UNIVERSITE PARIS DIDEROT-PARIS 7

ECOLE DOCTORALE DE SCIENCES MATHEMATIQUES DE PARIS CENTRE

EAST CHINA NORMAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND SOFTWARE ENGINEERING

Techniques for Formal Modelling and Verification on Dynamic Memory Allocators

Bin FANG

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

Thèse soutenue à Shanghai le Sep 10, 2018

Declaration

I, Bin FANG, declare that this thesis submitted in partial fulfilment of the requirements for the conferral of the degree of Doctor of Philosophy in Computer Science, from the UNIVERSITE PARIS DIDEROT–PARIS 7 and EAST CHINA NORMAL UNIVERSITY is wholly my own work unless otherwise referenced or acknowledged. This document has not been submitted for qualifications at any other academic institution.

Bin FANG
July 11, 2018

Abstract

This thesis contributes to the formal specification and verification of Sequential Dynamic Memory Allocators (SDMA, for short), which are key components of operating systems or standard libraries. SDMA manage the heap part of the data segment of processes. Their implementations employ both complex data structures and low-level operations manipulating the memory. This thesis focuses on SDMA using list data structures to manage the heap chunks available for allocation, i.e., also called free-list SDMA.

The first part of the thesis demonstrates how to obtain formal specifications of free-list SDMA using a refinement-based approach. The thesis defines a hierarchy of models ranked by the refinement relation that capture a large variety of techniques and policies employed by real-work SDMA. This hierarchy forms an algorithm theory for the free-list SDMA and could be extended with other policies. The formal specifications are written in Event-B and the refinements have been proved using the Rodin platform. The thesis investigates applications of the formal specifications obtained, such as model-based testing, code generation and verification.

The second part of the thesis defines a technique for inferring precise invariants of existing implementations of SDMA based abstract interpretation. For this, the thesis defines an abstract domain representing sets of states of the SDMA. The abstract domain is based on a fragment of Separation Logic, called SLMA. This fragment captures properties related with the shape and the content of data structures used by the SDMA to manage the heap. The abstract domain is defined as a specific product of an abstract domain for heap shapes with an abstract domain for finite arrays of locations. To obtain compact elements of this abstract domain, the thesis proposes an hierarchical organisation of the abstract values: a first level abstracts the list of all chunks while a second level selects only the chunks available for allocation. The thesis defines transformers of the abstract values that soundly capture the semantics of statements used in SDMA implementations. A prototype implementation of this abstract domain has been used to analyse simple implementations of SDMA.

Résumé

Cette thèse est une contribution à la spécification et à la vérification formelles des allocateurs de mémoire dynamiques séquentiels (SDMA, en abrégé), qui sont des composants clés des systèmes d'exploitation ou de certaines bibliothèques logiciel. Les SDMA gèrent la partie tas de la mémoire des processus. Leurs implémentations utilisent à la fois des structures de données complexes et des opérations de bas niveau. Cette thèse se concentre sur les SDMA qui utilisent des structures de données de type liste pour gérer les blocs du tas disponibles pour l'allocation (SDMA à liste).

La première partie de la thèse montre comment obtenir des spécifications formelles de SDMA à liste en utilisant une approche basée sur le raffinement. La thèse définit une hiérarchie de modèles classés par la relation de raffinement qui capture une grande variété de techniques et de politiques employées par le implémentations réelles de SDMA. Cette hiérarchie forme une théorie algorithmique pour les SDMA à liste et pourrait être étendue avec d'autres politiques. Les spécifications formelles sont écrites en Event-B et les raffinements ont été prouvés en utilisant la plateforme Rodin. La thèse étudie diverses applications des spécifications formelles obtenues: le test basé sur des modèles, la génération de code et la vérification.

La deuxième partie de la thèse définit une technique de vérification basée sur l'interprétation abstraite. Cette technique peut inférer des invariants précis des implémentations existantes de SDMA. Pour cela, la thèse définit un domaine abstrait dont les valeurs representent des ensembles d'états du SDMA. Le domaine abstrait est basé sur un fragment de la logique de séparation, appelé SLMA. Ce fragment capture les propriétés liées à la forme et au contenu des structures de données utilisées par le SDMA pour gérer le tas. Le domaine abstrait est défini comme un produit spécifique d'un domaine abstrait pour graphes du tas avec un domaine abstrait pour des sequences finies d'adresses mémoire. Pour obtenir des valueurs abstraites compactes, la thèse propose une organisation hiérarchique des valeurs abstraites : un premier niveau abstrait la liste de tous les blocs mémoire, alors qu'un second niveau ne sélectionne que les blocs disponibles pour l'allocation. La thèse définit les transformateurs des valeurs abstraites qui capturent la sémantique des instructions utilisées dans les implémentations des SDMA. Un prototype d'implémentation de ce domaine abstrait a été utilisé pour analyser des implémentations simples de SDMA.

Acknowledgements

The four-year scholarly pursuit of a Ph.D. in ECNU and Pairs-7 has been the greatest challenge in my life. It is my privilege to thank all the people that have influenced this endeavor. I am deeply indebted to my advisors, Mihaela Sighireanu, Geguang Pu and Ahmed Bouajjani, who have been the consistent sources of research guidance and assistance during my phd studying.

I would like to thank Jifeng He, Abrial Jean-Raymond, Peter Habermehl, Juliusz Chroboczek, Ting Su, Jianwen Li, Suha Orhun Mutluergil, Wen Su, Lei Qiao for their valuable researching suggestions. I would like to thank the Antoine Miné and Stephan Merz for their time and for their thoughtful comments that helped me improve this manuscript. Also, I would like to thank all the members of the jury for accepting to participate to the defense of my Ph.d thesis.

I would like to thank all the friends who I have met in IRIF (Paris) and LAB301 (Shanghai). And I owe much thanks to my parents. They have unceasingly supported and encouraged me.

Contents

CHAPTER 1

Introduction

In recent years, software plays an important role in our daily life. Because software is becoming more and more complex, building software without bugs becomes a difficult task. Especially for safety-critical systems, such as flight control systems or driverless cars, potential errors in the programs can lead to catastrophic consequences. Therefore, it is important to detect errors in the programs or prove that there are no bugs.

Obtaining correct software is not so simple. In industry, the most widely used method for ensuring the quality of software is testing. It checks informally the conformance of the software executions under the given inputs. However, testing cannot exhaust all possibilities, so it cannot eliminate all potential errors in the programs. Unlike software testing, formal methods use mathematical models and logics to analyze the programs. The formal verification of software is an active research field in computer science. Various technologies have been proposed, mainly including the following categories:

- **Deductive program verification** constructs a set of mathematical proof obligations based on the given specification (e.g., loop invariants, function contracts, termination proofs) which the programs must obey. The correctness of the programs is guaranteed by proving the obligations. They are discharged using either automated SMT solvers [BCD$^+$11, DMB08] or interactive theorem provers [NPW02, BC13].

- **Model checking** [McM93] explores exhaustively and automatically the models abstracting a program to decide if its executions satisfy the desired properties. This method faces several challenges, such as to handle states explosion [CGJ+01] and to design automatic transformation from programs to their models.

- **Static analysis** works on an abstraction of a program and is performed without actually executing programs. This method is sound, that is it never returns a false negative. However, it is restricted to properties over decidable domains. Abstract interpretation [CC76, CC77b] is a framework providing means to construct sound over-approximations of the programs.

1.1 Motivation

This thesis focuses on the formal modelling and verification of the dynamic memory allocators. Heap management is an indispensable module in many operating system kernels, and its correctness has a major impact on the entire system. Memory management is also a module where errors occur frequently in the system.

This thesis focuses on sequential dynamic memory allocator, i.e., memory allocators with no support for concurrent requests for memory. More precisely, a SDMA is a piece of software managing a reserved region of the program memory. It appears in general purpose libraries (e.g., C standard library) or as part of applications where the dynamic allocation shall be controlled to avoid failure due to memory exhaustion (e.g., embedded critical software). A client program interacts with the SDMA by requesting some amount of memory that it may free at any time. To offer this service, the SDMA manages the reserved memory region by partitioning it into variable or fixed sized memory blocks, also called *chunks*. When a chunk is allocated to a client program, the SDMA can not relocate it to compact the memory region (like in garbage collectors) and it is unaware about the kind (type or value) of data stored.

The existing implementations of SDMA use various data structures to manage the set of chunks created in the memory region. In this thesis, we focus on SDMA that record all chunks using a list, also called *heap list*. In this data structure, the chunks are stored in the increasing order of their start address and the successor relation between chunks is computed from some information stored in the start of the chunk, e.g., the size of the chunk. Notice that this data structures allows to manage both fixed or variable size chunks. To speed the allocation of a free chunk, SDMA index the set of chunks not in use (*free chunks*) in an additional data structure. We focus here on *free list allocators* [Knu73, WJNB95a] that record free chunks in a list. This class of list based SDMA is widespread and it includes textbook examples [Knu73, KR88] and real-world allocators [Lea12].

The aim of this thesis is to contribute at the design of SDMA with efficient and correct code. For this, the thesis considers two approaches. The first approach is building software correct-by-construction. This thesis defines a general framework for formal specification of existing efficient techniques employed by SDMA. The second approach is checking correctness by static analysis. This thesis designs an original static analysis that is able to infer and check complex invariant properties of SDMA.

The first aim is a challenging task for several reasons. Firstly, there is no optimal general solution to obtain SDMA that provide both low overhead for the management of the memory region and high speed in satisfying memory requests, as demonstrated in the survey [WJNB95a]. Consequently, the design of a SDMA shall take into account its specific use and adjusts the combination of techniques to obtain an optimal solution for this use. This leads to a wide variety of SDMA implementations to be specified and proved correct. Secondly, the formal methods used to prove correctness shall deal with such optimized implementations which are usually combining low level code (e.g., pointer arithmetics, bit fields) with efficient high level data structures (e.g., hash tables with doubly linked lists). The difficulty to formally analyse particular DMA implementations has been demonstrated by several projects [CDOY06, MAY06, TKN07a, KEH$^+$09, Chl11]. These projects make use of highly expressive logics to specify the memory organisation and content, e.g., second order logics or Separation Logic [ORY01], which need sophisticated tools to be dealt with. Finally, there is no evidence that the techniques developed in these projects may be applied to verify the correctness of SDMA implementations using different customizations.

The automated analysis of SDMA faces several challenges. Although the code of SDMA is not long (between one hundred to a thousand LOC), it is highly optimised to provide good performance. Low-level code (e.g., pointer arithmetics, bit fields, calls to system routines like `sbrk`) is used to manage efficiently (i.e., with low additional cost in memory and time) the operations on the chunks in the reserved memory region. At the same time, the free list is manipulated using high level operations over typed memory blocks (values of C structures) by mutating pointer fields without pointer arithmetic. The analyser has to deal efficiently with this *polar usage of the heap* made by the SDMA. The invariants maintained by the SDMA are complex. The memory region is organised into a *heap list* based on the size information stored in the chunk header such that chunk overlapping and memory leaks are avoided. The start addresses of chunks shall be aligned to some given constant. The free list may have complex shapes (cyclic, acyclic, doubly-linked) and may be sorted by the start address of chunks to ease free chunks coalescing. A precise analysis shall keep track of both numerical and shape properties to infer specifications implying such invariants for the allocation and deallocation methods of the SDMA.

1.2 Contributions

Refinement-based formalization: The work of the first part [FS17, FSG+17] of this thesis is a first step towards providing optimal and formally proved correct SDMA implementations. We adopt a correct-by-construction approach, which is different from most of research this area. In this approach, an abstract model is gradually refined to obtain a model that is detailed enough for code generation or code annotation. We apply this approach to the full class of list based SDMA. We obtain a set of formal models organised in a hierarchy ranked by refinement relations that establishes a formally specified taxonomy of the techniques employed by the implementations of list based SDMA. This formally specified taxonomy forms an *algorithm theory* [SL90] for the free list SDMA i.e., a structure common to all implementations in this class, which abstracts away specific implementation concerns. To limit the complexity of this work, we consider SDMA without support for concurrency, i.e., used in a sequential setting.

Our work has a more theoretical consequence. It reveals the class of logics necessary to specify precisely each of the design tactics considered and thus it is a useful guide for the formal verification of SDMA. For example, we identified the technique that requires second order logic to capture its precise state invariant: the use of a list of free chunks which is not sorted by the start address of chunks. Excepting this technique, the models proposed use only first order, universally quantified state invariants, which is a good class for automatic provers.

Logic-based abstract interpretation for SDMA: We propose a static analysis [FS16] that is able to infer the complex invariants of SDMA on both heap list and free list. We define an abstract domain which uses logic formulas to abstract SDMA configurations. The logic proposed extends the fragment of symbolic heaps of SL with a hierarchical composition operator, $\Supset$, to specify that the free list covers partially the heap list. This operator provides a hierarchical abstraction of the memory region under the SDMA control: the low-level memory manipulations are specified at the level of the heap list and propagated in a way controlled by the abstraction at the level of the free list. The shape specification is combined with a fragment of first order logic on arrays to capture properties of chunks in lists, similar to [BDES11]. This combination is done in an accurate way as regards the logic by including sequences of chunk addresses in the inductive definitions of list segments. To summarize, the main contributions of this thesis are:

- We formalize a hierarchy of models for a set of SDMA. The hierarchy is ranked by formally proved refinement relations and it includes complete and sound specifications of existing SDMA implementations. Although extensible to other design tactics, our hierarchy covers actually all the techniques used for the management heap lists.
- We propose an algorithm theory for SDMA and identify a signature representing an abstraction from implementation details of SDMA and link the formal models proposed with the concrete implementations.
- We illustrate the application of this work to model-based code generation, testing and verification techniques.
- We propose a Separation Logic fragment SLMA for expressing properties of SDMA concerning the shape of the memory structures as well as their sizes and the data values they contain. The logic contains an array logic fragment and we show that this logic is undecidable in general because of the undecidable array logic fragment.
- We propose an abstract domain whose elements are based on the subset of the logical formulae of SLMA. we give the a sound approximation of

the logic entailment for static analysis of SDMA. The abstract domain gives a high precision of the abstraction which is able to capture complex properties of SDMA implementations. The abstract domain is built in a modular way which permit to reuse existing abstract domains for the analysis of linked lists with integer data.

- We implemented the abstract domain in a plug-in of the Frama-C [KKP+15] platform and applied it to some simple SDMA.

1.3 Organisation

This thesis is organised as follows.

Part I describes the formalization of SDMA. It contains three chapters. Chapter 2 overviews the existing SDMA, explains their features and summarizes the challenges for formalization. Chapter 3 introduces preliminaries of the specification we used and presents the most abstract specification for SDMA. Also the formalization strategy is given which contains the refinement steps and principles used in the formalization. The last chapter of the first part Chapter 4 details the process of the refinement-based formalization, gives the related work and concludes this part.

Part II focuses on the static analysis of SDMA. The analysis is based on the framework of abstract interpretation. In Chapter 5, the separation logic fragment, SLMA, is defined and the properties of the logic fragment are studied. Chapter 6 introduces the preliminaries of abstract interpretation and defines the abstract domain used in the analysis. The abstract domain is built by combining other domains to capture complex properties. The elements of the abstract domain are represented by the formulae of SLMA. Chapter 7 presents the basic abstract operations of the domain and the abstract transformers for assignments and condition tests. The analysis algorithm and the results of experiments are also given in this chapter. Chapter 7 finally discusses the related work and gives the conclusion of the work of this part.

The final chapter, Chapter 8, concludes this thesis and gives future directions of research.

Part I

FORMALIZATION OF SEQUENTIAL DYNAMIC MEMORY ALLOCATORS

CHAPTER 2

Sequential Dynamic Memory Allocators

The first part of this thesis focuses on formal modelling of sequential allocators with no support for concurrent requests for memory. Such allocators are used either in sequential systems or integrated inside a concurrent system that ensures exclusive access to the memory allocator functions. A wide variety of SDMA have been proposed until now, [WJNB95b] surveys most of them. This variety is a consequence of the heterogeneity of systems employing SDMA, each such system having its own requirements on memory allocation. For example, Real-Time Operating Systems (RTOS) demand allocators with fast reply to memory requests. The SDMA designed for RTOS may consume more memory to reduce its response time. Embedded systems usually employ a limited amount of memory and require to satisfy as much requests as possible within the assigned memory. Therefore, the SDMA spend some time in avoiding memory fragmentation such that the available memory is used at its maximal capacity. Formalizing the SDMA requires to deal with such a diversity of policies.

This chapter is structured as follows. We first overview basic components of SDMA in § 2.1, including their interface and different techniques used in existing implementations. Second, we describe a taxonomy of SDMA in § 2.2. We classify SDMA according to the policies they adopted to manage memory blocks and fragmentation. Then, we detail a wide set of case studies in § 2.3. It covers a set of existing techniques. Finally, § 2.4 discusses the challenges for the formalization of SDMA.

2.1 Basic Elements of SDMA

2.1.1 Memory Structure

Without loss of generality, we consider that a SDMA manages *one memory region* which corresponds to a contiguous sequence of bytes starting from a fixed address, denoted by hst (for *heap start*) and ending before some address, denoted by hli (for *heap limit*). (The case where the SDMA manages several such atomic regions is a simple generalization. We also abstract away very low level mechanisms like memory virtualization.)

Inside the managed memory region, the SDMA maintains a set of pairwise disjoint *chunks*, which correspond to a sub-region used to satisfy client requests for memory. Figure 2.1 illustrates a memory region which contains five chunks. A chunk includes two kinds of information: a control part used to store information about the chunk (e.g., its size, its status – free or busy) and a data part used to store the client's data. Figure 2.1 illustrates the case where the control part is stored at the start of the chunk, as a *chunk header* and the data part, called *chunk body* occupies the remainder of the chunk. The data part shall be a contiguous sub-region inside the chunk. The size of the control part is fixed for a given SDMA implementation. Thus, the start and the end addresses of the chunk body are determined from the start address of the chunk, the size of the chunk header and the chunk size.

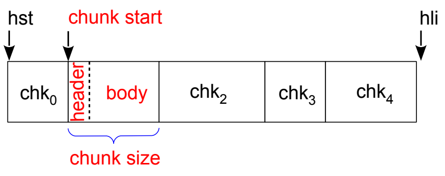


Figure 2.1: A memory region with five chunks

2.1.2 Interface for Clients

A memory allocator usually provides a method to clients to perform requests for a memory sub-region of some size. To optimize the usage of the memory resource, the allocator may provide a method to mark as not more in use by a client (or free) a memory sub-region that has been previously acquired.

Allocators providing both allocate and free methods are called *explicit allocators*. Some programming languages (e.g., Java, Lisp) provide only constructs for dynamic memory allocation but not for memory deallocation. Instead, they include a garbage collector [JL96] that recovers the memory not more in use. We call allocators with no deallocation method as *implicit allocators*. This thesis focuses on explicit SDMA, like the one used for C programming language.

The interface provided by the SDMA is shown in Figure 2.2. The method `init` initializes the set of chunks and marks them all to be *free*. A call `alloc(n)` searches a free chunk whose body has size (in bytes) at least `n`. If such a free chunk is found, it is marked as *busy* and the call to `alloc` returns the start address of the chunk body; otherwise, the call returns a fixed value denoted in the following by nil. A call `free(p)` succeeds if `p` is the start address of the body of a previous allocated (i.e., busy) chunk; the chunk is marked as *free* and the call returns true. Otherwise, the call does nothing and returns false. The size of a busy chunk of body starting at `p` can be changed to `n` using `realloc(p,n)`. If `n` is smaller than the size of the body starting at `p`, a new free chunk is created at the end of the chunk of `p` and the returned value is `p`. Otherwise, `realloc` either enlarges the chunk of `p` if there is enough free memory after it and returns `p`, or it allocates a new chunk, frees the chunk of `p` after copying its body in the new chunk, and returns the start address of the body in the new chunk. Some allocators may provide other functions. For example, the standard C library provides `calloc`, which allocates and initializes the allocated memory to zero (while `alloc` leaves the memory uninitialized). In this thesis, we focus on the interface shown in Figure 2.2 because these are the basic functionalities expected for SDMA.

```
void init(); //initialization
bool free(void* p); //deallocation
void* alloc(size_t sz); //allocation
void* realloc(void* p, size_t sz); //change size of p
```

Figure 2.2: Interface of SDMA

2.1.3 Interface Implementation

To introduce the main building blocks of the implementation of SDMA, we present two possible algorithms for allocation and deallocation in the Algorithm 1 and Algorithm 2, respectively. Each step of these algorithms is encoded by a function call. The result of the function is assigned (using the left arrow) to a variable or several variables composing a tuple. In these algorithms, we denote by M the set of memory chunks managed by the allocator.

2.1.3.1 Allocation

The first step of allocation algorithm, presented in Algorithm 1, is the search for a free chunk of size bigger than req. It is encoded in by the function `search`. The result of the search is assigned to a set, denoted by C. The next step is to select a free chunk from C if C has more than one suitable free chunk. Various selection strategies are described in Section 2.2.3.2. Moreover, some allocators combine the search and selection steps in a unique step. We present these variations in Section 2.2.3.

The `setBusy` function updates the state of the input chunk to *busy*. If the size of the selected chunk is larger than the request, the `split` function splits it into two parts: the first part, assigned to res, is for this allocation and the second part (assigned to $newChunk$) is a new free chunk. Not all SDMA split the selected chunk to fit the memory request because splitting may lead to memory fragmentation. The various choices for splitting in existing SDMA are explained in Section 2.2.2.

The set of memory chunks M is updated by adding the new chunk using the function `addChunk`. The algorithm ends by returning the start address of the selected chunk's body, as it is computed by the function `getBody`.

2.1.3.2 Deallocation

An algorithm for deallocation is presented in Algorithm 2. A valid input for this algorithm is the start address of the body of a busy chunk. This check is done using function `valid`, which also computes the start address of the chunk based on the the size of chunk header. The simplest deallocation only updates the status of the chunk to be free and stops. Deallocation may lead to a memory state with continuous free chunks, which increases the external fragmentation (elaborated in Section 2.2.2). To avoid such states, some allocators coalesce adjacent free chunks, like in Algorithm 2. Other SDMA do not coalesce free chunks during deallocation, but they do it if there is not enough free memory during the deallocation. These strategies are discussed in Section 2.2.2.

