Shape abstractions with support for sharing and disjunctions

Huisong Li

▶ To cite this version:

Huisong Li. Shape abstractions with support for sharing and disjunctions. Programming Languages [cs.PL]. Université Paris sciences et lettres, 2018. English. NNT : 2018PSLEE060 . tel-01963082v2

HAL Id: tel-01963082

https://theses.hal.science/tel-01963082v2

Submitted on 19 Feb 2020

THÈSE DE DOCTORAT

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

Préparée à l'Ecole Normale Supérieure

Abstractions de la Forme des Structures de Données Supportant Partage et Disjonctions

Shape Abstractions with Support for Sharing and Disjunctions

Ecole doctorale n°386

Ecole doctorale de Sciences Mathématiques de Paris Centre

Spécialité Informatique

COMPOSITION DU JURY :

M. VAFEIADIS Viktor
MPI-SWS, Rapporteur

M. RINETZKY Noam
Tel aviv university, Rapporteur

Mme. BLAZY Sandrine
University of Rennes 1, President du jury

M. ZAPPA NARDELLI Francesco
ENS & INRIA, Membre du jury

M. CHANG Bor Yuh Evan
University of Colorado Boulder,
Membre du jury

Mme. DRAGOI Cezara
ENS & INRIA, Membre du jury

M. RIVAL Xavier
Ens & INRIA, Membre du jury

Soutenue par
Huisong **LI**
le 8 mars 2018

Dirigée par Xavier **RIVAL**

Résumé

L'analyse statique des programmes permet de calculer automatiquement des propriétés sémantiques valides pour toutes les exécutions. En particulier, dans le cas des programmes manipulant des structures de données complexes en mémoire, l'analyse statique peut inférer des invariants utiles pour prouver la sûreté des accès à la mémoire ou la préservation d'invariants structurels. Beaucoup d'analyses de ce type manipulent des états mémoires abstraits représentés par des conjonctions en *logique de séparation* dont les prédicats de base décrivent des blocs de mémoire atomiques ou bien résument des régions non-bornées de la mémoire telles que des listes ou des arbres. De telles analyses utilisent souvent des disjonctions finies d'états mémoires abstraits afin de mieux capturer leurs dissimilarités. Les analyses existantes permettent de raisonner localement sur les zones mémoires mais présentent les inconvénients suivants:

(1) Les prédicats inductifs ne sont pas assez expressifs pour décrire précisément toutes les structures de données dynamiques, du fait de la présence de pointeurs vers des parties arbitraires (i.e., non-locales) de ces structures;

(2) Les opérations abstraites correspondant à des accès en lecture ou en écriture sur ces prédicats inductifs reposent sur une opération matérialisant les cellules mémoires correspondantes. Cette opération de matérialisation crée en général de nouvelles disjonctions, ce qui nuit à l'impératif d'efficacité. Hélas, les prédicats exprimant des contraintes de structure locale ne sont pas suffisants pour déterminer de façon adéquate les ensembles de disjonctions devant être fusionnés, ni pour définir les opérations d'union et d'élargissement d'états abstraits.

Cette thèse est consacrée à l'étude et la mise au point de prédicats en logique de séparation permettant de décrire des structures de données dynamiques, ainsi que des opérations abstraites afférentes. Nous portons une attention particulière aux opérations d'union et d'élargissement d'états abstraits. Nous proposons une méthode pratique permettant de raisonner globalement sur ces états mémoires au sein des méthodes existantes d'analyse de propriétés structurelles et autorisant la fusion précise et efficace de disjonctions.

Nous proposons et implémentons une abstraction structurelle basée sur les variables d'ensembles qui lorsque elle est utilisée conjointement avec les prédicats inductifs permet la spécification et l'analyse de propriétés structurelles globales. Nous avons utilisé ce domaine abstrait afin d'analyser une famille de programmes manipulant des graphes représentés par liste d'adjacence.

Nous proposons un *critère sémantique* permettant de fusionner les états mémoires abstraits similaires en se basant sur leur *silhouette*, cette dernière représentant certaines propriétés structurelles globales vérifiées par l'état correspondant. Les silhouettes s'appliquent non seulement à la fusion de termes dans les disjonctions d'états mémoires mais également à l'affaiblissement de conjonctions de prédicats de logique de séparation en prédicats inductifs. Ces contributions nous permettent de définir des opérateurs d'union et d'élargissement visant à préserver les disjonctions requises pour que l'analyse se termine avec succès. Nous avons implémenté ces contributions au sein de l'analyseur MemCAD et nous en avons évaluées l'impact sur l'analyse de bibliothèques existantes écrites en C et implémentant différentes structures de données, incluant des listes doublement chaînées, des arbres rouge-noir, des arbres AVL et des arbres "splay". Nos résultats expérimentaux montrent que notre approche est à même de contrôler la taille des disjonctions à des fins de performance sans pour autant nuire à la précision de l'analyse.

Abstract

Shape analyses rely on expressive families of logical properties to infer complex structural invariants, such that memory safety, structure preservation and other memory properties of programs dealing with dynamic data structures can be automatically verified. Many such analyses manipulate abstract memory states that consist of *separating conjunctions* of basic predicates describing atomic blocks or summary predicates that describe unbounded heap regions like lists or trees using inductive definitions. Moreover, they use finite disjunctions of abstract memory states in order to take into account dissimilar shapes. Although existing analyses enable local reasoning of memory regions, they do, however, have the following issues:

(1) The summary predicates are not expressive enough to describe precisely all the dynamic data structures. In particular, a fairly large number of data structures with *unbounded* sharing, such as graphs, cannot be described inductively in a local manner;

(2) Abstract operations that read or write into summaries rely on materialization of memory cells. The materialization operation in general creates new disjunctions, yet the size of disjunctions should be kept small for the sake of efficiency. However, local predicates are not enough to determine the right set of disjuncts that should be clumped together and to define precise abstract join and widen operations.

In this thesis, we study separating conjunction-based shape predicates and the related abstract operations, in particular, abstract joining and widening operations that lead to critical modifications of abstract states. We seek a lightweight way to enable some global reasoning in existing shape analyses such that shape predicates are more expressive for abstracting data structures with unbounded sharing and disjuncts can be clumped precisely and efficiently.

We propose a shape abstraction based on set variables that when integrated with inductive definitions enables the specification and shape analysis of structures with unbounded sharing. We implemented the shape analysis domain by combining a separation logic-based shape abstract domain of the MemCAD analyzer and a set abstract domain, where the set abstractions are used to track unbounded pointer sharing properties. Based on this abstract domain, we analyzed a group of programs dealing with adjacency lists of graphs.

We design a general *semantic criterion* to clump abstract memory states based on their *silhouettes* that express global shape properties, i.e., clumping abstract states when their silhouettes are similar. Silhouettes apply not only to the conservative union of disjuncts but also to the weakening of separating conjunctions of memory predicates into inductive summaries. Our approach allows us to define union and widening operators that aim at preserving the case splits that are required for the analysis to succeed. We implement this approach in the MemCAD analyzer and evaluate it on real-world C libraries for different data structures, including doubly-linked lists, red-black trees, AVL-trees and splay-trees. The experimental results show that our approach is able to keep the size of disjunctions small for scalability and preserve case splits that takes into account dissimilar shapes for precision.

Acknowledgments

My first thank goes to my advisor, Xavier Rival, for giving me the chance to do my PhD in France and work on very challenging and interesting topics. He always gave me great advice on research, paper writing, presentation, programming and so on, and was always kind and generous towards me.

I want to express my most sincere gratitude to my reviewers for accepting the time-consuming task of reading this manuscript. If I did my job right, it should be easier to read than it was to write. I'd like to extend my thanks to the other members of my PhD jury, who did me the honor of accepting to come from far to listen to me. Moreover, I want to thank Richard James, who read my thesis and gave me a lot of useful comments to help me to improve my English writing.

I also want to warmly thank my co-authors: Francois Beranger, Arlen Cox and Bor-Yuh Evan Chang for many great discussions, collaboration, and for their help. In particular, thanks to Francois for all the funny jokes we had in our office, for helping me deal with many administrative papers and for organizing many wine parties.

Next, I thank to all the people in my group: Jiangchao Liu, Cheng Tie, Antoine Toubhans, Hugo Illous, Francois Beranger, Arlen Cox, Pippijn Van Steenhoeven, Yoonseok Ko, Changhee Park for all the group meetings that helped me improve my papers, presentations and research ideas. In particular, I would like to thank Jiangchao for being helpful on uncountably many times during my PhD, and as we were doing our PhD at the same time, I was almost never alone in the office.

I am also very grateful to all the people in my lab including: Jerome Feret, Antoine Mine, Cezara Dragoi, Vincent Danos, Caterina Urban, Thibault Suzanne, Mehdi Bouaziz, Ferdinanda Camporesi, Lý Kim Quyên, Stan Le Cornec, Nicolas Behr, Andrea Beica, Marc Chevalier for all the help, discussions, parties and wonderful lunch times. Thanks to Caterina, Arlen, Jiangchao for watching many nice movies together when I was alone in Paris, Ferdinanda for being a nice friend and teaching me how to make pizza, and Jerome Feret for his invaluable help.

I am very grateful to all my friends who supported me in my effort toward this PhD, in particular, Chengqing Chen for the wonderful time we had when we shared a flat, Kailiang Ji who went on a nice travel with me when I was very depressed and Chao Wang, Lili Xu, Teng Long and Ye Jin who were always nice and helped me a lot.

Finally, I would like to thank all my family for supporting me throughout my studies, especially my boyfriend, Ilias Garnier, for all your love, support, understanding and tolerance, and for building a nice home with me.

Contents

Part I

Introduction to Shape Analysis

Chapter 1

Introduction

This chapter presents the main approaches to shape analysis and introduces the motivations of this thesis. First, the benefits and problems brought by pointers and dynamic data structures in programming languages and critical software are introduced in Section 1.1 and Section 1.2 respectively. Then, in Section 1.3, several methods that can reduce software problems or prove the absence of some software problems are presented. Section 1.4 mainly focuses on static analyses targeting memory properties, including pointer analyses and shape analyses. Finally, Section 1.5 and Section 1.6 discuss some open challenges in shape analysis and the contributions of the thesis.

1.1 Dynamic Memory Allocation and Dynamic Data Structures

Pointers and dynamic data structures are often found in low-level operating system codes, device drivers and other safety critical computer controlled systems in transportation and communication. The main reason is that pointers and dynamic data structures allow efficient memory management and access. Let us begin by reviewing common uses of pointers and dynamic data structures.

Pointers. Pointers are values denoting the address of a memory cell in programming languages, like C, C++ and Java. Therefore, pointers are often used as references to program variables, dynamically allocated memory blocks, and functions. As an example, the following C program defines an integer variable `i` and an integer pointer `p` that points-to `i`. A pointer can be dereferenced to read the value of the memory it refers to and also to write into the memory. For example, the program statement at line 3 is equivalent to the statement `i` = `i` + 1 as dereferencing `p` (that is `*p`) on the right side of the assignment will produce the value of `i`, while, on the left side, the dereference will produce the address `&i`.

```
1 int i = 3;
2 int * p = &i;
3 *p = *p + 1;
```

Pointers contribute to the expressive power of a programming language. For example, in C, pointers enable unbounded data structures and call-by-reference. In C, functions use call-by-value to pass arguments. A function call works on a copy of the input parameters, which means that any changes to the parameters inside the function have no effect on the original arguments. However, if a function uses pointers as parameters, the function can then modify the structure pointed to by the parameters through dereferencing pointers.

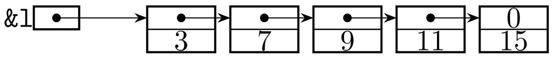
Moreover, pointers can also improve the performance of a programming language, since copying a pointer is often much cheaper both in time and space than copying the data it refers to.

Dynamic Memory Allocation. Dynamic memory management is a key use of pointers. In C and C++, programmers can dynamically allocate memory on the heap when necessary by using some library functions, like `malloc` in C, and `new` in C++. Dynamically allocated memory can only be read and written through pointers. For example, the following C program uses `malloc`() to dynamically allocate a memory area able to hold 10 integers and uses the pointer `p` referring to the beginning of the memory area. If the allocation fails (for instance, when not enough memory is available), `malloc` will return `NULL`. Therefore, the program performs a null check on the pointer `p` at line 3. If the allocation succeeds, the program then initializes the allocated area with 0 by relying on pointer arithmetic and pointer dereferences at line 5.

```
1 int * p = (int *) malloc (10 * sizeof(int));
2 int i = 0;
3 if(p == NULL) return;
4 while(i<10){
5   *(p+i) = 0;
6   i++;
7  }
```

Dynamic Data Structures. In many programs, pointers and dynamic memory allocation are used to implement dynamic data structures. Dynamic data structures refer to organizations of data elements in memory where data elements, called nodes, are stored in dynamically allocated memory blocks and are linked by pointers. Dynamic data structures play a big role in programming languages like C, C++ and Java, because they allow programmers to dynamically adjust the memory consumption according to the data size. This way, many dynamic data structures support very efficient insertion, deletion and search operations.

Linked lists and binary search trees are commonly used dynamic data structures. Linked lists store data nodes in a linear order. Singly linked lists are the most simple list data structures, where each node is made up of some data fields and a pointer to the next node in the sequence. For example, the following memory block contains a singly linked list pointed to by variable `l`.



Each list node contains two fields: an integer value and a pointer to the next node. Specifically, the last list node contains a null (0) pointer to specify the end of the list. Singly linked lists

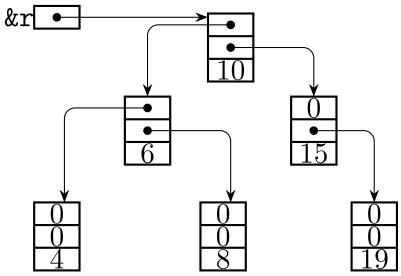


Figure 1.1: A binary search tree

can easily support node insertion and deletion operations with $O(1)$ time. However, a singly linked list can only be searched or traversed in one direction, from head to tail. In order to allow for more efficient algorithms, additional pointers are often added to list nodes. For instance, in doubly linked lists, each node contains two pointers: a pointer to the next node and a pointer to the previous node. In circular linked lists, the next pointer of the last list node points back to the head of the list instead of being null. Moreover, lists are often used to implement other more complex data structures, like, stacks, queues, dynamic arrays, adjacency lists and skip lists.

However, one of the primary disadvantages of linked lists is that searching for a list element usually takes $O(n)$ time. Binary search trees overcome this problem. A binary search tree node is usually composed of two pointers (respectively for the left child node and the right child node) and a data field. The data of the left sub-tree must be smaller than the current node, and that of the right sub-tree must be greater than the current node. As an example, Figure 1.1 shows a memory area containing a binary search tree, whose root is pointed to by pointer `r`. Search, insertion and deletion operations of binary search trees take $O(\log n)$ time on average (where n is the number of tree nodes). However, in the worst case when the tree is not well balanced, these operations take $O(n)$ time complexity. To further improve the worst case efficiency of binary search trees, self-balancing binary search trees, like red-black trees and AVL trees, and self-adjusting binary search trees, like splay trees, have been proposed.

Overall, compared to static data structures, like arrays, dynamic data structures are more flexible. Dynamic data structures support more efficient addition, deletion and search operations and allow more efficient use of memory.

1.2 Software Problems Involving Pointers and Dynamic Data Structures

On the one hand, pointers and dynamic data structures are useful as they allow flexible and efficient memory management and data organization. On the other hand, pointers and dynamic data structures may be dangerous as they can cause very severe software problems, including memory safety problems, memory leaks, and security issues. Let us review these problems.

```
struct list{
  int *data;
  struct list *next;
};
list* list_min_remove(struct list *hd){
  list * pre_min, *min, *c;
  c = hd;
  min = hd;
  while(c->next != NULL){
    if(*(c->next->data) < *(min->data)){
      pre_min = c;
      min = c->next;
    }
    c = c->next;
  }
  if(pre_min != NULL)
    pre_min->next = min->next;
  else
    hd = hd->next;
  free(min->data);
  free(min);
  return hd;
}
```

Figure 1.2: A minimum element deletion function of a singly linked list

1.2.1 Memory Safety Problems

Basic memory safety problems. Memory safety problems are due to memory access errors, including dereferencing null, dangling or uninitialized pointers, and illegal freeing of already freed or non dynamically allocated memory areas. As an example, the following C program may lead to a null dereference error when the function **malloc** returns NULL. This may happen when there is no available memory.

```
int * p = (int *) malloc (sizeof(int));
*p = 1;
```

Memory safety problems often cause runtime errors. In most programming languages, including C, dereferencing null pointers will make the program crash. Dereferencing dangling pointers may cause unpredictable behaviors, such as the corruption of unrelated data, a segmentation fault or an unexpected output, because dangling pointers refer to memory which has already been deallocated and the memory may now contain completely different data. In C, the default value of a pointer is undefined. Therefore, dereferencing an uninitialized pointer can also lead to erratic program behaviors. Moreover, only dynamically allocated memory can be freed through pointers. Double freeing of a dynamically allocated memory block or freeing of a non dynamically allocated memory region will cause a segmentation fault and also result in a crash of the program.

Data structure preservation. However, finding memory safety problems or proving certain memory safety properties of programs that manipulate dynamic data structures is often challenging because pointers are used as links in dynamic data structures. Memory safety properties of such programs rely on well formed dynamic data structures.

Let us consider the C program in Figure 1.2 as an example. The program defines a singly linked list type **list**, where each list element contains a `data` field that stores an integer pointer and a `next` pointer. The function `list_min_remove` takes a list head pointer `hd` as an input parameter and aims to remove the list element whose `data` field points to the minimum integer from the list. It first searches for the minimum list element using the **while** loop from line 9 to line 15. Pointers `min`, `pre_min` and `c` respectively store the minimum list element, the previous element and a cursor. Then, it removes the minimum list element from the list (lines 16 to 19) and frees the memory.

However, executing the `list_min_remove` function may lead to dereferencing null, uninitialized or dangling pointers, and illegal freeing.

Null pointer dereference. If the input parameter `hd` is `NULL`, pointer `c` will be initialized to `NULL` at line 7. Then, the dereference `c -> next` at loop head (line 9) will lead to a null pointer dereference error. Even when `hd` is not `NULL`, null pointer dereference may still happen at line 10 where the program dereferences the `data` field of list elements (`*(c -> next -> data)` and `*(min -> data)`), when some `data` fields are equal to `NULL`.

Uninitialized pointer dereference. In function `list_min_remove`, pointer `pre_min` is declared at line 6, but only initialized inside the **while** loop. When the loop is not executed, `pre_min` will stay uninitialized, the default value of which could be arbitrary. Therefore, the uninitialized pointer `pre_min` may pass the null pointer checking at line 16 and lead to an uninitialized pointer dereference error at line 17.

Dangling pointer dereference and illegal freeing. At line 20, the function frees the integer memory cell pointed by `min -> data`. This is fine if we assume that `min -> data` points to a dynamically allocated memory cell and the memory cell is only referenced by pointer `min->data`, but might otherwise cause illegal freeing at line 20 or cause a dangling pointer dereference error in the execution of some following code.

Therefore, in order to prove that a call to `list_min_remove` does not lead to any memory safety issues, we need to prove that the input parameter points to a well formed and non-empty list. That is, the list has at least one node and the `data` field of all the list nodes point to dynamically allocated and mutually disjoint memory cells. In turn, this requires proving the data structures are preserved by the list manipulation functions, like the `list_min_remove` function.

Memory Leaks. A memory leak occurs when a dynamically allocated memory block becomes unreachable or is not released after use. It can be seen as a special memory safety problem. In C and C++, programmers can dynamically allocate a chunk of memory from the system to store data. Dynamically allocated memory can only be accessed through pointers. Therefore, programmers should always make sure that there is at least one pointer to any dynamically allocated memory block until it is freed. Otherwise, the memory block will become unreachable. Moreover, C and C++ do not have built in automatic garbage collection. Programmers are responsible for freeing dynamically allocated memory after using it, otherwise the program may

```
1  struct list{
2    int data;
3    struct list *next;
4  };
5  list* alloc_list(){
6    list * hd = NULL;
7    list * tmp = NULL;
8    int i = 1;
9    while(i < 101){
10     tmp = (list*)malloc(sizeof(list));
11     if(tmp == NULL)
12       return void;
13     else{
14       tmp->next = hd;
15       hd = tmp;
16     }
17     i++;
18   }
19   return hd;
20 }
```

Figure 1.3: A list allocation program with memory leaks

consume much more memory than needed. Normally, unfreed dynamically allocated memory will be returned to the system when the program exits.

Memory leaks can be a major problem as computers and portable devices have a fixed amount of available memory. If a running software uses up all the available memory, any memory allocation operation will fail, which will cause all the processes trying to allocate more memory to terminate or crash. Typically, this can be solved by terminating the leaking software, so that the operating system can clean up the memory. However, this is not very satisfactory, especially for the kernels of operating systems and software in embedded devices and servers which may be left running for many years. Generally, memory leaks are likely to induce a failure of the operating system if they occur in the kernel. The same is true for embedded systems which have no sophisticated memory management.

As a memory leak example, let us consider the C program in Figure 1.3. The `alloc_list` function aims to dynamically allocate a list of length 100 and returns either the head pointer of the list, or `NULL` when the allocation fails. Specifically, it declares two pointers `hd` and `tmp` respectively for the head of the already allocated list and a temporary pointer for the newly allocated element and uses a **while** loop to perform the allocation: each loop iteration allocates one list element at line 10 and sets it as the list head at lines 14 to 15. However, the execution of the `alloc_list` function may cause memory leaks. When the memory allocation at line 10 fails, the function directly returns **void** without freeing all the previously allocated list elements. After the function returns, all the memory blocks allocated inside the function will become unreachable, and thus, can no longer be freed until the program exits.

1.2.2 Security Problems

Memory safety problems may cause not only runtime errors but also serious security problems, such as information leaks and privilege escalation.

Dangling pointer dereference bugs have frequently become security holes and have been known as “use after free” vulnerabilities. After freeing the memory a pointer refers to, the pointer becomes a dangling pointer and should not be dereferenced again as the memory may now contain completely different data. If the memory is used by higher privilege users or applications, writing or reading through a dangling pointer may cause privilege escalation. If the memory is reallocated by attackers and used to store attacker-controlled data, a dangling pointer dereference may cause the execution flow of the program to be controlled by attacker-controlled data. Moreover, if the dangling pointer is a function pointer, some malicious code written by attackers may be called.

A famous example of “use after free” vulnerability is CVE-2012-4969 [oM12], a zero-day Internet Explorer vulnerability, which was found to be exploited in-the-wild in September 2012. The main reason for the vulnerability was due to an incorrect reference count of an object which allowed the attacker to free it while it was in-use. Another example of a use-after-free vulnerability was CVE-2012-4792 [oM13] exploited in-the-wild in December 2012. This vulnerability was also due to an incorrect reference count of an object, which caused a dangling pointer dereference after the object was freed.

Apart from dangling pointer vulnerability, null pointer dereferencing can also be a potential security problem. For example, if a security check program contains a null pointer dereference bug, it may allow the attacker to bypass security checks or reveal valuable debugging information that can be used in subsequent attacks.

1.3 Improving Software Quality Against Memory problems

As memory related software problems can often have serious consequences, it is an important task to improve software quality against memory problems.

1.3.1 Systematic Software Development

Software engineering is an attempt to reduce software problems by following a systematic approach to the development of software. According to the software engineering approach, detailed software requirements specifications and design specifications should be established in order to guide the developers towards its implementation.

Some specific software development guidelines are used for critical software developments. For instance, in order to improve code safety, security, portability and reliability in embedded systems developments, a set of software development guidelines for the C programming language, MISRA C, was drawn up by Motor Industry Software Reliability Association. Nowadays, MISRA C has been widely followed by developers in automotive, telecom, medical devices, and railway systems. As another example, all the commercial software-based aeronautics follow the DO-178C rules. Specifically, DO-178C specifies the software safety level by examining the effects of a failure in the system and requires all software that commands, controls, and monitors safety-critical functions to be certified with respect to its level of criticality.

However, these approaches cannot prevent all bugs in programs. Firstly, programmers are humans and we know that humans can make mistakes. Secondly, critical systems are nowadays quite complex and have millions of lines of code. Code size is the worst enemy of software: as the code size grows, the number of bugs also increases. According to the book [McC93], on average in industry, "there are about 15 - 50 errors per 1000 lines of delivered code". Moreover, software guidelines are written in human languages and are verified manually. Human inspection is not always reliable.

1.3.2 Testing and Code Reviewing

Testing and code reviewing are important steps in software development. Software testing relies on executing software with a set of designed inputs to find software bugs. Code reviewing takes the form of a systematic examination of software source code by programmers. However, testing and code reviewing cannot identify all software problems.

A large software system often has a huge number of branches and loops. Therefore, testing cannot cover all possible executions of branches and any number of loop iterations of large software systems. Besides, testing cannot cover all the possible inputs as software may have a huge of or even infinite number of inputs. For example, we cannot test on all the inputs of a program which takes any length of singly linked lists as parameters. Moreover, memory errors can be very difficult to discover by testing. For certain memory errors, there is often a long delay from the point when a memory error occurs to the point when the program crashes or produces invalid outputs, e.g. memory leaks. Memory errors may also be hard to reproduce, e.g. memory errors due to dangling pointer dereferences or memory errors with intermittent symptoms.

Code reviewing may find some serious memory errors. However, human inspection is not reliable in general.

Both testing and code reviews are very time consuming and expensive. More importantly, they cannot provide a formal guarantee of the absence of program errors and cannot establish that a software program satisfies certain properties under certain conditions.

1.3.3 Formal Verification

Formal verification aims to formally prove that a program is correct with respect to a specification which could state either safety properties or functional properties. One example of a safety specification could be that a program does not contain null pointers and dangling pointer dereference errors. A functional specification could be that a program preserves a list structure and outputs a sorted list for any input list.

In order to formally verify a program, we first need to define a formal language, like a logic-based language or a graphic-based language, which is expressive enough to express program semantics and program specifications. Then, it remains to prove that the program is correct with respect to the formal language. However, proving by hand is impossible for real programs which are huge in size.

In contrast, automated formal verification techniques which aim to automatically produce correctness proofs for programs have made tremendous progress in recent years. Proving the correctness of programs is an undecidable problem [Ric53]: there does not exist a *sound* and *complete* algorithm which returns **true** if and only if the input program is correct. Therefore, designing an automatic sound and complete proof system is impossible. In general, formal verification techniques either rely on human intervention or drop the soundness or the completeness aspects in order to be fully-automated.

Model checking. Originally, model checking [EC80, CES86] aimed to automatically verify correctness properties of finite-state systems. Given a finite-state system, model checking proves that certain properties are satisfied by an exhaustive search of all possible states during all executions of the system.

Applying model checking techniques to program verification will therefore face a state explosion problem due to the infinite state-space of programs. To overcome this problem, many model checking techniques have been proposed for handling infinite state systems, including bounded model checking and model checking based on abstraction. Bounded model checking only considers a finite subset of an infinite state space. Model checking based on abstraction relies on finding a finite abstraction of infinite states. For example, some model checking techniques use tree or forest automata to finitely represent potentially infinite memory states which contain unbounded dynamic data structures. As the problems are undecidable, most model checking work focuses on developing semi-automatic verification algorithms or decidable results for restricted cases.

Automated theorem proving. Another way to prove program properties is to rely on proof assistants. A proof assistant, e.g. the Coq proof assistant[1] or the PVS verification system[2], is a software tool to help develop formal proofs through collaboration between humans and theorem provers. A proof assistant is often composed of a specification language and a semi-automated theorem prover. In order to prove program properties, humans are supposed to write program specifications and guide the theorem prover to search for proofs. However, prover-assisted approaches are not fully automated and human intervention may take a considerable amount of time. For hard problems, proficient users are required. Moreover, when the targeted program is slightly modified by programmers, the original proofs are hard to reuse. Usually, the whole proof process has to be run again.

Abstract interpretation. In the late 1970s, Patrick Cousot, together with Radhia Cousot, proposed abstract interpretation frameworks [CC77, CC92] that allow fully automatic static analyses to be designed. Specifically, an abstract interpretation framework provides a systematic set up that can be used by static analysis designers to derive computable and sound, but not necessarily complete, program analyses. That is, if the analysis returns **true** with respect to a specification, then the analyzed program satisfies the specification. However, not all the programs that satisfy the specification can be verified.

[1] `https://coq.inria.fr`

[2] `http://pvs.csl.sri.com`

In order to design an abstract interpretation based static analysis, one should first identify the set of target programs and the program properties of interest, based on which, one can design an abstract domain. An abstract domain is a lattice over a set of abstract elements. Each abstract element represents a set of concrete elements (program states). For example, one may abstract the concrete values of integer program variables by their signs $(+, -, 0)$. The relationship between abstract elements and concrete elements should be explicitly defined by a concretization function that maps each abstract element to a set of concrete elements.

In abstract interpretation, program semantics are described as abstract transfer functions that allow program semantics to be over-approximated at the abstract level. To compute abstract fix-points for loops and recursive procedures, abstract interpretation relies on a widening operator $\triangledown$ that over-approximates any two abstractions and guarantees termination for any fix-point computation process.

Recently, abstract interpretation based static analysis tools have made dramatic progress. The ASTRÉE analyzer [BCC$^+$03] is able to prove the absence of runtime numerical errors on avionic software [DS07] and spaceship control programs [BCC$^+$10]. Besides numerical analyses, abstract interpretation is also used in proving various memory properties of programs dealing with pointers and dynamic data structures. This will be discussed in the following section.

1.4 Pointer Analyses and Shape Analyses

As critical programs often manipulate pointers and complex dynamic data structures, pointer analyses and shape analyses, respectively aiming at discovering interesting pointer relations and structural properties are required. Section 1.4.1 presents pointer analyses. Then, Section 1.4.2 and Section 1.4.3 respectively introduce three-valued logic and separation logic based shape analyses. Finally, Section 1.4.4 briefly discusses some other graph-based and automata-based shape analyses.

1.4.1 Pointer analyses.

Pointer analyses [LH88, CBC93, And94] aim to build points-to relations (e.g., pointer "`x` points to `y`") or alias relations (e.g., pointers "`x` and `y` are aliased" i.e., they point to the same memory address) for each program point. The results of pointer analyses are often used in other more complex analyses, like escape analyses, construction of call graphs and aggressive compiler optimization.

In order to be more expressive, pointer analyses [JM82, CWZ90, HN90, HHN92] use finer memory models that are closer to the linked data structures manipulated by programs. Based on finer memory models, the analyses are able to establish relations between not only program variables but also memory locations reachable from program variables (e.g. memory location `x -> next -> next`). The analysis proposed in [Deu94] further enriched pointer analyses with numerical linear equalities to track more subtle relations, such as $\mathtt{x}.(\mathtt{next})^n = \mathtt{y}.(\mathtt{next})^n$. To gain more precision, pointer analyses [HL11, WL95, MRR05] of flow-sensitivity (that considers the order of program statements), context-sensitivity (that considers the calling context of function calls), field-sensitivity (that treats all fields of a struct separately) and object-sensitivity (that

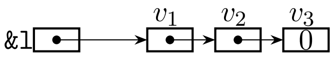


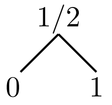
Figure 1.4: A concrete memory block with a singly linked list pointed to by variable `l`

considers the objects on which a method is applied, to determine the set of objects pointed to by reference variables and reference object fields) have been well studied.

While being fast enough to analyze large pieces of code, pointer analyses are not expressive enough to capture complex data structure invariants and to verify memory safety, data structure preservation and other memory properties. Therefore, more expressive shape analyses which aim at inferring precise structural properties of memory states and verifying memory safety and complex functional properties of programs dealing with pointers and dynamic data structures have been proposed.

1.4.2 Three-valued Logic Based Shape Analyses

Three-valued logic based shape analysis [SRW99] (TVLA) is a powerful parametric framework for abstracting concrete memory states using Kleene's 3-valued logic [Kle52]. Three-valued logic has three truth values $(0, 1, 1/2)$, where 0 and 1 are *definite* values and 1/2 is an *indefinite* value. The figure below shows the information order on the truth values, where all *definite* values are smaller than the *indefinite* value, which means a *definite* value has more information than an *indefinite* value.



Abstract memory states. According to TVLA, a shape invariant can be characterized by a set of logical predicates describing pointer variables, types of data structure elements, connectivity properties and so on. As an example, the concrete memory shown in Figure 1.4.2 contains a singly linked list pointed to by pointer `l`. After naming the three list nodes v_1, v_2 and v_3, the concrete memory can be described as the logical formula:

$$\texttt{l}(v_1) \wedge \texttt{next}(v_1, v_2) \wedge \texttt{next}(v_2, v_3) \wedge \texttt{next}(v_3, 0)$$

where, $\texttt{l}(v_1)$ means pointer `l` points to node v_1 and $\texttt{next}(v_1, v_2)$ means the `next` field of v_1 refers to node v_2. However, the set of concrete memory which contains a list is unbounded as the list could be of any length. In order to represent an unbounded set of concrete memory states, TVLA builds abstract states based on 3-valued logic formulas and summary nodes. As an example, the following abstract state abstracts all the concrete memory where `l` points to a singly linked list of at least length 1, including the concrete memory block shown in Figure 1.4.2.

$$\texttt{l}(v_1) = 1 \wedge \texttt{next}(v_1, v_2') = 1/2 \wedge \texttt{next}(v_2', v_2') = 1/2 \qquad \text{where } v_2' \text{ is a summary node}$$

To concretize the abstract state into the concrete memory, we need to concretize the summary node v_2' into nodes v_2 and v_3 and concretize the indefinite value 1/2 into definite values.

In order to perform assignment operations on abstract states, TVLA defines a “focus” operation that refines an abstract state into a set of abstract states, so that formulas related to the assignment only have definite values. In contrast to “focus”, “blur” is an operation that merges nodes into summary nodes, so that abstract states can stay simple and finite.

TVLA is a general framework that can be instantiated in different ways by varying the predicates. It can also be parameterized in the model of concrete memory states for different levels of precision. For example, the fine-grained semantic model [KSV10] is able to precisely present overlapping data structures, while the coarse-grained semantic model leads to a lighter, but less precise analysis.

However, in the TVLA framework, 3-valued logic can not capture disjoint properties of memory cells naturally. This may cause the analysis to produce imprecise results. Moreover, updates on abstract states containing complex predicates, like connectivity predicates or cyclicity predicates, are complex and expensive to compute, which may result in a scalability problem.

1.4.3 Separation Logic Based Shape Analysis

Separation logic. Separation logic [Rey02] was proposed for reasoning about memories as an extension of Hoare logic. The key feature of separation logic is the *separating conjunction* ($*$) which allows us to describe disjoint properties of memory cells. As an example, the concrete memory below contains two memory cells of variables x and l.

&x	3
&l	2

The concrete memory can be described as the separation logic formula:

$$\texttt{\&x} \mapsto 3 * \texttt{\&l} \mapsto 2$$

where, $\texttt{\&x} \mapsto 3$ destcribes the memory cell of variable x, $\texttt{\&l} \mapsto 2$ describes the memory cell of l, and the separating conjunction $*$ of the two predicates specifies that variables x and l are located in memory regions that do not overlap.

Separation logic supports the frame rule below, which plays an important role in enabling local reasoning.

$$\frac{\phi\ P\ \phi'}{\phi * \psi\ P\ \phi' * \psi}$$

The condition of the rule is that the execution of program P only reads or writes memory cells specified in memory state ϕ. It says that if executing program P from a memory state ϕ results in memory state ϕ', then, executing the program in any bigger memory state $\phi * \psi$ will not affect the additional part ψ of the memory state. Based on this rule, we can thus only consider the local effect of a program statement on a minimal sub-memory and derive its global effect on the whole memory.

Abstract memory states. In addition to the separating conjunction, separation logic based shape analyses [BCC$^+$07, CDOY09, DOY06, CR08] often rely on user-provided or built-in inductive definitions to summarize unbounded data structures. An inductive definition is a precise description of a data structure. For example, we can define singly linked lists by induction:

$$\alpha \cdot \mathbf{list} \quad \Longleftrightarrow \quad \alpha = 0 \ \vee \ \alpha \neq 0 \wedge \exists \beta.\ \alpha \mapsto \beta * \beta \cdot \mathbf{list}$$

where, α denotes the address of the list head. The definition says that either α is equal to 0 (null) which indicates the list is empty, or the list contains at least one node described by a points-to predicate $\alpha \mapsto \beta$ (α and β respectively describe the address and the value of the list node) and a list tail described by a recursive call $\beta \cdot \mathbf{list}$. The list tail is located at address β and located in a disjoint memory region with the list head node according to the separating conjunction $*$.

An abstract memory state is thus defined as a separating conjunction combination of atomic heap predicates:

$$\begin{array}{llll}
\text{abstract states:} & \overline{\mathrm{m}}(\in \overline{\mathcal{M}_\omega}) & ::= & \overline{\mathrm{p}} * \ldots * \overline{\mathrm{p}} \\
\text{atomic predicates:} & \overline{\mathrm{p}}(\in \overline{\mathcal{P}}) & ::= & \alpha \mapsto \beta \quad \text{(points-to predicate)} \\
 & & \mid & \alpha \cdot \mathbf{list} \quad \text{(inductive predicate)} \\
 & & \mid & \alpha \cdot \mathbf{tree} \quad \text{(inductive predicate)}
\end{array}$$

An atomic heap predicate is either a points-to predicate $\alpha \mapsto \beta$ representing a single memory cell or an inductive predicate, e.g. $\alpha \cdot \mathbf{list}$, summarizing a memory region. As an example, the concrete memory shown in Figure 1.4.2 can be abstracted by the abstract state below:

$$\exists \alpha_1, .\ \texttt{\&l} \mapsto \alpha_1 * \alpha_1 \cdot \mathbf{list}$$

where, `&l` is the symbolic address of variable `l`, α_1 is the symbolic address of the list head node, and inductive predicate $\alpha_1 \cdot \mathbf{list}$ summarizes the concrete list. Indeed, the abstract state describes an unbounded number of concrete memory regions, where the list is of any length.

Inductive predicates in separation logic based shape analyses play a similar role to summary nodes in TVLA. However, inductive predicates follow a more local approach by using rule-based inductive definitions, while summary-nodes quantify global facts over sets of memory locations. Similarly to the blur and focus operations in TVLA, separation logic based analyses provide fold and unfold operations to generate and materialize inductive predicates respectively.

As the separating conjunction is able to express the disjoint properties of memory cells, strong updates and local reasoning can be performed on memory abstractions based on the separating conjunction. Studies on separation logic based shape analysis have successfully led to many shape analyzers, such as SpaceInvader [DOY06], Predator [DPV11], MemCAD [CR13] and Infer [CDD$^+$15].

1.4.4 Other Shape Analyses

Automata-based shape analyses. Another family of shape analyses [HHR$^+$11, HHL$^+$15] makes use of tree or forest automata to abstract memories with complex dynamic data structures.

Memory abstractions are usually composed of several tree automata, where each automaton represents a separate memory area. Similarly to inductive definitions, user-defined forest automata are used to encode recursive data structures and are used as alphabets in the abstractions to allow unbounded data structures to be represented. Program semantics are thus encoded as operations on automata.

Graph-based shape analyses. Some other shape analyses [GH96, MHKS08] abstract memories as graphs and express program semantics as graph transformations. Specifically, in [GH96] memory areas are abstracted by considering the shape (tree, DAG, or cyclic graph) of the data structure accessible from heap directed pointers, and in [MHKS08], graph nodes are used to represent memory regions and edges are used to represent pointers.

1.5 Outstanding Challenges in Shape Analysis

Shape analysis aims at inferring precise structural properties of memory states. Precision is always a major concern in shape analysis as an analysis can easily fail due to a lack of precision. On the other hand, in order to analyze real-world codes, shape analyses must be able to scale up.

As the separating conjunction make it possible to express the disjoint properties of memory cells, separation logic based shape analyses can reason about memory states locally and can perform strong updates on memory abstractions. Thus, the analyses can efficiently and precisely analyze some programs manipulating singly linked lists and some simple tree data structures.

However, real-world programs often manipulate more complex data structures, e.g. nested data structures, overlaid data structures and data structures with sharing, for which separation logic is not expressive enough to precisely describe the structural properties. To solve this problem, many ideas have been proposed, including various combinations of abstract domains [TCR14, TCR13, SR12, CR08] and abstractions based on a per-field separating conjunction [DES13]. However, these approaches still have significant limitations when it comes to handling data structures with *unstructured sharing*. Section 1.5.1 will discuss the problem in detail and outline an approach that *combines shape predicates with set predicates to express unstructured sharing*.

Moreover, in order to precisely describe possibly dissimilar concrete memory states at a program point, shape analyses often rely on finite disjunctions of abstract memory states. Disjunctions should be kept small for scalability reasons, though precision often requires keeping additional case splits. In this context, deciding when and how to merge disjuncts and to replace them with precise over-approximations is critical both for precision and efficiency. Section 1.5.2 will set out the challenge in disjunction control and present some existing techniques in detail, as well as introducing the *semantic-directed clumping of disjunctive abstract states*.

1.5.1 Reasoning about Sharing in Separation Logic Based Shape Analysis

We say that a data structure is *unshared* if any node in the data structure is at most pointed to by one pointer. On the other hand, when any nodes in a data structure may be referred to by more than one pointer, the data structure is said to be *shared*. Singly-linked lists and

```
typedef struct node{
  struct node * next;
  int id;
  struct edge * edges;
} node;
typedef struct edge{
  struct node * dest;
  struct edge * next;
} edge;
```

Figure 1.5: Type definitions for adjacency lists

binary search trees are *unshared data structures*, while more complex structures, such as doubly linked lists, directed-acyclic graphs (DAGs) and some data structures representing graphs in general, are *shared data structures*. Some shared data structures have regular sharing patterns. Such sharing patterns can be described by using a bounded number of constraints on nodes. For example, doubly-linked lists have *structured sharing*. In doubly linked lists, the sharing occurs because each node is pointed to by both its predecessor (except for the first node) and its successor (except for the last node). Therefore, doubly-linked lists can be described by the following inductive definition:

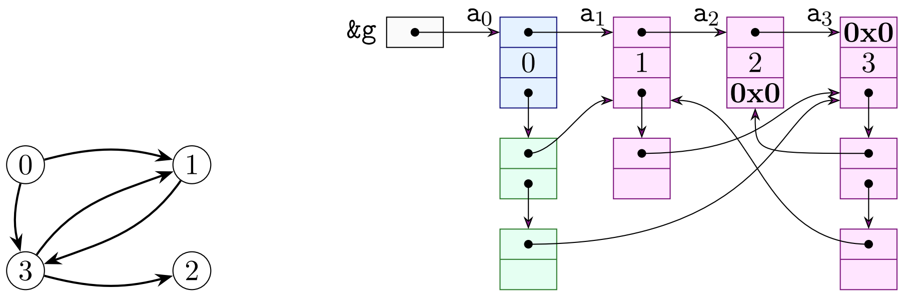
$$\alpha \cdot \mathbf{dll}(\delta) ::= (\mathbf{emp} \wedge \alpha = 0) \vee (\alpha.\mathtt{prev} \mapsto \delta * \alpha.\mathtt{next} \mapsto \beta * \beta \cdot \mathbf{dll}(\alpha) \wedge \alpha \neq 0)$$

This definition says that α points to a doubly-linked list if and only if it is either null (0) or a pointer to a list element: the `prev` field of the element points to δ and the `next` field of the element β points to a doubly-linked list such that the `prev` field of its first element should point back to α itself. As in doubly linked lists, each node has *exactly one* predecessor, which can be specified as the parameter δ of the inductive definition. As shown in paper [CRN07], skip lists can also be defined following the same schema.

However, precisely and recursively summarizing data structures with unstructured sharing using the separating conjunction is much more challenging because in such data structures, a node may be pointed to by an unbounded number of pointers and the pointers could be anywhere in the structure.

In order to make the challenges more concrete, let us consider *adjacency lists* (a data structure for representing graphs) as an example. Figure 1.5 shows a type definition for *adjacency lists*: a graph is a list of nodes, each node has a list of edges, and each edge is a pointer to its destination node. Figure 1.6(b) shows an adjacency list representation of the graph shown in Figure 1.6(a), where each node i in the simple graph is described by a node located at address $\mathtt{a}_i$.

In order to prove memory safety or functional properties of algorithms manipulating the data structure with separation logic based shape analysis techniques, we need to recursively define this data structure using the separating conjunction. A natural approach is to exploit the list-of-lists inductive skeleton of adjacency lists. As shown in Figure 1.7: for the node at $\mathtt{a}_0$, its list of edges can be inductively summarized in the green region, while the other nodes, together with their edge lists in the graph, can be summarized in the purple region. What is implicit in



(a) A simple graph

(b) Adjacency list representation of (a)

Figure 1.6: Example of an adjacency list

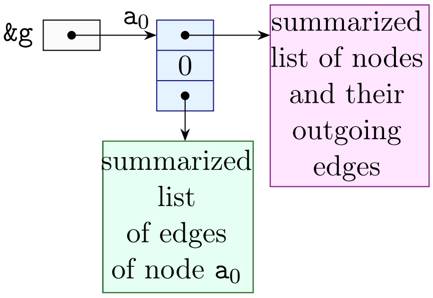

&g
a0
0
summarized list of nodes and their outgoing edges
summarized list of edges of node a0

Figure 1.7: A partially summarized adjacency list

this informal diagram is that, to achieve precision, these summarized regions must also capture complex, unstructured, cross pointer relations (i.e., the curving lines in Figure 1.6(b)).

To capture this unstructured sharing precisely, we observe that the correctness of the structure stems from the fact that each edge pointer points to an address in $\mathscr{E} = \{\mathtt{a}_0, \mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3\}$. Thus, the absence of invalid edges can be captured by adjoining a *set property* to a conventional list predicate. We also need to ensure that all edge pointers point to nodes belonging to the set $\mathscr{E}$ of valid nodes in the graph for each node's edge list. To give an inductive definition for the list of nodes, we need to ensure that this list of nodes is consistent with the set $\mathscr{E}$ of valid nodes, and thus we require a second set variable $\mathscr{F}$ that captures the nodes summarized in the purple region. For the node list summary in Figure 1.7 (shown in purple), this variable $\mathscr{F}$ should be the set $\{\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3\}$.

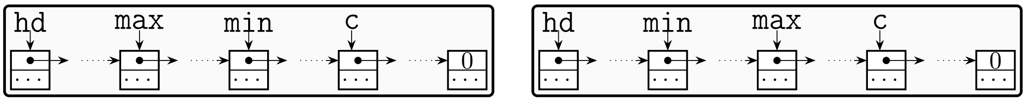
While we have outlined an approach to summarize adjacency lists using a combination of an inductive skeleton and relations over set-valued variables, using such summaries poses significant algorithmic challenges in abstract transfer functions, checking entailment of abstract states and computing over-approximations of abstract states.

1.5.2 Improving Scalability and Disjunction Control

Shape analyses, including TVLA and separation logic based shape analyses, rely on disjunctions of abstract states to capture precise structural properties of memory states. In separation logic

```
1  typedef struct list{
2    struct list *next;
3    int d;
4  } list;
5
6  void search_min_max(struct list *hd){
7   list * min, *max, *c;
8   min = max = c = hd;
9   while( c != NULL){
10    if(c->d < min->d) min = c;
11    if(c->d > max->d) max = c;
12    c = c->next;
13  }
14  printf("min: %d", min->d);
15  printf("max: %d", max->d);
16  return;
17 }
```

Figure 1.8: Searching the minimum and maximum list elements



(a) Concrete memory state 1

(b) Concrete memory state 2

Figure 1.9: Concrete memories with list segments

based shape analysis, disjunctive abstract states can be represented as:

$$\begin{array}{lrcl} \text{disjunctive abstract states:} & \overline{\mathrm{d}}(\in \overline{\mathcal{D}}) & ::= & \overline{\mathrm{m}} \vee \ldots \vee \overline{\mathrm{m}} \\ \text{abstract states:} & \overline{\mathrm{m}}(\in \overline{\mathcal{M}_\omega}) & ::= & \overline{\mathrm{p}} * \ldots * \overline{\mathrm{p}} \end{array}$$

while, in TVLA, disjunctive abstract states are disjunctions of abstract states built on the conjunction $\wedge$.

Disjunctions are necessary as programs can produce very different structures at a program point and it is often impossible to precisely abstract them all into a single abstract state either based on the conjunction $\wedge$ or the separating conjunction $*$. As an example, let us consider the program in Figure 1.8, which searches for the minimum and maximum list elements from any input singly linked list. At each program point of the **while** loop (line 9 to line 13), according to the input list, the program may produce a concrete memory state, where the pointer `min` points to a list element appearing in the list before the element pointed to by `max`, or a concrete memory state, where the pointer `max` points to a list element appearing in the list before the element pointed to by `min`. Figure 1.9 shows a pair of such memory states. Both memory states contain a list pointed to by the variable `hd`, and three "cursors" `min`, `max`, `c` pointing somewhere in that list. The only difference between these two memory states is the order of the cursors `min`, `max`. Shape analyses like [DOY06, CRN07] instantiate a generic list segment predicate

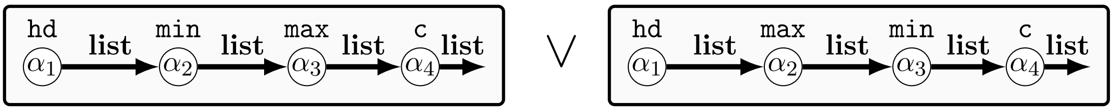


Figure 1.10: A disjunctive abstract state with list segments

$\alpha \cdot \mathbf{list} \mathrel{*\!\!=} \beta \cdot \mathbf{list}$ to abstract a segment of the list starting at α and finishing with a pointer to β. Using this abstraction, the concrete memory state in Figure 1.9(a) can be abstracted by:

$$\begin{aligned}&\exists \alpha_1, \alpha_2, \alpha_3, \alpha_4, \texttt{\&hd} \mapsto \alpha_1 * \texttt{\&max} \mapsto \alpha_2 * \texttt{\&min} \mapsto \alpha_3 * \texttt{\&c} \mapsto \alpha_4 \\ &\qquad\qquad * \alpha_1 \cdot \mathbf{list} \mathrel{*\!\!=} \alpha_2 \cdot \mathbf{list} * \alpha_2 \cdot \mathbf{list} \mathrel{*\!\!=} \alpha_3 \cdot \mathbf{list} * \alpha_3 \cdot \mathbf{list} \mathrel{*\!\!=} \alpha_4 \cdot \mathbf{list} * \alpha_4 \cdot \mathbf{list}\end{aligned}$$

The left side of Figure 1.10 shows a graph representation of the abstract state. However, the abstract state cannot abstract the concrete memory shown in Figure 1.9(b) at the same time as `min,max` appear in the reverse order, which could be abstracted by the abstract state on the right side of Figure 1.10. Therefore, we need the disjunctive abstract state shown in Figure 1.10 to precisely abstract the two concrete memory states.

In practice, disjunctions are a huge challenge to static analyses. While the creation of new disjunctions occurs naturally when the analysis needs to reason about operations that read or write into summary predicates, letting the number of disjuncts grow makes the analysis slower and consumes more memory. However, getting rid of unnecessary disjuncts turns out to be a much harder task than introducing them. To *clump* a disjunctive abstract state $\overline{\mathrm{d}}$, an analysis needs:

1. to *sort* the disjuncts of $\overline{\mathrm{d}}$ into sets of abstract states $\overline{\mathbb{M}}_0, \ldots, \overline{\mathbb{M}}_n$, such that all abstract states in $\overline{\mathbb{M}}_i$ are sufficiently similar;
2. to *compute* for each set $\overline{\mathbb{M}}_i$ an abstract state $\overline{\mathrm{m}}_i$ that conservatively over-approximates all the elements of $\overline{\mathbb{M}}_i$; this weakening should infer how sets of predicates can be folded into summary predicates.

Both steps are critical. The first step should determine the right set of disjuncts: leaving too many disjuncts would make it impossible to scale, whereas excessively reducing the size of the disjunction would prevent the weakening step from producing precise summaries.

Existing approaches all come with limitations and are often challenged by the first step (disjunct sorting). Canonicalization operators [SRW02] solve this problem by using a finite set of "canonical" abstract states, replacing each abstract state with a canonicalized version of it. Although the analysis may use an infinite domain, the precision of canonicalization outputs is limited by that of the finite set of canonical abstract states. The canonicalization operator of [DOY06] and the join operators of [CRN07, YLB^{+}08] utilize local rewriting rules based on the syntax of abstract states. They cannot reason about global shape properties, and thus may miss chances to clump some disjuncts. State partitioning [CC92] and trace partitioning [RM07, HT98] provide frameworks for sensitivity in static analysis, but do not provide a general strategy to choose which disjunctions to preserve. We observe that static strategies based on the control flow structure of programs (conditions, loops, etc.) are likely to produce inadequate disjunct clumps as they ignore the shapes. On the other hand, disjunctive completion [CC79] authorizes any disjunction of abstract states (so that simplification is never required); however, it cannot deal with infinite sets of abstract predicates and is prohibitively costly when the set of abstract states is finite. Pruning disjunctions is thus a major challenge in many memory reasoning tools.

In this thesis, we observe that semantic properties of abstract states can help to characterize the clumping of disjuncts. For instance, the reason why the abstract states in Figure 1.10 cannot be clumped together lies in the order of the pointers `min` and `max`. Indeed, if we let $\rightsquigarrow$ be the relation that states that there is a link path from one pointer to another, then we get a path-based abstraction $\mathtt{hd} \rightsquigarrow \mathtt{min} \rightsquigarrow \mathtt{max} \rightsquigarrow \mathtt{c}$ for the left abstract state and $\mathtt{hd} \rightsquigarrow \mathtt{max} \rightsquigarrow \mathtt{min} \rightsquigarrow \mathtt{c}$ for the right abstract state. Then, a "similarity" check can be performed on the two path-based abstractions, and, based on the checking result, the analysis can decide to not clump the two disjuncts in Figure 1.10.

We have described an approach that uses a path-based "*silhouette*" abstraction of the abstract states themselves to the clumping of disjunctive abstract states. In essence, this technique uses a form of lightweight canonicalization, but only as a guide to decide which disjuncts to clump, whereas the analysis computations (including the abstract join for clumping) all still take place in the initial, infinite lattice.

1.6 Outline and Contributions of the Thesis

This section presents an outline and the contributions of the thesis.

Chapter 2 provides some theoretical background of separation logic based shape analysis, including a simple imperative language and a basic memory abstract domain.

Chapters 4, 5, and 6 propose *shape analysis for unstructured sharing*: a shape analysis that tracks set properties to infer precise invariants of data structures with unstructured sharing.

- Chapter 4 formalizes memory abstractions that are parameterized by *inductive definitions with set-valued parameters* and *set abstractions*, where inductive definitions with set-valued parameters are used to summarize shared data structures and set abstractions are used to reason about relations over set variables.

- Chapter 5 introduces a general interface for set abstract domains and a formalization of a linear set domain used in our shape analysis.

- Chapter 6 proposes static analysis algorithms to infer invariants over data structures with unstructured sharing and then reports on a preliminary empirical evaluation of these algorithms implemented in the MemCAD analyzer.

Chapters 8, 9, 10 and 11 propose *semantic-directed clumping of disjunctive abstract states*: let silhouette abstractions of the abstract states guide the algorithms for *clumping* and *weakening* disjuncts.

- Chapter 8 sets up a path-based silhouette abstraction of abstract states to capture shape similarities and guide the weakening of abstract states.

- Chapter 9 uses silhouettes to guide the joining procedure of abstract memory states such that the procedure is more precise.

- Chapter 10 presents algorithms for the clumping and the widening of disjunctive abstract states based on silhouettes.

- Chapter 11 reports on the implementation of the silhouette-guided algorithms in the MemCAD analyzer and the efficiency assessment with the verification of several real-world C libraries that manipulate structures such as doubly-linked lists, red-black trees, AVL trees, and splay trees.

Finally, Chapter 12 concludes the thesis.

Chapter 2

Separation Logic Based Shape Analysis

This chapter serves as a formal introduction to separation logic based shape analysis. First, Section 2.1 sets out the formal syntax and semantics of a minimal core of C language. Section 2.2 presents some notions of abstract domains in the abstract interpretation framework. Then, Section 2.3 introduces a formal definition for separation logic based memory abstractions and Section 2.4 presents the abstract semantics of the program language. Finally, Section 2.5 sums up exhaustive signatures for the aforementioned abstract domain.

2.1 A Simple Imperative Language

The programs that are considered in this thesis are all written in the C language. For the sake of simplicity, in this section, we define a simple language that intends to establish a model for a fragment of the C language and a concrete denotational semantics for this language.

2.1.1 Syntax

Figure 2.1 presents the syntax of a simple imperative language that provides dynamic memory management and basic control instructions. We avoid types in the language and assume that any value is stored in a 4-byte long cell.

Program variables. We let $\mathcal{X} = \{\mathtt{x}, \mathtt{y}, \ldots\}$ denote a fixed infinite set of program variables, though a given program manipulates only a finite number of variables. Variables must be declared using the *variable declaration* command shown in Figure 2.1 before being manipulated by programs.

L-values and r-values. In our programming language, expressions are of two kinds: left value expressions that correspond to memory addresses and right value expressions that correspond to values. Memory addresses $a \in \mathcal{V}_{\text{addr}}$ ranging from 1 to $2^{32} - 1$ are also memory values $v \in \mathcal{V}$ ranging from 0 to $2^{32} - 1$. To follow the standard C notation, in the following, we present addresses as `0x` followed by the hexadecimal representation of the addresses, e.g., `0x...080`.

Binary operators	$\oplus$	::=		
		|	== | $\neq$ | < | <= | ...	
Left value expressions	`l`	::=		
		|	`x`	*variable*
		|	`*l`	*pointer dereference*
		|	`l.f`	*field access*
Right value expressions	`r`	::=		
		|	n	*integer*
		|	`l`	*l-value read*
		|	`r` $\oplus$ `r`	*binary operation*
		|	`&l`	*address of l-value*
Statements	`p`	::=		
		|	`x`	*variable declaration*
		|	`l` = `r`	*assigment*
		|	`l` = **malloc**(n)	*memory allocation*
		|	**free**(`l`, n)	*memory de-allocation*
		|	**assert**(`r`)	*assertion*
		|	**if**(`r`){`p`}**else**{`p`$'$}	*conditional branch*
		|	**while**(`r`){`p`}	*loop*
		|	`p`; `p`	*sequence*

Figure 2.1: The Syntax of a C-like simple imperative language.

A left value expression can be the address of a program variable `x`, the address `l.f` representing a left value `l` plus an offset `f` $\in \mathcal{F}$, or the address `*l` obtained by dereferencing a left value expression `l`. We note that the expression `l.f` corresponds to the field access operation of struct data in C. In the following, we let `f` $\in \mathcal{F}$ denotes both numerical offset and field name.

Similarly, a right value expression can be an integer value $n \in \mathbb{N}$, the value of the memory cell located at address `l`, denoted by `l`, the address `l` of a memory cell, denoted by `&l`, or a value `r` $\oplus$ `r` obtained by applying a binary operator $\oplus$ to two sub-expressions.

Statements. As shown in Figure 2.1, a program `p` is a sequence of statements that can be of different kinds.

The variable declaration statement `x` declares a variable `x` by allocating a memory cell of size 4-byte for `x`.

The assignment statement `l` = `r` writes the value denoted by right value expression `r` into the memory cell at the address denoted by left value expression `l`.

The memory allocation statement `l` = **malloc**(n) dynamically allocates a memory block of n memory cells and writes the base address of the block into the memory cell at the address denoted by left value expression `l`.

The memory de-allocation statement **free**(`l`, n) de-allocates n memory cells starting at the base address denoted by `l`.

The assertion statement **assert**(`r`) crashes the execution if the value denoted by `r` is equal to 0.

The conditional branch statement **if**(r){p}**else**{p′} computes the value denoted by r and continues with the execution of p if the value of r is not equal to 0, or continues with the execution of p′ otherwise.

The loop statement **while**(r){p} keeps running the program with p if the value of r is not equal to 0 and exits the loop when the value of r is equal to 0.

2.1.2 Concrete States

In order to define a concrete program semantics, we first need to define concrete memory states on which programs are operated.

Memory states. Intuitively, a memory state consists of two parts: an environment that relates symbolic addresses of program variables to their real addresses and a store that is made up of finite number of individual cells. However, an error may appear during the execution of a program, e.g., when a program attempts to read an invalid memory cell. Let ω denote an error state; we thus have the following definition:

Definition 2.1 (Memory states). *A memory state* $\mathrm{m} \in \mathcal{M}_\omega$ *is either an error state* ω *or a non-error state* $(\varepsilon, \sigma) \in \mathcal{M}$ *made up of an environment* $\varepsilon \in \mathcal{E}$ *and a store* $\sigma \in \mathcal{H}$*:*

$$\begin{array}{lll} (\varepsilon,\sigma) \in \mathcal{M} & ::= & \mathcal{E} \times \mathcal{H} \\ \mathrm{m} \in \mathcal{M}_\omega & ::= & \{\omega\} \uplus \mathcal{M} \\ \varepsilon \in \mathcal{E} & ::= & \mathcal{X}_\& \to \mathcal{V}_{\mathrm{addr}} \\ \sigma \in \mathcal{H} & ::= & \mathcal{V}_{\mathrm{addr}} \to \mathcal{V} \end{array}$$

We let $\mathcal{X}_\& = \{\&\boldsymbol{x}, \&\boldsymbol{y}, \ldots\}$ *denote the set of symbolic addresses of program variables. The environment* ε *maps symbolic addresses of program variables into their real addresses in the store. The store* σ *is a partial function from memory addresses* $a \in \mathcal{V}_{\mathrm{addr}}$ *to their values* $v \in \mathcal{V}$*.*

We note that 0 is a null address. We write [ ] for an empty store, $[a_1 \mapsto v_1, \ldots, a_k \mapsto v_k]$ for the store with exactly k allocated cells at addresses $a_1, \ldots, a_k$ and which contains values $v_1, \ldots, v_k$ respectively, $\sigma_1 \uplus \sigma_2$ for the "gluing" of the two stores σ_1 and σ_2 such that the intersection of the addresses of the cells of σ_1 with the addresses of the cells of σ_2 is empty, $\sigma_1 - \sigma_2$ for removing cells whose addresses are in $\mathrm{dom}(\sigma_2)$ from σ_1, and $\sigma[a \mapsto v]$ for substituting $\sigma(a)$ by v.

In order to define a concrete program semantics that manipulates memory states, let us first define several primary operations over stores:

$$\begin{array}{ll} \mathbf{read} & \in \mathcal{V}_{\mathrm{addr}} \times \mathcal{H} \to \mathcal{V} \uplus \{\omega\} \\ \mathbf{write} & \in \mathcal{V}_{\mathrm{addr}} \times \mathcal{V} \times \mathcal{H} \to \mathcal{H} \uplus \{\omega\} \\ \mathbf{create} & \in \mathbb{N} \times \mathcal{H} \to \mathcal{V}_{\mathrm{addr}} \times \mathcal{H} \uplus \{0\} \\ \mathbf{delete} & \in \mathcal{V}_{\mathrm{addr}} \times \mathbb{N} \times \mathcal{H} \to \mathcal{H} \uplus \{\omega\} \end{array}$$

The formal characterizations of the operations are shown in Figure 2.1.2. Intuitively, $\mathbf{read}(a, \sigma)$ reads the value $\sigma(a)$ of a memory cell at address a in the store σ. In the case where a is not a valid address, i.e., $a \notin \mathrm{dom}(\sigma)$, $\mathbf{read}(a, \sigma)$ returns a memory error ω. Similarly, $\mathbf{write}(a, v, \sigma)$ writes value v into the cell at address a of the store σ and returns a memory error ω when

$$\mathbf{read}(a,\sigma) ::= \begin{cases} \sigma(a) & \textit{if } a \in \mathrm{dom}(\sigma) \\ \omega & \textit{otherwise} \end{cases}$$

$$\mathbf{write}(a,v,\sigma) ::= \begin{cases} \sigma[a \mapsto v] & \textit{if } a \in \mathrm{dom}(\sigma) \\ \omega & \textit{otherwise} \end{cases}$$

$$\begin{array}{lll} \multicolumn{3}{l}{\mathbf{create}(n,\sigma)\ \textit{returns}} \\ & \textit{either} & (a, \sigma \uplus [a \mapsto v_1, \ldots, a+4(n-1) \mapsto v_n]) \\ & & \qquad \textit{such that } a, \ldots, a+4(n-1) \in \mathcal{V}_{\mathrm{addr}} - \mathrm{dom}(\sigma) \\ & \textit{or} & 0 \end{array}$$

$$\mathbf{delete}(a,n,\sigma) ::= \begin{cases} \sigma - [a \mapsto v_1, \ldots, a+4(n-1) \mapsto v_n] & \textit{if } a, \ldots, a+4(n-1) \in \mathrm{dom}(\sigma) \\ \omega & \textit{otherwise} \end{cases}$$

Figure 2.2: Characterizations of store operations

a is not a valid address. The functions **create** and **delete** are used to dynamically allocate and de-allocate memory cells respectively. Specifically, $\mathbf{create}(n, \sigma)$ attempts to allocate n continuous cells and adds them into the store σ. If the allocation succeeds, it returns the base address of the newly allocated block that contains n cells and the new store, and if the allocation fails to find enough space, it returns the null address 0. We note that this operation is non-deterministic as the location and the initial value of the newly allocated block depends on the implementation. Conversely, $\mathbf{delete}(a, n, \sigma)$ removes n continuous cells starting from address a in the store σ. It returns memory error ω if there exists an invalid address in the set of addresses $\{a, a+4, \ldots, a+4(n-1)\}$.

Limitations and discussion. For the sake of conciseness, we limit the complexity of our concrete memory model:

- We make no distinction between the stack region and the heap region. Thus, memory errors, such as illegal freeing of non-dynamically allocated memory blocks, do not show up in our memory model. However, our memory model can be easily extended to take these errors into account, for instance, one can augment a store σ with a set of dynamically allocated addresses or augment each address in the store with an attribute indicating a stack or heap cell.
- Our memory states are untyped. That is, we do not distinguish between pointer values and integers and assume all values, all addresses, and all memory cells are four bytes long. This can be extended by adding an environment that maps program variables to their types and a **sizeof** function that returns the size of any given type.

However, our memory model is rather low-level thereby allowing us to target at memory properties such as the absence of dereferencing of null pointers and dangling pointers, data structure preservation, and so on.

$$
\begin{array}{lcl}
\mathbf{eval}_l[\![\mathtt{l}]\!](\omega) & = & \omega \qquad\qquad (\omega{-}\mathit{strict}) \\
\mathbf{eval}_l[\![\mathtt{x}]\!](\varepsilon,\sigma) & = & \begin{cases} \varepsilon(\mathtt{x}) & \mathit{if}\ \mathtt{x} \in \mathrm{dom}(\varepsilon) \wedge \varepsilon(\mathtt{x}) \in \mathrm{dom}(\sigma) \\ \omega & \mathit{otherwise} \end{cases} \\
\mathbf{eval}_l[\![\mathtt{*l}]\!](\varepsilon,\sigma) & = & \mathbf{eval}_r[\![\mathtt{l}]\!](\varepsilon,\sigma) \\
\mathbf{eval}_l[\![\mathtt{l.f}]\!](\varepsilon,\sigma) & = & \begin{cases} \mathbf{eval}_l[\![\mathtt{l}]\!](\varepsilon,\sigma) + \mathtt{f} & \mathit{if}\ \mathbf{eval}_l[\![\mathtt{l}]\!](\varepsilon,\sigma) \neq \omega \\ \omega & \mathit{otherwise} \end{cases}
\end{array}
$$

(a) Semantics of left value expressions $\mathbf{eval}_l[\![\mathtt{l}]\!]$

$$
\begin{array}{lcl}
\mathbf{eval}_r[\![\mathtt{r}]\!](\omega) & = & \omega \qquad\qquad (\omega{-}\mathit{strict}) \\
\mathbf{eval}_r[\![n]\!](\varepsilon,\sigma) & = & n \\
\mathbf{eval}_r[\![\mathtt{\&l}]\!](\varepsilon,\sigma) & = & \mathbf{eval}_l[\![\mathtt{l}]\!](\varepsilon,\sigma) \\
\mathbf{eval}_r[\![\mathtt{r}_1 \oplus \mathtt{r}_2]\!](\varepsilon,\sigma) & = & [\![\oplus]\!](\mathbf{eval}_r[\![\mathtt{r}_1]\!](\varepsilon,\sigma), \mathbf{eval}_r[\![\mathtt{r}_2]\!](\varepsilon,\sigma)) \\
\mathbf{eval}_r[\![\mathtt{l}]\!](\varepsilon,\sigma) & = & \begin{cases} \mathbf{read}(\mathbf{eval}_l[\![\mathtt{l}]\!](\varepsilon,\sigma),\sigma) & \mathit{if}\ \mathbf{eval}_l[\![\mathtt{l}]\!](\varepsilon,\sigma) \neq \omega \\ \omega & \mathit{otherwise} \end{cases}
\end{array}
$$

(b) Semantics of right value expressions $\mathbf{eval}_r[\![\mathtt{r}]\!]$

Figure 2.3: Semantics of left and right value expressions

2.1.3 Concrete Semantics

In this section, we define a denotational semantics of programs, which focuses on constructing all the effects of programs on concrete memory states.

Semantics of operators. The semantics $[\![\oplus]\!]$ of an operator $\oplus$ (e.g., $+, -$) defined below usually denotes the mathematical meaning of the operator:

$$[\![\oplus]\!] \in \mathcal{V} \uplus \{\omega\} \times \mathcal{V} \uplus \{\omega\} \to \mathcal{V} \uplus \{\omega\}$$

For example, $[\![+]\!](2,4) = 6$. We note that, some inputs of an operator may lead to an error state ω, e.g., $[\![/]\!](2,0) = \omega$. Moreover, $[\![\oplus]\!]$ is $\omega-$strict, i.e., $[\![\oplus]\!]$ propagates the error state ω when the error state ω is an input.

Semantics of left and right value expressions. Figure 2.3 presents the formal definitions of the semantics of left and right value expressions. The semantics $\mathbf{eval}_l[\![\mathtt{l}]\!] \in \mathcal{M}_\omega \to \mathcal{V}_{\mathrm{addr}} \uplus \{\omega\}$ of a left value expression `l` is a $\omega-$strict function that propagates the error state $\omega \in \mathcal{M}_\omega$ and maps a non-error memory state $(\varepsilon,\sigma) \in \mathcal{M}$ either to an address denoted by `l` in the given state (ε,σ) or to an error state ω. Similarly, the semantics $\mathbf{eval}_r[\![\mathtt{r}]\!] \in \mathcal{M}_\omega \to \mathcal{V} \uplus \{\omega\}$ of a right value expression `r` is also a $\omega-$strict function that propagates the error state and maps a non-error memory state $(\varepsilon,\sigma) \in \mathcal{M}$ either to a value denoted by `r` in the given state or to an error state ω. The error state ω may appear both in mathematical operations and in store operations.

