pi,\tau) & ::= & \langle \mathbf{emp}, \pi = \tau \wedge |G| = 0\rangle \\ & \vee & \langle \pi@(\mathtt{next} \mapsto \pi') * \mathbf{lseg}(\pi', \tau), \pi \neq \tau \wedge |G| = 1\rangle \end{array}$$

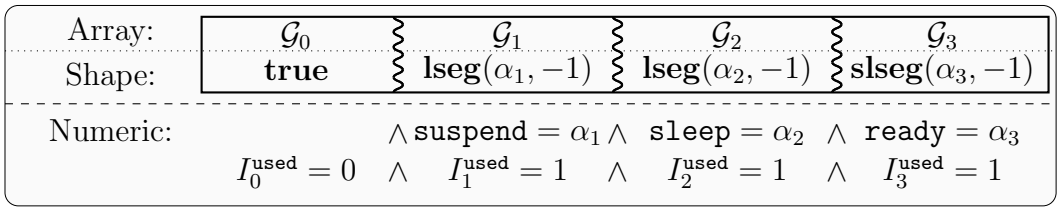
Example 6.3 (A coalescing inductive predicate on sorted lists in arrays)**.** *Inductive predicate* $\mathcal{G}$ & $\mathbf{slseg}(\pi, \tau)$ *describes a sorted list in an array. Its formal definition is shown as follows.*

$$\begin{array}{rcl} \mathcal{G} \ \& \ \mathbf{slseg}(\pi,\tau) & ::= & \langle \mathcal{G} \ \& \ \mathbf{emp}, \pi = \tau\rangle \\ & \vee & \langle \mathcal{G}' \ \& \ \pi@(\mathtt{next} \mapsto \pi', \mathtt{prio} \mapsto \pi'') * \mathcal{G}'' \ \& \ \mathbf{slseg}(\pi', \tau), \\ & & \pi \neq \tau \wedge \pi'' \leq I_{G''}^{\mathtt{prio}}\rangle \end{array}$$

In the second case of this definition, the linked list described by $\mathcal{G}$ & $\mathbf{slseg}(\pi, \tau)$ *can be split into two new groups:* $\mathcal{G}'$ *contains only one array cell which is also the list head and* $\mathcal{G}''$ *consists of the rest of the list. The sortedness in this definition is expressed by the relation on* π'' *and* $I_{G''}^{\mathtt{prio}}$*.*

$$\mathfrak{M}^{a/s} \;:\; \mathcal{G}_0 \,\&\, \mathbf{true} * \mathcal{G}_1 \,\&\, \mathbf{lseg}(\alpha_1, -1) * \mathcal{G}_2 \,\&\, \mathbf{lseg}(\alpha_2, -1) * \mathcal{G}_3 \,\&\, \mathbf{slseg}(\alpha_3, -1)$$
$$\mathfrak{N}^{a/s} \;:\; \mathtt{suspend} = \alpha_1 \;\wedge\; \mathtt{sleep} = \alpha_2 \;\wedge\; \mathtt{ready} = \alpha_3 \;\wedge\; I_0^{\mathtt{used}} = 0 \;\wedge\; I_1^{\mathtt{used}} = I_2^{\mathtt{used}} = I_3^{\mathtt{used}} = 1$$

(a) The formal description of $\mathcal{R}^{aos}$



(b) The graphical description of $\mathcal{R}^{aos}$

Figure 6.3 – Abstract state corresponding to $\mathcal{R}^{aos}$

Figure 6.3(a) shows the abstract state $(\mathfrak{M}^{a/s}, \mathfrak{N}^{a/s})$ corresponding to $\mathcal{R}^{aos}$. We also propose a graphical representation of the abstract state to make it easy to read in Figure 6.3(b). In this graphical representation, separating conjunction is expressed with winding lines and non-separating conjunction is expressed by putting the conjuncts in the same column. In the following, we will only show graphical representations by default.

The abstract state partitions the array `a` into four disjoint groups of cells, such that each group corresponds to cells with similar properties. Group G_0 collects all free slots, whereas groups G_1, G_2, and G_3 respectively account for the lists of suspended, sleeping and ready tasks. Following $\mathcal{R}^{aos}$, free slots are characterized with a `used` field storing 0 as $I_0^{\mathtt{used}} = 0$, which means that all values in set variable $I_0^{\mathtt{used}}$ are equal to 0 (though this set may also be empty). Predicates $\mathcal{G}_3$ & $\mathbf{lseg}(\alpha_3, -1)$ means that the cells in group G_3 correspond *exactly* to a list starting at index α_3. Relation $\mathtt{ready} = \alpha_3$ expresses that variable `ready` points to the head of the list.

As a comparison, the memory predicates in a reduced product of the array/shape abstractions would in the form of $(\mathcal{G}_0 * \mathcal{G}_1 * \mathcal{G}_2 * \mathcal{G}_3) \wedge (\mathbf{true} * \mathbf{lseg}(\alpha_1, -1) * \mathbf{lseg}(\alpha_2, -1) * \mathbf{slseg}(\alpha_3, -1))$, where the correspondance of $\mathcal{G}_1$ and $\mathbf{lseg}(\alpha_1, -1)$ is unclear.

6.3 Algorithms for Unfolding and Folding

Unfolding and folding are two basic operations in shape analysis [SRW99b, DOY06, CR08]. When the analysis needs to reason about an operation affecting a cell that is summarized as part of an inductive predicate, that predicate should first be *unfolded* according to its definition, which amounts to locally refining the inductive predicate. Since inductive predicates are based on disjunctions of inductive cases, the unfolding operation also returns disjunctions of abstract states. While unfolding decomposes inductive predicates, the analysis also needs a mechanism to re-construct such predicates. The goal of *folding*

$\overline{\textbf{unfold}}^{\overline{\mathtt{C}}}((\mathfrak{M} * \mathbf{i}^\bullet \,\&\, \mathbf{i}^\diamond, \mathfrak{N}), \mathbf{i}^\bullet \,\&\, \mathbf{i}^\diamond)$
$\overline{\mathtt{C}} = \emptyset$
foreach $\langle \overline{\mathtt{A}}^\bullet_{j,0} \,\&\, \overline{\mathtt{A}}^\diamond_{j,0} * \ldots * \overline{\mathtt{A}}^\bullet_{j,n_j} \,\&\, \overline{\mathtt{A}}^\diamond_{j,n_j}, \mathfrak{N}_j \rangle$
in the definition of $\mathbf{i}^\bullet \,\&\, \mathbf{i}^\diamond$))
$(., \mathfrak{N}^\bullet) = \overline{\textbf{unfold}}^\bullet(\mathbf{i}^\bullet, \mathfrak{N}, \mathbf{i}^\bullet := \langle \overline{\mathtt{A}}^\bullet_{j,0} * \ldots * \overline{\mathtt{A}}^\bullet_{j,n_j}, \mathfrak{N}_j \rangle)$
$(., \mathfrak{N}^\diamond) = \overline{\textbf{unfold}}^\diamond(\mathbf{i}^\diamond, \mathfrak{N}, \mathbf{i}^\diamond := \langle \overline{\mathtt{A}}^\diamond_{j,0} * \ldots * \overline{\mathtt{A}}^\diamond_{j,n_j}, \mathfrak{N}_j \rangle)$
$\mathfrak{M}_t = \mathfrak{M} * \overline{\mathtt{A}}^\bullet_{j,0} \,\&\, \overline{\mathtt{A}}^\diamond_{j,0} * \ldots * \overline{\mathtt{A}}^\bullet_{j,n_j} \,\&\, \overline{\mathtt{A}}^\diamond_{j,n_j}$
$\mathfrak{N}_t = \mathfrak{N}^\bullet_j \wedge \mathfrak{N}^\diamond_j$
$\overline{\mathtt{C}} = \overline{\mathtt{C}} \cup \{(\mathfrak{M}_t, \mathfrak{N}_t)\}$
return $\overline{\mathtt{C}}$

Figure 6.4 – Unfolding algorithm in coalescing domain

is to synthesize new inductive predicates according to the definition of such inductive predicates.

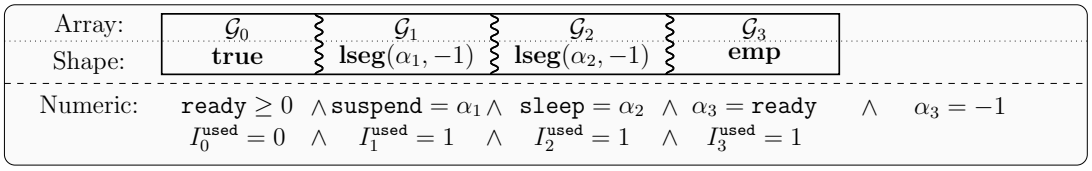
6.3.1 The Unfolding Algorithm in the Coalescing Domain

Unfolding refines the abstract state by rewriting an inductive predicate to the disjunctive cases of memory predicates in its definition. In our coalescing domain, it distributes the rewriting to the underlying domains.

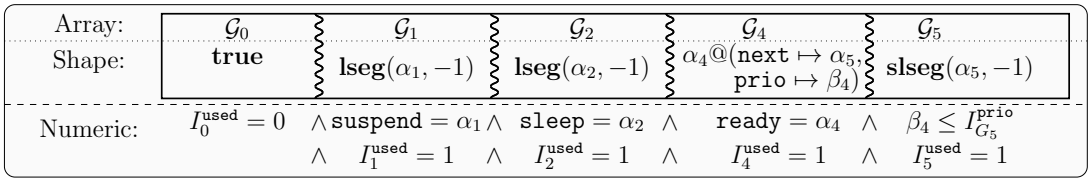
Definition 6.10 (Unfolding algorithm)**.** *The algorithm of unfolding* $\overline{\textbf{unfold}}^{\overline{\mathtt{C}}}$ *in coalescing domain is shown in Figure 6.4. It inputs an abstract state* $\overline{\mathtt{C}} = (\mathfrak{M} * \mathbf{i}^\bullet \,\&\, \mathbf{i}^\diamond, \mathfrak{N})$ *together with an inductive predicate* $\mathbf{i}^\bullet \,\&\, \mathbf{i}^\diamond$ *in that state, and returns a finite set of abstract states, obtained by unfolding this inductive predicate.*

Function $\overline{\textbf{unfold}}^{\overline{\mathtt{C}}}$ produces one disjunct per inductive case in the inductive definition of $\mathbf{i}^\bullet \,\&\, \mathbf{i}^\diamond$. Given an inductive case $\langle \overline{\mathtt{A}}^\bullet_{j,0} \,\&\, \overline{\mathtt{A}}^\diamond_{j,0} * \ldots * \overline{\mathtt{A}}^\bullet_{j,n_j} \,\&\, \overline{\mathtt{A}}^\diamond_{j,n_j}, \mathfrak{N}_j \rangle$, it calls the unfolding operator in the underlying domain ($\overline{\textbf{unfold}}^\bullet$ and $\overline{\textbf{unfold}}^\diamond$). Note that, in $\overline{\textbf{unfold}}^\bullet$ or $\overline{\textbf{unfold}}^\diamond$, only one inductive case is considered (e.g., $\mathbf{i}^\bullet := \langle \overline{\mathtt{A}}^\bullet_{j,0} * \ldots * \overline{\mathtt{A}}^\bullet_{j,n_j}, \mathfrak{N}_j \rangle$), and $(.,\mathfrak{N}^\bullet)$ means that only the numeric predicate component of the unfolding result in the underlying domain contributes to the final result. Note that, some numeric constraints in $\mathfrak{N}_j$ may not be supported (like $|G| = 1$ in the shape domain [CR08]) in the underlying domains. These unsupported numeric constraints are simply ignored, when they are encountered in the algorithms in the underlying domains.

Example 6.4 (Unfolding algorithm)**.** *Take* $\mathcal{G}_3 \,\&\, \mathbf{slseg}(\alpha_3, -1)$ *in Figure 6.3 for example. Suppose we have* $\mathtt{ready} \geq 0$ *in the numeric predicate, then Figure 6.5 shows the two disjuncts in the unfolding results. The first one (Figure 6.5(a)) comes from the empty disjunctive case* $\langle \mathcal{G} \,\&\, \mathbf{emp}, \pi = \tau \rangle$*. This state is actually unreachable since the original*



(a) The empty disjunctive case



(b) The non-empty disjunctive case

Figure 6.5 – The unfolding results

numeric predicates $\texttt{ready} \geq 0 \wedge \texttt{ready} = \alpha_3$ *conflict with the new numeric predicate* $\alpha_3 = -1$ *from the empty disjunctive case in unfolding. Therefore, the other one (Figure 6.5(b)) which splits the list into two parts (i.e., the head node and the rest of the list) is the only reachable state after unfolding.*

Theorem 6.1 (Soundness of unfolding algorithm)**.** *The application of the unfolding operator* $\overline{\textbf{unfold}}^{\overline{c}}$ *returns an over-approximation of its argument:*

$$\forall(\mathfrak{M}, \mathfrak{N}) \in \mathcal{C}, \forall \overline{\texttt{A}}_{\textbf{i}}^{\bullet} \,\&\, \overline{\texttt{A}}_{\textbf{i}}^{\diamond} \text{ in } \mathfrak{M},$$
$$\gamma^c(\mathfrak{M}, \mathfrak{N}) \subseteq \bigcup\{\gamma^c(\mathfrak{M}_t, \mathfrak{N}_t) \mid (\mathfrak{M}_t, \mathfrak{N}_t) \in \overline{\textbf{unfold}}^{\overline{c}}((\mathfrak{M}, \mathfrak{N}), \overline{\texttt{A}}_{\textbf{i}}^{\bullet} \,\&\, \overline{\texttt{A}}_{\textbf{i}}^{\diamond})\}$$

6.3.2 The Folding Algorithm in the Coalescing Domain

Folding is the reverse operation of unfolding. It synthesizes an inductive predicate according to its definition from a set of memory predicates in an abstract state. In our coalescing domain, the folding operator distributes synthesization to the underlying domains.

Definition 6.11 (Folding algorithm)**.** *Figure 6.6 shows the algorithm of folding* $\overline{\textbf{fold}}^{\overline{c}}$ *in our coalescing domain. It inputs an abstract state* $\overline{c} = (\mathfrak{M} * \overline{\texttt{A}}_{j,0}^{\bullet} \,\&\, \overline{\texttt{A}}_{j,0}^{\diamond} * \ldots * \overline{\texttt{A}}_{j,n_j}^{\bullet} \,\&\, \overline{\texttt{A}}_{j,n_j}^{\diamond}, \mathfrak{N})$ *together with an inductive definition* $\mathbf{i}^{\bullet} \,\&\, \mathbf{i}^{\diamond}$*, a separating conjunction of memory predicates* $\overline{\texttt{A}}_{j,0}^{\bullet} \,\&\, \overline{\texttt{A}}_{j,0}^{\diamond} * \ldots * \overline{\texttt{A}}_{j,n_j}^{\bullet} \,\&\, \overline{\texttt{A}}_{j,n_j}^{\diamond}$ *in that state, and returns an abstract state, obtained by folding these memory predicates.*

Operator $\overline{\textbf{fold}}^{\overline{c}}$ first looks for an inductive case of the parameter inductive definition that is satisfied by the parameter memory predicates and numeric conditions. If there exists one such case, it calls the folding operator in the underlying domain ($\overline{\textbf{fold}}^{\bullet}$ and $\overline{\textbf{fold}}^{\diamond}$). Similar as in $\overline{\textbf{unfold}}^{\overline{c}}$, the input state of $\overline{\textbf{fold}}^{\bullet}$ is limited to $\overline{\texttt{A}}_{j,0}^{\bullet} * \ldots * \overline{\texttt{A}}_{j,n_j}^{\bullet}$ and only one case is considered in the inductive definition (i.e., $\mathbf{i}^{\bullet} := \langle \overline{\texttt{A}}_{j,0}^{\bullet} * \ldots * \overline{\texttt{A}}_{j,n_j}^{\bullet}, \mathfrak{N}_j \rangle$).