Algorithm 1: Allocation Procedure

input : request with size req

output: 1. a start address p of a free chunk res's body and the body has the size bigger than or equal to the request req, or
2. nil if no memory left

```
begin
    $C \leftarrow$ search($\mathsf{M}, req$) ;
    if $C$ is not empty then
        $c \leftarrow$ select($C$);
        $\mathsf{M} \leftarrow$ setBusy($\mathsf{M}, c$);
        if the size of $c$'s body is larger than $req$ then
            $(res, newChunk) =$ split($c, req$) ;
            $\mathsf{M} \leftarrow$ addChunk($\mathsf{M}, newChunk$);
            return $p \leftarrow$ getBody($res$);
        else
            return $p \leftarrow$ getBody($c$);
        end
    else
        return $p \leftarrow$ nil;
    end
end
```

Algorithm 2: Deallocation Procedure

input : an address p

output: True if deallocation is effectively done, False otherwise

```
begin
    $c \leftarrow$ valid($p$);
    if $c$ is not nil then
        $\mathsf{M} \leftarrow$ setFree($\mathsf{M}, c$);
        if $c$ has free neighbor then
            $\mathsf{M} \leftarrow$ merge($\mathsf{M}, c$);
        end
        return True;
    else
        return False;
    end
end
```

2.2 Taxonomy of SDMA

We categorize SDMA based on the different policies used in their implementations of the interface fixed in Figure 2.2. In general, these policies concern the implementations of basic functions called in the algorithms presented in the previous section. One class of policies concerns the data structure used to manage the set of chunks in the memory region and involved in functions like `addChunk` and `valid`. The second class concerns the decision of splitting or merging free chunks (functions `split` and `merge`) in order to cope with fragmentation. The third class of policies concerns the function `search` used to find a fitting set of chunks. The efficiency of this step depends on the data structures used to manage the set of free chunks.

All the policies discussed here have been proposed in reference work on dynamic memory allocators, e.g., [Knu73, PN77].

2.2.1 Heap List

The memory region managed by the SDMA is a sequence of chunks, as illustrated on Figure 2.1. We call this sequence a *heap list*. We discuss in the following the different ways used to manage this list.

Fixed-size chunk: The simplest design is to fix the size of chunks and therefore keep the sequence of chunks as an array [TKN07b]. This policy is suitable for systems where clients require memory with similar sizes. It allows moving forward and backward in the sequence of chunks but its degree of fragmentation can not be improved.

Explicit heap list: The sequence may be encoded by storing the start address of the successor of each chunk in its control part. For example, the header part of the chunk may have a specific field to store this information. Usually, the heap list is an acyclic list starting at the address hst and ending with address hli, the two global variables marking the boundaries of the memory region.

Implicit heap list: Instead of storing explicitly the address of the next chunk, some SDMA use address arithmetics to get the successor of a chunk. Indeed, for a chunk of start address c and size sz (in bytes), the the start address of its successor in the sequence is $c + sz$. Address arithmetics is also used in the technique of *boundary tag* [Knu73] to implement the sequence of chunks as a doubly-linked list. The control part of each chunk contains a header (first bytes of the chunk) and a footer (last bytes of the chunk), each of fixed size. The footer repeats the information on the current chunk, mainly its size and status (free or busy). This information is used by its successor (through address arithmetics), e.g., to obtain the start of the chunk.

Buddy system: Buddy systems [Knu73, PN77] employ memory regions of size 2^n for some n. Each chunk has size 2^i for different i such that $0 < i \leq n$ and start address multiple of its size. A *buddy* is a sequence of two chunks of equal size, say 2^k, and start addresses c_0 and c_1 such that $c_0 + 2^k = c_1$ and $c_0 \bmod 2^{k+1} = 0$, i.e., the first chunk of the buddy may be the start of a bigger chunk, more precisely of size 2^{k+1}. To satisfy a request of size smaller than its half, a chunk (or the initial memory region) is split in two chunks of equal size forming a buddy. Thus, the entire memory region respects this hierarchical division. Every chunk has a *level* given by the logarithm of its size. The biggest level is given by n and the smallest level is fixed by the allocator.

There are several variants of this method, called binary buddy. For example, Fibonacci buddy systems [WJNB95b] use the Fibonacci numbers for buddy sizes. We focus only on binary buddies.

2.2.2 Fragmentation

Fragmentation of memory region is a phenomenon in which the memory bytes not used in the allocated memory can not be used for allocation, they are wasted. This phenomenon can not be perfectly circumvented in most allocator algorithms. There are two kinds of fragmentation: *internal* and *external*. *Internal fragmentation* is local to a chunk. It occurs if the payload (the amount requested by the client) is smaller than the size of the data part of the chunk. The bytes of the data part not used by the client is wasted. The allocator generates internal fragmentation if either it does not apply chunk splitting, or there is no free chunk which exactly fits the requested size, or it does not choose the perfectly fitting chunk during selection (to speed-up allocation, for example). *External fragmentation* occurs when the allocator creates small free chunks by splitting, all of them not being contiguous. A request for allocation may be rejected even if the sum of sizes of these free chunks is bigger than the requested size. There is enough memory, but no single free chunk is large enough to fulfill the request.

We present below some policies proposed to manage fragmentation.

Splitting thresholds: To avoid internal fragmentation, the function `split` in Algorithm 1 is applied only if the difference between the size of the chunk and the requested size is bigger than some threshold. For this, the allocator defines the minimal ($minsz$) and maximal values ($maxsz$) for the requests. When a chunk of size s is selected to satisfy a request of size req, the allocator splits the chunk the chunk if $s - req$ is bigger than $minsz$ plus the size of the control part of the chunk. Setting a splitting threshold reduces external fragmentation while internal fragmentation increases. Buddy systems support limited splitting. If the request is bigger than the half of the selected chunk, the chunk is not split. Otherwise, the selected chunk is split into two chunks and so on until the chunk fits the request.

There are two ways to place the chunk obtained by splitting in the original one: place it at the small address i.e., at start, or at the big address, i.e., at end.

Eager coalescing: One important technique for defragmentation is to merge adjacent free chunks together into one bigger free chunk. In *eager coalescing* policy, this is done during deallocation (see Algorithm 2). When an allocated chunk c is released and it is marked as *free*, allocators with eager coalescing policy check the state of its neighbors (predecessor and successor). The chunk c is merged with its free neighbors immediately. Thus, the heap list never contains two adjacent free chunks.

Partial coalescing: In buddy systems, a free chunk can only be merged with its buddy. Because this may lead to status of the memory where two free chunks are neighbors (because not belonging to the same buddy), this coalescing policy is called *partial coalescing* in this thesis.

Lazy coalescing: Allocators with *lazy coalescing* policy allow the heap list to contain two adjacent free chunks and do not coalesce adjacent free chunks during deallocation. They decide to coalesce adjacent free chunks depending on some conditions. For example, some allocators apply coalescing at allocation, if there is no single free chunk larger enough for the request. In this case, the allocator coalesces all adjacent free chunks in the memory region. Then, it searches again for a free chunk. Other allocators coalesce adjacent free chunks when external fragmentation (i.e., number of free chunks) reaches a threshold after deallocation.

No coalescing: Some allocators never coalesce adjacent free chunks. This policy is suitable when clients always ask for similar size memory regions. It is usually combined with the encoding of the set of chunks by an array (see Section 2.2.1).

2.2.3 Free List

2.2.3.1 Data Structure

The procedure `search` in Algorithm 1 shall build the set of candidate free chunks that satisfy the request. To build this list, the procedure may traverse the heap list, policy that we denote by *implicit free list*. When the heap list is long and full of busy chunks, this traversal impacts negatively the efficiency of the search. To speed-up the search, some SDMA use additional data structures to index free chunks. This thesis focuses on the following cases:

Explicit free list: The free chunks can be collected inside a free list using additional information in the chunk control part, usually a field storing the address of the next free chunk in the list. This field is undefined for busy chunks. The free list may have several shapes: singly or doubly linked, acyclic or circular, etc. The free chunks may be kept in order of their start address in order to accelerate coalescing.

Segregated free list: To increase the speed in searching a free chunk fitting a request, allocators use an array of free lists, each free list holding free chunks within a *size class*. The set of size classes is fixed such that classes partition the interval of legal size requests. Given a request for a particular size, the allocator computes the size class the request belongs to, then it searches for a large enough free chunk in the corresponding free list. If the free list of this size class is empty, the allocator chooses next larger size class until a fitting chunk is found. If the selected free chunk is split to satisfy the request, the remainder free chunk is inserted into the free list storing chunks of this size class. Although the search phase of a fitting chunks is accelerated, the segregation of the free list may decrease the efficiency of defragmentation. Indeed, if an allocator needs to merge a released chunk with its free neighbor, the free neighbor is first removed from its free list, which requires a list traversal. After coalescing the two chunks, the new bigger free chunk is inserted into the free list (usually in constant time).

2.2.3.2 Fit Policy

The *fit policy* is used by the `select` procedure in the allocation Algorithm 1. It may be combined with the search procedure and therefore contribute to the efficiency of finding a fitting chunk. The fitting policy may influence the organization of the free list.

Best fit: To avoid internal fragmentation, some allocators use a *best fit* policy that finds the smallest free chunk satisfying the request. A best fit search is expensive if the search space is big. Buddy systems use a variant of the best fit. They divide a free memory region into many different partitions to try to fit the request as closed as possible.

First fit: This policy returns the first free chunk of the list which is large enough to satisfy the request. If this first chunk is too large, it may be split depending on the splitting policy. Therefore, the free chunks near the beginning of the free list are more likely to be allocated and split. Moreover, the small free chunks resulting from splitting will accumulate at the start of the free list [Knu73]. If the free list is kept in the last-in-first-out order, recently freed chunks tend to be reused quickly with the first fit policy.

Next fit: To avoid accumulation of small free chunks at the start of the free list, the technique of the *roving pointer* has been proposed by Knuth [Knu73]. The roving pointer records the position where the last search found a free chunk such that the next search or insert in the list will start from this position. This policy is usually combined with an organization of the free list as a cyclic list.

2.3 Case Studies

We collect thirteen case studies implementing SDMA using lists as data structure to collect chunks. We summarize them on Table 2.1. This collection of case studies appears to us as representative because the list based SDMA included illustrate all the policies discussed in the previous sections. First, we provide a short description of each case study in the increasing order of complexity. Second, we give hints on works proposing other implementations for list based SDMA.

2.3.1 Collected SDMA

We identified three main classes of case studies.

Pure heap list SDMA: The first part of Table 2.1 contains SDMA that manage memory chunks only by using the heap list and do not keep a list of free chunks.

TOPSY allocator [RJLP03] is the memory manager of the TOPSY operating system. It uses an explicit heap list to keep track of memory blocks. The allocator deals with internal fragmentation by defining a splitting threshold: a free chunk fitting the request is split if it has a size at least bigger than the request by the size of the control part of a chunk. The external fragmentation is dealt using a lazy coalescing policy. Some of its properties (e.g., heap list well formed, chunk separation) have been already specified and proved using Coq [MAY06].

The allocator published by Doug Lea [Lea12, LG96] is used in several contexts, for example C and C++ libraries. It contains a lot of mechanisms in order to offer an allocator that performs well at various kinds of uses. For example, for requests of memory with sizes less than 256 bytes, the allocation is done in an array of chunks of fixed size. We select this part of the allocator, that we call DL-small in Table 2.1, to illustrate the fixed size policy for the heap list. The other mechanisms implemented in this allocator are selected for the next class of case studies.

IBM allocator is provided at [Bar04]. It uses an implicit heap list. Therefore, the chunk header -contains only two fields: chunk's state and size. The memory region may be extended when the available free chunks can not satisfy the request. For this, the code employs a system call, the routine `sbrk`, which extends the data segment of the current process.

We implemented in C the buddy allocator described in [Knu73]. The header of each chunk stores the level of the chunk (which also gives its size), its buddy type (i.e., left or right buddy), and its state (i.e., free or busy). For an allocation request, the level of the requested size is computed (using the binary representation of the size), and a free chunk of this level is searched in the (implicit) heap list of the buddy system. The first free chunk of this level is chosen. If it does not exist, the selected chunk is the first free chunk with the smallest higher level; it is then split to obtain a free chunk with the level given by the request.

Table 2.1: Three lasses of case studies ("$\rightarrow$" and "$\leftrightarrow$" denote singly resp. doubly linked list; "explicit" and "implicit" denote explicit resp. implicit heap list; "at start" and "at end" denote the two possible positions of the free chunk left after splitting)

	heap list				*free list*		*fit*
Case study	*linked*	*split*	*defrag./extensible*	*array*	*shape*	*sorted*	*policy*
TOPSY [RJLP03]	implicit, $\rightarrow$	at end	lazy/no	–	–	–	first
DL-small [Lea12]	implicit, $\rightarrow$	–	–/no	yes	–	–	first
IBM [Bar04]	implicit, $\rightarrow$	–	–/yes	–	–	–	first
Buddy [Knu73]	explicit, $\leftrightarrow$	at start	partial/no	–	–	–	first
L4 [TKN07a]	implicit, $\rightarrow$	–	–/no	yes	acyclic, $\rightarrow$	yes	first
DKFF [Knu73]	explicit, $\rightarrow$	at start	early/no	–	acyclic, $\rightarrow$	yes	first
DKBF [Knu73]	explicit, $\rightarrow$	at start	early/no	–	acyclic, $\rightarrow$	yes	best
LA [Ald08]	explicit, $\rightarrow$	at start	early/no	–	acyclic, $\rightarrow$	yes	first
DKNF [Knu73]	explicit, $\rightarrow$	at start	early/no	–	acyclic, $\rightarrow$	yes	next
KR [KR88]	explicit, $\rightarrow$	at start	early/yes	–	cyclic, $\rightarrow$	yes	next
DKBT [Knu73]	explicit, $\leftrightarrow$	at start	early/no	–	acyclic, $\leftrightarrow$	no	best
DL-list [Lea12]	explicit, $\leftrightarrow$	at start	early/yes	–	acyclic, $\leftrightarrow$	no	best
TLSF [MRCR04]	explicit, $\leftrightarrow$	at start	early/no	–	acyclic, $\leftrightarrow$	no	best

SDMA with singly linked free lists: The case studies in the second part of Table 2.1 use singly linked and address sorted free lists.

L4 is the memory allocator of the L4 microkernel [TKN07a]. It manages fixed size chunks and collects the free chunks in an acyclic singly linked list. When all the chunks are busy, the memory managed by L4 allocator is not extended. Tuch et al. have specified this allocator and verified some of its properties (e.g., heap list, chunk separation) using Isabelle/HOL [TKN07b].

The allocators DKFF and DKBF are our implementations of algorithms A and B in the Section 2.5 of [Knu73], which use the first-fit resp. best-fit policies. The allocator DKNF is also our implementation of the next-fit policy using the "roving pointer" technique proposed in [Knu73] (Exercise 6 in Section 2.5). The three allocators use an explicit heap list and manage free chunks in a singly acyclic free list.

LA allocator [Ald08] is the implementation of Knuth's algorithm A (i.e., first fit policy) [Knu73] by Leslie Aldridge. It keeps an explicit heap list and prevents fragmentation by using an early coalescing policy.

The allocator KR is the code published in [KR88]. It keeps a circular singly linked list and uses next fit policy. The free list is sorted by the chunk start addresses up to the circular list. To exploit the locality of allocation/deallocation, the start of the free list points to the last deallocated block. When the available free chunks are not big enough to satisfy a memory request, the allocator calls the system routine `sbrk` to extend the size of the managed memory.

SDMA with doubly linked free list: The third part of Table 2.1 contains case studies that enhance the implementation of coalescing by using a doubly linked free lists.

The allocator DKBT is our C implementation of the "boundary tag" technique introduced in [Knu73]. The implementation in C of the footer control part uses the classic ruse which includes the footer of a chunk in the header of its successor. The footer we implemented stores the status and the start address of the previous chunk in the heap list. This information is generally used only if the previous chunk is free by the coalescing function `merge` in Algorithm 2.

The allocator DL-list is the part of the Doug Lea's allocator [Lea12] which manages medium size requests. It uses a doubly-linked list to store these chunks and the boundary tag technique where the footer stores the size of the owner chunk. We remove from this code the parts concerning concurrency, portability, and some optimizations.

The allocator TLSF is described in [MRCR04] and its code freely available. The free lists in TLSF are indexed in a matrix. A free chunk belongs to the doubly linked list stored at the position (i, j) in this matrix if its size belongs to the interval $[2^i + 2^{i-1} \times j, 2^i + 2^{i-1} \times (j + 1)[$. The search for the best fitting set of chunks is done in constant time due to the use of a bitfield storing the non empty classes in the matrix. Therefore, when a class is empty, the next non empty larger class is found in constant time. The allocator uses early coalescing policy.

2.3.2 Related Work

The study of dynamic memory allocation algorithms has been an important topic in the operating systems research. Wilson et al. [WJNB95b] survey a variety of SDMA designs between 1965 and 1995 and point out issues relevant to their design and evaluation. Puaut [Pua02] gives the detailed average and worst-case measurements of the timing performance of SDMA, including different classes of existing allocators (first-fit, best-fit, segregated-fit, buddy system). Barootkoob et al. [BKSM11] present the parameters affecting the functionality of memory allocators for a Berkeley Software Distribution (BSD) operating system. In [DSDR12] is given a survey report on memory allocation strategies for RTOS. Payer discusses the fragmentation problem of the allocators used in RTOS [Pay07] which offer $\mathcal{O}(1)$ operations' response times. Some works improve the existing algorithms. Sun et al. [SWC07] improve the TLSF allocator which is widely used in RTOS. [BDM05] presents some enhancements to the conventional buddy system that improve the running time of allocation and deallocation.

2.4 Challenges for Formalization

Variety of SDMA: As we showed in the previous sections, there is a large spectrum of policies and techniques adopted in the design of SDMA. Each SDMA takes into account its specific use and chooses the combination of techniques to obtain an optimal solution for this use. This leads to a wide variety of SDMA. The formal specification of these combination of policies and techniques is very complex.

Expressivity of formal methods: Some existing work presents modeling and verification results for particular allocator algorithms. For example, Tuch et al. [TKN07b] use the proof assistant Isabelle/HOL [NPW02] formal method to model and verify the L4 allocator. Also, Su et al. [SAPF15] adopt Event-B [AH07] to build a formal model for the TLSF allocator. The difficulty to capture the formal properties and to verify particular SDMA implementations has been also demonstrated by several projects [CDOY06, MAY06, KEH$^+$09, Chl11]. These projects make use of highly expressive logics to specify the memory organization and content, e.g., second order logics or Separation Logic [ORY01]. These logics need sophisticated tools to be dealt with. Each of these approaches is tailored for a unique allocator. There is no evidence that the techniques developed in these projects may be applied to verify the correctness of SDMA implementations using different customizations. In conclusion, the formal methods chosen shall be expressive enough to capture the large variety of customization which usually combines highly optimized low level code (e.g., pointer arithmetics, bit fields) with efficient high level data structures (e.g., hash tables with doubly linked lists).

Modular specification: The naive way to deal with a set of SDMA implementations is to construct an individual formal model for each allocator algorithm. However, this approach is not satisfactory and time-consuming. It is not modular because when building models for allocators separately, the common components cannot be shared between models. To design a generic framework for formalizing a set of SDMA, two techniques are necessary: *abstraction* and *refinement*. The challenge when employing these well understood techniques is the choice of the abstraction and of the refinement steps in order to reuse models as much as possible and cover all the SDMA in our case studies. In the next chapter, we give an overview of our approach for formalizing the set of SDMA we collected.

CHAPTER 3

Formalization Strategy

This chapter presents the formal method used to obtain formal specifications for the SDMA and the principles that guided us in choosing and using a formal method to obtain such specifications. We choose to employ the formal method called Event-B [AH07] which has been used for the specification and verification of several critical systems [BY08, ZYSL15, SA17, MPS17]. This formal method has two main ingredients: state machines and refinement relations.

The state machines model the behavior of the specified system by providing (i) the invariant properties of the state of the system (here the SDMA) and (ii) the events that produce atomic transformations of the state (here the components of the main SDMA functions – initialization, allocation and deallocation). The properties and events are written using the second order theory over sets. The preservation of the state invariants by the events is proved by the proof system Rodin [ABH+10]. We introduce in this chapter the state machine that is the most abstract specification of an SDMA.

The refinement relations allow to derive elaborate models in an incremental way. Intuitively, a state machine R refines a state machine B if R has more details than B, but R includes all behaviors of B. The refinement relation is given by the specification; it is formally checked by Rodin. In this chapter, we define a hierarchy of policies in SDMA and we use it to guide the definitions of refinement relations between specifications of SDMA.

We start by preliminary definitions about the formal method employed in § 3.1. The most abstract specification of SDMA is presented in § 3.2 and our refinement strategy in § 3.3.

3.1 Preliminaries

A formal model specifying a system is a mathematical representation of the system. We use the formal method Event-B [AH07] to obtain formal models of SDMA. We introduce hereafter the main ingredients of this method: the state machines and the refinement relations.

3.1.1 State Machine

A state machine in Event-B is a set of *variables* satisfying some *invariant properties* and a set of *events* that produces atomic transformations of variables.

The variables are typed using either basic types (boolean, natural, integer, real), product of types or typed sets (set of integers, relations between integers, etc.). A state of the state machine is a valuation of its variables. The machine may have finitely or infinitely many states. For example, the formal models of SDMA include variables that represent the memory region managed by the allocator. One of these variables is the set of integers representing the start addresses of chunks stored in the memory region.

Only states that satisfy the invariants properties over the state machine variables are included in the behavior of the state machine. The invariant properties are constraints about the range of values for some variables or more complex relations between variables.

An event (or *transition*) updates the state using a set of assignments on variables (called *action*) that modifies the variables simultaneously. Events are atomic transformations: there is no interleaving between the assignments of two events. To be executed, an event shall satisfy its *precondition*. If the preconditions of all actions are not satisfied, the machine stops; this situation is called the *deadlock* of the machine. If several events may be executed in some state, any of them can be chosen. The execution of an event shall lead to a state that satisfies the invariant properties. Otherwise, the event is not *correct*.