Denotational semantics. The denotational semantics $[\![\mathtt{p}]\!] \in \mathcal{P}(\mathcal{M}_\omega) \to \mathcal{P}(\mathcal{M}_\omega)$ of program statements is presented in Figure 2.4. Intuitively, in the denotational semantics, programs are described as functions that map sets of initial memory states to sets of final memory states at the end of the execution of the program. The semantics is defined in a standard way by

$$\begin{aligned}&\mathbf{guard}\llbracket \mathtt{r} \rrbracket(\mathbb{M}) \ = \\ &\quad \{(\varepsilon,\sigma) \mid (\varepsilon,\sigma) \in \mathbb{M} \wedge \mathbf{eval}_r\llbracket \mathtt{r} \rrbracket(\varepsilon,\sigma) \notin \{0,\omega\}\} \cup \{\omega \mid \omega \in \mathbb{M} \vee (\varepsilon,\sigma) \in \mathbb{M}, \mathbf{eval}_r\llbracket \mathtt{r} \rrbracket(\varepsilon,\sigma) = \omega\}\end{aligned}$$

Variable declaration:

$$\begin{aligned}&\llbracket \mathtt{x} \rrbracket(\mathbb{M}) \ = \\ &\quad \{\omega \mid \omega \in \mathbb{M} \vee \exists(\varepsilon,\sigma) \in \mathbb{M}, \mathtt{x} \in \mathrm{dom}(\varepsilon) \vee 0 = \mathbf{create}(1,\sigma)\} \\ &\cup \{(\varepsilon \uplus [\mathtt{x} \mapsto a_1], \sigma_1) \mid (\varepsilon,\sigma) \in \mathbb{M} \wedge \mathtt{x} \notin \mathrm{dom}(\varepsilon) \wedge (a_1,\sigma_1) = \mathbf{create}(1,\sigma)\}\end{aligned}$$

Assignment:

$$\begin{aligned}&\llbracket \mathtt{l} = \mathtt{r} \rrbracket(\mathbb{M}) \ = \\ &\quad \{\omega \mid \omega \in \mathbb{M} \vee \exists(\varepsilon,\sigma) \in \mathbb{M}, \omega = \mathbf{eval}_l\llbracket \mathtt{l} \rrbracket(\varepsilon,\sigma) \vee \omega = \mathbf{eval}_r\llbracket \mathtt{r} \rrbracket(\varepsilon,\sigma)\} \\ &\cup \{\omega \mid \exists(\varepsilon,\sigma) \in \mathbb{M}, a = \mathbf{eval}_l\llbracket \mathtt{l} \rrbracket(\varepsilon,\sigma) \wedge v = \mathbf{eval}_r\llbracket \mathtt{r} \rrbracket(\varepsilon,\sigma) \wedge \omega = \mathbf{write}(a,v,\sigma)\} \\ &\cup \{(\varepsilon,\sigma_1) \mid (\varepsilon,\sigma) \in \mathbb{M}, a = \mathbf{eval}_l\llbracket \mathtt{l} \rrbracket(\varepsilon,\sigma) \wedge v = \mathbf{eval}_r\llbracket \mathtt{r} \rrbracket(\varepsilon,\sigma) \wedge \sigma_1 = \mathbf{write}(a,v,\sigma)\}\end{aligned}$$

Allocation:

$$\begin{aligned}&\llbracket \mathtt{l} = \mathbf{malloc}(n) \rrbracket(\mathbb{M}) \ = \\ &\quad \{\omega \mid \omega \in \mathbb{M} \vee \exists(\varepsilon,\sigma) \in \mathbb{M}, \omega = \mathbf{eval}_l\llbracket \mathtt{l} \rrbracket(\varepsilon,\sigma)\} \\ &\cup \{\omega \mid \exists(\varepsilon,\sigma) \in \mathbb{M}, a = \mathbf{eval}_l\llbracket \mathtt{l} \rrbracket(\varepsilon,\sigma) \wedge 0 = \mathbf{create}(n,\sigma) \wedge \omega = \mathbf{write}(a,0,\sigma)\} \\ &\cup \{\omega \mid \exists(\varepsilon,\sigma) \in \mathbb{M}, a = \mathbf{eval}_l\llbracket \mathtt{l} \rrbracket(\varepsilon,\sigma) \wedge (a_1,\sigma_1) = \mathbf{create}(n,\sigma) \wedge \omega = \mathbf{write}(a,v_1,\sigma_1)\} \\ &\cup \{(\varepsilon,\sigma_1) \mid \exists(\varepsilon,\sigma) \in \mathbb{M}, a = \mathbf{eval}_l\llbracket \mathtt{l} \rrbracket(\varepsilon,\sigma) \wedge 0 = \mathbf{create}(n,\sigma) \wedge \sigma_1 = \mathbf{write}(a,0,\sigma)\} \\ &\cup \{(\varepsilon,\sigma_2) \mid (\varepsilon,\sigma) \in \mathbb{M}, a = \mathbf{eval}_l\llbracket \mathtt{l} \rrbracket(\varepsilon,\sigma) \wedge (a_1,\sigma_1) = \mathbf{create}(n,\sigma) \wedge \sigma_2 = \mathbf{write}(a,v_1,\sigma_1)\}\end{aligned}$$

De-allocation:

$$\begin{aligned}&\llbracket \mathbf{free}(\mathtt{l},n) \rrbracket(\mathbb{M}) \ = \\ &\quad \{\omega \mid \omega \in \mathbb{M} \vee \exists(\varepsilon,\sigma) \in \mathbb{M}, \omega = \mathbf{eval}_l\llbracket \mathtt{l} \rrbracket(\varepsilon,\sigma)\} \\ &\cup \{\omega \mid \exists(\varepsilon,\sigma) \in \mathbb{M}, a = \mathbf{eval}_l\llbracket \mathtt{l} \rrbracket(\varepsilon,\sigma) \wedge \omega = \mathbf{delete}(a,n,\sigma)\} \\ &\cup \{(\varepsilon,\sigma_1) \mid (\varepsilon,\sigma) \in \mathbb{M}, a = \mathbf{eval}_l\llbracket \mathtt{l} \rrbracket(\varepsilon,\sigma) \wedge \sigma_1 = \mathbf{delete}(a,n,\sigma)\}\end{aligned}$$

Assertion:

$$\begin{aligned}&\llbracket \mathbf{assert}(\mathtt{r}) \rrbracket(\mathbb{M}) \ = \\ &\quad \{\omega \mid \omega \in \mathbb{M} \vee \exists(\varepsilon,\sigma) \in \mathbb{M}, \omega = \mathbf{eval}_r\llbracket \mathtt{r} \rrbracket(\varepsilon,\sigma) \vee 0 = \mathbf{eval}_r\llbracket \mathtt{r} \rrbracket(\varepsilon,\sigma)\} \\ &\cup \{(\varepsilon,\sigma) \mid (\varepsilon,\sigma) \in \mathbb{M}, v = \mathbf{eval}_r\llbracket \mathtt{r} \rrbracket(\varepsilon,\sigma) \wedge v \neq 0\}\end{aligned}$$

Conditional branch:

$$\llbracket \mathbf{if}(\mathtt{r})\{\mathtt{p}_1\}\mathbf{else}\{\mathtt{p}_2\} \rrbracket(\mathbb{M}) \ = \ \llbracket \mathtt{p}_1 \rrbracket \circ \mathbf{guard}\llbracket \mathtt{r} \neq 0 \rrbracket(\mathbb{M}) \cup \llbracket \mathtt{p}_2 \rrbracket \circ \mathbf{guard}\llbracket \mathtt{r} == 0 \rrbracket(\mathbb{M})$$

Loop:

$$\llbracket \mathbf{while}(\mathtt{r})\{\mathtt{p}\} \rrbracket(\mathbb{M}) \ = \ \mathbf{guard}\llbracket \mathtt{r} == 0 \rrbracket(\mathbf{lfp}_{\subseteq}\lambda\mathbb{M}_1.\, \mathbb{M} \cup \llbracket \mathtt{p} \rrbracket \circ \mathbf{guard}\llbracket \mathtt{r} \neq 0 \rrbracket(\mathbb{M}_1))$$

Sequence:

$$\llbracket \mathtt{p}_1; \mathtt{p}_2 \rrbracket(\mathbb{M}) \ = \ \llbracket \mathtt{p}_2 \rrbracket \circ \llbracket \mathtt{p}_1 \rrbracket(\mathbb{M})$$

Figure 2.4: Denotational semantics of programs

induction over the syntax of programs. It relies on the store operations **read**, **write**, **create** and **delete**, and the semantics of left and right value expressions $\mathbf{eval}_l[\![\mathtt{l}]\!]$ and $\mathbf{eval}_r[\![\mathtt{r}]\!]$. In order to handle control flow operations in branch statements and loop statements, it relies on a $\mathbf{guard}[\![\mathtt{r}]\!]$ operation that filters out memory states that do not satisfy the condition expression `r`:

$$\mathbf{guard}[\![\mathtt{r}]\!] \in \mathcal{P}(\mathcal{M}_\omega) \to \mathcal{P}(\mathcal{M}_\omega)$$

Note that, as defined in Figure 2.4, all the program statements propagate the error state ω, that is, for all `p` and for all $\mathbb{M} \subseteq \mathcal{M}_\omega$,

$$\omega \in \mathbb{M} \implies \omega \in [\![\mathtt{p}]\!](\mathbb{M})$$

In addition to that, the semantics of some statements on non-error states are discussed below.

The variable declaration statement, `x`, consists of creating a new cell for variable `x` for all the non-error states. If in a memory state (ε, σ), variable `x` is already declared, denoted as $\mathtt{x} \in \mathrm{dom}(\varepsilon)$, or the store allocation fails, denoted as $0 = \mathbf{create}(1, \sigma)$, executing the statement will output the error state ω in the final states.

The assignment statement, `l = r`, is carried out on each non-error state $(\varepsilon, \sigma) \in \mathbb{M}$ by evaluating the left-hand side `l` into an address a, evaluating the right-hand side `r` into a value v, and writing the value v into the cell at address a in the store σ. If one of the three steps fails, i.e., outputs the error state ω, the execution of the assignment statement will also output the error state ω in the final states.

The allocation statement, $\mathtt{l} = \mathbf{malloc}(n)$ first evaluates the left-hand side `l` into an address a on a non-error state $(\varepsilon, \sigma) \in \mathbb{M}$. If the evaluation process fails, it outputs the error state. Otherwise, it allocates a memory block of n memory cells in the store. When the allocation fails, i.e., returns a null address 0, it writes the null address 0 into the cell at address a. When the allocation succeeds, i.e., returns the base address a_1 of the newly allocated block and the new store σ_1, it writes address a_1 into the cell at address a in the new store σ_1.

The semantics for *a loop statement*, $\mathbf{while}(\mathtt{r})\{\mathtt{p}\}$, is defined on the least fix-point, denoted as $\mathbf{lfp}_\subseteq \lambda \mathbb{M}_1.\, \mathbb{M} \cup [\![\mathtt{p}]\!] \circ \mathbf{guard}[\![\mathtt{r} \neq 0]\!](\mathbb{M}_1)$, of the following function:

$$\lambda \mathbb{M}_1.\, \mathbb{M} \cup [\![\mathtt{p}]\!] \circ \mathbf{guard}[\![\mathtt{r} \neq 0]\!](\mathbb{M}_1)$$

Indeed, the function is monotone over the complete lattice $(\mathcal{P}(\mathcal{M}_\omega), \subseteq, \cup)$ as $\mathbf{guard}[\![\mathtt{r}]\!]$ and $[\![\mathtt{p}]\!]$ are monotone. This ensures that the function always has a least fix-point. Intuitively, a loop invariant covers all the reachable states during the execution of the loop from a set of initial states.

The semantics of the other statements are all quite natural, and are thus not discussed in detail.

Extensions. The programming language presented in Figure 2.1 is a small fragment of the C language, which allows a more concise presentation of the thesis. In practice, we could also consider other features of the C language:

- C programs that contain other control statements (e.g., **switch**, **break**, **goto**, **for**) can be transformed into equivalent programs with only **if** and **while** statements.

- The scopes of variables can be added into the environment in order to allow local variables to be defined in nested blocks.
- In order to support functions, we could add the call records into the environment.
- Since our language supports expressions of pointer dereference and field access, it can be easily extended to handle basic types (e.g., **bool**, **int**), pointer types, and structure types.

2.2 Abstract Interpretation

Abstract interpretation [CC77] is a theory that allows different program semantics to be compared by using abstractions. In this section, we recall the most basic notions of abstract interpretation, which are used in the following in order to design sound and automatic static analyzers to prove properties about programs.

As shown in Section 2.1, the behavior of a program can be formalized as a function that maps sets of initial memory states into sets of final memory states. In order to abstract such behaviors, we need to design an abstract domain that includes a set of abstract states, abstract operations and some over-approximation of lattice operations.

Abstract states. An abstract state $\overline{\mathrm{m}} \in \overline{\mathcal{M}_\omega}$ describes a set of concrete states. The relationship between abstract and concrete states is explicitly given by a *concretization function* that maps each abstract state into a set of concrete states:

$$\gamma : \overline{\mathcal{M}_\omega} \to \mathcal{P}(\mathcal{M}_\omega)$$

where $\mathcal{P}(\mathcal{M}_\omega)$ is the power-set of concrete states. In addition, abstract states often include the bottom and top elements $\bot, \top \in \overline{\mathcal{M}_\omega}$, which denote the empty and complete concrete sets:

$$\begin{aligned} \gamma(\bot) &= \emptyset \\ \gamma(\top) &= \mathcal{M}_\omega \end{aligned}$$

Example 2.1 (Interval abstractions). An interval abstraction $\overline{\mathrm{m}}$ is a map from each program variable $\mathtt{x} \in \mathcal{X}$ to an interval abstraction $[n_1, n_2]$. It abstracts away the concrete addresses of program variables and over-approximates the concrete value of program variables by intervals. The concretization function of interval abstractions is defined as:

$$\gamma(\overline{\mathrm{m}}) = \{(\varepsilon, \sigma) \mid \forall \mathtt{x} \mapsto [n_1, n_2] \in \overline{\mathrm{m}}, n_1 \leq \sigma \circ \varepsilon(\mathtt{x}) \leq n_2\}$$

Abstract operations. In order to describe manipulations of concrete states, abstract interpretation relies on abstract operations. Abstract operations correspond to concrete computations at the abstract level. Therefore, the abstract operations that an abstract domain should provide depend on the operations on the concrete elements. More precisely, for each concrete operation $\mathbf{op} : \mathcal{M}_\omega \times \mathcal{M}_\omega \to \mathcal{M}_\omega$, the analysis should use a correspondence abstract operation $\overline{\mathbf{op}} : \overline{\mathcal{M}_\omega} \times \overline{\mathcal{M}_\omega} \to \overline{\mathcal{M}_\omega}$. A sound abstract operation should over-approximate all the concrete behaviors of the concrete operation.

Definition 2.2 (Soundness of abstract operations). *An abstract operation* $\overline{\mathbf{op}} : \overline{\mathcal{M}_\omega} \times \overline{\mathcal{M}_\omega} \rightarrow \overline{\mathcal{M}_\omega}$ *is sound with respect to the concrete operation* $\mathbf{op} : \mathcal{M}_\omega \times \mathcal{M}_\omega \rightarrow \mathcal{M}_\omega$ *if and only if:*

$$\forall \overline{\mathrm{m}}_1, \overline{\mathrm{m}}_2 \in \overline{\mathcal{M}_\omega}, \forall \mathrm{m}_1 \in \gamma(\overline{\mathrm{m}}_1), \mathrm{m}_2 \in \gamma(\overline{\mathrm{m}}_2), \quad \mathbf{op}(\mathrm{m}_1, \mathrm{m}_2) \in (\gamma \circ \overline{\mathbf{op}})(\overline{\mathrm{m}}_1, \overline{\mathrm{m}}_2)$$

Example 2.2 (Abstract operations on the interval abstract domain). Corresponding to the addition $+$ and subtraction $-$ operations on concrete values of program variables, the interval abstract domain should define abstract operations $\overline{+}$ and $\overline{-}$ on the intervals:

$$\begin{aligned} \overline{+}([n_1, n_2], [n_3, n_4]) = &\quad [n_1 + n_3, n_2 + n_4] \\ \overline{-}([n_1, n_2], [n_3, n_4]) = &\quad [n_1 + n_3, n_2 + n_4] \end{aligned}$$

Approximation of lattice operations. Besides basic abstract operations, the analysis also needs to use approximation of lattice operations: $\sqsubseteq_{\overline{\mathcal{M}_\omega}}$ for the entailment checking of abstract states, $\sqcup_{\overline{\mathcal{M}_\omega}}$ for joining two abstract states, $\triangledown_{\overline{\mathcal{M}_\omega}}$ for ensuring termination when computing abstract fix-points. These operators have to be sound in order to derive sound proofs of program properties.

Entailment checking operator $\sqsubseteq_{\overline{\mathcal{M}_\omega}}$. In order to prove one abstract state can be over-approximated by another abstract state, the abstract domain needs to use an entailment checking operation $\sqsubseteq_{\overline{\mathcal{M}_\omega}}$: $\overline{\mathcal{M}_\omega} \times \overline{\mathcal{M}_\omega} \rightarrow \{\mathbf{true}, \mathbf{false}\}$. In the following, we simply write $\overline{\mathrm{m}}_1 \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{\mathrm{m}}_2$ when $\sqsubseteq_{\overline{\mathcal{M}_\omega}} (\overline{\mathrm{m}}_1, \overline{\mathrm{m}}_2) = \mathbf{true}$.

Definition 2.3 (Soundness of entailment checking operator $\sqsubseteq_{\overline{\mathcal{M}_\omega}}$). *The entailment checking operation* $\sqsubseteq_{\overline{\mathcal{M}_\omega}}$: $\overline{\mathcal{M}_\omega} \times \overline{\mathcal{M}_\omega} \rightarrow \{\mathbf{true}, \mathbf{false}\}$ *is sound if and only if:*

$$\forall \overline{\mathrm{m}}_1, \overline{\mathrm{m}}_2 \in \overline{\mathcal{M}_\omega}, \quad \overline{\mathrm{m}}_1 \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{\mathrm{m}}_2 \implies \gamma(\overline{\mathrm{m}}_1) \subseteq \gamma(\overline{\mathrm{m}}_2)$$

Example 2.3 (A sound entailment checking operator on intervals).

$$[n_1, n_2] \sqsubseteq_{\overline{\mathcal{M}_\omega}} [n_3, n_4] \iff n_1 \geq n_3 \wedge n_2 \leq n_4$$

Joining operator $\sqcup_{\overline{\mathcal{M}_\omega}}$. When analyzing a program, the number of abstract states may grow exponentially due to conditional statements and abstract operations. In order to have analyses that are scalable and always terminate, abstract states may need to be merged during the analysis. The joining operator $\sqcup_{\overline{\mathcal{M}_\omega}} : \overline{\mathcal{M}_\omega} \times \overline{\mathcal{M}_\omega} \rightarrow \overline{\mathcal{M}_\omega}$ merges two abstract states and returns an over-approximation.

Definition 2.4 (Soundness of joining operator $\sqcup_{\overline{\mathcal{M}_\omega}}$). *The joining operator* $\sqcup_{\overline{\mathcal{M}_\omega}} : \overline{\mathcal{M}_\omega} \times \overline{\mathcal{M}_\omega} \rightarrow \overline{\mathcal{M}_\omega}$ *is sound if and only if:*

$$\forall \overline{\mathrm{m}}_1, \overline{\mathrm{m}}_2 \in \overline{\mathcal{M}_\omega}, \quad \gamma(\overline{\mathrm{m}}_1) \subseteq \gamma(\overline{\mathrm{m}}_1 \sqcup_{\overline{\mathcal{M}_\omega}} \overline{\mathrm{m}}_2) \wedge \gamma(\overline{\mathrm{m}}_2) \subseteq \gamma(\overline{\mathrm{m}}_1 \sqcup_{\overline{\mathcal{M}_\omega}} \overline{\mathrm{m}}_2)$$

Example 2.4 (A sound joining operator on intervals).

$$[n_1, n_2] \sqcup_{\overline{\mathcal{M}_\omega}} [n_3, n_4] = [min(n_1, n_3), max(n_2, n_4)]$$

Widening operator $\triangledown_{\overline{\mathcal{M}_\omega}}$. As shown in Figure 2.4, the concrete semantics of a loop **while**$(\mathtt{r})\{\mathtt{p}\}$ on a set of concrete states $\mathbb{M}$ is defined as the least fix-point of the function $\lambda\mathbb{M}_1.\ \mathbb{M} \cup [\![\mathrm{p}]\!] \circ \textbf{guard}[\![\mathrm{r} \neq 0]\!](\mathbb{M}_1)$. In practice, the fix-point can be reached when $\mathbb{M}_{n+1} \subseteq \mathbb{M}_n$ with the following iterations:

$$\begin{cases} \mathbb{M}_0 = \mathbb{M} \\ \mathbb{M}_{n+1} = \mathbb{M}_n \cup [\![\mathrm{p}]\!] \circ \textbf{guard}[\![\mathrm{r} \neq 0]\!](\mathbb{M}_n) \qquad n \in \mathbb{N} \end{cases}$$

However, the sequences of iterates might be infinite. The same is true for the abstract iteration below in order to compute an abstract fix-point that over-approximates the concrete least fix-point:

$$\begin{cases} \overline{\mathrm{m}}_0 = \overline{\mathrm{m}} \\ \overline{\mathrm{m}}_{n+1} = \overline{\mathrm{m}}_n \sqcup_{\overline{\mathcal{M}_\omega}} \overline{[\![\mathrm{p}]\!]} \circ \overline{\textbf{guard}[\![\mathrm{r} \neq 0]\!]}(\overline{\mathrm{m}}_n) \qquad n \in \mathbb{N} \end{cases}$$

The abstract iteration starts from an abstract state $\overline{\mathrm{m}}$ that over-approximates $\mathbb{M}$ (i.e., $\mathbb{M} \subseteq \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}})$), and is performed on the abstract semantics $\overline{[\![\mathrm{p}]\!]}$ of the loop body and the abstract semantics $\overline{\textbf{guard}[\![\mathrm{r}]\!]}$ of loop condition.

However, in order to design a static analysis that always terminates, the number of abstract iterates has to be finite. Therefore, a widening operator $\triangledown_{\overline{\mathcal{M}_\omega}} : \overline{\mathcal{M}_\omega} \times \overline{\mathcal{M}_\omega} \to \overline{\mathcal{M}_\omega}$ that joins abstract states and provides a convergence acceleration for the iteration process is necessary.

Definition 2.5 (Widening operator). *A widening operator* $\triangledown_{\overline{\mathcal{M}_\omega}} \in \overline{\mathcal{M}_\omega} \times \overline{\mathcal{M}_\omega} \to \overline{\mathcal{M}_\omega}$ *should satisfy the two following properties for soundness and termination respectively:*

1. $\forall \overline{\mathrm{m}}_1, \overline{\mathrm{m}}_2 \in \overline{\mathcal{M}_\omega}, \quad \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}_1) \subseteq \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}_1 \triangledown_{\overline{\mathcal{M}_\omega}} \overline{\mathrm{m}}_2) \wedge \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}_2) \subseteq \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}_1 \triangledown_{\overline{\mathcal{M}_\omega}} \overline{\mathrm{m}}_2)$
2. *For any sequence* $(\overline{\mathrm{m}}_n)_{n\in\mathbb{N}}$*, the sequence* $(\overline{\mathrm{m}}'_n)_{n\in\mathbb{N}}$ *defined below is ultimately stationary:*

$$\begin{cases} \overline{\mathrm{m}}'_0 = \overline{\mathrm{m}}_0 \\ \overline{\mathrm{m}}'_{n+1} = \overline{\mathrm{m}}'_n \triangledown_{\overline{\mathcal{M}_\omega}} \overline{\mathrm{m}}_{n+1} \qquad n \in \mathbb{N} \end{cases}$$

Example 2.5 (A widening operator of interval abstractions).

$$[n_1, n_2] \triangledown_{\overline{\mathcal{M}_\omega}} [n_3, n_4] = [n_1 \leq n_3 ? n_1 : -\infty, n_2 \geq n_4 ? n_2 : +\infty]$$

With the widening operator $\triangledown_{\overline{\mathcal{M}_\omega}}$, we can thus have the following abstract iteration with widening:

$$\begin{cases} \overline{\mathrm{m}}_0 = \overline{\mathrm{m}} \\ \overline{\mathrm{m}}_{n+1} = \overline{\mathrm{m}}_n \triangledown_{\overline{\mathcal{M}_\omega}} \overline{[\![\mathrm{p}]\!]} \circ \overline{\textbf{guard}[\![\mathrm{r} \neq 0]\!]}(\overline{\mathrm{m}}_n) \qquad n \in \mathbb{N} \end{cases}$$

According to [CC77], the abstract iteration with widening is ultimately stationary and sound, i.e., it is able to compute a sound over-approximation of the concrete least fix-point in a finite number of iterates, which is the *abstract post fix-point* of the loop **while**(r){p} from the pre-condition $\overline{\mathrm{m}}$, denoted as $\overline{\mathbf{lfp}}_{\sqsubseteq_{\overline{\mathcal{M}_\omega}}}(\lambda \overline{\mathrm{m}}_1.\ \overline{\mathrm{m}} \nabla_{\overline{\mathcal{M}_\omega}} \overline{[\![\mathrm{p}]\!]} \circ \overline{\mathbf{guard}[\![\mathrm{r} \neq 0]\!]}(\overline{\mathrm{m}}_1))$.

However, in practice, the widening operator $\nabla_{\overline{\mathcal{M}_\omega}}$ can lead to a huge loss in precision. In order to compute more precise loop invariants, static analyses often make use of the widening operator to accelerate the iteration convergence after several abstract iterates with the join operator $\sqcup_{\overline{\mathcal{M}_\omega}}$.

Forward analyses. An analysis based on forward abstract interpretation starts from an abstract pre-condition that takes into account all the possible initial states and aims to automatically compute an abstract post-condition that covers all the final reachable states.

Example 2.6 (An example of a forward analysis using interval abstractions). Let us consider the analysis of the simple program below with the abstract pre-condition $\mathtt{i} \mapsto [-\infty, +\infty]$ as an example.

```
1 i = 0;
2 while(i<10){
3   i = i+1;
4 }
```

The abstract pre-condition states that variable `i` may be any integer value. Then, the assignment statement at line 1 writes 0 to `i`, which results in abstract state $i \mapsto [0, 0]$. Starting from the abstract state $i \mapsto [0, 0]$, the analysis then needs to iterate on the **while** loop to compute an abstract fix-point. The iteration process is shown below:

Iteration	loop head	loop end	join or widen
first:	$\mathtt{i} \mapsto [0,0]$	$\mathtt{i} \mapsto [1,1]$	$\mathtt{i} \mapsto [0,0] \sqcup_{\overline{\mathcal{M}_\omega}} \mathtt{i} \mapsto [1,1] = \mathtt{i} \mapsto [0,1]$
second:	$\mathtt{i} \mapsto [0,1]$	$\mathtt{i} \mapsto [1,2]$	$\mathtt{i} \mapsto [0,1] \sqcup_{\overline{\mathcal{M}_\omega}} \mathtt{i} \mapsto [1,2] = \mathtt{i} \mapsto [0,2]$
third:	$\mathtt{i} \mapsto [0,2]$	$\mathtt{i} \mapsto [1,3]$	$\mathtt{i} \mapsto [0,2] \nabla_{\overline{\mathcal{M}_\omega}} \mathtt{i} \mapsto [1,3] = \mathtt{i} \mapsto [0,+\infty[$
fourth:	$\mathtt{i} \mapsto [0,+\infty[$	$\mathtt{i} \mapsto [1,+\infty[$	$\mathtt{i} \mapsto [0,+\infty[\ \nabla_{\overline{\mathcal{M}_\omega}} \mathtt{i} \mapsto [1,+\infty[= \mathtt{i} \mapsto [0,+\infty[$

The first loop iteration increases the value of `i` by 1 at line 3, which results in abstract state $\mathtt{i} \mapsto [1, 1]$ at the end of the first loop iteration. Then, the analysis joins the two abstract states corresponding to the loop entry and to the end of the first iteration and gets abstract state $\mathtt{i} \mapsto [0, 1]$. The second loop iteration starts from the join result $\mathtt{i} \mapsto [0, 1]$ and ends in abstract state $\mathtt{i} \mapsto [1, 2]$. After joining the two abstract states $\mathtt{i} \mapsto [0, 1]$ and $\mathtt{i} \mapsto [1, 2]$, the analysis gets $\mathtt{i} \mapsto [0, 2]$. From the third iteration, the analysis starts to widen the abstract states corresponding to the loop entry and to the end of the iteration. In both the third the fourth iterations, the analysis gets abstract state $\mathtt{i} \mapsto [0, +\infty[$, thus the abstract iteration terminates and returns the abstract fix-point $\mathtt{i} \mapsto [0, +\infty[$.

2.3 Memory State Abstractions

This section and the next aim to introduce a memory abstract domain [CR08] that is based on a fragment of separation logic [Rey02]. In particular, the memory abstractions and the concretization function are presented in this section. We note that other separation logic based shape analyses [BCC$^+$07, CDOY09, DOY06] all share similar principles.

Abstract values. To abstract concrete memory states, we first need to abstract concrete values by abstract values. An abstract value $\overline{v} \in \overline{\mathcal{V}}$ is either a symbolic variable $\alpha \in \overline{\mathcal{V}}_\alpha$ or a symbolic address of a program variable $\&\mathtt{x} \in \mathcal{X}_\&$. To concretize abstract values, we need a *valuation* function $\mu \in \overline{\mathcal{V}} \to \mathcal{V}$ that maps an abstract value $\overline{v} \in \overline{\mathcal{V}}$ to a concrete value $v \in \mathcal{V}$. We let $\mu_{\lceil \mathcal{X}_\&}$ denote the restriction of a valuation μ to symbolic addresses of program variables $\mathcal{X}_\&$.

Abstract states. An abstract state $\overline{\mathrm{m}} \in \overline{\mathcal{M}_\omega}$ is either $\top$ (top), $\bot$ (bottom) or a pair $(\overline{\mathrm{g}}, \overline{\mathrm{n}}) \in \overline{\mathcal{G}} \times \overline{\mathcal{N}}$ that is composed of an abstract shape graph $\overline{\mathrm{g}} \in \overline{\mathcal{G}}$ and a numerical abstraction $\overline{\mathrm{n}} \in \overline{\mathcal{N}}$:

$$\overline{\mathrm{m}} \in \overline{\mathcal{M}_\omega} ::= \overline{\mathcal{G}} \times \overline{\mathcal{N}} \uplus \{\bot, \top\}$$

Intuitively, an abstract shape graph $\overline{\mathrm{g}} \in \overline{\mathcal{G}}$ describes the structural properties of concrete memory states by using separating conjunction formulas parameterized by abstract values $\overline{\mathcal{V}}$, and thus, is concretized into a set of pairs made up of a concrete store $\sigma \in \mathcal{H}$ and a valuation $\mu \in \overline{\mathcal{V}} \to \mathcal{V}$ of abstract values:

$$\gamma_{\overline{\mathcal{G}}} : \overline{\mathcal{G}} \to \mathcal{P}(\mathcal{H} \times (\overline{\mathcal{V}} \to \mathcal{V}))$$

A numerical abstraction $\overline{\mathrm{n}} \in \overline{\mathcal{N}}$ abstracts the numerical properties of abstract values and is concretized into a set of valuations of abstract values:

$$\gamma_{\overline{\mathcal{N}}} : \overline{\mathcal{N}} \to \mathcal{P}(\overline{\mathcal{V}} \to \mathcal{V})$$

Further details of abstract shape graphs and numerical abstractions are discussed in the following.

Given the concretization of abstract shape graphs and numerical abstractions, we can thus define the concretization of abstract states $\gamma_{\overline{\mathcal{M}_\omega}} : \overline{\mathcal{M}_\omega} \to \mathcal{P}(\mathcal{M}_\omega)$ as follows:

$$\begin{array}{lcl}
\gamma_{\overline{\mathcal{M}_\omega}}(\bot) & = & \emptyset \\
\gamma_{\overline{\mathcal{M}_\omega}}(\top) & = & \mathcal{M}_\omega \\
\gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{g}}, \overline{\mathrm{n}}) & = & \{\ (\mu_{\lceil \mathcal{X}_\&}, \sigma) \mid \exists \mu \in \gamma_{\overline{\mathcal{N}}}(\overline{\mathrm{n}}), (\sigma, \mu) \in \gamma_{\overline{\mathcal{G}}}(\overline{\mathrm{g}})\}
\end{array}$$

Note that an abstract state is concretized into a set of concrete states, where a concrete state is either an error state ω or is a pair (ε, σ) made up of an environment ε mapping symbolic addresses of program variables into their real addresses and a store σ. The bottom state $\bot$ is concretized into an empty set and the top state $\top$ stands for the complete set of concrete states. For each concrete state $(\varepsilon, \sigma) \in \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{g}}, \overline{\mathrm{n}})$, there exists a valuation μ that is coherent with the environment ε (i.e., $\varepsilon = \mu_{\lceil \mathcal{X}_\&}$), the abstract shape $\overline{\mathrm{g}}$ and the numerical abstraction $\overline{\mathrm{n}}$, i.e., $(\sigma, \mu) \in \gamma_{\overline{\mathcal{G}}}(\overline{\mathrm{g}})$ and $\mu \in \gamma_{\overline{\mathcal{N}}}(\overline{\mathrm{n}})$, denoted as $(\sigma, \mu) \vDash (\overline{\mathrm{g}}, \overline{\mathrm{n}})$.

Abstract shapes. An abstract shape $\overline{\mathrm{g}} \in \overline{\mathcal{G}}$ is a separating conjunction $*$ of region predicates:

$$\overline{\mathrm{g}}\ (\in \overline{\mathcal{G}}) ::= \overline{\mathrm{p}}_1 * \ldots * \overline{\mathrm{p}}_n$$

Each region predicate $\overline{\mathrm{p}}_i \in \overline{\mathcal{P}}$ describes a store that is disjoint from stores described by other region predicates. The concretization of an abstract shape can thus be defined as "gluing" disjoint stores, each of which concretizes a region predicate:

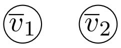
$$\gamma_{\overline{\mathcal{G}}}(\overline{\mathrm{p}}_1 * \ldots * \overline{\mathrm{p}}_n) = \{\ (\sigma_1 \uplus \ldots \uplus \sigma_n, \mu) \mid \forall 1 \leq i \leq n, (\sigma_i, \mu) \in \gamma_{\overline{\mathcal{P}}}(\overline{\mathrm{p}}_i)\}$$

A region predicate can be **emp** denoting an empty region, a points-to predicate (e.g., $\overline{v}_1.\mathtt{f} \mapsto \overline{v}_2$) denoting a single memory cell, an inductive predicate (e.g., $\overline{v}_2 \cdot \mathbf{list}$) describing a store with an inductive data structure or a segment predicate (e.g., $\overline{v}_1 \cdot \mathbf{list} \mathrel{*{=}} \overline{v}_2 \cdot \mathbf{list}$) describing a fragment of an inductive data structure.

The empty predicate. The empty predicate **emp** denotes an empty store, and thus can be concretized into a set of pairs composed of an empty store and any valuation:

$$\gamma_{\overline{\mathcal{P}}}(\mathbf{emp}) = \{\ ([\ ], \mu) \mid \mu \in \overline{\mathcal{V}} \to \mathcal{V}\}$$

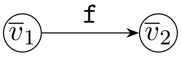
In the following, we represent the **emp** predicate as an empty graph with only abstract values and no edges, such as the graph below with only two nodes representing abstract values:



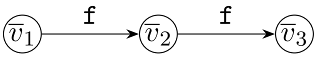
Points-to predicates. A points-to predicate $\overline{v}_1.\mathtt{f} \mapsto \overline{v}_2$ abstracts a store with only a concrete cell, where given a valuation μ of abstract values $\overline{v}_1$ and $\overline{v}_2$, the abstract address $\overline{v}_1 + \mathtt{f}$ denotes the concrete address $\mu(\overline{v}_1) + \mathtt{f}$ in the store and the abstract value $\overline{v}_2$ denotes the concrete value $\mu(\overline{v}_2)$ of the cell. Formally, a points-to edge is concretized as follows:

$$\gamma_{\overline{\mathcal{P}}}(\overline{v}_1.\mathtt{f} \mapsto \overline{v}_2) = \{\ ([\mu(\overline{v}_1) + \mathtt{f} \mapsto \mu(\overline{v}_2)], \mu) \mid \overline{v}_1, \overline{v}_2 \in \mathrm{dom}(\mu)\}$$

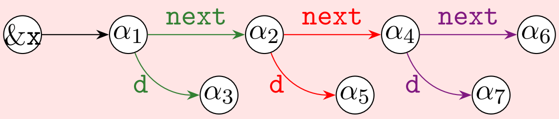
In the following, we represent below a points-to predicate $\overline{v}_1.\mathtt{f} \mapsto \overline{v}_2$ as an edge:



Moreover, the separating conjunction $*$ of region predicates is represented in sub-graphs with disjoint edges. For example, the separating conjunction of the two points-to predicates $\overline{v}_1.\mathtt{f} \mapsto \overline{v}_2 * \overline{v}_2.\mathtt{f} \mapsto \overline{v}_3$ is represented as the following graph with two disjoint edges:

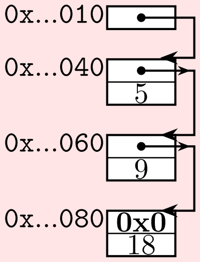


Example 2.7 (An abstract shape with only points-to edges). Let us consider the abstract shape graph below which abstracts a memory block with a list pointed to by variable `x`:



The points-to edge starting at node &`x` abstracts the memory cell of variable `x`. The two points-to edges starting at node α_1 respectively abstract the next field and the data field of the first list element. Similarly, the points-to edges starting at nodes α_2 and α_4 abstract the the following two list elements. The shape graph contains 7 points-to edges in total, and thus should be concretized into a set of pairs composed of a concrete store with exactly 7 memory cells and a valuation. One concrete example is the following pair (σ_1, μ_1), where, σ_1 is the graphical representation of $[\texttt{0x...010} \mapsto \texttt{0x...040}] \uplus [\texttt{0x...040} + \texttt{next} \mapsto \texttt{0x...060}] \uplus [\texttt{0x...040} + \texttt{d} \mapsto 5] \uplus [\texttt{0x...060} + \texttt{next} \mapsto \texttt{0x...080}] \uplus [\texttt{0x...060} + \texttt{d} \mapsto 9] \uplus [\texttt{0x...080} + \texttt{next} \mapsto \texttt{0x0}] \uplus [\texttt{0x...060} + \texttt{d} \mapsto 18]$:

$\sigma_1 \in \mathcal{H}$

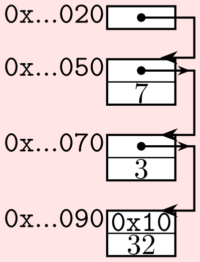


$\mu_1 \in \overline{\mathcal{V}} \to \mathcal{V}$

$$\begin{array}{lll@{\qquad}lll}
\&\texttt{x} & \mapsto & \texttt{0x...010} & & & \\
\alpha_1 & \mapsto & \texttt{0x...040} & \alpha_3 & \mapsto & 5 \\
\alpha_2 & \mapsto & \texttt{0x...060} & \alpha_5 & \mapsto & 9 \\
\alpha_4 & \mapsto & \texttt{0x...080} & \alpha_7 & \mapsto & 18 \\
\alpha_6 & \mapsto & \texttt{0x0} & & &
\end{array}$$

We note σ_1 contains a list of length 3 as the next pointer of the last list element is null. However, in the concretization, we could have different stores with different valuations, so that the store only contains a list fragment of length 3, such as the following one (σ_2, μ_2):

$\sigma_2 \in \mathcal{H}$



$\mu_2 \in \overline{\mathcal{V}} \to \mathcal{V}$

$$\begin{array}{lll@{\qquad}lll}
\&\texttt{x} & \mapsto & \texttt{0x...020} & & & \\
\alpha_1 & \mapsto & \texttt{0x...050} & \alpha_3 & \mapsto & 7 \\
\alpha_2 & \mapsto & \texttt{0x...070} & \alpha_5 & \mapsto & 3 \\
\alpha_4 & \mapsto & \texttt{0x...090} & \alpha_7 & \mapsto & 32 \\
\alpha_6 & \mapsto & \texttt{0x...010} & & &
\end{array}$$

Inductive predicates. While separating shape graphs containing only points-to edges are expressive enough to abstract linked lists of a certain length, they cannot describe sets of memory states containing linked lists of any length. Intuitively, some data structures e.g., linked lists and binary trees, present a recursive pattern that can be summarized by an inductive definition.

For example, linked lists can be defined as:

$$\begin{array}{lll} \alpha_0 \cdot \mathbf{list} & ::= & \\ & & \mathbf{emp} \wedge \alpha_0 = 0 \\ & \vee & \alpha_0.\mathtt{next} \mapsto \alpha_1 * \alpha_0.\mathtt{d} \mapsto \alpha_2 * \alpha_1 \cdot \mathbf{list} \wedge \alpha \neq 0 \end{array}$$

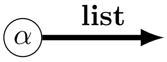
This says that a list starting from address α_0 is either empty (no list element) when α_0 is null or has a list element at address α_0 (which has a next field to α_1 and a data field) and a list $\alpha_1 \cdot \mathbf{list}$ starting from address α_1. Similarly, binary trees can be defined as:

$$\begin{array}{lll} \alpha_0 \cdot \mathbf{tree} & ::= & \\ & & \mathbf{emp} \wedge \alpha_0 = 0 \\ & \vee & \alpha_0.\mathtt{l} \mapsto \alpha_1 * \alpha_0.\mathtt{r} \mapsto \alpha_2 * \alpha_0.\mathtt{d} \mapsto \alpha_3 * \alpha_1 \cdot \mathbf{tree} * \alpha_2 \cdot \mathbf{tree} \wedge \alpha \neq 0 \end{array}$$

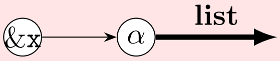
This says that a binary tree starting from address α_0 is either empty (no tree element) when α_0 is null or has a tree element at address α_0 (which contains a left field to α_1, a right field to α_2 and a data field), a left sub-tree $\alpha_1 \cdot \mathbf{tree}$ starting from address α_1 and a right sub-tree $\alpha_2 \cdot \mathbf{tree}$ starting from address α_2.

Following inductive definitions, we can thus extend abstract shape graphs with inductive predicates that are parameterized by inductive definitions. For instance, linked lists rooted at abstract address α can be abstracted by the inductive predicate $\alpha \cdot \mathbf{list}$ which follows the **list** inductive definition. Similarly, binary trees rooted at abstract address β can be abstracted by the inductive predicate $\beta \cdot \mathbf{tree}$ which follows the **tree** inductive definition.

In the following, we present inductive predicates as thick edges in graphs, for example, $\alpha \cdot \mathbf{list}$ is represented as:

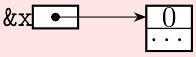


Example 2.8 (An abstract shape graph with inductive predicates). Given the **list** inductive definition, we can now abstract lists of any length that are pointed to by variable `x` by the following abstract shape graph:

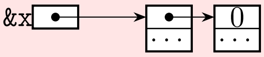


A few concrete examples that can be abstracted by the abstract shape graph are shown below:

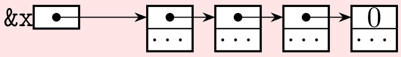
- Variables `x` points to a list with one element:



- Variable `x` points a list with two elements:



- Variable `x` points to a list with four elements:



Further details about inductive definitions and the concretization of inductive predicates are given in Chapter 4.

Segment predicates. Inductive definitions provide descriptions for complete data structures. However, in order to describe segments of data structures, we need segment definitions. Segment definitions can usually be derived fairly systematically from the initial inductive definitions. For example, the list segment definition $\alpha_0 \cdot \textbf{list} \mathrel{*\!\!=} \alpha_1 \cdot \textbf{list}$ below is derived from the **list** inductive definition:

$$\begin{array}{l}\alpha_0 \cdot \textbf{list} \mathrel{*\!\!=} \alpha_1 \cdot \textbf{list} ::= \\ \quad \textbf{emp} \wedge \alpha_0 = \alpha_1 \\ \quad \vee \;\; \alpha_0.\texttt{next} \mapsto \alpha_2 * \alpha_0.\texttt{d} \mapsto \alpha_3 * \alpha_2 \cdot \textbf{list} \mathrel{*\!\!=} \alpha_1 \cdot \textbf{list} \wedge \alpha_0 \neq \alpha_1\end{array}$$

It describes a list starting at address α_0 with a missing sub-list starting at address α_1, in other words by separating conjunction with a sub-list starting at address α_1, we can get a complete list starting at address α_0:

$$\alpha_0 \cdot \textbf{list} \mathrel{*\!\!=} \alpha_1 \cdot \textbf{list} * \alpha_1 \cdot \textbf{list} \iff \alpha_0 \cdot \textbf{list}$$

Similarly, a binary tree segment definition $\alpha_0 \cdot \textbf{tree} \mathrel{*\!\!=} \alpha_1 \cdot \textbf{tree}$ below is derived from the **tree** inductive definition:

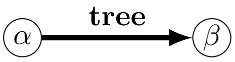
$$\begin{array}{l}\alpha_0 \cdot \textbf{tree} \mathrel{*\!\!=} \alpha_1 \cdot \textbf{tree} ::= \\ \quad \textbf{emp} \wedge \alpha_0 = \alpha_1 \\ \quad \vee \;\; \alpha_0.\texttt{l} \mapsto \alpha_2 * \alpha_0.\texttt{r} \mapsto \alpha_3 * \alpha_0.\texttt{d} \mapsto \alpha_4 * \alpha_2 \cdot \textbf{tree} \mathrel{*\!\!=} \alpha_1 \cdot \textbf{tree} * \alpha_3 \cdot \textbf{tree} \wedge \alpha_0 \neq \alpha_1 \\ \quad \vee \;\; \alpha_0.\texttt{l} \mapsto \alpha_2 * \alpha_0.\texttt{r} \mapsto \alpha_3 * \alpha_0.\texttt{d} \mapsto \alpha_4 * \alpha_3 \cdot \textbf{tree} \mathrel{*\!\!=} \alpha_1 \cdot \textbf{tree} * \alpha_2 \cdot \textbf{tree} \wedge \alpha_0 \neq \alpha_1\end{array}$$

It describes a binary tree rooted at abstract address α_0 with a missing sub-tree rooted at abstract address α_1, i.e., by separating conjunction with a sub-tree starting at address α_1, we can get a complete tree starting at address α_0:

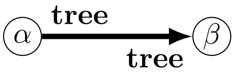
$$\alpha_0 \cdot \textbf{tree} \mathrel{*\!\!=} \alpha_1 \cdot \textbf{tree} * \alpha_1 \cdot \textbf{tree} \iff \alpha_0 \cdot \textbf{tree}$$

We note that, in the tree segment definition, the last two rules respectively describe cases where the missing sub-tree is in the left or the right branch of the tree node at α_0.

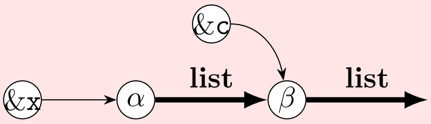
Parameterized by segment definitions, we can thus extend abstract shape graphs with segment predicates to describe partial data strcutures. In the following, we present segment predicates as thick edges with a source node and a destination node in graphs. For example, the graphical representation of a tree segment $\alpha \cdot \textbf{tree} \mathrel{*\!\!=} \beta \cdot \textbf{tree}$ is the following:



When necessary (e.g., when we have different inductive predicates at both ends or have inductive predicates with parameters), we may also use the following graphical representation:

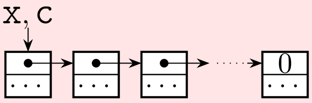


Example 2.9 (An abstract shape graph with segment predicates). The abstract shape graph below abstracts memory states occurring in a classical list traversal algorithm, where, `x` is a pointer to the first list element and `c` is a "cursor" pointing to an element in the list:

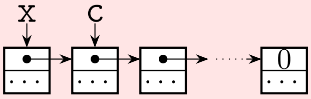


A few concrete examples are shown in the following, where for simplicity we leave out memory cells of variables `x` and `c`:

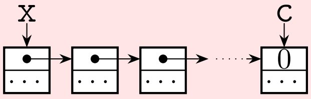
- The list segment pointed to by variable `x` and `c` is empty:



- The list segment pointed to by variable `x` and `c` has one element:



- The list segment pointed to by variable `x` and `c` contains all the list elements except the last one:



Further details about segment definitions and the concretization are discussed in Chapter 4.

Numerical abstractions. Numerical abstractions are used to abstract the numerical properties of abstract values, and are not the primary focus of the thesis. Therefore, in the thesis, we assume a numerical abstract domain $\overline{\mathcal{N}}$, which provides the following abstract operations:

abstract elements	$\overline{\mathrm{n}} \in \overline{\mathcal{N}}$
concretization	$\gamma_{\overline{\mathcal{N}}} : \overline{\mathcal{N}} \to \mathcal{P}(\overline{\mathcal{V}} \to \mathcal{V})$
guard	$\overline{\mathbf{guard}}_{\overline{\mathcal{N}}} : \overline{\mathcal{N}} \times \mathcal{C}_{\overline{\mathcal{V}}} \to \overline{\mathcal{N}}$
assign	$\overline{\mathbf{assign}}_{\overline{\mathcal{N}}} : \overline{\mathcal{N}} \times \overline{\mathcal{V}} \times \mathcal{R}_{\overline{\mathcal{V}}} \to \overline{\mathcal{N}}$
remove	$\overline{\mathbf{remove}}_{\overline{\mathcal{N}}} : \overline{\mathcal{N}} \times \overline{\mathcal{V}} \to \overline{\mathcal{N}}$
prove	$\overline{\mathbf{prove}}_{\overline{\mathcal{N}}} : \overline{\mathcal{N}} \times \mathcal{C}_{\overline{\mathcal{V}}} \to \{\mathbf{true}, \mathbf{false}\}$
rename	$\overline{\mathbf{rename}}_{\overline{\mathcal{N}}} : \overline{\mathcal{N}} \times (\overline{\mathcal{V}} \to \mathcal{P}(\overline{\mathcal{V}})) \to \overline{\mathcal{N}}$
abstract join	$\sqcup_{\overline{\mathcal{N}}} : \overline{\mathcal{N}} \times \overline{\mathcal{N}} \to \overline{\mathcal{N}}$
abstract widen	$\triangledown_{\overline{\mathcal{N}}} : \overline{\mathcal{N}} \times \overline{\mathcal{N}} \to \overline{\mathcal{N}}$
inclusion check	$\sqsubseteq_{\overline{\mathcal{N}}} : \overline{\mathcal{N}} \times \overline{\mathcal{N}} \to \{\mathbf{true}, \mathbf{false}\}$

We note that, $\overline{\mathbf{guard}}_{\overline{\mathcal{N}}}$ inputs a numerical abstraction $\overline{\mathrm{n}} \in \overline{\mathcal{N}}$ and a numerical constraint $\mathrm{c}_{\overline{\mathcal{V}}} \in \mathcal{C}_{\overline{\mathcal{V}}}$, and returns a numerical abstraction which enforces $\mathrm{c}_{\overline{\mathcal{V}}}$; $\overline{\mathbf{assign}}_{\overline{\mathcal{N}}}$ inputs a numerical abstraction $\overline{\mathrm{n}} \in \overline{\mathcal{N}}$, an abstract variable $\overline{v} \in \overline{\mathcal{V}}$ and a numerical expression $\mathbf{r}_{\overline{\mathcal{V}}} \in \mathcal{R}_{\overline{\mathcal{V}}}$, and returns a numerical abstraction where abstract variable $\overline{v}$ has been assigned to a new meaning that is computed based on $\mathbf{r}_{\overline{\mathcal{V}}}$; $\overline{\mathbf{remove}}_{\overline{\mathcal{N}}}$ inputs an abstraction $\overline{\mathrm{n}}$ and an abstract variable $\overline{v}$, and removes the abstract variable $\overline{v}$ from the abstraction $\overline{\mathrm{n}}$; $\overline{\mathbf{prove}}_{\overline{\mathcal{N}}}$ inputs an abstraction $\overline{\mathrm{n}}$ and a numerical constraint $\mathrm{c}_{\overline{\mathcal{V}}}$, and returns **true** when it successfully establishes that $\overline{\mathrm{n}}$ entails $\mathrm{c}_{\overline{\mathcal{V}}}$; $\overline{\mathbf{rename}}_{\overline{\mathcal{N}}}$ inputs an abstraction $\overline{\mathrm{n}}$ and a mapping from abstract variables in $\overline{\mathrm{n}}$ to other abstract variables, and returns another abstraction which renames abstract variables in $\overline{\mathrm{n}}$ according to the input mapping.

In practice, a suitable numerical domain is obtained as a reduced product of a numerical domain that only reasons about equalities and dis-equalities and any other numerical domains supported by the Apron library [JM09], e.g., octagons [Min06], convex polyhedra [CH78], intervals [CC77]. In the following, for simplicity, we sometimes present numerical constraints directly than specific numerical abstractions, present $\overline{\mathbf{guard}}_{\overline{\mathcal{N}}}(\overline{\mathrm{n}}, \mathrm{c}_{\overline{\mathcal{V}}})$ as $\overline{\mathrm{n}} \wedge \mathrm{c}_{\overline{\mathcal{V}}}$ and $\overline{\mathbf{prove}}_{\overline{\mathcal{N}}}(\overline{\mathrm{n}}, \mathrm{c}_{\overline{\mathcal{V}}}) =$ **true** as $\overline{\mathrm{n}} \implies \mathrm{c}_{\overline{\mathcal{V}}}$.

Disjunctive abstract states. In practice, shape analyses often rely on disjunctive abstract states $\overline{\mathrm{d}} \in \overline{\mathcal{D}}$ of the form $\overline{\mathrm{m}}_1 \vee \overline{\mathrm{m}}_2 \vee \ldots \vee \overline{\mathrm{m}}_n$ ($n \in \mathbb{N}$). The reason is that disjunctive abstract states $\overline{\mathrm{d}} \in \overline{\mathcal{D}}$ enable more precise abstraction of memory states than abstract states $\overline{\mathrm{m}} \in \overline{\mathcal{M}_\omega}$. Indeed, in real-world program analyses, a single abstract state $\overline{\mathrm{m}} \in \overline{\mathcal{M}_\omega}$ often cannot capture precise structural properties of a set of concrete memory states. The concretization of disjunctive abstract states is defined as the union of the concretization of each abstract state:

$$\gamma_{\overline{\mathcal{D}}}(\overline{\mathrm{m}}_1 \vee \overline{\mathrm{m}}_2 \vee \ldots \vee \overline{\mathrm{m}}_n) = \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}_1) \cup \ldots \cup \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}_n)$$

2.4 Abstract Semantics

In this section, we discuss the abstract semantics for the abstract states $\overline{\mathcal{M}_\omega}$ introduced in the previous section, from which the abstract semantics for the disjunctive abstract states $\overline{\mathcal{D}}$ can be

$$\overline{\mathbf{read}}(\overline{v},\mathtt{f},(\overline{\mathrm{g}},\overline{\mathrm{n}})) ::= \begin{cases} \overline{v}_1 & \textit{if } \exists \overline{\mathrm{g}}_1, \overline{v}_1,\ \overline{\mathrm{g}} = \overline{\mathrm{g}}_1 * \overline{v}.\mathtt{f} \mapsto \overline{v}_1 \\ \top & \textit{otherwise} \end{cases}$$

$$\overline{\mathbf{write}}(\overline{v},\mathtt{f},\overline{v}_1,(\overline{\mathrm{g}},\overline{\mathrm{n}})) ::= \begin{cases} (\overline{\mathrm{g}}_1 * \overline{v}.\mathtt{f} \mapsto \overline{v}_1, \overline{\mathrm{n}}) & \textit{if } \exists \overline{\mathrm{g}}_1, \overline{v}_2,\ \overline{\mathrm{g}} = \overline{\mathrm{g}}_1 * \overline{v}.\mathtt{f} \mapsto \overline{v}_2 \\ \top & \textit{otherwise} \end{cases}$$

$\overline{\mathbf{create}}(n,(\overline{\mathrm{g}},\overline{\mathrm{n}}))$ *returns*

$$(\alpha, (\overline{\mathrm{g}} * \alpha \mapsto \alpha_1 * \ldots * \alpha + 4(n-1) \mapsto \alpha_n, \overline{\mathrm{n}})),\ (\alpha, (\overline{\mathrm{g}}, \overline{\mathrm{n}} \wedge \alpha = 0))$$

$$\textit{where } \alpha, \alpha_1, \ldots, \alpha_n \textit{are fresh symbolic variables}$$

$$\overline{\mathbf{delete}}(\overline{v},n,(\overline{\mathrm{g}},\overline{\mathrm{n}})) ::= \begin{cases} (\overline{\mathrm{g}}_1, \overline{\mathrm{n}}) & \textit{if } \exists \overline{\mathrm{g}}_1, \overline{v}_1, \ldots, \overline{v}_n,\ \overline{\mathrm{g}} = \overline{\mathrm{g}}_1 * \overline{v} \mapsto \overline{v}_1 * \ldots * \overline{v} + 4(n-1) \mapsto \overline{v}_n \\ \top & \textit{otherwise} \end{cases}$$

Figure 2.5: Abstract store operations

derived naturally.

2.4.1 Abstract Store Operations

As presented in Figures 2.3 and 2.4, both the concrete program semantics and the expression evaluation involve the store operations shown in Figure 2.1.2. Therefore, in order to define the abstract semantics of program statements, we first need to provide abstract store operations. For simplicity, we assume here that the memory regions that store operations operate on are fully described by points-to edges. The case where the memory cells involved are summarized in inductive or segment predicates will be dealt with unfolding operations of summary predicates in Section 2.4.2.

Corresponding to the concrete store operations **read**, **write**, **create** and **delete**, we have the following abstract store operations:

$$\begin{array}{lcl} \overline{\mathbf{read}} & \in & \overline{\mathcal{V}} \times \mathcal{F} \times \overline{\mathcal{M}_\omega} \to \overline{\mathcal{V}} \uplus \{\top, \bot\} \\ \overline{\mathbf{write}} & \in & \overline{\mathcal{V}} \times \mathcal{F} \times \overline{\mathcal{V}} \times \overline{\mathcal{M}_\omega} \to \overline{\mathcal{M}_\omega} \\ \overline{\mathbf{create}} & \in & \mathbb{N} \times \overline{\mathcal{M}_\omega} \to (\overline{\mathcal{V}} \times \overline{\mathcal{M}_\omega})^2 \uplus \{\bot, \top\} \\ \overline{\mathbf{delete}} & \in & \overline{\mathcal{V}} \times \mathbb{N} \times \overline{\mathcal{M}_\omega} \to \overline{\mathcal{M}_\omega} \end{array}$$

All the operations are $\bot$−strict and $\top$−strict, i.e. when the input abstract state is $\bot$, the operation returns $\bot$ and the same for $\top$. We note that the $\top$ abstract state accounts for crashes of program analysis either due to program errors or the loss of precision during the analysis. Further characterizations of the abstract operations are shown in Figure 2.5.

Intuitively, the abstract operation $\overline{\mathbf{read}}(\overline{v},\mathtt{f},(\overline{\mathrm{g}},\overline{\mathrm{n}}))$ attempts to read the abstract value of the memory cell at abstract address $\overline{v} + \mathtt{f}$ by searching a points-to edge starting at $\overline{v}.\mathtt{f}$, e.g. $\overline{v}.\mathtt{f} \mapsto \overline{v}_1$, in the abstract shape graph $\overline{\mathrm{g}}$ and returns the abstract value $\overline{v}_1$. The abstract $\overline{\mathbf{write}}(\overline{v},\mathtt{f},\overline{v}_1,(\overline{\mathrm{g}},\overline{\mathrm{n}}))$ operation attempts to write the abstract value $\overline{v}_1$ into the memory cell at abstract address $\overline{v} + \mathtt{f}$. Therefore, it also searches for a points-to edge starting at $\overline{v}.\mathtt{f}$, e.g.

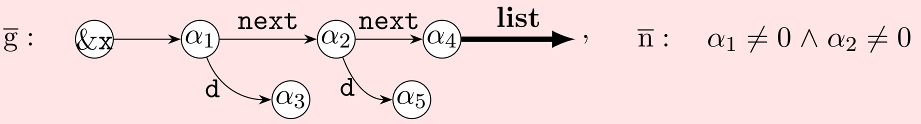
$\overline{v}.\mathtt{f} \mapsto \overline{v}_2$, in the abstract shape graph $\overline{\mathrm{g}}$, and then updates the points-to edge to $\overline{v}.\mathtt{f} \mapsto \overline{v}_1$. Since the concrete operation $\mathbf{create}(n, \sigma)$ may succeed (returning the base address of the newly allocated block and the new store) or fail (returning the null address 0), to over-approximate such behaviors, the abstract operation $\overline{\mathbf{create}}(n, (\overline{\mathrm{g}}, \overline{\mathrm{n}}))$ returns a pair made up of $(\alpha, (\overline{\mathrm{g}} * \alpha \mapsto \alpha_1 * \ldots * \alpha + 4(n-1) \mapsto \alpha_n, \overline{\mathrm{n}}))$ and $(\alpha, (\overline{\mathrm{g}}, \overline{\mathrm{n}} \wedge \alpha = 0))$. The first pair abstracts the concrete results of successful allocation by adding n continuous points-to edges starting at abstract addresses α in the abstract shape graph $\overline{\mathrm{g}}$, while the second pair corresponds to the case of a failure by returning a symbolic variable α that is constrained to be 0 in the numerical abstraction, denoted by $\overline{\mathrm{n}} \wedge \alpha = 0$. Conversely, the abstract operation $\overline{\mathbf{delete}}(\overline{v}, n, (\overline{\mathrm{g}}, \overline{\mathrm{n}}))$ removes n continuous points-to edges starting from abstract address $\overline{v}$ from the abstract shape graph. We note that the $\top$ abstract state may be returned during the store operations either due to imprecise abstract states or invalid store operations that cause the error state ω in the concrete part.

The abstract store operations are sound. For simplicity, we only present the soundness condition of abstract operation $\overline{\mathbf{read}}$ and the soundness conditions of other operations are similar.

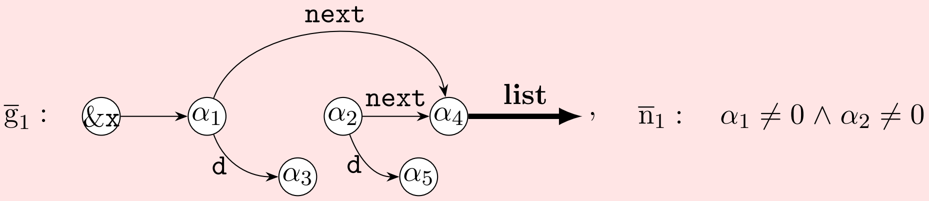
Condition 2.6 (Soundness of abstract read operation). *The abstract operation* $\overline{\mathbf{read}}$ *is sound if and only if the following property holds:*

$$\begin{array}{l} \overline{\mathbf{read}}(\overline{v}, \mathtt{f}, (\overline{\mathrm{g}}, \overline{\mathrm{n}})) = \overline{v}_1 \\ \Longrightarrow \\ \forall(\sigma, \mu) \vDash (\overline{\mathrm{g}}, \overline{\mathrm{n}}), \mathbf{read}(\mu(\overline{v}), \mathtt{f}, \sigma) \in \{\mu(\overline{v}_1) \mid (\sigma, \mu) \vDash (\overline{\mathrm{g}}, \overline{\mathrm{n}}), \} \end{array}$$

Example 2.10 (Abstract store operations). Let us consider some abstract store operations on the abstract state below:

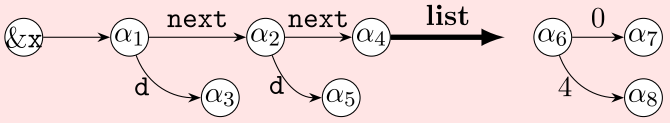

$\overline{\mathrm{g}}$:
&x
α_1
next
α_2
next
α_4
list
d
α_3
d
α_5
,
$\overline{\mathrm{n}}$: $\alpha_1 \neq 0 \wedge \alpha_2 \neq 0$

- $\overline{\mathbf{read}}(\alpha_1, \mathtt{next}, (\overline{\mathrm{g}}, \overline{\mathrm{n}})) = \alpha_2$ $\quad\overline{\mathbf{read}}(\alpha_1, \mathtt{d}, (\overline{\mathrm{g}}, \overline{\mathrm{n}})) = \alpha_3$
- $\overline{\mathbf{write}}(\alpha_1, \mathtt{next}, \alpha_4, (\overline{\mathrm{g}}, \overline{\mathrm{n}})) = (\overline{\mathrm{g}}_1, \overline{\mathrm{n}}_1)$ (the operation flips the `next` edge from node α_1 in $\overline{\mathrm{g}}$ to node α_4):


$\overline{\mathrm{g}}_1$:
&x
α_1
next
α_2
next
α_4
list
d
α_3
d
α_5
,
$\overline{\mathrm{n}}_1$: $\alpha_1 \neq 0 \wedge \alpha_2 \neq 0$

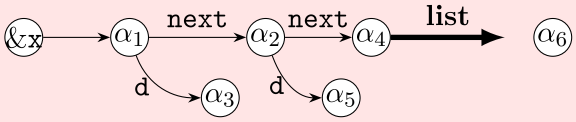
- $\overline{\mathbf{create}}(2, (\overline{\mathrm{g}}, \overline{\mathrm{n}})) = ((\alpha_6, (\overline{\mathrm{g}}_2, \overline{\mathrm{n}}_2)), (\alpha_6, (\overline{\mathrm{g}}_3, \overline{\mathrm{n}}_3)))$:

$\overline{\mathrm{g}}_2$:



, $\overline{\mathrm{n}}_2$: $\alpha_1 \neq 0 \land \alpha_2 \neq 0$

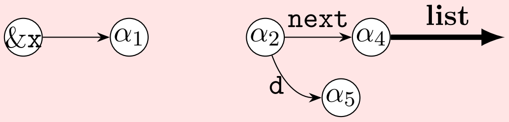
$\overline{\mathrm{g}}_3$:



, $\overline{\mathrm{n}}_3$: $\alpha_1 \neq 0 \land \alpha_2 \neq 0 \land \alpha_6 = 0$

- $\overline{\textbf{delete}}(\alpha_1, 2, (\overline{\mathrm{g}}, \overline{\mathrm{n}})) = (\overline{\mathrm{g}}_4, \overline{\mathrm{n}}_4)$ (the list tail can no longer be dereferenced from variable `x` after the first list element has been removed):

$\overline{\mathrm{g}}_4$:

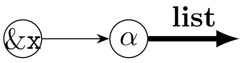


, $\overline{\mathrm{n}}_4$: $\alpha_1 \neq 0 \land \alpha_2 \neq 0$

2.4.2 Unfolding

In Section 2.4.1, we describe store operations based on the assumption that the memory regions that store operations operate on are fully described by points-to edges. However, store operations may involve memory cells that are summarized in inductive or segment predicates, for example the store operation $\overline{\textbf{read}}(\alpha, \texttt{next}, (\overline{\mathrm{g}}, \overline{\mathrm{n}}))$ reads the abstract value at abstract address $\alpha + \texttt{next}$ which is summarized in the inductive predicate $\alpha \cdot \textbf{list}$:

$\overline{\mathrm{g}}$:



, $\overline{\mathrm{n}}$: $\alpha \neq 0$

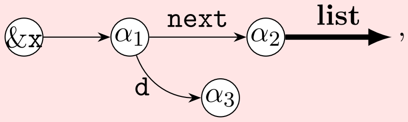
In such cases, in order to precisely support the store operations, the analysis needs to unfold summary predicates to expose points-to edges that store operations operate on.

Unfolding of inductive edges. The principle of unfolding an inductive predicate (e.g., $\alpha \cdot$ **list**) is to rewrite the inductive predicate by syntactically expanding all the rules of the inductive definition. Specifically, the unfolding operation $\overline{\textbf{unfold}} \in \overline{\mathcal{V}} \times \overline{\mathcal{M}_\omega} \to \overline{\mathcal{D}}$ takes an abstract address $\overline{v}$ which is the origin of an inductive edge and memory cells from which need to be materialized, and an abstract state $\overline{\mathrm{m}} \in \overline{\mathcal{M}_\omega}$. It outputs a disjunctive abstract state $\overline{\mathrm{d}} = \overline{\mathrm{m}}_1 \lor \ldots \lor \overline{\mathrm{m}}_k \in \overline{\mathcal{D}}$, where each abstract state $\overline{\mathrm{m}}_i$ refines the abstract state $\overline{\mathrm{m}}$ following a rule of the inductive definition. For simplicity, we present here only the soundness condition and an example of unfolding; the formal definition of unfolding is given in Chapter 6.

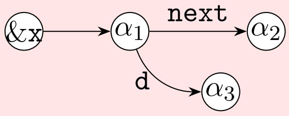
Condition 2.7 (Soundness of $\overline{\textbf{unfold}}$). *The unfolding operator* $\overline{\textbf{unfold}} \in \overline{\mathcal{V}} \times \overline{\mathcal{M}_\omega} \to \overline{\mathcal{D}}$ *is sound if and only if:*

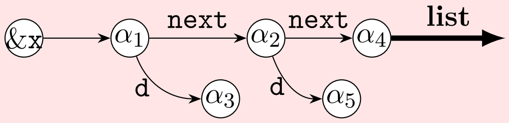
$$\forall \overline{\mathrm{m}} \in \overline{\mathcal{M}_\omega},\ \overline{\textbf{unfold}}(\overline{v}, \overline{\mathrm{m}}) = \overline{\mathrm{m}}_1 \lor \ldots \lor \overline{\mathrm{m}}_n \implies \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}) \subseteq \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}_1) \cup \ldots \cup \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}_n)$$

Example 2.11 (Unfolding a list inductive predicate). Let us consider the unfolding of the inductive predicate $\alpha_2 \cdot \mathbf{list}$ from the abstract state below:

$\overline{\mathrm{g}}$:  , $\overline{\mathrm{n}}$: $\alpha_1 \neq 0$

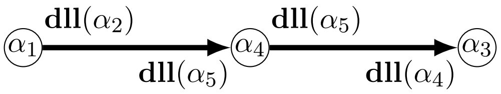
As the **list** inductive definition contains two rules, one corresponding to the empty list and the other to the non-empty list, unfolding the inductive predicate $\alpha_2 \cdot \mathbf{list}$ leads to the two abstract states below:

$\overline{\mathrm{g}}_1$:  , $\overline{\mathrm{n}}_1$: $\alpha_1 \neq 0 \wedge \alpha_2 = 0$

$\overline{\mathrm{g}}_2$:  , $\overline{\mathrm{n}}_2$: $\alpha_1 \neq 0 \wedge \alpha_2 \neq 0$

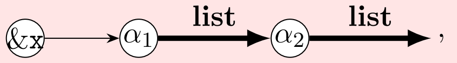
The first abstract state $(\overline{\mathrm{g}}_1, \overline{\mathrm{n}}_1)$ corresponds to the empty list case, where the list predicate is replaced by an empty region, and the numerical abstraction $\overline{\mathrm{n}}_1$ guarantees that α_2 is a null address. The second abstract state $(\overline{\mathrm{g}}_2, \overline{\mathrm{n}}_2)$ corresponds to the non-empty list case, where the list element at α_2 is completely exposed and α_2 is constrained to be non-null in $\overline{\mathrm{n}}_2$.