$\overline{\textbf{fold}}^{\overline{c}}((\mathfrak{M} * \overline{\mathtt{A}}^{\bullet}_{j,0}\ \&\ \overline{\mathtt{A}}^{\diamond}_{j,0} * \ldots * \overline{\mathtt{A}}^{\bullet}_{j,n_j}\ \&\ \overline{\mathtt{A}}^{\diamond}_{j,n_j}, \mathfrak{N}), \mathbf{i}^{\bullet}\ \&\ \mathbf{i}^{\diamond}, \overline{\mathtt{A}}^{\bullet}_{j,0}\ \&\ \overline{\mathtt{A}}^{\diamond}_{j,0} * \ldots * \overline{\mathtt{A}}^{\bullet}_{j,n_j}\ \&\ \overline{\mathtt{A}}^{\diamond}_{j,n_j})$
if$((\exists\langle\overline{\mathtt{A}}^{\bullet}_{j,0}\ \&\ \overline{\mathtt{A}}^{\diamond}_{j,0} * \ldots * \overline{\mathtt{A}}^{\bullet}_{j,n_j}\ \&\ \overline{\mathtt{A}}^{\diamond}_{j,n_j}, \mathfrak{N}_j\rangle$ in the definition of $(\mathbf{i}^{\bullet}\ \&\ \mathbf{i}^{\diamond})$ and $\overline{\textbf{isle}}_{\mathfrak{N}}(\mathfrak{N}, \mathfrak{N}_j))$
$(., \mathfrak{N}^{\bullet}) = \overline{\textbf{fold}}^{\bullet}(\overline{\mathtt{A}}^{\bullet}_{j,0} * \ldots * \overline{\mathtt{A}}^{\bullet}_{j,n_j}, \mathfrak{N}, \mathbf{i}^{\bullet} := \langle\overline{\mathtt{A}}^{\bullet}_{j,0} * \ldots * \overline{\mathtt{A}}^{\bullet}_{j,n_j}, \mathfrak{N}_j\rangle)$
$(., \mathfrak{N}^{\diamond}) = \overline{\textbf{fold}}^{\diamond}(\overline{\mathtt{A}}^{\diamond}_{j,0} * \ldots * \overline{\mathtt{A}}^{\diamond}_{j,n_j}, \mathfrak{N}, \mathbf{i}^{\diamond} := \langle\overline{\mathtt{A}}^{\diamond}_{j,0} * \ldots * \overline{\mathtt{A}}^{\diamond}_{j,n_j}, \mathfrak{N}_j\rangle)$
$\mathfrak{M} = \mathfrak{M} * \mathbf{i}^{\bullet}\ \&\ \mathbf{i}^{\diamond}$
$\mathfrak{N} = \mathfrak{N}^{\bullet} \wedge \mathfrak{N}^{\diamond}$
return $(\mathfrak{M}, \mathfrak{N})$
else return $(\mathfrak{M} * \overline{\mathtt{A}}^{\bullet}_{j,0}\ \&\ \overline{\mathtt{A}}^{\diamond}_{j,0} * \ldots * \overline{\mathtt{A}}^{\bullet}_{j,n_j}\ \&\ \overline{\mathtt{A}}^{\diamond}_{j,n_j}, \mathfrak{N})$

Figure 6.6 – Folding algorithm in coalescing domain

Example 6.5 (Folding algorithm)**.** *Consider the abstract state shown below.*

Array:	$\mathcal{G}_0$	$\mathcal{G}_1$	$\mathcal{G}_2$
Shape:	**true**	α_1@next $\mapsto \beta_1$	**lseg**$(\alpha_2, -1)$
Numeric:		$\alpha_1 \geq 0$	$\wedge\ \ \alpha_2 = \beta_1$ $\wedge\ 0 \leq G_2 \leq 10$

Since the memory predicates $G_1\ \&\ \alpha_1$@next $\mapsto \beta_1 * G_2\ \&$ **lseg**$(\alpha_2, -1)$ *correspond to the non-empty case in the definition of* $\mathcal{G}\ \&$ **lseg***, Let* $\overline{\textbf{fold}}^{a/s}$ *be the folding operator as defined in Definition 6.11. If we apply* $\overline{\textbf{fold}}^{a/s}((\mathfrak{M}^{a/s}, \mathfrak{N}^{a/s}), \mathcal{G}\ \&$ **lseg**$, \mathcal{G}_1\ \&\ (\alpha_1$@next $\mapsto \beta_1) * \mathcal{G}_2\ \&$ **lseg**$(\alpha_2, -1))$*, we would get the result as follows.*

Array:	$\mathcal{G}_0$	$\mathcal{G}_1$
Shape:	**true**	α_1@next $\mapsto \beta_1$
Numeric:		$\alpha_1 \geq 0$

One special memory predicate in our array/shape coalescing domain is $\mathcal{G}\ \&$ **true***. Basically, any separating conjunction of memory predicates can be seen as its inductive case. For instance, if we apply* $\overline{\textbf{fold}}^{a/s}((\mathfrak{M}^{a/s}, \mathfrak{N}^{a/s}), \mathcal{G}\ \&$ **true**$, \mathcal{G}_0\ \&$ **true** $* \mathcal{G}_3\ \&$ **lseg**$(\alpha_1, -1))$ *on the abstract state above, we get the abstract state as follows.*

Array:	$\mathcal{G}_4$
Shape:	**true**
Numeric:	$G_4 \geq 0$

The folding algorithm is sound in the sense that they return an over-approximation of the abstract states that it applies on.

Theorem 6.2 (Soundness of folding algorithm)**.** *Let* $(\mathfrak{M} * \overline{\mathtt{A}}^{\bullet}_{j,0} \,\&\, \overline{\mathtt{A}}^{\diamond}_{j,0} * \ldots * \overline{\mathtt{A}}^{\bullet}_{j,n_j} \,\&\, \overline{\mathtt{A}}^{\diamond}_{j,n_j}, \mathfrak{N})$ *and* $\mathbf{i}^{\bullet} \,\&\, \mathbf{i}^{\diamond}$ *be the input abstract state and the inductive definition respectively,* $\overline{\mathbf{fold}}^{\overline{\mathsf{c}}}$ *is sound in the following sense.*

$$\gamma^c((\mathfrak{M} * \overline{\mathtt{A}}^{\bullet}_{j,0} \,\&\, \overline{\mathtt{A}}^{\diamond}_{j,0} * \ldots * \overline{\mathtt{A}}^{\bullet}_{j,n_j} \,\&\, \overline{\mathtt{A}}^{\diamond}_{j,n_j}, \mathfrak{N})) \subseteq$$
$$\overline{\mathbf{fold}}^{\overline{\mathsf{c}}}((\mathfrak{M} * \overline{\mathtt{A}}^{\bullet}_{j,0} \,\&\, \overline{\mathtt{A}}^{\diamond}_{j,0} * \ldots * \overline{\mathtt{A}}^{\bullet}_{j,n_j} \,\&\, \overline{\mathtt{A}}^{\diamond}_{j,n_j}, \mathfrak{N}), \mathbf{i}^{\bullet} \,\&\, \mathbf{i}^{\diamond}, \overline{\mathtt{A}}^{\diamond}_{j,0} * \ldots * \overline{\mathtt{A}}^{\bullet}_{j,n_j} \,\&\, \overline{\mathtt{A}}^{\diamond}_{j,n_j})$$

6.4 Transfer Functions

In this section, we study the abstract transfer functions for tests and assignments.

6.4.1 Condition Tests

Recall that the concrete semantics of a condition test $\mathbf{guard}[\![\mathsf{r}]\!] : \mathcal{P}(\mathbb{S}) \to \mathcal{P}(\mathbb{S})$ filters out the concrete states that do not evaluate r to true. In the abstract level, the transfer function for condition tests adds constraints on numeric predicates $\mathfrak{N}$. However, when some memory locations that are accessed in the statements are summarized in the memory predicates $\mathfrak{M}$, these memory locations should be resolved first. After that, we can apply transfer functions in the underlying domains to add numeric constraints. Since the numeric predicates in the coalescing domain is a conjunction of those in the underlying domains ($\mathfrak{N} = \overline{\mathtt{N}}^{\bullet} \wedge \overline{\mathtt{N}}^{\diamond}$), a reduction is performed on $\overline{\mathtt{N}}^{\bullet} \wedge \overline{\mathtt{N}}^{\diamond}$ at last.

Resolving. In this step, our analysis looks for summarized memory locations in the input statement r and resolves them by calling $\overline{\mathbf{unfold}}^{\overline{\mathsf{c}}}$. If there is no such location (e.g., in condition tests on program scalar variables), it does nothing. This step is described by operator $\overline{\mathbf{resolve}}[\mathsf{r}] : \mathcal{C} \to \mathcal{P}(\mathcal{C})$ (note that it produces a finite disjunction of abstract states since it calls $\overline{\mathbf{unfold}}^{\overline{\mathsf{c}}}$).

Theorem 6.3 (Soundness of the resolving operator)**.** *For any right-value expression* r*, operator* $\overline{\mathbf{resolve}}$ *is sound in the sense that,*

$$\gamma^c(\overline{\mathsf{c}}) \subseteq \bigcup \gamma^c(\overline{\mathbf{resolve}}[\mathsf{r}](\overline{\mathsf{c}})))$$

Transfer function for condition tests in the underlying domains. The parameters of the transfer functions in the underlying domains for condition tests (i.e., $\overline{\mathbf{guard}}^{\bullet}[.] : \overline{D}^{\bullet} \to \overline{D}^{\bullet}$ and $\overline{\mathbf{guard}}^{\diamond}[.] : \overline{D}^{\diamond} \to \overline{D}^{\diamond}$) are abstract states in the underlying domains (i.e., $(\overline{\mathtt{M}}^{\bullet}, \overline{\mathtt{N}}^{\bullet})$ and $(\overline{\mathtt{M}}^{\diamond}, \overline{\mathtt{N}}^{\diamond})$). To apply the underlying transfer functions, our analysis needs to decompose $(\mathfrak{M}, \mathfrak{N})$ into underlying abstract states, so as to fit the parameters of $\overline{\mathbf{guard}}^{\bullet}[.]$

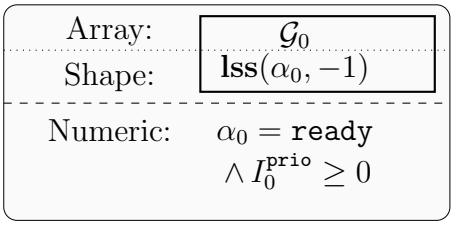


Figure 6.7 – The abstract state before guard

and $\overline{\mathbf{guard}}^{\diamond}[.]$. The numeric predicates in the coalescing domain $\mathfrak{N} = \overline{\mathtt{N}}^{\bullet} \wedge \overline{\mathtt{N}}^{\diamond}$ are trivial to decompose (i.e., split $\mathfrak{N}$ into two parts: $\overline{\mathtt{N}}^{\bullet}$ and $\overline{\mathtt{N}}^{\diamond}$). For the memory predicates, the decomposition is accomplished by two operators $\overline{\mathbf{decom}}^{\bullet} : \overline{\mathcal{D}}_{\mathfrak{M}} \rightarrow \overline{\mathcal{D}}^{\bullet}_{\overline{\mathtt{M}}}$ and $\overline{\mathbf{decom}}^{\diamond} : \overline{\mathcal{D}}_{\mathfrak{M}} \rightarrow \overline{\mathcal{D}}^{\diamond}_{\overline{\mathtt{M}}}$.

Definition 6.12 (The algorithm of the decomposition operators)**.** *The algorithm of* $\overline{\mathbf{decom}}^{\bullet}$ *(*$\overline{\mathbf{decom}}^{\diamond}$ *follows the same principles) can be defined recursively:*

$$\begin{array}{lcl} \overline{\mathbf{decom}}^{\bullet}(\overline{\mathtt{A}}^{\bullet} \,\&\, \overline{\mathtt{A}}^{\diamond}) & = & \overline{\mathtt{A}}^{\bullet} \\ \overline{\mathbf{decom}}^{\bullet}(\mathfrak{M} * \overline{\mathtt{A}}^{\bullet} \,\&\, \overline{\mathtt{A}}^{\diamond}) & = & \overline{\mathbf{decom}}^{\bullet}(\mathfrak{M}) * \overline{\mathtt{A}}^{\bullet} \end{array}$$

Note that, since guard function does not modify memory predicates, the resulting memory predicates $\overline{\mathtt{M}}^{\bullet}$ and $\overline{\mathtt{M}}^{\diamond}$ of $\overline{\mathbf{decom}}^{\bullet}$ and $\overline{\mathbf{decom}}^{\diamond}$ are only used as the parameters of underlying guard functions and do not constitute the final result.

Reduction. Reduction is performed after the guard function of underlying domains. It refines $\mathfrak{N} = \overline{\mathtt{N}}^{\bullet} \wedge \overline{\mathtt{N}}^{\diamond}$ by propagating constraints between the two components. Note that, its algorithm depends on the form of constraints in $\overline{\mathtt{N}}^{\bullet}$ and $\overline{\mathtt{N}}^{\diamond}$. Thus for different instances of our coalescing domain, we have to design specific reduction operation. This step is formalized as operator $\overline{\mathbf{reduce}} : \mathfrak{N} \rightarrow \mathfrak{N}$. In the array/shape coalescing domain, both $\overline{\mathtt{N}}^{\bullet}$ and $\overline{\mathtt{N}}^{\diamond}$ are instances from Maya+ domain. Thus $\overline{\mathbf{reduce}}$ is provided by the Maya+ domain.

Theorem 6.4 (Soundness of the reduction operator)**.** *Operator* $\overline{\mathbf{reduce}}$ *is sound in the sense that,*

$$\gamma^c(\mathfrak{M}, \mathfrak{N}) = \gamma^c(\mathfrak{M}, (\overline{\mathbf{reduce}}(\mathfrak{N})))$$

Definition 6.13 (The transfer function for condition tests)**.** *The transfer function for condition test* $\overline{\mathbf{guard}}^{\mathtt{c}}[.]$ *is formalized in Figure 6.10(a).*

Example 6.6 (The transfer function for condition tests)**.** *Given an abstract state in the array/shape coalescing domain in Figure 6.7 as the pre-condition of the condition test* `a[ready].prio` ≥ 1.