The proof system Rodin [ABH$^+$10] checks that the invariants properties are satisfiable (i.e., there exist states satisfying these constraints) and that any event specified is correct.

A detailed definition of the Event-B methods is given in [AH07]. We only shortly present some parts of it used in our formalization process as well as some shorthand notations we introduce for the clarity of presentation.

Underlying logic: The invariant properties, the preconditions and the actions are specified using formulas and terms in a multi-sorted second order logic including the set theory.

Table 3.1: Set-theoretical notations

	Set operators		
$\{e\}$	singleton set	$\{e_1, e_2, ..., e_n\}$	set enumeration
$\varnothing$	empty set	$S \cup T$	set union
$S \cap T$	set intersection	$S \setminus T$	set difference
$S \times T$	cartesian product	$\mathcal{P}(S)$	powset
$e_1 \mapsto e_2$	ordered pair	$m..n = \{i \mid m \leq i \leq n\}$	interval
	Set predicates		
$e \in S$	set membership	$S \subseteq T$	subset
$e \notin S$	set non-membership	$S \subset T$	strict subset
	Relations		
$S \leftrightarrow T$	relations: $\mathcal{P}(S \times T)$		
$dom(r)$	domain: $dom(r) \triangleq \{x \mid (\exists y \cdot x \mapsto y \in r)\}$		
$ran(r)$	range: $ran(r) \triangleq \{y \mid (\exists x \cdot x \mapsto y \in r)\}$		
r^{-1}	inverse: $r^{-1} \triangleq \{(y \mapsto x) \mid x \mapsto y \in r \}$		
$r[S]$	relational image: $r[S] \triangleq \{y \mid \exists x \cdot x \in S \land x \mapsto y \in r\}$		
$id(S)$	identity: $id(S) \triangleq \{(x \mapsto y) \mid x \in S \land y \in S \land x = y \}$		
$S \mathbin{\lhd\!\!\!-} r$	domain subtraction: $S \mathbin{\lhd\!\!\!-} r \triangleq \{(x \mapsto y) \mid x \mapsto y \in r \land x \notin S\}$		
$r \mathbin{\rhd\!\!\!-} T$	range subtraction: $r \mathbin{\lhd\!\!\!-} T \triangleq \{(x \mapsto y) \mid x \mapsto y \in r \land y \notin T\}$		
$r_1 \mathbin{\lhd\!\!\!+} r_2$	overriding: $r_1 \mathbin{\lhd\!\!\!+} r_2 \triangleq r_2 \cup (dom(r_2) \mathbin{\lhd\!\!\!-} r_1)$		
	Functions		
$q \circ p$	composition: $\forall q, p \cdot q \in S \leftrightarrow T \land p \in T \leftrightarrow U \Rightarrow$ $q \circ p = \{(x \mapsto y) \mid \exists z \cdot x \mapsto z \in q \land z \mapsto y \in p\}$		
$S \mathrel{+\!\!\!\!\!\rightarrow} T$	partial function: $S \mathrel{+\!\!\!\!\!\rightarrow} T \triangleq \{r \mid (r \subseteq id(T)) \circ (r \in S \leftrightarrow T \land r^{-1})\}$		
$S \rightarrow T$	total function: $S \rightarrow T \triangleq \{r \mid r \in S \mathrel{+\!\!\!\!\!\rightarrow} T \land dom(r) = S\}$		
$S \mathrel{\rightarrowtail\!\!\!\!\!\!+\;} T$	partial injection: $S \mathrel{\rightarrowtail\!\!\!\!\!\!+\;} T \triangleq \{r \mid r \in S \mathrel{\rightarrowtail\!\!\!\!\!\!+\;} T \land r^{-1} \in T \mathrel{+\!\!\!\!\!\rightarrow} S\}$		
$S \rightarrowtail T$	total injection: $S \rightarrowtail T \triangleq S \mathrel{\rightarrowtail\!\!\!\!\!\!+\;} T \cap S \rightarrow T$		
$S \mathrel{+\!\!\!\!\!\twoheadrightarrow} T$	partial surjection: $S \mathrel{+\!\!\!\!\!\twoheadrightarrow} T \triangleq \{r \mid r \in S \mathrel{+\!\!\!\!\!\rightarrow} T \land ran(r) = T\}$		
$S \twoheadrightarrow T$	total surjection: $S \twoheadrightarrow T \triangleq S \mathrel{+\!\!\!\!\!\twoheadrightarrow} T \cap S \rightarrow T$		
$S \mathrel{\rightarrowtail\!\!\!\!\!\twoheadrightarrow} T$	bijection: $S \mathrel{\rightarrowtail\!\!\!\!\!\twoheadrightarrow} T \triangleq S \rightarrowtail T \cap S \twoheadrightarrow T$		

The operators of the logic are classical: *disjunction* ($\vee$) *conjunction* ($\wedge$), *negation* ($\neg$), *universal quantification* ($\forall$), *existential quantification* ($\exists$), *implication* ($\Rightarrow$), *equivalence* ($\Leftrightarrow$), *true constant* ($\top$), and *false constant* ($\bot$). We denote by $P(X)$ a formula P with free variables in the set X.

The set theory includes operators, predicates, relations and functions presented in Table 3.1. In addition to these notations, we also introduce some shorthands for complex notions (e.g., relation overriding) in the following.

State machine: A state machine is a tuple $SM = (V, \mathcal{I}(V), E, \mathsf{e}_0)$ where:

- V is a set of *variables* defining the state of the machine,
- $\mathcal{I}(V)$ is a set of *invariants*, which are formulas in the underlying logic whose free variables are contained in V,
- E is a set of *events* representing state changes,
- $\mathsf{e}_0 \notin E$ is the *initialization event*, which fixes the initial set of states of the machine.

The variables in V that are not changed during the state machine behaviors are called *constants*. We identify them in specifications of state machines.

Event: An event e is specified using the following syntax:

$$\begin{aligned}
\mathsf{e} \;\triangleq\;\; & \textbf{begin any } X \textbf{ when } \mathcal{P}(X,V) \textbf{ then } \mathcal{A}(X,V) \textbf{ end} && (\mathsf{E1}) \\
\mid\; & \textbf{begin when } \mathcal{P}(V) \textbf{ then } \mathcal{A}(V) \textbf{ end} && (\mathsf{E2}) \\
\mid\; & \textbf{begin } \mathcal{A}(V) \textbf{ end,} && (\mathsf{E3}) \\
\mid\; & \mathit{SKIP} && (\mathsf{E4})
\end{aligned}$$

where X is the set of *local variables* of the event, $\mathcal{P}(V)$ (resp. $\mathcal{P}(X,V)$) are logic formulas specifying the *precondition* of the event by constraining V (resp. X and V) and $\mathcal{A}(V)$ (resp. $\mathcal{A}(X,V)$) is the *action part* which changes or only uses variables in V (resp. X and V). The most general form is given in (E1), which defines a list of local variables. The third form (E3) is for events whose precondition always holds, so it is omitted. A state machine has a unique event in form (E3), the initialization event e_0. The form (E4) is for the implicit event which does not modify variables.

Action: The action part of an event $\mathcal{A}(X,V)$ is a set of *assignments* of the following form:

$$\mathit{assign} \;\;::=\;\; x := \mathsf{t}(X,V) \;\mid\; r(x) := \mathsf{t}(X,V) \;\mid\; \mathsf{skip}$$

An assignment sets the variable x to the term t which has free variables in sets V and X. It also may assign the value bound to x in the relation variable r to a term t. Notice that this kind of assignment is a shorthand for the relation overriding symbol (e.g., $\Leftarrow\!\!+$) presented in Table 3.1, i.e., $r := r \Leftarrow\!\!+ r_1$.

The dynamic semantics of state machines is given in terms of labelled transition systems in [AH07]. We only recall hereafter the proof obligations to check the correctness of the state machine.

Before-after formulae: For each action, we define a formula that captures its effect on the state machine variables V. This formula uses a set of variables V' that duplicates the variables in V and represents the state of the machine after the execution of the action. For the action using only skip assignments, the before-after formula is $V' = V$, i.e., the conjunction of equalities between respective variables. For an action $\overrightarrow{x} := \overrightarrow{\mathsf{t}}(X, V)$, the before-after formula is $\overrightarrow{x} = \overrightarrow{\mathsf{t}}(X, V) \wedge \overrightarrow{y}' = \overrightarrow{y}$, where $\overrightarrow{y} = V \setminus \overrightarrow{x}$.

Invariant preservation: The set of invariants in a state machine $I(V)$ shall hold in every reachable state of the machine. Therefore, they shall be satisfied after the execution of the initial event e_0, i.e., the following formula shall be valid:

$$I(V) \wedge Q_{\mathsf{e}_0}(V, V') \Rightarrow I(V') \qquad \text{(INIT)}$$

where $Q_{\mathsf{e}_0}(V, V')$ is the before-after formula for the initial event e_0.

For an event $e \in E$ with precondition $P(X, V)$, the invariant is preserved by the execution of e if the following formula is valid:

$$I(V) \wedge P(X, V) \wedge Q_e(V, V') \Rightarrow I(V') \qquad \text{(INV)}$$

where $Q_e(V, V')$ is the before-after formula of e. The proofs of validity of INIT and INV belong to the set of proof obligations for the state machine correctness.

Absence of deadlock: A state machine $M = (V, I(V), E, \mathsf{e}_0)$ has a *deadlock* in some state if all its events are disabled. Assume that the set of events $E = (e_1, e_2, ..., e_n)$ are in the following form:

$$\mathsf{e}_i \quad \triangleq \quad \textbf{begin any } X_i \textbf{ when } \mathcal{P}_i(X_i, V) \textbf{ then } \mathcal{A}_i(X_i, V) \textbf{ end}$$

The proof obligation for deadlock-freedom (which is optional) is:

$$I(V) \Rightarrow (\exists X_1 \cdot \mathcal{P}_1(X_1, V)) \vee \cdots \vee (\exists X_n \cdot \mathcal{P}_n(X_n, V)) \qquad \text{(LOC)}$$

Example 1 (Set data structure)**.** We illustrate the state machine definition by providing the formal model for the set data structure. The set data structure is a container collecting a set of objects of the same type. Two operations over the collection are specified in this example: adding an element into and removing an element from the set. The machine, denoted by LM_0, is specified as follows:

variables	**invariants**
els	**inv1:** $els \subseteq \mathbb{N}$

initialisation e_0	$\triangleq$
act1:	$els := \varnothing$

```
Add  ≜
begin
  any    $x$
  when   $x \in \mathbb{N} \land x \notin els$
  then
    act1:  $els := els \cup \{x\}$
end
```

```
Remove  ≜
begin
  any    $x$
  when   $x \in els$
  then
    act1:  $els := els \setminus \{x\}$
end
```

We denote by variable els the set of elements. The invariant **inv1** constrains els to be a subset of natural numbers. The initial event e_0 assigns els to the empty set. The events Add and Remove specify respectively the operation of adding a new element into and removing an element from els. The proof obligations for invariant preservation of this machine are simple and can be easily proved using state of the art solvers connected at the Rodin platform. $\triangle$

Transition rules: For the sake of readability, we employ inference rules to specify events of state machines. For example, an event e with the local variables X, the pre-condition $\mathcal{P}(X, V)$ and the action defined by the set of assignments $\overrightarrow{x} := \overrightarrow{\mathsf{t}}(X, V)$ is specified using the following rule:

$$\mathsf{e}\ \frac{\mathcal{P}(X, V)}{s \xrightarrow{\mathsf{e}(X)} s[\overrightarrow{x} \leftarrow \overrightarrow{\mathsf{t}}(X, V)]} \qquad \text{(E-rule)}$$

where s denotes the state of the machine and $s[\overrightarrow{x} \leftarrow \overrightarrow{\mathsf{t}}(X, V)]$ represents a state update. Indeed, a state is a valuation of variables (i.e., a mapping from state machine variables to values) and the notation $\overrightarrow{x} \leftarrow \overrightarrow{\mathsf{t}}(X, V)$ denotes the update of the values for variables in $\vec{x}$ by the values computed in the respective term in $\overrightarrow{\mathsf{t}}(X, V)$. In some places, we use the simpler presentation, e.g., $s \xrightarrow{\mathsf{e}(X)} s'$, to describe the execution of the event $\mathsf{e}(X)$ of parameters in X. For $x \in X$, we denote by $s.x$ the value of x in the state s. The inference rule coding the initialization event e_0 of the state machine is described as follows:

$$\textsf{Initialization}\ \frac{}{s = [\overrightarrow{x} \leftarrow \overrightarrow{\mathsf{t}}]}$$

where $\overrightarrow{x}$ is the list of all variables of the state machine.

To illustrate this notation for events, we provide below the rules for the events Add and Remove specified in Example 1:

$$\textsf{Add}\ \frac{x \in \mathbb{N} \land x \notin els}{s \xrightarrow{\textsf{Add}(x)} s[els \leftarrow els \cup \{x\}]} \qquad \textsf{Remove}\ \frac{x \in els}{s \xrightarrow{\textsf{Remove}(x)} s[els \leftarrow els \setminus \{x\}]}$$

3.1.2 Refinement

The technique of *refinement* for system design has been introduced in [WIR83, Mor87, DREB98, BW12]. This technique is at the basis of the formal method Event-B and therefore it is intensively used to develop formal specifications with this method [AH07, Abr10].

A refinement is a verifiable transformation from an *abstract* model or specification into a *concrete* one. The transformation introduces some details in the concrete model that fix particular features or facets of the abstract model. For example, it may give the implementation details of the abstract model, therefore allowing to obtain an executable model. From now on, we limit our study to the refinement of state machines as it has been defined in Event-B [AH07].

Given two machines, $M^c(W, I^c(W), E^c, \mathsf{e}_0^c)$ and $M^a(V, I^a(V), E^a, \mathsf{e}_0^a)$, if M^c refines M^a, M^c can only behave in a way that corresponds to the behavior of M^a. The invariants in concrete machine can refer to the variables of the concrete and the abstract machine. If a invariant in the concrete machine refers to both, it is called *gluing invariant*. The gluing invariants are used to relate the states between the concrete and abstract machines. We assume that the invariant of both the concrete and abstract machines were valid before the event occurred.

To prove that M^c refines M^a, several conditions have to be satisfied. First, the set of variables in the concrete machine shall include the set of abstract variables, i.e., $V \subseteq W$. The second condition applies to events: there exists a mapping $\rho : E^c \twoheadrightarrow E^a$ that is surjective and injective on $\rho[E^a]$, that is, it maps every concrete event on an unique abstract event; the concrete events not mapped by ρ are implicitly mapped on SKIP. Moreover, this mapping shall satisfy additional constraints that ensure that the concrete event is a transformation of the abstract one.

Definition 3.1 (Event Refinement)**.** Let f $\in E^c$ and e $\in E^a$ be two events of the following form:

$$\begin{array}{lcll} \mathsf{e} & \triangleq & \textbf{begin any } X \textbf{ when } \mathcal{E}(X,V) \textbf{ then } \mathcal{R}(X,V) \textbf{ end} & (abstract) \\ \mathsf{f} & \triangleq & \textbf{begin any } Y \textbf{ when } \mathcal{F}(Y,W) \textbf{ then } \mathcal{S}(Y,W) \textbf{ end} & (concrete) \end{array}$$

The event f refines e with respect to the gluing invariant $J(V,W)$ iff the following constraints are valid:

$$I^a(V) \wedge J(V,W) \wedge \mathcal{F}(Y,W) \Rightarrow \mathcal{E}(X,V) \qquad \text{(GRD)}$$

$$I^a(V) \wedge J(V,W) \wedge \mathcal{F}(Y,W) \wedge Q_f(W,W') \Rightarrow \exists V' \cdot Q_e(V,V') \qquad \text{(SIM)}$$

$$I^a(V) \wedge J(V,W) \wedge \mathcal{F}(Y,W) \wedge Q_f(W,W') \Rightarrow J(V',W') \qquad \text{(GLU)}$$

where $Q_f(W,W')$ and $Q_e(V,V')$ denote the before-after predicate of f and e, respectively. ■

The first constraint, GRD, requires that the precondition of f is stronger than the one of e. This is called *guard strengthening* making sure that if the concrete event is enabled and the invariants hold, the abstract guards holds as well. The constraint SIM asks for the existence of a simulation relation between the abstract action $\mathcal{R}(X,V)$ and the concrete action $\mathcal{S}(Y,W)$ when the guard of f is satisfied. The last constraint, GLU requires proving that the gluing invariants are reestablished.

For concrete events f not mapped by ρ, they shall satisfy only the GLU constraint above. Notice that the constraints GRD and SIM are trivially valid in this case because, by convention, the guard and the before-after predicate of SKIP are trivially TRUE.

Definition 3.2 (State Machine Refinement)**.** A machine $M^c(W, I^c(W), E^c, \mathsf{e}_0^c)$ refines a machine $M^a(V, I^a(V), E^a, \mathsf{e}_0^a)$ iff the following conditions are met:

1. $V \subseteq W$,
2. there exists a partial mapping $\rho : E^c \nrightarrow E^a$ that is surjective and injective on $\rho[E^a]$ such that for every $f \in E^c$, if $f \in \rho[E^a]$ then f refines $\rho(f)$ with respect to the gluing invariant $J(V,W)$, otherwise f refines SKIP.

■

Example 2 (Refinement for LM_0)**.** We construct the refinement LM_1 for machine LM_0 which was presented in Example 1. We use a singly linked list to implement the set. The operations over the list are adding at the tail a new element into and removing an element from the list. The specification of state (variables, state invariants) of LM_1 is given below. The components inherited from the abstract machine LM_0 are shown in blue color. A new variable nxt, defined as a bijection in **inv2**, specifies the linkage of the elements of the set els. Two constants f and l are used to represent the head and the tail of the list. They are not in $\mathbb{N}$, they are distinct from elements of els. The invariant **inv3** specifies that the list has no cycles (see Table 3.1 for the meaning of symbols used).

constants	**variables**	**invariants**	**machine state**
$f, l \in \mathbb{Z} \setminus \mathbb{N}$	els	**inv1:** $nxt \in els \cup \{f\} \rightarrowtail\!\!\!\!\twoheadrightarrow els \cup \{l\}$	
axiom c	nxt	**inv2:** $\forall u \cdot u \subseteq nxt^{-1}[u] \Rightarrow u = \varnothing$	$s \triangleq \langle els, nxt \rangle$
$f \neq l$			

The *machine state* s recalls the components of the machine's state used in the transition rules. The events of LM_1 are presented below. They refine the events with the same name in LM_0. The parts in blue belong to the refined event in LM_0.

Transition Rules

Initialization $$\frac{}{s = [els \leftarrow \varnothing, nxt \leftarrow \{f \mapsto l\}]}$$

Add $$\frac{x \in \mathbb{N} \wedge x \notin els \ \wedge x \neq f \wedge x \neq l}{s \xrightarrow{\text{Add(x)}} s[els \leftarrow els \cup \{x\}, nxt(x) \leftarrow l, nxt(nxt^{-1}(l)) \leftarrow x]}$$

Remove $$\frac{x \in els}{s \xrightarrow{\text{Remove(x)}} s[els \leftarrow els \setminus \{x\}, nxt(nxt^{-1}(x)) \leftarrow nxt(x)]}$$

△

3.1.2.1 Stepwise Refinement

To ease the proof of refinement relations, the refinement process can be divided into several refinement steps, as shown in Figure 3.1 (b). Each refinement step reveals a few details. The model obtained in the middle of the process, $\mathsf{M_i}$, is more precise than the top-most model $\mathsf{M_a}$ and less precise than the model $\mathsf{M_c}$.

This way of using refinement is called *stepwise refinement*. It builds a sequence of refinement transformations of the abstract model to a more concrete model. The models obtained during this process form a sequence such that each model is a refinement of the one preceding it in the sequence. The stepwise refinement facilitates not only the proof of refinement but also the formalization process. It allows to gradually build formal models of large systems. We use stepwise refinement for building formal models for SDMA.

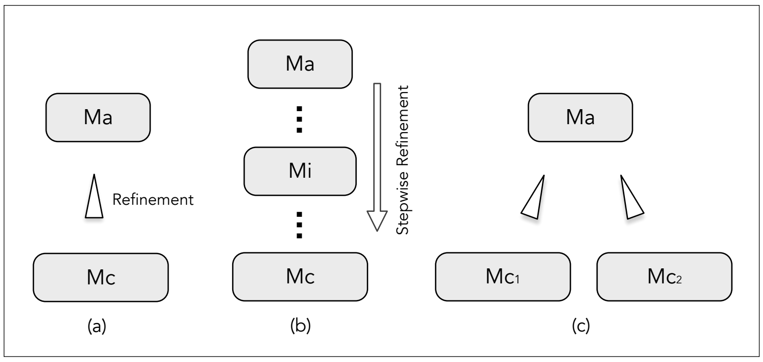


Figure 3.1: (a) refinement in one step, (b) stepwise refinement and (c) two refinement directions

We call *refinement trace* the sequence of stepwise refinements, ordered from the abstract to concrete ones. Figure 3.1 (b) provides an example of refinement trace: it starts from $\mathsf{M_a}$ and ends at $\mathsf{M_i}$. We denote it by $[\mathsf{M_a}; \tau; \mathsf{M_i}]$ where τ is the sequence of models between $\mathsf{M_a}$ and $\mathsf{M_i}$.

Property 3.1.1 (Transitivity of Refinement)**.** *If machine M_a refines machine M_b, and M_b refines M_c, then M_a refines M_c.*

An abstract model may have several refinements as shown in Figure 3.1 (c): both model $\mathsf{M_{c_1}}$ and model $\mathsf{M_{c_2}}$ refine the abstract model $\mathsf{M_a}$.