Unfolding of segment predicates. Segment predicates may need to be unfolded from either end, forward or backward. Forward unfolding is very useful for analyzing forward traversal programs of data structures, and conversely backward unfolding is for backward data structure traversal. Forward segment unfolding relies on rewriting the segment predicate by syntactically unfolding all the rules of the segment definition, which is similar to the unfolding of inductive predicates. while backward unfolding needs to first split a segment predicate into a separating conjunction of two segments and then performs forward unfolding at the split point. For example, backward unfolding at the previous node α_4 of node α_3 of the doubly linked list segment predicate $\alpha_1 \cdot \mathbf{dll}(\alpha_2) *\!\!= \alpha_3 \cdot \mathbf{dll}(\alpha_4)$ needs to split the segment at node α_4 (node α_5 denotes the previous node of node α_4):



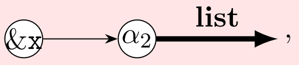
The soundness condition of segment unfolding can be expressed in the same way as in Definition 2.7.

Example 2.12 (Unfolding a list segment predicate). Let us consider unfolding the list segment from node α_1 to node α_2 in the abstract state below:

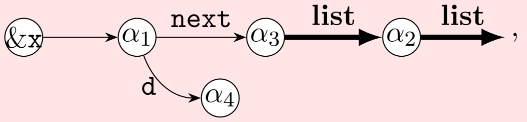
$\overline{\mathrm{g}}$:


, $\overline{\mathrm{n}}$: $\alpha_1 \neq 0$

According to the list segment definition, we get the two abstract states below:

$\overline{\mathrm{g}}_1$:


, $\overline{\mathrm{n}}_1$: $\alpha_1 \neq 0 \wedge \alpha_1 = \alpha_2$

$\overline{\mathrm{g}}_2$:


, $\overline{\mathrm{n}}_2$: $\alpha_1 \neq 0 \wedge \alpha_1 \neq \alpha_2$

The abstract state $(\overline{\mathrm{g}}_1, \overline{\mathrm{n}}_1)$ corresponds to the empty segment case, i.e., the segment is unfolded into an empty region and a constraint specifying α_1 is equal to α_2 is added into the numerical abstraction. In practice, the unfolding should then rename α_2 into α_1 everywhere in the graph and continue unfolding the inductive edge from node α_1 to expose memory cells at α_1. The abstract state $(\overline{\mathrm{g}}_2, \overline{\mathrm{n}}_2)$ refines the list segment to a non-empty list segment, which exposes the memory cells at address α_1 and imposes the constraint that α_1 is not equal to α_2 in $\overline{\mathrm{n}}_2$.

2.4.3 Abstract Evaluation of Left and Right Value Expressions

As the definition of right value expressions (presented in Figure 2.1) involves arithmetic operators, therefore we first formalize the abstract semantics of arithmetic operators.

Abstract semantics of arithmetic operators. While the concrete semantics of arithmetic operators are quite simple, precisely defining the abstract semantics of the operators is fairly complex as they may involve both abstract shape graphs and numerical abstractions. Figure 2.6 shows the abstract semantics of the additive operator + and the equal test operator ==. Intuitively, the abstract operator $\overline{[\![+]\!]}$ takes two abstract values $\overline{v}_1 \in \overline{\mathcal{V}}$ and $\overline{v}_2 \in \overline{\mathcal{V}}$ (additive operands), an abstract state $(\overline{\mathrm{g}}, \overline{\mathrm{n}}) \in \overline{\mathcal{M}_\omega}$, and returns a pair made up of an abstract value $\overline{v}_3$ and a new abstract state $(\overline{\mathrm{g}}, \overline{\mathbf{guard}}_{\overline{\mathcal{N}}}(\overline{\mathrm{n}}, \overline{v}_3 = \overline{v}_1 + \overline{v}_2))$ where $\overline{v}_3$ is enforced to the sum of $\overline{v}_1$ and $\overline{v}_2$ in the abstract state. Similarly, the abstract operator $\overline{[\![==]\!]}$ also takes two abstract values $\overline{v}_1$ and $\overline{v}_2$, an abstract state $(\overline{\mathrm{g}}, \overline{\mathrm{n}}) \in \overline{\mathcal{M}_\omega}$, but returns two pairs made up of boolean values (**true** and **false**) and abstract states which respectively enforce equality and dis-equality condition on $\overline{v}_1$ and $\overline{v}_2$. For precision, the equality (resp., dis-equality) condition should be not only enforced on the numerical abstraction $\overline{\mathrm{n}}$, denoted as $\overline{\mathbf{guard}}_{\overline{\mathcal{N}}}(\overline{\mathrm{n}}, \overline{v}_1 = \overline{v}_2))$ (resp., $\overline{\mathbf{guard}}_{\overline{\mathcal{N}}}(\overline{\mathrm{n}}, \overline{v}_1 \neq \overline{v}_2)))$, but also on the abstract shape graph $\overline{\mathrm{g}}$, denoted as $\overline{\mathbf{guard}}_{\overline{\mathcal{G}}}(\overline{\mathrm{g}}, \overline{v}_1 = \overline{v}_2)$ (resp., $\overline{\mathbf{guard}}_{\overline{\mathcal{G}}}(\overline{\mathrm{g}}, \overline{v}_1 \neq \overline{v}_2)$. We note that $\overline{\mathbf{guard}}_{\overline{\mathcal{G}}} \in \overline{\mathcal{G}} \times \mathcal{C}_{\overline{\mathcal{V}}} \rightarrow \overline{\mathcal{G}}$ is an operator of the abstract shape domain, which takes an abstract shape graph $\overline{\mathrm{g}} \in \overline{\mathcal{G}}$, a numerical constraint $\mathrm{c}_{\overline{\mathcal{V}}} \in \mathcal{C}_{\overline{\mathcal{V}}}$, and returns a new abstract shape graph where the numerical constraint has been enforced. Moreover, the abstract semantics $\overline{[\![\oplus]\!]}$

$$\begin{aligned}
&\overline{[\![+]\!]}(\overline{v}_1, \overline{v}_2, (\overline{\mathrm{g}}, \overline{\mathrm{n}})) = \\
&\qquad (\overline{v}_3, (\overline{\mathrm{g}}, \overline{\mathbf{guard}_{\mathcal{N}}}(\overline{\mathrm{n}}, \overline{v}_3 = \overline{v}_1 + \overline{v}_2))) \qquad \textit{($\overline{v}_3$ is a fresh symbolic variable)} \\[1em]
&\overline{[\![==]\!]}(\overline{v}_1, \overline{v}_2, (\overline{\mathrm{g}}, \overline{\mathrm{n}})) = \\
&\qquad (\mathbf{true}, (\overline{\mathbf{guard}_{\mathcal{G}}}(\overline{\mathrm{g}}, \overline{v}_1 = \overline{v}_2), \overline{\mathbf{guard}_{\mathcal{N}}}(\overline{\mathrm{n}}, \overline{v}_1 = \overline{v}_2))), \\
&\textit{and } (\mathbf{false}, (\overline{\mathbf{guard}_{\mathcal{G}}}(\overline{\mathrm{g}}, \overline{v}_1 \neq \overline{v}_2), \overline{\mathbf{guard}_{\mathcal{N}}}(\overline{\mathrm{n}}, \overline{v}_1 \neq \overline{v}_2)))
\end{aligned}$$

Figure 2.6: Abstract arithmetic operators

of any operator is a $\bot-$strict and $\top-$strict function that returns $\bot$ and $\top$ respectively when the input abstract state is $\bot$ and $\top$.

For simplicity, we only present here the soundness condition of the abstract semantics of the additive operator and the soundness condition of other abstract arithmetic operators can be defined similarly.

Condition 2.8 (Soundness of abstract operator $\overline{[\![+]\!]}$). *The abstract operation $\overline{[\![+]\!]}$ is sound if and only if the following property holds:*

$$\begin{aligned}
&\overline{[\![\oplus]\!]}(\overline{v}_1, \overline{v}_2, (\overline{\mathrm{g}}, \overline{\mathrm{n}})) = (\overline{v}_3, (\overline{\mathrm{g}}_1, \overline{\mathrm{n}}_1)) \implies \\
&\qquad \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{g}}, \overline{\mathrm{n}}) \subseteq \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{g}}_1, \overline{\mathrm{n}}_1) \\
&\quad \wedge \{[\![+]\!](\mu(\overline{v}_1), \mu(\overline{v}_2)) \mid (\sigma, \mu) \vDash (\overline{\mathrm{g}}_1, \overline{\mathrm{n}}_1)\} \subseteq \{\mu(\overline{v}_3) \mid (\sigma, \mu) \vDash (\overline{\mathrm{g}}_1, \overline{\mathrm{n}}_1)\}
\end{aligned}$$

Example 2.13 (Abstract equality test operation). Let us consider the abstract state $(\overline{\mathrm{g}}, \overline{\mathrm{n}})$ of Example 2.10. The abstract equality test operation $\overline{[\![==]\!]}(\alpha_1, \alpha_2, (\overline{\mathrm{g}}, \overline{\mathrm{n}}))$ tests the equality of abstract value α_1 and α_2 in the abstract state $(\overline{\mathrm{g}}, \overline{\mathrm{n}})$. According to Figure 2.6, the abstract semantics relies on both the equality and dis-equality test operations on the shape graph (i.e., $\overline{\mathbf{guard}_{\mathcal{G}}}(\overline{\mathrm{g}}, \overline{v}_1 = \overline{v}_2)$ and $\overline{\mathbf{guard}_{\mathcal{G}}}(\overline{\mathrm{g}}, \overline{v}_1 \neq \overline{v}_2)$) and on the numerical abstraction (i.e., $\overline{\mathbf{guard}_{\mathcal{N}}}(\overline{\mathrm{n}}, \overline{v}_1 = \overline{v}_2)$ and $\overline{\mathbf{guard}_{\mathcal{N}}}(\overline{\mathrm{n}}, \overline{v}_1 \neq \overline{v}_2)$). Indeed, in the abstract graph $\overline{\mathrm{g}}$, the test operations can be done in an exact manner: the separating conjunction of the points-to edges α_1`.next` $\mapsto \alpha_2$ and α_2`.next` $\mapsto \alpha_4$ entails that α_1 and α_2 are addresses of disjoint memory cells, thus, they cannot be equal. Therefore, we can get that $\overline{\mathbf{guard}_{\mathcal{G}}}(\overline{\mathrm{g}}, \overline{v}_1 = \overline{v}_2) = \bot$ and $\overline{\mathbf{guard}_{\mathcal{G}}}(\overline{\mathrm{g}}, \overline{v}_1 \neq \overline{v}_2) = \overline{\mathrm{g}}$. That is, $\overline{[\![==]\!]}(\alpha_1, \alpha_2, (\overline{\mathrm{g}}, \overline{\mathrm{n}}))$ returns the two pairs $(\mathbf{true}, \bot)$ and $(\mathbf{false}, (\overline{\mathrm{g}}, \overline{\mathbf{guard}_{\mathcal{N}}}(\overline{\mathrm{n}}, \overline{v}_1 \neq \overline{v}_2)))$.

Abstract evaluation of left and right value expressions. Intuitively, given an abstract state $\overline{\mathrm{m}}$, a left value expression `l` is possibly evaluated into an abstract address $(\overline{v}_1, \mathtt{f}) \in \overline{\mathcal{V}} \times \mathcal{F}$, top ($\top$) that accounts for a failure of the analysis, or bottom ($\bot$). However, unfolding may be triggered during the evaluation, which outputs a disjunctive abstract state $\overline{\mathrm{m}}_1 \vee \ldots \vee \overline{\mathrm{m}}_k$, on each disjunct of which the evaluation is then performed on. Therefore, the abstract semantics of left value expressions is a $\bot-$strict and $\top-$strict function of the form:

$$\overline{\textbf{eval}_l[\![\texttt{l}]\!]} \in \overline{\mathcal{M}_\omega} \to \bigvee((\overline{\mathcal{V}} \times \mathcal{F}) \times \overline{\mathcal{M}_\omega}) \uplus \{\bot, \top\}$$

We note that, in the following we sometimes write $(\overline{v}_1, \texttt{f})$ as $\overline{v}_1 + \texttt{f}$ or $\overline{v}_1$ when the offset is 0.

Similarly, unfolding may also be triggered in the evaluation of a right value expression $\texttt{r}$, and thus a right value expression $\texttt{r}$ is either evaluated into a disjunction of pairs of abstract values and abstract states, or $\top$, $\bot$. The abstract semantics of right value expressions is defined as $\bot-$strict and $\top-$strict functions of the form:

$$\overline{\textbf{eval}_r[\![\texttt{r}]\!]} \in \overline{\mathcal{M}_\omega} \to \bigvee(\overline{\mathcal{V}} \times \overline{\mathcal{M}_\omega}) \uplus \{\bot, \top\}$$

Similar to the concrete expression evaluation shown in Figure 2.3, a formal definition of abstract expression evaluation can be derived based on the syntax of expressions and the abstract semantics $\overline{\textbf{read}}$, $\overline{[\![\oplus]\!]}$ and $\overline{\textbf{unfold}}$, and thus for simplicity is omitted here. Instead, we present the soundness condition of the abstract right value expression evaluation and the soundness condition of the abstract left value expression evaluation can be defined similarly.

Condition 2.9 (Soundness of the abstract evaluation of the right value expression). *The abstract operation* $\overline{\textbf{eval}_r[\![r]\!]}$ *is sound if and only if the following property holds:*

$$\begin{cases} \overline{\textbf{eval}_r[\![r]\!]}(\overline{\mathrm{g}}, \overline{\mathrm{n}}) = \bigvee_{1 \le i \le n}(\overline{v}_i, (\overline{\mathrm{g}}_i, \overline{\mathrm{n}}_i)) \implies \\ \qquad \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{g}}, \overline{\mathrm{n}}) \subseteq \bigcup_{1 \le i \le n} \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{g}}_i, \overline{\mathrm{n}}_i) \\ \qquad \wedge\ \forall 1 \le i \le n, \{\textbf{eval}_r[\![r]\!](\varepsilon, \sigma) \mid (\varepsilon, \sigma) \in \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{g}}_i, \overline{\mathrm{n}}_i)\} \subseteq \{\mu(\overline{v}_i) \mid (\sigma, \mu) \vDash (\overline{\mathrm{g}}_i, \overline{\mathrm{n}}_i)\} \\ \overline{\textbf{eval}_r[\![r]\!]}(\overline{\mathrm{g}}, \overline{\mathrm{n}}) = \bot \implies \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{g}}, \overline{\mathrm{n}}) = \emptyset \end{cases}$$

Example 2.14 (Abstract evaluations). Let us consider the abstract state $(\overline{\mathrm{g}}, \overline{\mathrm{n}})$ of Example 2.10. The abstract left evaluation of $\texttt{x}$ is the symbolic address &x of variable x, i.e., $\overline{\textbf{eval}_l[\![\mathrm{x}]\!]}(\overline{\mathrm{g}}, \overline{\mathrm{n}}) = \&\mathrm{x}$, while the abstract right evaluation of $\texttt{x}$ is the symbolic variable α_1 which abstracts the value of varibale x, i.e., $\overline{\textbf{eval}_r[\![\mathrm{x}]\!]}(\overline{\mathrm{g}}, \overline{\mathrm{n}}) = \alpha_1$. Similarly, the abstract left evaluation of (*x).next is the abstract address $\alpha_1 + \texttt{next}$, i.e., $\overline{\textbf{eval}_l[\![(\texttt{*x}).\texttt{next}]\!]}(\overline{\mathrm{g}}, \overline{\mathrm{n}}) = \alpha_1 + \texttt{next}$, while the abstract right evaluation of (*x).next is the abstract value α_2 at abstract address $\alpha_1 + \texttt{next}$, i.e., $\overline{\textbf{eval}_r[\![(\texttt{*x}).\texttt{next}]\!]}(\overline{\mathrm{g}}, \overline{\mathrm{n}}) = \alpha_2$.

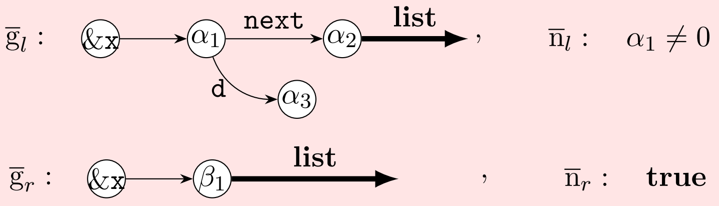
2.4.4 Folding

Section 2.4.2 shows the unfolding of inductive and segment edges, which is used in some abstract transfer functions. In contrast to unfolding, *folding* [CR08] folds back memory regions into summary predicates and is required in abstract lattice operators of memory states: $\sqsubseteq_{\overline{\mathcal{M}_\omega}}$, $\sqcup_{\overline{\mathcal{M}_\omega}}$ and $\triangledown_{\overline{\mathcal{M}_\omega}}$. However, folding turns out to be much harder than unfolding, and the main difficulties lie in finding memory regions that are candidates for folding and searching for folded summary predicates. In the following, we present only the principles of the operators that need folding and refer the readers to Section 6.3 for more details.

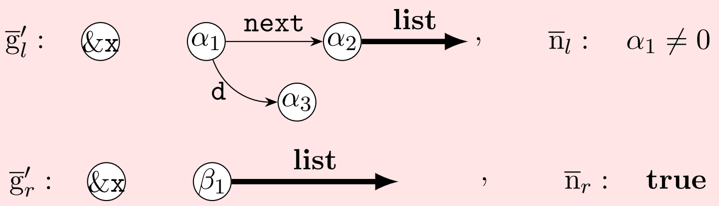
Inclusion checking $\sqsubseteq_{\overline{\mathcal{M}_\omega}}$. Inclusion checking takes two abstract states $(\overline{\mathrm{g}}_l, \overline{\mathrm{n}}_l)$ and $(\overline{\mathrm{g}}_r, \overline{\mathrm{n}}_r)$, and attempts to establish that the concretization of one abstract state $(\overline{\mathrm{g}}_l, \overline{\mathrm{n}}_l)$ is included in the concretization of the other abstract state $(\overline{\mathrm{g}}_r, \overline{\mathrm{n}}_r)$, that is $\gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{g}}_l, \overline{\mathrm{n}}_l) \subseteq \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{g}}_r, \overline{\mathrm{n}}_r)$. A first use of inclusion checking is to prove an abstract loop invariant has been reached after a series of abstract iterations. A second use is to determine whether a memory region can be over-approximated by a summary predicate in joining $\sqcup_{\overline{\mathcal{M}_\omega}}$ and widening $\triangledown_{\overline{\mathcal{M}_\omega}}$. In addition, inclusion checking is also used in proving post-conditions.

Specifically, the inclusion checking algorithm first aims to set up the inclusion of abstract shape graphs $\overline{\mathrm{g}}_l \sqsubseteq_{\overline{\mathcal{G}}} \overline{\mathrm{g}}_r$ based on a series of syntactic rewriting rules. As the names of symbolic variables in $\overline{\mathrm{g}}_l$ may have nothing to do with the names in $\overline{\mathrm{g}}_r$ (symbolic variables are existential qualified variables), the rewriting rules have do deal with the renaming of symbolic variables. Thus, a rewriting rule either matches identical memory regions of $\overline{\mathrm{g}}_l$ and $\overline{\mathrm{g}}_r$ up to node renaming or weakens a memory region of $\overline{\mathrm{g}}_l$ (resp., $\overline{\mathrm{g}}_r$) into a memory region of $\overline{\mathrm{g}}_r$ (resp., $\overline{\mathrm{g}}_l$). When the inclusion holds on abstract shape graphs, it then applies the numerical inclusion checking operator $\sqsubseteq_{\overline{\mathcal{N}}}$ to establish that $\overline{\mathrm{n}}_l \sqsubseteq_{\overline{\mathcal{N}}} \overline{\mathrm{n}}_r$.

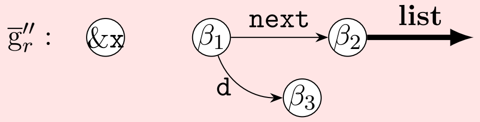
Example 2.15 (Inclusion checking). Let us try to check the inclusion of $(\overline{\mathrm{g}}_l, \overline{\mathrm{n}}_l)$ into $(\overline{\mathrm{g}}_r, \overline{\mathrm{n}}_r)$, where:


$\overline{\mathrm{g}}_l$:
&x
α_1
next
α_2
list
d
α_3
,
$\overline{\mathrm{n}}_l$: $\alpha_1 \neq 0$
$\overline{\mathrm{g}}_r$:
&x
β_1
list
,
$\overline{\mathrm{n}}_r$: **true**

As both abstract shape graphs contain a points-to edge from &x that are identical up to renaming node β_1 of $\overline{\mathrm{g}}_r$ into node α_1 of $\overline{\mathrm{g}}_l$, the inclusion checking algorithm thus binds β_1 to α_1 (i.e., $\beta_1 \mapsto \alpha_1$), and conclude that the points-to edge &x $\mapsto \alpha_1$ is included into &x $\mapsto \beta_1$. Then, the problem is reduced to checking the inclusion of $(\overline{\mathrm{g}}'_l, \overline{\mathrm{n}}_l)$ into $(\overline{\mathrm{g}}'_r, \overline{\mathrm{n}}_r)$ with node binding $\{\beta_1 \mapsto \alpha_1\}$, where:


$\overline{\mathrm{g}}'_l$:
&x
α_1
next
α_2
list
d
α_3
,
$\overline{\mathrm{n}}_l$: $\alpha_1 \neq 0$
$\overline{\mathrm{g}}'_r$:
&x
β_1
list
,
$\overline{\mathrm{n}}_r$: **true**

As $\overline{\mathrm{g}}'_l$ and $\overline{\mathrm{g}}'_r$ no longer have identical predicates, the algorithm thus tries to weaken $\overline{\mathrm{g}}'_l$ into the inductive predicate $\beta_1 \cdot$ **list** of $\overline{\mathrm{g}}'_r$. To do that, it unfolds the inductive predicate $\beta_1 \cdot$ **list**. One unfolding result is $\overline{\mathrm{g}}''_r$ with numerical constraint $\beta_1 \neq 0$, where:



The abstract graph $\overline{\mathrm{g}}''_r$ is identical with $\overline{\mathrm{g}}'_l$ up to node renaming $\{\beta_1 \mapsto \alpha_1, \beta_2 \mapsto \alpha_2, \beta_3 \mapsto \alpha_3\}$. The numerical constraint $\beta_1 \neq 0$ and numerical abstraction $\overline{\mathrm{n}}_r$ can be entailed by $\overline{\mathrm{n}}_l$ up to the node renaming. Therefore, the inclusion holds.

Joining $\sqcup_{\overline{\mathcal{M}_\omega}}$ and widening $\triangledown_{\overline{\mathcal{M}_\omega}}$. The joining operator $\sqcup_{\overline{\mathcal{M}_\omega}}$ takes two abstract states $(\overline{\mathrm{g}}_l, \overline{\mathrm{n}}_l)$ and $(\overline{\mathrm{g}}_r, \overline{\mathrm{n}}_r)$, and attempts to compute an abstract state $(\overline{\mathrm{g}}_o, \overline{\mathrm{n}}_o)$ that over-approximates both arguments, that is $\gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{g}}_l, \overline{\mathrm{n}}_l) \subseteq \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{g}}_o, \overline{\mathrm{n}}_o)$ and $\gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{g}}_r, \overline{\mathrm{n}}_r) \subseteq \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{g}}_o, \overline{\mathrm{n}}_o)$.

Similar to the inclusion checking, the joining algorithm also first joins abstract shape graphs, i.e., $\overline{\mathrm{g}}_o = \overline{\mathrm{g}}_l \sqcup_{\overline{\mathcal{G}}} \overline{\mathrm{g}}_r$, based on a series of syntactic rewriting rules. Each rewriting rule outputs a memory region into $\overline{\mathrm{g}}_o$, which is either identical to memory regions of $\overline{\mathrm{g}}_l$ and $\overline{\mathrm{g}}_r$ up to node renaming or over-approximates memory regions of $\overline{\mathrm{g}}_l$ and $\overline{\mathrm{g}}_r$. Then, the numerical joining operator $\sqcup_{\overline{\mathcal{N}}}$ is applied on $\overline{\mathrm{n}}_l$ and $\overline{\mathrm{n}}_r$ to compute a common over-approximation, i.e., $\overline{\mathrm{n}}_o = \overline{\mathrm{n}}_l \sqcup_{\overline{\mathcal{N}}} \overline{\mathrm{n}}_r$.

The widening operator $\triangledown_{\overline{\mathcal{M}_\omega}}$ is very similar to the joining operator, except that it relies on the numerical widening operator $\triangledown_{\overline{\mathcal{N}}}$ to enforce termination.

Disjunctive inclusion checking $\sqsubseteq_{\overline{\mathcal{D}}}$, joining $\sqcup_{\overline{\mathcal{D}}}$ and widening $\triangledown_{\overline{\mathcal{D}}}$. Lifting the inclusion checking of abstract states $\sqsubseteq_{\overline{\mathcal{M}_\omega}}$ to disjunctive inclusion checking $\sqsubseteq_{\overline{\mathcal{D}}}$ is obvious:

$$\overline{\mathrm{m}}_{l_1} \vee \ldots \vee \overline{\mathrm{m}}_{l_n} \sqsubseteq_{\overline{\mathcal{D}}} \overline{\mathrm{m}}_{r_1} \vee \ldots \vee \overline{\mathrm{m}}_{r_m} \iff \forall 1 \leq i \leq n, \exists 1 \leq j \leq m, \overline{\mathrm{m}}_{l_i} \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{\mathrm{m}}_{r_j}$$

However, an efficient disjunctive inclusion checking algorithm relies not only on efficient inclusion checking of abstract states $\sqsubseteq_{\overline{\mathcal{M}_\omega}}$ but also on an efficient algorithm (presented in Chapter 8) to choose the right abstraction $\overline{\mathrm{m}}_{r_j}$ for each $\overline{\mathrm{m}}_{l_i}$ such that $\overline{\mathrm{m}}_{l_i} \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{\mathrm{m}}_{r_j}$.

Given two disjunctive abstract states $\overline{\mathrm{m}}_{l_1} \vee \ldots \vee \overline{\mathrm{m}}_{l_n}$ and $\overline{\mathrm{m}}_{r_1} \vee \ldots \vee \overline{\mathrm{m}}_{r_m}$, a simple disjunctive joining $\sqcup_{\overline{\mathcal{D}}}$ can directly return the disjunction of them $\overline{\mathrm{m}}_{l_1} \vee \ldots \vee \overline{\mathrm{m}}_{l_n} \vee \overline{\mathrm{m}}_{r_1} \vee \ldots \vee \overline{\mathrm{m}}_{r_m}$. However, this joining operator can cause the disjunct number to grow exponentially. In order to limit the growth of the disjunct number, a disjunctive joining operator $\sqcup_{\overline{\mathcal{D}}}$ should partition the disjuncts of both arguments into different groups and compute an over-approximation for each group by repeatedly joining abstract states based on the joining operator $\sqcup_{\overline{\mathcal{M}_\omega}}$. We refer the reader to Chapter 10 for more details of the disjunctive joining operator.

A basic idea for widening two disjunctive abstract states $\overline{\mathrm{m}}_{l_1} \vee \ldots \vee \overline{\mathrm{m}}_{l_n} \triangledown_{\overline{\mathcal{D}}} \overline{\mathrm{m}}_{r_1} \vee \ldots \vee \overline{\mathrm{m}}_{r_m}$ is to partition the disjuncts of the right side into different groups according to the disjuncts of the left side, such that any disjunct $\overline{\mathrm{m}}_{l_i}$ of the left side can be widened with an over-approximation of a subset of disjuncts of the right side. However, in practice, designing a precise disjunctive widening operator $\triangledown_{\overline{\mathcal{D}}}$ is usually hard, especially when the underlying abstract domain has infinite abstract states since the widening operator has to guarantee that the disjunct number of any iteration sequence will finally stabilize. Chapter 10 presents a such widening operator.

$$\overline{\mathbf{guard}}_{\overline{\mathcal{D}}}\overline{[\![\mathrm{c}_{\overline{\mathcal{V}}}]\!]}(\overline{\mathrm{m}}) \;=\; \bigvee_{1\le i\le n}(\overline{\mathrm{g}}_i, \overline{\mathbf{guard}}_{\overline{\mathcal{N}}}(\overline{\mathrm{n}}_i, \overline{v}_i \neq 0))$$

$$\quad \textit{where } \overline{\mathbf{eval}_r[\![\mathrm{c}_{\overline{\mathcal{V}}}]\!]}(\overline{\mathrm{m}}) = \bigvee_{1\le i\le n}(\overline{v}_i, (\overline{\mathrm{g}}_i, \overline{\mathrm{n}}_i))$$

Assignment:

$$\overline{[\![\mathtt{l} = \mathtt{r}]\!]}(\overline{\mathrm{m}}) \;=\; \bigvee_{1\le i\le n}\bigvee_{1\le j\le k_i}\overline{\mathbf{write}}(\overline{v}_i, \mathtt{f}_i, \overline{v}_{i_j}, \overline{\mathrm{m}}_{i_j})$$

$$\quad \textit{where } \overline{\mathbf{eval}_l[\![\mathtt{l}]\!]}(\overline{\mathrm{m}}) = \bigvee_{1\le i\le n}((\overline{v}_i, \mathtt{f}_i), \overline{\mathrm{m}}_i)$$

$$\quad \textit{and } \forall 1 \le i \le n, \overline{\mathbf{eval}_r[\![\mathtt{r}]\!]}(\overline{\mathrm{m}}_i) = \bigvee_{1\le j\le k_i}(\overline{v}_{i_j}, \overline{\mathrm{m}}_{i_j})$$

Conditional branch:

$$\overline{[\![\mathbf{if}(\mathtt{r})\{\mathtt{p}_1\}\mathbf{else}\{\mathtt{p}_2\}]\!]}(\overline{\mathrm{m}}) \;=\; \overline{[\![\mathtt{p}_1]\!]} \circ \overline{\mathbf{guard}}_{\overline{\mathcal{D}}}\overline{[\![\mathtt{r} \neq 0]\!]}(\overline{\mathrm{m}}) \sqcup_{\overline{\mathcal{D}}} \overline{[\![\mathtt{p}_2]\!]} \circ \overline{\mathbf{guard}}_{\overline{\mathcal{D}}}\overline{[\![\mathtt{r} == 0]\!]}(\overline{\mathrm{m}})$$

Loop:

$$\overline{[\![\mathbf{while}(\mathtt{r})\{\mathtt{p}\}]\!]}(\overline{\mathrm{d}}) \;=\; \overline{\mathbf{guard}}_{\overline{\mathcal{D}}}\overline{[\![\mathtt{r} == 0]\!]}(\overline{\mathbf{lfp}}_{\overline{\sqsubseteq}_{\overline{\mathcal{D}}}}\lambda\overline{\mathrm{d}}_1.\, \overline{\mathrm{d}} \sqcup_{\overline{\mathcal{D}}} \overline{[\![\mathtt{p}]\!]} \circ \overline{\mathbf{guard}}_{\overline{\mathcal{D}}}\overline{[\![\mathtt{r} \neq 0]\!]}(\overline{\mathrm{d}}_1))$$

Figure 2.7: Abstract denotational semantics of program statements

2.4.5 Abstract Denotational Semantics

In this section, we formalize the abstract denotational semantics of programs that over-approximates the concrete denotational semantics defined in Figure 2.4 as:

$$\overline{[\![\mathtt{p}]\!]} \in \overline{\mathcal{D}} \to \overline{\mathcal{D}}$$

The abstract denotational semantics takes a disjunctive abstract state that over-approximates the pre-condition of the program `p` and outputs a disjunctive abstract state that over-approximates the concrete states after the execution of the program, and can be defined by induction over the syntax of programs in a very standard way. The soundness condition of the abstract denotational semantics is:

$$\overline{[\![\mathtt{p}]\!]}(\overline{\mathrm{d}}_1) = \overline{\mathrm{d}}_2 \implies \{[\![\mathtt{p}]\!](\varepsilon, \sigma) \mid (\varepsilon, \sigma) \in \gamma_{\overline{\mathcal{D}}}(\overline{\mathrm{d}}_1)\} \subseteq \gamma_{\overline{\mathcal{D}}}(\overline{\mathrm{d}}_2)$$

For simplicity, we only formalize the abstract semantics of assignment, condition branch and loop statements in Figure 2.7. The abstract semantics of the other program statements can be defined similarly.

Abstract semantics of assignment statements. Intuitively, given an abstract state $\overline{\mathrm{m}} \in \overline{\mathcal{M}_\omega}$, an abstract assignment operation $\overline{[\![\mathtt{l} = \mathtt{r}]\!]}$ should evaluate the left value expression into an abstract address $\overline{v}_1 + \mathtt{f}$, evaluate the right value expression into an abstract variable $\overline{v}_2$, and then write the abstract variable $\overline{v}_2$ into the abstract address $\overline{v}_1 + \mathtt{f}$. However, as the abstract evaluation of left and right value expressions may trigger unfolding that produces disjunctive abstract states, therefore as shown in Figure 2.7, the abstract assignment operation on a single disjunct returns a disjunctive abstract state. Though disjuncts can be collapsed together by the joining operator $\sqcup_{\overline{\mathcal{M}_\omega}}$, precision is often lost. The opposite direction consists in keeping all disjuncts as shown in 2.7, which would lead to a very precise but very costly analysis. In practice, in order to design scalable analysis, an operator that partially collapses disjuncts to maintain a low number of disjuncts while preserving the precision is necessary. The soundness can be proved by composing the soundness of abstract operations $\overline{\mathbf{eval}_l[\![\mathtt{l}]\!]}$, $\overline{\mathbf{eval}_r[\![\mathtt{r}]\!]}$ and $\overline{\mathbf{write}}$.

Abstract semantics of conditional branch statements. Figure 2.7 shows the definition of the abstract semantics of conditional branch statements on a single disjunct, which involves a guard function of the following form:

$$\overline{\mathbf{guard}}_{\overline{\mathcal{D}}} \in \mathcal{C}_{\overline{\mathcal{V}}} \to \overline{\mathcal{D}} \to \overline{\mathcal{D}}$$

It inputs a numerical constraint $\mathrm{c}_{\overline{\mathcal{V}}} \in \mathcal{C}_{\overline{\mathcal{V}}}$ expressing the condition to be guarded and a disjunctive abstract state $\overline{\mathrm{d}}_1 \in \overline{\mathcal{D}}$, and returns a disjunctive abstract state $\overline{\mathrm{d}}_2 \in \overline{\mathcal{D}}$ that over-approximates the concrete states of $\gamma_{\overline{\mathcal{D}}}(\overline{\mathrm{d}}_1)$, which satisfies $\mathrm{c}_{\overline{\mathcal{V}}}$. Figure 2.7 shows the formal definition of $\overline{\mathbf{guard}}_{\overline{\mathcal{D}}}$, which follows the soundness condition below.

Condition 2.10 (Soundness of the abstract guard function). *The abstract operation* $\overline{\mathbf{guard}}_{\overline{\mathcal{D}}}$ *is sound if and only if the following property holds:*

$$\overline{\mathbf{guard}}_{\overline{\mathcal{D}}}(\mathrm{c}_{\overline{\mathcal{V}}}, \overline{\mathrm{d}}_1) = \overline{\mathrm{d}}_2 \implies \{(\varepsilon, \sigma) \mid (\varepsilon, \sigma) \in \gamma_{\overline{\mathcal{D}}}(\overline{\mathrm{d}}_1) \wedge \mathbf{eval}_r[\![\mathrm{c}_{\overline{\mathcal{V}}}]\!](\varepsilon, \sigma) \notin \{0, \omega\}\} \subseteq \gamma_{\overline{\mathcal{D}}}(\overline{\mathrm{d}}_2)$$

Abstract semantics of loops. As the concrete denotational semantics of a loop $\mathbf{while}(\mathtt{r})\{\mathtt{p}\}$ shown in Figure 2.4 computes a least fix-point as a post-condition from a pre-condition, the abstract denotational semantics of loops defined in Figure 2.7 thus needs to compute an abstract post fix-point for the loop, denoted as $\overline{\mathbf{lfp}}_{\sqsubseteq_{\overline{\mathcal{D}}}} \lambda\overline{\mathrm{d}}_1.\ \overline{\mathrm{d}} \sqcup_{\overline{\mathcal{D}}} \overline{[\![\mathtt{p}]\!]} \circ \overline{\mathbf{guard}_{\overline{\mathcal{D}}}[\![\mathtt{r} \neq 0]\!]}(\overline{\mathrm{d}}_1)$, to over-approximate the concrete least fix-point. In practice, the abstract post fix-point can be computed by a series of iterations with the widening operator $\triangledown_{\overline{\mathcal{D}}}$ defined as: $\overline{\mathrm{d}}'_0 = \overline{\mathrm{d}}$ and $\overline{\mathrm{d}}'_{n+1} = \overline{\mathrm{d}}'_n \triangledown_{\overline{\mathcal{D}}} \overline{[\![\mathtt{p}]\!]} \circ \overline{\mathbf{guard}_{\overline{\mathcal{D}}}[\![\mathtt{r} \neq 0]\!]}(\overline{\mathrm{d}}'_n)$. The termination and soundness properties of the widening operator $\triangledown_{\overline{\mathcal{D}}}$ ensure an abstract post fix-point can always be reached.

Soundness of the analysis. Finally, let us establish the main theorem regarding the soundness of the analysis.

Theorem 2.1 (Soundness of the abstract denotational semantics.). *The abstract denotational semantics is sound with respect to the soundness condition:*

$$\overline{[\![p]\!]}(\overline{\mathrm{d}}_1) = \overline{\mathrm{d}}_2 \implies \{[\![p]\!](\varepsilon, \sigma) \mid (\varepsilon, \sigma) \in \gamma_{\overline{\mathcal{D}}}(\overline{\mathrm{d}}_1)\} \subseteq \gamma_{\overline{\mathcal{D}}}(\overline{\mathrm{d}}_2)$$

Proof. The proof can be done by induction over the syntax of programs, by composing:

- the soundness of the abstract store operations: $\overline{\mathbf{read}}$, $\overline{\mathbf{write}}$, $\overline{\mathbf{create}}$ and $\overline{\mathbf{delete}}$;
- the soundness of abstract evaluation operations: $\overline{\mathbf{eval}_l[\![\mathtt{l}]\!]}$ and $\overline{\mathbf{eval}_r[\![\mathtt{r}]\!]}$, and the abstract guard functions $\overline{\mathbf{guard}}_{\overline{\mathcal{N}}}$ and $\overline{\mathbf{guard}}_{\overline{\mathcal{G}}}$;
- the soundness of abstract lattice operators: $\sqsubseteq_{\overline{\mathcal{M}_\omega}}$, $\sqcup_{\overline{\mathcal{M}_\omega}}$, $\triangledown_{\overline{\mathcal{M}_\omega}}$.

■

2.5 Domain Signatures

Thus far, we have introduced an abstract domain for abstracting memory states that takes a numerical abstract domain as a parameter, and lifted the memory abstract domain to a disjunctive abstract domain. To sum up, in this section we give an exhaustive list of signatures of these abstract domains.

Signature for numerical abstract domains.

numerical abstract elements	$\overline{\mathrm{n}} \in \overline{\mathcal{N}}$
concretization	$\gamma_{\overline{\mathcal{N}}} : \overline{\mathcal{N}} \to \mathcal{P}(\overline{\mathcal{V}} \to \mathcal{V})$
guard	$\overline{\mathbf{guard}}_{\overline{\mathcal{N}}} : \overline{\mathcal{N}} \times \mathcal{C}_{\overline{\mathcal{V}}} \to \overline{\mathcal{N}}$
assign	$\overline{\mathbf{assign}}_{\overline{\mathcal{N}}} : \overline{\mathcal{N}} \times \overline{\mathcal{V}} \times \mathcal{R}_{\overline{\mathcal{V}}} \to \overline{\mathcal{N}}$
remove	$\overline{\mathbf{remove}}_{\overline{\mathcal{N}}} : \overline{\mathcal{N}} \times \overline{\mathcal{V}} \to \overline{\mathcal{N}}$
prove	$\overline{\mathbf{prove}}_{\overline{\mathcal{N}}} : \overline{\mathcal{N}} \times \mathcal{C}_{\overline{\mathcal{V}}} \to \{\mathbf{true}, \mathbf{false}\}$
rename	$\overline{\mathbf{rename}}_{\overline{\mathcal{N}}} : \overline{\mathcal{N}} \times (\overline{\mathcal{V}} \to \mathcal{P}(\overline{\mathcal{V}})) \to \overline{\mathcal{N}}$
abstract join	$\sqcup_{\overline{\mathcal{N}}} : \overline{\mathcal{N}} \times \overline{\mathcal{N}} \to \overline{\mathcal{N}}$
abstract widen	$\triangledown_{\overline{\mathcal{N}}} : \overline{\mathcal{N}} \times \overline{\mathcal{N}} \to \overline{\mathcal{N}}$
inclusion check	$\sqsubseteq_{\overline{\mathcal{N}}}: \overline{\mathcal{N}} \times \overline{\mathcal{N}} \to \{\mathbf{true}, \mathbf{false}\}$

Signature for memory abstract domains.

abstract states	$\overline{\mathrm{m}} \in \overline{\mathcal{M}_\omega}$
concretization	$\gamma_{\overline{\mathcal{M}_\omega}} : \overline{\mathcal{M}_\omega} \to \mathcal{P}(\mathcal{E} \times \mathcal{H})$
guard	$\overline{\mathbf{guard}}_{\overline{\mathcal{M}_\omega}} : \overline{\mathcal{M}_\omega} \times \mathcal{C}_{\overline{\mathcal{V}}} \to \bigvee \overline{\mathcal{M}_\omega}$
store read	$\overline{\mathbf{read}} : \overline{\mathcal{V}} \times \mathcal{F} \times \overline{\mathcal{M}_\omega} \to \overline{\mathcal{V}} \uplus \{\top, \bot\}$
store write	$\overline{\mathbf{write}} : \overline{\mathcal{V}} \times \mathcal{F} \times \overline{\mathcal{V}} \times \overline{\mathcal{M}_\omega} \to \overline{\mathcal{M}_\omega}$
store create	$\overline{\mathbf{create}} : \mathbb{N} \times \overline{\mathcal{M}_\omega} \to (\overline{\mathcal{V}} \times \overline{\mathcal{M}_\omega})^2 \uplus \{\bot, \top\}$
store delete	$\overline{\mathbf{delete}} : \overline{\mathcal{V}} \times \mathbb{N} \times \overline{\mathcal{M}_\omega} \to \overline{\mathcal{M}_\omega}$
left value evaluation	$\overline{\mathbf{eval}_l[\![\mathtt{l}]\!]} : \overline{\mathcal{M}_\omega} \to \bigvee((\overline{\mathcal{V}} \times \mathcal{F}) \times \overline{\mathcal{M}_\omega}) \uplus \{\bot, \top\}$
right value evaluation	$\overline{\mathbf{eval}_r[\![\mathtt{r}]\!]} : \overline{\mathcal{M}_\omega} \to \bigvee(\overline{\mathcal{V}} \times \overline{\mathcal{M}_\omega}) \uplus \{\bot, \top\}$
abstract join	$\sqcup_{\overline{\mathcal{M}_\omega}} : \overline{\mathcal{M}_\omega} \times \overline{\mathcal{M}_\omega} \to \overline{\mathcal{M}_\omega}$
abstract widen	$\triangledown_{\overline{\mathcal{M}_\omega}} : \overline{\mathcal{M}_\omega} \times \overline{\mathcal{M}_\omega} \to \overline{\mathcal{M}_\omega}$
inclusion check	$\sqsubseteq_{\overline{\mathcal{M}_\omega}}: \overline{\mathcal{M}_\omega} \times \overline{\mathcal{M}_\omega} \to \{\mathbf{true}, \mathbf{false}\}$

Signature for disjunctive memory abstract domains.

abstract states	$\overline{\mathrm{d}} \in \overline{\mathcal{D}}$
concretization	$\gamma_{\overline{\mathcal{D}}} : \overline{\mathcal{D}} \rightarrow \mathcal{P}(\mathcal{E} \times \mathcal{H})$
guard	$\overline{\mathbf{guard}}_{\overline{\mathcal{D}}} : \overline{\mathcal{D}} \times \mathcal{C}_{\overline{\mathcal{V}}} \rightarrow \overline{\mathcal{D}}$
abstract join	$\sqcup_{\overline{\mathcal{D}}} : \overline{\mathcal{D}} \times \overline{\mathcal{D}} \rightarrow \overline{\mathcal{D}}$
abstract widen	$\nabla_{\overline{\mathcal{D}}} : \overline{\mathcal{D}} \times \overline{\mathcal{D}} \rightarrow \overline{\mathcal{D}}$
inclusion check	$\sqsubseteq_{\overline{\mathcal{D}}}: \overline{\mathcal{D}} \times \overline{\mathcal{D}} \rightarrow \{\mathbf{true}, \mathbf{false}\}$

Part II

Shape Analysis for Unstructured Sharing

Chapter 3

Overview

In this part, we propose a shape analysis that tracks set properties to infer precise invariants about data structures with unstructured sharing. Before the formalization, we outline the analysis of a graph traversal algorithm, where graphs are represented as adjacency lists.

3.1 Abstraction

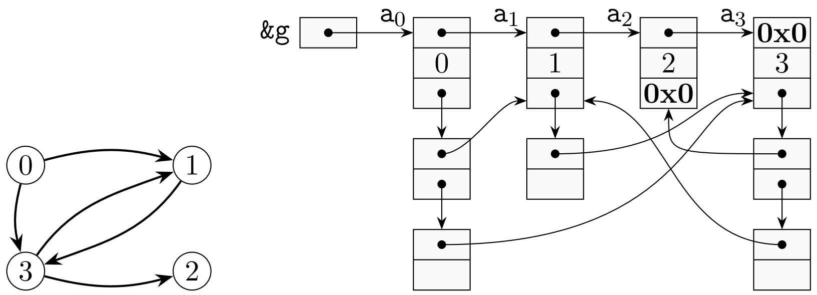
An adjacency list is a graph data structure with unstructured sharing as the number of predecessors of a node is unbounded and the predecessors could be anywhere in the structure. Figure 3.1 shows a type definition of adjacency lists: a graph is a list of nodes, each node has a list of edges which comprise pointers to successor nodes. Figure 3.2(b) shows an adjacency list representation of the graph shown in Figure 3.2(a).

Separation logic based shape analysis relies on the separating conjunction $*$ to express disjoint properties of memory regions and inductive definitions to summarize recursive data structures. However, summarizing adjacency lists in this framework turns out to be very challenging. Simply exploiting the list-of-lists inductive skeleton of adjacency lists based on the separation conjunction cannot capture the cross edge pointers precisely, as the separation conjunction only allow nodes to be described once. In order to capture the unstructured sharing of adjacency lists precisely, Section 1.5.1 outlined an approach to summarize adjacency lists using a combination of an inductive skeleton and relations over set-valued variables. However, using such summaries poses significant algorithmic challenges in analyzing graph manipulation programs. To be more concrete, in this chapter we consider analyzing a representative of graph traversal algorithms that manipulates graph edges. Intuitively, traversing graph edges amounts to traversing the cross pointers of Figure 3.2(b). Such steps make the shape analysis of graph manipulation programs tricky since they do not follow the inductive skeleton of the adjacency list—instead, they "jump" to some other node in the structure.

In the rest of this section, we first present the inductive definition of adjacency lists and then discuss abstractions of memory states comprising adjacency lists. Formalizations of the abstractions are presented in Chapter 4.

```
typedef struct node{
  struct node * next;
  int id;
  struct edge * edges;
} node;
typedef struct edge{
  struct node * dest;
  struct edge * next;
} edge;
```

Figure 3.1: Type definition of adjacency lists



(a) A simple graph

(b) Adjacency list representation of (a)

Figure 3.2: Example of an adjacency list

Graph inductive predicate. The first step towards an analysis to verify a graph algorithm is to set up inductive definitions to summarize the adjacency list structure. The set of outgoing edges of a node consists of a list of records, and thus the predicate to summarize such a region can be based on a classical list inductive definition:

$$\begin{array}{ll} \alpha \cdot \mathbf{list} ::= & \\ & (\mathbf{emp}, \alpha = 0) \\ \vee & (\alpha \cdot \mathtt{dest} \mapsto \beta * \alpha \cdot \mathtt{next} \mapsto \gamma * \gamma \cdot \mathbf{list}, \alpha \neq 0) \end{array}$$

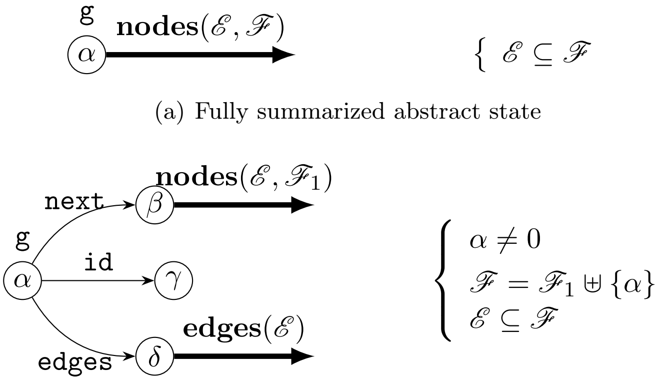
However, this definition does not express the fact that all instances of field `dest` contain a pointer to a node of the graph as the value β of that field is unconstrained. To resolve this issue, we simply need to add the constraint $\beta \in \mathscr{E}$, where $\mathscr{E}$ should denote the set of all node addresses in the graph. The abstract domain should also keep track of those predicates through folding and unfolding steps. Therefore, we obtain the inductive definition **edges**, as shown in Figure 3.3. The inductive definition **edges** takes the additional parameter $\mathscr{E}$, and the value predicate of the non-empty case has been strengthened with the set predicate $\beta \in \mathscr{E}$.

Moreover, the inductive definition of a graph needs to capture two set properties: (1) the destination of all edges are in set $\mathscr{E}$ (as described by inductive definition **edges**) and (2) the set of nodes in the adjacency list should contain the set $\mathscr{E}$. Thus, the inductive definition **nodes** presented in Figure 3.3 takes two set parameters: (1) $\mathscr{E}$ is constant over the whole induction and

$$
\begin{array}{ll}
\multicolumn{2}{l}{\alpha \cdot \mathbf{edges}(\mathscr{E}) ::=} \\
 & (\mathbf{emp}, \alpha = 0) \\
\vee & (\alpha \cdot \mathtt{dest} \mapsto \beta * \alpha \cdot \mathtt{next} \mapsto \gamma \\
 & * \gamma \cdot \mathbf{edges}(\mathscr{E}), \alpha \neq 0 \wedge \beta \in \mathscr{E} \wedge \beta \neq 0)
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\alpha \cdot \mathbf{nodes}(\mathscr{E}, \mathscr{F}) ::=} \\
 & (\mathbf{emp}, \alpha = 0 \wedge \mathscr{F} = \emptyset) \\
\vee & (\alpha \cdot \mathtt{next} \mapsto \beta * \alpha \cdot \mathtt{id} \mapsto \gamma * \alpha \cdot \mathtt{edges} \mapsto \delta \\
 & * \beta \cdot \mathbf{nodes}(\mathscr{E}, \mathscr{F}') * \delta \cdot \mathbf{edges}(\mathscr{E}), \\
 & \alpha \neq 0 \wedge \mathscr{F} = \mathscr{F}' \uplus \{\alpha\})
\end{array}
$$

Figure 3.3: Inductive definitions of adjacency lists


g
nodes($\mathscr{E}$, $\mathscr{F}$)
α
$\{ \mathscr{E} \subseteq \mathscr{F}$
nodes($\mathscr{E}$, $\mathscr{F}_1$)
next
β
g
id
α
γ
edges($\mathscr{E}$)
edges
δ
$\alpha \neq 0$
$\mathscr{F} = \mathscr{F}_1 \uplus \{\alpha\}$
$\mathscr{E} \subseteq \mathscr{F}$

(a) Fully summarized abstract state

(b) Partially summarized abstract state

Figure 3.4: Abstract states of graphs

(2) $\mathscr{F}$ stands for the set of nodes of a graph fragment described by an instance of **nodes**. We note that the set predicates $\mathscr{F} = \emptyset$ (base case) and $\mathscr{F} = \mathscr{F}' \uplus \{\alpha\}$ (inductive case) guarantee that $\mathscr{F}$ is exactly the set of nodes described by predicate $\alpha \cdot \mathbf{nodes}(\mathscr{E}, \mathscr{F})$.

Abstraction of memory states. Using these definitions, the complete adjacency list pointed to by pointer g shown in Figure 3.2(b) can be fully summarized by the abstract state shown in Figure 3.4(a), which contains an inductive predicate $\alpha \cdot \mathbf{nodes}(\mathscr{E}, \mathscr{F})$ (set variable $\mathscr{E}$ denotes the set of destination nodes of edges and set variable $\mathscr{F}$ denotes all the nodes of the concrete graph) and a set constraint $\mathscr{E} \subseteq \mathscr{F}$ specifying that all the destination nodes of edges are graph nodes. Similarly, Figure 3.4(b) displays a partially summarized abstraction, where points-to predicates starting from α abstract the first graph node, inductive predicate $\delta \cdot \mathbf{edges}(\mathscr{E})$ abstracts the edge list of the graph node at α, and inductive predicate $\beta \cdot \mathbf{nodes}(\mathscr{E}, \mathscr{F}_1)$ abstracts the adjacency list of the other nodes in the graph. Additional set predicates ($\mathscr{E} = \mathscr{F}_1 \uplus \{\alpha\} \wedge \mathscr{E} \subseteq \mathscr{F}$) express that all the destination nodes of edges are graph nodes. We note that set predicates and numerical predicates are represented in a *value-set abstract domain*, which is obtained as a reduced product [CC79] of a numerical abstract domain and a set abstract domain (presented in

```
1  void traversal(node* h){
2    node* c = h;
3    while(c != NULL){
4      edge* s = c -> edges;
5      if (s == NULL) break;
6      while(s->next != 0 && random()){
7        s = s->next;
8      }
9      c = s->dest;
10     printf("visiting: %d", c->id);
11   }
12 }
```

Figure 3.5: A representative path traversal function through a graph

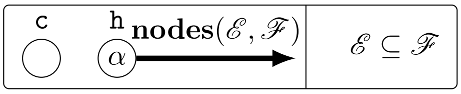
Chapter 5). For simplicity, we show here the predicates directly instead of value-set abstractions.

3.2 Analysis Algorithm

We now discuss automatic shape analysis algorithms (formalized in Chapter 6) to verify the memory safety property (the absence of null or dangling pointer dereferences) and the preservation of structural invariants of the graph random traversal program shown in Figure 3.5.

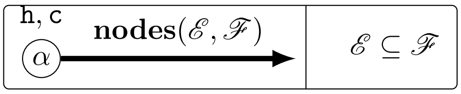
In particular, the analysis should start from a pre-condition specifying that variable `h` points-to a correct graph with the set of nodes $\mathscr{F}$:

$\overline{\mathrm{m}}_0$:


c
h
nodes($\mathscr{E}$, $\mathscr{F}$)
α
$\mathscr{E} \subseteq \mathscr{F}$

Then, at line 2, the cursor `c` is initialized to variable `h`:

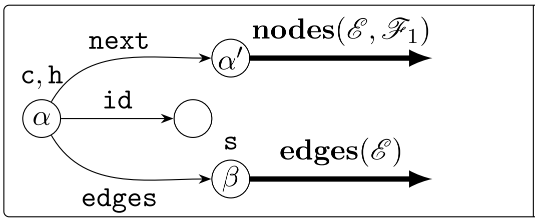
$\overline{\mathrm{m}}_1$:


h, c
nodes($\mathscr{E}$, $\mathscr{F}$)
α
$\mathscr{E} \subseteq \mathscr{F}$

Then, the loop body (lines 4 to 10) randomly selects a successor of the node pointed to by `c` (lines 4 to 8) and moves to that node at line 9. The analysis of the loop body needs to unfold summaries to perform mutation over summarized regions, and to utilize a widening operator for the convergence of the abstract iterates over both nested loops. However, to establish that no dangling pointer is dereferenced, the analysis should precisely track the fact that `c` either points to a valid node of the graph, or is a null pointer (which causes the program to exit) at all times. To show the challenges of the analysis, let us discuss the first abstract loop iteration in detail; the other iterations are similar.

Unfolding. At line 4, variable `s` is initialized to a pointer to the edge list of the node pointed-to by `c`. As variable `c` points-to an inductive predicate $\alpha \cdot \mathbf{nodes}(\mathscr{E}, \mathscr{F})$ as shown in the abstract state $\overline{\mathrm{m}}_1$, performing the abstract assignment requires unfolding the inductive predicate and other abstract operations, which leads to the abstract state below:

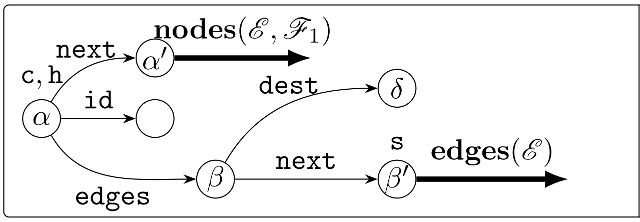
$\overline{\mathrm{m}}_2$:



$$\begin{aligned} & \alpha \neq 0 \wedge \mathscr{F} = \{\alpha\} \uplus \mathscr{F}_1 \\ \wedge \quad & \mathscr{E} \subseteq \mathscr{F} \end{aligned}$$

Moreover, the assignment at line 7 advances variable s to the pointer of its next edge, which relies on unfolding the edge predicate $\beta \cdot \textbf{edges}(\mathscr{E})$ as shown in the abstract state $\overline{\mathrm{m}}_2$ and leads to the abstract state $\overline{\mathrm{m}}_3$ after assignment:

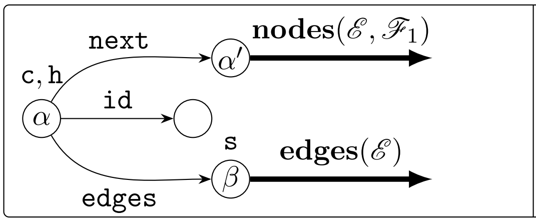
$\overline{\mathrm{m}}_3$:



$$\begin{aligned} & \alpha \neq 0 \wedge \beta \neq 0 \wedge \delta \neq 0 \wedge \beta' \neq 0 \\ \wedge \quad & \mathscr{F} = \{\alpha\} \uplus \mathscr{F}_1 \wedge \delta \in \mathscr{E} \wedge \mathscr{E} \subseteq \mathscr{F} \end{aligned}$$

Widening with set parameters. Then, the analysis needs to infer the loop invariant of the inner loop from line 6 to line 8, which relies on applying a widening operator ($\nabla_{\overline{\mathcal{M}_\omega}}$) on the two abstract states below at the end of the first loop iteration:

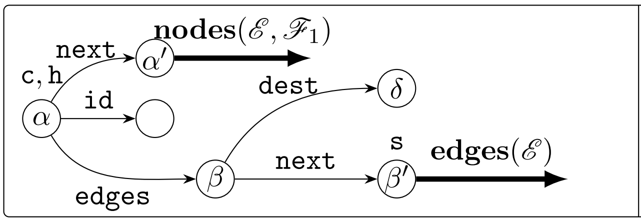
$\overline{\mathrm{m}}'_2$:



$$\begin{aligned} & \alpha \neq 0 \wedge \beta \neq 0 \\ \wedge \quad & \mathscr{E} \subseteq \mathscr{F} \wedge \mathscr{F} = \{\alpha\} \uplus \mathscr{F}_1 \end{aligned}$$

$\nabla_{\overline{\mathcal{M}_\omega}}$

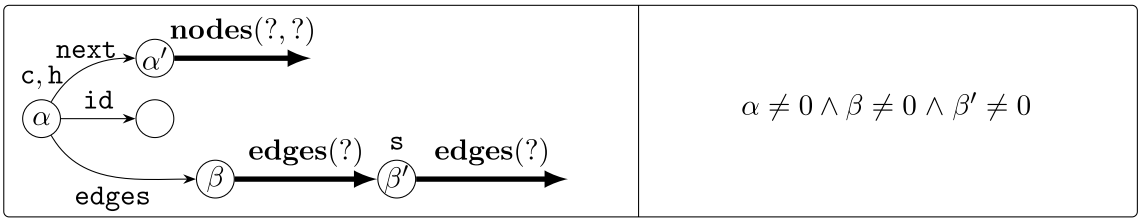
$\overline{\mathrm{m}}_3$:



$$\begin{aligned} & \alpha \neq 0 \wedge \beta \neq 0 \wedge \delta \neq 0 \wedge \beta' \neq 0 \\ \wedge \quad & \mathscr{F} = \{\alpha\} \uplus \mathscr{F}_1 \wedge \delta \in \mathscr{E} \wedge \mathscr{E} \subseteq \mathscr{F} \end{aligned}$$

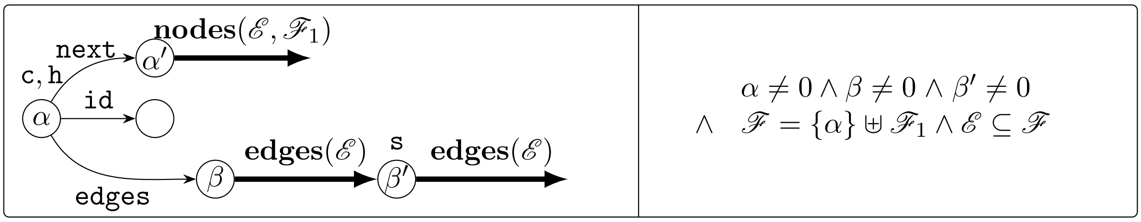
The two abstract shape graphs are very similar except that in $\overline{\mathrm{m}}'_2$, s is a pointer to the first element of the edge list of the node at α, and in $\overline{\mathrm{m}}_3$, s points-to the second element. Intuitively, the widening operation should compute a shape graph declaring that s refers to an element somewhere in the edge list of the node at α. This can be done by introducing an edge list segment from β to β' in the abstract graph below:

$\overline{\mathrm{m}}_4$:



Moreover, the widening operation should also synthesize set parameters for summary predicates, such that the structural invariants of adjacency lists can still be tracked. The abstract state below shows an abstract shape graph with precise set parameters:

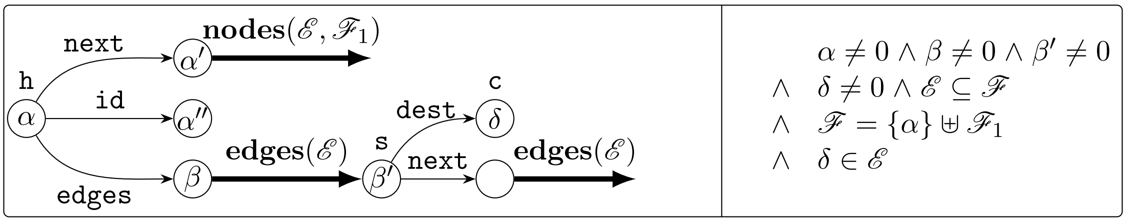
$\overline{\mathrm{m}}_4$:


nodes($\mathscr{E}, \mathscr{F}_1$)
next
c, h
α'
id
α
edges($\mathscr{E}$)
s
edges($\mathscr{E}$)
β
β'
edges
$\alpha \neq 0 \land \beta \neq 0 \land \beta' \neq 0$
$\land \quad \mathscr{F} = \{\alpha\} \uplus \mathscr{F}_1 \land \mathscr{E} \subseteq \mathscr{F}$

However, precise parameter synthesis requires reasoning about properties of the set parameters of the inductive definitions **nodes** and **edges**. For instance, the inference of set parameter $\mathscr{E}$ of the edge segment from β to β' is based on the fact that the inductive definition **edges** has a *constant* set parameter. Intuitively, a set parameter is said to be a *constant* of an inductive definition if it is propagated with no modification to all recursive calls of the inductive definition.

Non-local unfolding with set parameters. After the inner loop, variable `c` is set to the destination node of the edge pointed to by `s` at line 9, which leads to the abstract state:

$\overline{\mathrm{m}}_5$:

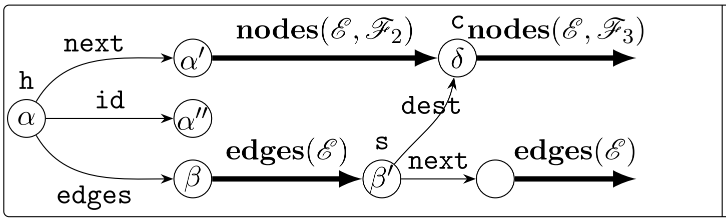

nodes($\mathscr{E}, \mathscr{F}_1$)
next
h
α'
c
id
dest
α
α''
δ
edges($\mathscr{E}$)
s
next
edges($\mathscr{E}$)
β
β'
edges
$\alpha \neq 0 \land \beta \neq 0 \land \beta' \neq 0$
$\land \quad \delta \neq 0 \land \mathscr{E} \subseteq \mathscr{F}$
$\land \quad \mathscr{F} = \{\alpha\} \uplus \mathscr{F}_1$
$\land \quad \delta \in \mathscr{E}$

In the abstract state $\overline{\mathrm{m}}_5$, `c` no longer appears to point to a node in the **nodes** inductive backbone. Yet, the dereferencing of `c -> id` at line 10 requires the materialization of an edge from that node, although no edge (points-to or summary) starts from node δ. However, the analysis infers that $\delta \in \mathscr{E} \land \mathscr{E} \subseteq \mathscr{F} \land \mathscr{F} = \{\alpha\} \uplus \mathscr{F}_1$, which indicates that either $\delta = \alpha$, or $\delta \in \mathscr{F}_1$. If $\delta = \alpha$, then the dereferencing of `c -> id` can be achieved by reading the value α'' of the points-to edge $\alpha.\mathtt{id} \mapsto \alpha''$. However, the case of $\delta \in \mathscr{F}_1$ is more complex.

We notice that the second parameter $\mathscr{F}$ of the **nodes** inductive definition is a *head parameter*: $\mathscr{F} = \emptyset$ in the empty rule and $\mathscr{F} = \{\alpha\} \uplus \mathscr{F}'$ in the second rule, where α is the address of the head of the structure and $\mathscr{F}'$ is the parameter of the tail. That is, the parameter collects the set of head nodes in all recursive inductive calls.

Therefore, in abstract state $\overline{\mathrm{m}}_5$, set parameter $\mathscr{F}_1$ denotes the set of addresses of graph nodes summarized in the inductive predicate $\alpha' \cdot \mathbf{nodes}(\mathscr{E}, \mathscr{F}_1)$. Thus, $\delta \in \mathscr{F}_1$ means that δ is the address of a graph node in the inductive predicate from α', and it can be *materialized* by splitting the inductive predicate into a node list segment from α' to δ and an inductive predicate from δ as shown below:

$\overline{\mathrm{m}}'_5$:



$$\begin{array}{ll} & \alpha \neq 0 \land \beta \neq 0 \land \beta' \neq 0 \\ \land & \delta \neq 0 \land \mathscr{E} \subseteq \mathscr{F} \\ \land & \mathscr{F} = \{\alpha\} \uplus \mathscr{F}_2 \uplus \mathscr{F}_3 \\ \land & \delta \in \mathscr{E} \end{array}$$

The side property $\mathscr{F} = \{\alpha\} \uplus \mathscr{F}_2 \uplus \mathscr{F}_3$ states that the set of graph nodes $\mathscr{F}_1$ summarized in the inductive edge $\alpha' \cdot \mathbf{nodes}(\mathscr{E}, \mathscr{F}_1)$ is split into the two partitions: $\mathscr{F}_2$ and $\mathscr{F}_3$, where $\mathscr{F}_2$ describes the set of graph nodes summarized in the segment and $\mathscr{F}_3$ describes the set of graph nodes summarized in the inductive predicate from δ.

This form of unfolding is much more complex than more conventional forms of inductive predicates unfolding. Indeed, the conventional unfolding techniques (except backward unfolding) look no further than the edges going out of the node where they need to do the unfolding while the unfolding presented here needs to utilize the set properties to localize δ. To achieve this, the analysis needs to track all set predicates through unfolding, updates and widening steps and also to infer properties of set parameters, i.e., *constant* and *head*.

Chapter 4

Abstractions for Data-Structures with Unstructured Sharing

This chapter formalizes the memory abstractions for reasoning about sharing. Specifically, Section 4.1 formalizes inductive definitions parameterized by set variables. Section 4.2 proposes memory abstractions parameterized by inductive definitions and an abstract domain for constraints over value and set variables. Section 4.3 discusses properties of set parameters: constant *and* head*, which are used to infer set parameters of summary predicates during program analysis.*

4.1 Inductive Definitions with Set Predicates

The analysis presented in this thesis is *parameterized* by a set of inductive definitions, which means that the abstract domain is generic, and can deal with a wide family of data-structures. In this section, we extend the relational inductive definitions of [CR08] with set predicates.