This pre-condition describes an array with all cells linked by a sorted list with respect to the value in field `prio` *in an increasing order. In the condition test, the accessed array*

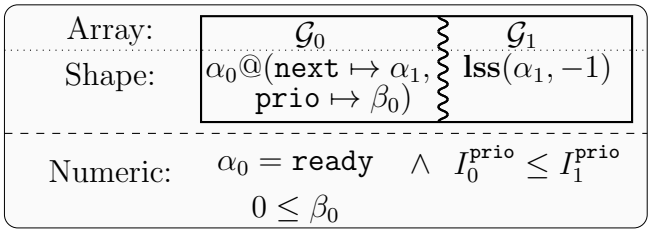


Figure 6.8 – The abstract state after resolving

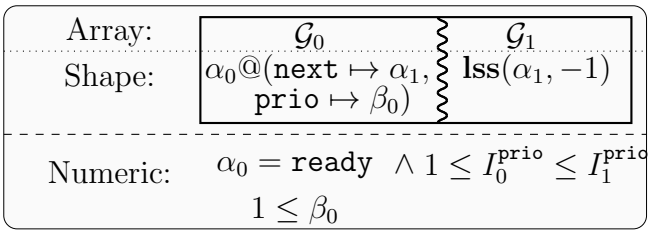


Figure 6.9 – The abstract state after guard

cell a[ready].prio *is summarized in the memory predicate* $\mathcal{G}_0$ & $\mathbf{lss}(\alpha, -1)$. *Thus* $\overline{\mathbf{resolve}}$ *calls* $\overline{\mathbf{unfold}}^{\overline{c}}$ *to materialize this array cell, and produces the abstract state in Figure 6.8.*

Then two transfer functions in the underlying domains $\overline{\mathbf{guard}}^{a}[I_0^{\texttt{prio}} \geq 1]$ *and* $\overline{\mathbf{guard}}^{s}[\beta_1 \geq 1]$ *are applied, which results in abstract state in Figure 6.9.*

Theorem 6.5 (Soundness of the transfer function for condition tests)**.** *The transfer function for condition tests* $\overline{\mathbf{guard}}^{\overline{c}}[.]$ *is sound in the following sense.*

$$\mathbf{guard}[\![\mathsf{r}]\!]\gamma^c(\overline{\mathsf{c}}) \subseteq \bigcup \gamma^c(\overline{\mathbf{guard}}^{\overline{c}}[\mathsf{r}](\overline{\mathsf{c}})))$$

Proof. The soundness of $\overline{\mathbf{guard}}^{\overline{c}}[.]$ follows the soundness of the resolving operation $\overline{\mathbf{resolve}}$, underlying transfer functions, and the reduction operation. □

6.4.2 Assignments

In the concrete semantics, an assignment $\mathbf{stat}[\![\mathsf{l} = \mathsf{r}]\!] : \mathcal{P}(\mathbb{S}) \to \mathcal{P}(\mathbb{S})$ updates the value stored in left-value expression l with evaluation of the right-value expression r. In the abstract level, the transfer function for assignments updates the numeric predicates and memory predicates (when the left-value expression l is not a program scalar variable). Same with the transfer function for condition tests, the abstract transfer function for assignments first resolves all the memory locations in the statement (i.e., both l and r), and applies transfer functions for assignments in the underlying domains. Finally it performs reduction on the resulting numeric predicates.

$\overline{\textbf{guard}}^{\overline{\mathtt{C}}}[\mathsf{r}, (\mathfrak{M}, \mathfrak{N})]$
  $\overline{\mathtt{C}} = \emptyset$
  foreach$((\mathfrak{M}, \mathfrak{N}) \in \overline{\textbf{resolve}}[\mathsf{r}](\mathfrak{M}, \mathfrak{N}))$
    $\overline{\mathtt{M}}^{\bullet} = \overline{\textbf{decom}}(\mathfrak{M})$
    $\overline{\mathtt{M}}^{\diamond} = \overline{\textbf{decom}}(\mathfrak{M})$
    $(., \overline{\mathtt{N}}^{\bullet}) = \overline{\textbf{guard}}^{\bullet}[\mathsf{r}, (\overline{\mathtt{M}}^{\bullet}, \overline{\mathtt{N}}^{\bullet})]$
    $(., \overline{\mathtt{N}}^{\diamond}) = \overline{\textbf{guard}}^{\diamond}[\mathsf{r}, (\overline{\mathtt{M}}^{\diamond}, \overline{\mathtt{N}}^{\diamond})]$
    $\mathfrak{N}_t = \overline{\textbf{reduce}}(\overline{\mathtt{N}}^{\bullet} \wedge \overline{\mathtt{N}}^{\diamond})$
    $\overline{\mathtt{C}} = \overline{\mathtt{C}} \cup \{(\mathfrak{M}, \mathfrak{N}_t)\}$
  return $\overline{\mathtt{C}}$

(a) The algorithm of guard

$\overline{\textbf{assign}}^{\overline{\mathtt{C}}}[\mathsf{l} = \mathsf{r}, (\mathfrak{M}, \mathfrak{N})]$
  $\overline{\mathtt{C}} = \emptyset$
  foreach$((\mathfrak{M}, \mathfrak{N}) \in \overline{\textbf{resolve}}[\mathsf{l} = \mathsf{r}](\mathfrak{M}, \mathfrak{N}))$
    $\overline{\mathtt{M}}^{\bullet} = \overline{\textbf{decom}}(\mathfrak{M})$
    $\overline{\mathtt{M}}^{\diamond} = \overline{\textbf{decom}}(\mathfrak{M})$
    $(\overline{\mathtt{M}}^{\bullet}, \overline{\mathtt{N}}^{\bullet}) = \overline{\textbf{assign}}^{\bullet}[\mathsf{l} = \mathsf{r}, (\overline{\mathtt{M}}^{\bullet}, \overline{\mathtt{N}}^{\bullet})]$
    $(\overline{\mathtt{M}}^{\diamond}, \overline{\mathtt{N}}^{\diamond}) = \overline{\textbf{assign}}^{\diamond}[\mathsf{l} = \mathsf{r}, (\overline{\mathtt{M}}^{\diamond}, \overline{\mathtt{N}}^{\diamond})]$
    $\mathfrak{M}_t = \overline{\textbf{recons}}(\overline{\mathtt{M}}^{\bullet}, \overline{\mathtt{M}}^{\diamond})$
    $\mathfrak{N}_t = \overline{\textbf{reduce}}(\mathfrak{N})$
    $\overline{\mathtt{C}} = \overline{\mathtt{C}} \cup \{(\mathfrak{M}_t, \mathfrak{N}_t)\}$
  return $\overline{\mathtt{C}}$

(b) The algorithm of assignment

Figure 6.10 – Transfer functions in the coalesced domain

The first and third steps of the transfer function for assignments are the same with that for condition tests. However, in the second step, memory predicates are possibly modified in the underlying transfer functions for assignments. Thus, after applying $\overline{\textbf{assign}}^{\bullet}[.]$ and $\overline{\textbf{assign}}^{\diamond}[.]$, our analysis needs to collect the resulting memory predicates in the underlying domains and re-construct the memory predicates in the coalescing domain. This process is formalized as operator $\overline{\textbf{recons}} : \overline{\mathcal{D}}^{\bullet}_{\overline{\mathtt{M}}} \times \overline{\mathcal{D}}^{\diamond}_{\overline{\mathtt{M}}} \to \overline{\mathcal{D}}_{\mathfrak{M}}$. It is the reverse operation of $\overline{\textbf{decom}}$ and coalesces the atomic memory predicates from in $\overline{\mathcal{D}}^{\bullet}_{\overline{\mathtt{M}}}$ and $\overline{\mathcal{D}}^{\diamond}_{\overline{\mathtt{M}}}$ with non-separating conjunction.

Definition 6.14 (The transfer function for assignments)**.** *The algorithm* $\overline{\textbf{assign}}^{\overline{\mathtt{C}}}[.]$ *in our coalescing domain is formalized in Figure 6.10(b).*

Example 6.7 (The transfer function for assignments)**.** *Given an abstract state in the array/shape coalescing domain in Figure 6.11 as the pre-condition of the assignment* $\mathtt{a}[\mathtt{ready}] = 0$*. Our transfer function first resolves* $\mathtt{a}[\mathtt{ready}]$ *and decomposes the resulting*

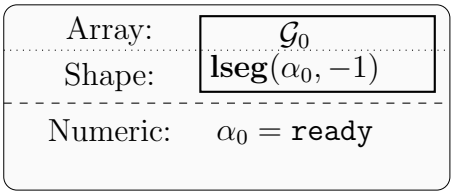
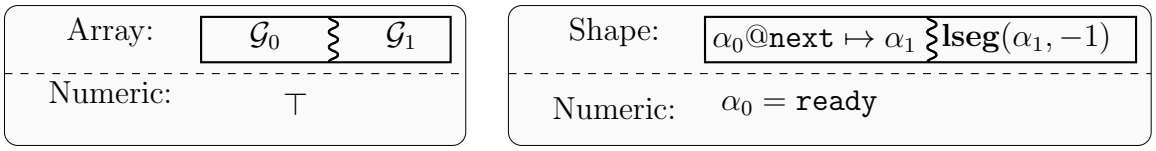


Figure 6.11 – The abstract state before the assignment



(a) The decomposed state in array domain (b) The decomposed state in shape domain

Figure 6.12 – The abstract state after resolving and decomposition

abstract state. The two abstract states in underlying domains are shown in Figure 6.12. Then the two states are updated by the transfer functions in the underlying domains. Note that in $\overline{\mathbf{assign}}^{s}[\mathsf{a}[\texttt{ready}] = 0]$*, memory predicate* $\alpha_0@\texttt{next} \mapsto \alpha_1$ *is updated to* $\alpha_0@\texttt{next} \mapsto \beta_0$ *and* $\beta_0 = 0$*. The updated abstract states in the underlying domains are re-constructed, as shown in Figure 6.13. Note that the correspondence of the memory predicates between abstract states in the underlying domains are recorded in the process of decomposition.*

Theorem 6.6 (Soundness of the transfer function for assignments)**.** *The transfer function for assignments is sound in the following sense.*

$$\mathbf{stat}[\![\mathsf{l} = \mathsf{r}]\!]\gamma^c(\overline{\mathsf{c}}) \subseteq \bigcup \gamma^c(\overline{\mathbf{assign}}^{\overline{\mathsf{c}}}[\mathsf{l} = \mathsf{r}](\overline{\mathsf{c}}))$$

Proof. The soundness of $\overline{\mathbf{assign}}^{\overline{\mathsf{c}}}[.]$ follows the soundness of the resolving operation $\overline{\mathbf{resolve}}$, underlying transfer functions, and the reduction operation. □

6.5 Lattice Operators

In the framework of abstract interpretation, the union of sets of concrete states is over-approximated by abstract join or widening [CC77] (which guarantees the termination of abstract iterations), and the order of sets of concrete states is over-approximated by abstract inclusion checking. In our coalescing domain, we first define these operators on *compatible* abstract states (i.e., they share the same memory predicates). Afterwards, we show how to repartition general abstract states to generate compatible abstract states.

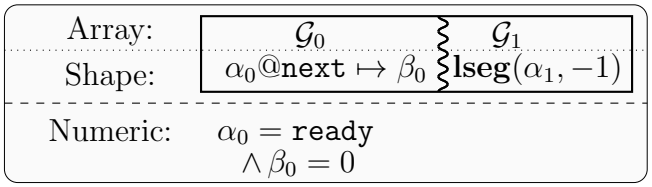


Figure 6.13 – The abstract state after the assignment

6.5.1 Lattice Operators over Compatible Abstract States

Definition 6.15 (Compatible abstract states)**.** *Given two abstract states* $(\mathfrak{M}_0, \mathfrak{N}_0)$ *and* $(\mathfrak{M}_1, \mathfrak{N}_1)$*, they are compatible, if and only if* $\mathfrak{M}_0 = \mathfrak{M}_1$*.*

Since the memory predicates are the same in compatible abstract states, the lattice operators take effect on numeric constraints.

Definition 6.16 (Lattice operators over compatible abstract states)**.** *We assume* $\overline{\mathbf{widen}}^{\mathfrak{N}}, \overline{\mathbf{join}}^{\mathfrak{N}}$ *and* $\overline{\mathbf{incl}}^{\mathfrak{N}}$ *are the widening, join and inclusion checking operators over the numeric predicates respectively, then we can define the lattice operators over compatible abstract states below.*

$$\begin{aligned}
\overline{\mathbf{widen}}^{\overline{\mathsf{c}}}_{\equiv}((\mathfrak{M}, \mathfrak{N}_0), (\mathfrak{M}, \mathfrak{N}_1)) &= (\mathfrak{M}, \overline{\mathbf{widen}}^{\mathfrak{N}}(\mathfrak{N}_0, \mathfrak{N}_1)) \\
\overline{\mathbf{join}}^{\overline{\mathsf{c}}}_{\equiv}((\mathfrak{M}, \mathfrak{N}_0), (\mathfrak{M}, \mathfrak{N}_1)) &= (\mathfrak{M}, \overline{\mathbf{join}}^{\mathfrak{N}}(\mathfrak{N}_0, \mathfrak{N}_1)) \\
\overline{\mathbf{incl}}^{\overline{\mathsf{c}}}_{\equiv}((\mathfrak{M}, \mathfrak{N}_0), (\mathfrak{M}, \mathfrak{N}_1)) &= (\mathfrak{M}, \overline{\mathbf{incl}}^{\mathfrak{N}}(\mathfrak{N}_0, \mathfrak{N}_1))
\end{aligned}$$

Theorem 6.7 (Soundness of lattice operators)**.** *Operators* $\overline{\mathbf{widen}}^{\overline{\mathsf{c}}}_{\equiv}$*,* $\overline{\mathbf{join}}^{\overline{\mathsf{c}}}_{\equiv}$ *and* $\overline{\mathbf{incl}}^{\overline{\mathsf{c}}}_{\equiv}$ *are sound in the following sense.*

$$\begin{aligned}
\gamma^c(\overline{\mathsf{c}}_0) \cup \gamma^c(\overline{\mathsf{c}}_1) &\subseteq \gamma^c(\overline{\mathbf{widen}}^{\overline{\mathsf{c}}}_{\equiv}(\overline{\mathsf{c}}_0, \overline{\mathsf{c}}_1)) \\
\gamma^c(\overline{\mathsf{c}}_0) \cup \gamma^c(\overline{\mathsf{c}}_1) &\subseteq \gamma^c(\overline{\mathbf{join}}^{\overline{\mathsf{c}}}_{\equiv}(\overline{\mathsf{c}}_0, \overline{\mathsf{c}}_1)) \\
\overline{\mathbf{incl}}^{\overline{\mathsf{c}}}_{\equiv}(\overline{\mathsf{a}}_0, \overline{\mathsf{a}}_1) = \boldsymbol{true} &\Longrightarrow \gamma^c(\overline{\mathsf{a}}_0) \subseteq \gamma^c(\overline{\mathsf{a}}_1)
\end{aligned}$$

Proof. The soundness of operators $\overline{\mathbf{widen}}^{\overline{\mathsf{c}}}_{\equiv}$, $\overline{\mathbf{join}}^{\overline{\mathsf{c}}}_{\equiv}$ and $\overline{\mathbf{incl}}^{\overline{\mathsf{c}}}_{\equiv}$ follow the soundness of $\overline{\mathbf{widen}}^{\mathfrak{N}}, \overline{\mathbf{join}}^{\mathfrak{N}}$ and $\overline{\mathbf{incl}}^{\mathfrak{N}}$ respectively. □

6.5.2 Processing on Non Compatible Abstract States

In most cases, the two input abstract states are not compatible, which means, they have different numbers of atomic memory predicates, or the correspondence between their atomic memory predicates is not obvious. Thus to make them compatible, we need to accomplish two tasks: (1) pairing the atomic memory predicate from two abstract states by their similarity; (2) folding the memory predicates in two abstract states when the

numbers of atomic memory predicates are different. The two tasks are performed by operator $\overline{\textbf{repartition}}$.