3.1.2.2 Modular Refinement

By applying different refinement traces to an abstract model, we obtain a set of concrete models. Figure 3.2 (a) illustrates this process with three refinement traces (t_1, t_2, t_3) that lead to three different concrete models $(M_{c_1}, M_{c_2}, M_{c_3})$. If the refinement traces share some transformations, the proof effort may be reduced by moving, if possible, the common refinement transformations at the start of the refinement traces. Indeed, as shown in Figure 3.2 (b), we have to conduct proofs once for the shared prefix of transformations and then diverge the proof for the distinct transformations. In Figure 3.2 (b), the common prefix of refinement traces is t_s. It leads to a model p. Starting from p, the refinement transformations are disjoint and lead to different concrete models. We call *modular stepwise refinement* this way of building concrete models by factorizing common transformations.

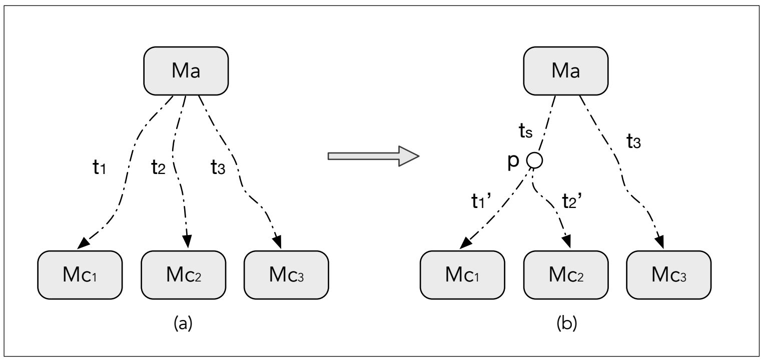


Figure 3.2: (a) refinement traces, (b) shared refinement trace t_s

To conduct modular stepwise refinement, one has to identify the similar transformations between models and then to group them at the start of the refinement. We will use this kind of refinement process for obtaining formal models for SDMA. Therefore, we have identified the similarities between SDMA which concern their policies. We will explain our findings for the refinement of SDMA abstract specification in Section 3.3.

3.2 Abstract Specification for SDMA

We provide in this section a formal model for SDMA that is the most abstract specification of the informal description in Section 2.1.3.

3.2.1 Formalization Hypotheses

We describe here the main hypotheses that underly this very abstract formal model of SDMA. Some of them are required to simplify the model, other will be changed by the refinement process.

The first hypothesis is that the set of chunks managed by the SDMA is fixed. This hypothesis is not satisfied by all SDMA. A notable exception are SDMA where is used an array of fixed size chunks. We will remove this hypothesis in the next chapter by introducing chunk splitting. However, even in this extension, we consider that the set of start addresses of chunks managed by SDMA is included in a constant subset of naturals represented by the interval $[\mathsf{hst}, \mathsf{hli}[$ (interval closed at hst and open at hli, i.e. $\mathsf{hst}..\mathsf{hli} - 1$), where hst and hli (introduced in Section 2.1.1) are specified as natural constants.

The second hypothesis is that the content of the body of chunks is ignored. Therefore, the formal specification of the `realloc` operation of SDMA can not be precisely captured.

The third hypothesis is abstraction of the implementation of the set of chunks. By adopting this hypothesis, we cannot specify the disposition of chunks inside the data segment, which controls important properties of SDMA implementations, in particular, the absence of memory leaks. This hypothesis will be removed by the refinement process.

3.2.2 State

Table 3.2 details the constants and variables which define the state of the formal model. This state specifies the memory region managed by the SDMA and its variables. The formal state is a tuple $s \triangleq \langle H, F, csz, cst \rangle$ where:

- H denotes the set of start addresses of chunks managed,
- csz is a mapping modeling the size of the chunk stored in its header,
- cst is a mapping modeling the status of the chunk (1 for free, 0 for busy) stored in its header, and
- for readability of specifications, we denote by F the set $cst^{-1}(1)$, i.e., the *set of free chunks*.

Meanwhile, there are several necessary constants. The limits of the domain for the addresses of chunks is given by the natural constants hst and hli. The size of the header is modeled by the constant chd; it also gives the offset of the starting address of the chunk body. The start address of every chunk is aligned; the alignment is represented by the natural constant cal. Given a chunk and some natural number representing the requested size in bytes, the fitting function fit returns a number presenting the size of the chunk that SDMA has to allocate to satisfy the request.

Table 3.2: Most abstract specification A: signature

signature	**description**	**type**
$\mathsf{hst}, \mathsf{hli} \in \mathbb{N}$	limits of the memory region	constant
nil	null memory address	constant
$\mathsf{chd}, \mathsf{cal} \in \mathbb{N}$	size of header resp. alignment	constant
$\mathsf{fit} : H \times \mathbb{N} \rightarrow \mathbb{N}$	fitting a chunk with a request	fixed function
$H, F \subset \mathbb{N}$	set of all chunks resp. free chunks	variable
$csz : H \rightarrow \mathbb{N}$	size of a chunk	variable
$cst : H \rightarrow \{0, 1\}$	status of a chunk (1–free, 0–busy)	variable
$s \triangleq \langle H, F, csz, cst \rangle$	memory state	

Table 3.3: Most abstract specification A: invariants

$I_1 : H \subseteq [\mathsf{hst}, \mathsf{hli}[$	chunks domain
$I_2 : \forall c \in H \cdot c \texttt{ MOD } \mathsf{cal} = 0$	chunks are aligned
$I_3 : \mathsf{chd} > 0$	size of chunk header
$I_4 : \forall c \in H \cdot csz(c) \geq \mathsf{chd}$	valid chunk's size
$I_5 : F \subseteq H \ \wedge \ \forall c \in H \cdot cst(c) = 1 \iff c \in F$	consistency
$I_6 : \forall b, c \in H \cdot c \neq b \Rightarrow$	
$\quad [c, c + csz(c)[\cap [b, b + csz(b)[= \varnothing$	do not overlap

3.2.3 Invariants

Table 3.3 contains the set of invariants I_1–I_6 for the abstract state machine. Property I_1 specifies that the elements of H shall be in the limits of the memory region managed. The alignment of the start addresses of chunks on multiples of the constant cal is specified by property I_2. Property I_6 requires that chunks in H occupy pairwise disjoint memory blocks. The relation between F and *cst* is specified by I_5.

3.2.4 Transition Rules

Table 3.4 details the inference rules specifying the behaviors of operations provided by the interface of SDMA.

The `init` method is specified by the Init rule with no precondition, therefore it represents the initial event of the state machine.

For deallocation, the two rules Free^S and Free^F specify the deallocation success and failure respectively. In Free^S, the precondition ensures that the chunk to be deallocated is a valid chunk, i.e., the given parameter is the body address of some chunk. The action is to set the state of the chunk to *free*. When the given parameter is not valid, the state is unchanged. This situation is specified by rule Free^F.

Similarly, there are two cases of allocation, allocation success and failure, represented by Alloc^S and Alloc^F. In rule Alloc^S, its precondition requires that the size of the suitable chunk has to be greater than the fit of the requested size. The action of this event updates the state of the selected chunk. The returned value is the start address of the chunk denoted by p. Allocation fails when all chunks can not satisfy the request. The method returns a special constant nil.

The specification of `realloc` includes three cases: the reallocation may use a neighbor of the allocated chunk to satisfy the request ($\mathsf{Realloc}_1^S$), the reallocation has to free the current chunk and find another one to satisfy the request ($\mathsf{Realloc}_2^S$), and finally the case where the reallocation fails. In this last case, the chunk is not freed, but the result of the operation is nil. As already said, our modeling hypotheses do not allow to specify the move of the content in the second case.

Table 3.4: Most abstract specification A: rules

Inference rules for methods: `init`, `free`, `alloc`, `realloc`	
$\mathsf{Init}\ \dfrac{}{s \xrightarrow{\texttt{init()}} s[F \leftarrow H]}$	initialization
$\mathsf{Free}^{S}\ \dfrac{\exists c \in s.H \setminus s.F \cdot p = c + \mathsf{chd}}{s \xrightarrow{\texttt{free}(p)\text{:true}} s[\mathit{cst}(c) \leftarrow 1]}$	deallocation success
$\mathsf{Free}^{F}\ \dfrac{\forall c \in s.H \setminus s.F \cdot p \neq c + \mathsf{chd}}{s \xrightarrow{\texttt{free}(p)\text{:false}} s}$	deallocation failure
$\mathsf{Alloc}^{S}\ \dfrac{(c \in s.F) \wedge (\mathsf{fit}(c, s) \leq s.\mathit{csz}(c)) \wedge (p = c + \mathsf{chd})}{s \xrightarrow{\texttt{alloc}(s)\text{:}p} s[\mathit{cst}(c) \leftarrow 0]}$	allocation success
$\mathsf{Alloc}^{F}\ \dfrac{\forall c \in s.F \cdot \mathsf{fit}(c, s) > s.\mathit{csz}(c)}{s \xrightarrow{\texttt{alloc}(s)\text{:nil}} s}$	allocation failure
$\mathsf{Realloc}_1^{S}\ \dfrac{\begin{array}{c}\exists c \in s.H \setminus s.F \cdot p = c + \mathsf{chd} \wedge \\ c + s.\mathit{csz}(c) \in s.F \wedge \mathsf{fit}(c, n) > s.\mathit{csz}(c)\end{array}}{s \xrightarrow{\texttt{realloc}(p,n)\text{:}p} s \left[\begin{array}{l}\mathit{csz}(c) \leftarrow n, \\ F \leftarrow ((F \setminus \{c + \mathit{csz}(c)\}) \\ \quad \cup \{c + n\})\end{array}\right]}$	expand chunk
$\mathsf{Realloc}_2^{S}\ \dfrac{\begin{array}{c}\exists c \in s.H \setminus s.F \cdot p = c + \mathsf{chd} \wedge \\ \mathsf{fit}(c, n) > s.\mathit{csz}(c) \wedge \\ s \xrightarrow{\texttt{alloc}(n)\text{:}q} s_1 \xrightarrow{\texttt{free}(p)\text{:}\mathit{true}} s_2\end{array}}{s \xrightarrow{\texttt{realloc}(p,n)\text{:}q} s_2}$	realloc a new chunk
$\mathsf{Realloc}^{F}\ \dfrac{\begin{array}{c}(p = c + \mathsf{chd}) \wedge (c \in s.H \setminus s.F) \wedge \\ (\mathsf{fit}(c, n) \leq s.\mathit{csz}(c))\end{array}}{s \xrightarrow{\texttt{realloc}(p,n)\text{:}p} s}$	realloc failure

3.3 Refinement Strategy for SDMA

We start from the most abstract specification of SDMA and we refine it incrementally by gradually introducing the specific design policies discussed in the previous chapter (e.g., lazy and eager coalescing of chunks, policies for choosing the fitting chunk, boundary tag technique). The order in which the design policies are considered is a contribution of this thesis and it allows to obtain a modular refinement process and therefore facilitates the proof of the refinement transformations.

3.3.1 Refinement Steps

Given a formal model of SDMA, M, and a set T of applicable policies for refinement, we choose a technique $t \in T$ such that t is required to be fixed for the refinement of M by policies in $T \setminus \{t\}$. We obtained this ordering on design policies by the analysis of the case studies collected in Table 2.1.

Refinement for heap list: First, we observe that SDMA employ a *heap list* data structure to implement the set of chunks. It fixes the policies employed by SDMA, like chunk coalescing, splitting or fitting. To apply this refinement, the events of the abstract specification are refined by adding details on how they are implemented if H is a heap list. For example, the refined initialization describes how the heap list is initialized. The refined allocation and deallocation explain how splitting and coalescing operations affect the heap list. Chunk splitting and coalescing require two elementary operations on heap lists: *inserting* a new chunk into the list of chunks and *removing* a chunk when merging it with some neighbor. These elementary operations on heap lists are specified independently by an event and called by the events modeling the SDMA operations. The way in which these elementary operations on heap list are employed leads to different splitting and coalescing policies and consequently different directions for the refinement.

Refinement for free list: The next set of policies that influence the refinement is the implementation of the set F by a free list. Even if the free list employed is singly linked, the refinements diverge on the type of the free list considered: acyclic or cyclic, ordered or unordered by addresses, etc. Like for the heap list, the operations of SDMA employ basic operations on free list: inserting and removing a free chunk, search for a free chunk. The refinement of the free list implementation is done in parallel with the refinement of the basic operations on the list. This difference on basic operations on the free list is used to produce the variety of policies and techniques presented in the previous chapter.

Refinement for fit policy: The fit policy does a search in the data structure used to manage the free chunks and therefore depends on its implementation. Consequently, it is introduced after the refinement of the free list. The refinement directions consist in providing different specifications for the search operation on the free list that are useful to obtain the three fit policies: best fit, first fit and next fit.

Refinement for doubly link free list: The refinement for doubly linked free list is introduced after the refinement for singly linked free list. It consists in adding a new relation specifying the backward link such that it is the inverse of forward link. Doubly linkage of free list is useful to accelerate free list operations and coalescing. This is the reason why we apply this refinement in the end.

3.3.2 Refinement Principles

From the above observation and our experience with refinement proofs, we extract the following principles of our formalization:

R1: Refinements of the heap list precede the ones of the free list.

R2: Refinements concern basic operations on heap (resp. free) list.

R3: Refinements of basic operations on heap (resp. free) list shall be independent and compose for the same implementation choice.

R4: Refinements of the fit policy shall be done after the refinement for the free list.

We applied the above refinement principles to obtain a hierarchy of models, part of it presented in Figure 3.3. This hierarchy mainly includes five layers, called *abstract*, *heap list*, *free list (SLL)*, *fit*, and *free list (DLL)* ordered according to the refinement steps.

The first layer is the abstract specification. The second layer contains five models. The dashed lines between the abstract specification and models in the *heap list* level denote that the refinement relation can not be established between these models. We explain this result in detail in Chapter 4. Intuitively, the main reason is the fact that the set of chunks H is variable in the models using the

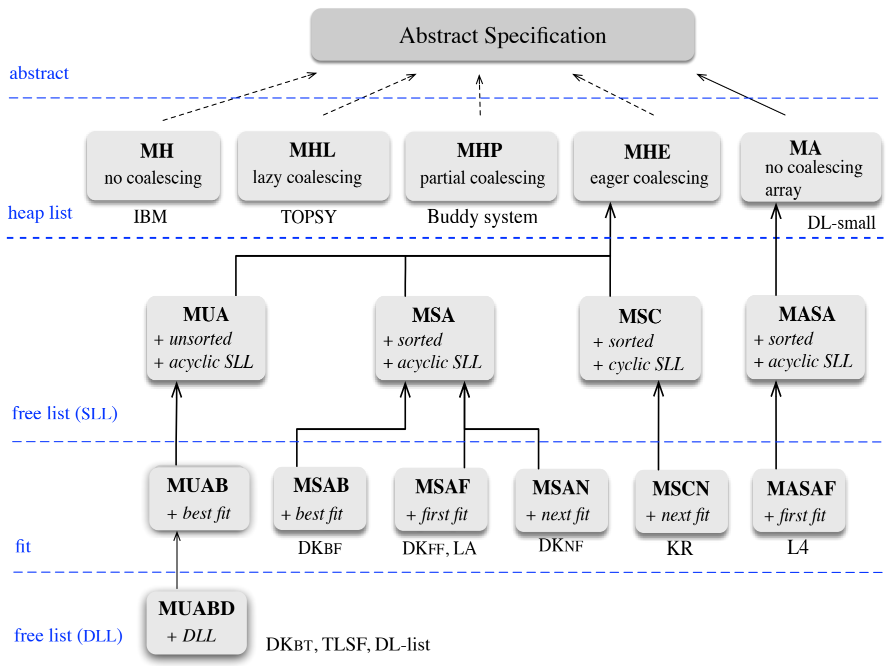


Figure 3.3: A partial view of the hierarchy of models and the case studies it covers

heap list (except the one based on arrays), which is not the case in the abstract specification. The models in the second layer modeling a particular organization of the heap list. For example, the models **MH** and **MA** do not coalesce free chunks but they use variable size and fixed size chunks, respectively. The model **MHL** uses a heap list with lazy coalescing policy, while **MHP** and **MHE** use partial coalescing and eager coalescing, respectively.

The models in the third layer refine the models in the second layer by introducing the design tactics for the singly linked free list. These design tactics are explicit in the box of each model. The refinement traces diverge considering the different types of free list. The black arrows between boxes are the refinement relations proved using Rodin. For example, the model **MUA** is a refinement of the model **MHE**.

The fourth layer contains models refining fit policies. For different fit policies, there are several refinement directions starting from the model in free list layer, e.g., the model **MSA** has three different refinements. Therefore, the model **MSAF** specifies a SDMA with an early coalescing heap list (it transitively refines **MHE**), a free list sorted by address and acyclic, and a first fit policy.

The final layer contains models specifying a doubly linked free list. From our experience, the proof effort is the same for both orders of applying the refinement steps on fit policy or doubly link. Therefore, the fit policy can also be introduced after the refinement of doubly link free list. Moreover, we can join these two refinement steps.

Figure 3.3 includes only the part of the hierarchy of models that covers our case studies, listed as labels of models. However, the refinement relations we define in the next chapter allow to obtain more models. Indeed, the hierarchy in Figure 3.3 can be extended to specify more cases. The refinements for free list can start from any model in the heap list layer. In this thesis, we mainly describe the branch refining **MHE** because it is the most complex model.

Some readers concerned by implementation details may get worried about some design choices that are not covered by the above presentation, e.g., alignment of start addresses for chunks, encoding of the free status of chunks in the header, the unit on which the size of the chunk is measured, the fitting size. These implementation choices may be kept abstract like in the abstract specification presented in Section 3.2 as we will show in the next chapter.

CHAPTER 4

Refinement-based Formalization

In the previous chapters, we introduce a set of SDMA and their diverse policies. To generally formalize all of them, we define the refinement strategy and propose some refinement principles for formalization. Meanwhile, we also give the most abstract specification of SDMA. In this chapter, we describe the refinement-based formalization process for SDMA. We explain how to obtain refinements from the abstract specification. As shown in Figure 3.3 in Chapter 3, there are four layers of refinement derived from the abstract specification. We detail the construction of each model in each layer.

This chapter is structured as follows. § 4.1 describes the refinement for heap list considering different coalescing policies. § 4.2 presents the refinement steps for free list starting from the most complex model in the heap list layer. The models in the free list layer cover diverse list organizations. § 4.3 describes the possible application of the models obtained by the formalization. § 4.4 describes the related work and gives a conclusion of this part on specification of SDMA.

4.1 Heap List Modelling

This section presents the models obtained by refining the representation of the set of chunks in a heap list. These models are in the second layer as shown in Figure 3.3. We obtain these models by composing refinements of basic operations on heap lists, such as inserting, deleting, etc. An important difference between refinements are the ways to deal with memory fragment.

4.1.1 State and Invariants

The abstract state of a heap list SDMA is defined in Table 4.1. It includes, in addition to elements of the abstract state defined in Table 3.2.2, the successor and predecessor relations between chunks, *cnx* resp. cpr. The mapping cpr is specified only for doubly linked heap lists. In addition to invariants in Table 3.3, we introduce the invariants I_7 and I_7' in Table 4.1 to characterise the two new relations. They assert that *cnx* is a bijection, and if cpr is defined, it is the inverse of *cnx*. A consequence of invariants I_7–I_8 is the following expected property:

Property 4.1.1. *The heap list is acyclic, starts in* hst*, and ends in* hli*.*

The invariant I_9 asserts that a chunk occupies exactly the space between its start and the start of the next chunk, which leads to the following property:

Property 4.1.2. *A heap list satisfying I_1–I_9 has no external memory leaks.*

Each model in the heap list layer satisfies the invariants I_1–I_9 and a subset of the last four invariants in Table 4.1. For example, the model **MA** includes only the invariant I_{ar} that fixes the size of chunks to some constant kb to specify the class of SDMA that manages an array of chunks. The invariant I_{ec} is added only for the model **MHE** to characterise the state of SDMA with early coalescing policy. It asserts that any two chunks, successive in the heap list, cannot be both free. The shorthand $\texttt{NAND}(p_1, p_2)$ stands for the predicate $(\neg(p_1 \wedge p_2))$.

The invariant I_{pc} is introduced for the model **MHP** to specifies heap list with partial coalescing. The invariant states that two adjacent free chunks can not be both free if they belong to the same buddy, which is expressed by the predicate $buddy(c_1, c_2)$, defined by $cnx(c_1) = c_2 \wedge csz(c_1) = csz(c_2) \wedge c_1 \texttt{ MOD } (2 \times csz(c_1)) = 0$. Notice that both I_{ec} and I_{pc} may be temporary broken during the execution of methods `free` and `realloc`. In addition to I_{pc}, the model **MHP** includes the invariant I_{by} that constrains the size of each chunk to be a power of two and its address to be aligned to this size.