Inductive definitions. An *inductive definition* $\alpha \cdot \mathbf{ind}(\alpha_0, \ldots, \alpha_m, \mathscr{E}_0, \ldots, \mathscr{E}_n) ::= r_0 \vee r_1 \vee \ldots \vee r_k$ takes a main parameter α, a list of pointer parameters $\alpha_0, \ldots, \alpha_m$ and a list of set parameters $\mathscr{E}_0, \ldots, \mathscr{E}_n$ and defines a scheme to summarize heap regions that satisfy some inductive property, specified as a disjunction of *rules* as presented in Figure 4.1. Each rule comprises a *heap* part and a *pure* part. The heap part is a separating conjunction of memory cells (points-to predicates of the form $\alpha \cdot \mathtt{f} \mapsto \beta$) and recursive calls to inductive definitions. The pure part comprises not only numerical constraints, but also set constraints, over the symbolic variables exposed in the heap part and the set parameters, as shown in F_{Pure}. The intuitive meaning of a set constraint such as $\alpha \in \mathscr{E}$ is that the concretization will map α into a numerical value that belongs to the concretization of $\mathscr{E}$. As a simple example, the inductive definition below characterizes the singly linked list starting at address α, such that the set of addresses of list elements is exactly $\mathscr{E}$:

$$\begin{array}{rcl} \alpha \cdot \mathbf{l}_{\mathrm{s}}(\mathscr{E}) & ::= & (\mathbf{emp}, \alpha = 0 \wedge \mathscr{E} = \emptyset) \\ & \vee & (\alpha \cdot \mathtt{n} \mapsto \beta_0 * \alpha \cdot \mathtt{v} \mapsto \beta_1 * \beta_0 \cdot \mathbf{l}_{\mathrm{s}}(\mathscr{E}'), \alpha \neq 0 \wedge \mathscr{E} = \{\alpha\} \uplus \mathscr{E}') \end{array}$$

$$
\begin{array}{rcll}
r & ::= & (F_{\text{Heap}}, F_{\text{Pure}}) & \\
F_{\text{Heap}} & ::= & \mathbf{emp} & \\
 & \mid & \alpha \cdot \mathtt{f} \mapsto \beta & \\
 & \mid & \alpha \cdot \mathbf{ind}(\alpha_0, \ldots, \alpha_n, \mathscr{E}_0, \ldots, \mathscr{E}_n) & \\
 & \mid & F_{\text{Heap}} * F_{\text{Heap}} & \\
F_{\text{Pure}} & ::= & \alpha = c & (c \in \mathcal{V}) \\
 & \mid & \alpha \neq c & (c \in \mathcal{V}) \\
 & \mid & \alpha \in \mathscr{E} & \\
 & \mid & \mathscr{E} = \{\alpha\} \uplus \mathscr{F} & \\
 & \mid & \mathscr{E} = \{\alpha\} \cup \mathscr{F} & \\
 & \mid & \ldots &
\end{array}
$$

Figure 4.1: Inductive rules

As another example, the inductive definition below characterizes the set of `d` fields of the singly linked list starting at address α as a subset of $\mathscr{E}$:

$$
\begin{array}{rcl}
\alpha \cdot \mathbf{l}_{\text{c}}(\mathscr{E}) & ::= & (\mathbf{emp}, \alpha = 0) \\
 & \vee & (\alpha \cdot \mathtt{n} \mapsto \beta_0 * \alpha \cdot \mathtt{d} \mapsto \beta_1 * \beta_0 \cdot \mathbf{l}_{\text{c}}(\mathscr{E}), \alpha \neq 0 \wedge \beta_1 \in \mathscr{E})
\end{array}
$$

Inductive definitions **edges** and **nodes** (Figure 3.3) capture set constraints over the nodes and edges of graphs in a similar way: **nodes** collects *exactly* the set of all nodes of the graph, whereas **edges** asserts all edges should point to one of the nodes of the graph.

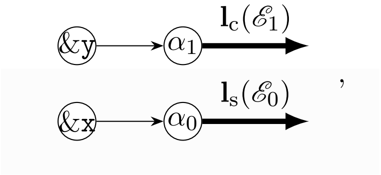
Since pointer parameters are not our main contribution, we will often omit them in this part of the thesis. However, the analysis that we describe supports them together with set parameters. Similarly, and for the sake of readability, we will often write inductive predicates with a single set parameter.

4.2 Composite Memory Abstraction with Set Predicates

We now formalize abstract memory states $\overline{\mathcal{M}_\omega}$.

Concrete values $\mathcal{V}$ are abstracted by abstract values $\overline{\mathcal{V}}$. An abstract value variable $\overline{v} \in \overline{\mathcal{V}}$ is either a symbolic variable $\alpha \in \overline{\mathcal{V}}_\alpha$ or the symbolic address &`x` of a program variable `x`. Set variables (e.g., $\mathscr{E}, \mathscr{F} \in \mathcal{T}$) abstract sets of concrete values. We let $\mu \in \mathcal{F}_\mu$ be a valuation function that maps each abstract value $\overline{v} \in \overline{\mathcal{V}}$ (resp., set variable $\mathscr{E} \in \mathcal{T}$) to a numerical value $\mu(\overline{v}) \in \mathcal{V}$ (resp., set of numerical values $\mu(\mathscr{E}) \in \mathcal{P}(\mathcal{V})$).

Moreover, abstract states are parameterized by a finite set of inductive definitions defined by the syntax shown in Section 4.1 and a value-set abstract domain $\overline{\mathcal{C}}$ that provides an abstraction for constraints over abstract values and set variables, such as value constraint $\alpha \neq 0 \wedge \alpha' \neq 0$, and set constraint $\mathscr{F} = \mathscr{F}_1 \uplus \mathscr{F}_2 \wedge \mathscr{E} \subseteq \mathscr{F}$. We will construct a set-value abstract domain in Chapter 5. Such a domain will be defined as a reduced product of an existing numeric abstract domain [Min06, CH78, CC77] and an abstract domain designed to describe constraints over set symbols.


&y
α_1
$\mathbf{l}_\mathrm{c}(\mathscr{E}_1)$
&x
α_0
$\mathbf{l}_\mathrm{s}(\mathscr{E}_0)$

, $\alpha_1 \neq 0 \wedge \mathscr{E}_1 \subseteq \mathscr{E}_0$

Figure 4.2: An abstract state

Abstract states. An *abstract state* is a pair $(\overline{\mathrm{g}}, \overline{\mathrm{c}})$ made up of an *abstract shape graph* $\overline{\mathrm{g}} \in \overline{\mathcal{G}}$, and a *value-set abstraction* $\overline{\mathrm{c}} \in \overline{\mathcal{C}}$. The syntax of abstract shapes is shown below:

$$\begin{array}{rcl} \overline{\mathrm{g}} & ::= & \mathbf{emp} \\ & | & \alpha \cdot \mathtt{f} \mapsto \beta \\ & | & \alpha \cdot \mathbf{ind}(\vec{\mathscr{E}}) \\ & | & \alpha \cdot \mathbf{ind}(\vec{\mathscr{E}}) \ast\!\!= \beta \cdot \mathbf{ind}(\vec{\mathscr{E}}) \\ & | & \overline{\mathrm{g}} \ast \overline{\mathrm{g}} \end{array}$$

An abstract shape graph $\overline{\mathrm{g}}$ is either empty, or a single points-to edge $\alpha \cdot \mathtt{f} \mapsto \beta$, or an inductive predicate $\alpha \cdot \mathbf{ind}(\vec{\mathscr{E}})$ (instantiating an inductive definition **ind**), or a segment $\alpha \cdot \mathbf{ind}(\vec{\mathscr{E}}) \ast\!\!= \beta \cdot \mathbf{ind}(\vec{\mathscr{E}})$ describing an incomplete inductive structure, or a separating product of such predicates. It should be noted that, this definition of abstract shape graphs is an extension of that given in Section 2.3. Specifically, the **emp** and points-to predicates are the same, yet the inductive predicates and segment predicates are extended with set parameters. Moreover, a value-set abstraction $\overline{\mathrm{c}} \in \overline{\mathcal{C}}$ is also an extension of a numerical abstraction $\overline{\mathrm{n}} \in \overline{\mathcal{N}}$, which allows us to reason about set properties. As an example, Figure 4.2 shows an abstract state parameterized by inductive definitions $\mathbf{l}_\mathrm{s}$ and $\mathbf{l}_\mathrm{c}$ defined in Section 4.1. It states that variable y points-to a list abstracted by $\alpha_1 \cdot \mathbf{l}_\mathrm{c}(\mathscr{E}_1)$, where the d field of each list element points to a list element of a list $\alpha_0 \cdot \mathbf{l}_\mathrm{s}(\mathscr{E}_0)$ pointed to by variable x.

Concretization. A value-set abstraction $\overline{\mathrm{c}} \in \overline{\mathcal{C}}$ is concretized into a set of valuations that map each abstract value variable and set variable respectively to a numerical value and a set of numerical values:

$$\gamma_{\overline{\mathcal{C}}} : \overline{\mathcal{C}} \to \mathcal{P}(\mathcal{F}_\mu)$$

The concretization of an abstract shape graph $\overline{\mathrm{g}} \in \overline{\mathcal{G}}$ is a set of pairs of concrete stores and valuations for abstract value variables and set variables, that is:

$$\gamma_{\overline{\mathcal{G}}} : \overline{\mathcal{G}} \to \mathcal{P}(\mathcal{H} \times \mathcal{F}_\mu)$$

The concretization of value-set abstractions and abstract shape graphs is very similar to that of numerical abstractions and abstract shape graphs introduced in Section 2.3 except that the valuations here concretize set variables. Thus, the concretization of an abstract memory state $\overline{\mathrm{m}} = (\overline{\mathrm{g}}, \overline{\mathrm{c}}) \in \overline{\mathcal{M}_\omega}$ is defined as a set of memory states, and for each memory state, a valuation of

$$\frac{}{([], \mu) \in \gamma_{\overline{\mathcal{G}}}(\mathbf{emp})} \qquad \frac{a = \mu(\alpha) + \mathtt{f} \qquad v = \mu(\beta)}{([a \mapsto v], \mu) \in \gamma_{\overline{\mathcal{G}}}(\alpha \cdot \mathtt{f} \mapsto \beta)}$$

$$\frac{\forall i,\ (\sigma_i, \mu) \in \gamma_{\overline{\mathcal{G}}}(\overline{\mathrm{g}}_i)}{(\sigma_0 \uplus \sigma_1, \mu) \in \gamma_{\overline{\mathcal{G}}}(\overline{\mathrm{g}}_0 * \overline{\mathrm{g}}_1)}$$

$$\frac{\overline{\mathrm{g}} \rightsquigarrow_{\mathrm{U}} (F_{\mathrm{Heap}}, F_{\mathrm{Pure}}) \qquad (\sigma, \mu) \in \gamma_{\overline{\mathcal{G}}}(F_{\mathrm{Heap}}) \qquad \mu \vdash F_{\mathrm{Pure}}}{(\sigma, \mu) \in \gamma_{\overline{\mathcal{G}}}(\overline{\mathrm{g}})}$$

Figure 4.3: Concretization rules

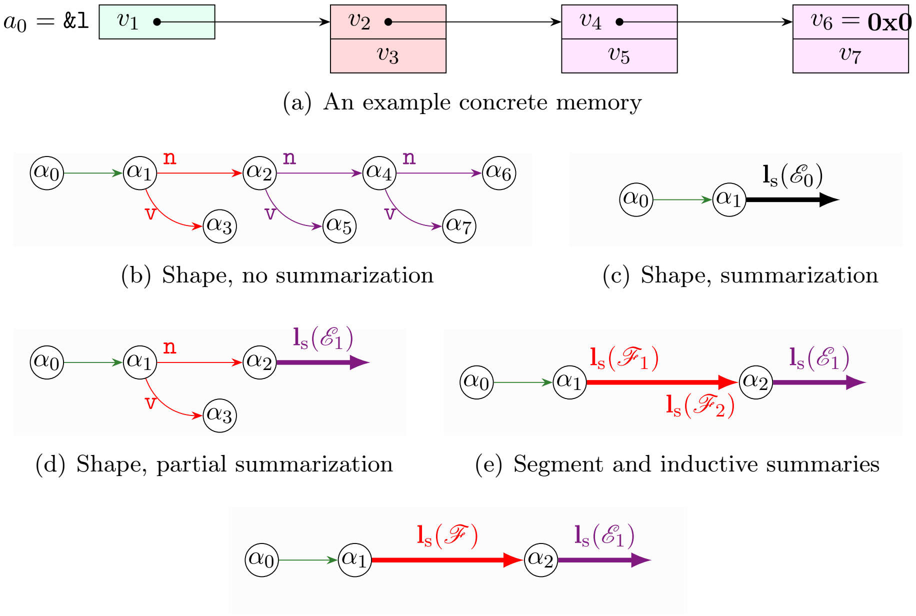
abstract value variables and set variables can be found that satisfies all constraints from $\overline{\mathrm{g}}$ and $\overline{\mathrm{c}}$:

$$\gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{g}}, \overline{\mathrm{c}}) = \{(\mu_{\lceil \mathcal{X}_\&}, \sigma) \in \mathcal{M} \mid \exists \mu \in \gamma_{\overline{\mathcal{C}}}(\overline{\mathrm{c}}),\ (\sigma, \mu) \in \gamma_{\overline{\mathcal{G}}}(\overline{\mathrm{g}})\}$$

Concretization of abstract shape graphs. Figure 4.3 shows the concretization rules for abstract shape graphs. The first three rules describe the usual concretization for empty shapes, single points-to edges and the separating conjunction. The last rule defines the concretization for inductive and segment predicates using the standard notion of *syntactic unfolding* [CR08]: the unfolding of an inductive or segment predicate selects a rule r in the corresponding inductive or segment definition, and replaces the predicate with the heap part of r and constrains value and set valuations with the pure part of r.

Example 4.1 (Abstract shape graphs and their concretization). Figure 4.4 shows a few abstractions of the concrete memory state σ shown in Figure 4.4(a), where `l` stores a pointer to a list of length 3, and where $v_0, v_1, \ldots, v_7$ denote numerical values / addresses. The shape of Figure 4.4(b) abstracts this state without any summarization (it contains no inductive predicate). Its concretization into m results in $\forall i,\ \mu(\alpha_i) = v_i$ (in particular, $\mu(\alpha_6) = v_6 = \mathbf{0x0}$), and can be fully expressed using the points-to and separating product rules of Figure 4.3. The shape of Figure 4.4(c) summarizes the list completely into a single inductive predicate $\alpha_1 \cdot \mathbf{l}_{\mathrm{s}}(\mathscr{E}_0)$. In this case, the concretization also needs to bind $\mathscr{E}_0$ to a set of concrete addresses: by the definition of $\mathbf{l}_{\mathrm{s}}$ (Section 4.1), this boils down to $\mu(\mathscr{E}_0) = \{v_1, v_2, v_4\}$. Moreover, this concretization needs to trigger the unfolding rule (last rule in Figure 4.3) four times in order to produce the shape of Figure 4.4(b). The first three unfoldings successively generate points-to edges at α_1, α_2 and α_4, and the last unfolding constrains α_6 to be null. The shape of Figure 4.4(d) summarizes only the last two elements of the list (in purple) while the first element (in red) is preserved in its unfolded form. Similarly to the previous case, the unfolding of this shape needs to trigger the unfolding rule three times in order to get the shape of Figure 4.4(b) and to map $\mathscr{E}_1$ to $\mu(\mathscr{E}_1) = \{v_2, v_4\}$.

Finally, the shape of Figure 4.4(e) summarizes two list elements with inductive predicate $\alpha_2 \cdot \mathbf{l}_{\mathrm{s}}(\mathscr{E}_3)$ (in purple) and the rest of the list with a segment predicate (in red). The meaning of the segment predicate $\alpha_1 \cdot \mathbf{l}_{\mathrm{s}}(\mathscr{F}_1) \mathrel{*\!\!=} \alpha_2 \cdot \mathbf{l}_{\mathrm{s}}(\mathscr{F}_2)$ can be expressed by the following segment


a_0 = &l
v_1
v_2
v_3
v_4
v_5
v_6 = 0x0
v_7
α_0
α_1
n
α_2
n
α_4
n
α_6
v
α_3
v
α_5
v
α_7
$\mathbf{l}_s(\mathscr{E}_0)$
$\mathbf{l}_s(\mathscr{E}_1)$
$\mathbf{l}_s(\mathscr{F}_1)$
$\mathbf{l}_s(\mathscr{F}_2)$
$\mathbf{l}_s(\mathscr{F})$

(a) An example concrete memory

(b) Shape, no summarization

(c) Shape, summarization

(d) Shape, partial summarization

(e) Segment and inductive summaries

(f) Alternative segment and inductive summaries

Figure 4.4: Abstract shape graphs and their concretization

definition derived from the inductive definition $\mathbf{l}_s$:

$$\begin{aligned}&\alpha \cdot \mathbf{l}_s(\mathscr{E}) \ast\!\!= \beta \cdot \mathbf{l}_s(\mathscr{F}) ::= \\ &\qquad \mathbf{emp} \wedge \alpha = \beta \wedge \mathscr{E} = \mathscr{F} \\ &\quad \vee \;\; \alpha \cdot \mathtt{n} \mapsto \alpha_0 \ast \alpha \cdot \mathtt{v} \mapsto \alpha_1 \ast \alpha_0 \cdot \mathbf{l}_s(\mathscr{E}_0) \ast\!\!= \beta \cdot \mathbf{l}_s(\mathscr{F}) \wedge \alpha_0 \neq \alpha_1 \wedge \mathscr{E} = \{\alpha\} \uplus \mathscr{E}_0\end{aligned}$$

Thus, set variables $\mathscr{F}_1, \mathscr{F}_2$ of the segment $\alpha_1 \cdot \mathbf{l}_s(\mathscr{F}_1) \ast\!\!= \alpha_2 \cdot \mathbf{l}_s(\mathscr{F}_2)$ describe two sets of addresses such that the set of addresses of the list elements summarized in the segment is exactly $\mu(\mathscr{F}_1) \setminus \mu(\mathscr{F}_2)$.

4.3 Properties of Set Parameters

Inductive definitions with set parameters (Section 4.1) provide a scheme to summarize data structures with sharing. However, automatically inferring accurate set parameters for summary predicates during shape analysis turns out to be a very hard task as they depend on complex properties of the data-structure shapes and contents.

As an example, let us consider folding the points-to edges from node α_1 in Figure 4.4(d) into the segment edge in Figure 4.4(e) and inferring set parameters $\mathscr{F}_1$ and $\mathscr{F}_2$. According to the segment definition shown in Example 4.1, the folding process infers that $\mathscr{F}_1 = \{\alpha_1\} \uplus \mathscr{F}_2$, which means set variables $\mathscr{F}_1$ and $\mathscr{F}_2$ can be concretized into any sets $\mu(\mathscr{F}_1)$ and $\mu(\mathscr{F}_2)$ such that $\mu(\mathscr{F}_1) \setminus \mu(\mathscr{F}_2) = \mu(\alpha_1)$. In other words, only the difference between these two set parameters matters. However, proving data structure preservation of the abstract state shown in Figure 4.4(e) requires the set predicate $\mathscr{F}_2 = \mathscr{E}_1$ which cannot be inferred directly according to the segment definition shown in Example 4.1. Indeed, as only the difference between set parameters $\mathscr{F}_1$ and $\mathscr{F}_2$ matters, the segment predicate can be simplified with only one set parameter $\mathscr{F}$ denoting the difference between $\mathscr{F}_1$ and $\mathscr{F}_2$, such as $\alpha_1 \cdot \mathbf{l}_\mathrm{s} \ast\!\!=_{(\mathscr{F})} \alpha_2 \cdot \mathbf{l}_\mathrm{s}$. Thus, set parameter $\mathscr{F}$ can be simply inferred as $\{\alpha_1\}$ in the folding process. Figure 4.4(f) shows the graphical notation of the segment predicate $\alpha_1 \cdot \mathbf{l}_\mathrm{s} \ast\!\!=_{(\mathscr{F})} \alpha_2 \cdot \mathbf{l}_\mathrm{s}$.

In the following, we identify so-called parameter kinds, that simplify the inference of segment parameters in join, and make such simplified definitions possible.

Constant parameters. Intuitively, a set parameter of an inductive definition is *constant* if it is propagated with no modification to all the recursive calls of the inductive definition. We note that the set parameter $\mathscr{E}$ of **edges** (Figure 3.3) and also the first set parameter $\mathscr{E}$ of **nodes** (Figure 3.3) are *constant* set parameters. In the following, we write $\mathbf{ind} \vdash \mathscr{E} : \mathbf{cst}$ to denote that parameter $\mathscr{E}$ of inductive definition **ind** is constant.

Inferring constant parameters. To automatically infer whether a set parameter $\mathscr{E}$ of an inductive definition $\alpha \cdot \mathbf{ind}(\mathscr{E}) ::= r_0 \vee r_1 \vee \ldots \vee r_k$ is *constant*, we use the following derivation rules on the inductive definition **ind**:

- parameter $\mathscr{E}$ is constant if and only if it is so for each heap part of a rule of **ind**:

$$\frac{\forall i, 0 \leq i \leq k \implies r_i \vdash_{\mathbf{ind}} \mathscr{E} : \mathbf{cst}}{\mathbf{ind} \vdash \mathscr{E} : \mathbf{cst}} \qquad \frac{F_{\mathrm{Heap}} \vdash_{\mathbf{ind}} \mathscr{E} : \mathbf{cst}}{(F_{\mathrm{Heap}}, F_{\mathrm{Pure}}) \vdash_{\mathbf{ind}} \mathscr{E} : \mathbf{cst}}$$

- when considering a part of F_{Heap} that consists only of a call to **ind**, then the parameter is constant if and only if it is applied to the *same* set parameter as the current call:

$$\frac{}{\beta \cdot \mathbf{ind}(\mathscr{E}) \vdash_{\mathbf{ind}} \mathscr{E} : \mathbf{cst}}$$

- when considering a part of F_{Heap} that consists only of a call to another inductive definition **ind**′, and provided that definition does not call **ind** (even indirectly), then the parameter is constant:

$$\frac{\mathbf{ind}' \text{ does not call } \mathbf{ind} \text{ directly or indirectly}}{\beta \cdot \mathbf{ind}'(\mathscr{F}) \vdash_{\mathbf{ind}} \mathscr{E} : \mathbf{cst}}$$

- finally, the “is constant” property propagates naturally through the separating product and points-to predicate:

$$\frac{F_{\mathrm{Heap}} \vdash_{\mathbf{ind}} \mathscr{E} : \mathbf{cst} \qquad F'_{\mathrm{Heap}} \vdash_{\mathbf{ind}} \mathscr{E} : \mathbf{cst}}{F_{\mathrm{Heap}} \ast F'_{\mathrm{Heap}} \vdash_{\mathbf{ind}} \mathscr{E} : \mathbf{cst}} \qquad \frac{}{\alpha \cdot \mathtt{f} \mapsto \beta \vdash_{\mathbf{ind}} \mathscr{E} : \mathbf{cst}}$$

Properties of constant parameters. When a set parameter $\mathscr{E}$ of an inductive definition $\alpha \cdot \mathbf{ind}(\mathscr{E}) ::= r_0 \vee r_1 \vee \ldots \vee r_k$ is *constant* ($\mathbf{ind} \vdash \mathscr{E} : \mathbf{cst}$), segment predicates (e.g., $\alpha_1 \cdot \mathbf{l}_\mathrm{s}(\mathscr{E}_0) \ast\!\!= \alpha_2 \cdot \mathbf{l}_\mathrm{s}(\mathscr{E}_1)$) can be simplified with only one set parameter (e.g., $\alpha_1 \cdot \mathbf{l}_\mathrm{s} \ast\!\!=_{(\mathscr{F})} \alpha_2 \cdot \mathbf{l}_\mathrm{s}$) as any state (σ, μ) in the concretization of the segment $\alpha_1 \cdot \mathbf{l}_\mathrm{s}(\mathscr{E}_0) \ast\!\!= \alpha_2 \cdot \mathbf{l}_\mathrm{s}(\mathscr{E}_1)$ is such that $\mu(\mathscr{E}_0) = \mu(\mathscr{E}_1)$.

Theorem 4.1 (Folding of summary predicates with constant parameters). *Given an inductive definition* $\alpha \cdot \mathbf{ind}(\mathscr{E}) ::= r_0 \vee r_1 \vee \ldots \vee r_k$*, if* $\mathbf{ind} \vdash \mathscr{E} : \mathbf{cst}$*, then:*

$$\begin{aligned}
&\gamma_{\overline{\mathcal{G}}}(\alpha_0 \cdot \mathbf{ind} \ast\!\!=_{(\mathscr{E})} \alpha_1 \cdot \mathbf{ind} \ast \alpha_1 \cdot \mathbf{ind}(\mathscr{E})) \subseteq \gamma_{\overline{\mathcal{G}}}(\alpha_0 \cdot \mathbf{ind}(\mathscr{E}))\\
&\gamma_{\overline{\mathcal{G}}}(\alpha_0 \cdot \mathbf{ind} \ast\!\!=_{(\mathscr{E})} \alpha_1 \cdot \mathbf{ind} \ast \alpha_1 \cdot \mathbf{ind} \ast\!\!=_{(\mathscr{E})} \alpha_2 \cdot \mathbf{ind}) \subseteq \gamma_{\overline{\mathcal{G}}}(\alpha_0 \cdot \mathbf{ind} \ast\!\!=_{(\mathscr{E})} \alpha_2 \cdot \mathbf{ind})
\end{aligned}$$

Proof. The theorem can be proved by induction on the unfolding depths of the segment predicate $\alpha_0 \cdot \mathbf{ind} \ast\!\!=_{(\mathscr{E})} \alpha_1 \cdot \mathbf{ind}$. ■

Head parameters The second parameter of **nodes** (Figure 3.3) is clearly not constant as it collects the set of head nodes in all recursive inductive calls, and can be computed exactly from the values of the same parameters in the recursive calls. We call such a parameter a *head parameter*. This definition generalizes to non-linear structures (i.e., with several recursive calls corresponding to distinct sub-structures, like trees). Parameter $\mathscr{E}$ of the $\mathbf{l}_\mathrm{s}$ definition is also a *head* set parameter. In the following, we write $\mathbf{ind} \vdash \mathscr{E} : \mathbf{head}$ to denote that parameter $\mathscr{E}$ of inductive definition **ind** is head.

Inferring head parameters. To automatically infer whether a set parameter $\mathscr{E}$ of an inductive definition $\alpha \cdot \mathbf{ind}(\mathscr{E}) ::= r_0 \vee r_1 \vee \ldots \vee r_k$ is *head*, we use the following derivation rules on the inductive definition **ind**:

- parameter $\mathscr{E}$ is head for **ind** if and only if it is so for each inductive rule:

$$\frac{\forall 0 \le i \le k,\ \ r_i \vdash_{\mathbf{ind}} \mathscr{E} : \mathbf{head}}{\mathbf{ind} \vdash \mathscr{E} : \mathbf{head}}$$

- parameter $\mathscr{E}$ is head for a given rule that unfolds cells at α if and only if it is provably equal to the disjoint union of $\{\alpha\}$ and the arguments of the recursive sub-calls to **ind**, that is if it can be partitioned into a set of sets corresponding to the argument of each call and $\{\alpha\}$:

$$\frac{F_{\mathrm{Heap}} \vdash_{\mathbf{ind}} \{\mathscr{E}_i \mid 0 \le i \le n\} \qquad F_{\mathrm{Pure}} \vDash \biguplus_{0=i}^{n} \mathscr{E}_i \uplus \{\alpha\} = \mathscr{E}}{(F_{\mathrm{Heap}}, F_{\mathrm{Pure}}) \vdash_{\mathbf{ind}} \mathscr{E} : \mathbf{head}}$$

- parameter $\mathscr{E}$ is head for a given rule that unfolds to **emp** if and only if $\mathscr{E}$ is inferred as $\emptyset$:

$$\frac{F_{\mathrm{Pure}} \vDash \mathscr{E} = \emptyset}{(\mathbf{emp}, F_{\mathrm{Pure}}) \vdash_{\mathbf{ind}} \mathscr{E} : \mathbf{head}}$$

- rules for the separating conjunction, empty and points-to predicates express the linearity of head parameters (where, T, T' denote sets of set variables):

$$\frac{F_{\mathrm{Heap}} \vdash_{\mathbf{ind}} T \quad F'_{\mathrm{Heap}} \vdash_{\mathbf{ind}} T'}{F_{\mathrm{Heap}} \ast F'_{\mathrm{Heap}} \vdash_{\mathbf{ind}} T \uplus T'} \qquad \frac{}{\mathbf{emp} \vdash_{\mathbf{ind}} \emptyset} \qquad \frac{}{\alpha \cdot \mathtt{f} \mapsto \beta \vdash_{\mathbf{ind}} \emptyset}$$

- an inductive call to **ind** accounts for a single set variable corresponding to its parameter (the head addresses in the structure are exactly the head addresses of the sub-call):

$$\frac{}{\beta \cdot \mathbf{ind}(\mathscr{E}_i) \vdash_{\mathbf{ind}} \{\mathscr{E}_i\}}$$

- finally, calls to other inductive definitions contribute an empty set of head addresses in a structure:

$$\frac{\mathbf{ind}' \text{ does not call } \mathbf{ind} \text{ directly or indirectly}}{\beta \cdot \mathbf{ind}'(\mathscr{F}) \vdash_{\mathbf{ind}} \emptyset}$$

Properties of head parameters. When a set parameter $\mathscr{E}$ of an inductive definition $\alpha \cdot \mathbf{ind}(\mathscr{E}) ::= r_0 \vee r_1 \vee \ldots \vee r_k$ is *head* ($\mathbf{ind} \vdash \mathscr{E} : \mathbf{head}$), segment predicates (e.g., $\alpha_1 \cdot \mathbf{l}_{\mathrm{s}}(\mathscr{E}_0) \mathrel{*\!\!=} \alpha_2 \cdot \mathbf{l}_{\mathrm{s}}(\mathscr{E}_1)$) can be simplified with only one set parameter (e.g., $\alpha_1 \cdot \mathbf{l}_{\mathrm{s}} \mathrel{*\!\!=}_{(\mathscr{F})} \alpha_2 \cdot \mathbf{l}_{\mathrm{s}}$) as for any concretization (σ, μ) of the segment $\alpha_1 \cdot \mathbf{l}_{\mathrm{s}}(\mathscr{E}_0) \mathrel{*\!\!=} \alpha_2 \cdot \mathbf{l}_{\mathrm{s}}(\mathscr{E}_1)$, only the difference between $\mathscr{E}_0$ and $\mathscr{E}_1$ matters, that is, we can always find another valuation μ' ($\mu' \neq \mu$) such that $(\sigma, \mu') \in \gamma_{\overline{\mathcal{G}}}(\alpha_1 \cdot \mathbf{l}_{\mathrm{s}}(\mathscr{E}_0) \mathrel{*\!\!=} \alpha_2 \cdot \mathbf{l}_{\mathrm{s}}(\mathscr{E}_1))$ and $\mu'(\mathscr{E}_0) \setminus \mu'(\mathscr{E}_1) = \mu(\mathscr{E}_0) \setminus \mu(\mathscr{E}_1)$.

Theorem 4.2 (Folding of summary predicates with head parameters). *Given an inductive definition $\alpha \cdot \mathbf{ind}(\mathscr{E}) ::= r_0 \vee r_1 \vee \ldots \vee r_k$, if $\mathbf{ind} \vdash \mathscr{E} : \mathbf{head}$, then: for all $(\sigma, \mu) \in \mathcal{H} \times \mathcal{F}_\mu$*

- $$\begin{array}{l} (\sigma, \mu) \in \gamma_{\overline{\mathcal{G}}}(\alpha_0 \cdot \mathbf{ind} \mathrel{*\!\!=}_{(\mathscr{E}_0)} \alpha_1 \cdot \mathbf{ind} * \alpha_1 \cdot \mathbf{ind}(\mathscr{E}_1)) \wedge \mu \vdash \mathscr{E} = \mathscr{E}_0 \uplus \mathscr{E}_1 \\ \Longrightarrow \\ (\sigma, \mu) \in \gamma_{\overline{\mathcal{M}_\omega}}(\alpha_0 \cdot \mathbf{ind}(\mathscr{E})) \end{array}$$

- $$\begin{array}{l} (\sigma, \mu) \in \gamma_{\overline{\mathcal{G}}}(\alpha_0 \cdot \mathbf{ind} \mathrel{*\!\!=}_{(\mathscr{E}_0)} \alpha_1 \cdot \mathbf{ind} * \alpha_1 \cdot \mathbf{ind} \mathrel{*\!\!=}_{(\mathscr{E}_1)} \alpha_2 \cdot \mathbf{ind}) \wedge \mu \vdash \mathscr{E} = \mathscr{E}_0 \uplus \mathscr{E}_1 \\ \Longrightarrow \\ (\sigma, \mu) \in \gamma_{\overline{\mathcal{M}_\omega}}(\alpha_0 \cdot \mathbf{ind} \mathrel{*\!\!=}_{(\mathscr{E})} \alpha_2 \cdot \mathbf{ind}) \end{array}$$

Proof. The theorem can be proved by induction on the unfolding depths of the segment predicate $\alpha_0 \cdot \mathbf{ind} \mathrel{*\!\!=}_{(\mathscr{E}_0)} \alpha_1 \cdot \mathbf{ind}$. ■

To conclude, we have shown two kinds of set parameters, *constant* and *head*. Yet there may also be other kinds of set parameters, the abstractions that we present in this chapter and the analysis algorithm presented in Chapter 6 are not limited to constant and head set parameters. Moreover, constant and head set parameters correspond to a solution to a sharing pattern that is not limited to adjacency lists. For example, memory states that are composed of binary trees and lists with pointers to the tree can also be abstracted based on inductive definitions with constant and head set parameters and such memory states often appear in tree traversal algorithms. In addition, the study on kinds of set parameters may also allow us to design efficient shape analysis algorithms to support inductive predicates with numerical properties, such as sorted lists, lists of certain lengths and binary search trees.

Chapter 5

Abstract Domains for Set Reasoning

This chapter presents the work carried out in collaboration with Arlen Cox on set abstract domains, thus only the part strictly related to the thesis is presented. In the analysis of sharing data structures, set abstract domains are used for reasoning about set predicates. Specifically, Section 5.1 presents a common interface for set abstract domains, which is needed by the program analysis. Section 5.2 mainly presents a set abstract domain that is based on linear constraints over sets and briefly introduces a BDD-based set domain.

5.1 Set Constraints and Abstractions

In this section, we first define set predicates that a set abstract domain needs to reason about and define concrete states as models of set predicates. Then, we specify a common interface of set abstract domains.

5.1.1 Concrete States of Set Variables

Concrete states. We recall that $\mathcal{V}$ denotes the set of all concrete values, $\mathscr{E}, \mathscr{F} \in \mathcal{T}$ are set variables, and $\overline{v}_1, \overline{v}_2 \in \overline{\mathcal{V}}$ are abstract value variables. A concrete state is a valuation function $\mu \in \mathcal{F}_\mu$ that maps each abstract value to a concrete value $v \in \mathcal{V}$ and each set variable to a set of concrete values $\mathbb{V} \in \mathcal{P}(\mathcal{V})$.

Set predicates. We now set up the language of set predicates and its concrete semantics.

Definition 5.1 (Set predicates). *Set predicates are defined by the following grammar:*

$$\begin{array}{ll} \textit{set predicates} & \mathrm{c}_{\mathcal{T}} ::= \mathrm{c}_{\mathcal{T}} \wedge \mathrm{c}_{\mathcal{T}} \mid \mathrm{e}_{\mathcal{T}} \subseteq \mathrm{e}_{\mathcal{T}} \mid \mathrm{e}_{\mathcal{T}} = \mathrm{e}_{\mathcal{T}} \mid \overline{v} \in \mathrm{e}_{\mathcal{T}} \mid \mathbf{true} \mid \mathbf{false} \\ \textit{set expressions} & \mathrm{e}_{\mathcal{T}} ::= \emptyset \mid \{\overline{v}\} \mid \mathscr{E} \mid \mathrm{e}_{\mathcal{T}} \cup \mathrm{e}_{\mathcal{T}} \mid \mathrm{e}_{\mathcal{T}} \uplus \mathrm{e}_{\mathcal{T}} \end{array}$$

A set predicate $\mathrm{c}_{\mathcal{T}} \in \mathcal{C}_{\mathcal{T}}$ denotes a set of valuations which satisfy the predicate, that is $\{\mu \mid \mu \vDash \mathrm{c}_{\mathcal{T}}\}$. For clarity, we give a formal definition of $\mu \vDash \mathrm{c}_{\mathcal{T}}$ in Figure 5.1, where the semantics $[\![\mathrm{e}_{\mathcal{T}}]\!]\mu$ of a set expression $\mathrm{e}_{\mathcal{T}}$ under a concrete valuation μ is a set of concrete values.

$$
\begin{aligned}
&[\![\emptyset]\!]\mu = \emptyset\\
&[\![\{\overline{v}\}]\!]\mu = \{\mu(\overline{v})\}\\
&[\![\mathscr{E}]\!]\mu = \mu(\mathscr{E})\\
&[\![\mathrm{e}_{\mathcal{T}} \cup \mathrm{e}'_{\mathcal{T}}]\!]\mu = [\![\mathrm{e}_{\mathcal{T}}]\!]\mu \cup [\![\mathrm{e}'_{\mathcal{T}}]\!]\mu\\
&[\![\mathrm{e}_{\mathcal{T}} \uplus \mathrm{e}'_{\mathcal{T}}]\!]\mu = [\![\mathrm{e}_{\mathcal{T}}]\!]\mu \uplus [\![\mathrm{e}'_{\mathcal{T}}]\!]\mu\\
&\mu \vDash \mathbf{true} \qquad \mu \nvDash \mathbf{false}\\
&\mu \vDash \overline{v} \in \mathrm{e}_{\mathcal{T}} \iff \mu(\overline{v}) \in [\![\mathrm{e}_{\mathcal{T}}]\!]\mu\\
&\mu \vDash \mathrm{e}_{\mathcal{T}} \subseteq \mathrm{e}'_{\mathcal{T}} \iff [\![\mathrm{e}_{\mathcal{T}}]\!]\mu \subseteq [\![\mathrm{e}'_{\mathcal{T}}]\!]\mu\\
&\mu \vDash \mathrm{e}_{\mathcal{T}} = \mathrm{e}'_{\mathcal{T}} \iff [\![\mathrm{e}_{\mathcal{T}}]\!]\mu = [\![\mathrm{e}'_{\mathcal{T}}]\!]\mu\\
&\mu \vDash \mathrm{c}_{\mathcal{T}} \wedge \mathrm{c}'_{\mathcal{T}} \iff \mu \vDash \mathrm{c}_{\mathcal{T}} \wedge \mu \vDash \mathrm{c}'_{\mathcal{T}}
\end{aligned}
$$

Figure 5.1: Semantics of set predicates

set abstraction	$\overline{\mathrm{s}} \in \overline{\mathcal{S}}$
bottom check	$\overline{\mathbf{isbot}}_{\overline{\mathcal{S}}} \in \overline{\mathcal{S}} \to \{\mathbf{true}, \mathbf{false}\}$
prove	$\overline{\mathbf{prove}}_{\overline{\mathcal{S}}} : \overline{\mathcal{S}} \times \mathcal{C}_{\mathcal{T}} \to \{\mathbf{true}, \mathbf{false}\}$
guard	$\overline{\mathbf{guard}}_{\overline{\mathcal{S}}} : \overline{\mathcal{S}} \times \mathcal{C}_{\mathcal{T}} \to \overline{\mathcal{S}}$
project	$\overline{\mathbf{project}}_{\overline{\mathcal{S}}} : \overline{\mathcal{S}} \times \overline{\mathcal{V}} \uplus \mathcal{T} \to \overline{\mathcal{S}}$
rename	$\overline{\mathbf{rename}}_{\overline{\mathcal{S}}} : \overline{\mathcal{S}} \times (\overline{\mathcal{V}} \uplus \mathcal{T} \to \mathcal{P}(\overline{\mathcal{V}} \uplus \mathcal{T})) \to \overline{\mathcal{S}}$
inclusion check	$\sqsubseteq_{\overline{\mathcal{S}}}: \overline{\mathcal{S}} \times \overline{\mathcal{S}} \to \{\mathbf{true}, \mathbf{false}\}$
join	$\sqcup_{\overline{\mathcal{S}}} : \overline{\mathcal{S}} \times \overline{\mathcal{S}} \to \overline{\mathcal{S}}$
widen	$\triangledown_{\overline{\mathcal{S}}} : \overline{\mathcal{S}} \times \overline{\mathcal{S}} \to \overline{\mathcal{S}}$

Figure 5.2: The signature of set domains

5.1.2 Set Abstractions

A set abstraction $\overline{\mathrm{s}} \in \overline{\mathcal{S}}$ is concretized into a set of concrete states:

$$\gamma_{\overline{\mathcal{S}}} : \overline{\mathcal{S}} \to \mathcal{P}(\mathcal{V})$$

Moreover, we assume that set abstract domains always contain the $\bot$ and $\top$ abstract elements, which are respectively concretized into the empty set $\emptyset$ and the full set $\mathcal{V}$.

Example 5.1 (Singleton set domain). A very basic example of such a domain is the singleton set domain that comprises: $\bot$, $\top$ and a function from set variables $\mathcal{T}$ into their singleton value $\overline{\mathcal{V}}$. For instance, the abstract element $\{\mathscr{E} \mapsto \alpha, \mathscr{F} \mapsto \beta\}$ stands for $\mathscr{E} = \{\alpha\} \wedge \mathscr{F} = \{\beta\}$ and can be concretized into $\{\mu \mid \mu \vDash \mathscr{E} = \{\alpha\} \wedge \mathscr{F} = \{\beta\}\}$.

Operations over set abstractions. The main operations that a set abstract domain should provide is formalized in Figure 5.2.

The abstract operator $\overline{\textbf{isbot}}_{\overline{\mathcal{S}}}$ is used to conservatively determine if an abstract state describes unsatisfiable set constraints. As an example, for the singleton set domain presented in Example 5.1, $\overline{\textbf{isbot}}_{\overline{\mathcal{S}}}(\overline{\mathrm{s}})$ returns **true** when the abstract element $\overline{\mathrm{s}}$ is $\perp$. The soundness condition of the operator is that:

$$\overline{\textbf{isbot}}_{\overline{\mathcal{S}}}(\overline{\mathrm{s}}) = \textbf{true} \implies \gamma_{\overline{\mathcal{S}}}(\overline{\mathrm{s}}) = \emptyset$$

The abstract operator $\overline{\textbf{prove}}_{\overline{\mathcal{S}}}$ is used to conservatively determine if a set constraint $\mathrm{c}_{\mathcal{T}} \in \mathcal{C}_{\mathcal{T}}$ is implied by a set abstract element $\overline{\mathrm{s}} \in \overline{\mathcal{S}}$. A singleton set abstract element can only imply set predicates of the forms $\mathscr{E} = \{\alpha\}$ and $\alpha \in \mathscr{E}$, e.g., $\overline{\textbf{prove}}_{\overline{\mathcal{S}}}(\{\mathscr{E} \mapsto \alpha, \mathscr{F} \mapsto \beta\}, \beta \in \mathscr{F}) = \textbf{true}$. The soundness condition of the operator is that:

$$\overline{\textbf{prove}}_{\overline{\mathcal{S}}}(\overline{\mathrm{s}}, \mathrm{c}_{\mathcal{T}}) = \textbf{true} \implies \gamma_{\overline{\mathcal{S}}}(\overline{\mathrm{s}}) \subseteq \gamma_{\mathcal{C}_{\mathcal{T}}}(\mathrm{c}_{\mathcal{T}})$$

The operator $\overline{\textbf{guard}}_{\overline{\mathcal{S}}}$ is used to enforce a set constraint $\mathrm{c}_{\mathcal{T}} \in \mathcal{C}_{\mathcal{T}}$ on a set abstraction $\overline{\mathrm{s}} \in \overline{\mathcal{S}}$. For instance, $\overline{\textbf{prove}}_{\overline{\mathcal{S}}}(\{\mathscr{E} \mapsto \alpha\}, \mathscr{E}' = \{\alpha\})$ enforces set predicate $\mathscr{E}' = \{\alpha\}$ on the singleton set abstract element $\{\mathscr{E} \mapsto \alpha\}$ and gets abstract element $\{\mathscr{E} \mapsto \alpha, \mathscr{E}' \mapsto \alpha\}$. The soundness condition of the operator is that:

$$\overline{\textbf{guard}}_{\overline{\mathcal{S}}}(\overline{\mathrm{s}}, \mathrm{c}_{\mathcal{T}}) = \overline{\mathrm{s}}' \implies \gamma_{\overline{\mathcal{S}}}(\overline{\mathrm{s}}) \cap \gamma_{\mathcal{C}_{\mathcal{T}}}(\mathrm{c}_{\mathcal{T}}) \subseteq \gamma_{\overline{\mathcal{S}}}(\overline{\mathrm{s}}')$$

The operator $\overline{\textbf{project}}_{\overline{\mathcal{S}}}$ is used to project out set variables or abstract value variables that become redundant from a set abstraction. For example, projecting out a set variable from a singleton set abstract element can simply remove the map of the set variable. The soundness condition of the operator is that:

$$\begin{array}{l} \overline{\textbf{project}}_{\overline{\mathcal{S}}}(\overline{\mathrm{s}}, \mathscr{E}) = \overline{\mathrm{s}}' \implies \forall \mu \in \gamma_{\overline{\mathcal{S}}}(\overline{\mathrm{s}}), \mu_{\lceil \mathrm{dom}(\overline{\mathrm{s}}) \setminus \mathscr{E}} \in \gamma_{\overline{\mathcal{S}}}(\overline{\mathrm{s}}') \\ \overline{\textbf{project}}_{\overline{\mathcal{S}}}(\overline{\mathrm{s}}, \overline{v}) = \overline{\mathrm{s}}' \implies \forall \mu \in \gamma_{\overline{\mathcal{S}}}(\overline{\mathrm{s}}), \mu_{\lceil \mathrm{dom}(\overline{\mathrm{s}}) \setminus \overline{v}} \in \gamma_{\overline{\mathcal{S}}}(\overline{\mathrm{s}}') \end{array}$$

The operator $\overline{\textbf{rename}}_{\overline{\mathcal{S}}}$ is used to rename set variables and abstract value variables in set abstract elements and is needed in the joining process of abstract memory states (presented in Chapter 6). Let us assume that $f \in \overline{\mathcal{V}} \uplus \mathcal{T} \to \mathcal{P}(\overline{\mathcal{V}} \uplus \mathcal{T})$ is a renaming map of a set abstract element $\overline{\mathrm{s}}$, which maps a set variable $\mathscr{E}$ (resp., an abstract value variable $\overline{v}$) of $\overline{\mathrm{s}}$ to a set of set variables (resp., a set of abstract value variables) that share the same property with $\mathscr{E}$ (resp., $\overline{v}$). As an example, renaming set variable $\mathscr{E}$ to $\{\mathscr{E}', \mathscr{F}'\}$ and symbolic variable α to α' of the singleton set abstract element $\{\mathscr{E} \mapsto \alpha\}$ gets $\{\mathscr{E}' \mapsto \alpha', \mathscr{F}' \mapsto \alpha'\}$. The soundness condition of the operator is that:

$$\begin{array}{ll} & \overline{\textbf{rename}}_{\overline{\mathcal{S}}}(\overline{\mathrm{s}}, f) = \overline{\mathrm{s}}' \\ \implies & \\ & \forall \mu \in \gamma_{\overline{\mathcal{S}}}(\overline{\mathrm{s}}), \forall \mu', \forall \mathscr{E} \in \mathrm{dom}(f), \forall \mathscr{E}' \in f(\mathscr{E}), \mu'(\mathscr{E}') = \mu(\mathscr{E}) \implies \mu' \in \gamma_{\overline{\mathcal{S}}}(\overline{\mathrm{s}}') \end{array}$$

The inclusion checking operator $\sqsubseteq_{\overline{\mathcal{S}}}$ conservatively decides implication among set abstract elements. It is necessary in verifying the convergence of abstract iterations and proving postconditions. As an example, checking the implication between singleton set abstract elements $\overline{\mathrm{s}}$

and $\bar{\mathrm{s}}'$, i.e., $\sqsubseteq_{\overline{\mathcal{S}}} (\bar{\mathrm{s}}, \bar{\mathrm{s}}')$, simply requires proving $\bar{\mathrm{s}}' \subseteq \bar{\mathrm{s}}$. The soundness condition of the operator is that:

$$\sqsubseteq_{\overline{\mathcal{S}}} (\bar{\mathrm{s}}, \bar{\mathrm{s}}') = \mathbf{true} \implies \gamma_{\overline{\mathcal{S}}}(\bar{\mathrm{s}}) \subseteq \gamma_{\overline{\mathcal{S}}}(\bar{\mathrm{s}}')$$

The join operator $\sqcup_{\overline{\mathcal{S}}}$ *and the widening operator* $\triangledown_{\overline{\mathcal{S}}}$ take two set abstract elements and aim to compute a common weakening of them. In addition to that, the widening operator $\triangledown_{\overline{\mathcal{S}}}$ ensures termination of any sequence of abstract iterates. As an example, joining (resp., widening) singleton set abstract elements $\bar{\mathrm{s}}$ and $\bar{\mathrm{s}}'$, i.e., $\sqcup_{\overline{\mathcal{S}}}(\bar{\mathrm{s}}, \bar{\mathrm{s}}')$ (resp., $\triangledown_{\overline{\mathcal{S}}}(\bar{\mathrm{s}}, \bar{\mathrm{s}}')$), simply requires computing the intersection $\bar{\mathrm{s}} \cap \bar{\mathrm{s}}'$. The soundness condition of the joining operator $\sqcup_{\overline{\mathcal{S}}}$ (which is similar for $\triangledown_{\overline{\mathcal{S}}}$) is that:

$$\sqcup_{\overline{\mathcal{S}}}(\bar{\mathrm{s}}_l, \bar{\mathrm{s}}_r) = \bar{\mathrm{s}}_o \implies \gamma_{\overline{\mathcal{S}}}(\bar{\mathrm{s}}_l) \subseteq \gamma_{\overline{\mathcal{S}}}(\bar{\mathrm{s}}_o) \land \gamma_{\overline{\mathcal{S}}}(\bar{\mathrm{s}}_r) \subseteq \gamma_{\overline{\mathcal{S}}}(\bar{\mathrm{s}}_o)$$

5.2 Set Domains

Set domains are needed in constructing some complex program analyses [CCR14, CCS13, LRC15] to reason about unbounded collections of elements. Different set domains usually precisely express different predicates. Several set domains have been developed including a linear set domain, a BDD-based set domain and a QUIC-graph based set domain.

- The *linear set domain* focuses on reasoning about linear partitions of sets and is mainly designed to be precise enough and efficient for the analysis of shared data structures in the MemCAD analyzer.
- The *BDD-based set domain* focuses on reasoning about symbolic set predicates, i.e., non content related set predicates, and was designed by Arlen Cox for automatic analysis of open objects in dynamic language programs [CCR14]; it was later also used in the MemCAD analyzer. The BDD-based set domain is more general and precise than the linear domain, but less reliable in terms of scalability.
- The *QUIC-graphs based set domain* [CCS13] uses directed hyper-graph data structures to represent constraints of the form $\mathscr{E}_1 \cap \ldots \cap \mathscr{E}_n \subseteq \mathscr{F}_1 \cup \ldots \cup \mathscr{F}_m$ on subset relations and contents of set variables. It is used in automatically inferring relations between the set of values stored in containers such as arrays, lists, dictionaries, and sets.

Moreover, precision and scalability of set domains can also be improved by an equality domain functor [CCLR15] which handles equality constraints externally and prevents them being seen by the underlying abstract domains, and a packing domain functor [CCLR15] which dynamically reduces a complex abstraction to a set of small clusters. In this section, we only present the linear set domain and the BDD-based set domain in detail and refer the readers to [CCLR15] for more set abstract domains or techniques to improve the performance of set abstract domains.

5.2.1 Linear Set Domain

Linear set abstractions aim to precisely abstract set predicates that are composed of linear set constraints of the form $\mathscr{E}_0 = \{\overline{v}_0, \ldots, \overline{v}_n\} \uplus \mathscr{E}_1 \uplus \ldots \uplus \mathscr{E}_m$, inclusion constraints of the forms $\mathscr{E} \subseteq \mathscr{F}$ and $\overline{v} \in \mathscr{E}$, and equality constraints of the form $\mathscr{E} = \mathscr{F}$. Such constraints are very useful in the analysis of sharing data structures and allow quite a straightforward normalization representation of set abstractions, for which efficient algorithms are possible.

Definition 5.2 (Linear set abstract elements). *A linear set abstract element is either* $\top$, $\bot$ *or a tuple* $(\overline{c_l}, \overline{c_e}, \overline{c_i})$ *defined by the following grammar:*

$$\begin{array}{lcl}
\overline{\mathrm{s}} \in \overline{\mathcal{S}} & ::= & \top \mid \bot \mid (\overline{c_l}, \overline{c_e}, \overline{c_i}) \\
\overline{c_l} & ::= & \textbf{true} \mid \mathscr{E}_0 = \{\overline{v}_0, \ldots, \overline{v}_n\} \uplus \mathscr{E}_1 \uplus \ldots \uplus \mathscr{E}_m \mid \overline{c_l} \wedge \overline{c_l} \\
\overline{c_e} & ::= & \textbf{true} \mid \mathscr{E} = \mathscr{F} \mid \overline{c_e} \wedge \overline{c_e} \\
\overline{c_i} & ::= & \textbf{true} \mid \overline{v} \in \mathscr{E} \mid \mathscr{F} \subseteq \mathscr{E} \mid \overline{c_i} \wedge \overline{c_i}
\end{array}$$

The linear formula $\overline{c_l}$ expresses linear set equality constraints over set variables and is implemented as a map, where each set variable may appear at most once as the left-hand side. Linear set equality constraints are always normalized in the abstraction by expanding nested linear equality constraints, for example, the set constraint $\mathscr{E} = \mathscr{E}_1 \uplus \mathscr{E}_2 \wedge \mathscr{E}_2 = \mathscr{E}_3 \uplus \mathscr{E}_4$ is rewritten as $\mathscr{E} = \mathscr{E}_1 \uplus \mathscr{E}_3 \uplus \mathscr{E}_4 \wedge \mathscr{E}_2 = \mathscr{E}_3 \uplus \mathscr{E}_4$ in $\overline{c_l}$ of a linear set abstract element. Moreover, the emptiness property of a set variable $\mathscr{E}$ can also be expressed in $\overline{c_l}$ by mapping $\mathscr{E}$ to an empty set. The equality formula $\overline{c_e}$ and the inclusion formula $\overline{c_i}$ respectively express equality constraints and inclusion constraints.

Example 5.2 (A precise linear set abstract element). The concrete states satisfying set predicate $\mathscr{E} \subseteq \mathscr{F} \wedge \mathscr{F} = \{\alpha\} \uplus \mathscr{F}_2 \uplus \mathscr{F}_3 \wedge \delta \in \mathscr{E}$ can be precisely abstracted by the linear set abstract element $(\overline{c_l}, \overline{c_e}, \overline{c_i})$, where:

$$\begin{array}{l}
\overline{c_l} = (\mathscr{F} = \{\alpha\} \uplus \mathscr{F}_2 \uplus \mathscr{F}_3) \\
\overline{c_e} = \textbf{true} \\
\overline{c_i} = (\mathscr{E} \subseteq \mathscr{F} \wedge \delta \in \mathscr{F} \wedge \delta \in \mathscr{E})
\end{array}$$

Example 5.3 (An imprecise linear set abstract element). The concrete states satisfying set predicate $\mathscr{E} \subseteq \mathscr{F} \wedge \mathscr{F} = \{\alpha\} \cup \mathscr{F}_2 \cup \mathscr{F}_3 \wedge \delta \in \mathscr{E}$ can be abstracted by the linear set abstract element $(\overline{c_l}, \overline{c_e}, \overline{c_i})$, where

$$\begin{array}{l}
\overline{c_l} = \textbf{true} \\
\overline{c_e} = \textbf{true} \\
\overline{c_i} = (\mathscr{E} \subseteq \mathscr{F} \wedge \delta \in \mathscr{F} \wedge \alpha \in \mathscr{F} \wedge \mathscr{F}_2 \subseteq \mathscr{F} \wedge \mathscr{F}_3 \subseteq \mathscr{F} \wedge \delta \in \mathscr{E})
\end{array}$$

We note that, as the linear set domain cannot express non disjoint union, the set constraint $\mathscr{F} = \{\alpha\} \cup \mathscr{F}_2 \cup \mathscr{F}_3$ is weakened into $\alpha \in \mathscr{F} \wedge \mathscr{F}_2 \subseteq \mathscr{F} \wedge \mathscr{F}_3 \subseteq \mathscr{F}$.

Concretization. As any linear set abstract element can be translated into a conjunction of set constraints, the concretization $\gamma_{\overline{\mathcal{S}}} \in \overline{\mathcal{S}} \to \mathcal{P}(\mathcal{V})$ of the linear set abstract element can thus be defined as sets of concrete states that satisfy the corresponding set predicate.

Definition 5.3 (Concretization). *Given a linear set abstract element* $(\overline{c_l}, \overline{c_e}, \overline{c_i}) \in \overline{\mathcal{S}}$*,*

$$\gamma_{\overline{\mathcal{S}}}(\overline{c_l}, \overline{c_e}, \overline{c_i}) = \{\mu \mid \mu \vDash \overline{c_l} \wedge \overline{c_e} \wedge \overline{c_i}\}$$

Abstract operators. A great advantage of linear set abstract domain is that it allows fast abstract operations.

The abstract operation $\overline{\textbf{isbot}}_{\overline{\mathcal{S}}}(\overline{\mathrm{s}})$ simply returns **true** if $\overline{\mathrm{s}}$ is $\bot$.

The abstract operation $\overline{\textbf{prove}}_{\overline{\mathcal{S}}}(\overline{\mathrm{s}}, \mathrm{c}_{\mathcal{T}})$ encodes the set predicate $\mathrm{c}_{\mathcal{T}}$ into an equivalent normal form predicate of $\overline{\mathrm{s}}$ and then checks that $\overline{\mathrm{s}}$ includes the normal form predicate.

The abstract operation $\overline{\textbf{guard}}_{\overline{\mathcal{S}}}(\overline{\mathrm{s}}, \mathrm{c}_{\mathcal{T}})$ can simply enforce inclusion constraints into $\overline{c_i}$ and equality constraints into $\overline{c_e}$. Linear constraints are first normalized by expanding nested linear constraints and then enforced into $\overline{c_l}$. For set predicates that are not of these forms, the operation then weakens the predicates into these forms, where precision may be lost. When a set variable $\mathscr{E}$ is already constrained in the linear map $\overline{c_l}$, e.g., $\overline{c_l} = (\mathscr{E} = \{\alpha\} \uplus \mathscr{F})$, the $\overline{\textbf{guard}}_{\overline{\mathcal{S}}}$ operation cannot add another linear constraint of $\mathscr{E}$ in $\overline{c_l}$, where weakening may be also necessary.

Example 5.4 (Abstract guard operations). Let $\overline{c_l} = (\mathscr{F} = \{\alpha\} \uplus \mathscr{E})$, $\overline{c_e} = \textbf{true}$ and $\overline{c_i} = \textbf{true}$, then:

$$\overline{\textbf{guard}}_{\overline{\mathcal{S}}}((\overline{c_l}, \overline{c_e}, \overline{c_i}), \mathscr{E} = \{\beta\} \uplus \mathscr{E}_1) = (\overline{c_l} \wedge \mathscr{E} = \{\beta\} \uplus \mathscr{E}_1, \overline{c_e}, \overline{c_i})$$

and

$$\overline{\textbf{guard}}_{\overline{\mathcal{S}}}((\overline{c_l}, \overline{c_e}, \overline{c_i}), \mathscr{F} = \{\beta\} \uplus \mathscr{E}_1) = (\overline{c_l}, \overline{c_e}, \beta \in \mathscr{F} \wedge \mathscr{E}_1 \subseteq \mathscr{F})$$

The abstract operation $\overline{\textbf{project}}_{\overline{\mathcal{S}}}(\overline{\mathrm{s}}, \mathscr{E})$ either drops all the constraints related to $\mathscr{E}$ from $\overline{c_l}$, $\overline{c_e}$ and $\overline{c_i}$ of $\overline{\mathrm{s}}$ or replaces $\mathscr{E}$ with another equal set variable.

Example 5.5 (Abstract project operations). Let $\overline{c_l} = (\mathscr{F} = \{\alpha\} \uplus \mathscr{E})$, $\overline{c_e} = (\mathscr{E} = \mathscr{E}_1)$ and $\overline{c_i} = \textbf{true}$, then:

$$\overline{\textbf{project}}_{\overline{\mathcal{S}}}((\overline{c_l}, \overline{c_e}, \overline{c_i}), \alpha) = (\textbf{true}, \overline{c_e}, \mathscr{E} \subseteq \mathscr{F})$$

and

$$\overline{\textbf{project}}_{\overline{\mathcal{S}}}((\overline{c_l}, \overline{c_e}, \overline{c_i}), \mathscr{E}) = (\mathscr{F} = \{\alpha\} \uplus \mathscr{E}_1, \textbf{true}, \textbf{true})$$

Let $\mathrm{f} \in \overline{\mathcal{V}} \uplus \mathcal{T} \to \mathcal{P}(\overline{\mathcal{V}} \uplus \mathcal{T})$ be a mapping from set variables (resp., abstract value variables) of set abstract element $\overline{\mathrm{s}}$ to sets of other set variables (resp., abstract value variables). *The abstract rename operation* $\overline{\textbf{rename}}_{\overline{\mathcal{S}}}(\overline{\mathrm{s}}, \mathrm{f})$ then simply replaces any set variable $\mathscr{E}$ (resp., abstract value variable $\overline{v}$) of set abstraction $\overline{\mathrm{s}}$ by another set variable $\mathscr{F} \in \mathrm{f}(\mathscr{E})$ (resp., $\overline{v}' \in \mathrm{f}(\overline{\mathrm{v}})$) that it should be renamed to. When a set variable $\mathscr{E}$ is mapped to more than one set variable, e.g., $\mathscr{E}_1, \mathscr{E}_2 \in \mathrm{f}(\mathscr{E})$, equality between set variables $\mathscr{E}_1$ and $\mathscr{E}_2$ should be established in the resulting abstract element. Moreover, set variables of $\overline{\mathrm{s}}$ that cannot be renamed are dropped.

Given $\overline{\mathrm{s}} = (\overline{c_l}, \overline{c_e}, \overline{c_i})$ and $\overline{\mathrm{s}}' = (\overline{c_l}', \overline{c_e}', \overline{c_i}')$, *the inclusion checking operation* $\sqsubseteq_{\overline{\mathcal{S}}} (\overline{\mathrm{s}}, \overline{\mathrm{s}}')$ returns **true** when $\mathrm{dom}(\overline{\mathrm{s}}') = \mathrm{dom}(\overline{\mathrm{s}})$ and the set of constraints expressed by $(\overline{c_l}', \overline{c_e}', \overline{c_i}')$ is a subset of the set of constraints expressed by $(\overline{c_l}, \overline{c_e}, \overline{c_i})$. Finally, $\sqcup_{\overline{\mathcal{S}}}$ and $\triangledown_{\overline{\mathcal{S}}}$ over-approximate set abstract *encoding set expressions:*

$$\mathrm{tr}_E(\emptyset) = \mathbf{false}, \mathbf{true}$$

$$\mathrm{tr}_E(\mathscr{E}) = \mathscr{E}, \mathbf{true}$$

$$\mathrm{tr}_E(\mathrm{e}_{\mathcal{T}} \cup \mathrm{e}'_{\mathcal{T}}) = e \vee e', c \wedge c' \quad \textit{where } e, c = \mathrm{tr}_E(\mathrm{e}_{\mathcal{T}}) \textit{ and } e', c' = \mathrm{tr}_E(\mathrm{e}'_{\mathcal{T}})$$

$$\mathrm{tr}_E(\mathrm{e}_{\mathcal{T}} \uplus \mathrm{e}'_{\mathcal{T}}) = e \vee e', c \wedge c' \wedge \neg(e \wedge e') \quad \textit{where } e, c = \mathrm{tr}_E(\mathrm{e}_{\mathcal{T}}) \textit{ and } e', c' = \mathrm{tr}_E(\mathrm{e}'_{\mathcal{T}})$$

encoding set predicates:

$$\mathrm{tr}_E(\mathbf{true}) = \mathbf{true}$$

$$\mathrm{tr}_E(\mathbf{false}) = \mathbf{false}$$

$$\mathrm{tr}_E(\mathrm{e}_{\mathcal{T}} \subseteq \mathrm{e}'_{\mathcal{T}}) = (e \implies e') \wedge c \wedge c' \quad \textit{where } e, c = \mathrm{tr}_E(\mathrm{e}_{\mathcal{T}}) \textit{ and } e', c' = \mathrm{tr}_E(\mathrm{e}'_{\mathcal{T}})$$

$$\mathrm{tr}_E(\mathrm{e}_{\mathcal{T}} = \mathrm{e}'_{\mathcal{T}}) = (e \iff e') \wedge c \wedge c' \quad \textit{where } e, c = \mathrm{tr}_E(\mathrm{e}_{\mathcal{T}}) \textit{ and } e', c' = \mathrm{tr}_E(\mathrm{e}'_{\mathcal{T}})$$

$$\mathrm{tr}_E(\mathrm{c}_{\mathcal{T}} \wedge \mathrm{c}'_{\mathcal{T}}) = c \wedge c' \quad \textit{where } c = \mathrm{tr}_E(\mathrm{c}_{\mathcal{T}}) \textit{ and } c' = \mathrm{tr}_E(\mathrm{c}'_{\mathcal{T}})$$

Figure 5.3: Translation function tr_E

elements in the same way. Given $\bar{\mathrm{s}} = (\overline{c_l}, \overline{c_e}, \overline{c_i})$ and $\bar{\mathrm{s}}' = (\overline{c_l}', \overline{c_e}', \overline{c_i}')$ with $\mathrm{dom}(\bar{\mathrm{s}}) = \mathrm{dom}(\bar{\mathrm{s}}')$, the over-approximation $(\overline{c_l}'', \overline{c_e}'', \overline{c_i}'')$ of them is the intersection of the set of constraints expressed by $(\overline{c_l}, \overline{c_e}, \overline{c_i})$ and $(\overline{c_l}', \overline{c_e}', \overline{c_i}')$. Moreover, as the number of constraints decreases during the over-approximation operation, the widening operation guarantees termination.

Example 5.6 (Abstract joining operation). Let $\overline{c_l} = (\mathscr{F} = \{\mathscr{E}_1\} \wedge \mathscr{E}_2 = \emptyset)$, $\overline{c_e} = \mathbf{true}$ and $\overline{c_i} = \mathbf{true}$, and $\overline{c_l}' = (\mathscr{F} = \{\alpha\} \uplus \mathscr{E}_1 \wedge \mathscr{E}_2 = \{\alpha\})$, $\overline{c_e}' = \mathbf{true}$ and $\overline{c_i}' = \mathbf{true}$ then:

$$\sqcup_{\overline{\mathcal{S}}}((\overline{c_l}, \overline{c_e}, \overline{c_i}), (\overline{c_l}', \overline{c_e}', \overline{c_i}')) = (\mathscr{F} = \mathscr{E}_1 \uplus \mathscr{E}_2, \overline{c_e}, \overline{c_i})$$

5.2.2 BDD-based Set Domain

In this section, we briefly introduce a set abstract domain based on binary decision diagrams (BDDs) [Som99] and refer the readers to [Cox15] for more details. The core idea of the set domain is to encode set predicates into their Boolean algebraic forms and then represent Boolean algebraic forms as binary decision diagrams. We note that, set content reasoning, e.g., $\alpha \in \mathscr{E}$, cannot be supported directly by BDD-based set abstract domain, yet can be supported by a reduced product of BDD-based set abstract domain and the singleton set abstract domain presented in Example 5.1. Therefore, in the following, we only consider set predicates among set variables.

Encoding set predicates. A powerset with basic set operations forms a Boolean algebra with the join operator $\vee$ being the set union operator $\cup$, the meet operator $\wedge$ being the the set intersection operator $\cap$, the negative operator $\neg$ being set complement operator, **false** being the empty set $\emptyset$ and **true** being the universal set. Therefore, we can encode any set predicate into its Boolean algebra formula with the translation function tr_E presented in Figure 5.3. Generally, the translation is the literal replacement of set operations with the corresponding Boolean operation. As set predicates may contain disjoint union expressions, a set expression $\mathrm{e}_{\mathcal{T}}$ is translated into an expression e with a side constraint c, i.e., $\mathrm{tr}_E(\mathrm{e}_{\mathcal{T}}) = e, c$, where the side constraint c enforces disjoint properties.

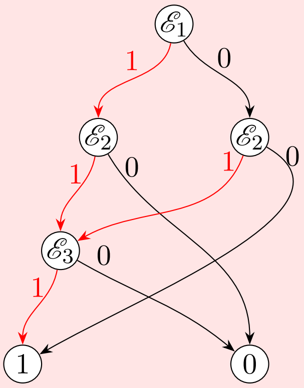
BDD-based set abstract elements. Canonical binary decision diagrams (BDDs) are used to canonically and efficiently represent Boolean algebras based on the if-then-else normal form and a total order $\prec$ of the variables. We define a BDD-based set abstract element as a canonical BDD.

Definition 5.4 (BDD-based set abstract elements). *A BDD-based set abstract element is composed of three basic syntactic elements:*

$$\bar{\mathrm{s}} ::= \textbf{true} \mid \textbf{false} \mid \textbf{ITE}(\mathscr{E}, \bar{\mathrm{s}}_1, \bar{\mathrm{s}}_2)$$

where, **true** *and* **false** *represent Boolean constants, and* $\textbf{ITE}(\mathscr{E}, \bar{\mathrm{s}}_1, \bar{\mathrm{s}}_2)$ *represent Boolean formula* $\mathscr{E} \implies \bar{\mathrm{s}}_1 \wedge \neg\mathscr{E} \implies \bar{\mathrm{s}}_2$.

Example 5.7 (A BDD-based set abstract element). The set predicate $\mathscr{E}_1 \subseteq \mathscr{E}_2 \wedge \mathscr{E}_2 \subseteq \mathscr{E}_3$ can be encoded as the Boolean algebraic formula $\mathscr{E}_1 \implies \mathscr{E}_2 \wedge \mathscr{E}_2 \implies \mathscr{E}_3$. Following the order $\mathscr{E}_1 \prec \mathscr{E}_2 \prec \mathscr{E}_3$, the Boolean algebraic formula can be represented as the BDD form:



$$\begin{aligned}
&\textbf{ITE}(\mathscr{E}_1,\\
&\quad \textbf{ITE}(\mathscr{E}_2, \textbf{ITE}(\mathscr{E}_3, \textbf{true}, \textbf{false}), \textbf{false}),\\
&\quad \textbf{ITE}(\mathscr{E}_2, \textbf{ITE}(\mathscr{E}_3, \textbf{true}, \textbf{false}), \textbf{true})\\
&)
\end{aligned}$$

Concretization. Conversely, a BDD-based set abstract element can be transferred into an equivalent set predicate and the abstract element can thus be concretized into a set of valuation functions satisfying the set predicate.