Ranking. Pairing is achieved with the help of a ranking function $\overline{\textbf{rank}}^{\overline{\mathtt{c}}} : \overline{\mathcal{D}}_{\mathfrak{M}} \times \overline{\mathcal{D}}_{\mathfrak{M}} \to \mathbb{N}$, which computes a logical distance between atomic memory predicates in different abstract states. A high value of $\overline{\textbf{rank}}^{\overline{\mathtt{c}}}(\overline{\mathtt{A}}_i, \overline{\mathtt{A}}_j)$ indicates $\overline{\mathtt{A}}_i$ of $\overline{\mathtt{c}}_0$ and $\overline{\mathtt{A}}_j$ of $\overline{\mathtt{c}}_1$ are likely to describe sets of cells with similar properties.

The value of $\overline{\textbf{rank}}^{\overline{\mathtt{c}}}(\overline{\mathtt{A}}_i, \overline{\mathtt{A}}_j)$ is determined by two factors:

- whether the two atomic memory predicates are of the same type of inductive predicates, and whether the parameters in the memory predicates correspond to the same program variables;
- if two atomic memory predicates are not inductive predicates, whether the value ranges of the addresses and contents are similar.

Re-partitioning. The lattice operators do not use exactly the same re-partitioning algorithm. Thus our analysis provides three versions of re-partitioning (i.e., $\overline{\textbf{repartition}}_{\overline{\textbf{widen}}^{\overline{\mathtt{c}}}}$, $\overline{\textbf{repartition}}_{\overline{\textbf{join}}^{\overline{\mathtt{c}}}}$ and $\overline{\textbf{repartition}}_{\overline{\textbf{incl}}^{\overline{\mathtt{c}}}}$) for $\overline{\textbf{widen}}^{\overline{\mathtt{c}}}$ (i.e., widening) , $\overline{\textbf{join}}^{\overline{\mathtt{c}}}$ (i.e., join) and $\overline{\textbf{incl}}^{\overline{\mathtt{c}}}$ (i.e., inclusion cheking) respectively. In this chapter, we only give the details of $\overline{\textbf{repartition}}_{\overline{\textbf{widen}}^{\overline{\mathtt{c}}}}$, since other versions follow very similar principles.

Operator $\overline{\textbf{repartition}}_{\overline{\textbf{widen}}^{\overline{\mathtt{c}}}}(\overline{\mathtt{c}}_0, \overline{\mathtt{c}}_1)$ first computes a *pairing* $\leftrightarrow \in \mathcal{P}(\overline{\mathcal{D}}_{\mathfrak{M}} \times \overline{\mathcal{D}}_{\mathfrak{M}})$, which is a set of relations between memory predicates of $\overline{\mathtt{c}}_0$ and $\overline{\mathtt{c}}_1$. The pairing is defined by the rules below:

1. operator $\overline{\textbf{repartition}}$ pairs each atomic memory predicate with the atomic memory predicate with which it has the highest ranking value;
2. if three relations of the form $\overline{\mathtt{A}}_i \leftrightarrow \overline{\mathtt{A}}_k$, $\overline{\mathtt{A}}_i \leftrightarrow \overline{\mathtt{A}}_j$ and $\overline{\mathtt{A}}_t \leftrightarrow \overline{\mathtt{A}}_j$ have been added to the pairing, then the "middle" relation $\overline{\mathtt{A}}_i \leftrightarrow \overline{\mathtt{A}}_j$ 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, operator $\overline{\textbf{repartition}}_{\overline{\textbf{widen}}^{\overline{\mathtt{c}}}}$ applies a *partition transforming* which transforms both arguments into "compatible" abstract states using the following (symmetric) principles:

- if $\overline{\mathtt{A}}_i \leftrightarrow \overline{\mathtt{A}}_j$ and $\overline{\mathtt{A}}_i \leftrightarrow \overline{\mathtt{A}}_k$, then $\overline{\mathtt{A}}_j$ and $\overline{\mathtt{A}}_k$ are merged by operator $\overline{\textbf{fold}}^{\overline{\mathtt{c}}}$ and the resulting memory predicate is paired with $\overline{\mathtt{A}}_i$;
- if $\overline{\mathtt{A}}_i$ is mapped only to $\overline{\mathtt{A}}_j$, $\overline{\mathtt{A}}_j$ is paired only to $\overline{\mathtt{A}}_i$, then their parameters are renamed to the same;

$$
\begin{array}{l}
\overline{\textbf{widen}}^{\overline{c}}(\overline{c}_0, \overline{c}_1)\{ \\
\quad \textbf{foreach}(\overline{A}_i \textbf{ in } \overline{c}_0) \\
\quad\quad \textbf{foreach}(\overline{A}_j \textbf{ in } \overline{c}_1) \\
\quad\quad\quad W_{ij} = \overline{\textbf{rank}}^{\overline{c}}(\overline{A}_i, \overline{A}_j); \\
\quad \overline{c}_0, \overline{c}_1 = \overline{\textbf{repartition}}_{\overline{\textbf{widen}}^{\overline{c}}}(W, \overline{c}_0, \overline{c}_1); \\
\quad \textbf{return } \overline{\textbf{widen}}^{\overline{c}}_{\equiv}(\overline{c}_0, \overline{c}_1); \\
\}
\end{array}
$$

Figure 6.14 – The algorithm of the widening operator

Widening. With operators $\overline{\textbf{rank}}^{\overline{c}}$ and $\overline{\textbf{repartition}}_{\overline{\textbf{widen}}^{\overline{c}}}$, we can define the widening operator in the coalescing domain as follows.

Definition 6.17 (The algorithm of widening)**.** *The algorithm of widening is formalized in Figure 6.14.*

Theorem 6.8 (Soundness and termination of the widening operator)**.** *Operator* $\overline{\textbf{widen}}^{\overline{c}}$ *is sound (for all abstract states* $\overline{c}_0, \overline{c}_1$*, the inclusion* $\gamma^c(\overline{c}_0) \cup \gamma^c(\overline{c}_1) \subseteq \gamma^c(\overline{\textbf{widen}}^{\overline{c}}(\overline{c}_0, \overline{c}_1))$ *holds) and ensures termination of abstract iterates.*

Proof. The soundness of $\overline{\textbf{widen}}^{\overline{c}}$ follows from the soundness of $\overline{\textbf{fold}}^{\overline{c}}$ and $\overline{\textbf{widen}}^{\mathfrak{N}}$. Since $\overline{\textbf{widen}}^{\overline{c}}$ changes the memory predicates only by applying foldings on non-empty disjunctive case, thus the memory predicate should stabilize after finitely many iterations. Therefore, since $\overline{\textbf{widen}}^{\overline{c}}$ applies $\overline{\textbf{widen}}^{\mathfrak{N}}$ on the numeric predicates component, it ensures the termination of any sequence of abstract iterates. □

Join and inclusion checking. A join operator $\overline{\textbf{join}}^{\overline{c}}$ (to over-approximate concrete unions) and an inclusion check operator $\overline{\textbf{isle}}^{\overline{c}}$ (to conservatively decide inclusion) can be defined in a very similar manner. The only difference lies in the re-partitioning operators: $\overline{\textbf{repartition}}_{\overline{\textbf{join}}^{\overline{c}}}$ may apply $\overline{\textbf{fold}}^{\overline{c}}$ on the empty case when one atomic memory predicate in an abstract state is paired with no atomic memory predicate in the other state and $\overline{\textbf{repartition}}_{\overline{\textbf{incl}}^{\overline{c}}}$ only applies $\overline{\textbf{fold}}^{\overline{c}}$ on the left-hand operand.

Definition 6.18 (The algorithms of join and inclusion checking)**.** *The algorithms of join and inclusion checking are formalized in Figure 6.15.*

Theorem 6.9 (Soundness of join and inclusion checking)**.** *Operators* $\overline{\textbf{join}}^{\overline{c}}$ *and* $\overline{\textbf{incl}}^{\overline{c}}$ *are sound in the following sense.*

$$
\begin{array}{c}
\gamma^c(\overline{c}_0) \cup \gamma^c(\overline{c}_1) \subseteq \gamma^c(\overline{\textbf{join}}^{\overline{c}}(\overline{c}_0, \overline{c}_1)) \\
\overline{\textbf{incl}}^{\overline{c}}(\overline{a}_0, \overline{a}_1) = \textit{true} \Longrightarrow \gamma^c(\overline{a}_0) \subseteq \gamma^c(\overline{a}_1)
\end{array}
$$

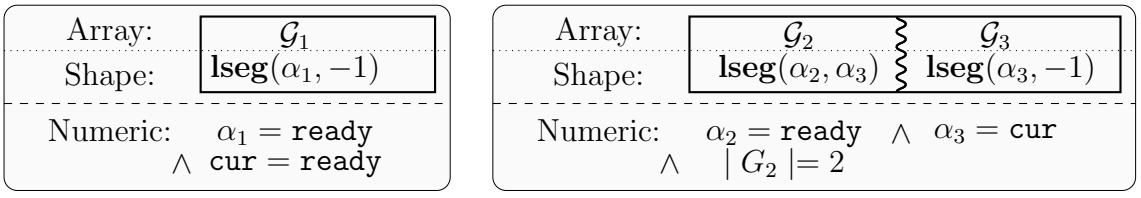
$\overline{\textbf{join}}^{\overline{\mathsf{c}}}(\overline{\mathsf{c}}_0, \overline{\mathsf{c}}_1)\{$
  foreach($\overline{\mathtt{A}}_i$ **in** $\overline{\mathsf{c}}_0$)
    foreach($\overline{\mathtt{A}}_j$ **in** $\overline{\mathsf{c}}_1$)
      $W_{ij} = \overline{\textbf{rank}}^{\overline{\mathsf{c}}}(\overline{\mathtt{A}}_i, \overline{\mathtt{A}}_j)$;
  $\overline{\mathsf{c}}_0, \overline{\mathsf{c}}_1 = \overline{\textbf{repartition}}_{\overline{\textbf{join}}^{\overline{\mathsf{c}}}}(W, \overline{\mathsf{c}}_0, \overline{\mathsf{c}}_1)$;
  return $\overline{\textbf{join}}^{\overline{\mathsf{c}}}_{\equiv}(\overline{\mathsf{c}}_0, \overline{\mathsf{c}}_1)$;
}

(a) The algorithm of the join operator

$\overline{\textbf{incl}}^{\overline{\mathsf{c}}}(\overline{\mathsf{c}}_0, \overline{\mathsf{c}}_1)\{$
  foreach($\overline{\mathtt{A}}_i$ **in** $\overline{\mathsf{c}}_0$)
    foreach($\overline{\mathtt{A}}_j$ **in** $\overline{\mathsf{c}}_1$)
      $W_{ij} = \overline{\textbf{rank}}^{\overline{\mathsf{c}}}(\overline{\mathtt{A}}_i, \overline{\mathtt{A}}_j)$;
  $\overline{\mathsf{c}}_0, \overline{\mathsf{c}}_1 = \overline{\textbf{repartition}}_{\overline{\textbf{incl}}^{\overline{\mathsf{c}}}}(W, \overline{\mathsf{c}}_0, \overline{\mathsf{c}}_1)$;
  return $\overline{\textbf{incl}}^{\overline{\mathsf{c}}}_{\equiv}(\overline{\mathsf{c}}_0, \overline{\mathsf{c}}_1)$;
}

(b) The algorithm of the inclusion checking operator

Figure 6.15 – The algorithms of the join and inclusion checking



(a) One input state $\overline{\mathsf{c}}_0$

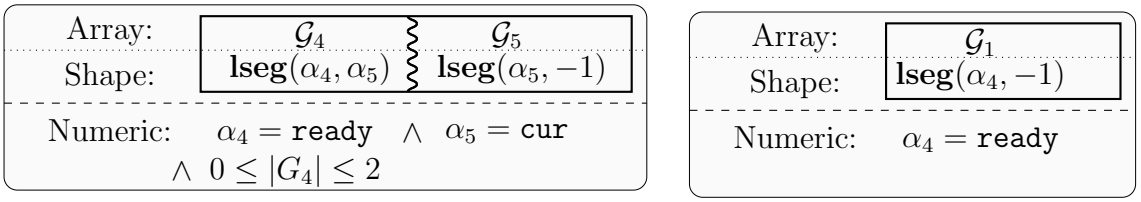
(b) The other input state $\overline{\mathsf{c}}_1$

Figure 6.16 – The input states for lattice operators

Example 6.8 (Join and widening)**.** *Suppose the abstract states in Figure 6.16(a) and Figure 6.16(b) are two input states:*

- *If we apply* $\overline{\textbf{join}}^{\overline{\mathsf{c}}}$ *on them, our analysis would pair* $\mathcal{G}_1$ *&* $\mathbf{lseg}(\alpha_1, -1)$ *in* $\overline{\mathsf{c}}_0$ *with* $\mathcal{G}_3$ *&* $\mathbf{lseg}(\alpha_3, -1)$ *in* $\overline{\mathsf{c}}_1$*, and no atomic memory predicate in* $\overline{\mathsf{c}}_0$ *is paired with* $\mathcal{G}_2$ *&* $\mathbf{lseg}(\alpha_2, \alpha_3)$ *in* $\overline{\mathsf{c}}_1$*. Thus a new atomic memory predicate* $\mathcal{G}_0$ *&* $\mathbf{lseg}(\alpha_1, \alpha_1)$ *is added to* $\overline{\mathsf{c}}_0$ *and paired with* $\mathcal{G}_2$ *&* $\mathbf{lseg}(\alpha_2, \alpha_3)$ *in* $\overline{\mathsf{c}}_1$*. The parameters of the paired memory predicates are renamed to the same. The final result is shown in Figure 6.17(a).*

- *If we apply* $\overline{\textbf{widen}}^{\overline{\mathsf{c}}}$ *on them, our analysis would pair both* $\mathcal{G}_2$ *&* $\mathbf{lseg}(\alpha_2, \alpha_3)$ *and* $\mathcal{G}_3$ *&* $\mathbf{lseg}(\alpha_3, -1)$ *in* $\overline{\mathsf{c}}_1$ *with* $\mathcal{G}_1$ *&* $\mathbf{lseg}(\alpha_1, -1)$ *in* $\overline{\mathsf{c}}_0$*. Then* $\mathcal{G}_2$ *&* $\mathbf{lseg}(\alpha_2, \alpha_3) * \mathcal{G}_3$ *&* $\mathbf{lseg}(\alpha_3, -1)$ *is folded into* $\mathcal{G}_2$ *&* $\mathbf{lseg}(\alpha_2, -1)$ *and paired to* $\mathcal{G}_1$ *&* $\mathbf{lseg}(\alpha_1, -1)$ *in* $\overline{\mathsf{c}}_0$*. The parameters of the paired memory predicates are renamed to the same. The final result is shown in Figure 6.17(b).*



(a) The join result (b) The widening result

Figure 6.17 – The result of lattice operators

6.6 Analysis

In this section, we formalize an abstract interpreter based on the coalescing domain for the language of Figure 5.1, and we discuss in detail the full analysis of the `create` example.