Table 4.1: Refinement of A for heap list models

State refinement		
signature	**description**	**type**
$\mathsf{hst}, \mathsf{hli} \in \mathbb{N}$	limits of the memory region	constant
nil	null memory address	constant
$\mathsf{chd}, \mathsf{cal} \in \mathbb{N}$	size of header resp. alignment	constant
$\mathsf{fit} : H \times \mathbb{N} \to \mathbb{N}$	fitting a chunk with a request	function
$H, F \subset \mathbb{N}$	set of all chunks resp. free chunks	variable
$csz : H \to \mathbb{N}$	size of a chunk	variable
$cst : H \to \{0, 1\}$	status of a chunk (1–free, 0–busy)	variable
$cnx : H \to (H \setminus \{\mathsf{hst}\}) \cup \{\mathsf{hli}\}$	next chunk	variable
$cpr : (H \setminus \{\mathsf{hst}\}) \cup \{\mathsf{hli}\} \to H$	previous chunk	variable
kb	fixed size of chunk	constant
$s \triangleq \langle H, F, csz, cst, cnx, cpr \rangle$	state of SDMA with heap list	

Invariants inherited from the abstract specification	
$I_1 : H \subseteq [\mathsf{hst}, \mathsf{hli}[$	chunks domain
$I_2 : \forall c \in H \cdot c \texttt{ MOD } \mathsf{cal} = 0$	chunks are aligned
$I_3 : \mathsf{chd} > 0$	size of chunk header
$I_4 : \forall c \in H \cdot csz(c) \geq \mathsf{chd}$	valid chunk's size
$I_5 : F \subseteq H \ \wedge \ \forall c \in F \cdot cst(c) = 1 \iff c \in F$	consistency
$I_6 : \forall b, c \in H \cdot c \neq b \Rightarrow [c, c + csz(c)[\cap [b, b + csz(b)[= \varnothing$	chunks not overlapped
Additional invariants	
$I_7 : cnx$ is a bijection	linked heap list
$I'_7 : cpr = cnx^{-1}$	doubly linked list
$I_8 : \mathsf{hst} \in H$	start in hst
$I_9 : \forall c \in H \cdot cnx(c) = c + csz(c)$	no memory leaks
$I_{ar} : \forall c \in H \cdot csz(c) = kb$	array heap list
$I_{ec} : \forall c_1, c_2 \in H \cdot \big(cnx(c_1) = c_2\big) \Rightarrow \texttt{NAND}(cst(c_1), cst(c_2))$	early coalescing
$I_{pc} : \forall c_1, c_2 \in H \cdot buddy(c_1, c_2) \Rightarrow \texttt{NAND}(cst(c_1), cst(c_2))$	partial coalescing
$I_{by} : \forall c \in H \cdot \exists k \in \mathbb{N} \cdot csz(c) = 2^k \wedge (c \texttt{ MOD } 2^k = 0)$	buddy size constraint

4.1.2 Basic Heap List Operations

The inference rules in Table 3.4 abstract away the implementation details of SDMA methods and do not capture the splitting of a fitting chunk or the merging of adjacent free chunks during allocation and deallocation. To refine these rules into ones that specify precisely the behaviour of methods of heap list SDMA, we use the following basic operations on the heap lists:

hremove	removes a free chunk (from F)
hinsert	inserts a free chunk (into F)
hsearch	searches a fitting free chunk
hsplit	splits a free chunk
hmerge_L	merges a free chunk with its free left neighbour
hmerge_R	merges a free chunk with its free right neighbour
$\mathsf{hmerge}_\forall$	merges all sequences of free chunks
hmerge_P	merges free chunks in same buddy

We explain the specifications of these basic operations for singly linked heap lists in this section. Notice that only remove, insert, and search operations are relevant for array based SDMA. *To simplify the presentation of rules, we adopt the convention that the elements of state s (the source state of the defined rule) may appear without the dotted notation in the rule.* For some operations, e.g., hsplit or hsearch, several refinements are provided. The main methods of the SDMA are specified in Table 4.6 - 4.8 using these basic operations.

4.1.2.1 Removing, inserting, searching

Table 4.2 specifies the operations hremove, hinsert, and hsearch. The rules hremove and hinsert describe that the status (free or busy) of the chunk c given as parameter is updated accordingly for removing and insertion in the free set. Operation hsearch(n) describes the search behaviour. In this level of refinement, the fit policies used in search are abstracted away. The rules $\mathsf{hsearch}^S$ and $\mathsf{hsearch}^F$ specify the search success and failure respectively.

4.1.2.2 Splitting

Table 4.3 specifies the refinements for the operation hsplit. This operation has as parameters a free chunk c and a natural n representing the size of the new chunk to be created inside c; this new chunk is set as busy and returned as a result

Table 4.2: Refinements of heap list operations for remove, insert, and search

$\mathsf{hremove}(c)$	$\mathsf{hremove}\ \dfrac{c \in s.F}{s \xrightarrow{\mathsf{hremove}(c)} s[F \leftarrow F \setminus \{c\}, cst(c) \leftarrow 0]}$
$\mathsf{hinsert}(c)$	$\mathsf{hinsert}\ \dfrac{c \in s.H \setminus s.F}{s \xrightarrow{\mathsf{hinsert}(c)} s[F \leftarrow F \cup \{c\}, cst(c) \leftarrow 1]}$
$\mathsf{hsearch}(n) : c$	$\mathsf{hsearch}^S\ \dfrac{\exists b \in s.F \cdot s.csz(b) \geq \mathsf{fit}(b, n)}{s \xrightarrow{\mathsf{hsearch}(n):b} s}$ $\mathsf{hsearch}^F\ \dfrac{\forall b \in s.F \cdot s.csz(b) < \mathsf{fit}(b, n)}{s \xrightarrow{\mathsf{hsearch}(n):\mathsf{nil}} s}$

of hsplit. The three refinements of hsplit, represented by behaviours hsplit_M, hsplit_B and hsplit_E, choose different ways to split the chunk: in two equal size parts, with n bytes at the beginning, or at the end respectively. The splitting rules call removing and inserting operations. After splitting, the memory selected for allocation is marked as busy. The remainder is inserted into the set of free chunks. The behaviour hsplit_P refines hsplit for buddy SDMA: it applies repeatedly hsplit_M (rule hsplit_P^S) until the requested size n fits in the chunk and it is bigger than half of the candidate chunk (rule hsplit_P^F).

4.1.2.3 Merging

Table 4.5 provides refinements of the operation hmerge that is called in `free` or `realloc` to join neighbouring free chunks in one. The invariants for early or partial coalescing (I_{ec} resp. I_{pc}) are broken temporarily in the state before calling hmerge.

Two basic operations are specified in Table 4.4. The behaviour hmerge_R joins the chunk parameter b with its right neighbour c if c is free; otherwise, the operation does nothing. For sake of symmetry with the second behaviour, hmerge_R returns its parameter. Similarly, the refinement hmerge_L merges a

Table 4.3: Refinements of heap list operation for chunk splitting

$\mathsf{hsplit}(c, n) : b$	$\mathsf{hsplit}_M\ \dfrac{(c \in s.F) \wedge (0 < n < s.csz(c)/2) \wedge (c' = c + s.csz(c)/2) \wedge (s \xrightarrow{\mathsf{hremove}(c)} s_1 \xrightarrow{\mathsf{hinsert}(c')} s_2)}{s \xrightarrow{\mathsf{hsplit}_M(c,n):c} s_2 \begin{bmatrix} H \leftarrow H \cup \{c'\}, \\ csz[c, c' \leftarrow csz(c)/2], \\ cnx[c \leftarrow c', c' \leftarrow cnx(c)] \end{bmatrix}}$
	$\mathsf{hsplit}_B\ \dfrac{(c \in s.F) \wedge (0 < n < s.csz(c)) \wedge (c' = c + n) \wedge (s \xrightarrow{\mathsf{hremove}(c)} s_1 \xrightarrow{\mathsf{hinsert}(c')} s_2)}{s \xrightarrow{\mathsf{hsplit}(c,n):c} s_2 \begin{bmatrix} H \leftarrow H \cup \{c'\}, \\ csz[c \leftarrow n, c' \leftarrow csz(c) - n], \\ cnx[c \leftarrow c', c' \leftarrow cnx(c)] \end{bmatrix}}$
	$\mathsf{hsplit}_E\ \dfrac{(c \in s.F) \wedge (0 < n < s.csz(c)) \wedge (c' = c + s.csz(c) - n)}{s \xrightarrow{\mathsf{hsplit}(c,s):c'} s \begin{bmatrix} H \leftarrow H \cup \{c'\}, cst(c') \leftarrow 0, \\ csz[c \leftarrow csz(c) - n, c' \leftarrow n], \\ cnx[c \leftarrow c', c' \leftarrow cnx(c)] \end{bmatrix}}$
$\mathsf{hsplit}_P(c, n) : b$	$\mathsf{hsplit}^S_P\ \dfrac{(b \in s.F) \wedge (s \xrightarrow{\mathsf{hsplit}_M(c,n):b} s_1 \xrightarrow{\mathsf{hsplit}_P(b,n):b'} s_2)}{s \xrightarrow{\mathsf{hsplit}_P(c,n):b'} s_2}$
	$\mathsf{hsplit}^F_P\ \dfrac{(c \in s.F) \wedge s.csz(c)/2 < n < s.csz(c)}{s \xrightarrow{\mathsf{hsplit}_P(c,n):c} s}$

free chunk with its left free neighbour.

In Table 4.5, the behaviour hmerge_N merges a free chunk with its free neighbors. It is called by deallocation method hfree. The behaviour $\mathsf{hmerge}_\forall$ merges any two successive free chunks in the entire memory region. The rule $\mathsf{hmerge}^F_\forall$ states that I_{ec} is satisfied and therefore the merging operation terminates. The refinement of hmerge for SDMA with partial coalescing is specified by hmerge_{PN} that joins only chunks in the same buddy. Operation hmerge_{PN} is called repeatedly by hmerge_P until the invariant I_{pc} is satisfied.

Table 4.4: Basic chunk merging operations

$\mathsf{hmerge}_R(b) : x$	$\mathsf{hmerge}_R^S \dfrac{b \in s.F \land c \in s.F \land c = s.cnx(b) \land s \xrightarrow{\mathsf{hremove}(c)} s_1}{s \xrightarrow{\mathsf{hmerge}_R(b):b} s_1 \left[\begin{array}{l} H \leftarrow H \setminus \{c\}, cnx[b \leftarrow cnx(c)], \\ csz[b \leftarrow csz(b) + csz(c)] \end{array}\right]}$ $\mathsf{hmerge}_R^F \dfrac{b \in s.F \land s.cnx(b) \notin s.F}{s \xrightarrow{\mathsf{hmerge}_R(b):b} s}$
$\mathsf{hmerge}_L(b) : x$	$\mathsf{hmerge}_L^S \dfrac{b \in s.F \land c \in s.F \land s.cnx(c) = b \land s \xrightarrow{\mathsf{hremove}(b)} s_1}{s \xrightarrow{\mathsf{hmerge}_L(b):c} s_1 \left[\begin{array}{l} H \leftarrow H \setminus \{b\}, cnx[c \leftarrow cnx(b)], \\ csz[c \leftarrow csz(b) + csz(c)] \end{array}\right]}$ $\mathsf{hmerge}_L^F \dfrac{b \in s.F \land s.cnx^{-1}(b) \notin s.F}{s \xrightarrow{\mathsf{hmerge}_L(b):b} s}$

4.1.3 Models for Heap List SDMA

The specifications of SDMA methods make use of basic operations presented above are shown in Table 4.6 - 4.8.

4.1.3.1 Method `init`

In Table 4.6, we provide two refinements for the method `init`: the rule $\mathsf{hinit}_{\mathrm{hl}}$ initialises the abstract state for SDMA with variable size chunks, while the rule $\mathsf{hinit}_{\mathrm{ar}}$ does initialisation for fixed size chunks, i.e., array based SDMA. The size of each chunk in the array is denoted by *kb*.

4.1.3.2 Method `alloc`

As shown in Table 4.7, the `alloc` method is refined to obtain three distinct behaviours: allocation in fixed chunk sized SDMA (rule $\mathsf{halloc}_{\mathrm{ar}}$), allocation without or without coalescing for variable chunk sizes (rule $\mathsf{halloc}_{\mathrm{eager/no}}$), allocation with lazy coalescing (rule $\mathsf{halloc}_{\mathrm{lazy}}$). The last two behaviours call the internal operation $\mathsf{halloc}_{\mathrm{i}}$, which does the main part of the work: it searches the free chunk fitting the request using $\mathsf{hsearch}$ and returns this chunk after changing its status. The rule $\mathsf{halloc}^S_{\mathrm{fit}}$ specifies the case where the fitting chunk does not need splitting; the rule $\mathsf{halloc}^S_{\mathrm{split}}$ specifies the splitting operations.

Table 4.5: Refinements of heap list operation for chunk merging

$\mathsf{hmerge}_N(b) : x$	hmerge_N^S	$\dfrac{b \in s.F \wedge s \xrightarrow{\mathsf{hmerge}_R(b):b} s_1 \xrightarrow{\mathsf{hmerge}_L(b):c} s_2}{s \xrightarrow{\mathsf{hmerge}_N(b):c} s_2}$
	hmerge_N^F	$\dfrac{b \in s.F \wedge s.cnx(b) \notin s.F \wedge s.cnx^{-1}(b) \notin s.F}{s \xrightarrow{\mathsf{hmerge}_N(b):b} s}$
$\mathsf{hmerge}_\forall$	$\mathsf{hmerge}_\forall^S$	$\dfrac{b \in s.F \wedge s \xrightarrow{\mathsf{hmerge}_R(b):b} s_1 \xrightarrow{\mathsf{hmerge}_L(b):c} s_2 \xrightarrow{\mathsf{hmerge}_\forall} s_3}{s \xrightarrow{\mathsf{hmerge}_\forall} s_3}$
	$\mathsf{hmerge}_\forall^F$	$\dfrac{I_{ec}}{s \xrightarrow{\mathsf{hmerge}_\forall} s}$
$\mathsf{hmerge}_{PN}(b) : x$	hmerge_{PN}^S	$\dfrac{b \in s.F \wedge c \in s.F \wedge buddy(b,c) \wedge s \xrightarrow{\mathsf{hremove}(c)} s_1}{s \xrightarrow{\mathsf{hmerge}_{PN}(b):b} s_1 \begin{bmatrix} H \leftarrow H \setminus \{c\}, \\ csz[b \leftarrow csz(b) + csz(c)], \\ cnx[b \leftarrow cnx(c)] \end{bmatrix}}$
	$\mathsf{hmerge}_{PN}^{S'}$	$\dfrac{b \in s.F \wedge c \in s.F \wedge buddy(c,b) \wedge s \xrightarrow{\mathsf{hremove}(b)} s_1}{s \xrightarrow{\mathsf{hmerge}_{PN}(b):c} s_1 \begin{bmatrix} H \leftarrow H \setminus \{b\}, \\ csz[c \leftarrow csz(b) + csz(c)], \\ cnx[c \leftarrow cnx(b)] \end{bmatrix}}$
$\mathsf{hmerge}_P(b)$	hmerge_P^S	$\dfrac{b \in s.F \wedge s \xrightarrow{\mathsf{hmerge}_{PN}(b):c} s_1 \xrightarrow{\mathsf{hmerge}_P(c)} s_2}{s \xrightarrow{\mathsf{hmerge}_P(b)} s_2}$
	hmerge_P^F	$\dfrac{I_{pc}}{s \xrightarrow{\mathsf{hmerge}_P(b)} s}$

Table 4.6: Refinements of method `init` for heap list

$\mathsf{hinit}()$	$\mathsf{hinit}_{\mathsf{hl}}$	$\dfrac{}{s \xrightarrow{\mathsf{hinit}()} \left\langle \begin{matrix} H \leftarrow \{\mathsf{hst}\}, F \leftarrow \{\mathsf{hst}\}, \\ cnx(\mathsf{hst}) \leftarrow \mathsf{hli}, cst(\mathsf{hst}) \leftarrow 1, \\ csz(\mathsf{hst}) \leftarrow \mathsf{hli} - \mathsf{hst} \end{matrix} \right\rangle}$
	$\mathsf{hinit}_{\mathsf{ar}}$	$\dfrac{}{s \xrightarrow{\mathsf{hinit}()} \left\langle \begin{matrix} H \leftarrow \{\mathsf{hst} + i \times kb \mid 0 \leq \mathsf{hli}/i\}, \\ F \leftarrow \{\mathsf{hst} + i \times kb \mid 0 \leq \mathsf{hli}/i\}, \\ cnx(c) \leftarrow c + kb, csz(c) \leftarrow kb \end{matrix} \right\rangle}$

Table 4.7: Refinements of method `alloc` for heap list

$\mathsf{halloc_i}(n) : p$	$\mathsf{halloc}^S_{\mathrm{fit}}\ \dfrac{(c \neq \mathsf{nil}) \wedge (p = c + \mathsf{chd}) \wedge (\mathsf{fit}(c,n) = s.csz(c)) \wedge (s \xrightarrow{\mathsf{hsearch}(n):c} s \xrightarrow{\mathsf{hremove}(c)} s_1)}{s \xrightarrow{\mathsf{halloc_i}(n):p} s_1}$
	$\mathsf{halloc}^S_{\mathrm{split}}\ \dfrac{(c \neq \mathsf{nil}) \wedge (p = b + \mathsf{chd}) \wedge (\mathsf{fit}(c,n) < s.csz(c)) \wedge (s \xrightarrow{\mathsf{hsearch}(n):c} s \xrightarrow{\mathsf{hsplit}(c,\mathsf{fit}(c,n)):b} s_1)}{s \xrightarrow{\mathsf{halloc_i}(n):p} s_1}$
	$\mathsf{halloc}^F_{\mathrm{i}}\ \dfrac{s \xrightarrow{\mathsf{hsearch}(n):\mathsf{nil}} s}{s \xrightarrow{\mathsf{halloc_i}(n):\mathsf{nil}} s}$
$\mathsf{halloc}(n) : p$	$\mathsf{halloc}^S_{\mathrm{ar}}\ \dfrac{s \xrightarrow{\mathsf{hsearch}(n):p} s \xrightarrow{\mathsf{hremove}(p)} s_1 \wedge p \neq \mathsf{nil}}{s \xrightarrow{\mathsf{halloc}(n):p} s_1}$
	$\mathsf{halloc}^S_{\mathrm{eager/no}}\ \dfrac{s \xrightarrow{\mathsf{halloc_i}(n):p} s_1}{s \xrightarrow{\mathsf{halloc}(n):p} s_1}$
	$\mathsf{halloc}^S_{\mathrm{lazy}}\ \dfrac{s \xrightarrow{\mathsf{halloc_i}(n):\mathsf{nil}} s \xrightarrow{\mathsf{hmerge}_{\forall}} s_1 \xrightarrow{\mathsf{halloc_i}(n):p} s_2}{s \xrightarrow{\mathsf{halloc}(n):p} s_2}$
	$\mathsf{halloc}^F_{*}\ \dfrac{s \xrightarrow{\mathsf{hsearch}(n):\mathsf{nil}} s}{s \xrightarrow{\mathsf{halloc}(n):\mathsf{nil}} s}$

Notice that $\mathsf{halloc}^S_{\mathrm{fit}}$ allows to define behaviours for allocation without splitting: if $\mathsf{fit}(c,n)$ returns $csz(c)$ for $csz(c) \geq n$.

4.1.3.3 Method `free`

The specification of the method `free` is refined similarly to obtain its behaviours for eager, lazy and no coalescing policies, represented by $\mathsf{hfree}_{\mathrm{eager}}$, $\mathsf{hfree}_{\mathrm{lazy}}$ and $\mathsf{hfree}_{\mathrm{no}}$. In rule $\mathsf{hfree}_{\mathrm{eager}}$, after freeing the chunk, the invariant I_{ec} is established by calling the merging with the free neighbours (if any). In rule $\mathsf{hfree}_{\mathrm{lazy}}$, the chunk to be released is directly put back into the set of free chunks, no coalescing occurrence. The rule $\mathsf{hfree}_{\mathrm{no}}$ for deallocation without coalescing is similar to the rule describing lazy coalescing.

Table 4.8: Refinements of the `free` method on heap list

$\mathsf{hfree}(p) : t$	$\mathsf{hfree}^{S}_{\text{eager}}\ \dfrac{(p = b + \mathsf{chd}) \wedge (b \in s.H \setminus s.F) \wedge (s \xrightarrow{\mathsf{hinsert}(b):b} s_1 \xrightarrow{\mathsf{hmerge}_N(b):c} s_2)}{s \xrightarrow{\mathsf{hfree}(p):\text{true}} s_2}$
	$\mathsf{hfree}^{S}_{\text{lazy/no}}\ \dfrac{(p = b + \mathsf{chd}) \wedge (b \in s.H \setminus s.F) \wedge (s \xrightarrow{\mathsf{hinsert}(b)} s_1)}{s \xrightarrow{\mathsf{hfree}(p):\text{true}} s_1}$
	$\mathsf{hfree}^{F}_{*}\ \dfrac{\forall b \in s.H \setminus s.F \cdot p \neq b + \mathsf{chd}}{s \xrightarrow{\mathsf{hfree}(p):\text{false}} s}$

4.1.3.4 Method `realloc`

Table 4.9 presents the specification of the method `realloc` based on the refinement for heap list. $\mathsf{hrealloc}^S_1$ describes that if the request is same as the size of the current chunk, `realloc` returns the current chunk. If the additional size requested is available after the old chunk, then the old chunk will be merged with its right neighbor in rule $\mathsf{hrealloc}^S_2$. The refinement $\mathsf{hrealloc}^S_3$ specifies that if there is not enough free space after the old chunk, the old chunk will be released, and the `alloc` is called again with the requested size. Notice that, this case is abstract and the new chunk does not copy the data stored in the old chunk. $\mathsf{hrealloc}^S_4$ describes the case of contracting the old chunk, it calls the hsplit operator. If the given parameter is a null address, `realloc` will call halloc. This is formalized by $\mathsf{hrealloc}^S_5$. The final rule $\mathsf{hrealloc}^F$ specifies that if there is not enough memory, `realloc` will return a null address.