Definition 5.5 (Concretization). *Let $\mathcal{V}$ be the set of all values. Let* tr_R *be a function mapping a BDD-based set abstract element into a set expression:*

$$\begin{aligned}&\mathrm{tr}_R(\mathbf{true}) = \mathcal{V}\\&\mathrm{tr}_R(\mathbf{false}) = \emptyset\\&\mathrm{tr}_R(\mathbf{ITE}(\mathscr{E}, \bar{\mathrm{s}}_1, \bar{\mathrm{s}}_2)) = (\mathscr{E}^c \cup \mathrm{tr}_R(\bar{\mathrm{s}}_1)) \cap (\mathscr{E} \cup \mathrm{tr}_R(\bar{\mathrm{s}}_2))\end{aligned}$$

The concretization of a BDD-based set abstract element $\bar{\mathrm{s}}$ *is then defined as a set of valuations, where the semantics of the set expression* $\mathrm{tr}_R(\bar{\mathrm{s}})$ *is the set of all values* $\mathcal{V}$ *:*

$$\gamma_{\overline{\mathcal{S}}}(\bar{\mathrm{s}}) = \{\mu \mid [\![\mathrm{tr}_R(\bar{\mathrm{s}})]\!]\mu = \mathcal{V})\}$$

Abstract operators. Typical BDD implementations provide the basic logical operations $\wedge$, $\vee$, $\neg$, universal and existential quantifications $\exists$, $\forall$, and the validity checking functionality, from which the abstract operations can be derived straightforwardly. Thus, for the sake of simplicity, we only present a part of operations.

The operation $\overline{\mathbf{project}}_{\overline{\mathcal{S}}}(\bar{\mathrm{s}}, \mathscr{E})$ uses existential quantifier elimination provided by BDDs to drop variable $\mathscr{E}$:

$$\overline{\mathbf{project}}_{\overline{\mathcal{S}}}(\bar{\mathrm{s}}, \mathscr{E}) = \exists \mathscr{E}, \bar{\mathrm{s}}$$

The operations $\overline{\mathbf{prove}}_{\overline{\mathcal{S}}}$ and $\sqsubseteq_{\overline{\mathcal{S}}}$ simply rely on the validity checking functionality provided by BDDs.

The operations $\sqcup_{\overline{\mathcal{S}}}$ and $\triangledown_{\overline{\mathcal{S}}}$ can be implemented with the $\vee$ operation provided by BDDs, which is precise and does not need any rules or heuristics:

$$\sqcup_{\overline{\mathcal{S}}}(\bar{\mathrm{s}}_1, \bar{\mathrm{s}}_2) = \triangledown_{\overline{\mathcal{S}}}(\bar{\mathrm{s}}_1, \bar{\mathrm{s}}_2) = \bar{\mathrm{s}}_1 \vee \bar{\mathrm{s}}_2$$

Chapter 6

Static Analysis Algorithms for Unstructured Sharing Abstractions

This chapter proposes a static analysis algorithm to infer precise properties of shared data structures. Specifically, Section 6.1 recalls the abstract domain; Section 6.2 presents some basic abstract transfer functions and a novel operation, i.e., ***non-local unfolding****. Non-local unfolding enables "jumping" over inductive predicates. Section 6.3 shows algorithms that allow abstract memory regions to be conservatively folded into inductive predicates with* ***inductive set parameters synthesis****. Non-local unfolding and inductive set parameters synthesis are the new notions of the analysis algorithm. Finally, Section 6.5 reports on an empirical evaluation of the analysis algorithm.*

6.1 Abstract states

In Section 4.2, we define an *abstract state* as a pair $(\overline{\mathrm{g}}, \overline{\mathrm{c}})$ made up of an *abstract shape graph* $\overline{\mathrm{g}} \in \overline{\mathcal{G}}$ and a value-set abstract element $\overline{\mathrm{c}} \in \overline{\mathcal{C}}$. The concretization of an abstract memory state $\overline{\mathrm{m}} = (\overline{\mathrm{g}}, \overline{\mathrm{c}}) \in \overline{\mathcal{M}_\omega}$ is defined as a set of memory states such that for each memory state, a valuation of abstract value variables and set variables can be found that satisfy all constraints from $\overline{\mathrm{g}}$ and $\overline{\mathrm{c}}$:

$$\gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{g}}, \overline{\mathrm{c}}) = \{(\mu_{\lceil \mathcal{X}_\&}, \sigma) \in \mathcal{M} \mid \exists \mu \in \gamma_{\overline{\mathcal{C}}}(\overline{\mathrm{c}}),\ (\sigma, \mu) \in \gamma_{\overline{\mathcal{G}}}(\overline{\mathrm{g}})\}$$

Abstract shape graphs describe the basic structure of memory states and are defined as separating products of points-to predicates of the form $\alpha \cdot \mathtt{f} \mapsto \beta$, inductive predicates of the form $\alpha \cdot \mathbf{ind}(\vec{\mathscr{E}})$, and segment predicates of the form $\alpha \cdot \mathbf{ind}(\vec{\mathscr{E}}) \mathrel{*\!\!=} \beta \cdot \mathbf{ind}(\vec{\mathscr{E}})$.

A value-set abstract domain $\overline{\mathcal{C}}$ reasons about both numerical predicates and set predicates. It is built as a reduced product of a numerical domain and a set domain that is presented in Chapter 5, i.e., $\overline{\mathrm{c}} \in \overline{\mathcal{C}} ::= \overline{\mathcal{N}} \times \overline{\mathcal{S}}$, and thus each abstract operation of the value-set abstract domain can be simply passed to the underlying numerical domain and set domain. Let $\mathcal{C}$ denote the conjunction of numerical predicates $\mathcal{C}_{\overline{\mathcal{V}}}$ and set predicates $\mathcal{C}_{\mathcal{T}}$, the signature of value-set domains is defined as:

$$\begin{array}{ll}
\textit{abstraction} & \overline{\mathrm{c}} \in \overline{\mathcal{C}} \\
\textit{bottom check} & \overline{\mathbf{isbot}}_{\overline{\mathcal{C}}} \in \overline{\mathcal{C}} \to \{\mathbf{true}, \mathbf{false}\} \\
\textit{prove} & \overline{\mathbf{prove}}_{\overline{\mathcal{C}}} : \overline{\mathcal{C}} \times \mathcal{C} \to \{\mathbf{true}, \mathbf{false}\} \\
\textit{guard} & \overline{\mathbf{guard}}_{\overline{\mathcal{C}}} : \overline{\mathcal{C}} \times \mathcal{C} \to \overline{\mathcal{C}} \\
\textit{remove} & \overline{\mathbf{remove}}_{\overline{\mathcal{C}}} : \overline{\mathcal{C}} \times \overline{\mathcal{V}} \uplus \mathcal{T} \to \overline{\mathcal{C}} \\
\textit{rename} & \overline{\mathbf{rename}}_{\overline{\mathcal{C}}} : \overline{\mathcal{C}} \times (\overline{\mathcal{V}} \uplus \mathcal{T} \to \mathcal{P}(\overline{\mathcal{V}} \uplus \mathcal{T})) \to \overline{\mathcal{C}} \\
\textit{join} & \sqcup_{\overline{\mathcal{C}}} : \overline{\mathcal{C}} \times \overline{\mathcal{C}} \to \overline{\mathcal{C}} \\
\textit{widen} & \triangledown_{\overline{\mathcal{C}}} : \overline{\mathcal{C}} \times \overline{\mathcal{C}} \to \overline{\mathcal{C}} \\
\textit{inclusion check} & \sqsubseteq_{\overline{\mathcal{C}}} : \overline{\mathcal{C}} \times \overline{\mathcal{C}} \to \{\mathbf{true}, \mathbf{false}\}
\end{array}$$

6.2 Computation of Abstract Post-conditions

We have extended the abstract memory states defined in Section 2.3 by replacing numerical abstractions with value-set abstractions. We now need to extend the abstract operators and unfolding presented in Section 2.4 in order to design a forward abstract interpretation to compute *sound abstract post-conditions* from abstract pre-conditions. Moreover, the new abstract states allow a novel operator, namely non-local unfolding, which we highlight in this section.

6.2.1 Abstract Operators and Unfolding

Abstract operators. The basic abstract operations include the abstract store operations $\overline{\mathbf{read}}$, $\overline{\mathbf{write}}$, $\overline{\mathbf{create}}$ and $\overline{\mathbf{delete}}$, and abstract arithmetic operations, e.g., $\overline{[\![+]\!]}$, $\overline{[\![==]\!]}$. These operations can be defined very similarly to those defined in Section 2.4 by simply replacing numerical abstract operations with corresponding value-set abstract operations. For simplicity, we only show here the abstract store operation $\overline{\mathbf{create}}$ and the abstract arithmetic operation $\overline{[\![==]\!]}$.

The abstract store operation $\overline{\mathbf{create}} \in \mathbb{N} \times \overline{\mathcal{M}_\omega} \to (\overline{\mathcal{V}} \times \overline{\mathcal{M}_\omega})^2 \uplus \{\bot, \top\}$ is formalized as:

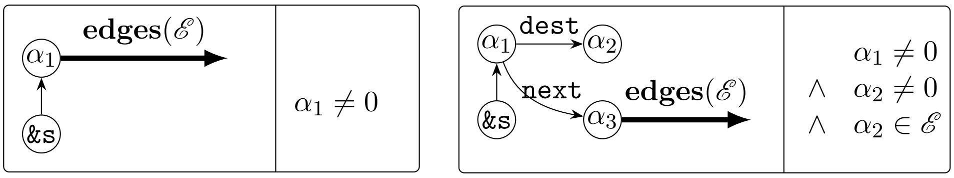
$$\begin{array}{l}
\overline{\mathbf{create}}(n, (\overline{\mathrm{g}}, \overline{\mathrm{c}}))\ \textit{returns} \\
\qquad (\alpha, (\overline{\mathrm{g}} * \alpha \mapsto \alpha_1 * \ldots * \alpha + 4(n-1) \mapsto \alpha_n, \overline{\mathrm{c}}))\ \textit{and}\ (\alpha, (\overline{\mathrm{g}}, \overline{\mathbf{guard}}_{\overline{\mathcal{C}}}(\overline{\mathrm{c}}, \alpha = 0))
\end{array}$$

The abstract operation $\overline{\mathbf{create}}(n, (\overline{\mathrm{g}}, \overline{\mathrm{c}}))$ returns a pair of abstract states respectively corresponding to a successful allocation of n cells and a failure case. In the second case, the symbolic address α is constrained to be 0 in the value-set abstraction by the abstract operation $\overline{\mathbf{guard}}_{\overline{\mathcal{C}}}(\overline{\mathrm{c}}, \alpha = 0)$.

The abstract arithmetic operation $\overline{[\![==]\!]} \in \overline{\mathcal{V}} \times \overline{\mathcal{V}} \times \overline{\mathcal{M}_\omega} \to (\{\mathbf{true}, \mathbf{false}\} \times \overline{\mathcal{M}_\omega})^2 \uplus \{\bot, \top\}$ is formalized as:

$$\begin{array}{l}
\overline{[\![==]\!]}(\overline{v}_1, \overline{v}_2, (\overline{\mathrm{g}}, \overline{\mathrm{c}})) = \\
\qquad (\mathbf{true}, (\overline{\mathbf{guard}}_{\overline{\mathcal{G}}}(\overline{\mathrm{g}}, \overline{v}_1 = \overline{v}_2), \overline{\mathbf{guard}}_{\overline{\mathcal{C}}}(\overline{\mathrm{c}}, \overline{v}_1 = \overline{v}_2))), \\
\textit{and}\ (\mathbf{false}, (\overline{\mathbf{guard}}_{\overline{\mathcal{G}}}(\overline{\mathrm{g}}, \overline{v}_1 \neq \overline{v}_2), \overline{\mathbf{guard}}_{\overline{\mathcal{C}}}(\overline{\mathrm{c}}, \overline{v}_1 \neq \overline{v}_2)))
\end{array}$$

The abstract operation $\overline{[\![==]\!]}(\overline{v}_1, \overline{v}_2, (\overline{\mathrm{g}}, \overline{\mathrm{c}}))$ takes two abstract values $\overline{v}_1$, $\overline{v}_2$ and an abstract state $(\overline{\mathrm{g}}, \overline{\mathrm{c}}) \in \overline{\mathcal{M}_\omega}$ as inputs, and respectively enforces equality and dis-equality conditions of $\overline{v}_1$ and $\overline{v}_2$ on the abstract state $(\overline{\mathrm{g}}, \overline{\mathrm{c}})$.



(a) Folded "pre"-shape $\overline{\mathrm{m}}$

(b) "Unfolded"-shape $\overline{\mathrm{m}}'$

Figure 6.1: Abstract states and unfolding

The soundness condition of the abstract operation can be similarly defined as in Section 2.4 and is omitted here.

Unfolding. When abstract operations involve memory regions that are summarized in inductive or segment predicates, the analysis needs to unfold summary predicates to expose memory regions on which store operations act. As an example, reading the abstract value denoted by `s -> next` of the abstract state shown in Figure 6.1(a) needs to unfold the inductive predicate $\alpha_1 \cdot \mathbf{edges}(\mathscr{E})$.

The principles of unfolding inductive predicates and segment predicates are very similar, as presented in Section 2.4.2, thus, we only formalize the unfolding of inductive predicates. Generally, unfolding an inductive predicate in one abstract state follows its inductive definition to produce a disjunction of cases, where one case is for one inductive rule. Following the inductive definitions presented in Section 4.1, each inductive rule comprises a heap part and a pure part, and the pure part comprises numerical constraints and also set constraints, thus, the unfolding replaces the inductive predicate with the heap part in the abstract shape graph, and enforces the pure constraints in the value-set abstraction. Symbolic variables and set variables that are not parameters of the inductive definition are all existential variables that need to be replaced by fresh variables of the abstract state.

Definition 6.1 (Unfolding). *Given inductive definition* $\alpha \cdot \mathbf{ind}(\mathscr{E}) ::= \bigvee_{i=1}^{k}(F_{\text{Heap},i}, F_{\text{Pure},i})$, *the unfolding operation* $\overline{\mathbf{unfold}} \in \overline{\mathcal{V}} \times \overline{\mathcal{M}_{\omega}} \to \overline{\mathcal{D}}$ *is defined as:*

$$\overline{\mathbf{unfold}}(\overline{v}, (\overline{\mathrm{g}} * \overline{v} \cdot \mathbf{ind}(\mathscr{F}), \overline{\mathrm{c}})) = \bigvee_{i=1}^{k}(\overline{\mathrm{g}} * F'_{\text{Heap},i}, \overline{\mathbf{guard}}_{\overline{\mathcal{C}}}(\overline{\mathrm{c}}, F'_{\text{Pure},i}))$$

where, $\mathrm{dom}(F'_{\text{Heap},i}, F'_{\text{Pure},i}) \cap \mathrm{dom}(\overline{\mathrm{g}}, \overline{\mathrm{c}}) = \{\overline{\mathrm{v}}, \mathscr{F}\}$ *and* $\exists \mathrm{f} \in \mathcal{V} \uplus \mathcal{T} \to \mathcal{V} \uplus \mathcal{T},\ \{\alpha \mapsto \overline{\mathrm{v}}, \mathscr{E} \mapsto \mathscr{F}\} \subseteq \mathrm{f},\ \mathrm{F}'_{\mathrm{Heap,i}} = \mathrm{F}_{\mathrm{Heap,i}} \circ \mathrm{f}, \mathrm{F}'_{\mathrm{Pure,i}} = \mathrm{F}_{\mathrm{Pure,i}} \circ \mathrm{f}.$

Example 6.1 (Unfolding). Let us consider unfolding the inductive predicate $\alpha_1 \cdot \mathbf{edges}(\mathscr{E})$

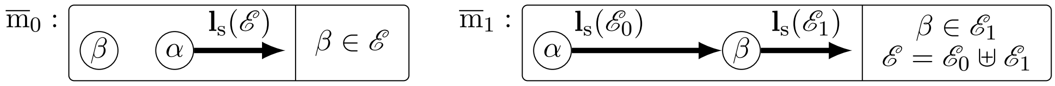

$\overline{\mathrm{m}}_0$:
β
α
$\mathbf{l}_{\mathrm{s}}(\mathscr{E})$
$\beta \in \mathscr{E}$
$\overline{\mathrm{m}}_1$:
α
$\mathbf{l}_{\mathrm{s}}(\mathscr{E}_0)$
β
$\mathbf{l}_{\mathrm{s}}(\mathscr{E}_1)$
$\beta \in \mathscr{E}_1$
$\mathscr{E} = \mathscr{E}_0 \uplus \mathscr{E}_1$

Figure 6.2: Non-local unfolding

of the abstract state presented in Figure 6.1(a) with the following inductive definition:

$$\begin{aligned}
&\alpha \cdot \mathbf{edges}(\mathscr{E}) ::= \\
&\qquad (\mathbf{emp}, \alpha = 0) \\
&\quad \vee \quad (\alpha \cdot \mathtt{dest} \mapsto \beta * \alpha \cdot \mathtt{next} \mapsto \gamma \\
&\qquad\ * \gamma \cdot \mathbf{edges}(\mathscr{E}), \alpha \neq 0 \wedge \beta \in \mathscr{E} \wedge \beta \neq 0)
\end{aligned}$$

Unfolding the inductive predicate $\alpha_1 \cdot \mathbf{edges}(\mathscr{E})$ with the first inductive rule produces a bottom abstract state $\bot$, since the abstract state $\overline{\mathrm{m}}$ contains constraint $\alpha_1 \neq 0$ in the value-set abstraction component. Unfolding $\alpha_1 \cdot \mathbf{edges}(\mathscr{E})$ with the second inductive rule produces the abstract state presented in Figure 6.1(b), where symbolic variables α, β, γ of the rule have been renamed to $\alpha_1, \alpha_2, \alpha_3$ respectively in the abstract state. That is:

$$\begin{aligned}
&\overline{\mathbf{unfold}}(\alpha_1, (\&\mathtt{s} \mapsto \alpha_1 * \alpha_1 \cdot \mathbf{edges}(\mathscr{E}), \overline{\mathrm{c}})) = \\
&\ \&\mathtt{s} \mapsto \alpha_1 * \alpha_1 \cdot \mathtt{dest} \mapsto \alpha_2 * \alpha_1 \cdot \mathtt{next} \mapsto \alpha_3 * \alpha_3 \cdot \mathbf{edges}(\mathscr{E}), \overline{\mathbf{guard}}_{\overline{\mathcal{C}}}(\overline{\mathrm{c}}, \alpha_2 \neq 0 \wedge \alpha_2 \in \mathscr{E})
\end{aligned}$$

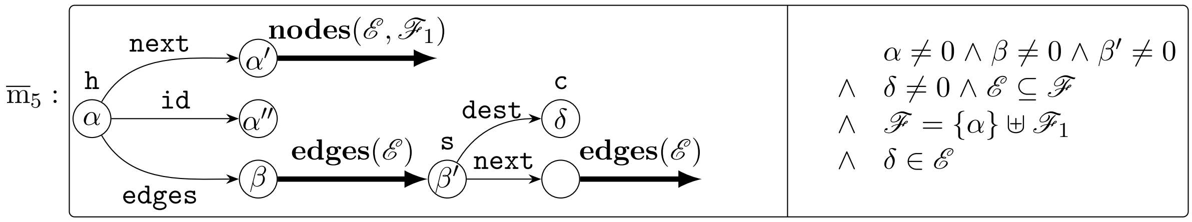
Theorem 6.1 (Soundness of $\overline{\mathbf{unfold}}$). *The unfolding operator* $\overline{\mathbf{unfold}}$ *is sound with respect to the soundness Condition 2.7, i.e.,*

$$\forall \overline{\mathrm{m}} \in \overline{\mathcal{M}_\omega},\ \overline{\mathbf{unfold}}(\overline{v}, \overline{\mathrm{m}}) = \overline{\mathrm{m}}_1 \vee \ldots \vee \overline{\mathrm{m}}_n \implies \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}) \subseteq \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}_1) \cup \ldots \cup \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}_n)$$

Proof. The soundness can be easily derived from the concretization rule of inductive predicates presented in Figure 4.3. ∎

6.2.2 Non-local Unfolding

Limitations of standard unfolding. The unfolding case studied so far is quite straightforward as the node at which the inductive predicate should be unfolded is well specified by the transfer function (e.g., α_1 in Figure 6.1(a)). However, the above unfolding cannot work when analyzing an instruction reading `c -> id` (line 10 of the graph traversal function shown in Figure 3.5) from the abstract state below:


$\overline{\mathrm{m}}_5$:
h
α
next
α'
$\mathbf{nodes}(\mathscr{E}, \mathscr{F}_1)$
id
α''
edges
β
$\mathbf{edges}(\mathscr{E})$
s
β'
dest
c
δ
next
$\mathbf{edges}(\mathscr{E})$
$\alpha \neq 0 \wedge \beta \neq 0 \wedge \beta' \neq 0$
$\wedge\ \delta \neq 0 \wedge \mathscr{E} \subseteq \mathscr{F}$
$\wedge\ \mathscr{F} = \{\alpha\} \uplus \mathscr{F}_1$
$\wedge\ \delta \in \mathscr{E}$

In the abstract state c points to δ in the abstract level, but node δ is not the origin of a points-to predicate nor of an inductive predicate that could be unfolded.

Principle of non-local unfolding. Intuitively, δ could be any node in the graph and thus we expect the abstract memory state to reflect this. This intuitive idea is formalized as follows: the second parameter of **nodes** is a *head parameter* (Section 4.3), and the side predicates express the fact that $\delta \in \{\alpha\} \uplus \mathscr{F}_1$, where $\mathscr{F}_1$ appears as a second parameter of the **nodes** predicate in the shape, which allows δ to be localized. This principle is a direct consequence of a property of head parameters:

Theorem 6.2 (Non-local unfolding principle). *Let* **ind** *be a single parameter inductive, such that* $\alpha \cdot \mathbf{ind}(\mathscr{E}) \vdash \mathscr{E} : \mathbf{head}$. *Let* $(\sigma, \mu) \in \gamma_{\overline{\mathcal{G}}}(\alpha \cdot \mathbf{ind}(\mathscr{E}))$ *such that* $\mu(\beta) \in \mu(\mathscr{E})$. *Given* $\mathscr{E}_0, \mathscr{E}_1$ *fresh set variables,* μ *can be extended into* μ'*, such that* $(\sigma, \mu') \in \gamma_{\overline{\mathcal{G}}}(\alpha \cdot \mathbf{ind} *\!=_{(\mathscr{E}_0)} \beta \cdot \mathbf{ind} * \beta \cdot \mathbf{ind}(\mathscr{E}_1))$*,* $\mu'(\mathscr{E}) = \mu'(\mathscr{E}_0) \uplus \mu'(\mathscr{E}_1)$ *and* $\mu'(\beta) \in \mu'(\mathscr{E}_1)$*.*

Proof. According to the definition of head parameters,

$$\alpha \cdot \mathbf{ind}(\mathscr{E}) \vdash \mathscr{E} : \mathbf{head} \implies \forall (\sigma, \mu) \in \gamma_{\overline{\mathcal{G}}}(\alpha \cdot \mathbf{ind}(\mathscr{E})), \mu(\mathscr{E}) = \emptyset \vee \mu(\alpha) \in \mu(\mathscr{E})$$

As $\mu(\beta) \in \mathscr{E}$ indicates $\mu(\mathscr{E}) \neq \emptyset$, we then need to prove that for all $1 \leq |\mu(\mathscr{E})|$, the theorem holds:

- if $|\mu(\mathscr{E})| = 1$, we have $\mu(\alpha) = \mu(\beta)$. Let $\mu' = \mu \uplus \{\mathscr{E}_0 \mapsto \emptyset, \mathscr{E}_1 \mapsto \mu(\mathscr{E})\}$, the theorem obviously holds.

- if $|\mu(\mathscr{E})| = n + 1$, we assume that the theorem holds when $|\mu(\mathscr{E})| \leq n$.

  In the case where $\mu(\beta) = \mu(\alpha)$, let $\mu' = \mu \uplus \{\mathscr{E}_0 \mapsto \emptyset, \mathscr{E}_1 \mapsto \mu(\mathscr{E})\}$, the theorem obviously holds.

  In the case where $\mu(\beta) \neq \mu(\alpha)$, the inductive predicate $\alpha \cdot \mathbf{ind}(\mathscr{E})$ can be unfolded as a separating conjunction of a set of inductive calls to **ind** and other predicates, denoted as $\alpha_1 \cdot \mathbf{ind}(\mathscr{F}_1) * \ldots * \alpha_k \cdot \mathbf{ind}(\mathscr{F}_k) * \overline{\mathrm{g}}$, such that the disjoint union of $\{\alpha\}$ and the arguments of the recursive sub-calls is equal to $\mathscr{E}$, i.e., $\mathscr{E} = \{\alpha\} \uplus \{\mathscr{F}_1, \ldots, \mathscr{F}_k\}$ and $\sigma = \sigma_0 \uplus \ldots \uplus \sigma_k$ such that $(\sigma_i, \mu) \in \gamma_{\overline{\mathcal{G}}}(\alpha_i \cdot \mathbf{ind}(\mathscr{F}_i))$.

  Let us assume that $\mu(\beta) \in \mu(\mathscr{F}_i)$ $(1 \leq i \leq k)$, since $|\mu(\mathscr{F}_i)| \leq n$, μ can be extended to μ'_i such that $(\sigma_i, \mu'_i) \in \gamma_{\overline{\mathcal{G}}}(\alpha_i \cdot \mathbf{ind} *\!=_{(\mathscr{X}_0)} \beta \cdot \mathbf{ind} * \beta \cdot \mathbf{ind}(\mathscr{X}_1))$, $\mu'_i(\mathscr{F}_i) = \mu'_i(\mathscr{X}_0) \uplus \mu'_i(\mathscr{X}_1)$ and $\mu'_i(\beta) \in \mu'_i(\mathscr{E}_1)$.

  Let $\mu' = \mu'_i \uplus \{\mathscr{E}_1 \mapsto \mu'_i(\mathscr{X}_1), \mathscr{E}_0 \mapsto \mu(\mathscr{E}) \setminus \mu'_i(\mathscr{X}_1)\}$, we get:
  $(\sigma, \mu') \in \gamma_{\overline{\mathcal{G}}}(\alpha_1 \cdot \mathbf{ind}(\mathscr{F}_1) * \ldots * \alpha_i \cdot \mathbf{ind} *\!=_{(\mathscr{X}_0)} \beta \cdot \mathbf{ind} * \beta \cdot \mathbf{ind}(\mathscr{E}_1)) * \ldots \alpha_k \cdot \mathbf{ind}(\mathscr{F}_k) * \overline{\mathrm{g}})$
  According to the inductive definition, finally we get: $(\sigma, \mu') \in \gamma_{\overline{\mathcal{G}}}(\alpha \cdot \mathbf{ind} *\!=_{(\mathscr{E}_0)} \beta \cdot \mathbf{ind} * \beta \cdot \mathbf{ind}(\mathscr{E}_1))$.

■

Figure 6.2 illustrates this *non-local* unfolding principle. While theorem 6.2 states the result for inductive definitions with a single set parameter, the result generalizes directly to the case of definitions with several parameters (only the parameter supporting non-local unfolding then needs to be a head parameter). It also generalizes to segments.

Non local unfolding algorithm. Given an inductive definition $\alpha \cdot \mathbf{ind}(\mathscr{E}) ::= \bigvee_{i=1}^{k}(F_{\mathrm{Heap},i}, F_{\mathrm{Pure},i})$, where $\alpha \cdot \mathbf{ind}(\mathscr{E}) \vdash \mathscr{E} : \mathbf{head}$, when unfolding at an abstract address α of an abstract state $(\overline{\mathrm{g}}, \overline{\mathrm{c}})$, the unfolding algorithm first searches for a local unfolding at α, that is either a segment or an inductive predicate starting from α. When no such local unfolding can be found, the non-local unfolding operation $\overline{\mathbf{unfold}}_{nc}$ is then performed. It searches for predicates of the form $\alpha \in \{\alpha_0, \ldots, \alpha_k\} \uplus \mathscr{E}_0 \uplus \ldots \uplus \mathscr{E}_l$, where $\mathscr{E}_0, \ldots, \mathscr{E}_l$ appear as head parameters. When it finds such a predicate, the analysis produces a disjunction of cases, where either $\alpha = \alpha_i$, or where $\alpha \in \mathscr{E}_i$ and it performs a non-local unfolding of the corresponding predicate (Theorem 6.2);

Application to the computation of post-conditions of assignments. The analysis of an assignment statement proceeds along the following steps:

1. it attempts to transform all l-values to edges and r-values to symbolic variables;
2. when step 1 fails because no points-to edge can be found at offset $\alpha \cdot \mathtt{f}$, it unfolds points-to predicates at α either by local unfolding or non-local unfolding.
3. The unfolding process usually produces a disjunction of cases. Then, the analysis performs the abstract operation $\overline{\mathbf{write}}$ on each unfolded disjunct.

Note that failure to fully materialize all required nodes would produce imprecise results, thus, in the absence of information about parameters, the analysis may fail to produce a precise post-condition as it did before. The purpose of non-local unfolding is to avoid imprecise results when dealing with structures with sharing.

6.3 Abstract Lattice Operations

While transfer functions *unfold* inductive predicates, inclusion checking, join and widening operators need to discover valid set parameters so as to *fold* them back. However, folding turns out to be much harder than unfolding, and the main difficulties for our abstract domian lie in searching for set parameters for folded summary predicates.

6.3.1 Inclusion Checking

As introduced in Section 2.4.4, given two abstract memory states $(\overline{\mathrm{g}}_l, \overline{\mathrm{n}}_l)$ and $(\overline{\mathrm{g}}_r, \overline{\mathrm{n}}_r)$, the inclusion checking abstract operation $\sqsubseteq_{\overline{\mathcal{M}_\omega}}$ attempts to establish that $\gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{g}}_l, \overline{\mathrm{n}}_l)$ is included in $\gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{g}}_r, \overline{\mathrm{n}}_r)$.

Original inclusion checking algorithm. The original inclusion checking algorithm of abstract states $\overline{\mathcal{G}} \times \overline{\mathcal{N}}$ presented in [CR08] is based on a series of syntactic rewriting rules of abstract shape graphs to establish the inclusion of abstract shape graphs and an implication checking of numerical abstractions to determine whether inclusion does indeed hold. Figure 6.3 shows such a system. The **i-pt** and **i-ind** rules state that any points-to and inductive predicates are included in themselves; the same principle actually also applies to segment predicates. The **i-sep** rule splits abstract shapes according to the separation principle. The **i-uf** rule unfolds the right hand side shape and tries to match the left-hand side with one of the disjuncts. The **i-segind** rule applies when trying to compare a segment on the left and an inductive predicate

$$\frac{}{\overline{\mathrm{n}}_l \vdash \alpha \cdot \mathtt{f} \mapsto \beta \sqsubseteq_{\overline{\mathcal{G}}} \alpha \cdot \mathtt{f} \mapsto \beta} \qquad (\mathbf{i-pt})$$

$$\frac{}{\overline{\mathrm{n}}_l \vdash \alpha \cdot \mathbf{ind} \sqsubseteq_{\overline{\mathcal{G}}} \alpha \cdot \mathbf{ind}} \qquad (\mathbf{i-ind})$$

$$\frac{\overline{\mathrm{n}}_l \vdash \overline{\mathrm{g}}_l \sqsubseteq_{\overline{\mathcal{G}}} \overline{\mathrm{g}}_r \qquad \overline{\mathrm{n}}_l \vdash \overline{\mathrm{g}}'_l \sqsubseteq_{\overline{\mathcal{G}}} \overline{\mathrm{g}}'_r}{\overline{\mathrm{n}}_l \vdash \overline{\mathrm{g}}_l * \overline{\mathrm{g}}'_l \sqsubseteq_{\overline{\mathcal{G}}} \overline{\mathrm{g}}_r * \overline{\mathrm{g}}'_r} \qquad (\mathbf{i-sep})$$

$$\frac{\overline{\mathrm{g}}_r \rightsquigarrow_{\mathrm{U}} (\overline{\mathrm{g}}_u, F_{\mathrm{Pure}}) \qquad \overline{\mathrm{n}}_l \vdash \overline{\mathrm{g}}_l \sqsubseteq_{\overline{\mathcal{G}}} \overline{\mathrm{g}}_u \quad \overline{\mathbf{prove}}_{\overline{\mathcal{N}}}(\overline{\mathrm{n}}_l, F_{\mathrm{Pure}}) = \mathbf{true}}{\overline{\mathrm{n}}_l \vdash \overline{\mathrm{g}}_l \sqsubseteq_{\overline{\mathcal{G}}} \overline{\mathrm{g}}_r} \qquad (\mathbf{i-uf})$$

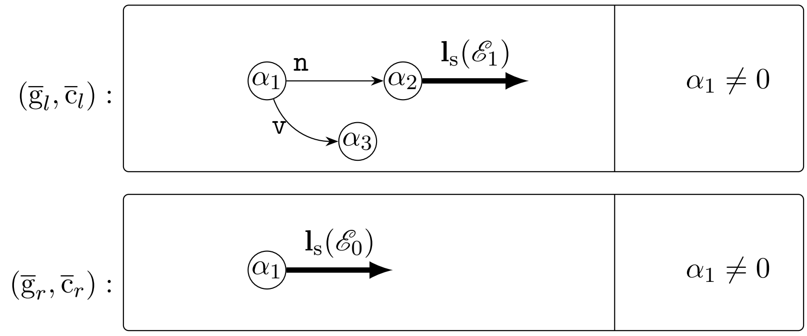
$$\frac{\overline{\mathrm{n}}_l \vdash \overline{\mathrm{g}}_l \sqsubseteq_{\overline{\mathcal{G}}} \beta \cdot \mathbf{ind}(\mathscr{E})}{\overline{\mathrm{n}}_l \vdash \alpha \cdot \mathbf{ind} \mathrel{*\!=} \beta \cdot \mathbf{ind} * \overline{\mathrm{g}}_l \sqsubseteq_{\overline{\mathcal{G}}} \alpha \cdot \mathbf{ind}} \qquad (\mathbf{i-segind})$$

$$\frac{\overline{\mathrm{n}}_l \vdash \overline{\mathrm{g}}_l \sqsubseteq_{\overline{\mathcal{G}}} \overline{\mathrm{g}}_r \qquad \overline{\mathrm{n}}_l \sqsubseteq_{\overline{\mathcal{N}}} \overline{\mathrm{n}}_r}{(\overline{\mathrm{g}}_l, \overline{\mathrm{n}}_l) \sqsubseteq_{\overline{\mathcal{M}_\omega}} (\overline{\mathrm{g}}_r, \overline{\mathrm{n}}_r)} \qquad (\mathbf{i-val})$$

Figure 6.3: Original inclusion checking over shapes and abstract states $\overline{\mathcal{G}} \times \overline{\mathcal{N}}$

on the right, that correspond to the same definition and the same origin. The last rule, **i-val**, returns **true** when both the comparison of shapes and of numerical abstractions return **true**.

However, the system cannot be simply extended when summary predicates are extended with set parameters and numerical abstractions are extended with set abstractions. As an example, let us consider proving the inclusion of the following abstract states ($\alpha \cdot \mathbf{l}_{\mathrm{s}}(\mathscr{E}) \vdash \mathscr{E}$: **head**):


$(\overline{\mathrm{g}}_l, \overline{\mathrm{c}}_l)$:
α_1
n
α_2
$\mathbf{l}_{\mathrm{s}}(\mathscr{E}_1)$
v
α_3
$\alpha_1 \neq 0$
$(\overline{\mathrm{g}}_r, \overline{\mathrm{c}}_r)$:
α_1
$\mathbf{l}_{\mathrm{s}}(\mathscr{E}_0)$
$\alpha_1 \neq 0$

According to Figure 6.3, we can apply the **i-uf** rule: unfolding the inductive predicate $\alpha_1 \cdot \mathbf{l}_{\mathrm{s}}(\mathscr{E}_0)$ of $\overline{\mathrm{g}}_r$ which produces an identical shape with $\overline{\mathrm{g}}_l$ and a set predicate $\mathscr{E}_0 = \{\alpha_1\} \uplus \mathscr{E}_1$ and proving that $\overline{\mathrm{c}}_l$ implies the set predicate by $\overline{\mathbf{prove}}_{\overline{\mathcal{C}}}(\overline{\mathrm{c}}_l, \mathscr{E}_0 = \{\alpha_1\} \uplus \mathscr{E}_1)$, which apparently returns **false**. Therefore, the inclusion proof fails. However, the set of concrete memory states abstracted by $(\overline{\mathrm{g}}_l, \overline{\mathrm{c}}_l)$ is indeed included in the set of concrete memory states abstracted by $(\overline{\mathrm{g}}_r, \overline{\mathrm{c}}_r)$. Though inclusion checking is typically incomplete, we may hope that basic cases still get proved.

Extended inclusion checking algorithm. We extend the original inclusion checking algorithm as presented in Figure 6.4, where we let $\mathrm{c} \in \mathcal{C}$ be the conjunction of numerical and set predicates, $\mathrm{c}_{\lceil \overline{\mathcal{V}}}$ be the restriction of c to numerical predicates, and $\mathrm{c}_{\lceil \mathcal{T}}$ be the restriction of c to set predicates. The major difference of the extended system is to collect set predicates

$$\frac{}{\overline{\mathrm{c}}_l, \mathbf{true} \vdash \alpha \cdot \mathtt{f} \mapsto \beta \sqsubseteq_{\overline{\mathcal{G}}} \alpha \cdot \mathtt{f} \mapsto \beta} \qquad (\mathbf{ei-pt})$$

$$\frac{}{\overline{\mathrm{c}}_l, \mathbf{true} \vdash \alpha \cdot \mathbf{ind}(\mathscr{E}) \sqsubseteq_{\overline{\mathcal{G}}} \alpha \cdot \mathbf{ind}(\mathscr{E})} \qquad (\mathbf{ei-ind})$$

$$\frac{\overline{\mathrm{c}}_l, \mathrm{c} \vdash \overline{\mathrm{g}}_l \sqsubseteq_{\overline{\mathcal{G}}} \overline{\mathrm{g}}_r \qquad \overline{\mathrm{c}}_l, \mathrm{c}' \vdash \overline{\mathrm{g}}'_l \sqsubseteq_{\overline{\mathcal{G}}} \overline{\mathrm{g}}'_r}{\overline{\mathrm{c}}_l, \mathrm{c} \wedge \mathrm{c}' \vdash \overline{\mathrm{g}}_l * \overline{\mathrm{g}}'_l \sqsubseteq_{\overline{\mathcal{G}}} \overline{\mathrm{g}}_r * \overline{\mathrm{g}}'_r} \qquad (\mathbf{ei-sep})$$

$$\frac{\overline{\mathrm{g}}_r \leadsto_{\mathrm{U}} (\overline{\mathrm{g}}_u, F_{\mathrm{Pure}}) \quad \overline{\mathrm{c}}_l, \mathrm{c} \vdash \overline{\mathrm{g}}_l \sqsubseteq_{\overline{\mathcal{G}}} \overline{\mathrm{g}}_u \quad \overline{\mathbf{prove}}_{\overline{\mathcal{C}}}(\overline{\mathrm{c}}_l, F_{\mathrm{Pure}\lceil\overline{\mathcal{V}}}) = \mathbf{true}}{\overline{\mathrm{c}}_l, \mathrm{c} \wedge F_{\mathrm{Pure}\lceil\mathcal{T}} \vdash \overline{\mathrm{g}}_l \sqsubseteq_{\overline{\mathcal{G}}} \overline{\mathrm{g}}_r} \qquad (\mathbf{ei-uf})$$

$$\frac{\overline{\mathrm{c}}_l, \mathrm{c} \vdash \overline{\mathrm{g}}_l \sqsubseteq_{\overline{\mathcal{G}}} \beta \cdot \mathbf{ind}(\mathscr{E}) \qquad \alpha \cdot \mathbf{ind}(\mathscr{E}) \vdash \mathscr{E} : \mathbf{cst}}{\overline{\mathrm{c}}_l, \mathrm{c} \vdash \alpha \cdot \mathbf{ind} *\!=_{(\mathscr{E})} \beta \cdot \mathbf{ind} * \overline{\mathrm{g}}_l \sqsubseteq_{\overline{\mathcal{G}}} \alpha \cdot \mathbf{ind}(\mathscr{E})} \qquad (\mathbf{ei-segind_{cst}})$$

$$\frac{\mathscr{E}_1 \text{ fresh} \quad \overline{\mathrm{c}}_l, \mathrm{c} \vdash \overline{\mathrm{g}}_l \sqsubseteq_{\overline{\mathcal{G}}} \beta \cdot \mathbf{ind}(\mathscr{E}_1) \qquad \alpha \cdot \mathbf{ind}(\mathscr{E}) \vdash \mathscr{E} : \mathbf{head}}{\overline{\mathrm{c}}_l, \mathrm{c} \wedge \mathscr{E} = \mathscr{E}_0 \uplus \mathscr{E}_1 \vdash \alpha \cdot \mathbf{ind} *\!=_{(\mathscr{E}_0)} \beta \cdot \mathbf{ind} * \overline{\mathrm{g}}_l \sqsubseteq_{\overline{\mathcal{G}}} \alpha \cdot \mathbf{ind}(\mathscr{E})} \qquad (\mathbf{ei-segind_{head}})$$

$$\frac{\mathscr{E} \notin \mathrm{dom}(\overline{\mathrm{c}}) \quad \forall 0 \leq i, j \leq k, 0 \leq p, q \leq l, (\overline{\mathrm{g}}_l, \overline{\mathrm{c}}) \implies \alpha_i \neq \alpha_j \wedge \alpha_i \notin \mathscr{E}_p \wedge \mathscr{E}_p \cap \mathscr{E}_q = \emptyset}{\overline{\mathrm{c}} \overset{i}{\leadsto} \overline{\mathbf{guard}}_{\overline{\mathcal{C}}}(\overline{\mathrm{c}}, \mathscr{E} = \{\alpha_0, \ldots, \alpha_k\} \uplus \mathscr{E}_0 \uplus \ldots \uplus \mathscr{E}_l)} \qquad (\mathbf{ei-inst})$$

$$\frac{\overline{\mathrm{c}}_l, \mathrm{c} \vdash \overline{\mathrm{g}}_l \sqsubseteq_{\overline{\mathcal{G}}} \overline{\mathrm{g}}_r \quad \overline{\mathrm{c}}_l \overset{i}{\leadsto}^{*} \overline{\mathrm{c}}'_l \quad \overline{\mathbf{prove}}_{\overline{\mathcal{C}}}(\overline{\mathrm{c}}'_l, \mathrm{c}) = \mathbf{true} \quad \overline{\mathrm{c}}'_l \sqsubseteq_{\overline{\mathcal{C}}} \overline{\mathrm{c}}_r}{(\overline{\mathrm{g}}_l, \overline{\mathrm{c}}_l) \sqsubseteq_{\overline{\mathcal{M}_\omega}} (\overline{\mathrm{g}}_r, \overline{\mathrm{c}}_r)} \qquad (\mathbf{ei-val})$$

Figure 6.4: Extended inclusion checking over shapes and abstract states $\overline{\mathcal{G}} \times \overline{\mathcal{C}}$

c that need to be proved during the inclusion checking process over shapes $\overline{\mathrm{g}}_l$ and $\overline{\mathrm{g}}_r$, i.e., $\overline{\mathrm{c}}_l, \mathrm{c} \vdash \overline{\mathrm{g}}_l \sqsubseteq_{\overline{\mathcal{G}}} \overline{\mathrm{g}}_r$, and then, compute $\overline{\mathrm{c}}'_l$ by *instantiation* of the value of fresh set variables of $\overline{\mathrm{c}}_l$, denoted as $\overline{\mathrm{c}}_l \overset{i}{\leadsto}^{*} \overline{\mathrm{c}}'_l$, such that $\overline{\mathbf{prove}}_{\overline{\mathcal{C}}}(\overline{\mathrm{c}}'_l, \mathrm{c}) = \mathbf{true}$.

Inclusion rules for shapes. The **ei-pt** and **ei-ind** rules respectively correspond to the **i-pt** and **i-ind** rules of the original system, which state that any shape is included in itself and for simplicity is omitted. The **ei-sep** and **ei-uf** rules respectively correspond to the **i-sep** and **i-uf** rules of the original system. The only difference is that the **ei-uf** rule only proves unfolded numerical predicates $F_{\mathrm{Pure}\lceil\overline{\mathcal{V}}}$ while preserving the unfolded set predicates $F_{\mathrm{Pure}\lceil\mathcal{T}}$ in the proof condition. The $\mathbf{ei-segind_{cst}}$ and $\mathbf{ei-segind_{head}}$ rules correspond respectively to the **i-segind** rule for the cases of a *constant* set parameter and a *head* set parameter when matching segments and inductive predicates. The $\mathbf{ei-segind_{cst}}$ rule simply requires inductive and segment to share the same set parameter while the $\mathbf{ei-segind_{head}}$ rule enforces the additiveness of head parameters by choosing fresh $\mathscr{E}_1$ so that $\mathscr{E} = \mathscr{E}_0 \uplus \mathscr{E}_1$.

Instantiation of fresh set variables. The **ei-inst** rule instantiates a fresh set variable $\mathscr{E}$ by enforcing an equality constraint between the fresh set variable and a set expression $\{\alpha_0, \ldots, \alpha_k\} \uplus \mathscr{E}_0 \uplus \ldots \uplus \mathscr{E}_l$ of non-fresh set variables. We note that a disjoint union expression $\mathrm{e}_{\mathcal{T}} \uplus \mathrm{e}'_{\mathcal{T}}$ has a

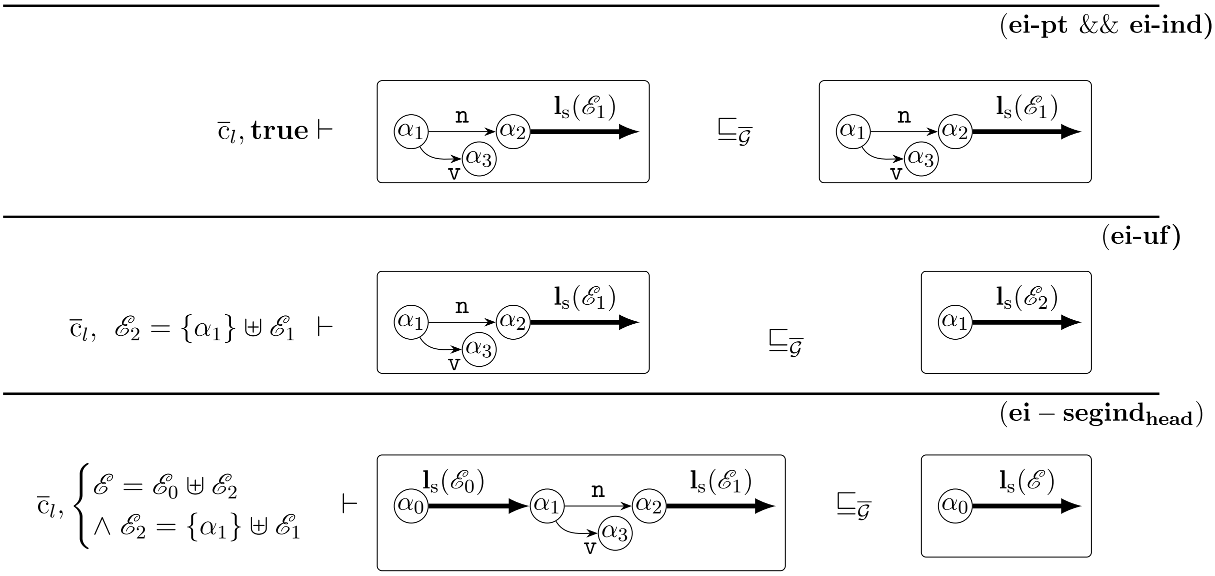

(ei-pt && ei-ind)
$\overline{c}_l$, true ⊢
α_1
n
α_2
$\mathbf{l}_s(\mathscr{E}_1)$
v
α_3
$\sqsubseteq_{\overline{\mathcal{G}}}$
(ei-uf)
$\overline{c}_l,\ \mathscr{E}_2 = \{\alpha_1\} \uplus \mathscr{E}_1\ \vdash$
$\mathbf{l}_s(\mathscr{E}_2)$
(ei − segind$_{\mathbf{head}}$)
$\overline{c}_l, \begin{cases} \mathscr{E} = \mathscr{E}_0 \uplus \mathscr{E}_2 \\ \wedge\ \mathscr{E}_2 = \{\alpha_1\} \uplus \mathscr{E}_1 \end{cases}$ ⊢
α_0
$\mathbf{l}_s(\mathscr{E}_0)$
$\mathbf{l}_s(\mathscr{E})$

Figure 6.5: An example of inclusion checking over shapes

side constraint $e_{\mathcal{T}} \cap e'_{\mathcal{T}} = \emptyset$ that needs to be proved for soundness. Specifically, the disjointness can either be proved by the value-set abstraction, i.e., $\overline{\mathbf{prove}}_{\overline{\mathcal{C}}}(\overline{c}_l, e_{\mathcal{T}} \cap e'_{\mathcal{T}} = \emptyset) = \mathbf{true}$, or be proved based on the property of the separating conjunction $*$ and head parameters:

- $\alpha_1 \cdot \mathtt{f} \mapsto \beta_1 * \alpha_2 \cdot \mathtt{f} \mapsto \beta_2 * \overline{g} \implies \alpha_1 \neq \alpha_2$.
- in the case of $\alpha \cdot \mathbf{ind}(\mathscr{E}) \vdash \mathscr{E} : \mathbf{head}$, $\alpha_1 \cdot \mathbf{ind}(\mathscr{E}_1) * \alpha_2 \cdot \mathbf{ind}(\mathscr{E}_2) * \overline{g} \implies \mathscr{E}_1 \cap \mathscr{E}_2 = \emptyset$.
- in the case of $\alpha \cdot \mathbf{ind}(\mathscr{E}) \vdash \mathscr{E} : \mathbf{head}$, $\alpha_1 \cdot \mathbf{ind}(\mathscr{E}_1) * \alpha_2 \cdot \mathtt{f} \mapsto \beta_2 * \overline{g} \implies \alpha_2 \notin \mathscr{E}_1$ if the inductive definition $\alpha \cdot \mathbf{ind}(\mathscr{E})$ contains a rule $r = (F_{\text{Heap}}, F_{\text{Pure}})$, where $F_{\text{Heap}} = \alpha \cdot \mathtt{f} \mapsto \alpha' * \ldots$ and $F_{\text{Pure}} = \mathscr{E} = \{\alpha\} \uplus \ldots \wedge \ldots$.

We note that this rule is very important for folding back unfolded predicates into summary predicates with set parameters.

Finally, the **ei-val** rule corresponding to the **i-val** rule decomposes the inclusion checking of abstract states into the inclusion checking over abstract shapes, the instantiation of fresh set variables, and the implication process over value-set abstractions.

Example 6.2 (Inclusion checking). Figure 6.5 illustrates the inclusion checking algorithm on abstract states $(\overline{g}_l, \overline{c}_l)$ and $(\overline{g}_r, \overline{c}_r)$, where $\overline{g}_l$ and $\overline{g}_r$ are the bottom shapes shown in Figure 6.5, and $\overline{c}_l$ and $\overline{c}_r$ are respectively an abstraction of $\alpha_1 \neq 0$ and **true**. We recall that the $\mathbf{l}_s$ definition (presented in Section 4.1) has a head parameter ($\alpha \cdot \mathbf{l}_s(\mathscr{E}) \vdash \mathscr{E} : \mathbf{head}$), which behaves similarly to **nodes** (Figure 3.3) in its second parameter. The example resembles inclusion tests run in the analysis of the program shown in Figure 3.5.

The inclusion proof search over shapes starts from the bottom shapes $\overline{g}_l$ and $\overline{g}_r$, where α_0

is the origin of a segment on the left, and of an inductive predicate on the right. Since $\mathbf{l}_\mathrm{s}$ has a *head* parameter, the $\mathbf{ei} - \mathbf{segind}_{\mathbf{head}}$ rule specific to this case applies, and the inclusion test "consumes" the segment, effectively removing it from the left argument, and adding a fresh $\mathscr{E}_2$ variable, such that $\mathscr{E} = \mathscr{E}_0 \uplus \mathscr{E}_2$ is added in the proof condition. Then, since α_1 is the origin of points-to predicates on the left and of inductive predicate on the right, the **ei-uf** rule is applied to unfold the inductive predicate, where the constraint $\mathscr{E}_2 = \{\alpha_1\} \uplus \mathscr{E}_1$ is added into the proof condition after proving that $\overline{\mathbf{prove}}_{\overline{\mathcal{C}}}(\bar{\mathrm{c}}_l, \alpha_1 \neq 0) = \mathbf{true}$. Finally, the algorithm derives that inclusion holds on the shapes after matching three pairs of identical predicates.

Then, the algorithm needs to prove that the condition $\mathscr{E} = \mathscr{E}_0 \uplus \mathscr{E}_2 \wedge \mathscr{E}_2 = \{\alpha_1\} \uplus \mathscr{E}_1$ is implied by $\bar{\mathrm{c}}_l$ ($\mathrm{dom}(\bar{\mathrm{c}}_l) = \mathrm{dom}(\bar{\mathrm{g}}_l)$). Since set variables $\mathscr{E}$ and $\mathscr{E}_2$ do not exist in $\bar{\mathrm{c}}_l$, the instantiation of $\mathscr{E}$ and $\mathscr{E}_2$ is necessary. According to the **ei-inst** rule, we can first instantiate $\mathscr{E}_2$ to $\{\alpha_1\} \uplus \mathscr{E}_1$, i.e., $\bar{\mathrm{c}} \overset{i}{\rightsquigarrow} \overline{\mathbf{guard}}_{\overline{\mathcal{C}}}(\bar{\mathrm{c}}_l, \mathscr{E}_2 = \{\alpha_1\} \uplus \mathscr{E}_1)$. We note that, the side condition $\{\alpha_1\} \cap \mathscr{E}_1 = \emptyset$ is implied by the abstract shape $\bar{\mathrm{g}}_l$, in which α_1 is the origin of points-to predicates and $\mathscr{E}_1$ is the head parameter of inductive predicate $\alpha_2 \cdot \mathbf{l}_\mathrm{s}(\mathscr{E}_1)$. Then, similarly we can instantiate $\mathscr{E}$ to $\{\alpha_1\} \uplus \mathscr{E}_0 \uplus \mathscr{E}_1$, thus we have:

$$\bar{\mathrm{c}}_l' = \overline{\mathbf{guard}}_{\overline{\mathcal{C}}}(\overline{\mathbf{guard}}_{\overline{\mathcal{C}}}(\bar{\mathrm{c}}_l, \mathscr{E}_2 = \{\alpha_1\} \uplus \mathscr{E}_1), \mathscr{E} = \{\alpha_1\} \uplus \mathscr{E}_0 \uplus \mathscr{E}_1) \textit{ and } \bar{\mathrm{c}}_l \overset{i}{\rightsquigarrow} \bar{\mathrm{c}}_l'$$

Indeed, the instantiation of set variables $\mathscr{E}$ and $\mathscr{E}_2$ is constructed according to the proof condition c in the bottom rule. Intuitively, when proving c requires an equality constraint between a fresh set variable $\mathscr{E}$ and a set expression $\mathrm{e}_{\mathcal{T}}$ of non-fresh set variables, we instantiate $\mathscr{E}$ to $\mathrm{e}_{\mathcal{T}}$.

Finally, we have $(\bar{\mathrm{g}}_l, \bar{\mathrm{c}}_l) \sqsubseteq_{\overline{\mathcal{M}_\omega}} (\bar{\mathrm{g}}_r, \bar{\mathrm{c}}_r)$ after verifying that $\overline{\mathbf{prove}}_{\overline{\mathcal{C}}}(\bar{\mathrm{c}}_l', \mathrm{c}) = \mathbf{true}$ and $\bar{\mathrm{c}}_l' \sqsubseteq_{\overline{\mathcal{C}}} \bar{\mathrm{c}}_r$.

Theorem 6.3 (Soundness condition of inclusion checking over shapes). *If*

$$\bar{\mathrm{c}}_l, \mathrm{c} \vdash \bar{\mathrm{g}}_l \sqsubseteq_{\overline{\mathcal{G}}} \bar{\mathrm{g}}_r$$

then:

$$\forall (\sigma, \mu) \vDash (\bar{\mathrm{g}}_l, \bar{\mathrm{c}}_l) \qquad \exists \mu' \textit{ extending } \mu, \mu' \vDash \mathrm{c} \implies (\sigma, \mu') \vDash \bar{\mathrm{g}}_r$$

Proof. The proof can be derived by induction over inclusion rules of shapes.

- For the **ei-pt** and **ei-ind** rules, it obviously holds.
- For the **ei-sep** rule, for all $\mu \vDash \bar{\mathrm{c}}_l$, $(\sigma, \mu) \vDash \bar{\mathrm{g}}_l$ and $(\sigma', \mu) \vDash \bar{\mathrm{g}}_l'$, according to the condition, $\exists \mu'$ *extending* μ, $\mu' \vDash \mathrm{c} \implies (\sigma, \mu') \vDash \bar{\mathrm{g}}_r$ and $\mu' \vDash \mathrm{c}' \implies (\sigma', \mu') \vDash \bar{\mathrm{g}}_r'$. Thus, we have $\mu' \vDash \mathrm{c} \wedge \mathrm{c}' \implies (\sigma \uplus \sigma', \mu') \vDash \bar{\mathrm{g}}_r * \bar{\mathrm{g}}_r'$.
- For the **ei-uf** rule, for all $\mu \vDash \bar{\mathrm{c}}_l$ and $(\sigma, \mu) \vDash \bar{\mathrm{g}}_l$, according to the condition $\bar{\mathrm{c}}_l, \mathrm{c} \vdash \bar{\mathrm{g}}_l \sqsubseteq_{\overline{\mathcal{G}}} \bar{\mathrm{g}}_u$, $\exists \mu'$ *extending* μ, $\mu' \vDash \mathrm{c} \implies (\sigma, \mu') \vDash \bar{\mathrm{g}}_u$. According to $\overline{\mathbf{prove}}_{\overline{\mathcal{C}}}(\bar{\mathrm{c}}_l, F_{\mathrm{Pure}\lceil\overline{\mathcal{V}}}) = \mathbf{true}$, $\forall \mu'$ *extending* $\mu, \mu' \vDash F_{\mathrm{Pure}\lceil\overline{\mathcal{V}}}$. As $\bar{\mathrm{g}}_r \rightsquigarrow_{\mathrm{U}} (\bar{\mathrm{g}}_u, F_{\mathrm{Pure}})$ and $F_{\mathrm{Pure}} = F_{\mathrm{Pure}\lceil\overline{\mathcal{V}}} \wedge F_{\mathrm{Pure}\lceil\mathcal{T}}$, according to the concretization rule in Figure 4.3, we have $\exists \mu'$ *extending* μ, $\mu' \vDash \mathrm{c} \wedge F_{\mathrm{Pure}\lceil\mathcal{T}} \implies \mu' \vDash F_{\mathrm{Pure}} \wedge (\sigma, \mu') \vDash \bar{\mathrm{g}}_u \implies (\sigma, \mu') \vDash \bar{\mathrm{g}}_r$.

- For the $\mathbf{ei} - \mathbf{segind_{cst}}$ rule, for all $\mu \models \overline{\mathrm{c}}_l$ and $(\sigma, \mu) \models \alpha \cdot \mathbf{ind} \ast\!\!=_{(\mathscr{E})} \beta \cdot \mathbf{ind}$ and $(\sigma', \mu) \models \overline{\mathrm{g}}_l$, according to the condition $\overline{\mathrm{c}}_l, \mathrm{c} \vdash \overline{\mathrm{g}}_l \sqsubseteq_{\overline{\mathcal{G}}} \beta \cdot \mathbf{ind}(\mathscr{E})$, $\exists \mu'$ *extending* μ, $\mu' \models \mathrm{c} \implies (\sigma', \mu') \models \beta \cdot \mathbf{ind}(\mathscr{E})$. Thus, $\exists \mu'$ *extending* μ, $\mu' \models \mathrm{c} \implies (\sigma \uplus \sigma', \mu') \models \alpha \cdot \mathbf{ind} \ast\!\!=_{(\mathscr{E})} \beta \cdot \mathbf{ind} \ast \beta \cdot \mathbf{ind}(\mathscr{E})$. According to Theorem 4.1, $\exists \mu'$ *extending* μ, $\mu' \models \mathrm{c} \implies (\sigma \uplus \sigma', \mu') \models \alpha \cdot \mathbf{ind}(\mathscr{E})$.

- For the $\mathbf{ei} - \mathbf{segind_{head}}$ rule, for all $\mu \models \overline{\mathrm{c}}_l$ and $(\sigma, \mu) \models \alpha \cdot \mathbf{ind} \ast\!\!=_{(\mathscr{E}_0)} \beta \cdot \mathbf{ind}$ and $(\sigma', \mu) \models \overline{\mathrm{g}}_l$, according to the condition $\overline{\mathrm{c}}_l, \mathrm{c} \vdash \overline{\mathrm{g}}_l \sqsubseteq_{\overline{\mathcal{G}}} \beta \cdot \mathbf{ind}(\mathscr{E}_1)$, $\exists \mu'$ *extending* μ, $\mu' \models \mathrm{c} \implies (\sigma', \mu') \models \beta \cdot \mathbf{ind}(\mathscr{E}_1)$. Thus, $\exists \mu'$ *extending* μ, $\mu' \models \mathrm{c} \implies (\sigma \uplus \sigma', \mu') \models \alpha \cdot \mathbf{ind} \ast\!\!=_{(\mathscr{E}_0)} \beta \cdot \mathbf{ind} \ast \beta \cdot \mathbf{ind}(\mathscr{E}_1)$. According to Theorem 4.2, $\exists \mu'$ *extending* μ, $\mu' \models \mathrm{c} \land \mathscr{E} = \mathscr{E}_0 \uplus \mathscr{E}_1 \implies (\sigma \uplus \sigma', \mu') \models \alpha \cdot \mathbf{ind}(\mathscr{E})$.

■

Theorem 6.4 (Soundness condition of instantiation). *Given* $(\overline{\mathrm{g}}, \overline{\mathrm{c}})$*, if* $\overline{\mathrm{c}} \overset{i}{\rightsquigarrow} \overline{\mathrm{c}}'$ *then:*

$$\forall (\sigma, \mu) \models (\overline{\mathrm{g}}, \overline{\mathrm{c}}), \exists \mu' \textit{ extending } \mu, (\sigma, \mu') \models (\overline{\mathrm{g}}, \overline{\mathrm{c}}')$$

Proof. For all $(\sigma, \mu) \models (\overline{\mathrm{g}}, \overline{\mathrm{c}})$, $\forall 0 \leq i, j \leq k, 0 \leq p, q \leq l$, $\mu(\alpha_i) \neq \mu(\alpha_j) \land \mu(\alpha_i) \notin \mu(\mathscr{E}_p) \land \mu(\mathscr{E}_p) \cap \mu(\mathscr{E}_q) = \emptyset$. Thus, let $\mu' = \mu \uplus [\mathscr{E} \mapsto \{\mu(\alpha_0), \ldots, \mu(\alpha_k)\} \uplus \mu(\mathscr{E}_0) \uplus \ldots \uplus \mu(\mathscr{E}_l)]$, we have $(\sigma, \mu') \models (\overline{\mathrm{g}}, \overline{\mathbf{guard}}_{\overline{\mathcal{C}}}(\overline{\mathrm{c}}, \mathscr{E} = \{\alpha_0, \ldots, \alpha_k\} \uplus \mathscr{E}_0 \uplus \ldots \uplus \mathscr{E}_l))$. ■

Theorem 6.5 (Soundness condition of inclusion checking). *Given* $(\overline{\mathrm{g}}_l, \overline{\mathrm{c}}_l)$ *and* $(\overline{\mathrm{g}}_r, \overline{\mathrm{c}}_r)$*, if*

$$(\overline{\mathrm{g}}_l, \overline{\mathrm{c}}_l) \sqsubseteq_{\overline{\mathcal{M}_\omega}} (\overline{\mathrm{g}}_r, \overline{\mathrm{c}}_r)$$

then

$$\gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{g}}_l, \overline{\mathrm{c}}_l) \subseteq \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{g}}_r, \overline{\mathrm{c}}_r)$$

Proof. According to Theorem 6.4 and $\overline{\mathrm{c}}_l \overset{i}{\rightsquigarrow}{}^{*} \overline{\mathrm{c}}'_l$, we have $\forall (\sigma, \mu) \models (\overline{\mathrm{g}}_l, \overline{\mathrm{c}}), \exists \mu'$ *extending* $\mu, (\sigma, \mu') \models (\overline{\mathrm{g}}_l, \overline{\mathrm{c}}'_l)$. As $\overline{\mathbf{prove}}_{\overline{\mathcal{C}}}(\overline{\mathrm{c}}'_l, \mathrm{c}) = \mathbf{true}$, $\overline{\mathrm{c}}'_l \sqsubseteq_{\overline{\mathcal{C}}} \overline{\mathrm{c}}_r$, and $\overline{\mathrm{c}}_l, \mathrm{c} \vdash \overline{\mathrm{g}}_l \sqsubseteq_{\overline{\mathcal{G}}} \overline{\mathrm{g}}_r$, according to Theorem 6.3, we get $\forall (\sigma, \mu) \models (\overline{\mathrm{g}}_l, \overline{\mathrm{c}}), \exists \mu'$ *extending* $\mu, \mu' \models \mathrm{c} \land \mu' \models \overline{\mathrm{c}}_r \land (\sigma, \mu') \models \overline{\mathrm{g}}_r$.

Therefore, we get $\forall (\sigma, \mu) \models (\overline{\mathrm{g}}_l, \overline{\mathrm{c}}), \exists \mu'$ *extending* $\mu, (\sigma, \mu') \models (\overline{\mathrm{g}}_r, \overline{\mathrm{c}}_r)$. ■

6.3.2 Joining and Widening

As introduced in Section 2.4.4, the joining and widening abstract operations $\sqcup_{\overline{\mathcal{M}_\omega}}$ and $\nabla_{\overline{\mathcal{M}_\omega}}$ input two abstract memory states $\overline{\mathrm{m}}_l$ and $\overline{\mathrm{m}}_r$ and return an over-approximation of them $\overline{\mathrm{m}}_o$, that is $\gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}_l) \subseteq \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}_o)$ and $\gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}_r) \subseteq \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}_o)$. In addition to that, the widening operator will enforce termination, that is, any sequence of iterations of the widening operator will eventually become stationary. A basic version of the joining and widening algorithm is formalized in [CR08], where in the shape domain $\overline{\mathcal{G}}$, joining and widening rely on the same algorithm. Here we extend the algorithm so as to handle set parameters.

$$\frac{}{\overline{\mathrm{n}}_l, \overline{\mathrm{n}}_r \vdash \alpha \cdot \mathtt{f} \mapsto \beta \sqcup_{\overline{\mathcal{G}}} \alpha \cdot \mathtt{f} \mapsto \beta = \alpha \cdot \mathtt{f} \mapsto \beta} \qquad (\mathbf{j} - \mathbf{pt})$$

$$\frac{}{\overline{\mathrm{n}}_l, \overline{\mathrm{n}}_r \vdash \alpha \cdot \mathbf{ind} \sqcup_{\overline{\mathcal{G}}} \alpha \cdot \mathbf{ind} = \alpha \cdot \mathbf{ind}} \qquad (\mathbf{j} - \mathbf{ind})$$

$$\frac{\overline{\mathrm{n}}_r \vdash \overline{\mathrm{g}}_r \sqsubseteq_{\overline{\mathcal{G}}} \alpha \cdot \mathbf{ind}}{\overline{\mathrm{n}}_l, \overline{\mathrm{n}}_r \vdash \alpha \cdot \mathbf{ind} \sqcup_{\overline{\mathcal{G}}} \overline{\mathrm{g}}_r = \alpha \cdot \mathbf{ind}} \qquad (\mathbf{j} - \mathbf{weak})$$

$$\frac{\overline{\mathrm{n}}_l \vdash \overline{\mathrm{g}}_l \sqsubseteq_{\overline{\mathcal{G}}} \alpha \cdot \mathbf{ind} \mathrel{*\!\!=} \beta \cdot \mathbf{ind} \qquad \overline{\mathbf{prove}}_{\overline{\mathcal{N}}}(\overline{\mathrm{n}}_r, \alpha = \beta) = \mathbf{true}}{\overline{\mathrm{n}}_l, \overline{\mathrm{n}}_r \vdash \overline{\mathrm{g}}_l \sqcup_{\overline{\mathcal{G}}} \mathbf{emp} = \alpha \cdot \mathbf{ind} \mathrel{*\!\!=} \beta \cdot \mathbf{ind}} \qquad (\mathbf{j} - \mathbf{intro})$$

$$\frac{\overline{\mathrm{n}}_l, \overline{\mathrm{n}}_r \vdash \overline{\mathrm{g}}_l \sqcup_{\overline{\mathcal{G}}} \overline{\mathrm{g}}_r = \overline{\mathrm{g}}_o \quad \overline{\mathrm{n}}_l, \overline{\mathrm{n}}_r \vdash \overline{\mathrm{g}}'_l \sqcup_{\overline{\mathcal{G}}} \overline{\mathrm{g}}'_r = \overline{\mathrm{g}}'_o}{\overline{\mathrm{n}}_l, \overline{\mathrm{n}}_r \vdash \overline{\mathrm{g}}_l * \overline{\mathrm{g}}'_l \sqcup_{\overline{\mathcal{G}}} \overline{\mathrm{g}}_r * \overline{\mathrm{g}}'_r = \overline{\mathrm{g}}_o * \overline{\mathrm{g}}'_o} \qquad (\mathbf{j} - \mathbf{sep})$$

$$\frac{\overline{\mathrm{n}}_l, \overline{\mathrm{n}}_r \vdash \overline{\mathrm{g}}_l \sqcup_{\overline{\mathcal{G}}} \overline{\mathrm{g}}_r = \overline{\mathrm{g}}_o \quad \overline{\mathrm{n}}_o = \overline{\mathrm{n}}_l \sqcup_{\overline{\mathcal{N}}} \overline{\mathrm{n}}_r}{(\overline{\mathrm{g}}_l, \overline{\mathrm{n}}_l) \sqcup_{\overline{\mathcal{M}_\omega}} (\overline{\mathrm{g}}_r, \overline{\mathrm{n}}_r) = (\overline{\mathrm{g}}_o, \overline{\mathrm{n}}_o)} \qquad (\mathbf{j} - \mathbf{val})$$

Figure 6.6: Original joining over shapes of abstract states $\overline{\mathcal{G}} \times \overline{\mathcal{N}}$

Original joining algorithm. The original joining process designed for abstract states $\overline{\mathcal{G}} \times \overline{\mathcal{N}}$ is formalized in Figure 6.6, and consists of a joining process over abstract shapes and a joining process over numerical abstractions, as shown by the **j-val** rule. The principle of the joining algorithm over abstract shapes is to select pairs of regions in both input shapes and compute an over-approximation for each pair of regions relying on the **j-sep** rule. Specifically, based on the **j-pt** and **j-ind** rules, when both input regions contain syntactically equal atomic predicates, the join then simply returns the same shape as an over-approximation. When one input region contains a summary predicate, the **j-weak** rule attempts to weaken the other region into the summary predicate with the inclusion checking algorithm. In addition, the **j-intro** rule synthesizes a segment between α and β when either input contains a region subsumed by the segment, and the other input implies $\alpha = \beta$ which corresponds to an empty segment.