6.6.1 Abstract Semantics and Implementation

Abstract semantics. The definition of abstract semantics follows the principles defined in Chapter 3. Given a statement $\mathtt{s}$, the analysis function $[\![\mathtt{s}]\!] : \mathcal{C} \to \mathcal{C}$ inputs an abstract pre-condition and returns an abstract post-condition. Our analysis uses $\mathbf{lfp}^{\#}$ defined in Chapter 3 to compute least fix points, and uses $\overline{\mathbf{fusion}} : \mathcal{P}(\mathcal{C}) \to \mathcal{C}$ to compute the over-approximation of any number of abstract states (by applying $\overline{\mathbf{join}}^{\overline{\mathtt{c}}}$ repeatedly).

Definition 6.19 (Abstract semantics in the coalescing domain)**.** *The* abstract semantics $[\![\mathtt{s}]\!]$ *of a program* $\mathtt{s}$ *is defined by:*

$$
\begin{array}{rcl}
[\![\mathtt{l} = \mathtt{e}]\!](\overline{\mathtt{c}}) & = & \overline{\mathbf{fusion}} \circ \overline{\mathbf{assign}}^{\overline{\mathtt{c}}}[\![\mathtt{l} = \mathtt{e}]\!](\overline{\mathtt{c}}) \\
[\![\mathtt{s}_0; \mathtt{s}_1]\!](\overline{\mathtt{c}}) & = & [\![\mathtt{s}_1]\!] \circ [\![\mathtt{s}_0]\!](\overline{\mathtt{c}}) \\
[\![\mathbf{if}(\mathtt{e}_0 \lessdot \mathtt{e}_1)\{\mathtt{s}_0\}\mathbf{else}\{\mathtt{s}_1\}]\!](\overline{\mathtt{c}}) & = & \overline{\mathbf{join}}^{\overline{\mathtt{c}}}([\![\mathtt{s}_0]\!] \circ \overline{\mathbf{fusion}} \circ \overline{\mathbf{guard}}^{\overline{\mathtt{c}}}[\![\mathtt{e}_0 \lessdot \mathtt{e}_1]\!](\overline{\mathtt{c}}), \\
 & & \qquad [\![\mathtt{s}_1]\!] \circ \overline{\mathbf{fusion}} \circ \overline{\mathbf{guard}}^{\overline{\mathtt{c}}}[\![\mathtt{e}_0 \gtrdot \mathtt{e}_1]\!](\overline{\mathtt{c}})) \\
[\![\mathbf{while}(\mathtt{e}_0 \lessdot \mathtt{e}_1)\{\mathtt{s}\}]\!](\overline{\mathtt{c}}) & = & \overline{\mathbf{fusion}} \circ \overline{\mathbf{guard}}^{\overline{\mathtt{c}}}[\![\mathtt{e}_0 \gtrdot \mathtt{e}_1]\!]\mathbf{lfp}^{\#}_{\overline{\mathtt{c}}} \ \mathbf{F}^{\#}) \\
\multicolumn{3}{l}{\quad \textit{where } \mathbf{F}^{\#}(\overline{\mathtt{c}}) = [\![\mathtt{s}]\!](\overline{\mathbf{fusion}} \circ \overline{\mathbf{guard}}^{\overline{\mathtt{c}}}[\![\mathtt{e}_0 \lessdot \mathtt{e}_1]\!](\overline{\mathtt{c}})) } \\
\multicolumn{3}{l}{\quad \textit{and } \mathbf{lfp}^{\#}_{\overline{\mathtt{c}}} \textit{ computes post-fixpoint with } \overline{\mathbf{widen}}^{\overline{\mathtt{c}}}, \ \overline{\mathbf{join}}^{\overline{\mathtt{c}}} \textit{ and } \overline{\mathbf{isle}}^{\overline{\mathtt{c}}}}
\end{array}
$$

Theorem 6.10 (Soundness of the abstract semantics in the coalescing domain)**.** *For all program* $\mathtt{s}$ *and abstract pre-condition* $\overline{\mathtt{c}}$*,*

$$[\![\mathtt{s}]\!](\gamma^c(\overline{\mathtt{c}})) \subseteq \gamma^c([\![\mathtt{s}]\!](\overline{\mathtt{c}}))$$

Implementation. We have implemented the array/shape coalescing domain inside the MemCAD analyzer [SR12] using the Apron numeric abstract domain library [JM09].

In this section, we describe the details of the analysis on the motivating example in Section 6.1. In the next chapter, we will carry out experiments on multiple operating system components, which shows that our analysis can be parameterized and applied to other structures.

6.6.2 Example `create` Revisited

Now we look at the analysis on the function `create`, the code and the analysis of which are shown in Figure 6.18 and Figure 6.19.

The analysis starts with the global correctness condition $\mathcal{R}^{aos}$ as pre-condition, which is recalled at ① in Figure 6.18. The assignment at line 2 updates the value of the variable `i` to 0. In the first iteration of the **while** loop at line 3, the group that the cell `a[i]` belongs to is unknown. However, our analysis infers that `a[i]` stores 0 in field `used` thanks to the condition test **if**(`a[i].used` $== 0$). Thus our analysis infers that `a[i]` is a cell belonging to group G_0, since numeric predicates on array contents entail that $I_1^{\texttt{used}} = I_2^{\texttt{used}} = I_3^{\texttt{used}} = 1$ (i.e., cells in groups G_1, G_2, G_3 only store 1 in field `used`).

The assignments at line 5 and 6 update the contents of cell `a[i]`. The fields of this cell are summarized in atomic memory predicate $\mathcal{G}_0$ & **true**. To perform strong updates on assignments, the transfer function unfolds $\mathcal{G}_0$ & **true**. The empty disjunctive case is unreachable since $|G_0| \geq 0$ (this is inferred by $\texttt{i} \in G_0$). In the non-empty disjunctive case, our analysis singles out the modified cell: the new atomic memory predicate $\mathcal{G}_4$ & **true** contains exactly this cell (i.e., $\texttt{i} \in G_4$ and $|G_4| = 1$), and lets G_0 represent the other cells. After unfolding, it performs a strong update on $I_4^{\texttt{used}}$. The abstract state after line 6 is shown at ②.

We now discuss the analysis of the assignments at lines 19 and 20. The abstract state before line 19 is actually quite similar to that at ②, except that the numeric predicates include $0 \leq \texttt{i} \leq 99 \wedge \texttt{ready} \neq -1$ instead of $\texttt{i} = 0$. The analysis of the assignment at line 19 is trivial and only requires the update of numeric predicates by adding $\texttt{pre} = \texttt{ready}$. Line 20 causes the reading of the array cell `a[ready]` which is also the head of the list of ready processes (because of $\texttt{ready} = \alpha_3$ and $\mathcal{G}_3$ & $\mathbf{slseg}(\alpha_3, -1)$). This cell is part of an inductive predicate, thus the analysis carries out the unfolding. Since the abstract state contains predicates $\texttt{ready} \neq -1$ and $\texttt{ready} = \alpha_3$, only the non empty disjunctive case is possible. The unfolding result generates an atomic memory predicate $\mathcal{G}_5$ & α_5@($\texttt{next} \mapsto \alpha_6, \texttt{prio} \mapsto \beta_5$) that describes the array cell to be updated. The assignment result after line 20 is shown at ③.

Now we look at the analysis on the loop at line 21 in Figure 6.18. To compute precise loop invariants, the analysis unrolls loops once, and then computes an abstract iteration sequence with widening. When widening is applied for the first time in the sequence for the loop at line 21, the arguments have different number of atomic memory predicates because the unfolding at line 22 increases the number by one at each iteration.

The abstract states after the first and the second iteration in this loop are shown at ④ and ⑤, where atomic memory predicates $\mathcal{G}_0$ & $\ldots * \mathcal{G}_1$ & $\ldots * \mathcal{G}_2$ & $\ldots * \mathcal{G}_4$ & $\ldots$

are omitted since they are the same as ③. Our widening algorithm performs the following operations to make the two input states compatible.

- folding $\mathcal{G}_5$ & α_5@(next $\mapsto \alpha_7$, prio $\mapsto \beta_5$) and $\mathcal{G}_7$ & α_7@(next $\mapsto \alpha_8$, prio $\mapsto \beta_7$), which results in the atomic memory predicate $\mathcal{G}_{10}$ & **slseg**(α_{10}, α_8).
- associating $\mathcal{G}_5$ & ..., G_6 & ... and G_7 & ... at ④with G_{10} & ..., G_8 & ... and G_9 & ... at ⑤ respectively.

The widening result is shown at ⑥.

6.7 Related Work

Another way of combining abstract domains $(\overline{D}_0, \ldots, \overline{D}_n)$ is reduced product [CC79], which expresses the logical conjunction of abstract elements in the form of $\overline{\mathtt{c}}_0 \wedge \ldots \wedge \overline{\mathtt{c}}_n$, where $\forall i \in [0, n], \overline{\mathtt{c}}_i \in \overline{D}_i$. The ASTRÉE analyzer [BCC+03a] utilizes reduced product to combine numeric domains into more expressive ones. In [CR08], memory predicates and shape predicates are combined to track both shape and data properties. The forms of reduced products that are introduced in [LYP11] and [TCR13] combine shape domains to abstract overlaid data structures. However, in reduced product, the correspondence between atomic memory predicates in two input domains are unclear, thus it is less precise than our coalescing domain.

One application of our coalescing domain is to combine an array domain and a shape domain to tackle dynamic structures nested into arrays. Few analyses have been developed to tackle such nested structures. In one hand, a large family of works have targeted numeric arrays, and often use segment abstractions [GRS05, HP08, CCL11], which prevents the inference of properties of non-contiguous sets of cells. Similar abstractions have been used in invariant generation, model checking and theorem proving [AGS14, JM07, KV09]. While such analyses can verify sortedness, they cannot cope with nested structural invariants such as the property $\mathcal{R}^{aos}$ defined in Section 6.1. Fluid updates [DDA10, DDA11] allow a precise tracking of container properties, and analyze precisely operations such as a vector copy, but cannot capture nested structure properties. The analysis of Chapter 5 handles non-contiguous regions, and can compute abstractions of numeric constraints (for instance that all cells in a group G_0 store an index in G_1), but cannot infer a precise invariant such as $\mathcal{R}^{aos}$, as it lacks a proper memory abstraction. Shape analysis for dynamic structures like [SRW99b, DOY06, CR08] do not cope with array-specific statements, like accesses with random indexes. One could suggest interpreting the array indexes as if they were pointers. However, index arithmetics, even being interpreted by pointer arithmetics, is beyond the scope of shape analyses targeted at dynamic structures. Thus, we would be back to the same problem: dealing with both array and linked structures in the same analysis. An additional difficulty for this idea is localization by contents. Condition test on the values stored in the data field of an array cell is often used to find out whether that cell is a list node. This kind of statements are rare in dynamic structures manipulating codes but are common in our target programs. To the best of our knowledge, there exists

no shape analysis could infer whether a memory cell is a list node (not necessarily the head node) by its contents on data field. Thus classical shape analyses would just fail to localize an array cell, which could cause huge precision loss.

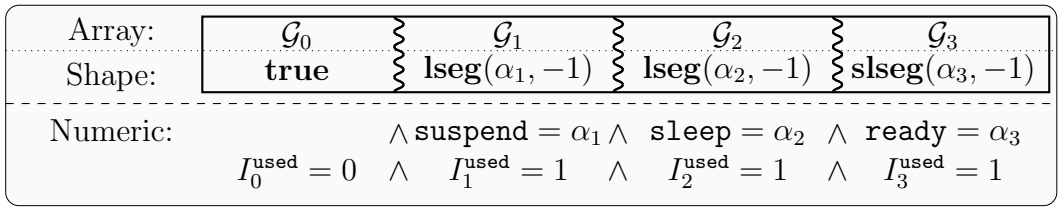
In the other hand, significant progresses have been achieved in the analysis of programs with dynamic structures. Such works either use three-valued logic [SRW99b] or separation logic [Rey02], and allow the verification of programs that manipulate dynamically linked data-structures such as variants of lists [SRW99b, DOY06, BCC+07] and trees [CR08]. In the other hand, these shape analyses cannot express that a structure lies inside an array, or a fixed contiguous space. Our work also extends the notion of abstraction parameterized by user supplied structure definitions of [CR08] to also deal with structures stored in arrays.

A notable exception is [SR12], which extends a shape analysis with structures nested into abstractions of memory blocks. This abstraction is limited to the case where a structure is stored in a non empty and contiguous region, and cannot cope with the examples discussed in Section 6.1. To achieve this more powerful association of array and structure reasoning, our analysis restricts the form of inductive predicates to one atomic inductive summary per region, which simplifies analysis algorithms, yet allows to deal with more complex structures.

6.8 Conclusion

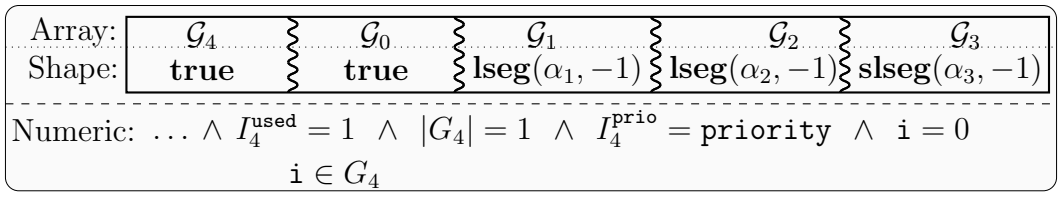
To summarize, our work contributes a novel method of combining memory abstractions, which is called coalescing. In the logical point of view, the coalescing abstraction consists of local conjunctions of predicates taken in two different memory abstract domains. By keeping conjunctions of memory predicates local, it provides a greater precision than a conventional reduced product would. Coalescing an array abstraction and a dynamic structure abstraction constructs an analysis on structures nested into non-contiguous blocks in arrays.