4.1.3.5 Statistics

Table 4.10 sums up the main characteristics of each model resulting from the refinement of heap list operations: the specific invariants, the heap list operations used, and the size of the model. We coded these specifications in Event-B [Abr10]. The correctness of the refinement, stated by the following

Table 4.9: Refinements of `realloc` method on heap list

$\mathsf{hrealloc}(p,n) : q$	$\mathsf{hrealloc}_1^S \; \dfrac{b \in s.H \setminus s.F \wedge n > 0 \wedge p = b + \mathsf{chd} \wedge \mathsf{fit}(b,n) = csz(b)}{s \xrightarrow{\mathsf{hrealloc}(p,n):p} s}$
	$\mathsf{hrealloc}_2^S \; \dfrac{\begin{array}{c} b \in s.H \setminus s.F \wedge n > 0 \wedge p = b + \mathsf{chd} \wedge \\ s.cnx(b) = c \wedge c \in s.F \wedge \mathsf{fit}(b,n) > s.csz(b) \wedge \\ \mathsf{fit}(c, n - s.csz(b)) \leq s.csz(c) \wedge \\ s \xrightarrow{\mathsf{hinsert}(b)} s \xrightarrow{\mathsf{hmerge}_R(b):b} s_1 \xrightarrow{\mathsf{hsplit}_B(b,n):b'} s_2 \xrightarrow{\mathsf{hremove}(b)} s_3 \end{array}}{s \xrightarrow{\mathsf{hrealloc}(p,n):p} s_3}$
	$\mathsf{hrealloc}_3^S \; \dfrac{\begin{array}{c} b \in s.H \setminus s.F \wedge n > 0 \wedge p = b + \mathsf{chd} \wedge \\ s.cnx(b) = c \wedge c \in s.F \wedge \\ \mathsf{fit}(c, n - s.csz(b)) > s.csz(c) \wedge \mathsf{fit}(b,n) > s.csz(b) \wedge \\ s \xrightarrow{\mathsf{hfree}(p):true} s_1 \xrightarrow{\mathsf{halloc}(n):q} s_2 \end{array}}{s \xrightarrow{\mathsf{hrealloc}(p,n):q} s_2}$
	$\mathsf{hrealloc}_4^S \; \dfrac{\begin{array}{c} n > 0 \wedge p = b + \mathsf{chd} \wedge b \in s.H \setminus s.F \wedge \\ \mathsf{fit}(b,n) < s.csz(b) \wedge s \xrightarrow{\mathsf{hinsert}(b)} s_1 \xrightarrow{\mathsf{hsplit}_B(b,\mathsf{fit}(b,n)):b} s_2 \end{array}}{s \xrightarrow{\mathsf{hrealloc}(p,n):p} s_2}$
	$\mathsf{hrealloc}_5^S \; \dfrac{n > 0 \wedge p = \mathsf{nil} \wedge s \xrightarrow{\mathsf{halloc}(n):q} s_1}{s \xrightarrow{\mathsf{hrealloc}(p,n):q} s_1}$
	$\mathsf{hrealloc}^F \; \dfrac{\forall c \in s.F \cdot (\mathsf{fit}(c,n) \geq s.csz(c))}{s \xrightarrow{\mathsf{hrealloc}(p,n):\mathsf{nil}} s}$

theorem, is translated into a set of proof obligations which are proved with the Rodin tool [ABH+10] and the connected solvers. Table 4.11 provides statistics about the proofs conducted to obtain this theorem.

Theorem 4.1 (Correctness of the models)**.** *For any model of the heap list SDMA (i.e., **MH**, **MHA**, **MHL**, **MHE**, **MA**), the operations specifying a SDMA method preserve the invariants of the model.* ∎

Table 4.10: Overview of heap list models

Models	Specific invariants	Rules		
		`init, alloc, free`	split	merge
MH	none	$\mathsf{hinit}_{\mathrm{hl}}, \mathsf{halloc}_{\mathrm{no}}, \mathsf{hfree}_{\mathrm{no}}$	hsplit_B	–
MHL	none	$\mathsf{hinit}_{\mathrm{hl}}, \mathsf{halloc}_{\mathrm{lazy}}, \mathsf{hfree}_{\mathrm{lazy}}$	hsplit_B	$\mathsf{hmerge}_\forall$
MHE	I_{ec}	$\mathsf{hinit}_{\mathrm{hl}}, \mathsf{halloc}_{\mathrm{eager}}, \mathsf{hfree}_{\mathrm{eager}}$	hsplit_E	hmerge_N
MHP	I_{pc}, I_{by}	$\mathsf{hinit}_{\mathrm{hl}}, \mathsf{halloc}_{\mathrm{eager}}, \mathsf{hfree}_{\mathrm{partial}}$	hsplit_P	hmerge_P
MA	I_{ar}	$\mathsf{hinit}_{\mathrm{ar}}, \mathsf{halloc}_{\mathrm{ar}}, \mathsf{hfree}_{\mathrm{no}}$	–	–

Table 4.11: Statistics on proofs

Models	LOC	Proof obligations	Automatically discharged	Interactive proofs
MH	114	39	27(69%)	12(31%)
MHL	176	8	8(100%)	0(0%)
MHE	183	82	58(70%)	24(30%)
MHP	383	143	140(98%)	3(2%)
MA	168	20	20(100 %)	0 (0%)

4.2 Free List Modelling

This section defines the refinements applied to capture the different design choices related with the use of a list for the set of free chunks. Following the principle R_1 in Chapter 3.3.2, these refinements are applied to models obtained by the refinement of the heap list. Because they are the most interesting, we comment two refinement branches, one is from the **MHE** model with early coalescing and another one is from the **MA** model.

To conform to principle R_4, we define a set of basic operations on free lists. These operations are the counterpart of the ones defined for the heap list in Section 4.1.2: fremove, finsert, fsplit, fmerge, and fsearch. We define four directions of refinement, each dealing with a specific feature of the free list:

1. shape of the free list, with values acyclic (A) and cyclic (C),
2. ordering of chunks by addresses in the free list, with values unordered (U) and sorted (S),
3. cells linking, with values singly (default) and doubly (D), and
4. searching for fit policy, with values first (F), best (B) and next (N) fit.

Each direction corresponds to specific state elements, state invariants, or refinements of basic operations on lists and SDMA methods as summarised up in Table 4.21 (page 62). The notations used in this table are introduced in the following sections.

4.2.1 States and Invariants

Table 4.12 defines the states and the invariants used by the refinement directions on free list. Notice that a free list state extends a state of the heap list model with at least one mapping, the bijection *fnx*, that models the linking in the free list. For doubly linked lists, the linking backward is modelled by the mapping *fpr*. Although we introduce doubly linked list in this section, in fact, the backward link can be introduce later, in the next separated refinement step. The invariants satisfied by the linking mappings are I_{fnx} and I_{fpr}. To capture easily all the shapes of the free lists in our modelling framework, we use two constants, *fbe* and *fen* which delimit the start resp. target (end) of the free list. The variable *rp* is used by the state modelling the next fit policy to mark the last used free chunk. Thus, we could employ the invariant $I_{\ell s}$ for both cyclic and acyclic lists to ensure the following property:

Property 4.2.1. *If a state satisfies I_{fnx}, $I_{\varnothing}$ (and I_C), and $I_{\ell s}$ then the mapping fnx_A (resp. fnx_C) defines an acyclic (resp. cyclic) list starting in $fnx_A(fbe)$ and including all free chunks.*

Table 4.12: States and invariants used by free list refinements; $x \in \{A, C\}$ denotes refinements for the shape of the free list

Refined states	
$fbe, fen \notin H$	constants
$F^+ = F \cup \{fbe, fen\}$	extended free set
$fnx_A : (F \cup \{fbe\}) \to (F \cup \{fen\})$	next free chunk
$fpr_A : (F \cup \{fen\}) \to (F \cup \{fbe\})$	previous free chunk
$fnx_C, fpr_C : F^+ \to F^+$	cyclic next resp. previous
$s \triangleq \langle H, F, csz, cst, cnx, fnx_x \rangle$	state for SLL ($x \in \{A, C\}$)
$s_D \triangleq \langle H, F, csz, cst, cnx, fnx_x, fpr_x \rangle$	state for DLL ($x \in \{A, C\}$)
$s_N \triangleq \langle H, F, csz, cst, cnx, fnx_x, rp \rangle$	state for SLL, next fit policy ($x \in \{A, C\}$)
Additional invariants	
$I_{fnx} : fnx_x$ total bijection	
$I_{fpr} : fpr_x = fnx_x^{-1}$	
$I_{\varnothing} : fnx_x(fbe) = fen \iff F = \varnothing$	empty list
$I_{\ell s} : \forall F' \subseteq F \cdot F' \subseteq fnx_x^{-1}(F') \Rightarrow F' = \varnothing$	no clique
$I_C : fnx_C(fen) = fbe$	*fen* ends the cycle
$I_S : \forall c \in F \cdot fnx_x(fbe) \leq c \leq fnx_x^{-1}(fen)$ $\wedge (fnx_x(c) = fen \vee c < fnx_x(c))$	sorted list
$I_{rp} : F \neq \varnothing \Rightarrow rp \in F$	*rp* is free

Notice that reachability is a second order property. $I_{\ell s}$ is a manner to express this property, inspired by [Abr10]; it states that *fnx* does not define a clique inside F. This is the only place where we need a second order property. For tools with support limited to first-order logic, $I_{\ell s}$ may be replaced with a first order invariant if the free list is address sorted, a property specified by the invariant I_S. Indeed, the following property is a corollary of *fnx* being bijective and strictly increasing:

Property 4.2.2. *If a state satisfies I_{fnx}, $I_{\varnothing}$ (and I_C), and I_S then the mapping fnx defines an acyclic (resp. cyclic) list starting in fnx(fbe) and including all free chunks.*

For models using unsorted free lists, we use the invariant $I_{\ell s}$ due to the fact that Rodin provides means to deal with second order logic properties on sets.

Table 4.13: Refined removing operation on free list

fremove(c)	$\text{fremove}_A\ \dfrac{c \in s.F}{s \xrightarrow{\text{fremove}(c)} s \left[\begin{array}{l} F \leftarrow F \setminus \{c\}, cst(c) \leftarrow 0, \\ fnx(fnx^{-1}(c)) \leftarrow fnx(c) \end{array}\right]}$
	$\text{fremove}_S\ \dfrac{c \in s.F}{s \xrightarrow{\text{fremove}(c)} s \left[\begin{array}{l} F \leftarrow F \setminus \{c\}, cst(c) \leftarrow 0, \\ fnx(fnx^{-1}(c)) \leftarrow fnx(c) \end{array}\right]}$
	$\text{fremove}_D\ \dfrac{c \in s.F}{s \xrightarrow{\text{fremove}(c)} s \left[\begin{array}{l} F \leftarrow F \setminus \{c\}, cst(c) \leftarrow 0, \\ fnx(fnx^{-1}(c)) \leftarrow fnx(c), \\ fpr(fnx(c)) \leftarrow fpr(c) \end{array}\right]}$

Table 4.14: Refined inserting operation on free list

finsert(c)	$\text{finsert}_U^B\ \dfrac{c \in s.H \setminus s.F}{s \xrightarrow{\text{finsert}(c)} s \left[\begin{array}{l} F \leftarrow F \cup \{c\}, cst(c) \leftarrow 1, \\ fnx(fbe) \leftarrow c, fnx(c) \leftarrow fnx(fbe) \end{array}\right]}$
	$\text{finsert}_U^E\ \dfrac{c \in s.H \setminus s.F}{s \xrightarrow{\text{finsert}(c)} s \left[\begin{array}{l} F \leftarrow F \cup \{c\}, cst(c) \leftarrow 1, \\ fnx^{-1}(fen) \leftarrow c, fnx(c) \leftarrow fen \end{array}\right]}$
	$\text{finsert}_S^M\ \dfrac{c \in s.H \setminus s.F \wedge p = max\{b' \mid b' \in s.F \wedge\ b' < c\}}{s \xrightarrow{\text{finsert}(c)} s \left[\begin{array}{l} F \leftarrow F \cup \{c\}, cst(c) \leftarrow 1, \\ fnx(p) \leftarrow c, fnx(c) \leftarrow fnx(p) \end{array}\right]}$
	$\text{finsert}_S^B\ \dfrac{c \in s.H \setminus s.F \wedge \forall b \in s.F \cdot c < b}{s \xrightarrow{\text{finsert}(c)} s \left[\begin{array}{l} F \leftarrow F \cup \{c\}, cst(c) \leftarrow 1, \\ fnx(fbe) \leftarrow c, fnx(c) \leftarrow fnx(fbe) \end{array}\right]}$

4.2.2 Basic Free List Operations

4.2.2.1 Removing, inserting, searching

We give a sample of rules defining the refinements of basic operations on the free list for a free chunk removing, insertion, and searching.

As shown in Table 4.13, the rule $\mathsf{fremove}_A$ refines the $\mathsf{hremove}$ basic operation for acyclic singly linked free lists; it simply updates the relation *fnx*. The rule $\mathsf{fremove}_S$ is defined for sorted singly linked list and it has the same definition as $\mathsf{fremove}_A$. The rule $\mathsf{fremove}_D$ specifies the removing operation in doubly linked lists.

In Table 4.14, the rules $\mathsf{finsert}_U^*$ specify refinements of an insertion operation for singly linked free lists which are unsorted. For sorted lists, the corresponding refinements are presented by rules $\mathsf{finsert}_S^*$. In the unsorted free list, the new chunk is inserted at the head or the end of the list. These two cases are specified by $\mathsf{finsert}_U^B$ and $\mathsf{finsert}_U^E$. The rule $\mathsf{finsert}_S^M$ specifies the case of the insertion of a chunk in the sorted free list. The inserting position is denoted by p in its precondition. The another case is specified by $\mathsf{finsert}_S^B$ when the inserted chunk has an address smaller than all the free chunks in the list, it is inserted at the beginning of the list.

The rules for $\mathsf{fsearch}$, shown in Table 4.15, refine the rules for $\mathsf{hsearch}$ with the specific fit policy. We define refinements of this operation, $\mathsf{fsearch}_{FF}$, $\mathsf{fsearch}_{BF}$ and $\mathsf{fsearch}_{NF}$ for first-fit, best-fit and next-fit policy. The next fit policy uses the variable *rp* as start of the search of the fitting chunk.

4.2.2.2 Splitting

The refinement of hsplit splitting operation, denoted by fsplit, is described in Table 4.16. The refined splitting operations in free-list are obtained by replacing the sub-operations with the refined operations. Rule fsplit_B calls the refined removing and inserting operations, $\mathsf{fremove}$ and $\mathsf{finsert}$. The rule fsplit_E's definition is similar to the rule hsplit_E.

4.2.2.3 Merging

The refined merging operations for free lists are presented in Table 4.17. The basic merging operations, merging a chunk with its left and right free neighbors

Table 4.15: Refined searching operation on free list

$\mathsf{fsearch}(n) : c$	$\mathsf{fsearch}^S_{FF}\ \dfrac{c \in s.F \wedge \mathsf{fit}(c,n) \leq s.csz(c) \wedge \\ (\forall b \in s.F \cdot b < c \Rightarrow s.csz(b) < \mathsf{fit}(b,n))}{s \xrightarrow{\mathsf{fsearch}(n):c} s}$
	$\mathsf{fsearch}^S_{BF}\ \dfrac{c \in s.F \wedge \mathsf{fit}(c,n) \leq s.csz(c) \wedge \\ (\forall b \in F \cdot (c \neq b \wedge \mathsf{fit}(b,n) \leq csz(b)) \\ \Rightarrow (csz(b) - \mathsf{fit}(b,n) \geq csz(c) - \mathsf{fit}(c,n)))}{s \xrightarrow{\mathsf{fsearch}(n):c} s}$
	$\mathsf{fsearch}^S_{NF}\ \dfrac{c \in s.F \wedge \mathsf{fit}(c,n) \leq s.csz(c) \wedge \\ \forall k, \ell \geq 0, b \in s.F \cdot (c = s.fnx^k(rp) \wedge \\ b = s.fnx^\ell(rp) \wedge \mathsf{fit}(b,n) \leq csz(b)) \Rightarrow (k < \ell)}{s \xrightarrow{\mathsf{fsearch}(n):c} s}$
	$\mathsf{fsearch}^F_*\ \dfrac{\forall b \in s.F \cdot s.csz(b) < \mathsf{fit}(b,n)}{s \xrightarrow{\mathsf{fsearch}(n):\mathsf{nil}} s}$

Table 4.16: Refinements of split operation on free list

$\mathsf{fsplit}(c,n) : c'$	$\mathsf{fsplit}_B\ \dfrac{c \in s.F \wedge 0 < n \leq s.csz(c) \wedge c' = c + n \\ \wedge (s \xrightarrow{\mathsf{fremove}(c)} s_1 \xrightarrow{\mathsf{finsert}(c')} s_2)}{s \xrightarrow{\mathsf{fsplit}(c,n):c'} s_2 \begin{bmatrix} H \leftarrow H \cup \{c'\}, \\ csz[c \leftarrow n, c' \leftarrow csz(c) - n], \\ cnx[c \leftarrow c', c' \leftarrow cnx(c)] \end{bmatrix}}$
	$\mathsf{fsplit}_E\ \dfrac{c \in s.F \wedge 0 < n \leq s.csz(c) \wedge c' = c + s.csz(c) - n}{s \xrightarrow{\mathsf{fsplit}(c,s):c'} s \begin{bmatrix} H \leftarrow H \cup \{c'\}, cst(c') \leftarrow 0 \\ csz[c \leftarrow csz(c) - n, c' \leftarrow n], \\ cnx[c \leftarrow c', c' \leftarrow cnx(c)] \end{bmatrix}}$

denoted by fmerge_L and fmerge_R, refine the corresponding operations of the heap list level. fmerge_N is the refinement of hmerge_N.

Table 4.17: Refinements of merge operation on free list

$\mathsf{fmerge}(b) : x$	$\mathsf{fmerge}_R^S \dfrac{b \in s.F \wedge c = s.cnx(b) \wedge c \in s.F \wedge s \xrightarrow{\mathsf{fremove}(c)} s_1}{s \xrightarrow{\mathsf{fmerge}(b):b} s_1 \left[\begin{array}{l} H \leftarrow H \setminus \{c\}, cnx[b \leftarrow cnx(c)], \\ csz[b \leftarrow csz(b) + csz(c)] \end{array}\right]}$
	$\mathsf{fmerge}_R^F \dfrac{b \in s.F \wedge c = s.cnx(b) \wedge c \in s.H \setminus s.F}{s \xrightarrow{\mathsf{fmerge}(b):b} s}$
	$\mathsf{fmerge}_L^S \dfrac{b \in s.F \wedge s.cnx(c) = b \wedge c \in s.F \wedge s \xrightarrow{\mathsf{fremove}(b)} s_1}{s \xrightarrow{\mathsf{fmerge}(b):c} s_1 \left[\begin{array}{l} H \leftarrow H \setminus \{b\}, cnx[c \leftarrow cnx(b)], \\ csz[c \leftarrow csz(b) + csz(c)] \end{array}\right]}$
	$\mathsf{fmerge}_L^F \dfrac{b \in s.F \wedge s.cnx(c) = b \wedge c \in s.H \setminus s.F}{s \xrightarrow{\mathsf{fmerge}(b):b} s}$
$\mathsf{fmerge}_N(b)$	$\mathsf{fmerge}_N^S \dfrac{b \in s.F \wedge s \xrightarrow{\mathsf{fmerge}_R(b):b} s_1 \xrightarrow{\mathsf{fmerge}_L(b):c} s_2}{s \xrightarrow{\mathsf{fmerge}_N(b)} s_2}$
	$\mathsf{fmerge}_N^F \dfrac{b \in s.F \wedge s.cnx(b) \notin s.F \wedge s.cnx^{-1}(b) \notin s.F}{s \xrightarrow{\mathsf{fmerge}_N(b)} s}$

4.2.3 Models for Free List SDMA

We developed several models which describe the free list and refine the models in heap list level. The two refinement branches are shown in Figure 3.3, starting from model **MHE** and model **MA**.

4.2.3.1 Method `init`

The refinements for the `init` method are specified in Table 4.18. They describe the initialization for different cases of free list. Each rule initializes the state

Table 4.18: Refinements of `init` on free list

finit	finit_A	$s \xrightarrow{\mathsf{finit}()} \left\langle \begin{array}{l} H \leftarrow \{\mathsf{hst}\}, F \leftarrow \{\mathsf{hst}\}, cst(\mathsf{hst}) \leftarrow 1, \\ cnx(\mathsf{hst}) \leftarrow \mathsf{hli}, csz(\mathsf{hst}) \leftarrow \mathsf{hli} - \mathsf{hst}, \\ fnx_A(fbe) \leftarrow \mathsf{hst}, fnx_A(\mathsf{hst}) \leftarrow fen \end{array} \right\rangle$
	finit_D	$s \xrightarrow{\mathsf{finit}()} \left\langle \begin{array}{l} H \leftarrow \{\mathsf{hst}\}, F \leftarrow \{\mathsf{hst}\}, cst(\mathsf{hst}) \leftarrow 1, \\ cnx(\mathsf{hst}) \leftarrow \mathsf{hli}, csz(\mathsf{hst}) \leftarrow \mathsf{hli} - \mathsf{hst}, \\ fpr(fen) \leftarrow \mathsf{hst}, fpr(\mathsf{hst}) \leftarrow fbe, \\ fnx_A(fbe) \leftarrow \mathsf{hst}, fnx_A(\mathsf{hst}) \leftarrow fen \end{array} \right\rangle$
	finit_C	$s \xrightarrow{\mathsf{finit}()} \left\langle \begin{array}{l} H \leftarrow \{\mathsf{hst}\}, F \leftarrow \{\mathsf{hst}\}, cst(\mathsf{hst}) \leftarrow 1, \\ cnx(\mathsf{hst}) \leftarrow \mathsf{hli}, csz(\mathsf{hst}) \leftarrow \mathsf{hli} - \mathsf{hst}, \\ fnx_C(\mathsf{hst}) \leftarrow fen, fnx_C(fbe) \leftarrow \mathsf{hst}, \\ fnx_C(fen) \leftarrow fbe \end{array} \right\rangle$
	finit_N	$s \xrightarrow{\mathsf{finit}()} \left\langle \begin{array}{l} H \leftarrow \{\mathsf{hst}\}, F \leftarrow \{\mathsf{hst}\}, cst(\mathsf{hst}) \leftarrow 1, \\ cnx(\mathsf{hst}) \leftarrow \mathsf{hli}, csz(\mathsf{hst}) \leftarrow \mathsf{hli} - \mathsf{hst}, \\ fnx(fbe) \leftarrow \mathsf{hst}, fnx(\mathsf{hst}) \leftarrow fen, rp \leftarrow \mathsf{hst} \end{array} \right\rangle$

defined in Table 4.12. Rules finit_A and finit_C present the initialization for the singly linked acyclic and cyclic free list. Rule finit_D initializes the acyclic doubly linked free list.

4.2.3.2 Method `alloc`

The refinements for the internal allocation operation and the `alloc` method are defined in Table 4.19. They refine rules described in Table 4.7 by replacing basic operations in heap list with the refined ones.

4.2.3.3 Method `free`

The rule $\mathsf{ffree}^S_{\mathrm{eager}}$ uses the operation finsert (instead of hinsert) to update the links used by the free list and then tries to merge the inserted chunk with its neighbours using the operation fmerge_N which refines hmerge_N.