Extended joining algorithm. Figure 6.7 shows the extended joining algorithm of abstract states $\overline{\mathcal{G}} \times \overline{\mathcal{C}}$ that is able to handle set parameters of summary predicates. Similarly to the join system shown in Figure 6.6, the extended join algorithm over shapes first selects regions of both inputs that can be over-approximated based on the **ej-sep** rule and then relies on two generic sets of rules to join each region. The first set of rules states that, if both inputs contain a same region $\overline{\mathrm{g}}$, then these regions can be joined immediately. It is applied for points-to and inductive predicates as shown by the **ej-pt** and **ej-ind** rules, and segment predicates, for which the rule is omitted for simplicity. The second set of rules applies to cases where the input regions are different and thus weakening is necessary: if a common over-approximation $\overline{\mathrm{g}}_o$ for $\overline{\mathrm{g}}_l$ and $\overline{\mathrm{g}}_r$ can be found and checked with $\sqsubseteq_{\overline{\mathcal{G}}}$, abstract join can rewrite these into $\overline{\mathrm{g}}_o$. We note that this set of rules may *extend* the set predicates, as fresh set variables are introduced by the inclusion check algorithm. Nevertheless, this process is non-trivial, since the joining algorithm needs to *infer* relations over the new set variables, from the results of the inclusion checking. Specifically, the joining algorithm attempts to perform the weakening in the following cases:

$$\dfrac{}{(\overline{\mathrm{c}}_l, \mathbf{true}), (\overline{\mathrm{c}}_r, \mathbf{true}) \vdash \alpha \cdot \mathtt{f} \mapsto \beta \sqcup_{\overline{\mathcal{G}}} \alpha \cdot \mathtt{f} \mapsto \beta = \alpha \cdot \mathtt{f} \mapsto \beta} \quad (\mathbf{ej-pt})$$

$$\dfrac{}{(\overline{\mathrm{c}}_l, \mathscr{E} = \mathscr{E}_1), (\overline{\mathrm{c}}_r, \mathscr{E} = \mathscr{E}_2) \vdash \alpha \cdot \mathbf{ind}(\mathscr{E}_1) \sqcup_{\overline{\mathcal{G}}} \alpha \cdot \mathbf{ind}(\mathscr{E}_2) = \alpha \cdot \mathbf{ind}(\mathscr{E})} \quad (\mathbf{ej-ind})$$

$$\dfrac{\overline{\mathrm{c}}_r, \mathrm{c}_r \vdash \overline{\mathrm{g}}_r \sqsubseteq_{\overline{\mathcal{G}}} \alpha \cdot \mathbf{ind}(\mathscr{E})}{(\overline{\mathrm{c}}_l, \mathscr{E} = \mathscr{E}_1), (\overline{\mathrm{c}}_r, \mathrm{c}_r) \vdash \alpha \cdot \mathbf{ind}(\mathscr{E}_1) \sqcup_{\overline{\mathcal{G}}} \overline{\mathrm{g}}_r = \alpha \cdot \mathbf{ind}(\mathscr{E})} \quad (\mathbf{ej-weak})$$

$$\dfrac{\overline{\mathrm{c}}_l, \mathrm{c}_l \vdash \overline{\mathrm{g}}_l \sqsubseteq_{\overline{\mathcal{G}}} \alpha \cdot \mathbf{ind} \ast\!\!=_{(\mathscr{E})} \beta \cdot \mathbf{ind} \quad \overline{\mathrm{c}}_r \implies \alpha = \beta \quad \alpha \cdot \mathbf{ind}(\mathscr{E}) \vdash \mathscr{E} : \mathbf{head}}{(\overline{\mathrm{c}}_l, \mathrm{c}_l), (\overline{\mathrm{c}}_r, \mathscr{E} = \emptyset) \vdash \overline{\mathrm{g}}_l \sqcup_{\overline{\mathcal{G}}} \mathbf{emp} = \alpha \cdot \mathbf{ind} \ast\!\!=_{(\mathscr{E})} \beta \cdot \mathbf{ind}} \quad (\mathbf{ej-intro_{head}})$$

$$\dfrac{\overline{\mathrm{c}}_l, \mathrm{c}_l \vdash \overline{\mathrm{g}}_l \sqsubseteq_{\overline{\mathcal{G}}} \alpha \cdot \mathbf{ind} \ast\!\!=_{(\mathscr{E})} \beta \cdot \mathbf{ind} \quad \overline{\mathrm{c}}_r \implies \alpha = \beta \quad \alpha \cdot \mathbf{ind}(\mathscr{E}) \vdash \mathscr{E} : \mathbf{cst}}{(\overline{\mathrm{c}}_l, \mathrm{c}_l), (\overline{\mathrm{c}}_r, \mathbf{true}) \vdash \overline{\mathrm{g}}_l \sqcup_{\overline{\mathcal{G}}} \mathbf{emp} = \alpha \cdot \mathbf{ind} \ast\!\!=_{(\mathscr{E})} \beta \cdot \mathbf{ind}} \quad (\mathbf{ej-intro_{cst}})$$

$$\dfrac{(\overline{\mathrm{c}}_l, \mathrm{c}_l), (\overline{\mathrm{c}}_r, \mathrm{c}_r) \vdash \overline{\mathrm{g}}_l \sqcup_{\overline{\mathcal{G}}} \overline{\mathrm{g}}_r = \overline{\mathrm{g}}_o \quad (\overline{\mathrm{c}}_l, \mathrm{c}'_l), (\overline{\mathrm{c}}_r, \mathrm{c}'_r) \vdash \overline{\mathrm{g}}'_l \sqcup_{\overline{\mathcal{G}}} \overline{\mathrm{g}}'_r = \overline{\mathrm{g}}'_o}{(\overline{\mathrm{c}}_l, \mathrm{c}_l \wedge \mathrm{c}'_l), (\overline{\mathrm{c}}_r, \mathrm{c}_r \wedge \mathrm{c}'_r) \vdash \overline{\mathrm{g}}_l \ast \overline{\mathrm{g}}'_l \sqcup_{\overline{\mathcal{G}}} \overline{\mathrm{g}}_r \ast \overline{\mathrm{g}}'_r = \overline{\mathrm{g}}_o \ast \overline{\mathrm{g}}'_o} \quad (\mathbf{ej-sep})$$

$$\dfrac{(\overline{\mathrm{c}}_l, \mathrm{c}_l), (\overline{\mathrm{c}}_r, \mathrm{c}_r) \vdash \overline{\mathrm{g}}_l \sqcup_{\overline{\mathcal{G}}} \overline{\mathrm{g}}_r = \overline{\mathrm{g}}_o \quad \forall i \in \{l, r\}, \overline{\mathrm{c}}_i \overset{i}{\rightsquigarrow}^{*} \overline{\mathrm{c}}'_i \wedge \overline{\mathrm{c}}'_i \implies \mathrm{c}_i \quad \overline{\mathrm{c}}_o = \overline{\mathrm{c}}'_l \sqcup_{\overline{\mathcal{C}}} \overline{\mathrm{c}}'_r}{(\overline{\mathrm{g}}_l, \overline{\mathrm{c}}_l) \sqcup_{\overline{\mathcal{M}_\omega}} (\overline{\mathrm{g}}_r, \overline{\mathrm{c}}_r) = (\overline{\mathrm{g}}_o, \overline{\mathrm{c}}_o)} \quad (\mathbf{ej-val})$$

Figure 6.7: Extended joining over shapes and abstract states $\overline{\mathcal{G}} \times \overline{\mathcal{C}}$

- the **ej-weak** rule applies when either input contains an inductive predicate and the other input contains a region that can be subsumed by the same inductive predicate proved by inclusion checking. Constraints over set parameter $\mathscr{E}$ may be added after the inclusion checking. In particular, in the case of a head set parameter, these constraints should capture the linearity over the head set parameter. The same principle also applies to segment predicates.

- the $\mathbf{ej-intro_{head}}$ and $\mathbf{ej-intro_{cst}}$ rules apply when either input contains a region subsumed by a segment between α and β, and $\alpha = \beta$ in the other input which corresponds to an empty segment. We note that since set parameter $\mathscr{E}$ is fresh, the joining algorithm needs to rely on both the inclusion checking algorithm to discover what $\mathscr{E}$ actually represents on one side and on the property of the set parameter (head or constant) on the other side. Specifically, when $\mathbf{ind} \vdash \mathscr{E} : \mathbf{head}$, the set parameter $\mathscr{E}$ of the empty segment is enforced to be empty, i.e., $\mathscr{E} = \emptyset$, as shown in the $\mathbf{j-intro_{head}}$ rule, and in the case of $\mathbf{ind} \vdash \mathscr{E} : \mathbf{cst}$, no new constraint is added, as shown in the $\mathbf{j-intro_{cst}}$ rule.

Finally, the **ej-val** rule instantiates fresh set variables in both value-set abstract elements $\overline{\mathrm{c}}_l$ and $\overline{\mathrm{c}}_r$ based on set constraints c_l and c_r collected during the shape joining and indirectly stemming from the constant or head kind of the set parameters, i.e., $\overline{\mathrm{c}}_l \overset{i}{\rightsquigarrow}^{*} \overline{\mathrm{c}}'_l$ and $\overline{\mathrm{c}}_r \overset{i}{\rightsquigarrow}^{*} \overline{\mathrm{c}}'_r$, and then joins the enriched value-set abstract elements $\overline{\mathrm{c}}_o = \overline{\mathrm{c}}'_l \sqcup_{\overline{\mathcal{C}}} \overline{\mathrm{c}}'_r$ after proving the soundness condition $\overline{\mathrm{c}}'_l \implies \mathrm{c}_l \wedge \overline{\mathrm{c}}'_r \implies \mathrm{c}_r$.

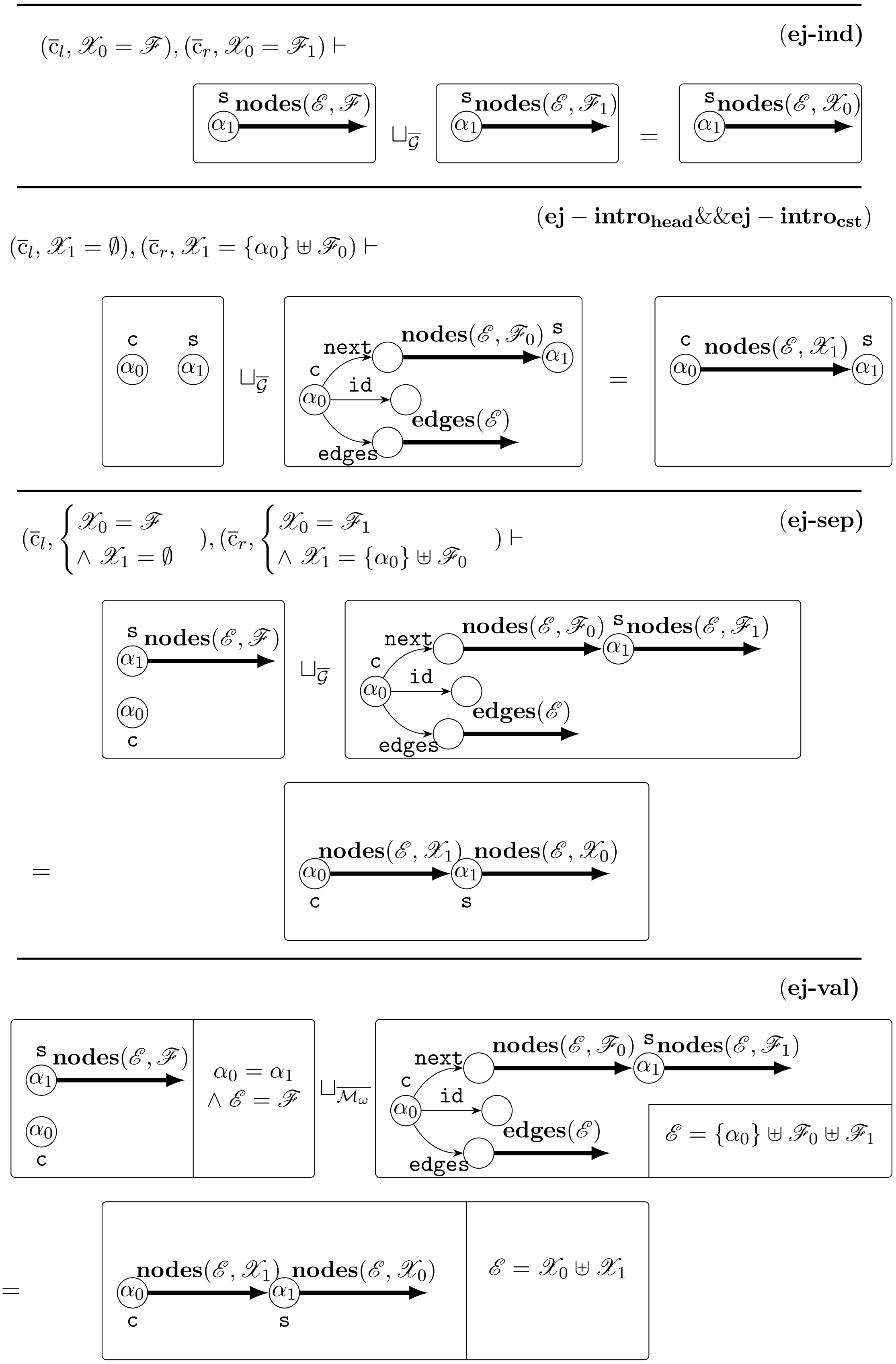

(ej-ind)
$(\bar{c}_l, \mathscr{X}_0 = \mathscr{F}), (\bar{c}_r, \mathscr{X}_0 = \mathscr{F}_1) \vdash$
s nodes($\mathscr{E}, \mathscr{F}$) α_1
$\sqcup_{\overline{\mathcal{G}}}$
s nodes($\mathscr{E}, \mathscr{F}_1$) α_1
=
s nodes($\mathscr{E}, \mathscr{X}_0$) α_1
$(\mathbf{ej} - \mathbf{intro_{head}} \&\& \mathbf{ej} - \mathbf{intro_{cst}})$
$(\bar{c}_l, \mathscr{X}_1 = \emptyset), (\bar{c}_r, \mathscr{X}_1 = \{\alpha_0\} \uplus \mathscr{F}_0) \vdash$
c α_0 s α_1
$\sqcup_{\overline{\mathcal{G}}}$
next id edges c α_0 nodes($\mathscr{E}, \mathscr{F}_0$) s α_1 edges($\mathscr{E}$)
=
c α_0 nodes($\mathscr{E}, \mathscr{X}_1$) s α_1
(ej-sep)
$(\bar{c}_l, \begin{cases} \mathscr{X}_0 = \mathscr{F} \\ \wedge\ \mathscr{X}_1 = \emptyset \end{cases}), (\bar{c}_r, \begin{cases} \mathscr{X}_0 = \mathscr{F}_1 \\ \wedge\ \mathscr{X}_1 = \{\alpha_0\} \uplus \mathscr{F}_0 \end{cases}) \vdash$
s nodes($\mathscr{E}, \mathscr{F}$) α_1 α_0 c
$\sqcup_{\overline{\mathcal{G}}}$
next id edges c α_0 nodes($\mathscr{E}, \mathscr{F}_0$) s α_1 nodes($\mathscr{E}, \mathscr{F}_1$) edges($\mathscr{E}$)
=
nodes($\mathscr{E}, \mathscr{X}_1$) nodes($\mathscr{E}, \mathscr{X}_0$) α_0 c α_1 s
(ej-val)
s nodes($\mathscr{E}, \mathscr{F}$) α_1 α_0 c
$\alpha_0 = \alpha_1$
$\wedge\ \mathscr{E} = \mathscr{F}$
$\sqcup_{\overline{\mathcal{M}_\omega}}$
next id edges c α_0 nodes($\mathscr{E}, \mathscr{F}_0$) s α_1 nodes($\mathscr{E}, \mathscr{F}_1$) edges($\mathscr{E}$)
$\mathscr{E} = \{\alpha_0\} \uplus \mathscr{F}_0 \uplus \mathscr{F}_1$
=
nodes($\mathscr{E}, \mathscr{X}_1$) nodes($\mathscr{E}, \mathscr{X}_0$) α_0 c α_1 s
$\mathscr{E} = \mathscr{X}_0 \uplus \mathscr{X}_1$

Figure 6.8: An example of joining

Example 6.3 (Joining). Figure 6.8 shows a simplified instance of a join taken from the analysis of the program of Figure 3.5. The input abstract states and output abstract state are shown at the bottom.

Initially, both input shapes contain **nodes** inductive predicates at α_1, therefore the **ej-ind** rule can be applied to output a **nodes** inductive predicate. The first parameter is constant, and thus is equal to $\mathscr{E}$ everywhere. The second parameter is a head parameter, so a new set variable $\mathscr{X}_0$ is introduced, and c_l (resp., c_r) is enriched with constraint $\mathscr{X}_0 = \mathscr{F}$ (resp., $\mathscr{X}_0 = \mathscr{F}_1$).

Then, the $\mathbf{ej-intro_{head}}$ and $\mathbf{ej-intro_{cst}}$ rules are applied to introduce a segment between α_0 and α_1 as constraint $\alpha_0 = \alpha_1$ is indicated by the left value-set abstraction $\overline{\mathrm{c}}_l$. Again, the constant parameter is equal to $\mathscr{E}$ everywhere. In addition, set variable $\mathscr{X}_1$ is introduced as the second parameter. On the left, constraint $\mathscr{X}_1 = \emptyset$ is added. On the right, the inclusion checking discovers constraint $\mathscr{X}_1 = \{\alpha_0\} \uplus \mathscr{F}_0$.

Finally, rule **ej-sep** combines the joining result of the above rules, which leads to the final joining shape. In addition, rule **ej-val** instantiates fresh set variables $\mathscr{X}_0$ and $\mathscr{X}_1$ in $\overline{\mathrm{c}}_l$ and $\overline{\mathrm{c}}_r$ according to the set constraints collected during the shape joining and then computes the final value-set abstract element as an abstraction of $\mathscr{X}_1 = \{\alpha_0\} \uplus \mathscr{F}_0$.

Theorem 6.6 (Soundness of join). *Given* $(\overline{\mathrm{g}}_l, \overline{\mathrm{c}}_l)$ *and* $(\overline{\mathrm{g}}_r, \overline{\mathrm{c}}_r)$*, if*

$$(\overline{\mathrm{g}}_l, \overline{\mathrm{c}}_l) \sqcup_{\overline{\mathcal{M}_\omega}} (\overline{\mathrm{g}}_r, \overline{\mathrm{c}}_r) = (\overline{\mathrm{g}}_o, \overline{\mathrm{c}}_o)$$

then:

$$\gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{g}}_l, \overline{\mathrm{c}}_l) \subseteq \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{g}}_o, \overline{\mathrm{c}}_o) \wedge \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{g}}_r, \overline{\mathrm{c}}_r) \subseteq \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{g}}_o, \overline{\mathrm{c}}_o)$$

Proof. According to the **ej-val** rule, let us assume $(\overline{\mathrm{c}}_l, \mathrm{c}_l), (\overline{\mathrm{c}}_r, \mathrm{c}_r) \vdash \overline{\mathrm{g}}_l \sqcup_{\overline{\mathcal{G}}} \overline{\mathrm{g}}_r = \overline{\mathrm{g}}_o$, $\forall i \in \{l, r\}, \overline{\mathrm{c}}_i \overset{i}{\rightsquigarrow}^{*} \overline{\mathrm{c}}'_i \wedge \overline{\mathrm{c}}'_i \implies \mathrm{c}_i$ and $\overline{\mathrm{c}}_o = \overline{\mathrm{c}}'_l \sqcup_{\overline{\mathcal{C}}} \overline{\mathrm{c}}'_r$.

We first need to prove that

$$(\overline{\mathrm{c}}_l, \mathrm{c}_l), (\overline{\mathrm{c}}_r, \mathrm{c}_r) \vdash \overline{\mathrm{g}}_l \sqcup_{\overline{\mathcal{G}}} \overline{\mathrm{g}}_r = \overline{\mathrm{g}}_o \implies (\overline{\mathrm{c}}_l, \mathrm{c}_l) \vdash \overline{\mathrm{g}}_l \sqsubseteq_{\overline{\mathcal{G}}} \overline{\mathrm{g}}_o \wedge (\overline{\mathrm{c}}_r, \mathrm{c}_r) \vdash \overline{\mathrm{g}}_r \sqsubseteq_{\overline{\mathcal{G}}} \overline{\mathrm{g}}_o$$

This can be proved by induction over the joining rules over shapes.

Then, according to the soundness condition of Theorem 6.3 and Theorem 6.4, we can get:

$$\forall i \in \{l, r\}, \forall (\sigma, \mu) \vDash (\overline{\mathrm{g}}_i, \overline{\mathrm{c}}_i), \exists \mu' \textit{ extending } \mu, \mu' \vDash \mathrm{c}_i \wedge \mu' \vDash \overline{\mathrm{c}}'_i \wedge (\sigma, \mu') \vDash \overline{\mathrm{g}}_o$$

Finally, according to the soundness condition of the operator $\sqcup_{\overline{\mathcal{C}}}$,

$$\forall i \in \{l, r\}, \forall (\sigma, \mu) \vDash (\overline{\mathrm{g}}_i, \overline{\mathrm{c}}_i), \exists \mu' \textit{ extending } \mu, (\sigma, \mu') \vDash \overline{\mathrm{g}}_o \wedge \mu' \vDash \overline{\mathrm{c}}_o$$

$\blacksquare$

Widening. The widening operator $\nabla_{\overline{\mathcal{M}_\omega}}$ follows the same rewriting rules on abstract shape graphs, but applies a widening operator $\nabla_{\overline{\mathcal{C}}}$ on value-set abstractions instead of $\sqcup_{\overline{\mathcal{C}}}$. That is,

$$\frac{(\overline{\mathrm{c}}_l, \mathrm{c}_l), (\overline{\mathrm{c}}_r, \mathrm{c}_r) \vdash \overline{\mathrm{g}}_l \sqcup_{\overline{\mathcal{G}}} \overline{\mathrm{g}}_r = \overline{\mathrm{g}}_o \quad \forall i \in \{l, r\}, \overline{\mathrm{c}}_i \overset{i}{\rightsquigarrow}^{*} \overline{\mathrm{c}}'_i \wedge \overline{\mathrm{c}}'_i \implies \mathrm{c}_i \quad \overline{\mathrm{c}}_o = \overline{\mathrm{c}}'_l \nabla_{\overline{\mathcal{C}}} \overline{\mathrm{c}}'_r}{(\overline{\mathrm{g}}_l, \overline{\mathrm{c}}_l) \nabla_{\overline{\mathcal{M}_\omega}} (\overline{\mathrm{g}}_r, \overline{\mathrm{c}}_r) = (\overline{\mathrm{g}}_o, \overline{\mathrm{c}}_o)}$$

Theorem 6.7 (Widening). *The operator $\nabla_{\overline{\mathcal{M}_\omega}}$ is a widening operator on $\overline{\mathcal{M}_\omega}$.*

Proof. Based on Theorem 6.6, the soundness of the operator $\nabla_{\overline{\mathcal{M}_\omega}}$ can be proved easily.

The property that the operator $\nabla_{\overline{\mathcal{M}_\omega}}$ can ensure convergence is originally proved in [CR08]. The core idea is that the number of edges of $\overline{\mathrm{g}}_o$ will be stable:

- the number of points-to edges and inductive edges of $\overline{\mathrm{g}}_o$ is limited by the number of points-to edges and inductive edges in both inputs.
- the **ej-intro** rule may introduce segment edges, but this rule can only be applied when a pair of input nodes are equal, and thus it can not be applied more than a fixed number of times.

Once the abstract shape is stable, the value-set abstract element will eventually be stable since $\nabla_{\overline{\mathcal{C}}}$ is a widening operator over $\overline{\mathcal{C}}$. ■

6.4 Soundness of The Analysis

Finally, with all the sound underlying abstract operations, we can build an analysis that inputs inductive definitions as parameters, a C program, and a pre-condition, and it automatically computes an abstract post-condition. The analysis is implemented by composing abstract transfer functions of each program statement. Each abstract transfer function inputs an abstract state and computes an abstract post-state that is sound with respect to the abstract semantics of the program statement shown in Figure 2.7. Thus, the analysis is sound with respect to the abstract semantics of programs, shown in Theorem 2.1.

6.5 Implementation and Experimental Evaluation

We implemented inductive definitions with set predicates into the MemCAD static analyzer [TCR13, TCR14] and extended abstract operations of the analyzer to handle set variables and set predicates. The analyzer takes a set abstract domain (Chapter 5) as a parameter to represent set constraints and a numerical domain [CR13] as a parameter to represent numerical constraints.

In the following, we consider two set abstract domains:

- a BDD-based set domain (Section 5.2.2) that is based on an encoding of set constraints into BDDs, and utilizes a BDD library [Fil]. We call this domain “BDD” in the results table;
- a linear set domain (Section 5.2.1) that relies on a compact representation of constraints of the form $\mathscr{E}_i = \{\alpha_0, \ldots, \alpha_n\} \uplus \mathscr{F}_0 \uplus \ldots \uplus \mathscr{F}_m$ as well as set equalities, inclusion and membership constraints. We call this domian “LIN” in the results table, since the main set constraints expressed here are of “linear” form.

For the numerical domain, we consider a numerical domain that is obtained as a reduced product of a numerical domain that only reasons about equalities and dis-equalities and the convex polyhedra domain [CH78] supported by the Apron library [JM09]. All the abstract domains were implemented in OCaml, and integrated into the MemCAD static analyzer.

Description	LOCs	Nested	“BDD” time (ms)			“BDD”	“LIN” time (ms)			“LIN”
		loops	Total	Shape	Set	Property	Total	Shape	Set	Property
Node: add	27	0	44	0.3	11	**yes**	28	0.3	0.2	**yes**
Edge: add	26	0	31	0.2	4	**yes**	27	0.2	0.1	**yes**
Edge: delete	22	0	45	0.4	16	**yes**	30	0.3	0.2	**yes**
Node list traversal	25	1	117	1.5	87	**yes**	28	0.5	0.3	**yes**
Edge list iteration + dest. read	34	1	332	2.7	293	**yes**	36	3.5	2.4	**yes**
Graph path: deterministic	31	2	360	2.7	323	**yes**	35	2.4	2	**yes**
Graph path: random	43	2	765	7.1	711	**yes**	41	4.1	3	**yes**

Table 6.1: Analysis of a set of fundamental graph manipulation functions. Analysis times (in milliseconds) are measured on one core of an Intel Xeon at 3.20GHz with 16GB of RAM running Ubuntu 14.04.2 LTS (we show overall time including front-end and iterator shape domain and set domain), with “BDD” and “LIN” set domains. “Property” columns: inference of structural properties.

We want to assess: (1) whether the analysis achieves the verification of structure preservation in the presence of sharing, and (2) whether set domains are sufficiently precise and provide scalability, (3) whether the efficiency of the extended memory abstract domain is preserved compared to the original memory abstract domain presented in Sections 2.3 and 2.4.

Therefore, we parameterize the analysis with the inductive definitions of adjacency lists shown in Figure 3.3. We ran the analysis on a basic graph library, specifically chosen to assess the handling of shared structures (addition or removal operations, structure traversals, and traversals following paths, including the program of Figure 3.5). For each program, we input a pre-condition that abstracts all the valid input data structures of the program, and a post-condition that needs to be verified by the analysis for proving data structure preservation. The results of the analysis are presented in Table 6.1.

As shown in the “Property” columns, in all cases, the analysis with both set domains successfully establishes memory safety (absence of null / dangling pointer dereferences), structural preservation for the graph modifying functions and traversal functions. Note that, in the case of path traversals, memory safety requires the analysis to localize the cursor as a valid graph node, at all times (the strongest set property, captured by the graph inductive definitions of Figure 3.3). As an example, we show in Figure 6.9 the invariants computed during the first loop iteration of the analysis of the random graph path traversal program of Figure 3.5 and in Figure 6.10 the invariants obtained after reaching an abstract post-fixpoint.

Both the BDD-based set domain and the linear set domain are sufficiently precise to analyze the benchmarks of Table 6.1. However, the linear set domain is more efficient than the BDD-based set domain as shown in the “Set” columns which are the analysis times spent on the set domain. Yet the BDD-based set domain proves to be inefficient in this situation and takes up most of the analysis time for three reasons: (1) it keeps properties that are not relevant to the analysis and (2) renaming set variables (required after joins) necessitates full re-computation of BDDs. In contrast, the linear set domain is tailored for the predicates required in the analysis, and produces very quick analysis run-times. (3) BDDs may need more iterations for convergence. To conclude, the linear set domian has reliable performance and predictable precision, which

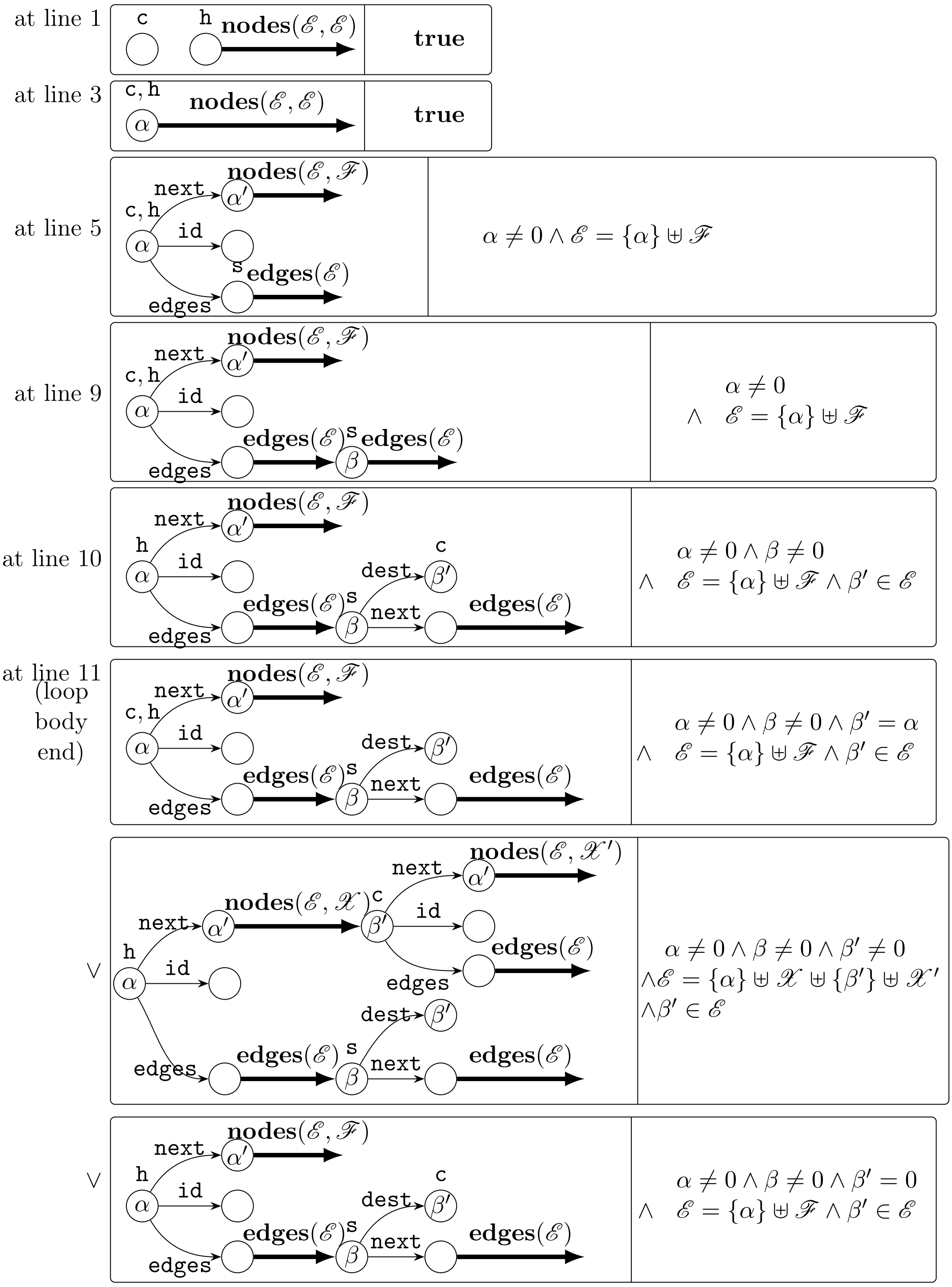


Figure 6.9: Local invariants computed over the first iteration

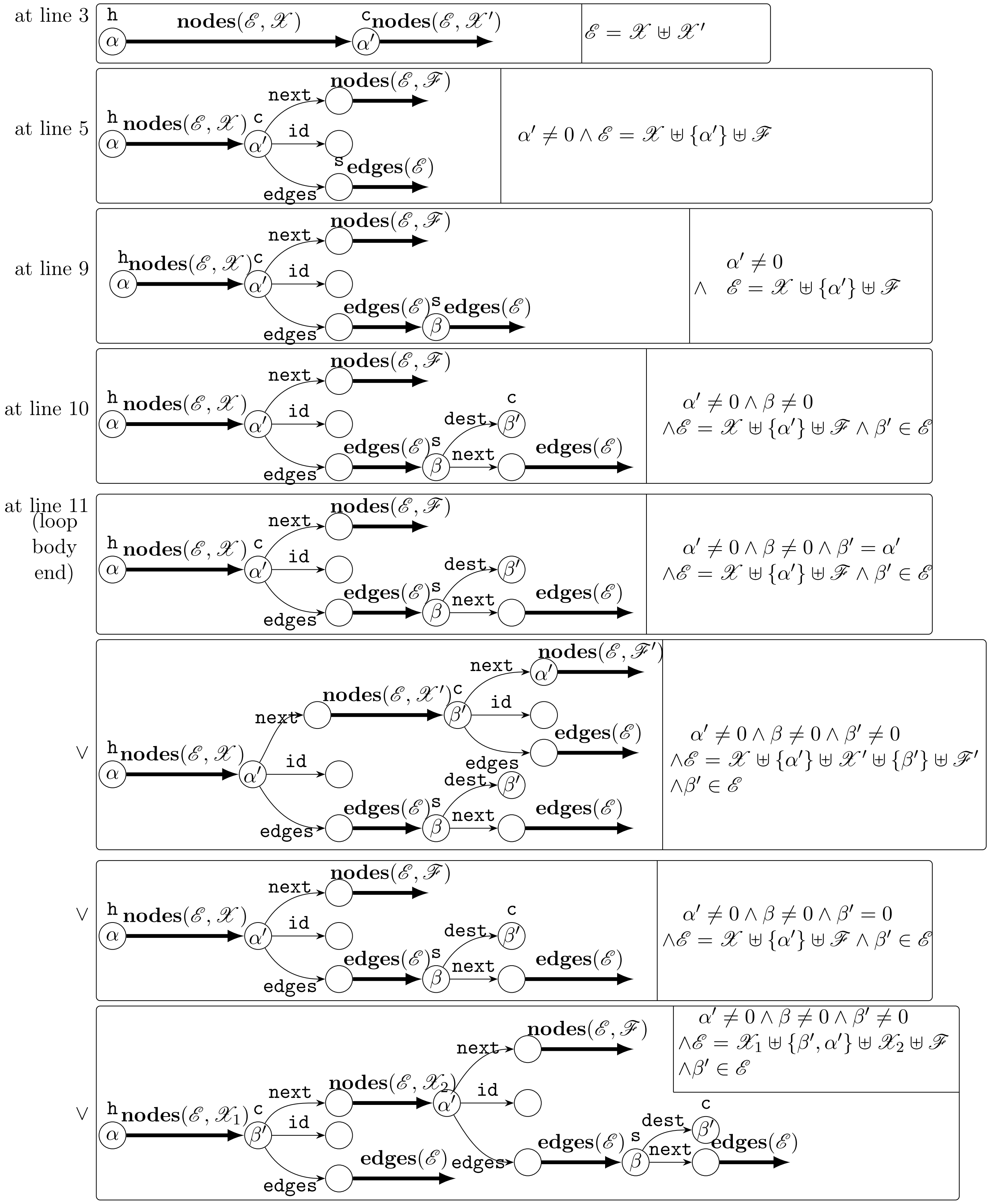

at line 3
at line 5
at line 9
at line 10
at line 11 (loop body end)
∨
∨
∨
h
c
s
next
id
edges
dest
α
α'
β
β'
nodes$(\mathscr{E}, \mathscr{X})$
nodes$(\mathscr{E}, \mathscr{X}')$
nodes$(\mathscr{E}, \mathscr{F})$
nodes$(\mathscr{E}, \mathscr{F}')$
nodes$(\mathscr{E}, \mathscr{X}_1)$
nodes$(\mathscr{E}, \mathscr{X}_2)$
edges$(\mathscr{E})$
$\mathscr{E} = \mathscr{X} \uplus \mathscr{X}'$
$\alpha' \neq 0 \wedge \mathscr{E} = \mathscr{X} \uplus \{\alpha'\} \uplus \mathscr{F}$
$\alpha' \neq 0$
$\wedge \quad \mathscr{E} = \mathscr{X} \uplus \{\alpha'\} \uplus \mathscr{F}$
$\alpha' \neq 0 \wedge \beta \neq 0$
$\wedge \mathscr{E} = \mathscr{X} \uplus \{\alpha'\} \uplus \mathscr{F} \wedge \beta' \in \mathscr{E}$
$\alpha' \neq 0 \wedge \beta \neq 0 \wedge \beta' = \alpha'$
$\wedge \mathscr{E} = \mathscr{X} \uplus \{\alpha'\} \uplus \mathscr{F} \wedge \beta' \in \mathscr{E}$
$\alpha' \neq 0 \wedge \beta \neq 0 \wedge \beta' \neq 0$
$\wedge \mathscr{E} = \mathscr{X} \uplus \{\alpha'\} \uplus \mathscr{X}' \uplus \{\beta'\} \uplus \mathscr{F}'$
$\wedge \beta' \in \mathscr{E}$
$\alpha' \neq 0 \wedge \beta \neq 0 \wedge \beta' = 0$
$\wedge \mathscr{E} = \mathscr{X} \uplus \{\alpha'\} \uplus \mathscr{F} \wedge \beta' \in \mathscr{E}$
$\alpha' \neq 0 \wedge \beta \neq 0 \wedge \beta' \neq 0$
$\wedge \mathscr{E} = \mathscr{X}_1 \uplus \{\beta', \alpha'\} \uplus \mathscr{X}_2 \uplus \mathscr{F}$
$\wedge \beta' \in \mathscr{E}$

Figure 6.10: Local invariants computed after reaching a post-fixpoint

can be used in a specific analysis, while the BDD-based set domain can always provide excellent precision but with a risk of less reliable performance.

The "Total" columns in Table 6.1 show the total analysis time using different set domains and the "Shape" columns show the time spent on the shape domain. We find that, in all cases, the total analysis time is less than one second, and in particular the analysis parameterized by the linear set domain takes less than 50ms run-time, even in complex cases which require non-local unfolding and instantiation of fresh set variables. Moreover, the time spent in the shape domain is in line with those usually observed in the analyzer [TCR13, TCR14]. Therefore, the performance of the analysis is well preserved with an efficient set domain.

6.6 Related Works of The Analysis for Unstructured Sharing

A wide family of shape analysis techniques have been proposed to deal with inductive structures, often based on 3-valued logic [SRW02, LAS00] or on separation logic [BCO05, DOY06, BCC$^+$07, NDQC07, CDOY09, CR08]. Such analyses often deal very well with list- and tree- like structures, but are often challenged with *unbounded* sharing.

In this part of the thesis, we augmented a separation logic-based analysis [CRN07, CR08] with set predicates to handle unbounded sharing, both in summaries and in unfolded regions, and retain the parameterizability of this analysis.

A shape analysis tracking properties of structure contents was presented in [Vaf09], although with a less general set abstraction interface, and without support for unfolding guided by set parameters. Another related approach was proposed in [CRB10], that utilizes a set of data-structure specific analysis rules and encodes sharing information into instrumentalized variables in order to analyze programs manipulating trees and graphs. In contrast, our analysis does no such instrumentation and requires no built-in inductive definitions.

Recently, a set of studies [DES13, KSV10, LYP11, TCR13] targeted *overlaid* structures, which feature some form of *structured* sharing, such as a tree overlaid on a list. Typically, these analyses combine several abstractions specific to the different layers. We believe that the problem is orthogonal to ours, since we consider a form of sharing that is not structured, and we need to achieve *non-local* materialization.

Another line of work that is slightly related to ours are the hybrid analyses that aim at discovering relations between the structures and their contents [CR08, FFJ12, BDES12]. Our set predicates actually fit in the domain product formalized in [CR08], and can also indirectly capture relations between structures and their contents through sets. Abstractions of set properties have recently been used in order to capture relations between sets of keys of dictionaries [DDA11, CCR14] or groups of array cells [LR15]. A noticeable result is that our analysis tracks very similar predicates, although for a radically different application.

6.7 Conclusion on The Analysis for Unstructured Sharing

In this part of the thesis, we have set up a shape analysis able to deal with unbounded sharing. This analysis combines separation logic-based shape abstractions and a *set abstract domain* to track pointer sharing properties. Reduction across domains is done lazily at non-local materialization and join. This abstraction was implemented in the MemCAD static analyzer and could cope with graphs described by adjacency lists. Future work will experiment with other set abstract domains and combine this abstraction with other memory abstractions [TCR13, TCR14].

Part III

Silhouette-guided Disjunct Clumping

Chapter 7

Overview

In this part, we propose semantic-directed clumping of disjunctive abstract states: a semantic criterion *to clump abstract states based on their silhouette, which applies not only to the conservative union of disjuncts but also to the weakening of separating conjunctions of memory predicates into inductive summaries. Before the formalization, in this chapter, we first introduce the disjunct clumping problem and describe the core idea of using silhouettes in disjunct clumping of abstract memory states at a high level.*

7.1 Semantic-Directed Disjunct Clumping

In this section, we first recall the disjunct clumping problem in shape analysis, which is originally introduced in Section 1.5.2. Then, we overview our approach to disjunct clumping based on the use of an *abstraction of abstract states.*

7.1.1 Disjunct Clumping Problem

Many separation conjunction based shape analyses, e.g., the analysis presented in [BCO05, DOY06, CRN07, YLB+08] and in Part II, manipulate abstract memory states that consist of *separating conjunctions* of basic region predicates. As formalized in Section 4.2, region predicates typically either describe a finite region very precisely, e.g., points-to predicates, or summarize an unbounded region, e.g., inductive and segment predicates. That is, an abstract state can be viewed as a logical formula described by a grammar of the form:

$$\begin{array}{lrcll}
\text{abstract states:} & \overline{\mathrm{m}}(\in \overline{\mathcal{M}_{\omega}}) & ::= & \overline{\mathrm{p}} * \ldots * \overline{\mathrm{p}} & \\
\text{region predicates:} & \overline{\mathrm{p}}(\in \overline{\mathcal{P}}) & ::= & \overline{\mathrm{p}}_{\mathrm{e}} & \text{(exact description)} \\
 & & \mid & \overline{\mathrm{p}}_{\mathrm{s}} & \text{(summary)}
\end{array}$$

Indeed, apart from separation conjunction based shape analysis, many other analyses also rely on expressive sets of predicates in order to abstract structures, such as shape analysis based on three-valued logic [SRW02], array analysis [CCL11] dealing with contiguous structures indexed

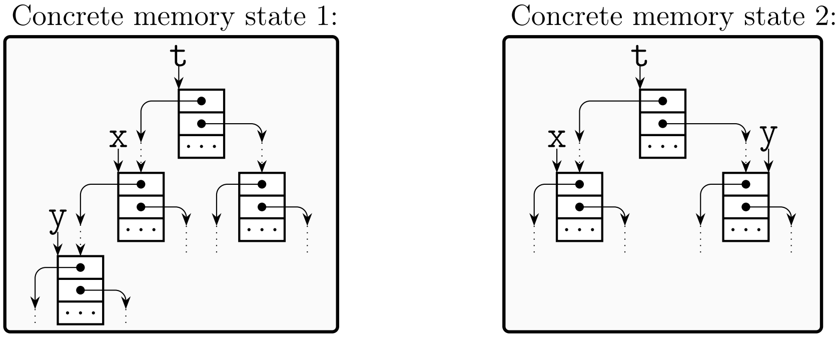


Figure 7.1: Tree fragments and need for disjunctions

by ranges of integers, and dictionary analysis [DDA11, CCR14] that handles structures indexed using sets of keys.

However, such a set of logical predicates is often not sufficient to describe a set of concrete memory states precisely as programs can produce very different structures at a program point. As an example, it is impossible to precisely abstract the concrete memory states shown in Figure 7.1 into a single abstract state, as the relative positions of variables t, x and y in the tree are very different, yet they can be precisely abstracted by a disjunction of two abstract states. Therefore, disjunctive abstract states $\overline{\mathrm{d}}(\in \overline{\mathcal{D}}) ::= \overline{\mathrm{m}} \vee \ldots \vee \overline{\mathrm{m}}$ are necessary in analysis.

In practice, disjunctions are a huge challenge to static analysis. The creation of new disjunctions occurs naturally when the analysis manipulates branch statements and loops, and when the analysis needs to unfold summary predicates to reason about operations that read or write into summary predicates. However, letting the number of disjuncts grow makes the analysis slower and consumes more memory. Nonetheless, getting rid of unnecessary disjuncts turns out to be a much harder task than introducing them.

We let $\sqcup_{\overline{\mathcal{M}_\omega}}$ denote a computable join operation over the set of abstract states that over-approximates unions of sets of memory states. It is always sound for the analysis to replace a disjunctive abstract state $\overline{\mathrm{m}}_0 \vee \overline{\mathrm{m}}_1$ with a non-disjunctive abstract state $\sqcup_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}_0, \overline{\mathrm{m}}_1)$, but this operation generally leads to a loss of information. Disjunct clumping aims at identifying a partition of a finite set of disjuncts so that each component of the partition can be joined using the join operator $\sqcup_{\overline{\mathcal{M}_\omega}}$ without a significant loss of precision. However, existing approaches all come with limitations. For instance, both the local rewriting rules-based canonicalization operator of [DOY06] and the control flow structure (conditions, loops, etc.) based partition strategies are likely to produce inadequate and imprecise disjunct clumps as they either totally ignore the shapes or ignore global shape properties.

Therefore, to better solve the disjunct clumping problem, we need to:

- identify global shape properties of abstract states, such that abstract states sharing similar global shape properties can be joined using the join operator $\sqcup_{\overline{\mathcal{M}_\omega}}$ without a significant loss of precision. For instance, abstract states of the concrete states shown in Figure 7.1 should have different global shape properties as the joining operation cannot produce a precise over-approximation.

- design a disjunct clumping strategy that considers global shape properties of abstract states and can quickly identify a partition of a finite set of disjuncts, such that abstract states in each partition group share similar global shape properties.

7.1.2 Clumping Disjuncts Based on their Abstraction

In this part, we propose an approach to disjunct clumping that is based on using *silhouettes* (an abstraction of abstract states) to represent global properties of abstract states:

- we design a set of logical predicates $\overline{\mathfrak{S}}$ that describe the *silhouettes* of abstract states in a compact and simple representation, and a computable silhouette equivalence relation $\sim$ that defines which silhouettes are similar;
- we define a computable abstraction function $\theta : \overline{\mathcal{M}_\omega} \longrightarrow \overline{\mathfrak{S}}$ that maps an abstract state into its silhouette.

Using this notion of silhouette, we define the clumping algorithm **clump** that inputs a disjunctive abstract state $\overline{\mathrm{d}}$ and returns another disjunctive abstract state **clump**($\overline{\mathrm{d}}$) that over-approximates $\overline{\mathrm{d}}$ with at most as many (though usually fewer) disjuncts:

1. it inputs a disjunctive abstract state $\overline{\mathrm{m}}_0 \vee \ldots \vee \overline{\mathrm{m}}_n$;
2. it computes $\theta(\overline{\mathrm{m}}_0), \ldots, \theta(\overline{\mathrm{m}}_n)$, and packs together abstract states with the same image by θ up to the equivalence relation $\sim$;
3. it reduces each group into a single abstract state, by repeatedly applying $\sqcup_{\overline{\mathcal{M}_\omega}}$, and returns a more compact disjunction.

Example 7.1 (Silhouettes in numerical analysis). In this example, we consider programs that manipulate integer variables, and perform basic arithmetic operations. A common goal for static analyses consists of verifying the absence of division by zero errors. Let us assume such an analysis relies on interval constraints over program variables: a division by x can be proved safe if the analysis computes for x either an interval of strictly negative numbers or an interval of strictly positive numbers. We note that joining together an interval of strictly negative numbers and an interval of strictly positive numbers will return an interval that contains zero. Thus, after such a join, crucial information will be lost to verify that a division by x is safe.

The sign abstraction provides an obvious characterization which interval unions will produce such an imprecise result: the sign abstract domain is made up of four abstract values $\bot, \ominus, \oplus, \top$ that respectively denote the empty set, any set of strictly negative integers, any set of strictly positive integers, and any set of integers. For instance, if two intervals have sign $\ominus$, they can be joined without losing information with respect to zero, whereas the join of an interval with sign $\ominus$ with an interval with sign $\oplus$ will cause a loss of precision.

Therefore, a principle similar to silhouettes allows us to enhance the precision of the analysis, with respect to the goal of proving variables are not equal to zero:

- disjunctive states composed of finite disjunctions of intervals can keep more precise information about sets of states where a variable x may be positive or negative;
- the "silhouette" of intervals defined by their sign abstraction characterizes when joining two intervals will cause a precision loss, with respect to the property of interest.

7.2 Analysis of an AVL Tree Insertion Function

In this section, we define an instance of clumping of disjunctive predicates, by defining $\overline{\mathfrak{S}}$, θ and $\sim$ and we demonstrate how this approach enables the verification of a function that inserts an element into an AVL tree, while limiting the size of disjunctions.

Example: insertion into an AVL tree. AVL trees achieve balancing by enforcing that the heights of two subtrees of a single node differ by at most one, and by storing that difference on each node, so that insertion and removal algorithms can re-balance subtrees incrementally, using "rotation" operations. This property makes insertion and removal complicated, thus the preservation of structural invariants and absence of memory errors (such as illegal pointer operations or leakage of subtrees) are difficult to verify. In particular, the insertion and removal functions need to distinguish many cases, which the functions' verification should also distinguish. Thus, verification algorithms will produce large numbers of disjuncts, and could greatly benefit from clumping.

Fig. 7.2 shows an extract of an AVL insertion function. While the verification of the full code (from [Wal03]) is presented in Chapter 11, we study here a simplified version for the sake of clarity. We only consider the handling of disjunctions of memory shapes; in particular, we ignore the AVL trees numeric balancing constraints (for our purposes, this restriction has no impact on the analysis).

The fragment in Fig. 7.2 handles the insertion into a non-empty tree, and carries it out in three phases. First, the node `p` at which the new element should be inserted is localized, as well as the deepest edge (with source `x` and target `y`) in the tree where the balancing property is locally broken by this insertion (indeed, this is the only point where a re-balancing will actually be required, since a rotation at this point will prevent any other balancing constraint from being broken). Second, a new node is allocated and inserted at position `p`. Finally, the re-balancing itself is performed. We consider in detail only the first phase. This phase should compute pointers $\mathtt{p}, \mathtt{x}, \mathtt{y}$. To verify this function, the analysis should compute invariants that characterize precisely the shape of trees, subtrees and the relative positions of pointers $\mathtt{p}, \mathtt{x}, \mathtt{y}, \mathtt{t}$ in all cases, including the corner case where tree `t` consists of a single node, and the insertion is actually done above it.

Abstract states and analysis. To express invariants over the code in Fig. 7.2, the analysis needs inductive predicates and segment predicates parameterized by inductive definitions to describe trees of unbounded size as well as cursors inside trees. Let us assume a **tree** definition (introduced in Section 2.3) describing either an empty tree or a tree with an AVL node and two disjoint subtrees:

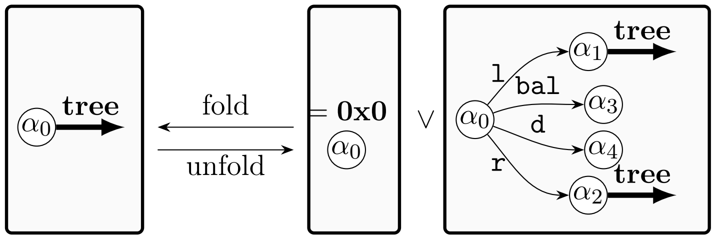
$$\begin{array}{lll}
\alpha_0 \cdot \mathbf{tree} & ::= & \\
 & & \mathbf{emp} \wedge \alpha_0 = 0 \\
 & \vee & \alpha_0.\mathtt{l} \mapsto \alpha_1 * \alpha_0.\mathtt{r} \mapsto \alpha_2 * \alpha_0.\mathtt{bal} \mapsto \alpha_3 * \alpha_0.\mathtt{d} \mapsto \alpha_3 * \alpha_1 \cdot \mathbf{tree} * \alpha_2 \cdot \mathbf{tree} \wedge \alpha \neq 0
\end{array}$$

This definition boils down to a pair of fold (Section 2.4.4) / unfold (Section 2.4.2) operations , that fully characterize **tree**:

```
typedef struct node_t {
  struct node_t *l, *r; // left, right child
  int bal, d; // balancing and content
} node;

int insert_non_empty( node * t, int i ){
  assume( t != null );
  node *h = (node*) malloc(sizeof(node));
  node *x, *y, *p, *q;
  h->l = y = p = t;
  x = h;
  // phase 1: insertion point localization
  while( 1 ){
    if( i < p->d )
      q = p->l;
    else
      q = p->r;
    if( q == null )
      break;
    if( q->bal != 0 ){
      x = p;
      y = q;
    }
    p = q;
  }
  // phase 2: insertion at position p...
  // phase 3: rebalancing at position x, y...
}
```

Figure 7.2: Extract of C source code for the AVL tree insert function.



Following the inductive definition, the abstract state on the right hand side of Figure 7.3 describes states where `t` points to a tree, and `x` points to a (possibly non-strict) subtree of the tree pointed to by `t`. It is made up of the separating conjunction of a tree segment $\alpha \cdot \mathbf{tree} \ast\!\!= \beta \cdot \mathbf{tree}$ summary and an inductive predicate $\beta \cdot \mathbf{tree}$. An example of concrete memory is shown on the left hand side of the figure.

The analysis performs a forward abstract interpretation [CC77], starting from an abstract precondition that takes into account all the possible valid call states, that is all the memory

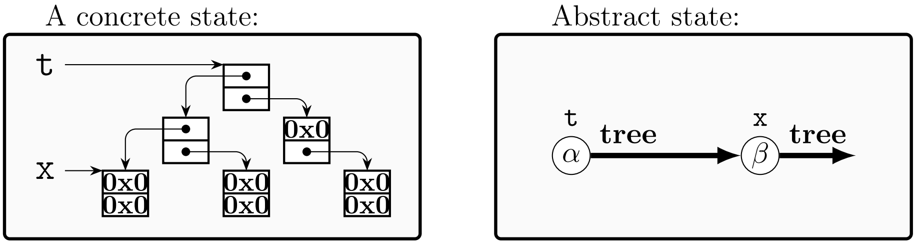


Figure 7.3: Abstract states.

states where `t` points to a non-empty, well-formed tree (described by **tree**). It computes abstract post-conditions for each statement, and unfolds summaries on-demand: for instance, in the first iteration `p` = `t`, and at line 14, the reading of field `d` of the node pointed to by `p` requires unfolding the **tree** summary predicate attached to `t`. Such unfoldings generate disjunctions of case splits. Conversely, join and widening applied at loop head should fold back case splits, so as to compute a loop invariant.

The disjuncts merging challenge. Fig. 7.4 describes a few abstract states that are observed at line 25, at the exit of the first loop, and that illustrate the challenges of clumping disjuncts. The figure shows abstract states, the generic form of concrete memories they represent and their silhouettes, which will be explained below. For concision, and as only the relative positions of cursors matter, we focus on `t`, `x`, `y` and omit fields `d`, `bal`, and variables `h`, `p`, `q`. Labels `t`, `x`, `y` decorate the nodes that represent their value. Moreover, we show only a sample of 4 out of 32 disjuncts:

- Abstract state $\overline{\mathrm{m}}_0$ abstracts memories where `x`, `y` were not advanced.
- Abstract state $\overline{\mathrm{m}}_1$ abstracts memories where `x` was advanced to the root of the tree, and the search continued in the left subtree.
- Abstract state $\overline{\mathrm{m}}_2$ abstracts memories where the search visits the left subtree of `t`, `x` is advanced into that subtree, and `y` is the left child of `x`.
- Abstract state $\overline{\mathrm{m}}_3$ describes a similar condition to $\overline{\mathrm{m}}_2$ but when the search visits the right subtree of `t` and `x` is a left child.

Some of these abstract states are very similar to each other and can be joined to reduce the cost without significantly affecting the precision of the analysis. Indeed, `x` and `y` occupy the same relative positions in $\overline{\mathrm{m}}_2$ and $\overline{\mathrm{m}}_3$; moreover, in both cases the two cursors point to subtrees of `t`. Thus, both $\overline{\mathrm{m}}_2$ and $\overline{\mathrm{m}}_3$ can be approximated by an abstract state with a segment from `t` to `x` and where `y` is the left child of `x`. Furthermore, $\overline{\mathrm{m}}_1$ can also be weakened similarly: the relative positions of `x`, `y` are the same, and the only minor difference is that `x` is not a strict subtree of `t` since `t` = `x`, but this equality can also be described by an (empty) segment. On the other hand, $\overline{\mathrm{m}}_0$ abstracts very different memories, where `x` is *not* a subtree of `t`, so it cannot be described with a segment from `t` to `x`. Any abstract state that over-approximates both $\overline{\mathrm{m}}_0$ and

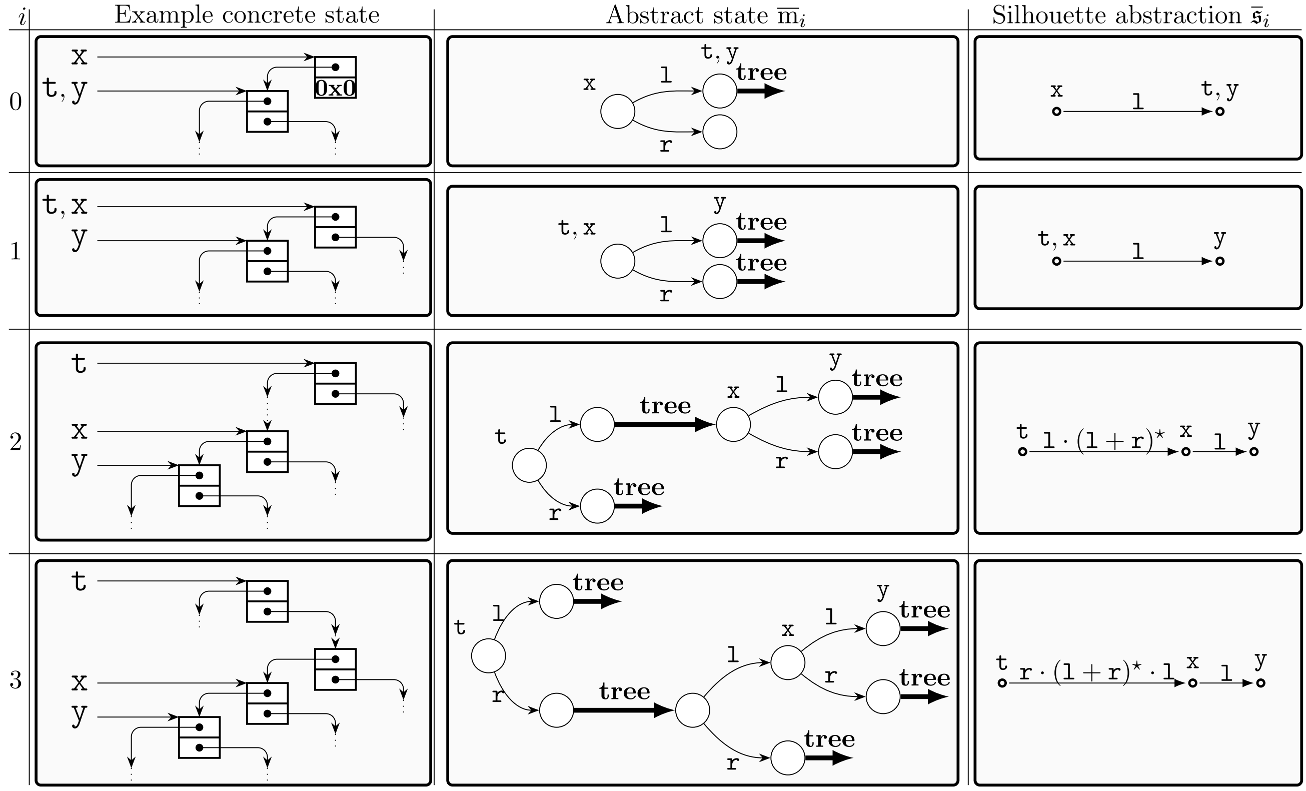

i
Example concrete state
Abstract state $\overline{\mathfrak{m}}_i$
Silhouette abstraction $\bar{\mathfrak{s}}_i$
0
x
t, y
0x0
x
l
t, y
tree
r
x
l
t, y
1
t, x
y
t, x
l
y
tree
tree
r
t, x
l
y
2
t
x
y
t
l
tree
r
tree
x
l
y
tree
r
tree
t
$\mathtt{l}\cdot(\mathtt{l}+\mathtt{r})^\star$
x
l
y
3
t
x
y
t
l
tree
r
tree
l
x
l
y
tree
r
tree
r
tree
t
$\mathtt{r}\cdot(\mathtt{l}+\mathtt{r})^\star\cdot\mathtt{l}$
x
l
y

Figure 7.4: Selected abstract states from the analysis of an insertion into an AVL tree (4 disjuncts out of 32).

$\overline{\mathfrak{m}}_1$ would discard all information about either `t` or `x`, which would make the proof of structural preservation impossible. Therefore, an ideal clumping would join $\overline{\mathfrak{m}}_1, \overline{\mathfrak{m}}_2, \overline{\mathfrak{m}}_3$ together but keep $\overline{\mathfrak{m}}_0$ separate.

Silhouette abstraction. Intuitively, abstract states where the relative positions of cursors `t`, `x`, `y` in the tree are similar can be clumped together with no severe precision loss. In line with this intuition, the notion of *silhouette* (formalized in Chapter 8) retains only the relative positions of `t`, `x`, `y` and the access paths between them, as shown in the last column of Fig. 7.4. As access paths may be of unbounded length, we abstract them with regular expressions describing sequences of fields dereferenced between nodes. For instance, the silhouette $\bar{\mathfrak{s}}_0$ of $\overline{\mathfrak{m}}_0$ boils down to a single edge, with a single path labeled by `l`. Even the more complex $\overline{\mathfrak{m}}_3$ is characterized by only two edges respectively labeled by $\mathtt{r}\cdot(\mathtt{l}+\mathtt{r})^\star\cdot\mathtt{l}$ (between `t` and `x`) and by `l` (between `x` and `y`).

Clumping disjuncts. Silhouettes make the dissimilarity of $\overline{\mathfrak{m}}_0$ with the other states in Fig. 7.4 obvious, due to the incompatible order of cursors. However, we can also see that the silhouettes

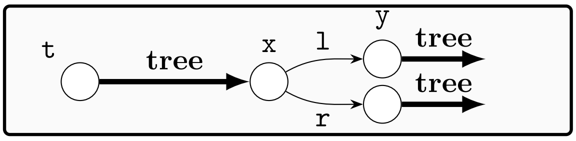


Figure 7.5: Weakened abstract state $\overline{\mathbb{m}}_{1,2,3}$.

of $\overline{\mathbb{m}}_2$ and $\overline{\mathbb{m}}_3$, while not syntactically identical are actually equal up-to a generalization of the path regular expression for the left edge into $(\mathtt{l}+\mathtt{r})^\star$. This generalization matches the fact that any structure segment can be weakened into a tree segment predicate, whatever the sequence of "left-right" branches it encompasses:

$$\bar{\mathbb{s}}_\sim = \boxed{\overset{\mathtt{t}}{\circ}\xrightarrow{(\mathtt{l}+\mathtt{r})^\star}\overset{\mathtt{x}}{\circ}\xrightarrow{\mathtt{l}}\overset{\mathtt{y}}{\circ}}$$

The silhouette of $\overline{\mathbb{m}}_1$ also corresponds to a special case of $\bar{\mathbb{s}}_\sim$. Therefore, we let $\sim$ denote the similarity of silhouettes up-to generalization of access paths (formalized in Chapter 10). With this notation, we have:

$$\bar{\mathbb{s}}_0 \not\sim \bar{\mathbb{s}}_1 \qquad \bar{\mathbb{s}}_0 \not\sim \bar{\mathbb{s}}_2 \qquad \bar{\mathbb{s}}_0 \not\sim \bar{\mathbb{s}}_3 \qquad \bar{\mathbb{s}}_1 \sim \bar{\mathbb{s}}_2 \sim \bar{\mathbb{s}}_3$$

Therefore, the groups computed here are $\{\overline{\mathbb{m}}_0\}, \{\overline{\mathbb{m}}_1, \overline{\mathbb{m}}_2, \overline{\mathbb{m}}_3\}$, and the four disjuncts of Fig. 7.4 are clumped into a disjunctive abstract state composed of only two disjuncts $\overline{\mathbb{m}}_0$ and $\overline{\mathbb{m}}_{1,2,3}$. The computation of the $\overline{\mathbb{m}}_{1,2,3}$ (which we discuss in the next paragraph) may use any weakening algorithm for abstract states, such as canonical abstraction [SRW02], canonicalization of symbolic heaps [DOY06], or shape graph join [CRN07].

In essence, clumping relies on a weak canonicalization [SRW02] that returns silhouettes, and then selects groups based on an equivalence relation over the set of silhouettes. To ensure the termination of abstract iterates over loops, the clumping relation $\sim$ is required to be finite (i.e., it should have a finite set of equivalence classes), though the set of silhouettes may still be infinite. However, clumping may be performed at other points than loop heads (in order to shrink abstract states, without discarding any information), and then it does not require a finite $\sim$. From this point of view, clumping provides a more flexible approach to the handling of disjunctions than canonicalization, since it does not need to project abstract states into a finite set of predicates, and since it may still take advantage of precise binary operators for weakening.

Clumping region predicates. The information computed in silhouettes also provides a guideline for weakening abstract states (formalized in Chapter 9). Indeed, let us consider silhouette $\bar{\mathbb{s}}_\sim$, which is weaker than the silhouettes of $\overline{\mathbb{m}}_1, \overline{\mathbb{m}}_2, \overline{\mathbb{m}}_3$. As it contains an edge labeled by $(\mathtt{l}+\mathtt{r})^\star$ between `t` and `x`, it suggests weakening the fragment of these abstract states that are between `t` and `x` into a tree segment predicate. The soundness of such a weakening can be verified by checking entailment of a fragment of $\overline{\mathbb{m}}_1$ (resp., $\overline{\mathbb{m}}_2, \overline{\mathbb{m}}_3$) and a **treeseg** predicate. The resulting weaker abstract state $\overline{\mathbb{m}}_{1,2,3}$ is shown in Figure 7.5. Here, the role of the silhouette is to provide guidance on how abstract states may be weakened. The advantage of this approach to the computation of weakening is that it provides global semantic information about the structure of abstract states, which would be missed if weakening operators based only on syntactic rules (like the canonical heap abstraction of [DOY06] and the join of [CRN07]).

Chapter 8

Silhouette Abstractions: Abstraction of Memory Abstract States

This chapter formalizes silhouettes, and shows that they provide a useful abstraction to reason about the weakening of abstract states. Specifically, Section 8.1 formalizes silhouettes and their computation; Section 8.2 studies the relationship between entailment checking of abstract states and silhouettes.

8.1 Silhouette Abstraction

In this section, we first formalize silhouettes in Section 8.1.1, and then provide the computation of silhouettes in Section 8.1.2.

8.1.1 Definitions

Before formalizing silhouettes, we first simplify the abstract states introduced in Section 2.3 so as to be able to give a clear presentation of the principle underlying our approach. Specifically, we omit abstracting memory cells of program variables as they do not have any effect on clumping disjuncts, and we replace numerical abstractions with simple numerical constraints expressing equalities and disequalities.

Abstract states. An abstract value $\overline{v} \in \overline{\mathcal{V}}$ is either a symbolic variable $\alpha \in \overline{\mathcal{V}}_\alpha$ or the symbolic value of a program variable $\mathtt{x} \in \mathcal{X}$. An *abstract state* $\overline{\mathrm{m}} \in \overline{\mathcal{M}_\omega}$ (formalized in Figure 8.1) describes a set of concrete memory states. It is either $\bot$, $\top$ or a separating conjunction of region predicates that abstract *separate* memory regions [Rey02] in conjunction with numerical constraints such as equalities and disequalities.

The logical meaning of an abstract state $\overline{\mathrm{m}}$ is defined by its concretization $\gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}) \subseteq \mathcal{H} \times (\mathcal{X} \to \mathcal{V})$, as a set of pairs made up of a memory store $\sigma \in \mathcal{H}$ and an evaluation function $\mu \in \mathcal{X} \to \mathcal{V}$ that maps each program variable to its concrete value. In addition, for each concrete state $(\sigma, \mu) \in \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}})$, there exists a valuation function $\mu' \in \overline{\mathcal{V}}_\alpha \to \mathcal{V}$ mapping each symbolic

$$
\begin{array}{lcll}
\overline{\mathrm{p}} & ::= & \mathbf{emp} & \text{(empty memory)} \\
 & \mid & \overline{v} \cdot \mathtt{f} \mapsto \overline{v}' & \text{(single memory cell)} \\
 & \mid & \overline{v} \cdot \mathbf{ind} & \text{(inductive summary predicate)} \\
 & \mid & \overline{v} \cdot \mathbf{ind} \mathrel{*\!\!=} \overline{v}' \cdot \mathbf{ind} & \text{(segment summary predicate)} \\
\overline{\mathrm{g}} & ::= & \overline{\mathrm{p}} * \ldots * \overline{\mathrm{p}} & \text{(abstract shape graph)} \\
\mathrm{c} & ::= & \overline{v} \odot \mathbf{0x0} \quad (\odot \in \{=, \neq\}) & \\
 & \mid & \overline{v} = \overline{v}' & \\
 & \mid & \mathrm{c} \wedge \mathrm{c} & \\
\overline{\mathrm{m}} & ::= & \overline{\mathrm{g}} \wedge \mathrm{c} & \\
\overline{\mathrm{d}} & ::= & \overline{\mathrm{m}} \vee \ldots \vee \overline{\mathrm{m}} &
\end{array}
$$

Figure 8.1: Definition of abstract states

variable into its concrete counterpart in σ, such that $(\sigma, \mu \uplus \mu') \vDash \overline{\mathrm{m}}$, i.e.,

$$\gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}) = \{(\sigma, \mu_{\lceil \mathcal{X}}) \mid (\sigma, \mu) \vDash \overline{\mathrm{m}}\}$$

The detailed definitions of $\gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}})$ and $\gamma_{\overline{\mathcal{D}}}(\overline{\mathrm{d}})$ are omitted as they can be formalized similarly as in Sections 2.3 and 4.2. We note that symbolic variables are existentially quantified, and thus the concretization of abstract states is unchanged by renaming symbolic variables. Similarly, swapping node names or merging equal nodes preserves the meaning of abstract states.