①




```
1   void create(int priority){
2     int i = 0;
3     while(i < 100){
4       if(a[i].used == 0){
5         a[i].used = 1;
6         a[i].prio = priority;
```

②




```
7           break;
8       }
9       i ++;
10    }
11    // corner cases
12      ...
13    // insert a[i] to ready list
14    // for case ready  = -1
15      ...
16    // for case a[i].prio ≤ a[ready].prio
17        ...
18    int pre, cur;
19    pre = ready;
20    cur = a[ready].next;
```

③

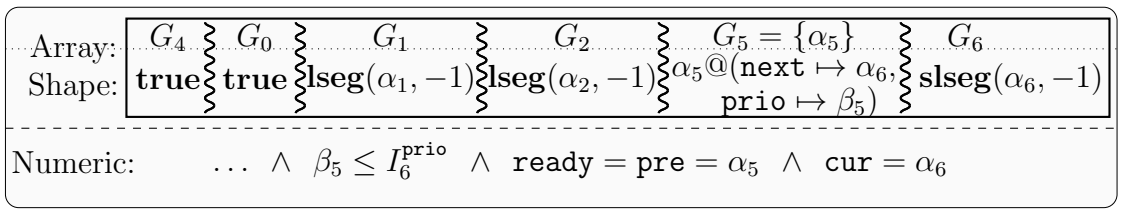
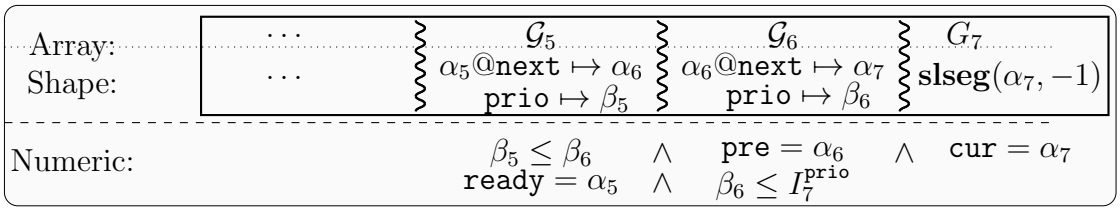
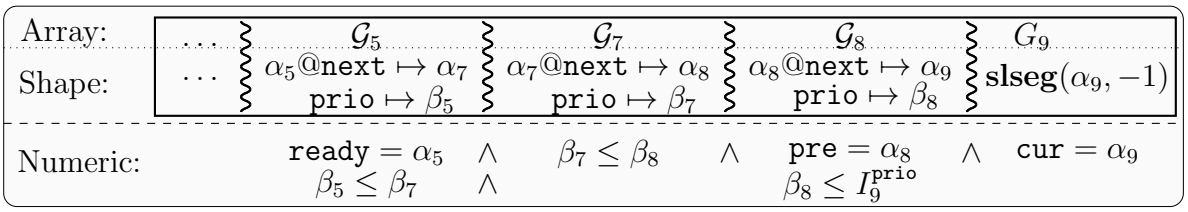


Figure 6.18 – The analysis on function `create`: part 1

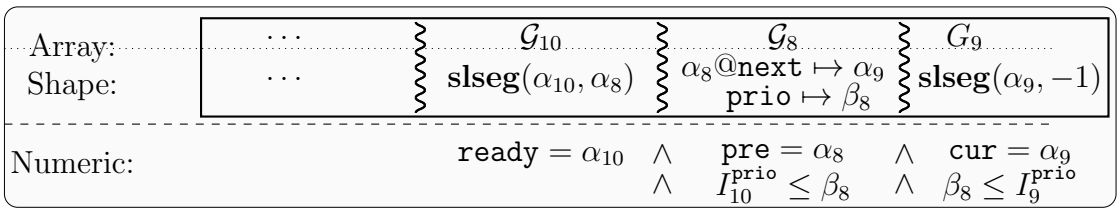
④



⑤



⑥




```
21    while(cur != −1){
22      if(a[cur].prio > priority)
23        break;
24      pre = cur;
25      cur = a[cur].next;
26    }
27    a[pre].next = i;
28    a[i].next = cur;
29    // other fields initialization
30      ...
31    return i;
32  }
```

Figure 6.19 – The analysis on function `create`: part 2

Chapter 7

Experiments on OS Components

In this chapter, we evaluate the effectiveness of the techniques presented in this manuscript via experiments on components of operating systems with the implementation of our array/shape coalescing domain. It is an overall evaluation of all the techniques that we have introduced, since the array/shape coalescing domain is built on top of a shape domain [CR08] and the non-contiguous domain presented in Chapter 5, which utilizes the Maya+ domain formalized in Chapter 4 to describe numeric predicates. From the evaluation, we expect to show three aspects of our static analysis, which we detail below.

Expressiveness. Expressiveness is one important criteria to evaluate a static analysis since it determines the properties that can be verified. Our coalescing domain can describe structural invariants on overlaid data structures, thus it can be used to verify properties like "the preservation of sorted lists in an array". This kind of properties are often necessary in some low-level programs like OS components, where dynamic memory allocation is not always allowed.

Efficiency. The efficiency of an analysis determines its scalability. In our case, since the target programs are system calls in real-time operating systems, the benchmarks are not large (usually around 100 LOC). We believe that the analysis for one system call taking several seconds is acceptable in practice (not too long to distract users' attention).

User friendliness. One barrier that prevents some static analyses being used in industry is the efforts needed to use them. These efforts consist of necessary training on users, specifications writing and sometimes modification on the code. We believe that the efforts (mainly including the specification for pre- and post-conditions of system calls) to employ our static analysis is reasonable.

7.1 Experiments Setup

7.1.1 Target Programs

We choose a set of OS components as the benchmarks to evaluate our analysis. This choice is based on the following facts.

- **Verification of OS compoments is important.** Since any fault in the implementation of an operating system could affect the user processes running on it or even halt the whole system unexpectedly, the reliability of a computer system can only be as good as that of the operating system in it. Thus the verification on operating system components is important by all means.

- **Verification of OS compoments is challenging**. There has been many attempts [KEH+09, YH10, PF10, OMLB16] to ensure the correctness of components of OS by formal verification. Most of them are based on theorem proving and specific to one version of a certain OS. Verification by static analysis is much harder since the data structures in OS are usually complex and it is non-trivial to design automatic reasoning algorithms of proper precision to prove meaningful properties.

- **Verification of OS compoments shows the strength of our anlysis.** Our coalescing domain can describe overlaid data structures, especially dynamic structures in arrays, which happen to be common in real time operating systems. Thus experiments on these operating systems could demonstrate the strength of our analysis.

The benchmarks are summarized in Table 7.1.

- AOS is an industrial embedded real-time operating system. It is not open source and we are not authorized to give the background of it. The task scheduler in AOS maintains three lists in an array to record all the running tasks in the states of "suspend", "sleep" and "ready" respectively. The list of "ready" tasks are sorted with respect to the priority level. We take the task scheduler into our benchmark since it is representative and challenging (multiple lists including a sorted one in an array).

- TinyOS [LMP+05] is an embedded, application-specific operating system designed for sensor networks. Each node in such networks integrates a low-power CPU with limited memory and radio or optical communication, so that they can interact with the environment and each other through sensors, actuators and communication. The limited resources on each node is for sake of cost, which is pivotal to the application of sensor networks, since typical applications of sensor networks like environmental monitoring and seismic analysis of structures, could easily require thousands of nodes. Moore's law will be applied to reduce size and cost rather than increase capability, to make Internet of things practical which needs huge amount of nodes.

Thus operating systems which run with limited resources and low power such as TinyOS are expected.

TinyOS is implemented in the nesC language [GLvB+03], which is an extension to the C programming language. nesC is a component-based, event-driven programming language, developed for networked embedded system. One feature of nesC is that it does not allow dynamic memory allocation. However, in the task scheduler in TinyOS, dynamicly linked lists are needed and implemented in arrays. Thus we choose the task scheduler in TinyOs as one of our benchmarks.

- Minix is a Unix-like multitasking computer operating system [TWTT87]. 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*.

  In the memory management service, a list recording all allocated memory blocks is stored in an array, and all free slots in that array is also linked as a list. Thus the two lists occupy the whole array. We take the system calls manipulating this array as parts of our benchmarks.

- Eicon [eic] is a company providing telephony boards for PC servers. Their products include Diva server, which is a range of telocoms products for voice, speech, conferencing and fax. It supports various protocols such as T1/E1 and ISDN.

  In the linux driver for Diva server [div], the adapter request queue is maintained by a list in an array. One thing special about this list is that a variable recording the length of the list makes some operations faster (e.g., judging whether there are free slots in the array). We also include this driver in our benchmarks.

- Nordic nRF51 series [nor] is a family of system-on-chip (SoC) devices for ultra-low power wireless applications. They support a range of protocol stacks including Bluetooth low energy and ANT.

  To satisfy the requirement on low energy consumption, the applications running on it often avoids using dynamic memory allocation. In its timer application [tim], a sorted list is maintained in an array. We also put that application into our benchmarks.

All these programs manipulate one common type of overlaid data structure "lists inside arrays". However, we choose them as our benchmarks since they are representive in their fields and the structural properties that each of these modules relies on have different characteristics, that are also summarized in Table 7.1: (1) The numbers of lists stored in a single array in each module are various, as indicated in the row "Lists". (2) In the benchmark from Minix, all array cells are linked in two lists, thus one property of that array is that no array cell should be out of the control of the lists. This characteristic

System	AOS	TinyOS	Minix	Linux	Nordic
Module	task scheduler	task scheduler	memory management	Eicon network driver	application timer
Functions	`tinit` `tcreat` `tstop` `tsched` `tstart`	`push_task` `pop_task`	`tinit` `alloc_mem` `free_mem` `max_hole`	`insert` `delete` `traversal`	`insert` `delete`
Lists	3	1	2	1	1
Free slots	Yes	Yes	No	Yes	Yes
Tail pointer	No	Yes	No	Yes	No
Length information	No	No	No	Yes	No
Sortedness	Yes	No	No	No	Yes

Table 7.1 – Analyzed programs and invariants

is indicated in the row "Free slots", which specifies whether there are array cells storing "free" elements. (3) Some modules use a special variable to hold the index of the last element of each list. This is indicated in the row "Tail pointer". (4) The row "Length information" specifies whether a special variable holds the length of each list. (5) The row "Sortedness" says whether the structural invariant also relies on the sortedness of the lists.

7.1.2 Verification Framework

Environment. We implemented the analysis inside the MemCAD analyzer [SR12], which is a forward abstract interpreter that performs intra-procedural static analysis or fully context-sensitive inter-procedural inter-procedure analysis (does not support recursion) based on the ASTs generated by front-end Clang [App]. All our experiments are carried out on on Ubuntu 12.04.4, with 16 Gb of RAM, on an Intel Xeon E3 desktop, running at 3.2 GHz.

Verification methodology. Each of the considered cases relies on a structure that can be easily described using a structural inductive definition (e.g., Example 6.1). These definitions serve as specifications that drive the abstraction, and are used as basis of pre- and post-conditions, which may also include additional numeric relations. With these specifications, the verification process can be formalized as follows.

$$\textbf{assume}(\mathcal{R});\ \texttt{systemcall}();\ \textbf{assert}(\mathcal{R});$$

That is, given a structural invariant $\mathcal{R}$ as pre-condition, we let our analysis compute an abstract post-condition and check that $\mathcal{R}$ still holds at the exit of the function.

Analysis options. We choose to use the "New Polka" polyhedra domain in APRON library [JM09] as the underlying abstract numeric domain. Our strategy for loop iteration is applying join for the first iterate and then widening for the rest. The maximal time for the analysis on one function is set to be 1 minute.

7.2 Verified Properties

In this section, we show the properties that have been verified by our static analysis on the four OS components.

AOS. In Chapter 6, we have presented the analysis on examples from the task scheduler in AOS. Now we recall the invariants $\mathcal{R}^{aos}$ as follows.

Array:	$\mathcal{G}_0$	$\mathcal{G}_1$	$\mathcal{G}_2$	$\mathcal{G}_3$
Shape:	$\mathbf{true}$	$\mathbf{lseg}(\alpha_1, -1)$	$\mathbf{lseg}(\alpha_2, -1)$	$\mathbf{slseg}(\alpha_3, -1)$
Numeric:	$\wedge\, \mathtt{suspend} = \alpha_1 \wedge$	$\mathtt{sleep} = \alpha_2 \wedge$	$\mathtt{ready} = \alpha_3$	
	$I_0^{\mathtt{used}} = 0 \wedge$	$I_1^{\mathtt{used}} = 1 \wedge$	$I_2^{\mathtt{used}} = 1 \wedge$	$I_3^{\mathtt{used}} = 1$

where

$$\begin{array}{rcl} \mathcal{G}\ \&\ \mathbf{lseg}(\pi,\tau) & ::= & \langle \mathcal{G}\ \&\ \mathbf{emp}, \pi = \tau\rangle \\ & \vee & \langle \mathcal{G}'\ \&\ \pi@(\mathtt{next} \mapsto \pi') * \mathcal{G}''\ \&\ \mathbf{lseg}(\pi',\tau), \pi \neq \tau\rangle \end{array}$$

and

$$\begin{array}{rcl} \mathcal{G}\ \&\ \mathbf{slseg}(\pi,\tau) & ::= & \langle \mathcal{G}\ \&\ \mathbf{emp}, \pi = \tau\rangle \\ & \vee & \langle \mathcal{G}'\ \&\ \pi@(\mathtt{next} \mapsto \pi', \mathtt{prio} \mapsto \pi'') * \mathcal{G}''\ \&\ \mathbf{slseg}(\pi',\tau), \\ & & \pi \neq \tau\ \wedge\ \pi'' \leq I_{G'}^{\mathtt{prio}}\rangle \end{array}$$

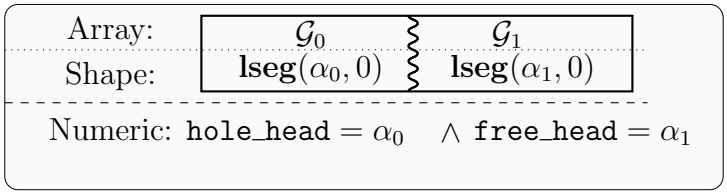
This property indicates that variables `ready`, `sleep` and `suspend` should point to the heads of three well-formed acyclic disjoint lists, where the list with head `ready` is sorted with respect to the values in field `prio`, and all free slots and used nodes in the array are distinguished by the values stored in their `used` field (0 for free slots and 1 for used slots).