4.2.3.4 Statistics

Table 4.21 sums up the main ingredients used by the refinement directions to obtain the models presented in Figure 3.3. Like for the heap list models, the

Table 4.19: Refinements of `alloc` method on free list

$\textsf{falloc}_\textsf{i}(n) : b$	$\textsf{falloc}^S_{\text{fit}}\ \dfrac{\begin{array}{c} s \xrightarrow{\textsf{fsearch}(n):c} s \land c \in s.F \land \textsf{fit}(c,n) = s.csz(c) \\ \land(p = c + \textsf{chd}) \land s \xrightarrow{\textsf{fremove}(c)} s_1 \end{array}}{s \xrightarrow{\textsf{falloc}_\textsf{i}(n):p} s_1}$
	$\textsf{falloc}^S_{\text{split}}\ \dfrac{\begin{array}{c} s \xrightarrow{\textsf{fsearch}(n):c} s \land c \in s.F \land \textsf{fit}(c,n) < s.csz(c) \\ \land(p = b + \textsf{chd}) \land s_1 \xrightarrow{\textsf{fsplit}(c,\textsf{fit}(c,n)):b} s_2 \end{array}}{s \xrightarrow{\textsf{falloc}_\textsf{i}(n):p} s_2}$
	$\textsf{falloc}^F\ \dfrac{s \xrightarrow{\textsf{fsearch}(n):\textsf{nil}} s_1}{s \xrightarrow{\textsf{falloc}_\textsf{i}(n):\textsf{nil}} s_1}$
$\textsf{falloc}(n) : b$	$\textsf{falloc}^S_{\text{eager}}\ \dfrac{s \xrightarrow{\textsf{falloc}_\textsf{i}:p} s_1}{s \xrightarrow{\textsf{falloc}(n):p} s_1}$ $\quad \textsf{falloc}^F_{\text{eager}}\ \dfrac{s \xrightarrow{\textsf{falloc}_\textsf{i}(n):\textsf{nil}} s}{s \xrightarrow{\textsf{falloc}(n):\textsf{nil}} s}$

Table 4.20: Refinement of methods for free list

$\textsf{ffree}(p) : b$	$\textsf{ffree}^S_{\text{eager}}\ \dfrac{p = b + \textsf{chd} \land b \in s.H \setminus s.F \land s \xrightarrow{\textsf{finsert}(b)} s_1 \xrightarrow{\textsf{fmerge}_N(b)} s_2}{s \xrightarrow{\textsf{ffree}(p):\text{true}} s_2}$
	$\textsf{ffree}^F_{\text{eager}}\ \dfrac{\forall b \in s.H \setminus s.F \cdot p \neq b + \textsf{chd}}{s \xrightarrow{\textsf{ffree}(p):\text{false}} s}$

rules are specified as events in the machine. We prove with the Rodin tool the following correctness and refinement theorem. Table 4.22 provides statistics about the proofs conducted to obtain the below theorem.

Theorem 4.2 (Correctness of the operations)**.** *Every operation of a model for SDMA with free list preserves the invariants of the model. Moreover, the refinement relations in Figure 3.3 are valid.* ■

Table 4.21: Overview of free list models

Models	State&New invariants	Rules			
		`init`	remove	insert	search
MUA	s,fnx_A	finit_A	$\mathsf{fremove}_A$	$\mathsf{finsert}_U$	–
MSA	s,fnx_A, I_S	finit_A	$\mathsf{fremove}_S$	$\mathsf{finsert}_S$	–
MSC	s,fnx_C, I_C, I_S	finit_C	$\mathsf{fremove}_S$	$\mathsf{finsert}_S$	–
MSAB	s,fnx_A, I_S	finit_A	$\mathsf{fremove}_S$	$\mathsf{finsert}_S$	$\mathsf{fsearch}_{BF}$
MSAF	s,fnx_A, I_S	finit_A	$\mathsf{fremove}_S$	$\mathsf{finsert}_S$	$\mathsf{fsearch}_{FF}$
MSAN	s_N,fnx_A, I_S, I_{rp}	finit_N	$\mathsf{fremove}_S$	$\mathsf{finsert}_S$	$\mathsf{fsearch}_{NF}$
MSCN	s_N,fnx_C, I_C, I_S, I_{rp}	finit_C	$\mathsf{fremove}_S$	$\mathsf{finsert}_S$	$\mathsf{fsearch}_{NF}$
MUABD	s_D,fnx_A,fpr_A, I_{fpr}	finit_D	$\mathsf{fremove}_D$	$\mathsf{finsert}_U$	$\mathsf{fsearch}_{BF}$
MASA	s,fnx_A, I_S	finit_A	$\mathsf{fremove}_A$	$\mathsf{finsert}_U$	–
MASAF	s,fnx_A, I_S	finit_A	$\mathsf{fremove}_A$	$\mathsf{finsert}_U$	$\mathsf{fsearch}_{FF}$

Table 4.22: Statistics on proofs

Models	LOC	Proof	Automatically	Interactive
		obligations	discharged	proofs
MUA	219	36	30(83%)	6(17%)
MSA	197	41	27(66%)	14(34%)
MSC	205	37	30(82%)	7(18%)
MSAB	202	2	2(100%)	0(0%)
MSAF	202	2	2(100%)	0(0%)
MSAN	200	2	2(100%)	0(0%)
MSCN	221	40	36(88%)	4(12%)
MUABD	241	9	9(100%)	0(0%)
MASA	182	21	18(85.6%)	3(14.4%)
MASF	186	2	2(100%)	0(0%)

4.3 Applications of Formal Models

Refinement towards SDMA implementations In our models, the constants and state elements abstract the following implementation details: the boundaries of the memory region used by the SDMA (variables hst and hli), the type of the header (constants cdt, cal, mappings csz, cst, cnx, cpr, fnx, fpr), the algorithm deciding which is the number of bytes needed to satisfy a client request (mapping fit), and the boundaries of the free list (variables fbe and fen). Let $\mathbb{S}$ denote the above set of symbols.

The refinement to code is defined by associating to each element in $\mathbb{S}$ an expression using the types and variables of the SDMA implementation such that the semantics of the element is fulfilled. We provide three examples of such refinements in Table 4.23. Notice that some elements of $\mathbb{S}$ may be left unspecified (entries with '$-$') if they are not used in the model (we omit the elements of $\mathbb{S}$ which are not specified in all the examples from the table). We obtain these relations by inspecting the code of each allocator. However, we believe that some automatic analysis may be designed to extract automatically such information.

Code generation The elements of $\mathbb{S}$ and the rules presented in the previous sections may be exploited to generate code for SDMA modelled by our specifications. In particular, the rules provide an operational semantics of SDMA methods and a decomposition of these methods into calls to list (heap or free) operations. The invariants specified for each state may be translated into code and therefore provide means for run time verification of the correctness of a particular state of the SDMA.

Model-based testing We experimented model-based test case generation using the tool published in [MLL09] which implements several methods for Event-B models. We focused on the generation of test cases that are finite sequences of calls to `alloc` and `free` and end in a fail behaviour of `free` (ffree^F). A first observation concerns the scalability of this tool, which is not related with its particular implementation, but with the methodology it employs, which is based on queries to SMT-solvers. We were able to generate test cases for models which are on top of our hierarchy in Figure 3.3. The models in the lower part, which have more complex invariants, cannot be dealt by the theories available in the SMT-solvers connected with this tool. We expect that this situation is

Table 4.23: Examples of refinement to code

$\mathbb{S}$	TOPSY [RJLP03] $\models$ **MHL**	LA [Ald08] `text` $\models$ **MSA**$_{FF}$
hst	`start`	`_hsta`
hli	`end`	`sbrk(0)`
cdt	`sizeof(HmEntryDesc)`	`sizeof(HDR)`
cal	4	`sizeof(HDR)`
csz(x)	`(long)x->next-(long)x`	`x->size*sizeof(HDR)`
cst(x)	`x->status`	–
cnx(x)	`x->next`	`(HDR*)x+x->size`
fnx	–	`x->ptr`
fbe	–	`frhd`
fit(c,n)	(*csz*(c) `>((n+3)&0x0F+8))?` `((n+3)&0x0F))`: *csz*(c)	`(n+3)/4 + 1`

reproduced in other model-based test case generators using different input languages. Our hierarchy is a solution for this scalability problem because it provides reasonable size abstractions for the complex models of free list SDMA. A second observation is related with the concretisation of the signature $\mathbb{S}$ to the code under test. Not all the elements of $\mathbb{S}$ shall be instantiated to apply the tool: only the mappings *csz* and fit shall be fixed because they are important for inferring the parameters for the calls to `alloc`; the other elements of $\mathbb{S}$ can be dealt in a symbolic way by the tool.

Static analysis Several static analysis techniques have been developed to analyse particular SDMA implementations, e.g. [CDOY06, LR15, FS16]. They employ complex abstractions of SDMA state to capture precisely some properties of SDMA, e.g., the shape of the lists, the overlapping between heap and free list. These abstractions are usually based on second order logics over graphs to capture reachability between locations and shapes of data structures. The analyses aim to infer the invariants and the pre/post-conditions of SDMA methods. In this context, our models provide a sound reference for the inferred specifications and highlight the logic fragments needed to capture precisely the SDMA properties. These logic fragments may inspire the design of new abstractions for such analysis. We present such an analysis in part II.

4.4 Related Work and Conclusion

To our knowledge, this work[FS17, FSG$^+$17] is the first defining a complete hierarchy of models for the full class of list based SDMA. The same approach of top-down modelling is employed in [SAPF15] to obtain the formal specification of one SDMA, the TLSF allocator [MRCR04]. Our set of specifications is complete for the techniques utilized in the list based SDMA.

Several projects report on the mechanical proofs using theorem provers of (partial) correctness of code for specific purpose SDMA, e.g., [MAY06, TKN07a, KEH$^+$09, HP09, Chl11]. Most of these works use Separation Logic (SL) [ORY01] which provides a scalable and expressive reasoning framework. [MAY06] targets the verification of the TOPSY SDMA using the Coq theorem prover. For this, they developed a Coq library for SL which is employed to specify only some of the invariants we provide for the heap list. The Bedrock framework [Chl11] is another Coq library that has been used to verify SDMA code with only acyclic free lists and no coalescing. [TKN07a] proposes a formal memory model that captures both the low level (heap list) and the abstract level (free list) of the memory organisation in SDMA. The low level model is based on the set theory available in Isabelle/HOL; the abstract level uses a fragment of SL encoded in Isabelle/HOL. The approach was used to formally verify the code of the SDMA used by the L4 microkernel [KEH$^+$09]. [HP09] employs Boogie and Z3 to verify a realistic garbage collector whose code has been annotated with a particular region logic. Our work is complementary to these projects. We provide reusable and complete specifications for all list based SDMA by applying several refinement steps, while they focus on the verification of specifications for a particular SDMA code.

Verification of SDMA code by static analysis has been considered in [CDOY06, LR15, FS16]. All these methods infer only some properties for particular allocators. Indeed, they employ fragments of SL or some logics over arrays which are not expressive enough to cover fully the invariants of the SDMA analysed (e.g., the fit policy). Our work provides reference specifications to compare with the inferred ones, in a logic fragment more general than SL. It could motivate the extension or the direct application of general purpose methods based on SL, e.g., [CDNQ12, QHL$^+$14].

Part II

Static Analysis of Sequential Dynamic Memory Allocators

CHAPTER 5

Separation Logic Fragment for SDMA

Verification of implementations of SDMA needs expressive formalism to reason about their complex program configurations. In this thesis, we propose an extension of the symbolic heap graphs fragment [DOY06] of *Separation Logic (SL)* for reasoning SDMA, called SLMA. Our logic fragment can track the complex structural and numerical properties on the structure of memory and on its size and content. SLMA is parameterized by the type of chunk header and contains a set of predicates describing the heap list and the free list. Also, SLMA provides the composition operator for linking the two abstractions of the heap, the heap list and the free list. This operator increases the readability and modularity of specifications as well as the modularity of the static analysis method designed in Chapter 6, which is based on a lattice built over formulas in SLMA. For this logic fragment, we study the decidability of satisfiability and entailment checking problems.

This chapter is structured as follows. Section 5.1 presents preliminaries on Separation Logic (SL) and how it is used to reason about heap-manipulating programs. Section 5.2 formally defines the logic SLMA and Section 5.3 study the properties of SLMA.

5.1 Preliminaries

5.1.1 Hoare Logic

Hoare logic was proposed by Tony Hoare [Hoa69] and is inspired by the earlier work of Floyd [Flo93]. It is a formal system used to show the correctness of computer programs. The key feature of it is the *Hoare triple*. A Hoare triple is composed of two assertions ϕ and ψ, which are formulas in predicate logic, and a command C of the program:

$$\{\phi\}C\{\psi\}$$

This triple means that given a program state where ϕ holds, after executing C and if C terminates, ψ holds in the new state. Usually, ϕ is named *precondition* and ψ is *postcondition*. Hoare logic gives axioms and inference rules for constructs of a simple imperative programming language. For example, the assignment and sequencing axiom schemata are defined as:

$$\frac{}{\{\phi[e/x]\}x := e\{\phi\}} \qquad \frac{\{\phi\}C_1\{\psi\} \quad \{\psi\}C_2\{\varphi\}}{\{\phi\}C_1;C_2\{\varphi\}}$$

The assignment axiom means that the value of a variable x after the execution of an assignment command $x := e$ is equal to the value of the expression e in the state before executing it. Here, the notation $\phi[e/x]$ denotes the result of replacing all occurrences of x in ϕ by e. For example, the following simple Hoare triple

$$\{x = 1 \wedge y = 1\}\ x := 2\ \{x = 2 \wedge y = 1\}$$

holds for programming languages without pointers. The assignment $x := 2$ does not affect the value of y. When the programming language uses pointers, which involves the addressable memory, the above assignment axiom is not valid. Fox example, in the following triple

$$\{*x = 1 \wedge *y = 1\}\ *x := 2\ \{*x = 2 \wedge *y = 1\},$$

two pointer variables x and y may be aliased, i.e., they may refer to the same region of the memory.

5.1.2 Symbolic Heap Fragment of Separation Logic

To reason on programs that access and mutate data in memory, Reynolds, O'Hearn et al. developed Separation Logic [ORY01,Rey02] which is an extension of Hoare logic. The problem of aliasing mentioned above is solved thanks to a special operator, called *separating conjunction*, denoted by $*$. $\phi * \psi$ asserts that ϕ and ψ hold in separate portions of the memory. The above triple is specified as follows:

$$\{x \mapsto 1 * y \mapsto 1\} \ *x := 2 \ \{x \mapsto 2 * y \mapsto 1\}$$

The assignment only updates the content of the memory pointed by x while the memory region pointed by y is not affected.

Instead of recalling all elements of the Separation Logic in this section, we introduce one fragment of Separation Logic, called *symbolic heaps* [BCO05b, DOY06]. Our logic explained in the next chapter is the extension of the symbolic heaps fragment. Notice that there are many fragments of Separation Logic with different restrictions.

Syntax: Let PVar be a set of program variables, ranged over using x, y, z and LVar a set of logical variables, ranged over using x', y', z'. The set of logical variables is disjoint from program variables. The symbolic heaps fragment formulas are specified in the following grammar (P is a spatial predicate symbol whose parameter is a vector of variables, denoted by $\vec{E}$, a special variable nil represents the null address):

$$\begin{array}{rcll}
E, F & ::= & \mathsf{Pvar} \mid \mathsf{LVar} \mid \mathsf{nil} & \\
\Pi & ::= & \mathsf{true} \mid E = F \mid E \neq F \mid \Pi \wedge \Pi & \text{pure formulas} \\
\Sigma & ::= & \mathsf{emp} \mid E \mapsto F \mid \Sigma * \Sigma \mid P(\vec{E}) & \text{spatial formulas} \\
\phi & ::= & \vee \exists \vec{x'}. \Pi \wedge \Sigma &
\end{array}$$

Definition 5.1 (Symbolic heap)**.** A symbolic heap $\Pi \wedge \Sigma$ consists of a finite set Π of equalities and disequalities, and a finite set Σ of heap predicates. In the symbolic heap $\Pi \wedge \Sigma$, Π is called the pure part of it and Σ is the spatial part. ■

Semantics: The domain of heap addresses is denoted by $\mathbb{A}$ while the domain of values stored in the heap is generically denoted by $\mathbb{V}$. Separation Logic is interpreted over programs states $m = (s, h)$ composed of a *stack* s and a *heap* h. A stack is a function mapping program variables and to values. A heap is a partial function mapping each memory address to the content at this address. We denote by $\mathsf{dom}(f)$ the domain of function f.

Given two heaps, h_1 and h_2, they are said to be disjoint, denoted by $h_1 \bot h_2$, if the domains of h_1 and h_2 are disjoint, i.e., $\mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) = \varnothing$. The union

$$\begin{array}{rclll}
s & \in & \mathsf{Stacks} & \triangleq & \mathsf{PVar} \cup \mathsf{LVar} \to \mathbb{V} \\
h & \in & \mathsf{Heaps} & \triangleq & \mathbb{A} \rightharpoonup \mathbb{V} \\
m & \in & \mathsf{States} & \triangleq & \mathsf{Stacks} \times \mathsf{Heaps}
\end{array}
\qquad
\begin{array}{rcl}
[\![E]\!] & \in & \mathsf{Stacks} \to \mathbb{V} \\
\mathbb{A} & \subseteq & \mathbb{V} \\
\mathsf{nil} & \in & \mathbb{A}
\end{array}$$

Figure 5.1: Memory model of Separation Logic

of two heaps is denoted by $h_1 \uplus h_2$, and $\mathsf{dom}(h_1 \uplus h_2) = \mathsf{dom}(h_1) \cup \mathsf{dom}(h_2)$. We denote by $s, h \models \phi$ the satisfaction of an assertion ϕ by the stack s and heap h. The basic spatial assertions are:

- emp asserts an empty heap, i.e., $(s, h) \models \mathsf{emp}$ if $\mathsf{dom}(h) = \varnothing$;
- $E \mapsto F$ asserts that E points to F and the heap has exactly one memory cell, i.e., $(s, h) \models E \mapsto F$ if $h(s(E)) = s(F) \wedge \mathsf{dom}(h) = \{s(E)\}$;
- $\Sigma_1 * \Sigma_2$ is the separating conjunction asserting that the heap is divided into two disjoint parts such that one satisfies Σ_1 and the other satisfies Σ_2, i.e., $(s, h) \models \Sigma_1 * \Sigma_2$ if $\exists h_1, h_2$ such that $h_1 \perp h_2 \wedge (s, h_1) \models \Sigma_1 \wedge (s, h_2) \models \Sigma_2$ and $h = h_1 \uplus h_2$;

Separation Logic enables local reasoning. This fact is formalized by the *frame rule* which states that if the Hoare triple $\{\phi\}C\{\psi\}$ holds, then $\{\phi * \varphi\}C\{\psi * \varphi\}$ also holds if the variables referenced by C are disjoint from the variables in φ.

$$\frac{\{\phi\}C\{\psi\}}{\{\phi * \varphi\}C\{\psi * \varphi\}}$$

Inductively defined predicates: To specify an unbounded heap, usually Separation Logic is equipped with inductive predicates. For example, the segment of simply linked list structure is defined using the following rule:

$$\begin{aligned}
\mathsf{lsg}(E, F) \triangleq\ & (\mathsf{emp} \wedge E = F) \\
& \vee\ \exists G \cdot E \mapsto G * \mathsf{lsg}(G, F) \wedge E \neq F
\end{aligned}$$

The predicate $\mathsf{lsg}(E, F)$ specifies either the empty list segment $E = F$ or the list that can be split into two disjoint parts such that one part is that location E stores the address G and G is the starting point of a list segment ending in F in the other part.

5.2 Separation Logic for Memory Allocators

We formalise in this section a fragment of Separation Logic, SLMA, which is used with some syntactical restrictions to define the values of our abstract domain in Chapter 6. SLMA extends the symbolic heap fragment proposed in [DOY06, CDOY06] with data constraints over words storing integers and spatial predicates specifying different types of heap and free lists used in SDMA.

5.2.1 Syntax of SLMA

Variables: In our programs, i.e., the implementation of SDMA, a heap consists of a set of memory cells. Each memory cell has a heap address. Let AVar be a set of *location variables* representing heap addresses ranged over using X, Y, Z. Let SVar be a set of *sequence variables*, interpreted as sequences of heap addresses. A sequence variable could represent an empty term denoted by ϵ or a memory cell $[X]$ with address X or a sequence of heap addresses $[X_1, X_2, ..., X_n]$ whose length is n. Let $w[i]$ denotes by the address at index i in the sequence represented by w. Given two sequences of heap addresses, w_1 and w_2, they can be composed as one bigger sequence. The concatenated sequence is denoted by $w_1.w_2$. Let IVar be a set of *integer variables*. The full set of *logic variables* in SLMA is denoted by $\mathsf{LVar} = \mathsf{AVar} \cup \mathsf{SVar} \cup \mathsf{IVar}$. The vector of variables is denoted by $\vec{x}$.

Function symbols of fields: The field is the offset at some address. To simplify the presentation, we fix `HDR`, the type of chunk headers, and its fields $\mathbb{F} = \{\mathtt{size}, \mathtt{fnx}, \mathtt{isfree}\}$. Let $\mathcal{F} = \{\mathcal{F}_{\mathtt{size}}, \mathcal{F}_{\mathtt{fnx}}, \mathcal{F}_{\mathtt{isfree}}\}$ be a set of function symbols, each function takes one argument $\mathcal{F}_f : \mathsf{LVar} \rightharpoonup \mathbb{V}$ (f is limited in our logic to the set $\mathbb{F}$ but may vary). The term $\mathcal{F}_{\mathtt{size}}(X)$ represents the value that the field `size` stores in the chunk header at location X. Thus, the field access $X.\mathtt{size}$ is represented by $\mathcal{F}_{\mathtt{size}}(X)$.

Formulas: The syntax of formulas is given in Table 5.1. Formulas are in disjunctive normal form. Each disjunct is built from a pure formula Π and a spatial formula Σ. Pure formulas Π characterise the values of logic variables using comparisons between location variables, e.g., $X - Y = 0$, constraints Δ over integer terms, and sequence constraints. We let constraints in Δ unspecified, though we assume that they belong to decidable theories, e.g., linear arithmetic.