Silhouettes. Abstract states provide a precise description of sets of concrete states. Even summarized regions are characterized by inductive predicates that convey very detailed information about their structure. The purpose of silhouettes, as introduced in Section 7.1.2, is to identify similarities among abstract states without looking into the details of the shapes of the structures. Thus, we define silhouettes as graphs, yet with edges that retain less information than region predicates. We notice that information about reachability on pointer paths is relevant to the characterization of groups of abstract states that could be clumped together. Paths can be described using basic regular expressions over fields. We let $\mathscr{E}$ denote the set of the regular expressions that satisfy the grammar $\mathrm{e} ::= \epsilon \mid \mathtt{f} \mid (\mathtt{f}_0 + \ldots + \mathtt{f}_n)^\star \mid \mathrm{e} \cdot \mathrm{e}$ which are respectively adequate to describe equalities, points-to edges, segment edges, and sequences of edges. This leads us to the following definition:

Definition 8.1 (Silhouette). *A* silhouette $\bar{\mathfrak{s}}$ *is a graph defined by a set of nodes* $\mathrm{N} \subseteq \overline{\mathcal{V}}$*, and a set of edges* E *that are labeled by regular expressions in* $\mathscr{E}$ *(we write* $(\overline{v}, \mathrm{e}, \overline{v}')$ *for such an edge).*

Concretization. A silhouette collects a conjunction of reachability constraints over paths. Thus, the concretization $\gamma_{\overline{\mathfrak{S}}}(\bar{\mathfrak{s}})$ of a silhouette $\bar{\mathfrak{s}}$ is defined as the set of pairs (σ, μ) made up of a memory store σ and an evaluation function of program variables, and for each pair (σ, μ), there exists an evaluation function μ' of symbolic variables, such that $(\sigma, \mu \uplus \mu')$ satisfies all the constraints defined by the edges of $\bar{\mathfrak{s}}$. Deciding whether $(\sigma, \mu \uplus \mu')$ satisfies the constraint defined by an edge $(\overline{v}, \mathrm{e}, \overline{v}')$ comes down to evaluating the values described by $\overline{v}, \overline{v}'$ into values a, a' and checking whether dereferencing a sequence of fields described by e starting from a makes it possible to reach a'. This leads us to the following definition:

Definition 8.2 (Silhouette concretization). *The following set of rules formalizes whether* (σ, μ) *satisfies the constraint defined by a silhouette edge* $(\overline{v}, \mathrm{e}, \overline{v}')$*:*

$$\frac{(\sigma,\mu) \vDash \mu(\overline{v}), \mathrm{e}, \mu(\overline{v}')}{(\sigma,\mu) \vDash (\overline{v}, \mathrm{e}, \overline{v}')} \qquad \frac{}{(\sigma,\mu) \vDash a, (\mathtt{f}_0 + \ldots + \mathtt{f}_n)^\star, a}$$

$$\frac{\sigma(a + \mathtt{f}) = a'}{(\sigma,\mu) \vDash a, \mathtt{f}, a'} \qquad \frac{\exists i,\ (\sigma,\mu) \vDash \sigma(a + \mathtt{f}_i), (\mathtt{f}_0 + \ldots + \mathtt{f}_n)^\star, a'}{(\sigma,\mu) \vDash a, (\mathtt{f}_0 + \ldots + \mathtt{f}_n)^\star, a'}$$

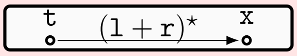
$$\frac{}{(\sigma,\mu) \vDash a, \epsilon, a} \qquad \frac{\exists a'',\ (\sigma,\mu) \vDash a, \mathrm{e}, a'' \wedge (\sigma,\mu) \vDash a'', \mathrm{e}', a'}{(\sigma,\mu) \vDash a, \mathrm{e} \cdot \mathrm{e}', a'}$$

Then, the concretization of silhouettes is defined as:

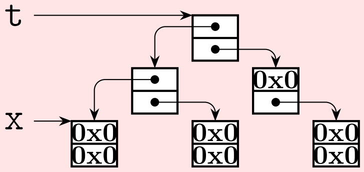
$$\gamma_{\overline{\mathfrak{S}}}(\overline{\mathfrak{s}}) = \{(\sigma, \mu_{\lceil \mathcal{X}}) \mid \forall (\overline{v}, \mathrm{e}, \overline{v}') \in \overline{\mathfrak{s}}, (\sigma,\mu) \vDash (\overline{v}, \mathrm{e}, \overline{v}')\}$$

Example 8.1 (Silhouette). Let us consider the silhouette shown below with its graphical representation on the right:

$$\overline{\mathfrak{s}} = \{(\mathtt{t}, (\mathtt{l} + \mathtt{r})^\star, \mathtt{x})\}$$



Its concretization includes memory states such as the following memory state:



8.1.2 Computation of Silhouettes

To allow silhouettes to assist in clumping abstract states, it is crucial to have an efficient way to compute them. First, empty regions, equalities/disequalities to **0x0**, and full inductive predicates (of the form $\alpha \cdot \mathbf{ind}$) do not contribute to the silhouette. Second, a points-to edge $\overline{v} \cdot \mathtt{f} \mapsto \overline{v}'$ simply contributes an edge $(\overline{v}, \mathtt{f}, \overline{v}')$. Last, a segment predicate $\overline{v} \cdot \mathbf{ind} \mathrel{*\!\!=} \overline{v}' \cdot \mathbf{ind}$ contributes an edge $(\overline{v}, \overline{\mathbf{path}}(\mathbf{ind}), \overline{v}')$, where $\overline{\mathbf{path}}(\mathbf{ind})$ denotes the set of paths that can be induced by a segment of inductive definition **ind**: in the case of the **tree** definition shown in Section 7.2, $\overline{\mathbf{path}}(\mathbf{tree}) = (\mathtt{l} + \mathtt{r})^\star$. Moreover, the silhouette of an abstract state is obtained by collecting the contribution of each region predicate.

Definition 8.3 (Silhouette computation). *The silhouette of an abstract state is defined as a set of edges computed by the translation function* Π *defined by:*

$$\begin{array}{ll}
\Pi(\overline{v} \odot \boldsymbol{\mathit{0x0}}) = \emptyset & \Pi(\overline{v} = \overline{v}') = \{(\overline{v}, \epsilon, \overline{v}')\} \\
\Pi(\mathbf{emp}) = \emptyset & \Pi(\overline{v} \cdot \mathtt{f} \mapsto \overline{v}') = \{(\overline{v}, \mathtt{f}, \overline{v}')\} \\
\Pi(\mathbf{ind}(\overline{v})) = \emptyset & \Pi(\mathbf{ind}(\overline{v}, \overline{v}')) = \{(\overline{v}, \overline{\mathbf{path}}(\mathbf{ind}), \overline{v}')\} \\
\multicolumn{2}{l}{\Pi(\overline{\mathrm{p}}_0 * \ldots * \overline{\mathrm{p}}_k \wedge \overline{\mathrm{n}}_0 \wedge \ldots \wedge \overline{\mathrm{n}}_l) =} \\
\multicolumn{2}{r}{\Pi(\overline{\mathrm{p}}_0) \cup \ldots \cup \Pi(\overline{\mathrm{p}}_n) \cup \Pi(\overline{\mathrm{n}}_0) \cup \ldots \cup \Pi(\overline{\mathrm{n}}_n)}
\end{array}$$

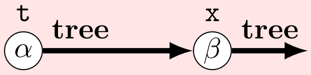
This translation function Π is sound since all concrete states described by the abstract state $\overline{\mathrm{m}}$ are also described by its silhouette $\Pi(\overline{\mathrm{m}})$.

Theorem 8.1 (Soundness of silhouette computation). *The silhouette translation function* Π *is* sound*: for each memory state* $\overline{\mathrm{m}}$*,*

$$\gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}) \subseteq \gamma_{\overline{\mathfrak{S}}}(\Pi(\overline{\mathrm{m}}))$$

Proof. The proof proceeds by induction over abstract states, and follows the definition of Π. The cases of equality and disequality constraints, empty regions, points-to regions, and inductive predicates are trivial. The case of segments requires unrolling their concretization which is itself based on a fix-point. Finally, the case of a conjunction of predicates follows from the fact that the semantics of a silhouette seen as a set of constraints is the intersection of the concretization of each individual constraint. ■

Example 8.2 (Silhouette computation). Let us consider the abstract state $\overline{\mathrm{m}}$ below:



We have $\Pi(\alpha \cdot \mathbf{tree} \mathrel{*\!=} \beta \cdot \mathbf{tree}) = \{(\alpha, (\mathtt{l} + \mathtt{r})^*, \beta)\}$ and $\Pi(\beta \cdot \mathbf{tree}) = \emptyset$. Thus, $\Pi(\overline{\mathrm{m}})$ is the silhouette shown in Example 8.1.

Weakening silhouettes. Silhouettes describe conjunctions of constraints, thus they can be weakened into coarser approximations of sets of memory states, either by dropping or by weakening some constraints. As only a part of silhouette nodes may play a special role, we define here a weakening of silhouettes by restricting silhouettes to a subset of nodes.

Definition 8.4 (Silhouette restriction). *Given a silhouette* $\overline{\mathfrak{s}} = (\mathrm{N}, \mathrm{E})$ *and a set of nodes* $\mathrm{N}' \subseteq \mathrm{N}$*, the* restriction *of* $\overline{\mathfrak{s}}$ *to* N'*, denoted as* $\overline{\mathfrak{s}}_{\lceil \mathrm{N}'}$*, is defined as* $(\mathrm{N}', \mathrm{E}')$*, i.e., the set of nodes* N' *and the set of edges* E' *obtained as acyclic concatenations of edges of* $\overline{\mathfrak{s}}$ *forming paths from* N' *to* N'*:*

$$\forall \overline{v}_0, \overline{v}_k \in \mathrm{N}', (\overline{v}_0, \mathrm{e}, \overline{v}_k) \in \mathrm{E}' \iff \exists (\overline{v}_0, \mathrm{e}_1, \overline{v}_1), \ldots, (\overline{v}_{k-1}, \mathrm{e}_k, \overline{v}_k) \in \mathrm{E}, \mathrm{e} = \mathrm{e}_1 \cdot \ldots \cdot \mathrm{e}_k$$

This weakening effectively allows us to ignore some nodes. For instance, if $\mathbb{X} \subseteq \mathcal{X}$ is a set of variables that play a special role, then $\Pi(\overline{\mathrm{m}})_{\lceil \mathbb{X}}$ is a silhouette of $\overline{\mathrm{m}}$ that will only retain information about the variables in $\mathbb{X}$. We note this silhouette $\Pi(\overline{\mathrm{m}}, \mathbb{X})$.

Example 8.3 (Silhouette restriction). Let us consider the abstract state shown in Example 8.2, $\Pi(\overline{\mathrm{m}}, \{\mathtt{t}\})$ is $\emptyset$ and $\Pi(\overline{\mathrm{m}}, \{\mathtt{x}\})$ is $\emptyset$.

Theorem 8.2 (Soundness of silhouette restriction). *Given a silhouette* $\bar{\mathfrak{s}} = (\mathrm{N}, \mathrm{E})$ *and a set of nodes* $\mathrm{N}' \subseteq \mathrm{N}$*,*

$$\gamma_{\mathfrak{S}}(\bar{\mathfrak{s}}) \subseteq \gamma_{\mathfrak{S}}(\bar{\mathfrak{s}}_{\lceil \mathrm{N}'})$$

Proof. Based on the concretization rules of Definition 8.2, We can easily prove that:

$$\forall (\sigma, \mu) \vDash \bar{\mathfrak{s}}, \forall (\overline{v}, \mathrm{e}, \overline{v}') \in \bar{\mathfrak{s}}_{\lceil \mathrm{N}'}, (\sigma, \mu) \vDash (\overline{v}, \mathrm{e}, \overline{v}')$$

■

8.2 Silhouette-based Weak Entailment Checking

In this section, we study the relationship between entailment checking of abstract states and the silhouettes, and we hope to improve the efficiency of the entailment checking of abstract states with silhouette entailment checking.

8.2.1 Silhouette Entailment Check

Before introducing silhouette entailment checking, let us first recall entailment check for abstract states.

Entailment check for abstract states. As introduced in Section 6.3.1, the entailment checking of abstract states $\sqsubseteq_{\overline{\mathcal{M}_\omega}}$ [BCO05, DOY06, CRN07] is usually implemented as a proof search system based on sets of proof rules that establish inclusion locally. The proof rules shown below (for clarity, numerical constraints are elided) are able to express separation, reflexivity, and unfolding of inductive summary predicates in general:

$$\frac{\overline{\mathrm{m}}_0 \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{\mathrm{m}}'_0 \quad \overline{\mathrm{m}}_1 \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{\mathrm{m}}'_1}{\overline{\mathrm{m}}_0 * \overline{\mathrm{m}}_1 \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{\mathrm{m}}'_0 * \overline{\mathrm{m}}'_1} \qquad \mathbf{i-sep}$$

$$\frac{\overline{\mathrm{m}} \text{ is of the form } \overline{v} \cdot \mathtt{f} \mapsto \overline{v}' \text{ or } \overline{v} \cdot \mathbf{ind} \text{ or } \overline{v} \cdot \mathbf{ind} *\!= \overline{v}' \cdot \mathbf{ind}}{\overline{\mathrm{m}} \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{\mathrm{m}}} \qquad \mathbf{i-id}$$

$$\frac{\overline{\mathrm{m}}' \text{ is of the form } \overline{v} \cdot \mathbf{ind} \text{ or } \overline{v} \cdot \mathbf{ind} *\!= \overline{v}' \cdot \mathbf{ind} \text{ and } \overline{\mathrm{m}}' \xrightarrow{\text{unfold}} \overline{\mathrm{m}}}{\overline{\mathrm{m}} \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{\mathrm{m}}'} \qquad \mathbf{i-uf}$$

$$\frac{\overline{\mathrm{m}} \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{v}' \cdot \mathbf{ind}}{\overline{v} \cdot \mathbf{ind} *\!= \overline{v}' \cdot \mathbf{ind} * \overline{\mathrm{m}} \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{v} \cdot \mathbf{ind}} \qquad \mathbf{i-segind}$$

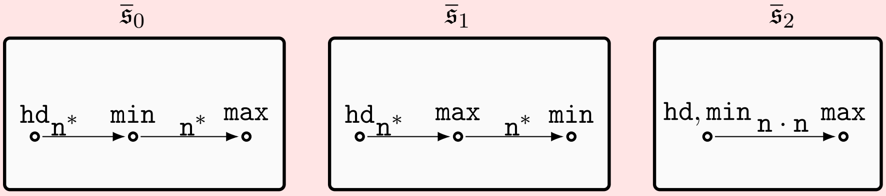
The entailment checking algorithm is in general *sound* and *incomplete*: when $\sqsubseteq_{\overline{\mathcal{M}_\omega}} (\overline{\mathrm{m}}_0, \overline{\mathrm{m}}_1)$ returns **true** (denoted as $\overline{\mathrm{m}}_0 \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{\mathrm{m}}_1$), then $\gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}_0) \subseteq \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}_1)$, yet the reverse implication does not hold in general. Indeed, complete proof search algorithms with backtracking are expensive and generally can not be ensured in the presence of more complex sets of numerical constraints [CRN07] than in the restricted set of abstract states used here. Moreover, when the inclusion does not hold, i.e., $\gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}_0) \nsubseteq \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}_1)$, which happens very often in the analysis, the inclusion checking algorithm still needs to apply a set of proof rules to return **false**, which can be costly.

Silhouette entailment check. As an alternative to the abstract state entailment check, we propose to first use a weaker and cheaper entailment check on silhouettes, which is based on a classical inclusion of constraints. We let $\mathscr{L}(\mathrm{e})$ denote the language of e.

Definition 8.5 (Silhouette entailment check). *Let $\bar{\mathfrak{s}}_0, \bar{\mathfrak{s}}_1 \in \overline{\mathfrak{S}}$. We let $\sqsubseteq_{\overline{\mathfrak{S}}} (\bar{\mathfrak{s}}_0, \bar{\mathfrak{s}}_1)$ return* **true** *(denoted as $\bar{\mathfrak{s}}_0 \sqsubseteq_{\overline{\mathfrak{S}}} \bar{\mathfrak{s}}_1$) if and only if for each edge $(\overline{v}, \mathrm{e}, \overline{v}')$ of $\bar{\mathfrak{s}}_1$ there exists a (possibly empty) sequence of edges $(\overline{v}_0, \mathrm{e}_1, \overline{v}_1), \ldots, (\overline{v}_{k-1}, \mathrm{e}_k, \overline{v}_k)$ in $\bar{\mathfrak{s}}_0$ such that $\overline{v}_0 = \overline{v}$, $\overline{v}_k = \overline{v}'$ and $\mathscr{L}(\mathrm{e}_1 \cdot \ldots \cdot \mathrm{e}_k) \subseteq \mathscr{L}(\mathrm{e})$ (or, if the sequence is empty, $\epsilon \in \mathscr{L}(\mathrm{e})$).*

We note that $\sqsubseteq_{\overline{\mathfrak{S}}}$ forms an order relation, i.e., given $\bar{\mathfrak{s}}_0 \sqsubseteq_{\overline{\mathfrak{S}}} \bar{\mathfrak{s}}_1$ and $\bar{\mathfrak{s}}_1 \sqsubseteq_{\overline{\mathfrak{S}}} \bar{\mathfrak{s}}_2$, we can get $\bar{\mathfrak{s}}_0 \sqsubseteq_{\overline{\mathfrak{S}}} \bar{\mathfrak{s}}_2$.

Example 8.4 (Silhouette entailment check). Let us consider the silhouettes below drawn from the analysis of the program searching for the minimum and maximum list elements shown in Figure 1.8:



We have: $\sqsubseteq_{\overline{\mathfrak{S}}} (\bar{\mathfrak{s}}_0, \bar{\mathfrak{s}}_1) =$ **false**, $\sqsubseteq_{\overline{\mathfrak{S}}} (\bar{\mathfrak{s}}_1, \bar{\mathfrak{s}}_0) =$ **false**, $\sqsubseteq_{\overline{\mathfrak{S}}} (\bar{\mathfrak{s}}_2, \bar{\mathfrak{s}}_0) =$ **true**.

Theorem 8.3 (Soundness). *The entailment check $\sqsubseteq_{\overline{\mathfrak{S}}}$ is sound: given $\bar{\mathfrak{s}}_0, \bar{\mathfrak{s}}_1 \in \overline{\mathfrak{S}}$,*

$$\text{if } \bar{\mathfrak{s}}_0 \sqsubseteq_{\overline{\mathfrak{S}}} \bar{\mathfrak{s}}_1, \text{ then } \gamma_{\overline{\mathfrak{S}}}(\bar{\mathfrak{s}}_0) \subseteq \gamma_{\overline{\mathfrak{S}}}(\bar{\mathfrak{s}}_1)$$

Proof. Let us assume $\sqsubseteq_{\overline{\mathfrak{S}}} (\bar{\mathfrak{s}}_0, \bar{\mathfrak{s}}_1) =$ **true** and $(\overline{v}, \mathrm{e}, \overline{v}')$ is an edge of $\bar{\mathfrak{s}}_1$. Then, according to Definition 8.5, there exists a (possibly empty) sequence of edges $(\overline{v}_0, \mathrm{e}_1, \overline{v}_1), \ldots, (\overline{v}_{k-1}, \mathrm{e}_k, \overline{v}_k)$ in $\bar{\mathfrak{s}}_0$ such that $\overline{v}_0 = \overline{v}$, $\overline{v}_k = \overline{v}'$ and $\mathscr{L}(\mathrm{e}_1 \cdot \ldots \cdot \mathrm{e}_k) \subseteq \mathscr{L}(\mathrm{e})$ (or, if the sequence is empty, $\epsilon \in \mathscr{L}(\mathrm{e})$). By the definition of the silhouette concretization, the concretization of $\{(\overline{v}_0, \mathrm{e}_1, \overline{v}_1), \ldots, (\overline{v}_{k-1}, \mathrm{e}_k, \overline{v}_k)\}$ is included in that of $(\overline{v}, \mathrm{e}, \overline{v}')$. As the concretization of $\bar{\mathfrak{s}}_1$ is the intersection of the concretization of its edges, we can thus derive the theorem. ■

8.2.2 Weak Entailment Checking

The most important result on silhouette entailment checking is that it is weaker than abstract states entailment checking:

Theorem 8.4 (Weak entailment). *Let* $\overline{\mathrm{m}}_0, \overline{\mathrm{m}}_1 \in \overline{\mathcal{M}_\omega}$ *and* $\bar{\mathfrak{s}}_0 = (\mathrm{N}_0, \mathrm{E}_0) = \Pi(\overline{\mathrm{m}}_0)$, $\bar{\mathfrak{s}}_1 = (\mathrm{N}_1, \mathrm{E}_1) = \Pi(\overline{\mathrm{m}}_1)$, *then:*

$$\overline{\mathrm{m}}_0 \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{\mathrm{m}}_1 \Longrightarrow \bar{\mathfrak{s}}_0 \sqsubseteq_{\overline{\mathfrak{S}}} \bar{\mathfrak{s}}_1$$

Moreover, let $\mathrm{N}' \subseteq \mathrm{N}_0 \land \mathrm{N}' \subseteq \mathrm{N}_1$, *then:*

$$\overline{\mathrm{m}}_0 \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{\mathrm{m}}_1 \Longrightarrow \bar{\mathfrak{s}}_{0\lceil \mathrm{N}'} \sqsubseteq_{\overline{\mathfrak{S}}} \bar{\mathfrak{s}}_{1\lceil \mathrm{N}'}$$

Proof. We can prove that $\overline{\mathrm{m}}_0 \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{\mathrm{m}}_1 \Longrightarrow \bar{\mathfrak{s}}_0 \sqsubseteq_{\overline{\mathfrak{S}}} \bar{\mathfrak{s}}_1$ by induction on the derivation rules of $\sqsubseteq_{\overline{\mathcal{M}_\omega}}$:

- Case of the **i-id** rule:

  $$\frac{\overline{\mathrm{m}} \text{ is of the form } \overline{v} \cdot \mathtt{f} \mapsto \overline{v}' \text{ or } \overline{v} \cdot \mathbf{ind} \text{ or } \overline{v} \cdot \mathbf{ind} \mathrel{*\!\!=} \overline{v}' \cdot \mathbf{ind}}{\overline{\mathrm{m}} \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{\mathrm{m}}}$$

  Since both sides are equal, their image by Π are equal too.

- Case of the **i-uf** rule (for inductive predicates):

  $$\frac{\overline{v} \cdot \mathbf{ind} \xrightarrow{\text{unfold}} \overline{\mathrm{m}}}{\overline{\mathrm{m}} \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{v} \cdot \mathbf{ind}}$$

  The right-hand side is a full inductive predicate, which thus has a silhouette with no edge, which concretizes in the set of all pairs made up of a state and a valuation.
- Case of the **i-uf** rule (for segment predicates):

  $$\frac{\overline{v} \cdot \mathbf{ind} \mathrel{*\!\!=} \overline{v}' \cdot \mathbf{ind} \xrightarrow{\text{unfold}} \overline{\mathrm{m}}}{\overline{\mathrm{m}} \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{v} \cdot \mathbf{ind} \mathrel{*\!\!=} \overline{v}' \cdot \mathbf{ind}}$$

  By the definition of the unfolding of a segment and of $\overline{\mathbf{path}}$, $\overline{\mathrm{m}}$ is a memory state fragment such that there exists a path from $\overline{v}$ to $\overline{v}'$ described by an expression the whose language is included in that of $\overline{\mathbf{path}}(\mathbf{ind})$. Thus, $\Pi(\overline{\mathrm{m}}) \sqsubseteq_{\overline{\mathfrak{S}}} \{(\overline{v}, \overline{\mathbf{path}}(\mathbf{ind}), \overline{v}')\}$.

- Case of the **i-segind** rule:

  $$\frac{\overline{\mathrm{m}} \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{v}' \cdot \mathbf{ind}}{\overline{v} \cdot \mathbf{ind} \mathrel{*\!\!=} \overline{v}' \cdot \mathbf{ind} * \overline{\mathrm{m}} \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{v} \cdot \mathbf{ind}}$$

  The right-hand side is a full inductive predicate, which thus has a silhouette with no edge, which concretizes in the set of all pairs made up of a state and a valuation.

```
input:    disjunctive abstract state $\overline{\mathrm{m}}_0 \vee \ldots \vee \overline{\mathrm{m}}_n$
          disjunctive abstract state $\overline{\mathrm{m}}'_0 \vee \ldots \vee \overline{\mathrm{m}}'_k$
          set of program variables $\mathbb{X}$
output:   true or false
$\overline{\mathfrak{s}}_0 \leftarrow \Pi(\overline{\mathrm{m}}_0, \mathbb{X}); \ldots; \overline{\mathfrak{s}}_n \leftarrow \Pi(\overline{\mathrm{m}}_n, \mathbb{X})$;
$\overline{\mathfrak{s}}'_0 \leftarrow \Pi(\overline{\mathrm{m}}'_0, \mathbb{X}); \ldots; \overline{\mathfrak{s}}'_k \leftarrow \Pi(\overline{\mathrm{m}}'_k, \mathbb{X})$;
for $p = 0$ to $n$
    $is\_leq$ ← false; $q \leftarrow 0$;
    while $q <= k$ && $is\_leq$ == false
        if $\sqsubseteq_{\overline{\mathfrak{S}}} (\overline{\mathfrak{s}}_p, \overline{\mathfrak{s}}'_q)$ then $is\_leq = \sqsubseteq_{\overline{\mathcal{M}_\omega}} (\overline{\mathrm{m}}_p, \overline{\mathrm{m}}'_q)$;
        $q++$;
    if $is\_leq$ == false then return false;
return true;
```

Figure 8.2: Weak entailment check algorithm $\sqsubseteq_{\overline{\mathcal{D}}}$.

- Case of the **i-sep** rule:

$$\frac{\overline{\mathrm{m}}_0 \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{\mathrm{m}}'_0 \quad \overline{\mathrm{m}}_1 \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{\mathrm{m}}'_1}{\overline{\mathrm{m}}_0 * \overline{\mathrm{m}}_1 \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{\mathrm{m}}'_0 * \overline{\mathrm{m}}'_1}$$

  Applying the induction hypothesis to the proofs of $\overline{\mathrm{m}}_0 * \overline{\mathrm{m}}_1$ and $\overline{\mathrm{m}}'_0 * \overline{\mathrm{m}}'_1$, we get $\Pi(\overline{\mathrm{m}}_0) \sqsubseteq_{\overline{\mathfrak{S}}} \Pi(\overline{\mathrm{m}}_1))$ and $\Pi(\overline{\mathrm{m}}'_0) \sqsubseteq_{\overline{\mathfrak{S}}} \Pi(\overline{\mathrm{m}}'_1))$. Since $\Pi(\overline{\mathrm{m}}_0 * \overline{\mathrm{m}}_1) = \Pi(\overline{\mathrm{m}}_0) \cup \Pi(\overline{\mathrm{m}}_1)$ and $\Pi(\overline{\mathrm{m}}'_0 * \overline{\mathrm{m}}'_1) = \Pi(\overline{\mathrm{m}}'_0) \cup \Pi(\overline{\mathrm{m}}'_1)$ and by definition of $\sqsubseteq_{\overline{\mathfrak{S}}}$, $\Pi(\overline{\mathrm{m}}_0 * \overline{\mathrm{m}}_1) \sqsubseteq_{\overline{\mathfrak{S}}} \Pi(\overline{\mathrm{m}}'_0 * \overline{\mathrm{m}}'_1)$.

Then, based on Definition 8.4 and Definition 8.5, given $\overline{\mathfrak{s}}_0 \sqsubseteq_{\overline{\mathfrak{S}}} \overline{\mathfrak{s}}_1$, we can easily prove that for every edge $(\overline{v}_0, \mathrm{e}, \overline{v}_1)$ of $\overline{\mathfrak{s}}_{1\lceil \mathrm{N}'}$, there exists $(\overline{v}_0, \mathrm{e}', \overline{v}_1)$ of $\overline{\mathfrak{s}}_{0\lceil \mathrm{N}'}$, such that $\mathscr{L}(\mathrm{e}') \subseteq \mathscr{L}(\mathrm{e})$. That is, $\overline{\mathfrak{s}}_{0\lceil \mathrm{N}'} \sqsubseteq_{\overline{\mathfrak{S}}} \overline{\mathfrak{s}}_{1\lceil \mathrm{N}'}$. ■

Weak entailment check algorithm. From this theorem follows the core principle of our approach: while the most direct way to check if $\gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}_0) \subseteq \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}_1)$ consists in applying the (fairly expensive) $\sqsubseteq_{\overline{\mathcal{M}_\omega}} (\overline{\mathrm{m}}_0, \overline{\mathrm{m}}_1)$ proof search algorithm, an alternative approach consists in computing $\sqsubseteq_{\overline{\mathfrak{S}}} (\Pi(\overline{\mathrm{m}}_0), \Pi(\overline{\mathrm{m}}_1))$ first and computing $\sqsubseteq_{\overline{\mathcal{M}_\omega}}$ only when $\sqsubseteq_{\overline{\mathfrak{S}}}$ returns **true**. Indeed, if $\sqsubseteq_{\overline{\mathfrak{S}}}$ returns **false**, then $\sqsubseteq_{\overline{\mathcal{M}_\omega}}$ will also definitely return **false**, though at a much higher computational cost. Following the same principle, in Figure 8.2 we show the silhouette-based weak entailment check algorithm on disjunctive abstract states.

Example 8.5 (Entailment). Figure 8.3 shows a few abstract states and their silhouettes drawn from the example given in Chapter 7. Let us consider checking the disjunctive abstract state $\overline{\mathrm{d}} = \overline{\mathrm{m}}_0 \vee \overline{\mathrm{m}}_1 \vee \overline{\mathrm{m}}_2$ is included in itself according to the algorithm shown in Figure 8.2.

First, as $\sqsubseteq_{\overline{\mathfrak{S}}} (\overline{\mathfrak{s}}_0, \overline{\mathfrak{s}}_0) =$ **true**, we thus need to prove $\sqsubseteq_{\overline{\mathcal{M}_\omega}} (\overline{\mathrm{m}}_0, \overline{\mathrm{m}}_0) =$ **true**. Then, as $\sqsubseteq_{\overline{\mathfrak{S}}} (\overline{\mathfrak{s}}_1, \overline{\mathfrak{s}}_0) =$ **false**, we can omit a complex computation of $\sqsubseteq_{\overline{\mathcal{M}_\omega}} (\overline{\mathrm{m}}_1, \overline{\mathrm{m}}_0)$, which will definitely return **false**. Moreover, as $\sqsubseteq_{\overline{\mathfrak{S}}} (\overline{\mathfrak{s}}_2, \overline{\mathfrak{s}}_0) =$ **false** and $\sqsubseteq_{\overline{\mathfrak{S}}} (\overline{\mathfrak{s}}_2, \overline{\mathfrak{s}}_1) =$ **false**, we can also omit complex computations of $\sqsubseteq_{\overline{\mathcal{M}_\omega}} (\overline{\mathrm{m}}_2, \overline{\mathrm{m}}_0)$ and $\sqsubseteq_{\overline{\mathcal{M}_\omega}} (\overline{\mathrm{m}}_2, \overline{\mathrm{m}}_1)$. Indeed, with the

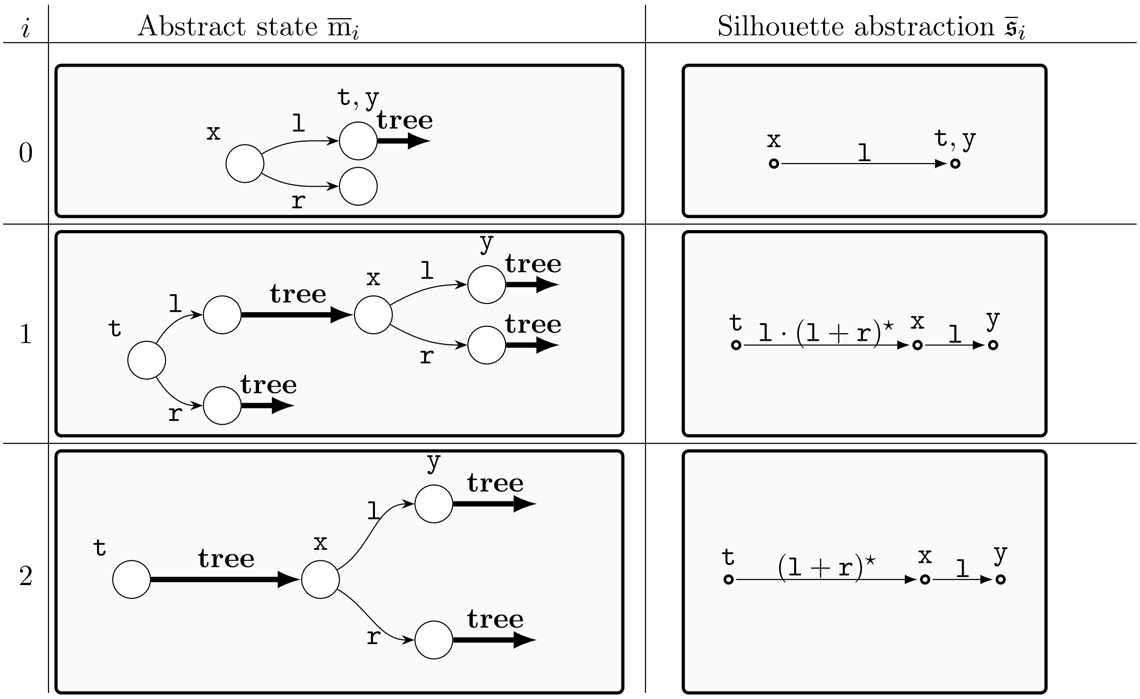

i
Abstract state $\overline{\mathrm{m}}_i$
Silhouette abstraction $\bar{\mathfrak{s}}_i$
0
x
l
t, y
tree
r
x
l
t, y
1
t
l
tree
x
l
y
tree
tree
r
tree
r
t
$\mathtt{l} \cdot (\mathtt{l}+\mathtt{r})^\star$
x
l
y
2
t
tree
x
l
y
tree
r
tree
t
$(\mathtt{l}+\mathtt{r})^\star$
x
l
y

Figure 8.3: Abstract states and silhouettes

silhouette entailment checking, we only need to compute that $\sqsubseteq_{\overline{\mathcal{M}_\omega}} (\overline{\mathrm{m}}_0, \overline{\mathrm{m}}_0)$, $\sqsubseteq_{\overline{\mathcal{M}_\omega}} (\overline{\mathrm{m}}_1, \overline{\mathrm{m}}_1)$ and $\sqsubseteq_{\overline{\mathcal{M}_\omega}} (\overline{\mathrm{m}}_2, \overline{\mathrm{m}}_2)$, while a normal disjunctive entailment checking will require six times computations of the entailment checking of abstract states.

Chapter 9

Silhouette-Guided Joining of Abstract Memory States

This chapter shows that silhouettes can be used to make the joining procedure of abstract memory states more precise. Specifically, Section 9.1 first recalls the existing abstract states join procedure and discusses possible loss of precision in the joining. Then, Section 9.2 introduces silhouette guided abstract states join.

9.1 Existing Abstract States Join Procedure and Precision Loss

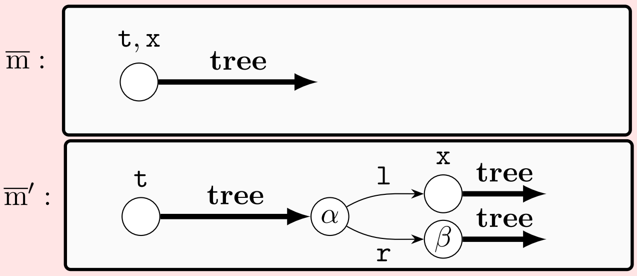
Before introducing the silhouette-guided joining of abstract states, let us first recall the main principles of the existing abstract state joining operator $\sqcup_{\overline{\mathcal{M}_\omega}}$ presented in Section 6.3.2. Based on separation [Rey02] and local reasoning, the joining operator $\sqcup_{\overline{\mathcal{M}_\omega}}$ applies two main kinds of rules on abstract shapes, as presented in Figure 6.6:

- **Weakening guided by existing region predicates.** The first set of rules (e.g., **j-pt, j-ind, j-weak**) searches for cases where $\overline{\mathrm{g}} \sqsubseteq_{\overline{\mathcal{G}}} \overline{\mathrm{g}}'$ (resp. $\overline{\mathrm{g}}' \sqsubseteq_{\overline{\mathcal{G}}} \overline{\mathrm{g}}$) holds so that a valid choice for $\overline{\mathrm{g}}^{\sqcup}$ is $\overline{\mathrm{g}}'$ (resp., $\overline{\mathrm{g}}$). The most common case is when $\overline{\mathrm{g}} = \overline{\mathrm{g}}'$. Another important case is when $\overline{\mathrm{g}}' = \alpha \cdot \mathbf{ind}$ and $\overline{\mathrm{g}} \sqsubseteq_{\overline{\mathcal{G}}} \alpha \cdot \mathbf{ind}$; then $\overline{\mathrm{g}}$ gets weakened into a summary predicate, that was already present in $\overline{\mathrm{g}}'$.

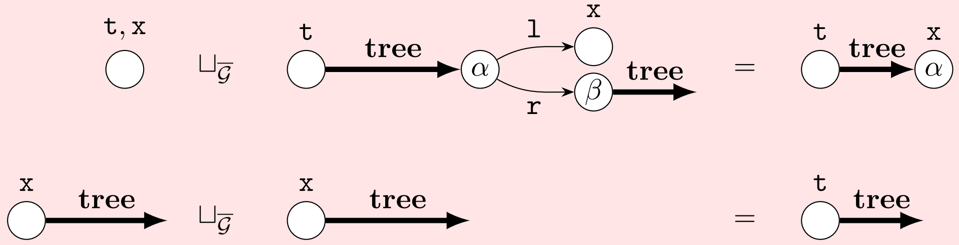
- **Synthesis of summary predicates.** The second set of rules (e.g., **j-intro**) *synthesizes* new summary predicates, in order to weaken specific patterns. For instance, the introduction of a segment predicate is a particular case of summary predicate synthesis, that weakens an empty region into a segment:

$$\left.\begin{array}{r} \overline{\mathrm{g}}_i = \mathbf{emp} \wedge \alpha = \beta) \\ \overline{\mathrm{g}}'_i \sqsubseteq_{\overline{\mathcal{G}}} \alpha \cdot \mathbf{ind} \mathrel{*\!=} \beta \cdot \mathbf{ind} \end{array}\right\} \rightsquigarrow \overline{\mathrm{g}}_i^{\sqcup} = \alpha \cdot \mathbf{ind} \mathrel{*\!=} \beta \cdot \mathbf{ind}$$

Example 9.1 (Abstract states join). As an example, let us consider the abstract states below:



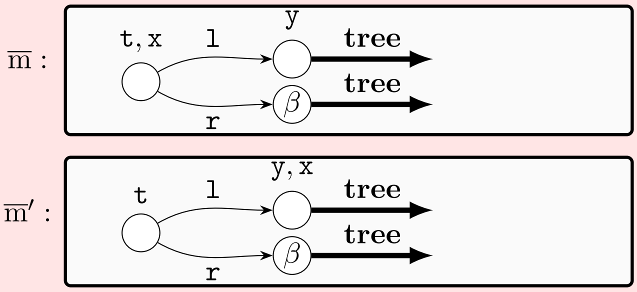
We note that in $\overline{\mathrm{m}}$, $\mathtt{t} = \mathtt{x}$. Then, both abstract states $\overline{\mathrm{m}}$, $\overline{\mathrm{m}}'$ can be split into two regions, and the following joining rules are applied to each region:



Therefore, the two states $\overline{\mathrm{m}}$, $\overline{\mathrm{m}}'$ can be joined into $\mathtt{t} \cdot \mathbf{ind} \mathrel{*}= \mathtt{x} \cdot \mathbf{ind} * \mathtt{x} \cdot \mathbf{ind}$.

Challenges. In practice, however, joining rules are applied in sequential orders and it is often the case that more than one rule can be applied at one point. Yet, different rule application orders may produce very different results. However, determining which rule, when applied, can lead to a precise joining results in huge difficulty in the abstract join because of the large search space that needs to be examined. Moreover, in certain cases, there exists no rule order that produces a precise common over-approximation. Even when a solution does exist, it may be non-unique, and there may be no universally best solution, as illustrated by the following example.

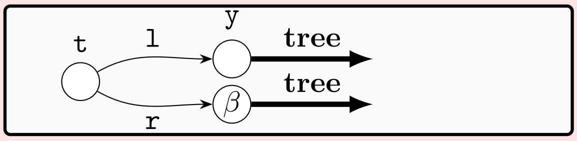
Example 9.2 (Non-unique joining rule order). We let $\overline{\mathrm{m}}, \overline{\mathrm{m}}'$ be defined by:



There are several possible joinings, two of which are presented below:

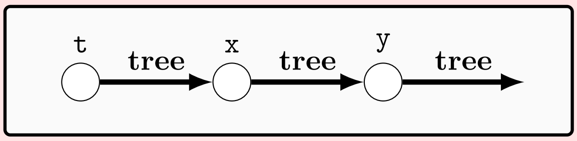
- first pairing each of the four edges of $\overline{\mathrm{m}}$ with an edge of $\overline{\mathrm{m}}'$ by applying the **j-pt** rule produces a joining result with very precise information about `t`, `y`, but which discards all information about `x`:

$\overline{\mathrm{m}}_1^{\sqcup}$:



- synthesizing a segment predicate between `t` and `x` by applying the **j-intro** rule first and then synthesizing another segment predicate between `x` and `y` by applying the same rule produces a joining result, where the fact that `x`, `y` are cursors in the tree pointed to by `t` is preserved, but the information that `t` · `l` points to `y` is lost:

$\overline{\mathrm{m}}_2^{\sqcup}$:



In this situation, to make the best joining choice, the analysis should take into account future uses of `t`, `x`, `y`. If `x` is unused after the point in the program where the joining is performed, the first joining is the best. If the fact that `x` points somewhere in the tree matters, the second joining is better. Finally, if the fact that `y` is a child of `t` and the fact that `x` points somewhere in the tree both matter, then no precise joining result exists.

9.2 Silhouette Guided Abstract States Joining

In this section, we first define silhouette joining in Section 9.2.1 and in Section 9.2.2 we demonstrate that silhouettes can help to determine which rule to apply during abstract state join for a precise joining result. Then, in Section 9.2.3, we show that silhouettes can be restricted so as to be more concise and relevant to guiding abstract states join, by considering the analysis goal.

9.2.1 Silhouettes Joining

We now define a join operator over silhouettes that takes any two silhouettes as inputs and returns their over-approximation.

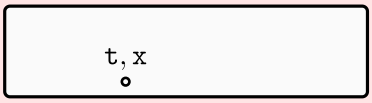
Definition 9.1 (Silhouette join). *Let us first define the* join of silhouette edges. *Silhouette edges* $(\overline{v}_0, \mathrm{e}, \overline{v}_1)$ *and* $(\overline{v}_0, \mathrm{e}', \overline{v}_1)$ *can be joined as the edge* $(\overline{v}_0, \mathrm{e}^{\sqcup}, \overline{v}_1)$*, where* $\mathrm{e}^{\sqcup}$ *keeps the same left side as* e *and* e' *and joins the other part into the smallest and more general regular expression of the form* $(\mathtt{f}_0 + \ldots + \mathtt{f}_k)^{\star}$.

Then, given two silhouettes $\overline{\mathfrak{s}} = (\mathrm{N}, \mathrm{E}), \overline{\mathfrak{s}}' = (\mathrm{N}', \mathrm{E}')$*, we define their* silhouette join *noted* $\sqcup_{\overline{\mathbb{S}}}(\overline{\mathfrak{s}}, \overline{\mathfrak{s}}')$ *as the silhouette* $\overline{\mathfrak{s}}^{\sqcup} = (\mathrm{N}^{\sqcup}, \mathrm{E}^{\sqcup})$ *such that* $\mathrm{N}^{\sqcup} = \mathrm{N} \cap \mathrm{N}'$ *and* $\mathrm{E}^{\sqcup}$ *collect the pairwise join of the edges of* $\overline{\mathfrak{s}}_{\lceil \mathrm{N}^{\sqcup}}$ *and of* $\overline{\mathfrak{s}}'_{\lceil \mathrm{N}^{\sqcup}}$.

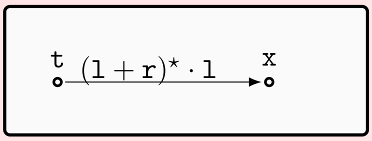
Silhouettes join acts like a "lightweight" abstract states join.

Example 9.3 (Silhouettes join). Let us consider the abstract states $\overline{\mathrm{m}}$ and $\overline{\mathrm{m}}'$ presented in Example 9.1, and let $\mathbb{X} = \{\mathtt{t}, \mathtt{x}\}$, then we obtain the silhouettes below:

$\overline{\mathfrak{s}} = \Pi(\overline{\mathrm{m}}, \widehat{\mathbb{X}}) =$

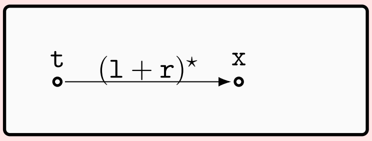


$\overline{\mathfrak{s}}' = \Pi(\overline{\mathrm{m}}', \widehat{\mathbb{X}}) =$



As in $\overline{\mathfrak{s}}$, the access path from $\mathtt{t}$ to $\mathtt{x}$ is ϵ (empty), and in $\overline{\mathfrak{s}}'$, $\mathtt{t}$ can reach $\mathtt{x}$ through access path $(\mathtt{l} + \mathtt{r})^* \cdot \mathtt{l}$, therefore joining the two silhouettes gets the silhouette below, where the access path $(\mathtt{l} + \mathtt{r})^*$ over-approximates ϵ and $(\mathtt{l} + \mathtt{r})^* \cdot \mathtt{l}$:

$\overline{\mathfrak{s}} \sqcup_{\overline{\mathfrak{S}}} \overline{\mathfrak{s}}' =$



Theorem 9.1 (Soundness). *The silhouette join is sound:*

$$\forall \overline{\mathfrak{s}}, \overline{\mathfrak{s}}' \in \overline{\mathfrak{S}},\ \sqcup_{\overline{\mathbb{S}}}(\overline{\mathfrak{s}}, \overline{\mathfrak{s}}') = \overline{\mathfrak{s}}_o \implies \gamma_{\overline{\mathbb{S}}}(\overline{\mathfrak{s}}) \cup \gamma_{\overline{\mathbb{S}}}(\overline{\mathfrak{s}}') \subseteq \gamma_{\overline{\mathbb{S}}}(\overline{\mathfrak{s}}_o)$$

Proof. According to Definition 9.1, for any $(\overline{v}, \mathrm{e}_o, \overline{v}') \in \overline{\mathfrak{s}}_o$, there exists $(\overline{v}, \mathrm{e}, \overline{v}') \in \overline{\mathfrak{s}}$ and $(\overline{v}, \mathrm{e}', \overline{v}') \in \overline{\mathfrak{s}}'$ such that, $\mathscr{L}(\mathrm{e}) \cup \mathscr{L}(\mathrm{e}') \subseteq \mathscr{L}(\mathrm{e}_0)$. Therefore, $\gamma_{\overline{\mathbb{S}}}(\overline{\mathfrak{s}}) \subseteq \gamma_{\overline{\mathbb{S}}}(\overline{\mathfrak{s}}_o) \wedge \gamma_{\overline{\mathbb{S}}}(\overline{\mathfrak{s}}') \subseteq \gamma_{\overline{\mathbb{S}}}(\overline{\mathfrak{s}}_o)$. ■

9.2.2 Guided Abstract States Joining

When two abstract states can be joined into a segment, their silhouette is a refinement of the silhouette of a segment (by Th. 8.4):

Theorem 9.2 (Guided segment synthesis weak characterization). *Let* $\overline{\mathrm{m}}, \overline{\mathrm{m}}'$ *be two abstract states such that* $\overline{\mathrm{m}} \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{v}_0 \cdot \mathbf{ind} \mathrel{*\!\!=} \overline{v}_1 \cdot \mathbf{ind}$ *and* $\overline{\mathrm{m}}' \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{v}_0 \cdot \mathbf{ind} \mathrel{*\!\!=} \overline{v}_1 \cdot \mathbf{ind}$*, then*

$$\begin{aligned}\gamma_{\overline{\mathfrak{S}}}(\mathbf{join}_{\overline{\mathfrak{S}}}(\Pi(\overline{\mathrm{m}}, \{\overline{v}_0, \overline{v}_1\}), \Pi(\overline{\mathrm{m}}', \{\overline{v}_0, \overline{v}_1\}))) \\ \subseteq \gamma_{\overline{\mathfrak{S}}}(\{(\overline{v}_0, \overline{\mathbf{path}}(\mathbf{ind}), \overline{v}_1)\})\end{aligned}$$

Based on the theorem, we derive the semantic guided segment introduction rule below, where the application of the rule does not rely on the syntax of abstract states, but relies on their silhouette join.

Definition 9.2 (Semantic guided segment introduction rule).

$$\begin{array}{ll}
\textit{let } \overline{\mathfrak{s}} = \Pi(\overline{\mathrm{m}}, \{\overline{v}_0, \overline{v}_1\}) & \textit{and } \overline{\mathfrak{s}}' = \Pi(\overline{\mathrm{m}}', \{\overline{v}_0, \overline{v}_1\}) \\
\quad \textit{if} & \sqcup_{\overline{\mathbb{S}}}(\overline{\mathfrak{s}}, \overline{\mathfrak{s}}') = (\mathtt{f}_0 + \ldots + \mathtt{f}_k)^\star \\
\quad \textit{and} & \sqsubseteq_{\overline{\mathbb{S}}} (\sqcup_{\overline{\mathbb{S}}}(\overline{\mathfrak{s}}, \overline{\mathfrak{s}}'), \{(\overline{v}_0, \overline{\mathbf{path}}(\mathbf{ind}), \overline{v}_1)\}) = \mathbf{true}, \\
\qquad \textit{then if} & \sqsubseteq_{\overline{\mathcal{M}_\omega}} (\overline{\mathrm{m}}, \overline{v}_0 \cdot \mathbf{ind} \mathrel{*\!\!=} \overline{v}_1 \cdot \mathbf{ind}) = \mathbf{true} \\
\qquad \textit{and} & \sqsubseteq_{\overline{\mathcal{M}_\omega}} (\overline{\mathrm{m}}', \overline{v}_0 \cdot \mathbf{ind} \mathrel{*\!\!=} \overline{v}_1 \cdot \mathbf{ind}) = \mathbf{true}, \\
\qquad\quad \textit{then,} & \sqcup_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}, \overline{\mathrm{m}}') = \overline{v}_0 \cdot \mathbf{ind} \mathrel{*\!\!=} \overline{v}_1 \cdot \mathbf{ind}
\end{array}$$

Intuitively, this rule will only attempt to synthesize a segment when the join of the silhouettes can be weakened into the silhouette of a segment. Only in that case will it call the shape inclusion checking function $\sqsubseteq_{\overline{\mathcal{M}_\omega}}$ to determine if a segment can be introduced. This rule supersedes the classical syntactic rule and avoids many attempts to run the costly $\sqsubseteq_{\overline{\mathcal{M}_\omega}}$, which would fail (since $\sqsubseteq_{\overline{\mathfrak{S}}}$ provides a cheaper, weak entailment test).

In the following, we let $\sqcup_{\overline{\mathcal{M}_\omega}}$ denote a join operator for abstract states, that takes as input a third argument for a silhouette which it uses as a guide to introduce segments. Given two abstract states and a silhouette, the guided joining operator first attempts to apply the semantic guided segment introduction rule, and then applies the other joining rules, as presented in Figure 6.6.

Theorem 9.3 (Soundness). *The guided joining is sound:*

$$\sqcup_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}, \overline{\mathrm{m}}', \overline{\mathfrak{s}}) = \overline{\mathrm{m}}_o \implies \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}) \subseteq \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}_o) \wedge \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}') \subseteq \gamma_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}_o)$$

Proof. The soundness can be easily proved based on the soundness of each rule. ■

Example 9.4. We consider the abstract states $\overline{\mathrm{m}}, \overline{\mathrm{m}}'$ of Example 9.1. Then:

$$\overline{\mathfrak{s}}^{\sqcup} = \sqcup_{\overline{\mathbb{S}}}(\Pi(\overline{\mathrm{m}}, \{\mathtt{x}, \mathtt{t}\}), \Pi(\overline{\mathrm{m}}', \{\mathtt{x}, \mathtt{t}\})) = \{(\mathtt{t}, (\mathtt{l} + \mathtt{r})^\star, \mathtt{x})\}$$

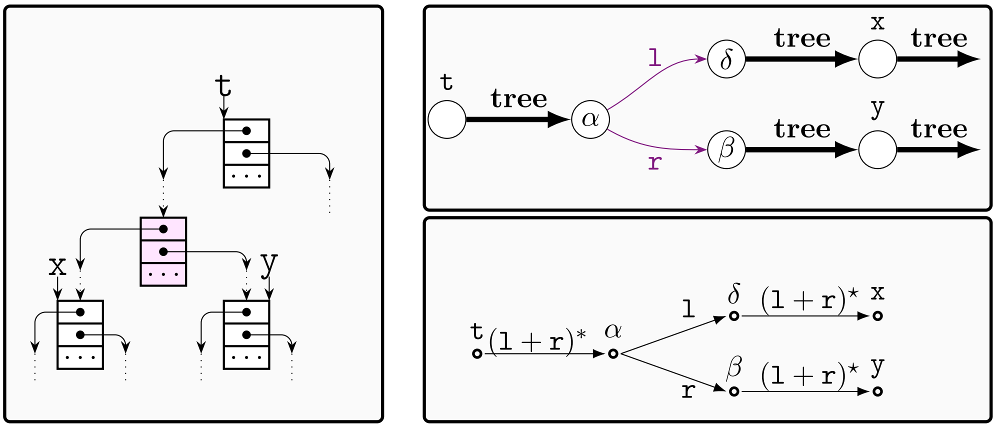
Thus, the semantic guided segment introduction rule can be applied to synthesize a segment predicate between `t` and `x`, and after which a normal joining rule can be applied to match the inductive predicate at `x`:

$$\sqcup_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}, \overline{\mathrm{m}}', \overline{\mathfrak{s}}^{\sqcup}) = \mathtt{t} \cdot \mathbf{tree} \mathrel{*\!\!=} \mathtt{x} \cdot \mathbf{tree} * \mathtt{x} \cdot \mathbf{tree}$$

As another example, let us consider the abstract states $\overline{\mathrm{m}}, \overline{\mathrm{m}}'$ of Example 9.2, where

$$\overline{\mathfrak{s}}^{\sqcup} = \sqcup_{\overline{\mathbb{S}}}(\Pi(\overline{\mathrm{m}}, \{\mathtt{x}, \mathtt{t}, \mathtt{y}\}), \Pi(\overline{\mathrm{m}}', \{\mathtt{x}, \mathtt{t}, \mathtt{y}\})) = \{(\mathtt{t}, \mathtt{l}^\star, \mathtt{x}), (\mathtt{x}, \mathtt{l}^\star, \mathtt{y})\}$$

As $\overline{\mathbf{path}}(\mathbf{tree}) = (\mathtt{l} + \mathtt{r})^*$, thus the guided segment introduction rule can be applied to synthesize a segment predicate between `t` and `x` and a segment predicate between `x` and `y`,



(a) Concrete memory state

(b) Abstract state and silhouette

Figure 9.1: Branching node

after which a normal joining rule can be applied to match the inductive predicate at y:

$$\sqcup_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}, \overline{\mathrm{m}}', \bar{\mathfrak{s}}^{\sqcup}) = \mathtt{t} \cdot \textbf{tree} \ast\!= \mathtt{x} \cdot \textbf{tree} \ast \mathtt{x} \cdot \textbf{tree}$$

We note that the silhouette guided abstract states join could compute more precise join results when the search space for the join is larger, as evaluated in Chapter 11. The syntactic rule-based join that we presented in Figure 6.6 tends to fail in such cases, and that the failure generally stems from a lack of understanding of the semantic properties of the abstract state.

9.2.3 Taking Advantage of the Analysis Goal

Live variables. As remarked in Example 9.2, *live* program variables play a big role in determining whether a joining result is precise or not. The reason is that live variables may be read later in the program being analyzed before they are overwritten. Thus the joining process of abstract states should always try to apply rules that are able to keep precise information of live variables since if, during joining, the information about live variables is lost, then the analysis is very likely to fail. As shown in Section 9.2.2, silhouettes can guide abstract state join to keep important precise information that is relevant to silhouettes. Therefore, we can make silhouettes more concise and relevant by collecting only nodes that are live variables. We note that a standard liveness compiler analysis [App08] can compute a set of live program variables at each program point.

Branching nodes. When abstracting concrete memories, memory nodes that have nested pointers to several sub-structures which can also be accessed by dereferencing different program variables are usually important as such nodes should not be summarized in abstract states for precision. We call such a memory node a "branching node" in the following. As an example, the memory node in purple in Figure 9.1(a) is a branching node. If it were summarized, one of the cursors `x`, `y` would have to be dropped. A precise abstraction would abstract the branch node exactly with points-to edges, as shown at the top of Figure 9.1(b). As silhouettes are used to guide the synthesizing of segment predicates in abstract state join operations, which may cause precision loss when branching nodes are summarized, we need to identify “branching nodes” in silhouettes and always keep precise information about them, such as the silhouette shown at the bottom of Figure 9.1(b).

Chapter 10

Silhouette-Guided Clumping and Widening

This chapter demonstrates that silhouettes can be used in order to precisely compute the clumping and widening of disjunctive abstract states. Specifically, Section 10.1 presents silhouette-guided clumping based on an equivalence relation of silhouettes and silhouette-guided joining. Furthermore, Section 10.2 defines silhouette-based widening of disjunctive abstract states, i.e., relying on silhouettes to select which pair of disjuncts to be widened together for more precise widening results.

10.1 Silhouette Guided Clumping of Abstract States

In this section, we first show that silhouettes need to be generalized in order to easily recognize silhouette similarities at the silhouette level in Section 10.1.1. Then, we define a silhouette clumping equivalence relation in Section 10.1.2. Intuitively, the relation is used to indicate whether abstract states can be joined precisely. In Section 10.1.3, we present a clumping algorithm.

10.1.1 Silhouette Generalization

While the silhouettes of abstract states defined in Sect. 9.2.3 are restrictions of silhouettes directed to live variables, they still adhere too closely to the structure of the abstract states to highlight all the possibilities for clumping. For instance, Fig. 10.1 shows two abstract states $\overline{\mathrm{m}}, \overline{\mathrm{m}}'$ taken from the abstract states of the analysis of the AVL tree insertion program shown in Figure 7.2. The silhouettes of $\overline{\mathrm{m}}$ and $\overline{\mathrm{m}}'$ are shown on the right side, and apparently are not equal. Yet, the two abstract states could be clumped as they both can be weakened into a **`tree`** segment predicate from `t` to `x` without harming the analysis as discussed in Section 7.2.

To recognize silhouette similarities in configurations such as that given in Figure 10.1, constraints should be generalized so as to make segment patterns easier to recognize at the silhouette level. Since segments correspond to regular expressions of the form $(\mathtt{f}_0+\ldots+\mathtt{f}_k)^\star$, the following generalization function makes such patterns appear more prominently:

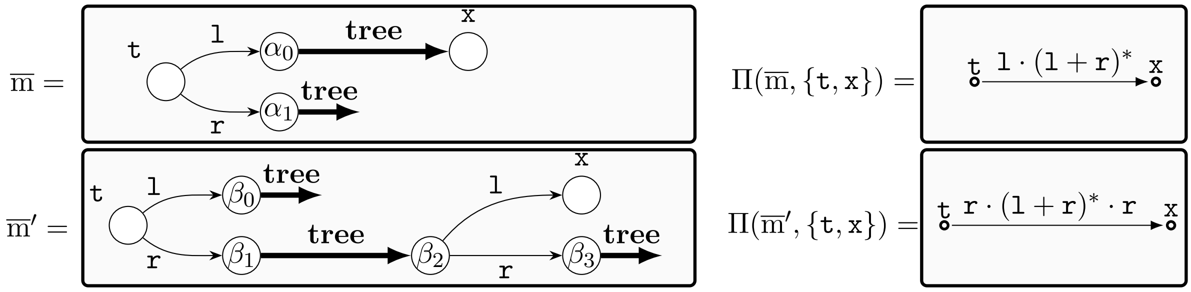

$\overline{\mathrm{m}} =$
t
l
r
α_0
α_1
tree
tree
x
$\overline{\mathrm{m}}' =$
t
l
r
β_0
β_1
β_2
β_3
tree
tree
tree
l
r
x
$\Pi(\overline{\mathrm{m}}, \{\mathtt{t}, \mathtt{x}\}) =$
t
$\mathtt{l} \cdot (\mathtt{l} + \mathtt{r})^*$
x
$\Pi(\overline{\mathrm{m}}', \{\mathtt{t}, \mathtt{x}\}) =$
t
$\mathtt{r} \cdot (\mathtt{l} + \mathtt{r})^* \cdot \mathtt{r}$
x

Figure 10.1: Abstract state candidates for clumping.

Definition 10.1 (Silhouette generalization). *The* generalization $\phi(\mathrm{e})$ *of a regular expression* e *of the form* $\mathtt{f}'_0 \cdot \ldots \cdot \mathtt{f}'_m \cdot (\mathtt{f}_0 + \ldots + \mathtt{f}_k)^\star \cdot \mathtt{f}''_0 \cdot \ldots \cdot \mathtt{f}''_n$*, where* $\{\mathtt{f}'_0, \ldots, \mathtt{f}'_m, \mathtt{f}''_0, \ldots, \mathtt{f}''_n\} \subseteq \{\mathtt{f}_0, \ldots, \mathtt{f}_k\}$*, is the regular expression* $\phi(\mathrm{e}) = (\mathtt{f}_0 + \ldots + \mathtt{f}_k)^\star$*. For all other regular expressions* e*, we let* $\phi(\mathrm{e}) = \mathrm{e}$.

The generalization *of a silhouette* $\overline{\mathfrak{s}}$ *is obtained by replacing any edge* $(\overline{v}, \mathrm{e}, \overline{v}')$ *of* $\overline{\mathfrak{s}}$ *by edge* $(\overline{v}, \phi(\mathrm{e}), \overline{v}')$*. It is noted* $\Phi(\overline{\mathfrak{s}})$.

This operation defines a weakening over silhouettes, since for all silhouettes $\overline{\mathfrak{s}}$, we can prove that $\gamma_{\overline{\mathfrak{S}}}(\overline{\mathfrak{s}}) \subseteq \gamma_{\overline{\mathfrak{S}}}(\Phi(\overline{\mathfrak{s}}))$.

Example 10.1. After generalization, we obtain:

$$\Phi(\Pi(\overline{\mathrm{m}}, \{\mathtt{t}, \mathtt{x}\})) = \{(\mathtt{t}, (\mathtt{l} + \mathtt{r})^\star, \mathtt{x})\} = \Phi(\Pi(\overline{\mathrm{m}}', \{\mathtt{t}, \mathtt{x}\}))$$

10.1.2 Clumping Relation

We now consider a criterion for the computation of candidate groups of abstract states for clumping based on silhouettes.

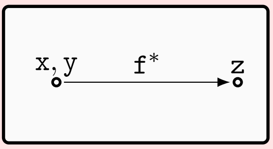
As introduced in Figure. 6.6, many the join rules implicitly rely on $\sqsubseteq_{\overline{\mathcal{M}_\omega}}$. Therefore, we can use silhouette entailment check $\sqsubseteq_{\overline{\mathbb{S}}}$ to prune out abstract state entailment checks that do not hold in the silhouettes based on Theorem. 8.4. However, silhouette-guided joining allows weakening rules to be applied on both sides, as presented in Definition 9.2, during the computation of a single join of two abstract states $\overline{\mathrm{m}}, \overline{\mathrm{m}}'$. Thus, we need a symmetric characterization of situations where a weakening may be performed either in $\overline{\mathrm{m}}$, in $\overline{\mathrm{m}}'$ or in both, which is the purpose of the following relation $\bowtie$.

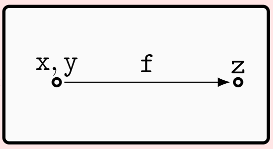
Definition 10.2 (Silhouette association relation). *Let* $\overline{\mathfrak{s}}'_{\lceil \mathrm{N}'}$ *be the restriction of a silhouette* $\overline{\mathfrak{s}}'$ *to a set of nodes* N'*. Let* $\overline{\mathfrak{s}}_0, \overline{\mathfrak{s}}_1$ *be two silhouettes with the same set of nodes* N*. We let* $\overline{\mathfrak{s}}'_0 = \Phi(\overline{\mathfrak{s}}_0)$ *and* $\overline{\mathfrak{s}}'_1 = \Phi(\overline{\mathfrak{s}}_1)$*. We write* $\overline{\mathfrak{s}}_0 \bowtie \overline{\mathfrak{s}}_1$ *if and only if there exist* $\mathrm{N}_0, \mathrm{N}_1$ *such that* $\mathrm{N} = \mathrm{N}_0 \cup \mathrm{N}_1$ *and:*

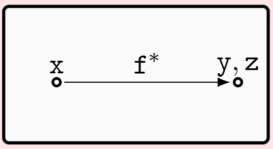
$$\begin{array}{rl} & \overline{\mathfrak{s}}'_0 = \overline{\mathfrak{s}}'_{0\lceil \mathrm{N}_0} \cup \overline{\mathfrak{s}}'_{0\lceil \mathrm{N}_1} \quad \wedge \quad \sqsubseteq_{\overline{\mathbb{S}}} (\overline{\mathfrak{s}}'_{0\lceil \mathrm{N}_0}, \overline{\mathfrak{s}}'_{1\lceil \mathrm{N}_0}) = \mathbf{true} \\ \wedge & \overline{\mathfrak{s}}'_1 = \overline{\mathfrak{s}}'_{1\lceil \mathrm{N}_0} \cup \overline{\mathfrak{s}}'_{1\lceil \mathrm{N}_1} \quad \wedge \quad \sqsubseteq_{\overline{\mathbb{S}}} (\overline{\mathfrak{s}}'_{1\lceil \mathrm{N}_1}, \overline{\mathfrak{s}}'_{0\lceil \mathrm{N}_1}) = \mathbf{true} \end{array}$$

This relation is symmetric and reflexive, but not transitive.

Example 10.2. Let $\bar{\mathfrak{s}}_0, \bar{\mathfrak{s}}_1, \bar{\mathfrak{s}}_2$ be defined by:

$\bar{\mathfrak{s}}_0 =$ 

$\bar{\mathfrak{s}}_1 =$ 

$\bar{\mathfrak{s}}_2 =$ 

Let $\mathrm{N}_0 = \{\mathtt{x}, \mathtt{y}\}$ and $\mathrm{N}_1 = \{\mathtt{y}, \mathtt{z}\}$, we get that $\bar{\mathfrak{s}}_{0\restriction \mathrm{N}_0} \sqsubseteq_{\mathbb{S}} \bar{\mathfrak{s}}_{1\restriction \mathrm{N}_0}$, $\bar{\mathfrak{s}}_{0\restriction \mathrm{N}_0} \sqsubseteq_{\mathbb{S}} \bar{\mathfrak{s}}_{2\restriction \mathrm{N}_0}$, and $\bar{\mathfrak{s}}_{1\restriction \mathrm{N}_1} \sqsubseteq_{\mathbb{S}} \bar{\mathfrak{s}}_{0\restriction \mathrm{N}_1}$, $\bar{\mathfrak{s}}_{2\restriction \mathrm{N}_1} \sqsubseteq_{\mathbb{S}} \bar{\mathfrak{s}}_{1\restriction \mathrm{N}_1}$. Thus, we have that $\bar{\mathfrak{s}}_0 \bowtie \bar{\mathfrak{s}}_1$ and $\bar{\mathfrak{s}}_0 \bowtie \bar{\mathfrak{s}}_2$ hold. However, $\bar{\mathfrak{s}}_1 \bowtie \bar{\mathfrak{s}}_2$ does not hold.

The silhouette association relation soundly characterizes the cases where a precise join can be computed using rules that perform weakening guided by existing predicates:

Theorem 10.1 (Silhouette association and weakening). *Let $\overline{\mathrm{m}}, \overline{\mathrm{m}}'$ be two abstract states. We assume that $\overline{\mathrm{m}} = \overline{\mathrm{m}}_0 * \overline{\mathrm{m}}_1$ and $\overline{\mathrm{m}}' = \overline{\mathrm{m}}'_0 * \overline{\mathrm{m}}'_1$. Then:*

$$\overline{\mathrm{m}}_0 \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{\mathrm{m}}'_0 \wedge \overline{\mathrm{m}}'_1 \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{\mathrm{m}}_1 \Longrightarrow \Phi(\Pi(\overline{\mathrm{m}})) \bowtie \Phi(\Pi(\overline{\mathrm{m}}'))$$

Proof. According to Definition 8.3, $\Pi(\overline{\mathrm{m}}) = \Pi(\overline{\mathrm{m}}_0) \cup \Pi(\overline{\mathrm{m}}_1)$ and $\Pi(\overline{\mathrm{m}}') = \Pi(\overline{\mathrm{m}}'_0) \cup \Pi(\overline{\mathrm{m}}'_1)$. According to Theorem 8.4, $\overline{\mathrm{m}}_0 \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{\mathrm{m}}'_0 \Longrightarrow \Pi(\overline{\mathrm{m}}_0) \sqsubseteq_{\mathfrak{S}} \Pi(\overline{\mathrm{m}}'_0)$ and $\overline{\mathrm{m}}'_1 \sqsubseteq_{\overline{\mathcal{M}_\omega}} \overline{\mathrm{m}}_1 \Longrightarrow \Pi(\overline{\mathrm{m}}'_1) \sqsubseteq_{\mathfrak{S}} \Pi(\overline{\mathrm{m}}_1)$. Thus, we have $\Pi(\overline{\mathrm{m}}) \bowtie \Pi(\overline{\mathrm{m}}')$, which implies $\Phi(\Pi(\overline{\mathrm{m}})) \bowtie \Phi(\Pi(\overline{\mathrm{m}}'))$. ■

The theorem implies that:

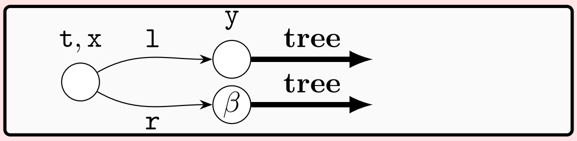
whenever a join can be computed by weakening part of $\overline{\mathrm{m}}$ into predicates present in $\overline{\mathrm{m}}'$ and part of $\overline{\mathrm{m}}'$ into predicates present in $\overline{\mathrm{m}}$, then relation $\bowtie$ holds.