System calls that manipulate this data structure include `tinit` (initialize the array and the three list variables), `tcreat` (locate a free slot in the array and insert it into the ready list), `tstop` (release a list node to be free), and `tsched` (move array nodes between lists), and `tstart` (move one node from the sleeping list to the ready list). The verification is carried out by proving the following assertions.

$$\begin{array}{lll}
\textbf{assume}(\top) & \texttt{tinit}() & \textbf{assert}(\mathcal{R}^{aos}); \\
\textbf{assume}(\mathcal{R}^{aos}) & \texttt{tcreat}() & \textbf{assert}(\mathcal{R}^{aos}); \\
\textbf{assume}(\mathcal{R}^{aos}) & \texttt{tstop}() & \textbf{assert}(\mathcal{R}^{aos}); \\
\textbf{assume}(\mathcal{R}^{aos}) & \texttt{tsched}() & \textbf{assert}(\mathcal{R}^{aos}); \\
\textbf{assume}(\mathcal{R}^{aos}) & \texttt{tstart}() & \textbf{assert}(\mathcal{R}^{aos});
\end{array}$$

7.2.1 Minix

The Minix memory management module maintains two lists inside an array: one of them stores the allocated blocks whereas the other stores the available nodes. Any cell in the array belongs to either of these two lists. An interesting property of these two lists is that they occupy the whole array, and no array cell is leaked during operations on the lists. This fact is expressed in our abstraction by partitioning all the array cells into only two groups, each containing one list. The invariant $\mathcal{R}^m$ is shown as follows.

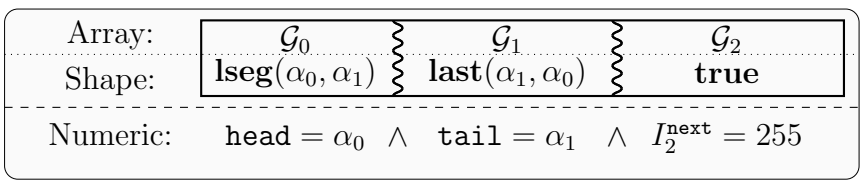


The functions manipulating this array include `tinit` (initialization), `alloc_mem` (move one node from the list of available nodes list to the list of allocated blocks), `free_mem` (the reverse operation of `alloc_mem`), `max_hole` (perform a traversal following a list structure). The verification is carried out by proving the following assertions.

$$\begin{array}{lll}
\textbf{assume}(\top) & \texttt{tinit}() & \textbf{assert}(\mathcal{R}^{m}); \\
\textbf{assume}(\mathcal{R}^{m}) & \texttt{alloc_mem}() & \textbf{assert}(\mathcal{R}^{m}); \\
\textbf{assume}(\mathcal{R}^{m}) & \texttt{free_mem}() & \textbf{assert}(\mathcal{R}^{m}); \\
\textbf{assume}(\mathcal{R}^{m}) & \texttt{max_hole}() & \textbf{assert}(\mathcal{R}^{m});
\end{array}$$

7.2.2 TinyOS

The task scheduler of TinyOS maintains one singly linked list (see row "Lists" in Table 7.1) in an array, the head and tail nodes of which are indexed by two integer variables `head` and `tail` respectively (see row "Tail pointer"). The array also contains free slots besides list nodes (see row "Free slots"). The invariant $\mathcal{R}^t$ is shown as follows.



where

$$\begin{array}{rcl} \mathcal{G} \,\&\, \mathbf{lseg}(\pi,\tau) & ::= & \langle \mathcal{G} \,\&\, \mathbf{emp}, \pi = \tau\rangle \\ & \vee & \langle \mathcal{G}' \,\&\, \pi@(\mathtt{next} \mapsto \pi') * \mathcal{G}'' \,\&\, \mathbf{lseg}(\pi',\tau), \pi \neq \tau\rangle \end{array}$$

and

$$\begin{array}{rcl} \mathcal{G} \,\&\, \mathbf{last}(\pi,\tau) & ::= & \langle \mathcal{G} \,\&\, \mathbf{emp}, \pi = 255 \wedge \tau = 255\rangle \\ & \vee & \langle \mathcal{G} \,\&\, \pi@(\mathtt{next} \mapsto \pi'), \pi \neq 255 \wedge \tau \neq 255 \wedge \pi' = 255\rangle \end{array}$$

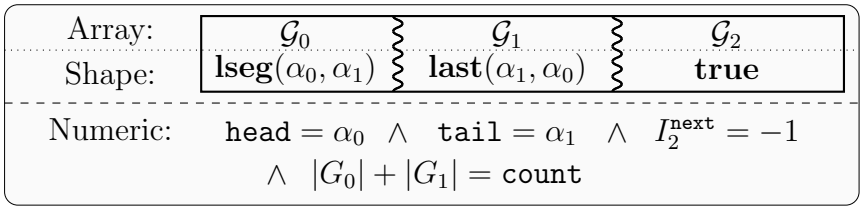
Note that, even if there are only two lists, our abstraction partitions the array into three groups, since the additional group G_2 is needed to store free elements (that are in neither of these lists). The `next` field of these cells stores value 255 ($I_2^{\mathtt{next}} = 255$).

The functions manipulating this array include the system call `pop_task` which pops a task from the list head, and the system call `push_task` which pushes one task to the list tail. The verification is carried out by proving the following assertions.

$$\begin{array}{ccc} \mathbf{assume}(\mathcal{R}^t) & \mathtt{pop\_task}() & \mathbf{assert}(\mathcal{R}^t); \\ \mathbf{assume}(\mathcal{R}^t \wedge 0 \leq \mathtt{id} \leq 255) & \mathtt{push\_task}(\mathbf{int}\ \mathtt{id}) & \mathbf{assert}(\mathcal{R}^t); \end{array}$$

7.2.3 Eicon

The Eicon network driver for Linux maintains a list in an array to deal with adapter request queue. They also maintain two variables `head` and `tail` which point to the head and tail of the list (just like TinyOS). A specific feature of the Eicon network driver is that it also has a variable `count` to record the length of the list (as shown in row "Length information"). By comparing `count` with the length of the array, the driver can quickly know whether there are free slots in the array. The invariant $\mathcal{R}^e$ is shown as below.

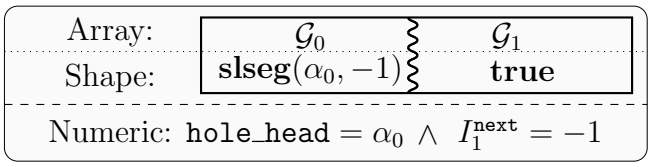


Programs manipulating this array include `insert` (insert one request to the tail of the list), `delete` (delete one request at the head of the list), `traversal` (traverse the list). The verification is carried out by proving the following assertions.

$$\begin{array}{ccc} \mathbf{assume}(\mathcal{R}^e) & \mathtt{insert}() & \mathbf{assert}(\mathcal{R}^e); \\ \mathbf{assume}(\mathcal{R}^e) & \mathtt{delete} & \mathbf{assert}(\mathcal{R}^e); \\ \mathbf{assume}(\mathcal{R}^e) & \mathtt{traversal}() & \mathbf{assert}(\mathcal{R}^e); \end{array}$$

7.2.4 Nordic

The timer application taken from the Nordic nRF51 SoC maintains a sorted list inside an array. Each node records an application with running time information. The list is sorted according to the ticks left for each application. The invariant $\mathcal{R}^n$ is shown as below.



Programs manipulating this array include `insert` (insert one timer according to its id) and `delete` (delete one timer according to its id). The verification is carried out by proving the following assertions.

$\mathbf{assume}(\mathcal{R}^n \wedge 0 \leq \texttt{timer\_id} < \texttt{max\_timers})$ `insert(int timer_id)` $\mathbf{assert}(\mathcal{R}^n)$;
$\mathbf{assume}(\mathcal{R}^n \wedge 0 \leq \texttt{timer\_id} < \texttt{max\_timers})$ `delete(int timer_id)` $\mathbf{assert}(\mathcal{R}^n)$;

7.3 Efficiency

Table 7.2 shows the results of the analysis on the functions mentioned in Table 7.1. The table indicates the number of lines of codes and analysis times. Note that we distinguish the definition of complex data structures (noted as LOCs(d)) and other codes (noted as LOCs(f)), since in some benchmarks like Eicon, the definition of data structures could account for the most of LOC.

The analysis successfully verifies all these programs, using the aforementioned structure specifications. In each case, it verifies both memory safety and the preservation of the structural invariants attached to each case. In most cases, analysis run-times are under one second. While the programs are not very large, they are fairly subtle and with typical operating system primitives manipulating the pattern under study.

7.4 Effort Needed for Verification

In this section, we summarize the efforts that we have made to carry out all the experiments in this Chapter. This reflects how easy our static analysis is to use.

The effort to use our analysis lies in two aspects: writing specification and pre-processing the code: (1) Our analysis is parameterized by the specification describing the properties to be to verify (e.g., the invariants we show in Section 7.2). The specification should be provided by the users. (2) Since the targeting properties are at function level, thus we need to pre-process the benchmarks by extracting the system calls out to verify them as libraries. Another reason for pre-processing code is that our analyzer does not support recursion. Therefore we need to eliminate recursion where it emerges.

System	Program	LOCs(d)	LOCs(f)	Time
AOS	`tinit`	6	36	0.12
	`tcreat`	6	54	0.81
	`tstop`	6	83	1.68
	`tsched`	6	71	1.36
	`tstart`	6	110	2.29
TinyOS	`push_task`	1	30	0.11
	`pop_task`	1	24	0.11
Minix	`tinit`	4	13	0.19
	`alloc_mem`	4	46	0.38
	`free_mem`	4	59	0.58
	`max_hole`	4	15	0.31
Eicon	`insert`	157	43	0.24
	`delete`	157	18	0.12
	`traversal`	157	19	0.28
Nordic	`insert`	14	56	1.03
	`delete`	14	47	0.59

Table 7.2 – Average times in seconds

The specification on each system call is fast to write (just a few lines). However, to figure out the invariants may take several minuets or more than one hour, depending on the complexity of the function. In the case of pre-processing the code, the time we have spent on each system call varies: (1) it could be ignorable, when the system call is self-contained (such as the two functions in TinyOS); (2) it could take more than one hour when the dependency of data structures is deep (e.g., Nordic) or recursion arises in some functions (e.g., Minix).

We believe that real users may spend less time than us for two reasons: (1) the syntax of the specification in our analysis is easy to learn (they are based on simple separation logic), and inductive definitions (e.g., **lseg**) can be shared by all tasks; (2) users can do pre-processing and invariants extraction much faster than us since they know their systems much better than us.

7.5 Related Work and Conclusion

The importance of the verification of operating systems has been widely realized, and quite a few works on this issue have been presented.

Most of these works [KEH+09, YH10, PF10] rely on theorem proving. The mathematical logics (e.g., first-order logic) used in theorem proving are usually very expressive, thus theorem proving has the potential to fully prove the functional correctness of a given operating system. However, rich expressiveness comes with the sacrifice of automation. Even with the help of interactive provers, the verification is quite time-consuming. Take

project seL4 [KEH+09] as an example, which implements seL4 (a third-generation microkernel of L4 provenance) comprising 8700 lines of C code and 600 lines of assembler, and verifies the correctness of it except the virtual memory manger, the initialization code and the assembler. The implementation of the operating system took 2.2 py, and the proof cost 20 py. The proof is also sensitive to changes in the operating system. In the experience of seL4, adding a complex new data structure to the kernel supporting new API calls could cost 1.5-2 py to re-verify.

Verifications of operating systems by static analysis [OMLB16, WCM+16] utilize less expressive logic but are usually fully automatic. Compared to static analysis on user programs, static analysis on operating systems has to deal with difficulties like hardware abstraction, interrupts and complex data structures. In [OMLB16], the semantics of low-level hardware interactions are modeled as a register automaton. In [WCM+16], interrupts are considered into account and handled by sequentialization. Static analysis is specific to certain properties and can hardly prove the correctness of a system in all aspects. However, the advantage of static analysis is that it is not specific to a certain version of a given operating system, and could be applied to other versions or even other operating systems easily.

Our static analysis focuses on the automatic reasoning on properties of complex data structures, which is orthogonal to those in [OMLB16, WCM+16]. However, our work and those in [OMLB16, WCM+16] all contribute to the precision improvement of static analysis on operating systems. In particular, our experiments demonstrate that static analysis can verify non-trivial safety and functional properties on various OS components which may manipulate complex data structures.

Chapter 8

Conclusion and Discussion for The Future Work

In this thesis, we have contributed a series of techniques that can work together to address the difficulties of the verification of complex structural properties on array contents, that we have mentioned in the beginning of the thesis.

- The Maya(+) domain addresses the difficulty of describing numerical properties of array contents, the size of which could be unbounded;
- The non-contiguous partition array domain addresses the difficulty of extracting dynamic structures out of arrays, which usually occupy non-contiguous regions of arrays;
- The coalescing domain addresses the difficulty of reasoning about accesses into intertwined data structures (like dynamic structures in arrays) at the same time.

The Maya domain extends conventional numeric domains with the ability of abstracting optional variables (i.e., variables that may have no value). This domain can be used for numeric analysis on languages with optional data types. It can also be used to describe programs with dynamic allocations, where memory locations allocated in conditional branches are actually "optional". The Maya+ domain extends conventional numeric domains with the ability of abstracting possibly empty set variables (i.e., variables that may have a possibly empty set of values). In this thesis, it is used to summarize array indexes and contents.

Our array domain can describe numeric properties of non-contiguous cells. Compared with conventional static analysis on array contents, it enjoys several advantages: (1) it supportes non-contiguous partition, thus it can describe arrays where cells with similar properties are not contiguous more precisely; (2) it allows empty groups, thus the number of disjunction is fewer than those do not allow; (3) it is semantic, thus the partitioning is carried out during the analysis which avoids a syntactic pre-analysis and is more precise since semantic information can help the partitioning.

Our coalesced domain can describe and automatically reason about inter-wined data structures. We have shown its ability in verifying the safety and functional properties of programs manipulating "lists nested in arrays". These properties are not widely realized in the literature, but we believe that they are important, especially in low-level software where arrays with structural contents are often used.

Our experiments demonstrate the effectiveness of our techniques. The benchmarks are all operating system components, including task schedulers, memory management and drivers. The safety of these programs are essential and the verification of them is non-trivial, considering the complex data structures that they utilize.

As for the future work, we believe that the following directions are promising.

- The Maya(+) domain can be extended with the ability of describing more set relations (like set inclusion). In this way, it turns into a set domain supporting numeric constraints on set variables. This kind of domain has potential use in many fields, like abstraction for general containers.

- The performance of the non-contiguous partition domain can be improved by controlling the number of groups with a heuristic algorithm. From our experiments, the time consumption of our analysis is mainly determined by the number of partitioned groups. In this thesis, we do not set a threshold for the number of groups, which limits the use of this domain on large code bases. In the future work, we can develop some heuristic strategies to control the number of groups.

- The future work for our coalescing domain is using it to combine more shape domains. This can be used to abstract more intertwined data structures like "trees on a list".

Bibliography

[AGS13] 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)*, 2013.