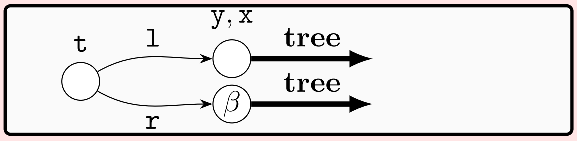
The contraposition entails that:

when $\bowtie$ does not hold, no precise join can be found using only rules that weaken abstract states based on existing predicates.

Therefore, Theorem 10.1 will prevent clumping from attempting to compute some but not all joins that will fail. However, the characterization that the theorem provides is actually very accurate, as shown by our experiments in Chapter 11.

Example 10.3 (Association relation). Let us recall the abstract states of Example. 9.2:

$\overline{\mathrm{m}}$: 

$\overline{\mathrm{m}}'$: 

If $\mathbb{X} = \{\mathtt{t}, \mathtt{y}\}$, then $\Pi(\overline{\mathrm{m}}, \widehat{\mathbb{X}}) = \{(\mathtt{t}, \mathtt{l}, \mathtt{y})\} = \Pi(\overline{\mathrm{m}}', \widehat{\mathbb{X}})$. Thus, $\Phi(\Pi(\overline{\mathrm{m}}, \mathtt{s})) \bowtie \Phi(\Pi(\overline{\mathrm{m}}', \mathtt{s}))$ holds. As observed in Example. 9.2, joining these two states is precise with respect to $\mathbb{X}$.

On the other hand, if $\mathbb{X} = \{\mathtt{t}, \mathtt{x}, \mathtt{y}\}$, we obtain $\Pi(\overline{\mathrm{m}}, \widehat{\mathbb{X}}) = \{(\mathtt{t}, \epsilon, \mathtt{x}), (\mathtt{x}, \mathtt{l}, \mathtt{y})\}$ and $\Pi(\overline{\mathrm{m}}', \widehat{\mathbb{X}}) = \{(\mathtt{t}, \mathtt{l}, \mathtt{x}), (\mathtt{x}, \epsilon, \mathtt{y})\}$, and thus $\Phi(\Pi(\overline{\mathrm{m}}, \widehat{\mathbb{X}})) \bowtie \Phi(\Pi(\overline{\mathrm{m}}', \widehat{\mathbb{X}}))$ does not hold. Indeed, we observed that joining them would incur a precision loss.

With the silhouette association relation, we now can define the equivalence silhouette clumping relation $\sim$.

Definition 10.3 (Clumping relation $\sim$). *Given a set of silhouettes $\overline{\mathbb{S}} = \{\bar{\mathfrak{s}}_0, \ldots, \bar{\mathfrak{s}}_n\}$, the clumping relation $\sim$ is defined as the transitive closure of $\bowtie$:*

$$\forall \bar{\mathfrak{s}}_i, \bar{\mathfrak{s}}_j \in \overline{\mathbb{S}},\ \bar{\mathfrak{s}}_i \sim \bar{\mathfrak{s}}_j \iff \bar{\mathfrak{s}}_i \bowtie \bar{\mathfrak{s}}_j \vee \exists \bar{\mathfrak{s}}_k \in \overline{\mathbb{S}},\ \bar{\mathfrak{s}}_i \sim \bar{\mathfrak{s}}_k \wedge \bar{\mathfrak{s}}_k \sim \bar{\mathfrak{s}}_j$$

The clumping relation $\sim$ is symmetric, reflexive, and transitive.

Example 10.4 (Clumping relation). Let us consider the silhouette in Example 10.2, we can get that: $\bar{\mathfrak{s}}_0 \sim \bar{\mathfrak{s}}_1 \sim \bar{\mathfrak{s}}_2$.

10.1.3 Clumping Algorithm

Fig. 10.2 summarizes the **clump** algorithm for clumping abstract states, which takes as input a disjunctive abstract state and the set of live variables at the current location in the program being analyzed and outputs a clumped disjunctive abstract state with fewer disjuncts:

1. It first computes the silhouette of all the abstract states and the generalized form of these silhouettes, where the generalized forms are used in order to compute $\bowtie$.

2. It then groups silhouettes that are connected components of the silhouette association relation $\bowtie$, such that each group is an equivalence class of silhouette clumping relation $\sim$. Moreover, each group is sorted as a sequence of silhouettes such that any silhouette (except the first one) in the sequence is associated with a previous silhouette.

3. For each group, the sequence of silhouettes suggests a sequence of abstract state joins, that are expected to preserve silhouette joins by utilizing silhouette-guided joins to enable semantic-guided synthesis of summary predicates as shown in Chapter 9. We note that this join order has no relevance to the soundness of the algorithm.

This clumping algorithm is *sound*: it returns a disjunctive abstract state that over-approximates $\overline{\mathrm{m}}_0, \ldots, \overline{\mathrm{m}}_n$:

Theorem 10.2 (Clumping soundness). *For each disjunctive abstract state $\overline{\mathrm{m}}_0 \vee \ldots \vee \overline{\mathrm{m}}_n$ and for every set of variables $\mathbb{X}$, we have:*

$$\gamma_{\overline{\mathcal{D}}}(\overline{\mathrm{m}}_0 \vee \ldots \vee \overline{\mathrm{m}}_n) \subseteq \gamma_{\overline{\mathcal{D}}}(\mathbf{clump}(\overline{\mathrm{m}}_0 \vee \ldots \vee \overline{\mathrm{m}}_n, \mathbb{X}))$$

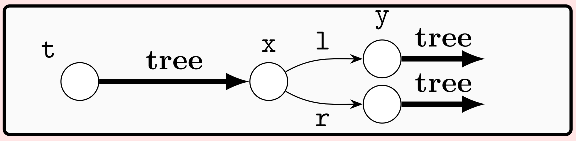
Proof. The soundness is a direct consequence of the soundness of silhouette-guided joining. ■

```
input:   disjunctive abstract state $\overline{\mathrm{m}}_0 \vee \ldots \vee \overline{\mathrm{m}}_n$
         set of live variables $\mathbb{X}$
output:  clumped disjunctive abstract state $\overline{\mathrm{m}}'_0 \vee \ldots \vee \overline{\mathrm{m}}'_k$
$\overline{\mathfrak{s}}_0 \leftarrow \Pi(\overline{\mathrm{m}}_0, \widehat{\mathbb{X}}); \ldots ; \overline{\mathfrak{s}}_n \leftarrow \Pi(\overline{\mathrm{m}}_n, \widehat{\mathbb{X}});$
$\overline{\mathfrak{s}}'_0 \leftarrow \Phi(\overline{\mathfrak{s}}_0); \ldots ; \overline{\mathfrak{s}}'_n \leftarrow \Phi(\overline{\mathfrak{s}}_n);$
computation of relation $\bowtie$ over $\overline{\mathfrak{s}}'_0, \ldots, \overline{\mathfrak{s}}'_n$
   and of the connected components $\overline{\mathbb{S}}_0, \ldots, \overline{\mathbb{S}}_k$ of $\bowtie$
for each connected component $\overline{\mathbb{S}}_j$ of $\bowtie$
      sort the elements of $\overline{\mathbb{S}}_j$ into $\overline{\mathfrak{s}}'_{i_0} \prec \ldots \prec \overline{\mathfrak{s}}'_{i_l}$
         such that $\forall p, \exists q \leq p,\ \overline{\mathfrak{s}}'_{i_q} \bowtie \overline{\mathfrak{s}}'_{i_{p+1}}$
      $\overline{\mathrm{m}}'_j \leftarrow \overline{\mathrm{m}}_{i_0}; \overline{\mathfrak{s}}^{\sqcup} \leftarrow \overline{\mathfrak{s}}_{i_0};$
      for $p = 1$ to $l$
           $\overline{\mathfrak{s}}^{\sqcup} \leftarrow \mathbf{join}_{\overline{\mathfrak{S}}}(\overline{\mathfrak{s}}^{\sqcup}, \overline{\mathfrak{s}}_{i_p});$
           $\overline{\mathrm{m}}'_j \leftarrow \mathbf{join}_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}'_j, \overline{\mathrm{m}}_{i_p}, \overline{\mathfrak{s}}^{\sqcup});$
```

Figure 10.2: Clumping algorithm **clump**.

Following Theorem 10.1 and Theorem 9.2, this algorithm will not attempt to compute abstract state joins that will not succeed in producing a precise result in the silhouette level. However, clumping may propose to clump states that do not join well, for example, when the join of silhouettes indicates synthesizing a segment in abstract states join, yet the synthesizing fails. However, the experimental results of Sect. 11 do not show any occurrence of such a precision loss.

Example 10.5 (The algorithm clump). Let us consider the abstract states in Figure 10.3. The **clump** algorithm computes that $\overline{\mathfrak{s}}_1 \sim \overline{\mathfrak{s}}_2 \sim \overline{\mathfrak{s}}_3$, and therefore clumps $\overline{\mathrm{m}}_1, \overline{\mathrm{m}}_2, \overline{\mathrm{m}}_3$ into the abstract state below:



We note that the segment predicate in the abstract state is synthesized with the silhouette-guided segment introduction rule (Definition 9.2). At the same time, it keeps $\overline{\mathrm{m}}_0$ separate. Here, the clumping result is precise for the analysis to infer interesting properties, as discussed in Chapter 7, and in addition, it is also optimal regarding the number of disjuncts.

10.2 Silhouette-guided Widening of Disjunctive Abstract states

To infer loop invariants from an abstract pre-condition, the analysis needs to use a *widening* operator $\nabla_{\overline{\mathcal{D}}}$ over disjunctive abstract states. This operator should over-approximate concrete union, and ensure that any sequence $(\overline{\mathrm{d}}_n)_n$ of the form $\overline{\mathrm{d}}_{n+1} = \nabla_{\overline{\mathcal{D}}}(\overline{\mathrm{d}}_n, \overline{\mathrm{d}}'_n)$ terminates. In this

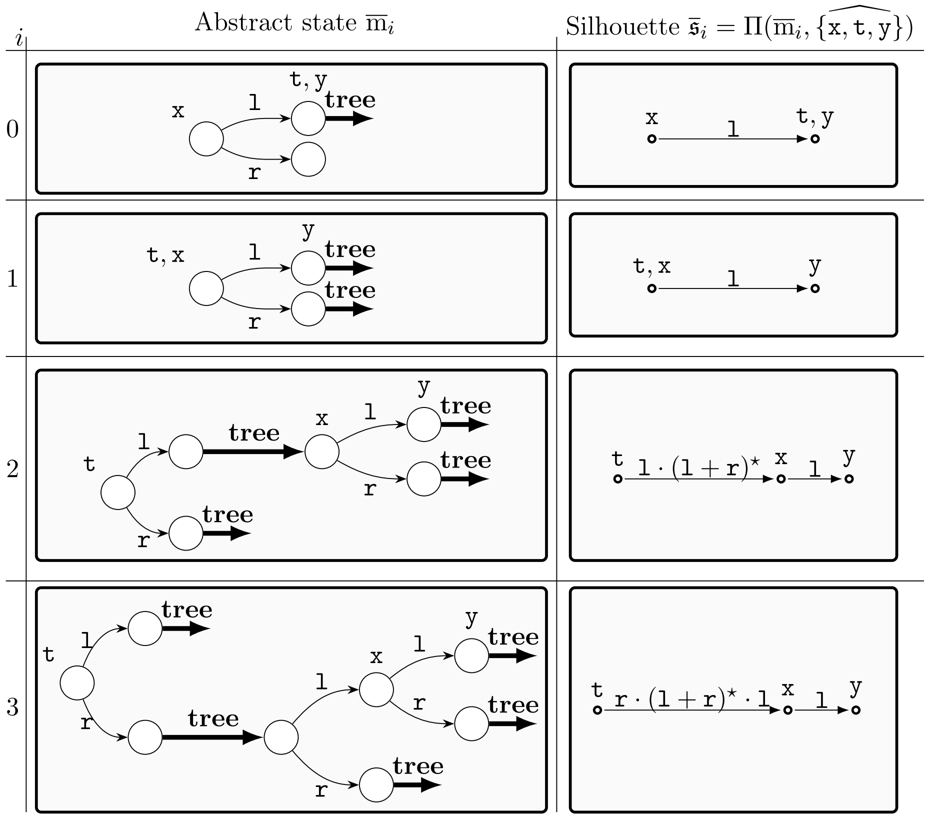

i
Abstract state $\overline{m}_i$
Silhouette $\overline{\mathfrak{s}}_i = \Pi(\overline{m}_i, \widehat{\{x, t, y\}})$
0
x
l
t, y
tree
r
x
l
t, y
1
t, x
l
y
tree
tree
r
t, x
l
y
2
t
l
tree
x
l
y
tree
tree
r
r
tree
t
$l \cdot (l + r)^\star$
x
l
y
3
t
l
tree
r
tree
l
x
l
y
tree
r
tree
r
tree
t
$r \cdot (l + r)^\star \cdot l$
x
l
y

Figure 10.3: Clumping abstract states from the analysis of the AVL tree insertion program (given in Figure 7.2) with their respective silhouettes

section, we assume that $\triangledown_{\overline{\mathcal{M}_\omega}}$ is a widening over abstract states, using similar algorithms to $\sqcup_{\overline{\mathcal{M}_\omega}}$. Such an operator can often be based on $\sqcup_{\overline{\mathcal{M}_\omega}}$ [CRN07].

Widening based on silhouette-guided pairing. Intuitively, a widening of $\overline{d}$ with $\overline{d}'$ should widen disjuncts of $\overline{d}$ with disjuncts of $\overline{d}'$, using some sort of pairing to select which pairs of disjuncts are to be widened together. Since silhouettes aim at capturing abstract states that can be joined precisely, we build our widening around a pairing function that we build based on the silhouette.

Definition 10.4 (Pairing function). *Given $\overline{d} = \overline{m}_0 \vee \ldots \vee \overline{m}_n$ and $\overline{d}' = \overline{m}'_0 \vee \ldots \vee \overline{m}'_{n'}$, a* pairing *of $\overline{d}, \overline{d}'$ is a function $\pi_{\overline{d},\overline{d}'}$ from $\{0, \ldots, n\}$ into the powerset of $\{0, \ldots, n'\}$ such that $i \neq j$ implies $\pi(i) \cap \pi(j) = \emptyset$.*

If a pairing family $\pi_{\overline{d},\overline{d}'}$ is defined for all $\overline{d}, \overline{d}' \in \overline{\mathcal{D}}$, we can define an operator over disjunctive abstract states that lets the pairing function define which disjuncts of the right-hand argument are widened with each disjunct of the left-hand argument, and preserves the disjuncts of the right-hand argument that do not appear in the pairing.

Definition 10.5 (Disjunctive widening operator). *Let* $\overline{\mathrm{d}} = \overline{\mathrm{m}}_0 \vee \ldots \vee \overline{\mathrm{m}}_n$ *and* $\overline{\mathrm{d}}' = \overline{\mathrm{m}}'_0 \vee \ldots \vee \overline{\mathrm{m}}'_{n'}$. *We let* $\nabla_{\overline{\mathcal{D}}}(\overline{\mathrm{d}}, \overline{\mathrm{d}}')$ *be defined as the abstract state* $\overline{\mathrm{m}}''_0, \ldots, \overline{\mathrm{m}}''_{n+k}$, *where:*

- $\overline{\mathrm{m}}''_i = \nabla_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}_i, \sqcup_{\overline{\mathcal{M}_\omega}}(\{\overline{\mathrm{m}}'_l \mid l \in \pi_{\overline{\mathrm{d}},\overline{\mathrm{d}}'}(i)\}))$ *if* $i \leq n$;
- $\{\overline{\mathrm{m}}''_{n+i} \mid 1 \leq i \leq k\} = \{\overline{\mathrm{m}}'_j \mid \forall i,\ j \notin \pi_{\overline{\mathrm{d}},\overline{\mathrm{d}}'}(i)\}$.

This operator always returns a sound over-approximation of its arguments. Yet, depending on the pairing family, it may fail to guarantee termination. In the following, we define a pairing family that ensures termination so that $\mathbf{widen}_{\overline{\mathcal{D}}}$ does indeed define a widening.

A good pairing should map abstract states that produce a precise widening, in the same way as clumping for join. It should also drive the introduction of summary predicates. As observed in Chapter 9, silhouettes hold information capable of guiding this process. However, for termination, we need to bound silhouettes. Thus, we define a pairing that is parameterized by an integer bound b, and that associates abstract states with silhouettes that are equivalent when regular expressions are smashed upon exceeding length b (in the following, let $b = 1$):

Definition 10.6 (Silhouette-based pairing). *Let* $\mathbb{F}$ *denote the set of all field names and let* b *be an integer bound.*

We define the regular expression bounded abstraction by:

$$\omega_b(\mathrm{e}_1 \cdot \ldots \cdot \mathrm{e}_k) = \begin{cases} \mathrm{e}_1 \cdot \ldots \cdot \mathrm{e}_b \cdot \mathbb{F}^\star & \textit{if } k > b \\ \mathrm{e}_1 \cdot \ldots \cdot \mathrm{e}_k \cdot \mathbb{F}^\star & \textit{otherwise} \end{cases}$$

Let function Ω_b *abstract a silhouette by replacing each edge* $(\overline{v}, \mathrm{e}, \overline{v}')$ *of the silhouette by* $(\overline{v}, \omega_b(\mathrm{e}), \overline{v}')$.

Then, given silhouettes $\overline{\mathfrak{s}}, \overline{\mathfrak{s}}'$, *we write* $\overline{\mathfrak{s}} \rtimes_b \overline{\mathfrak{s}}'$ *if and only if* $\sqsubseteq_{\overline{\mathbb{S}}} (\Omega_b(\overline{\mathfrak{s}}'), \Omega_b(\overline{\mathfrak{s}})) = \mathbf{true}$.

Last, given disjunctive abstract states $\overline{\mathrm{d}} = \overline{\mathrm{m}}_0 \vee \ldots \vee \overline{\mathrm{m}}_n$ *and* $\overline{\mathrm{d}}' = \overline{\mathrm{m}}'_0 \vee \ldots \vee \overline{\mathrm{m}}'_{n'}$, *the pairing function* $\pi_{\overline{\mathrm{d}},\overline{\mathrm{d}}'}$ *of* $\overline{\mathrm{d}}, \overline{\mathrm{d}}'$ *satisfies:*

$$j \in \pi_{\overline{\mathrm{d}},\overline{\mathrm{d}}'}(i) \implies \overline{\mathfrak{s}}_i \rtimes_b \overline{\mathfrak{s}}'_j$$

Theorem 10.3 (Disjunctive widening). *Using the pairing of Definition 10.6, the widening operator* $\nabla_{\overline{\mathcal{D}}}$ *of Definition 10.5 enforces termination of abstract iterates.*

Proof. The proof relies on the finiteness of the image of Ω_b, which entails that, for any sequence $(\overline{\mathrm{d}}_n)_n$ of widened iterates (such that $\overline{\mathrm{d}}_{n+1} = \nabla_{\overline{\mathcal{D}}}(\overline{\mathrm{d}}_n, \overline{\mathrm{d}}'_n)$), the disjuncts in $\overline{\mathrm{d}}_n$ eventually stabilize to a set corresponding to a fixed, finite set of silhouettes. ■

‘

Example 10.6 (Widening). Fig. 10.4 displays a few of the disjuncts that arise in the second and third iteration over the first loop in the analysis of the AVL tree insertion program presented in Fig. 7.2.

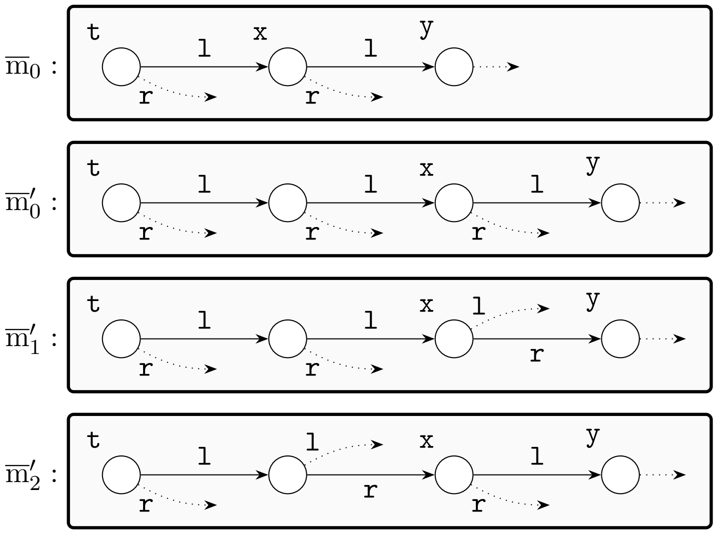

$\overline{\mathrm{m}}_0$:
t
x
y
l
l
r
r
$\overline{\mathrm{m}}'_0$:
t
x
y
l
l
l
r
r
r
$\overline{\mathrm{m}}'_1$:
t
x
l
y
l
l
r
r
r
$\overline{\mathrm{m}}'_2$:
t
l
x
y
l
l
r
r
r

Figure 10.4: Widening disjunctive abstract states.

Disjunct $\overline{\mathrm{m}}_0$ occurs in the second iteration whereas disjuncts $\overline{\mathrm{m}}'_0, \overline{\mathrm{m}}'_1, \overline{\mathrm{m}}'_2$ arise in the third iteration. For clarity, we only show the nodes on the path between `t`, `x` and `y`. In the table below, we show their silhouettes before and after applying the bounded abstraction:

	silhouette	bounded abstraction
$\overline{\mathrm{m}}_0$	$\{(\mathtt{t}, \mathtt{l}, \mathtt{x}), (\mathtt{x}, \mathtt{l}, \mathtt{y})\}$	$\{(\mathtt{t}, \mathtt{l} \cdot \mathbb{F}^\star, \mathtt{x}), (\mathtt{x}, \mathtt{l} \cdot \mathbb{F}^\star, \mathtt{y})\}$
$\overline{\mathrm{m}}'_0$	$\{(\mathtt{t}, \mathtt{l} \cdot \mathtt{l}, \mathtt{x}), (\mathtt{x}, \mathtt{l}, \mathtt{y})\}$	$\{(\mathtt{t}, \mathtt{l} \cdot \mathbb{F}^\star, \mathtt{x}), (\mathtt{x}, \mathtt{l} \cdot \mathbb{F}^\star, \mathtt{y})\}$
$\overline{\mathrm{m}}'_1$	$\{(\mathtt{t}, \mathtt{l} \cdot \mathtt{l}, \mathtt{x}), (\mathtt{x}, \mathtt{r}, \mathtt{y})\}$	$\{(\mathtt{t}, \mathtt{l} \cdot \mathbb{F}^\star, \mathtt{x}), (\mathtt{x}, \mathtt{r} \cdot \mathbb{F}^\star, \mathtt{y})\}$
$\overline{\mathrm{m}}'_2$	$\{(\mathtt{t}, \mathtt{l} \cdot \mathtt{r}, \mathtt{x}), (\mathtt{x}, \mathtt{l}, \mathtt{y})\}$	$\{(\mathtt{t}, \mathtt{l} \cdot \mathbb{F}^\star, \mathtt{x}), (\mathtt{x}, \mathtt{l} \cdot \mathbb{F}^\star, \mathtt{y})\}$

Then, $\Pi(\overline{\mathrm{m}}_0) \rtimes_\flat \Pi(\overline{\mathrm{m}}'_0)$ and $\Pi(\overline{\mathrm{m}}_0) \rtimes_\flat \Pi(\overline{\mathrm{m}}'_2)$ hold whereas $\Pi(\overline{\mathrm{m}}_0) \rtimes_\flat \Pi(\overline{\mathrm{m}}'_1)$ does not. Thus, the pairing is defined by $\pi(0) = \{0, 2\}$. The widening will thus preserve the information which establishes whether `y` is the left or right child of `x`.

10.3 Static Analysis

We now summarize the whole analysis.

As a forward abstract interpretation [CC77], it starts with an abstract pre-condition that over-approximates initial memory states with a single disjunct which abstracts either the set of all memory states, with no structure allocated for whole program analysis, or a pre-condition describing the valid call states for the analysis of a library function.

It computes disjunctive abstract post-conditions for each statement. Post-conditions of assignment, test, allocation and deallocation statements are computed locally on each disjunct in parallel as shown in Section 2.4.5. Before a control flow join that joins two disjunctive abstract states $\overline{\mathrm{d}}$ and $\overline{\mathrm{d}}'$, the analysis either simply returns $\overline{\mathrm{d}} \vee \overline{\mathrm{d}}'$ or applies clumping (Section 10.1) to $\overline{\mathrm{d}} \vee \overline{\mathrm{d}}'$ in order to reduce the disjunction size. For loops, the silhouette-guided $\nabla_{\overline{\mathcal{D}}}$ operator (Section 10.2) enforces the convergence of sequences of disjunctive abstract states in order to compute an abstract fix-point.

The analysis is sound as formalized in Theorem 2.1, which states that the analysis returns abstract post-conditions that over-approximate final states.

Chapter 11

Experimental Evaluation of Silhouette-Directed Clumping

The silhouette-directed algorithms are implemented in the MemCAD *analyzer. This chapter assesses the efficiency of silhouette-directed methods in improving the static analysis of various data structures from real-world C libraries. Specifically, Section 11.1 presents some research hypotheses, Section 11.2 sets up the benchmarks and experimental methodology and Section 11.3 evaluates the hypotheses.*

11.1 Research Hypotheses

In this section, we empirically evaluate whether or not semantic-directed clumping is effective in improving the static analysis of data structures from real-world C libraries. We implemented clumping in the MemCAD analyzer [CR13]. We seek to provide evidence for or against the following hypotheses:

RH1 (Clumping is effective). *Semantic-directed clumping with guided join is necessary and effective for analyzing data structure operations from existing, real-world libraries.* The underlying premise of this work is that inferring disjunctive loop invariants is necessary to effectively analyze real-world data structures. While certain code may be reasonably adapted to avoid the need for disjunctive invariants, analyzing existing, real-world code typically involves handling corner cases in separate disjuncts. We assess the impact of silhouettes on the analysis and make a comparision with other techniques.

RH2 (Guided join is necessary). *Guided join is necessary to avoid unacceptable precision loss.* Semantic-directed clumping uses silhouette abstraction to select the disjuncts to join. Guided join then subsequently uses these same silhouettes to perform the shape join. We seek to evaluate the need for this latter step.

RH3 (Clumping has a low overhead). *The overhead of semantic-directed clumping is reasonable.* We hope and expect that the additional overhead in computing and comparing silhouettes is outweighed by the benefits of increased precision and improved scalability. This aspect must be tested empirically as the number of silhouette comparisons is quadratic in the number of disjuncts.

RH4 (Clumping limits disjunctive explosion). *Semantic-directed clumping improves the scalability of disjunctive analysis by limiting the number of disjuncts in the abstract state.* Without clumping, the number of disjuncts grows exponentially from control-flow paths and the unfolding of inductive predicates. The scalability of disjunctive analysis (and thus most shape analyses) is limited by the exponential growth in the number of disjuncts in the abstract state. Thus the technical challenge is to avoid piling up more and more unnecessary disjuncts while analyzing sequences of operations.

11.2 Experimental Methodology

To evaluate the effectiveness of semantic-based clumping, we consider 26 benchmarks of varying implementation styles and degrees of difficulty to analyze. These include operations over singly-linked and doubly-linked lists, as well as various binary trees with different kinds of invariants and pointer patterns (search, splay, red black, or AVL and with various sharing patterns). The operations consist of variants of finding an element, inserting, deleting, reversing, and sorting.

Notably, 21 benchmarks are from external sources. Some of these come with typically simple, user-specified assertions (e.g., x != NULL). Each benchmark consists of a top-level function implementing a data structure operation with a pre-condition and a post-condition that specify the preservation of precise shape invariants. Some routines are recursive whereas the others contain nested loops.

Table 11.1 lists these benchmarks with metrics to get a sense of the difficulty to analyze, including the presence of recursion, lines of code, numbers of sub-routines, numbers of loops and acyclic paths. The number of loops and acyclic paths gives the number of disjuncts that a naïve path and context-sensitive analysis would have at the exit point. The final metric is the maximum number of simultaneous pointers into a data structure instance (column “Simult. Pointer”). The summary row gives total counts of LOCs, assertions, functions, loops and paths, and the average number of simultaneous pointers (As discussed in Chapter 7, the number of simultaneous pointers into an instance is related to the number of disjuncts needed to represent the program invariant).

The analysis is parameterized by the definition of **ind** (or infers it for basic list and tree cases). It attempts to prove memory safety, any user-specified assertions, and the (more complex) post-condition given a C program and optionally inductive predicates summarizing memory regions. Clumping is applied at the beginning and the end of functions, at loop heads, at loop exits, and at the end of branches.

We evaluate and compare the following clumping strategies:

- `ClumpG` and `Clump` do silhouette-guided clumping and widening (Chapter 10); `ClumpG` uses guided join (Chapter 9), whereas `Clump` does not;
- `None` is the baseline technique, and does not compute silhouettes to perform clumping or guided joining. Instead, the joining keeps all disjuncts of both inputs and the widening merges all disjuncts of each input into one;

		User				Simult.
Benchmark	LOC	Assert	Fun	Loop	Path	Pointer
singly-linked list						
sll-delmin	25	0	1	1	12	5
sll-delmin†	26	0	2	1†	6	5
sll-delminmax	49	0	1	1	248	7
sll-delminmax†	52	0	2	1†	124	7
binary search tree						
bstree-find	26	0	1	1	4	3
bstree-find†	26	0	2	1†	4	3
Predator singly-linked list						
psll-reverse	11	0	1	1	2	3
psll-isort	20	0	1	2*	5	5
psll-bsort	25	0	1	2*	10	4
GDSL doubly-linked list (back pointers) with sentinel head and tail						
gdll-findmin	49	14	8	1	3	5
gdll-index	55	14	9	2	24	2
gdll-findmax	58	14	8	1	3	5
gdll-find	78	26	10	1	18	5
gdll-delete	107	26	12	1	72	5
GDSL binary search tree with leaf-to-root and back pointers						
gbstree-find	53	8	7	1	20	3
gbstree-insert	133	15	12	1	7680	5
gbstree-delete	165	9	15	1	23040	10
BSD splay tree						
bsplay-find	81	0	4	1	56	5
bsplay-delete	95	0	4	2	448	5
bsplay-insert	101	0	4	1	43	5
BSD red black tree with back pointers						
brbtree-find	29	0	3	1	4	2
brbtree-insert	177	0	4	2	3036	7
brbtree-delete	329	0	5	3	$1.e+8$	12
JSW AVL tree						
javl-find	25	0	3	1	26	2
javl-free	27	0	3	1	3	3
javl-insert	95	0	6	2	$1.e+8$	6
summary	1917	126	123	34	$2.e+8$	5.0

Table 11.1: List of benchmarks, divided into internal, micro-benchmarks (top) and benchmarks from external sources (bottom) including the Predator test suite [DPV11], the GNU Data Structure Library (GDSL), the BSD library, and a tutorial implementation of AVL trees (JSW) [Wal03]. Some external libraries include typically simple, user-specified assertions (User Assert). A † indicates that the routine is recursive, while a $*$ indicates that the loops are nested. The subsequent columns provide metrics for the complexity of the code, including the total number of functions (Fun), the number of loops (Loop), the number of acyclic paths (Path), and the maximum number of simultaneous pointers into a data structure instance (Simult. Pointer).

- `Canon` and `CanonG` conservatively model canonicalization operators [DOY06, SRW02]: they compute silhouettes but only join abstractions when their silhouettes are exactly the same (after folding nodes which are not pointed to by live variables); `CanonG` uses guided joining whereas `Canon` does not (we expect these strategies will still compute fewer disjunctions than a purely syntactic canonicalization would).

11.3 Experimental Evaluation

RH1: Clumping is effective. Our analyzer attempts to infer complex disjunctive shape invariants in loops, which is particularly challenging considering the benchmarks shown in Table 11.1 with not only forward and back pointers but also possibly unbounded sharing patterns. Back pointers (as in doubly-linked lists or trees with parent pointers) require not only forward unfolding of inductive summaries but also backward unfolding [CR08]. The BSD red-black tree has parent pointers, which is heavily used in rebalancing. The leaves of the GDSL binary search tree all point back to the root node.

We observe a number of implementation idioms that lead to a need for disjunctions:

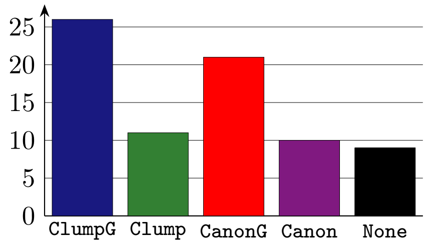
- First, the maximum number of simultaneous pointers into a data structure instance ("Simult. Pointer") is one driving factor as mentioned above, which typically increases as the operations get more complex (e.g., 12 for brbtree-delete).

- Second, the GDSL doubly-linked list uses sentinel nodes for the list head and tail. Because they are different to normal nodes, they cannot be summarized into the normal list segment, which yields more disjuncts to represent the possible points-to relationships between those nodes and the internal list segment.

- Third, nodes sometimes need to remain materialized between loops. Disjunctions are needed to represent the cases where the materialized node can occur but the number of disjuncts explodes exponentially in the number of materialized nodes in a tree. For example, delete in a red-black tree (brbtree-delete) requires three loops in sequence: find the node n to delete, find the minimum node m in the right subtree of n (i.e., the next in-order node to preserve the binary search invariant), rebalance the tree from the right subtree of that minimum node m. The node to delete n must be kept materialized between the first and second loop (to be able to track the swap of m and n) but becomes irrelevant between the second and the third loop (after the swap).

While maintaining a disjunct, or trace partition, for every acyclic path may be sufficiently precise in many or even most cases, it is clear from the path counts (column Path) that this choice is utterly infeasible in practice. The essence of semantic-directed clumping is to identify simultaneous pointers, sentinel nodes and the sort of relevance or irrelevance of a materialized node by computing silhouettes (i.e., "abstracting the abstraction").

In Figure 11.1(a), we compare clumping with guided join with the other strategies, showing the number of benchmarks that can be successfully verified memory safe (i.e., free of null or dangling pointer dereferences), the user-specified asserts, and the shape preservation post-condition using the different strategies (see also the bottom line of Figure 11.1(b)). We find that clumping



(a) Number of benchmarks verified using each strategy.

Benchmark	ClumpG	Clump	CanonG	Canon	None
sll-delmin	0.04	$\top$	0.05	$\top$	$\top$
sll-delmin†	0.04	0.05	$\top$	$\top$	$\top$
sll-delminmax	0.12	$\top$	0.42	$\top$	$\top$
sll-delminmax†	0.20	0.20	$\top$	$\top$	$\top$
bstree-find	0.03	$\top$	0.05	$\top$	$\top$
bstree-find†	0.04	0.04	0.11	0.11	$\top$
psll-reverse	0.02	0.02	0.02	0.02	0.02
psll-isort	0.03	0.03	0.04	0.04	$\top$
psll-bsort	0.04	0.04	0.04	0.04	0.06
gdll-findmin	0.61	0.61	0.62	$\top$	0.61
gdll-index	0.61	$\top$	0.62	$\top$	0.61
gdll-findmax	0.61	0.61	0.62	$\top$	0.60
gdll-find	0.62	0.62	0.63	0.65	$\top$
gdll-delete	0.62	0.63	0.63	0.64	$\top$
gbstree-find	0.59	$\top$	0.59	$\top$	0.58
gbstree-insert	0.65	$\top$	$\top$	$\top$	$\top$
gbstree-delete	1.64	$\top$	1.71	$\top$	1.39
bsplay-find	0.28	$\top$	0.56	0.56	$\top$
bsplay-delete	0.48	$\top$	1.08	1.07	$\top$
bsplay-insert	0.30	$\top$	0.62	0.62	$\top$
brbtree-find	0.36	$\top$	0.36	$\top$	$\top$
brbtree-insert	1.07	$\top$	$\top$	$\top$	$\top$
brbtree-delete	6.06	$\top$	$\top$	$\top$	$\top$
javl-find	0.19	$\top$	0.19	$\top$	0.19
javl-free	0.18	0.19	0.19	0.18	0.19
javl-insert	1.84	$\top$	5.22	$\top$	$\top$
average (all)	0.66	0.28	0.68	0.39	0.47
verified	26	11	21	10	9

(b) Analysis times are only reported if all assertions are successfully verified, including the preservation invariant in post-conditions. $\top$ indicates a failure to verify (due to precision loss). Run times (in seconds) measured on one core of a 3.20GHz Intel Xeon with 16GB of RAM. Times are reported as an average of three runs.

Figure 11.1: Successful verification times with different analysis strategies. The clumping with guided join strategy can verify significantly more benchmarks than any other strategy and over 3x more than the `None` baseline—and all at similar costs in analysis time.

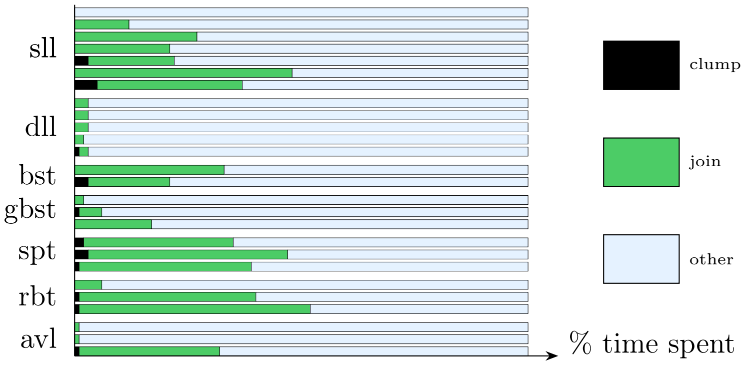
with guided join (ClumpG) is significantly more effective (able to verify benchmarks) than the baseline and also more capable than any other strategy. CanonG fails to verify, for example, the particularly complex red-black tree insertion and deletion operations.

Clumping with guided join is able to verify these additional benchmarks with an analysis time that is comparable with any other configuration, as shown in Figure 11.1(b). The analysis time is larger for more complex benchmarks (e.g., for red-black tree delete brbtree-delete), which, as expected, raises its average analysis time, but ClumpG is also the *only* strategy that succeeds on this code. ClumpG is also notably faster than CanonG on javl-insert. Therefore, we conclude that ClumpG takes comparable or even less time than other strategies on benchmarks where other strategies succeed (e.g., in the bsplay benchmarks).

Another measure of effectiveness of ClumpG is that analysis logs show this strategy led to no precision loss in joins.

RH2: Guided join is necessary. Figure 11.1(a) and 11.1(b) also show not only that clumping is effective but that guided join is also necessary. While Clump does slightly improve on the baseline (11 versus 9), where two more benchmarks are verified, ClumpG (clumping with guided joining) is able to verify all the 26 benchmarks. Moreover, CanonG (canonicalization with guided joining) successfully verifies 21 benchmarks, while Canon alone only verifies 10 benchmarks. Thus, the guided join strategies (ClumpG and CanonG) are significantly better (26 and 21, respectively). Moreover, we find that Clump and Canon tend to fail more in the tree benchmarks, as there is a larger search space for the join; guiding appears more critical when the search space for join is larger.

RH3: Clumping has a low overhead. From Figure 11.1(b), we see that ClumpG is comparable in the analysis time with any other configuration, which provides some evidence that clumping has a reasonable overhead. To see this more directly, in the graph below, we show the percentage of the analysis time spent on clumping, join, and other operations for each benchmark. We find that join is a very expensive operation and still takes about half of the analysis time in many cases despite effective clumping. Moreover, we find that in all cases the time spent on clumping (which includes the time to compute silhouettes) is a very small percentage of the total analysis time (no more than a few percent), and much smaller than the time spent on join. Therefore, we can conclude that clumping has a reasonably low overhead.



	`ClumpG`			`CanonG`			`None`		
Benchmark	Fix	Max	Post	Fix	Max	Post	Fix	Max	Post
psll-reverse	1	1	1	1	1	1	1	1	1
psll-isort	1	2	1	1	3	2	1	$\top$	$\top$
psll-bsort	1	5	1	1	4	1	1	8	1
gdll-findmin	2	3	1	4	7	2	1	3	2
gdll-index	1	7	1	1	5	2	1	5	5
gdll-findmax	2	3	1	4	7	2	1	3	2
gdll-find	2	6	1	2	6	2	$\top$	$\top$	$\top$
gdll-delete	2	6	1	2	6	2	$\top$	$\top$	$\top$
gbstree-find	1	3	1	1	3	3	1	3	3
gbstree-insert	2	4	1	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
gbstree-delete	1	69	1	2	68	1	1	54	54
bsplay-find	3	42	1	5	89	1	$\top$	$\top$	$\top$
bsplay-delete	3	42	1	5	89	1	$\top$	$\top$	$\top$
bsplay-insert	3	42	1	5	89	1	$\top$	$\top$	$\top$
brbtree-find	1	13	1	2	8	2	$\top$	$\top$	$\top$
brbtree-insert	3	51	1	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
brbtree-delete	3	108	1	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
javl-find	1	3	1	1	3	1	1	3	3
javl-free	1	2	1	1	2	1	1	$\top$	$\top$
javl-insert	3	120	1	18	240	1	$\top$	$\top$	$\top$
max	3	120	1	18	240	3	1	54	54

(a) The maximum number of disjuncts at a loop head (Fix), at any program point (Max), and at the exit point (Post) produced by the `ClumpG`, `CanonG`, and `None` analysis configurations.

	`ClumpG`		`CanonG`		`None`	
Benchmark	Time	Post	Time	Post	Time	Post
gbstree-find	3.3	1	2.18	1	40.79	158
brbtree-insert	60.2	1	$\top$	$\top$	$\top$	$\top$
javl-find	0.63	1	$\top$	$\top$	3.77	158
javl-insert	129.9	1	526.17	1	$\top$	$\top$
average (all)	48.5075	1	264.175	1	22.28	158

(b) Synthesize a new benchmark by sequentially composing a data structure operation 32 times to simulate multiple operations in sequence (times in seconds).

Figure 11.2: Clumping limits disjunctive explosion.

RH4: Clumping limits disjunctive explosion. Figure 11.2(a) digs more deeply into the verification times from Figure 11.1(b) by considering the maximum number of disjuncts produced at a loop head, any program point, and at the exit point of the operation. First, we observe that interestingly, the `None` baseline configuration only succeeds when the number of disjuncts at a loop head is 1. When more than one disjunct is needed for the loop invariant, the silhouette abstraction seems crucial to guiding the inference of a sufficiently precise invariant. Second, in all cases where both `ClumpG` and `CanonG` succeed, `ClumpG` uses fewer disjuncts in the loop invariant, and the difference can be quite significant (e.g., for javl-insert, 3 versus 18). And interestingly, `ClumpG` is always able to get to a single disjunct at the exit point while proving the post-condition.

From these observations, we hypothesize that being able to clump into the minimal number of necessary disjuncts is crucial to analyzing data structure operations in a client program that makes several such calls. To test this hypothesis, we perform a controlled experiment by sequentially composing a given operation with itself 32 times and trying to prove the shape preservation invariant at the end. The resulting analysis is notably more complex than for the initial code, due to increasingly complex abstract states over call sequences. The results in Figure 11.2(b) show that `ClumpG` keeps the number of disjuncts constant and scales well, whereas `CanonG` and `None` suffer a significant slow-down in the cases where they do not fail.

11.4 Related Work on Silhouette-guided Clumping

Several generic techniques are used to handle disjunctions in static analyses. Disjunctive completion [CC79] adds support for disjunctions to an abstract domain, but never collapses them, and thus is too expensive in practice. Similarly, [GR98] introduces the least disjunctive basis of an abstract domain as a compact domain with the same disjunctive completion, though this construction does not minimize the size of the representation of the abstract elements. State partitioning [CC92, JHR99] attaches abstract states to specific sets of states, which effectively allows one to support disjunctive properties while providing a way to control disjunct numbers via the definition of partitions. Trace partitioning [HT98, RM07] achieves a similar result using information about traces. However, these studies provide frameworks, and do not address the problem of finding a criterion to clump disjuncts. Silhouettes provide such a criterion, based on the properties of join.

Shape analyses based on three-valued logic [SRW02], and on separation logic [BCO05, BCC^{+}07, CRN07, DPV11, BDES12] are known to require disjunctions. The same goes for array analyses [HP08]. Several strategies have been designed to limit the number of disjunctions. When summary predicates denote non-empty regions, possibly empty regions create an additional need for disjunctions. Thus, several proposals have been made to let summary predicates denote possibly empty regions [CRN07, YLB^{+}08, CCL11]. Canonicalization operators [LAS00, SRW02, BCO05, DOY06] collapse abstract states into a smaller, finite lattice, and thereby bound the number of disjuncts. While the analysis may use a larger lattice, precision after applying this operation is limited by the smaller lattice. Join operators [CRN07, YLB^{+}08] do not require a smaller lattice (and can therefore keep more information), but lack a mechanism to bound the cardinality of disjunctions. To apply join or canonicalization operators efficiently, several strategies have been designed. Arnold [Arn06] groups abstract states that satisfy some inclusion relation. Moreover, [YLB$^+$08] performs a partial join that groups disjuncts only when it syntactically verifies the absence of precision loss. In contrast, our work proposes a criterion based on an abstraction of abstract states, and on the fact that this abstraction is cheaper to compute. A more related study is that by Manevich [MSRF04], that extends TVLA [LAS00] with a grouping of three-valued abstract states based on a partial graph isomorphism.

Techniques to merge disjuncts have been developed in static analyses for numerical properties as well. For instance, a notion of affinity between polyhedra is used in [PC06] in order to decide whether they can be joined without too significant a precision loss. This approach is extended in [PTTC11] to deal with set properties. Bagnara [BHZ04] proposed a widening over disjunctions of polyhedra, that tests for the implication of abstract states to better bound the precision loss.

Existing off-line approaches for the parameterization of static analyses and abstract domains (such as the selection of the abstract domain to use, and of the abstract values to keep) include syntactic heuristics based on code patterns [BCC$^+$03], machine learning techniques [LTN11, OYY15], and semantic impact pre-analysis methods [OLH$^+$14]. The disjunct clumping problem is tied to the abstract states that arise during the analysis, and it is therefore not surprising that it requires an on-line abstraction of these states at analysis time.

Finally, we remark that other analyses that abstract structures with summaries [CCR14], heap abstractions [DDA10], rich type systems [KRJ09, RKJ10] or quantified logical assertions [GMT08] may benefit from adapted forms of silhouette abstraction when facing the disjunction problem.

11.5 Conclusion on Silhouette-guided Clumping

In this part of the thesis, we introduced *silhouettes* that abstract the abstract states. The information enclosed in the silhouettes proves useful not only to clump disjunctions of abstract states, but also to compute better abstract joins. These results were achieved by selecting a definition of silhouettes that provides a weak entailment check over abstract states, and thereby accurately characterizes abstract states that are likely to join well. This characterization is conservative with respect to the standard analysis algorithms: while it will always suggest clumping abstract states that can be joined precisely, it may also suggest clumping abstract states that cannot be joined precisely, although we never observed this behavior in our experiments. This situation is quite similar to that of a static analysis that can be proven sound but is conservative in theory, yet computes very precise results in practice. Our experimental evaluation confirms the effectiveness of the silhouette abstraction, which allows our implementation to verify a large collection of challenging benchmarks at a reasonable cost. The overhead inherent in computing silhouettes is outweighed by the benefits of increased precision and improved scalability.

Part IV

Conclusion

Chapter 12

Conclusion and Future Directions

12.1 Conclusion

Separation-logic based shape analysis relies on the separating conjunction ($*$) to describe disjoint properties of local memory regions and inductive predicates to summarize unbounded dynamic data structures within heaps. However, the local predicates are not expressive enough to describe global properties that some complex data structures rely on, and to partition disjunctions.

In Part II, we have set up a shape analysis that is able to cope with data structures with unbounded sharing. The analysis combines separation logic based shape abstractions and a *set abstract domain* that tracks pointer sharing properties. The real difficulties of the analysis lie in the *synthesis of set parameters* for summary predicates during folding operations, and the *non-local unfolding of summary predicates* that "jumps" to somewhere in an inductive predicate. To deal with these difficulities, we have specified two kinds of set parameters of inductive predicates, *head and constant*, and designed *an instantiation process* for synthesizing set parameters. We have implemented the analysis in the MemCAD static analyzer by augmenting an existing shape domain in MemCAD with a set domain. The advantages of the analysis include that the original domain structures of MemCAD remain mostly unchanged, the analysis is still parametric in the structure to verify, and the modularity of the analyzer is preserved. We have obtained positive results in analyzing a basic graph library, where graphs are described as adjacency lists.

In Part III, we have set up a general framework for disjunct clumping based on abstraction of abstract states, i.e., *silhouettes*. Generally, abstract states can be clumped if their silhouettes are similar according to a computable silhouette equivalence relation. We have defined and formalized an instance of this framework in separation logic-based shape analysis, where silhouettes simply abstract access path relations between live program variables. It turns out that silhouettes apply not only to the clumping of disjuncts but also to the weakening of separating conjunctions of memory predicates into inductive summaries. We have implemented this approach in the MemCAD analyzer and evaluated it on real-world C codes from existing libraries dealing with doubly-linked lists, red-black trees, AVL-trees and splay-trees. Our experimental evaluation shows that silhouette-based clumping is effective in keeping the size of disjunctions small while preserving the case splits that are required for the analysis to succeed at a reasonable cost.

12.2 Future Directions

Several interesting routes for future work could be explored.

Numerical properties of data structures. Invariants of data structures, e.g., binary search trees and red-black trees, consist not only of shape invariants but also of numerical invariants. In order to keep precise numerical properties of data structures, the work presented in paper [CR08] proposed inductive predicates with numerical parameters. However, automatically inferring numerical parameters for summary predicates during folding remains a huge challenge. Therefore, similarly to set parameters, identifying the kinds of numerical parameters and designing a precise *instantiation process* for the kinds of numerical parameters are necessary. In addition, a disjunct clumping that is based not only on shape properties but also on numerical properties should be explored. As a consequence, silhouettes and silhouette equivalence relations for the clumping of different numerical abstractions should be designed.

Abstractions of DAGs. A directed acyclic graph (DAG) is a finite directed graph with a topological order of nodes, such that any node of the graph can only have out-edges to nodes later in the order. Our abstract domain does not provide a way to abstract the topological order of nodes, and thus cannot abstract DAGs precisely. As DAGs can be used to model many different kinds of data information, such as ordering the compilation operations in a makefile and representing sequences of binary choices, it is necessary to extend the domain in order to analyze programs that handle DAGs is necessary. Intuitively, the extension may require the set domain to be extended in order to abstract orders of sets.

Sharing properties of recursive procedures. Recursive procedures that manipulate dynamic data structures often lead to pointer sharing properties between the call stack and the dynamic data structure, and deferences of the data structure from the call stack. The sharing is usually unbounded. As our analysis is able to track unbounded pointer sharing properties with set abstractions and deferences of the data structure from the call stack might be handled by non-local unfolding, an application of our analysis to recursive procedures might be explored.

Clumping abstract states in other analyses. Similarly to shape analysis, array analysis (which deals with contiguous structures indexed by ranges of integers) and dictionary analysis (which handles structures indexed using sets of keys) also implicitly or explicitly use separation and summarization to abstract sets of concrete memory states, and rely on disjunctions for the sake of precision. In these analyses, deciding how to create summarization in joining and whether or not to keep case splits is also critical. Therefore, the disjunct clumping of abstract states based on silhouettes in these analyses might also be studied.

Bibliography

[And94] Lars O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, DIKU, 1994.

[App08] Andrew Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 2008.

[Arn06] Gilad Arnold. Specialized 3-valued logic shape analysis using structure-based refinement and loose embedding. In *Static Analysis Symposium (SAS)*, pages 204–220. Springer, 2006.

[BCC+03] 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 Ron Cytron and Rajiv Gupta, editors, *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, pages 196–207. ACM, 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 Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *LNCS*, pages 178–192. Springer, 2007.

[BCC+10] Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Static analysis and verification of aerospace software by abstract interpretation. In *AIAA Infotech@ Aerospace 2010*, page 3385. 2010.

[BCO05] Josh Berdine, Cristiano Calcagno, and Peter O'Hearn. Symbolic execution with separation logic. In *Asian Conference on Programming Languages and Software*, pages 52–68. Springer, 2005.

[BDES12] Ahmed Bouajjani, Cezara Drăgoi, Constantin Enea, and Mihaela Sighireanu. Abstract domains for automated reasoning about list-manipulating programs with infinite data. In *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 1–22. Springer, 2012.

[BHZ04] Roberto Bagnara, Patricia M Hill, and Enea Zaffanella. Widening operators for powerset domains. In *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 135–148. Springer, 2004.

[BS11] Thomas Ball and Mooly Sagiv, editors. *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. ACM, 2011.

[CBC93] Jong-Deok Choi, Michael G. Burke, and Paul R. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In Mary S. Van Deusen and Bernard Lang, editors, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, pages 232–245. ACM Press, 1993.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen, editors, *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, pages 269–282. ACM Press, 1979.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2&3):103–179, 1992.

[CCL11] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In Ball and Sagiv [BS11], pages 105–118.

[CCLR15] Arlen Cox, Bor-Yuh Evan Chang, Huisong Li, and Xavier Rival. Abstract domains and solvers for sets reasoning. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 356–371. Springer, 2015.

[CCR14] Arlen Cox, Bor-Yuh Evan Chang, and Xavier Rival. Automatic analysis of open objects in dynamic language programs. In *Static Analysis Symposium (SAS)*, pages 134–150, 2014.

[CCS13] Arlen Cox, Bor-Yuh Evan Chang, and Sriram Sankaranarayanan. Quic graphs: Relational invariant generation for containers. In *European Conference on Object-Oriented Programming*, pages 401–425. Springer, 2013.

[CDD$^+$15] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Methods Symposium*, pages 3–11. Springer, 2015.

[CDOY09] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *ACM SIGPLAN Notices*, volume 44, pages 289–300. ACM, 2009.

[CES86] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski, editors, *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 84–96. ACM Press, 1978.

[Cox15] Arlen Cox. Binary-Decision-Diagrams for Set Abstraction. *ArXiv e-prints*, March 2015.

[CR08] Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 247–260. ACM, 2008.

[CR13] Bor-Yuh Evan Chang and Xavier Rival. Modular construction of shape-numeric analyzers. In EPTCS, editor, *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday, Manhattan, Kansas, USA, 19-20th September 2013.*, volume 129 of *EPTCS*, pages 161–185, 2013.

[CRB10] Renato Cherini, Lucas Rearte, and Javier Blanco. A shape analysis for non-linear data structures. In *Static Analysis Symposium (SAS)*, pages 201–217. Springer, 2010.

[CRN07] Bor-Yuh Evan Chang, Xavier Rival, and George C. Necula. Shape analysis with structural invariant checkers. In Hanne Riis Nielson and Gilberto Filé, editors, *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*, volume 4634 of *Lecture Notes in Computer Science*, pages 384–401. Springer, 2007.

[CWZ90] David R. Chase, Mark N. Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In Bernard N. Fischer, editor, *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*, pages 296–310. ACM, 1990.

[DDA10] Isil Dillig, Thomas Dillig, and Alex Aiken. Symbolic heap abstraction with demand-driven axiomatization of memory invariants. pages 397–410. ACM, 2010.

[DDA11] I. Dillig, T. Dillig, and A. Aiken. Precise reasoning for programs using containers. In *Principles Of Programming Languages (POPL)*, pages 187–200, 2011.

[DES13] Cezara Drăgoi, Constantin Enea, and Mihaela Sighireanu. Local shape analysis for overlaid data structures. In *International Static Analysis Symposium*, pages 150–171. Springer, 2013.

[Deu94] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In Vivek Sarkar, Barbara G. Ryder, and Mary Lou Soffa, editors, *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*, pages 230–241. ACM, 1994.

[DOY06] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings*, volume 3920 of *LNCS*, pages 287–302. Springer, 2006.

[DPV11] Kamil Dudka, Petr Peringer, and Tomáš Vojnar. Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In *International Conference on Computer Aided Verification*, pages 372–378. Springer, 2011.

[DS07] David Delmas and Jean Souyris. Astrée: from research to industry. In *International Static Analysis Symposium*, pages 437–451. Springer, 2007.

[EC80] E Emerson and Edmund Clarke. Characterizing correctness properties of parallel programs using fixpoints. *Automata, Languages and Programming*, pages 169–181, 1980.

[FFJ12] Pietro Ferrara, Raphael Fuchs, and Uri Juhasz. TVLA+ : TVLA and value analyses together. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings*, volume 7504 of *Lecture Notes in Computer Science*, pages 63–77. Springer, 2012.

[Fil] Jean-Christophe Filliatre. Bdd ocaml library. `https://www.lri.fr/~filliatr/ftp/ocaml/bdd/`.

[GH96] Rakesh Ghiya and Laurie J Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–15. ACM, 1996.

[GMT08] Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting abstract interpreters to quantified logical domains. In *Principles Of Programming Languages (POPL)*, pages 235–246. ACM, 2008.

[GR98] Roberto Giacobazzi and Francesco Ranzato. Optimal domains for disjunctive abstract interpretation. *Science of Computer Programming*, 32(1):177–210, 1998.

[HHL$^+$15] Lukáš Holík, Martin Hruška, Ondřej Lengál, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. Forester: Shape analysis using tree automata. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 432–435. Springer, 2015.

[HHN92] Laurie J. Hendren, Joseph Hummel, and Alexandru Nicolau. Abstractions for recursive pointer data structures: Improving the analysis of imperative programs. In Stuart I. Feldman and Richard L. Wexelblat, editors, *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI), San Francisco, California, USA, June 17-19, 1992*, pages 249–260. ACM, 1992.

[HHR$^+$11] Peter Habermehl, Lukáš Holík, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. Forest automata for verification of heap manipulation. In *International Conference on Computer Aided Verification*, pages 424–440. Springer, 2011.

[HL11] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, pages 289–298. IEEE, 2011.

[HN90] Laurie J Hendren and Alexandru Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on parallel and distributed systems*, 1(1):35–47, 1990.

[HP08] Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 339–348. ACM, 2008.

[HT98] Maria Handjieva and Stanislav Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *Static Analysis Symposium (SAS)*, pages 200–214. Springer, 1998.

[JHR99] Bertrand Jeannet, Nicolas Halbwachs, and Pascal Raymond. Dynamic partitioning in analyses of numerical properties. In *Static Analysis Symposium (SAS)*, pages 39–50. Springer, 1999.

[JM82] Neil D Jones and Steven S Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 66–74. ACM, 1982.

[JM09] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer, 2009.

[Kle52] Stephen C. Kleene. *Introduction to metamathematics.* Bibliotheca Mathematica. North-Holland, Amsterdam, 1952.

[KRJ09] Ming Kawaguchi, Patrick Rondon, and Ranjit Jhala. Type-based data structure verification. In *Programming Languages Design and Implementation (PLDI)*, pages 304–315. ACM, 2009.

[KSV10] Jörg Kreiker, Helmut Seidl, and Vesal Vojdani. Shape analysis of low-level C with overlapping structures. In Gilles Barthe and Manuel V. Hermenegildo, editors, *Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings*, volume 5944 of *LNCS*, pages 214–230. Springer, 2010.

[LAS00] Tal Lev-Ami and Mooly Sagiv. Tvla: A system for implementing static analyses. In *Static Analysis Symposium (SAS)*, pages 280–301. Springer, 2000.

[LH88] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 21–34. ACM, 1988.

[LR15] Jiangchao Liu and Xavier Rival. Abstraction of arrays based on non contiguous partitions. In *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 282–299. Springer, 2015.

[LRC15] Huisong Li, Xavier Rival, and Bor-Yuh Evan Chang. Shape analysis for unstructured sharing. In *International On Static Analysis*, pages 90–108. Springer, 2015.

[LTN11] Percy Liang, Omer Tripp, and Mayur Naik. Learning minimal abstractions. In *Principles Of Programming Languages (POPL)*, pages 31–42. ACM, 2011.

[LYP11] Oukseh Lee, Hongseok Yang, and Rasmus Petersen. Program analysis for overlaid data structures. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 592–608. Springer, 2011.

[McC93] Steve McConnell. Code complete: a practical handbook of software construction (redmond, wa, 1993.

[MHKS08] Mark Marron, Manuel Hermenegildo, Deepak Kapur, and Darko Stefanovic. Efficient context-sensitive shape analysis with graph based heap models. In *International Conference on Compiler Construction*, pages 245–259. Springer, 2008.

[Min06] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[MRR05] Ana Milanova, Atanas Rountev, and Barbara G Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(1):1–41, 2005.

[MSRF04] Roman Manevich, Mooly Sagiv, Ganesan Ramalingam, and John Field. Partially disjunctive heap abstraction. In *Static Analysis Symposium (SAS)*, pages 265–279. Springer, 2004.

[NDQC07] H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated verification of shape and size properties via separation logic. In Byron Cook and Andreas Podelski, editors, *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 4349 of *Lecture Notes in Computer Science*, pages 251–266. Springer, 2007.

[OLH$^+$14] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective context-sensitivity guided by impact pre-analysis. In *Programming Languages Design and Implementation (PLDI)*, pages 475–484. ACM, 2014.

[oM12] Security TechCenter of Microsoft. Microsoft security bulletin ms12-063 - critical, 2012.

[oM13] Security TechCenter of Microsoft. Microsoft security bulletin ms13-008 - critical, 2013.

[OYY15] Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. Learning a strategy for adapting a program analysis via bayesian optimisation. pages 572–588. ACM, 2015.

[PC06] Corneliu Popeea and Wei-Ngan Chin. Inferring disjunctive postconditions. In *ASIAN*, pages 331–345. Springer, 2006.

[PTTC11] Tuan-Hung Pham, Minh-Thai Trinh, Anh-Hoang Truong, and Wei-Ngan Chin. Fixbag: A fixpoint calculator for quantified bag constraints. In *Conference on Computer Aided Verification (CAV)*, pages 656–662. Springer, 2011.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002.

[Ric53] Henry G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):pp. 358–366, 1953.

[RKJ10] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Low-level liquid types. In *Principles Of Programming Languages (POPL)*, pages 131–144. ACM, 2010.

[RM07] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Transactions on Programming Languages And Systems*, 29(5):26–69, 2007.

[Som99] Fabio Somenzi. Binary decision diagrams. *NATO ASI SERIES F COMPUTER AND SYSTEMS SCIENCES*, 173:303–368, 1999.

[SR12] Pascal Sotin and Xavier Rival. Hierarchical shape abstraction of dynamic structures in static blocks. In Ranjit Jhala and Atsushi Igarashi, editors, *Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings*, volume 7705 of *Lecture Notes in Computer Science*, pages 131–147. Springer, 2012.

[SRW99] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In Andrew W. Appel and Alex Aiken, editors, *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, pages 105–118. ACM, 1999.

[SRW02] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages And Systems (TOPLAS)*, 24(3):217–298, 2002.

[TCR13] Antoine Toubhans, Bor-Yuh Evan Chang, and Xavier Rival. Reduced product combination of abstract domains for shapes. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 7737 of *Lecture Notes in Computer Science*, pages 375–395. Springer, 2013.

[TCR14] Antoine Toubhans, Bor-Yuh Evan Chang, and Xavier Rival. An abstract domain combinator for separately conjoining memory abstractions. In Markus Müller-Olm and Helmut Seidl, editors, *Static Analysis Symposium (SAS)*, volume 8723 of *Lecture Notes in Computer Science*, pages 285–301. Springer, 2014.

[Vaf09] Victor Vafeiadis. Shape-value abstraction for verifying linearizability. In *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 335–348. Springer, 2009.

[Wal03] Julienne Walker. AVL balanced tree library, 2003. `http://www.eternallyconfuzzled.com/libs/jsw_avltree.zip`.

[WL95] Robert P Wilson and Monica S Lam. *Efficient context-sensitive pointer analysis for C programs*, volume 30. ACM, 1995.

[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 *Conference on Computer Aided Verification (CAV)*, pages 385–398. Springer, 2008.

Résumé

L'analyse statique des programmes permet de calculer automatiquement des propriétés sémantiques valides pour toutes les exécutions. En particulier, dans le cas des programmes manipulant des structures de données complexes en mémoire, l'analyse statique peut inférer des invariants utiles pour prouver la sûreté des accès à la mémoire ou la préservation d'invariants structurels. Beaucoup d'analyses de ce type manipulent des états mémoires abstraits représentés par des conjonctions en logique de séparation dont les prédicats de base décrivent des blocs de mémoire atomiques ou bien résument des régions non-bornées de la mémoire telles que des listes ou des arbres. De telles analyses utilisent souvent des disjonctions finies d'états mémoires abstraits afin de mieux capturer leurs dissimilarités. Les analyses existantes permettent de raisonner localement sur les zones mémoires mais présentent les inconvénients suivants:
(1) Les prédicats inductifs ne sont pas assez expressifs pour décrire précisément toutes les structures de données dynamiques, du fait de la présence de pointeurs vers des parties arbitraires (i.e., non-locales) de ces structures~;
(2) Les opérations abstraites correspondant à des accès en lecture ou en écriture sur ces prédicats inductifs reposent sur une opération matérialisant les cellules mémoires correspondantes. Cette opération de matérialisation crée en général de nouvelles disjonctions, ce qui nuit à l'impératif d'efficacité. Hélas, les prédicats exprimant des contraintes de structure locale ne sont pas suffisants pour déterminer de façon adéquate les ensembles de disjonctions devant être fusionnés, ni pour définir les opérations d'union et d'élargissement d'états abstraits. Cette thèse est consacrée à l'étude et la mise au point de prédicats en logique de séparation permettant de décrire des structures de données dynamiques, ainsi que des opérations abstraites afférentes. Nous portons une attention particulière aux opérations d'union et d'élargissement d'états abstraits. Nous proposons une méthode pratique permettant de raisonner globalement sur ces états mémoires au sein des méthodes existantes d'analyse de propriétés structurelles et autorisant la fusion précise et efficace de disjonctions.

Nous proposons et implémentons une abstraction structurelle basée sur les variables d'ensembles qui lorsque elle est utilisée conjointement avec les prédicats inductifs permet la spécification et l'analyse de propriétés structurelles globales. Nous avons utilisé ce domaine abstrait afin d'analyser une famille de programmes manipulant des graphes représentés par liste d'adjacence.

Nous proposons un critère sémantique permettant de fusionner les états mémoires abstraits similaires en se basant sur leur silhouette, cette dernière représentant certaines propriétés structurelles globales vérifiées par l'état correspondant. Les silhouettes s'appliquent non seulement à la fusion de termes dans les disjonctions d'états mémoires mais également à l'affaiblissement de conjonctions de prédicats de logique de séparation en prédicats inductifs.
Ces contributions nous permettent de définir des opérateurs d'union et d'élargissement visant à préserver les disjonctions requises pour que l'analyse se termine avec succès. Nous avons implémenté ces contributions au sein de l'analyseur MemCAD et nous en avons évaluées l'impact sur l'analyse de bibliothèques existantes écrites en C et implémentant différentes structures de données, incluant des listes doublement chaînées, des arbres rouge-noir, des arbres AVL et des arbres ``splay''. Nos résultats expérimentaux montrent que notre approche est à même de contrôler la taille des disjonctions à des fins de performance sans pour autant nuire à la précision de l'analyse.

Mots Clés

analyse statique, structures de données dynamique, logique de séparation, disjonctions

Abstract

Shape analyses rely on expressive families of logical properties to infer complex structural invariants, such that memory safety, structure preservation and other memory properties of programs dealing with dynamic data structures can be automatically verified. Many such analyses manipulate abstract memory states that consist of separating conjunctions of basic predicates describing atomic blocks or summary predicates that describe unbounded heap regions like lists or trees using inductive definitions. Moreover, they use finite disjunctions of abstract memory states in order to take into account dissimilar shapes. Although existing analyses enable local reasoning of memory regions, they do, however, have the following issues:
(1) The summary predicates are not expressive enough to describe precisely all the dynamic data structures. In particular,,a fairly large number of data structures with unbounded sharing, such as graphs, cannot be described inductively in a local manner;
(2) Abstract operations that read or write into summaries rely on materialization of memory cells. The materialization operation in general creates new disjunctions, yet the size of disjunctions should be kept small for the sake of efficiency. However, local predicates are not enough to determine the right set of disjuncts that should be clumped together and to define precise abstract join and widen operations. In this thesis, we study separating conjunction-based shape predicates and the related abstract operations, in particular, abstract joining and widening operations that lead to critical modifications of abstract states. We seek a lightweight way to enable some global reasoning in existing shape analyses such that shape predicates are more expressive for abstracting data structures with unbounded sharing and disjuncts can be clumped precisely and efficiently.

We propose a shape abstraction based on set variables that when integrated with inductive definitions enables the specification and shape analysis of structures with unbounded sharing. We implemented the shape analysis domain by combining a separation logic-based shape abstract domain of the MemCAD analyzer and a set abstract domain, where the set abstractions are used to track unbounded pointer sharing properties. Based on this abstract domain, we analyzed a group of programs dealing with adjacency lists of graphs.

We design a general *semantic criterion* to clump abstract memory states based on their *silhouettes* that express global shape properties, \ie, clumping abstract states when their silhouettes are similar. Silhouettes apply not only to the conservative union of disjuncts but also to the weakening of separating conjunctions of memory predicates into inductive summaries. Our approach allows us to define union and widening operators that aim at preserving the case splits that are required for the analysis to succeed. We implement this approach in the MemCAD analyzer and evaluate it on real-world C libraries for different data structures, including doubly-linked lists, red-black trees, AVL-trees and splay-trees. The experimental results show that our approach is able to keep the size of disjunctions small for scalability and preserve case splits that takes into account dissimilar shapes for precision.

Keywords

static analysis, dynamic data structures, separation logic, disjunctions