[AGS14] Francesco Alberti, Silvio Ghilardi, and Natasha Sharygina. Decision procedures for flat array properties. In *TACAS*, 2014.

[All81] Frances E. Allen. The history of language processor technology in IBM. *IBM Journal of Research and Development*, 25(5):535–548, 1981.

[App] Apple, Inc. clang: a C language family frontend for LLVM. `http://clang.llvm.org/` ; accessed 01-Oct-2017.

[BCC+03a] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, 2003.

[BCC+03b] 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 *PLDI*, 2003.

[BCC+07] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *CAV*, 2007.

[BCLR04] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. In *Integrated Formal Methods, 4th International Conference, IFM 2004, Canterbury, UK, April 4-7, 2004, Proceedings*, pages 1–20, 2004.

[BDES12] Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. Accurate invariant checking for programs manipulating lists and arrays with infinite data. In *Automated Technology for Verification and Analysis - 10th International Symposium, ATVA 2012, Thiruvananthapuram, India, October 3-6, 2012. Proceedings*, pages 167–182, 2012.

[BHMR07] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Invariant synthesis for combined theories. In *VMCAI*, 2007.

[BHR84] Stephen D Brookes, Charles AR Hoare, and Andrew W Roscoe. A theory of communicating sequential processes. *Journal of the ACM (JACM)*, 31(3):560–599, 1984.

[BL68] Rodney M Burstall and Peter J Landin. Programs and their proofs: an algebraic approach. Technical report, DTIC Document, 1968.

[BMS06] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *VMCAI*, 2006.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.

[CCF13] Agostino Cortesi, Giulia Costantini, and Pietro Ferrara. A survey on product operators in abstract interpretation. *arXiv preprint arXiv:1309.5146*, 2013.

[CCL11] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*, 2011.

[CCR14] Arlen Cox, Bor-Yuh Evan Chang, and Xavier Rival. Automatic analysis of open objects in dynamic language programs. In *SAS*, 2014.

[CCS13] Arlen Cox, Bor-Yuh Evan Chang, and Sriram Sankaranarayanan. QUIC graphs: Relational invariant generation for containers. In *ECOOP*, 2013.

[CCS15] A. Cox, B.-Y. E. Chang, and S. Sankaranarayanan. QUIC graphs: relational invariant generation for containers. In *VMCAI*, 2015.

[CE82] Edmund Clarke and E Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *Logics of programs*, pages 52–71, 1982.

[CH78a] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, 1978.

[CH78b] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'78)*, pages 84–96, Tucson, Arizona, 1978. ACM.

[CLM+14] Liqian Chen, Jiangchao Liu, Antoine Miné, Deepak Kapur, and Ji Wang. An abstract domain to infer octagonal constraints with absolute value. In *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings*, pages 101–117, 2014.

[CMC08] Liqian Chen, Antoine Miné, and Patrick Cousot. A sound floating-point polyhedra abstract domain. In *APLAS*, 2008.

[CR08] Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *POPL*, 2008.

[CR13] Bor-Yuh Evan Chang and Xavier Rival. Modular construction of shape-numeric analyzers. In *Semantics, Abstract Interpretation, and Reasoning about Programs (SAIRP)*, 2013.

[DDA10] Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid updates: Beyond strong vs. weak updates. In *ESOP*, 2010.

[DDA11] Isil Dillig, Thomas Dillig, and Alex Aiken. Precise reasoning for programs using containers. In *POPL*, 2011.

[div] Linux driver for eicon diva server. `https://github.com/torvalds/linux/blob/master/drivers/isdn/hardware/eicon/io.c`. Accessed: 2017-10-18.

[DOY06] Dino Distefano, Peter W O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *TACAS*. 2006.

[eic] Eiconworks.com: Dialogic products and solutions. `http://www.eiconworks.com/`. Accessed: 2017-10-18.

[Flo67] Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.

[GDD+04] Denis Gopan, Frank DiMaio, Nurit Dor, Thomas W. Reps, and Shmuel Sagiv. Numeric domains with summarized dimensions. In *TACAS*, 2004.

[GIB+12] Khalil Ghorbal, Franjo Ivančić, Gogul Balakrishnan, Naoto Maeda, and Aarti Gupta. *Donut Domains: Efficient Non-convex Domains for Abstract Interpretation*, pages 235–250. 2012.

[GLvB+03] David Gay, Philip Levis, J. Robert von Behren, Matt Welsh, Eric A. Brewer, and David E. Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, pages 1–11, 2003.

[GMT08] Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, 2008.

[GRS05] Denis Gopan, Thomas W. Reps, and Shmuel Sagiv. A framework for numeric analysis of array operations. In *POPL*, 2005.

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[HP08] Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *PLDI*, 2008.

[JM07] Ranjit Jhala and Kenneth L. McMillan. Array abstraction from proofs. In *CAV*, 2007.

[JM09] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, 2009.

[Kar76] M. Karr. Affine relationships among the variables of a program. *Acta Informatica*, 1976.

[KdBdGZ52] Stephen Cole Kleene, NG de Bruijn, J de Groot, and Adriaan Cornelis Zaanen. *Introduction to metamathematics*, volume 483. van Nostrand New York, 1952.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[KV09] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over array using a theorem prover. In *FASE*, 2009.

[LMP+04] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. Tinyos: An operating system for sensor networks. In *in Ambient Intelligence*. Springer Verlag, 2004.

[LMP+05] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient intelligence*. 2005.

[LR15] Jiangchao Liu and Xavier Rival. Abstraction of arrays based on non contiguous partitions. In *VMCAI*, 2015.

[LYP11] Oukseh Lee, Hongseok Yang, and Rasmus Peteren. Program analysis for overlaid data structures. In *CAV*, 2011.

[MA15] David Monniaux and Francesco Alberti. A simple abstraction of arrays and maps by program translation. In *SAS*, pages 217–234, 2015.

[McM08] Kenneth L. McMillan. Quantified invariant generation using an interpolation saturation prover. In *TACAS*, 2008.

[MG16] David Monniaux and Laure Gonnord. Cell morphing: From array programs to array-free horn clauses. In *SAS*, 2016.

[Min04a] Antoine Miné. Weakly relational abstract domains. *Phd.D. thesis*, 2004.

[Min04b] A. Miné. Relational domains for the detection of floating point run-time errors. In *ESOP*, 2004.

[Min06] Antoine Miné. The octagon abstract domain. *HOSC*, 2006.

[nor] Nordic semiconductor. `http://www.nordicsemi.com/`. Accessed: 2017-10-18.

[OMLB16] Abdelraouf Ouadjaout, Antoine Miné, Noureddine Lasla, and Nadjib Badache. Static analysis by abstract interpretation of functional properties of device drivers in tinyos. *Journal of Systems and Software*, 120:114–132, 2016.

[PF10] Zhong Shao PI and Bryan Ford. Advanced development of certified os kernels. 2010.

[PTS+11] 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)*, 2011.

[Rey02] John Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[RTC14] Xavier Rival, Antoine Toubhans, and Bor-Yuh Evan Chang. Construction of abstract domains for heterogeneous properties (position paper). In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 489–492. Springer, 2014.

[Sch98] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[Sco70] Dana Scott. *Outline of a mathematical theory of computation*. 1970.

[Seb12] Robert W. Sebesta. Pearson, 10th edition, 2012.

[s:i02] The chinook helicopter disaster. *IMIS journal*, 12(2), 2002.

[SK05] Axel Simon and Andy King. Exploiting sparsity in polyhedral analysis. In *SAS*, volume 3672, pages 336–351. Springer, 2005.

[SMS13] H. Siegel, B. Mihaila, and A. Simon. The undefined domain: precise relational information for entities that do not exist. In *APLAS*, 2013.

[SPW09] Mohamed Nassim Seghir, Andreas Podelski, and Thomas Wies. Abstraction refinement for quantified array assertions. In *SAS*, 2009.

[SR12] Pascal Sotin and Xavier Rival. Hierarchical shape abstraction of dynamic structures in static blocks. In *APLAS*, 2012.

[SRW99a] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, 1999.

[SRW99b] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, 1999.

[SS12] H. Siegel and A. Simon. Summarized dimensions revisited. *NSAD*, 2012.

[Tas02] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 7007(011), 2002.

[TCR13] Antoine Toubhans, Bor-Yuh Evan Chang, and Xavier Rival. Reduced product combination of abstract domains for shapes. In *VMCAI*, 2013.

[tim] Timer application. `https://github.com/finnurtorfa/nrf51/blob/master/lib/nrf51sdk/Nordic/nrf51822/Board/nrf6310/ble/ble_app_gzll/ble_gzll_app_timer.c`. Accessed: 2016-10-18.

[TWTT87] Andrew S Tanenbaum, Albert S Woodhull, Andrew S Tanenbaum, and Andrew S Tanenbaum. *Operating systems: design and implementation*, volume 2. 1987.

[WCM+16] Xueguang Wu, Liqian Chen, Antoine Miné, Wei Dong, and Ji Wang. Static analysis of runtime errors in interrupt-driven programs via sequentialization. *ACM Trans. Embedded Comput. Syst.*, 15(4):70:1–70:26, 2016.

[YH10] Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *ACM Sigplan Notices*, volume 45, pages 99–110. ACM, 2010.

[YLB+08] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In *CAV*, 2008.

List of Figures

List of Definitions

List of Theorems

List of Examples

Résumé

Dans cette thèse, nous étudions l'analyse statique par interprétation abstraites de programmes manipulant des tableaux, afin d'inférer des propriétés sur les valeurs numériques et les structures de données qui y sont stockées.

Les tableaux sont omniprésents dans de nombreux programmes, et les erreurs liées à leur manipulation sont difficile à éviter en pratique. De nombreux travaux de recherche ont été consacrés à la vérification de tels programmes. Les travaux existants s'intéressent plus particulièrement aux propriétés concernant les valeurs numériques stockées dans les tableaux. Toutefois, les programmes bas-niveau (comme les systèmes embarqués ou les systèmes d'exploitation temps-réel) utilisent souvent des tableaux afin d'y stocker des structures de données telles que des listes, de manière à éviter d'avoir recours à l'allocation de mémoire dynamique. Dans cette thèse, nous présentons des techniques permettant de vérifer par interprétation abstraite des propriétés concernant à la fois les données numériques ainsi que les structures composites stockées dans des tableaux.

Notre première contribution est une abstraction qui permet de décrire des stores à valeurs numériques et avec valeurs optionnelles (i.e., lorsqu'une variable peut soit avoir une valeur numérique, soit ne pas avoir de valeur du tout), ou bien avec valeurs ensemblistes (i.e., lorsqu'une variable est associée à un ensemble de valeurs qui peut être vide ou non). Cette abstraction peut être utilisée pour décrire des stores où certaines variables ont un type option, ou bien un type ensembliste. Elle peut aussi servir à la construction de domaines abstraits pour décrire des propriétés complexes à l'aide de variables symboliques, par exemple, pour résumer le contenu de zones dans des tableaux.

Notre seconde contribution est un domaine abstrait pour la description de tableaux, qui utilise des propriétés sémantiques des valeurs contenues afin de partitioner les cellules de tableaux en groupes homogènes. Ainsi, des cellules contenant des valeurs similaires sont décrites par les mêmes prédicats abstraits. De plus, au contraire des analyses de tableaux conventionnelles, les groupes ainsi formés ne sont pas nécessairement contigüs, ce qui contribue à la généralité de l'analyse. Notre analyse peut regrouper des cellules non-congitües, lorsque celles-ci ont des propriétés similaires. Ce domaine abstrait permet de construire des analyses complètement automatiques et capables d'inférer des invariants complexes sur les tableaux.

Notre troisième contribution repose sur une combinaison de cette abstraction des tableaux avec différents domaines abstraits issus de l'analyse de forme des structures de données et reposant sur la logique de séparation. Cette combinaison appelée coalescence opère localement, et relie des résumés pour des structures dynamiques à des groupes de cellules du tableau. La coalescence permet de définir de manière locale des algorithmes d'analyse statique dans le domaine combiné. Nous l'utilisons pour relier notre domaine abstrait pour tableaux et une analyse de forme générique, dont la tâche est de décrire des structures chaînées. L'analyse ainsi obtenue peut vérifier à la fois des propriétés de sûreté et des propriétés de correction fonctionnelle.

De nombreux programmes bas-niveau stockent des structures ynamiques chaînées dans des tableaux afin de n'utiliser que des zones mémoire allouées statiquement. La vérification de tels programmes est difficile, puisqu'elle nécessite à la fois de raisonner sur les tableaux et sur les structures chaînées. Nous construisons une analyse statique reposant sur ces trois contributions, et permettant d'analyser avec succés de tels programmes. Nous présentons des résultats d'analyse permettant la vérification de composants de systèmes d'exploitation et pilotes de périphériques.

Mots Clés

Analyse statique, interprétation abstraite,
structures de données complexes, abstraction de tableaux

Abstract

We study the static analysis on both numeric and structural properties of array contents in the framework of abstract interpretation.

Since arrays are ubiquitous in most software systems, and software defects related to mis-uses of arrays are hard to avoid in practice, a lot of efforts have been devoted to ensuring the correctness of programs manipulating arrays. Current verification of these programs by static analysis focuses on numeric content properties. However, in some low-level programs (like embedded systems or real-time operating systems), arrays often contain structural data (e.g., lists) without using dynamic allocation. In this manuscript, we present a series of techniques to verify both numeric and structural properties of array contents.

Our first technique is used to describe properties of numerical stores with optional values (i.e., where some variables may have no value) or sets of values (i.e., where some variables may store a possibly empty set of values). Our approach lifts numerical abstract domains based on common linear inequality into abstract domains describing stores with optional values and sets of values. This abstraction can be used in order to analyze languages with some form of option scalar type. It can also be applied to the construction of abstract domains to describe complex memory properties that introduce symbolic variables, e.g., in order to summarize unbounded memory blocks like in arrays.

Our second technique is 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. Compared to conventional array partitioning analyses that split arrays into contiguous partitions to infer properties of sets of array cells. Our analysis can group together non-contiguous cells when they have similar properties. Our abstract domain can infer complex array invariants in a fully automatic way.

The third technique is used to combine different shape domains. This combination locally ties summaries in both abstract domains and is called a coalesced abstraction. Coalescing allows to define efficient and precise static analysis algorithms in the combined domain. We utilize it to combine our array abstraction (i.e., our second technique) and a shape abstraction which captures linked structures with separation logic-based inductive predicates. The product domain can verify both safety and functional properties of programs manipulating arrays storing dynamically linked structures, such as lists.

Storing dynamic structures in arrays is a programming pattern commonly used in low-level systems, so as to avoid relying on dynamic allocation. The verification of such programs is very challenging as it requires reasoning both about the array structure with numeric indexes and about the linked structures stored in the array. Combining the three techniques that we have proposed, we can build an automatic static analysis for the verification of programs manipulating arrays storing linked structures. We report on the successful verification of several operating system kernel components and drivers.

Keywords

Static analysis, abstract interpretation,
complex data structures, array abstraction