Table 5.1: Logic syntax

$X, Y, \mathsf{nil}, \mathsf{hli}$	$\in$	AVar	location variables
W	$\in$	SVar	sequence variables
i, j	$\in$	IVar	integer variables
$\#$	$\in$	$\{=, \neq, \leq, \geq\}$	comparison operators
x	$\in$	$\mathsf{LVar} = \mathsf{AVar} \cup \mathsf{SVar} \cup \mathsf{IVar}$	logic variable
$\vec{x}, \vec{y}$	$\in$	LVar^*	vectors of variables
$\mathbb{F}$	$=$	$\{\texttt{size}, \texttt{fnx}, \texttt{isfree}\}$	fields in chunk header
$\mathcal{F}_f(X)$			field access term ($f \in \mathbb{F}$)
t, Δ			integer term resp. formula

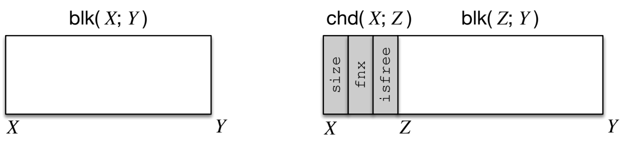
φ	$::=$	$\Pi \wedge \Sigma \mid \varphi \vee \varphi \mid \exists x \cdot \varphi$	formulas

L	$::=$	$X \mid \mathcal{F}_{\texttt{fnx}}(X)$	
Π	$::=$	$A \mid \forall X \in W \cdot A \Rightarrow A \mid W = w \mid \Pi \wedge \Pi$	pure formulas
A	$::=$	$L - L \# t \mid \Delta \mid A \wedge A$	location/integer constraints
w	$::=$	$\epsilon \mid [X] \mid W \mid w.w$	sequence terms

Σ	$::=$	$\Sigma_H \Supset \Sigma_F$	spatial formulas
Σ_H	$::=$	$\mathsf{emp} \mid X \mapsto x \mid \mathsf{blk}(X; Y) \mid \mathsf{chd}(X; Y)$	heap formulas
		$\mid \mathsf{chk}(X; Y) \mid \mathsf{hls}(X; Y)[W] \mid \mathsf{hlsc}(X, f_p; Y, f_l)[W]$	
		$\mid \Sigma_H * \Sigma_H$	
Σ_F	$::=$	$\mathsf{emp} \mid \mathsf{fck}(X; Y) \mid \mathsf{fls}(X; Y)[W]$	
		$\mid \mathsf{flso}(X, x; Y, y)[W] \mid \Sigma_F * \Sigma_F$	free list formulas

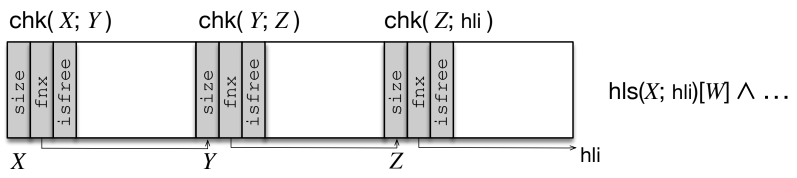
The integer terms t are built over integer variables and field accesses using classic arithmetic operations and constants. We denote by $\Pi_\forall$ (resp. Π_W, $\Pi_\exists$) the set of sub-formulas of Π built from universal constraints (resp. sequence constraints, quantifier free arithmetic constraints).

Inductive predicates: A spatial formula has two components: Σ_H specifies the heap list and the locations outside the region managed by the SDMA; Σ_F specifies only the free list. The operator $\Supset$ ensures that all locations specified by Σ_F are start addresses of chunks in the heap list.



The block atom $\mathsf{blk}(X;Y)$ [CDOY06] holds iff the heap contains a sequence of bytes starting at location X ending before the location Y. We call the memory region specified by blk predicate the *raw* memory region. The other predicates are derived from blk and defined in Table 5.2. The chunk header atom $\mathsf{chd}(X;Y)$ specifies the chunk header based on the atom $\mathsf{blk}(X;Y)$. We do not expose the values stored in the chunk header because we found it easier to specify the coalescing of block and chunk atoms into a single block.

The chunk atom $\mathsf{chk}(X;Y)$ and the free chunk atom $\mathsf{fck}(X;Y)$ are defined using $\mathsf{blk}(X;Y)$ by constraining the size of the block and the values stored in the header. The heap list predicate $\mathsf{hls}(X;Y)[W]$ specifies a heap list segment in the memory region with chunk start addresses stored (in order) in W. $\mathsf{hls}(X;X)[\epsilon]$ describes the empty heap list segment and is equivalent to emp.

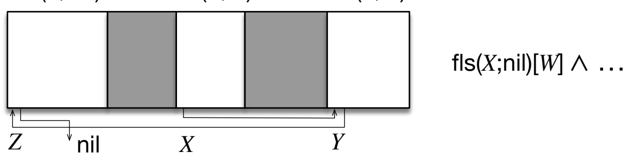


As the above picture shows, the three contiguous chunks in the heap can be specified as a heap list $\mathsf{hls}(X;\mathsf{hli})[W]$ or two non-overlapping parts $\mathsf{chk}(X;Y) * \mathsf{hls}(Y;\mathsf{hli})[W']$ (here hli denotes the address after the end of the heap which has the same meaning as the definition in Section 2.1.1, page 9). Obviously, the heap list can not be cyclic, thus $\mathsf{hls}(X;Y)[X] * \mathsf{hls}(Y;X)[Y]$ is a invalid formula.

Table 5.2: Derived predicates

$\mathsf{chd}(X;Y) \triangleq$	$\mathsf{blk}(X;Y) \wedge \texttt{sizeof(HDR)} = Y - X \wedge X \equiv_{\texttt{sizeof(HDR)}} 0$
$\mathsf{chk}(X;Y) \triangleq$	$\exists Z \cdot \mathsf{chd}(X;Z) * \mathsf{blk}(Z;Y)$
	$\wedge\, \mathcal{F}_{\texttt{size}}(X) \times \texttt{sizeof(HDR)} = Y - X$
$\mathsf{fck}(X;Y) \triangleq$	$\exists Z \cdot \mathsf{chk}(X;Z) \wedge \mathcal{F}_{\texttt{isfree}}(X) = 1 \wedge \mathcal{F}_{\texttt{fnx}}(X) = Y$
$\mathsf{hls}(X;Y)[W] \triangleq$	$\mathsf{emp} \wedge X = Y \wedge W = \epsilon$
	$\vee\ \exists Z, W' \cdot \mathsf{chk}(X;Z) * \mathsf{hls}(Z;Y)[W']$
	$\wedge\, W = \texttt{[}X\texttt{]}.W'$
$\mathsf{hlsc}(X, f_p; Y, f_l)[W] \triangleq$	$\mathsf{emp} \wedge X = Y \wedge W = \epsilon \wedge 0 \leq f_p + f_l \leq 1$
	$\vee\ \exists Z, W', f \cdot \mathsf{chk}(X;Z) * \mathsf{hlsc}(Z, f; Y, f_l)[W']$
	$\wedge\, W = \texttt{[}X\texttt{]}.W'$
	$\wedge\ f = \mathcal{F}_{\texttt{isfree}}(X) \wedge 0 \leq \mathcal{F}_{\texttt{isfree}}(X) + f_p \leq 1$
$\mathsf{fls}(X;Y)[W] \triangleq$	$\mathsf{emp} \wedge X = Y \wedge W = \epsilon$
	$\vee\ \exists Z, W' \cdot \mathsf{fck}(X;Z) * \mathsf{fls}(Z;Y)[W']$
	$\wedge\, W = \texttt{[}X\texttt{]}.W' \wedge X \neq Y$
$\mathsf{flso}(X, x; Y, y)[W] \triangleq$	$\mathsf{emp} \wedge X = Y \wedge W = \epsilon \wedge x - y \leq 0$
	$\vee\ \exists Z, W' \cdot \mathsf{fck}(X;Z) * \mathsf{flso}(Z, X; Y, y)[W']$
	$\wedge\ W = \texttt{[}X\texttt{]}.W' \wedge x - X \leq 0$

The compact heap list $\mathsf{hlsc}(X, f_p; Y, f_l)[W]$ specifies a heap list segment where the free-chunks are not consecutive, where f_p and f_l are the free status of the before the first resp. last chunk in the list. We use a numerical constraint $0 \leq f_1 + f_2 \leq 1$ to specify that two contiguous chunks can not be both free, i.e., both have a free status of 1.



The predicate $\mathsf{fls}(X;Y)[W]$ specifies an acyclic free list segment rooted at

X and whose last element refers to Y; the sequence W stores in order the start addresses of the free chunks in the list. In the above picture, three free chunks (in white) are linked and specified by $\mathsf{fls}(X; \mathsf{nil})[W]$.

A free list sorted by the start addresses of free-chunks is specified by $\mathsf{flso}(X, x; Y, y)[W]$, where x and y are less resp. greater than all addresses in the free list. We denote by $\mathbb{P} = \{\mathsf{blk}, \mathsf{chd}, \mathsf{chk}, \mathsf{fck}, \mathsf{hlsc}, \mathsf{hls}, \mathsf{fls}, \mathsf{flso}\}$ the set of predicate symbols in the logic, each with associated arity.

5.2.2 Semantics of SLMA

Formulas of SLMA are interpreted over pairs (I, h) where I is an *interpretation* of logic variables and h is a *heap*. Formally, an interpretation I is a partial function such that nil, location and integer variables are mapped to singleton sequences, and sequence variables are mapped to sequences of values. A heap h is a partial function mapping a location to a non empty sequence of values stored at this location. $\mathbb{V}^+$ denotes a non-empty sequence of values and its length is denoted by $|\mathbb{V}^+|$. A location l is *allocated* in (I, h) if and only if l is in the domain of h. Let $h(\ell)[i]$ denote the $(i+1)$th element of $h(\ell)$ (the sequence of $h(\ell)$ is zero-based indexing).

I	$\in$	$\mathbb{I}$	$\triangleq$	$((\mathsf{AVar} \cup \mathsf{IVar}) \rightharpoonup \mathbb{V}) \cup (\mathsf{SVar} \rightharpoonup \mathbb{V}^*)$	interpretation
h	$\in$	$\mathbb{H}$	$\triangleq$	$\mathbb{A} \rightharpoonup \mathbb{V}^+$	heap

The semantic rules are defined in Table 5.3. A formula φ is satisfied in (I, h) is denoted by $(I, h) \models \varphi$. The set of models satisfying φ is denoted by $[\![\varphi]\!] \triangleq \{(I, h) \mid (I, h) \models \varphi\}$. Given two formulas φ and ψ, we say φ entails ψ, denoted by $\varphi \Rightarrow \psi$ or $\varphi \vdash \psi$, iff $[\![\varphi]\!] \subseteq [\![\psi]\!]$. In SLMA, the separation of two heaps, h_1 and h_2, is denoted by $h_1 \circledast h_2$. It requires both domains and ranges of h_1 and h_2 being disjoint. The union of two heaps, denoted by $h_1 \uplus h_2$, is similar to the definition in 5.1.2.

5.3 Properties

We study here some properties of SLMA that are important to measure its usability for the analysis of SDMA.

Table 5.3: Logic semantics

$[\![x]\!]_I$	$\triangleq$	$I(x)$	$[\![[X]]\!]_I$	$\triangleq$	$[I(X)]$
$[\![\epsilon]\!]_I$	$\triangleq$	$\varnothing$	$[\![[X,Y]]\!]_I$	$\triangleq$	$[I(X), I(Y)]$
$[\![W]\!]_I$	$\triangleq$	$I(W)$	$[\![w_1.w_2]\!]_I$	$\triangleq$	$[\![w_1]\!]_I.[\![w_2]\!]_I$

$I, h \models L_1 - L_2 \# t$	iff	$[\![L_1]\!]_I - [\![L_2]\!]_I \# t$ holds in I
$I, h \models A_1 \wedge A_2$	iff	$I, h \models A_1$ and $I, h \models A_2$
$I, h \models \forall X \in W \cdot A_1 \Rightarrow A_2$	iff	$I(W) = [a_1, \ldots, a_n]$ s.t. $\forall i \in (1..n)\ I[X \mapsto a_i], h \models A_1 \Rightarrow A_2$
$I, h \models W = w$	iff	$[\![W]\!]_I = [\![w]\!]_I$
$I, h \models \Pi_1 \wedge \Pi_2$	iff	$I, h \models \Pi_1$ and $I, h \models \Pi_2$
$I, h \models \Sigma_H \Supset \Sigma_F$	iff	$I, h \models \Sigma_H$ and $\exists h' \subseteq h$ s.t. $I, h' \models \Sigma_F$ $\forall \ell \in \mathsf{dom}(h') \cdot h'(\ell)[\mathtt{isfree}] = 1$
$I, h \models \mathsf{emp}$	iff	$\mathsf{dom}(h) = \varnothing$
$I, h \models \mathsf{blk}(X; Y)$	iff	$\mathsf{dom}(h) = I(X) \ \wedge\ I(Y) - I(X) = \lvert h(I(X)) \rvert$
$I, h \models X \mapsto x$	iff	$\mathsf{dom}(h) = I(X) \ \wedge\ h(I(X))[0] = I(x)$
$I, h \models P(\vec{x})[W]$	iff	$P(\vec{y})[W'] \triangleq \phi$ ($P \in \mathbb{P}$) and $I, h \models \phi[\vec{x}/\vec{y}][W/W']$
$I, h \models \Sigma_1 * \Sigma_2$	iff	$\exists h_1, h_2$ s.t. $h = h_1 \uplus h_2$ and $I, h_i \models \Sigma_i$ for $i = 1, 2$
where		
$h_1 \subseteq h_2$	iff	$\mathsf{dom}(h_1) \subseteq \mathsf{dom}(h_2)$ and $\forall \ell \in \mathsf{dom}(h_1) \cdot h_1(\ell) = h_2(\ell)$
$h_1 \circledast h_2$	iff	$\forall\ l_1 \in \mathsf{dom}(h_1), l_2 \in \mathsf{dom}(h_2) \cdot\ l_1 \neq l_2 \wedge \big((l_1..l_1 + \lvert h_1(l_1) \rvert - 1) \cap (l_2..l_2 + \lvert h_2(l_2) \rvert - 1) = \varnothing\big)$
$h = h_1 \uplus h_2$	iff	$h_1 \circledast h_2$, $\mathsf{dom}(h) = \mathsf{dom}(h_1) \uplus \mathsf{dom}(h_2)$, and $(h_1 \uplus h_2)(\ell) \triangleq \begin{cases} h_1(\ell) & \text{if } \ell \in \mathsf{dom}(h_1) \\ h_2(\ell) & \text{if } \ell \in \mathsf{dom}(h_2) \end{cases}$
$I, h \models \Pi \wedge \Sigma$	iff	$I, h \models \Pi$ and $I, h \models \Sigma$
$I, h \models \varphi_1 \vee \varphi_2$	iff	$I, h \models \varphi_1$ or $I, h \models \varphi_2$
$I, h \models \exists x \cdot \varphi$	iff	$\exists v \in \mathbb{V}$ s.t. $I[x \mapsto v], h \models \varphi$

5.3.1 Expressiveness

Formulas in SLMA are able to capture the complex invariants of SDMA presented in the first part of this document. For example, the invariant specifying the memory's structure of an SDMA in the class **MSA** (i.e., with eager coalescing and sorted acyclic free list) is given by the following SLMA formula:

$$\mathsf{hlsc}(\mathsf{hst}, 0; \mathsf{hli}, 0)[W_H] \Supset \mathsf{flso}(\mathit{fbe}, \mathsf{hst}; \mathsf{nil}, \mathsf{hli})[W_F]$$

where *fbe* is the start of the free list.

The post-condition of the allocation method of a such SDMA with first-fit policy (i.e., class **MSAF**) is specified using SLMA by a formula that includes the following disjunct:

$$\begin{array}{l}\mathsf{hlsc}(\mathsf{hst}, 0; \mathsf{hli}, 0)[W] \land W_H = w \\ \Supset \\ \mathsf{flso}(\mathit{fbe}, \mathsf{hst}; \mathtt{r}, \ell)[W_1] * \mathsf{fck}(\mathtt{r}, \ell') * \mathsf{flso}(\ell', \mathtt{r}; \mathsf{nil}, \ell'')[W_2] \land\ \mathcal{F}_{\mathtt{size}}(\mathtt{r}) \geq s \\ \land \forall X \in W_1 \cdot \mathcal{F}_{\mathtt{size}}(X) < s \land W_F = W_1.[r].W_2\end{array}$$

where s is the size requested for allocation and $\mathtt{r}$ is the address of the first fitting chunk.

5.3.2 Transformation Rules

The predicate definitions in Table 5.2 imply a set of lemmas that will be used in Chapter 6 to transform the abstract values encoded by formulas. The first set of lemmas is obtained by directing predicate definitions in both directions. For example, each definition $P(\ldots) \triangleq \vee_i \varphi_i$ introduces a set of *folding* lemmas $\varphi_i \Rightarrow P(\ldots)$ and an *unfolding* lemma $P(\ldots) \Rightarrow \vee_i \varphi_i$.

The second class of lemmas concerns list segment predicates in Table 5.2. The inductive definitions of these predicates satisfy the syntactic constraints defined in [ESW15] for *compositional predicates*. Thus, every $P \in \{\mathsf{hls}, \mathsf{hlsc}, \mathsf{fls}, \mathsf{flso}\}$ satisfies the following *segment composition lemma*:

Lemma 1. $P(X, \vec{x}; Y, \vec{y})[W_1] * P(Y, \vec{y}; Z, \vec{z})[W_2] \land W = W_1.W_2 \Rightarrow P(X, \vec{x}; Z, \vec{z})[W]$

The reverse implication is applied to split non empty list segments. Finally, the block sub-formulas are removed, split, or folded using the following lemmas:

Lemma 2. $\mathsf{blk}(X;Y) \ \wedge \ X \geq Y \quad \Rightarrow \quad \mathsf{emp}$

Lemma 3. $\mathsf{blk}(X;Y) \ \wedge \ X < Y \quad \Rightarrow \quad \exists Z \cdot \mathsf{blk}(X;Z) \ * \ \mathsf{blk}(Z;Y) \wedge X \leq Z \leq Y$

Lemma 4. $\mathsf{blk}(X;Y) \ * \ \mathsf{blk}(Y';Z) \ \wedge \ X \leq Y = Y' \leq Z \quad \Rightarrow \quad \mathsf{blk}(X;Z).$

Lemma 2 is used to transform incorrect block into empty case. Lemma 4 is the reverse of Lemma 3.

5.3.3 Satisfiability

The satisfiability of formulas in SLMA is important for the static analysis method because it allows to determine if the set of concrete states represented by an abstract value encoded by a formula is empty or not, i.e., if it is equal to the least element of the lattice. Although the satisfiability of the logic underlying the abstract values is not decidable, the static analyses methods employ sound methods to check it, i.e., method that when they respond negatively, the formula is unsatisfiable, otherwise, we do not known its exact status. Moreover, this approximation method may be applied even for logics with a decidable satisfiability problem because it may be faster than the decision procedure.

I conjecture that the satisfiability problem for SLMA is undecidable and I explain this in the remainder of this section by relating SLMA with existing logic fragments. The next chapter contains a sound procedure for checking satisfiability of an abstract value encoded by a formula in SLMA.

5.3.3.1 Satisfiability of Pure Part

The pure part of our logic fragment includes constraints over integer terms and sequence constraints. Integer constraints belong to decidable theories, thus the decidability for satisfiability of the pure part depends on the sequence constraints. The sequence constrains Π_W in SLMA are formulas of some array logic fragment, we denote this fragment by T_W. We wish to check whether the satisfiability of T_W is decidable or not. It is denoted by T_W-satisfiability.

Array logic formalizes mapping from an *index type* to an *element type*. Let a be an array. The basic operations on arrays includes: $a[i]$ is the value of the element at index i of array a (***read***), and $a[i \leftarrow v]$ represents the element at index i is replaced by v (***write***). In our case, a sequence variable represents an array which is indexed by the addresses and the values of it are in $\mathbb{V}$.

Satisfiability of the full array logic is not decidable. Several works show that satisfiability for some restricted quantifier-free fragments of the array logic are decidable [Kin69,SBDL01]. The decision procedures for fragments which permit quantification with some restrictions are presented in [Bra07]. One decidable fragment, denoted by T_A^Z, is with integer indices and considers the formula of the form $\forall \vec{i} \cdot F[\vec{i}] \rightarrow G[\vec{i}]$ where $\vec{i}$ is a list of variables , $F[\vec{i}]$ and $G[\vec{i}]$ are index and value constraints respectively. The indexes are in integer type and constraints over them can be equations and inequalities.

The decision procedure of integer-indexed array T_A^Z given in [Bra07] is to reduce universal quantification to finite conjunctions. To decide the satisfiability of T_A^Z is equal to deciding the satisfiability of $T_A \cup T_Z \cup T$ where T_A represents quantifier-free fragment, T is the element theory and T_Z is the integer index theory (formulas in T_Z with no inequalities).

T_W is a variant of T_A^Z while T_W allows disequalities in index constraints. In T_W, the number of variables is bounded by the quantifier, while in T_A^Z, it is not bounded. The equation of sequences ($w_1 = w_2$), is the same as the array equality ($\forall i \cdot a_1[i] = a_2[i]$). However, T_W has the concatenation operator which makes decidability difficult to deal with. Consider the concatenation of two arrays $a_1.a_2$, this involves the length of the array, i.e., the fixed bound is on the index. Thus, the decision procedure of T_A^Z-satisfiability can not be directly applied to data words T_W.

5.3.3.2 Satisfiability of Spatial Part

The problem of deciding satisfiability of the spatial part $I, h \models \Sigma$ is a challenging work for the inductive predicates which are defined with sequence constrains. Berdine et al. [BCO05a] presented the decision procedure for satisfaction of a fragment Separation Logic. The fragment includes only one kind of inductive predicate: linked list. The idea is that a list is equi-satisfiable with the lists of lengths zero and two.

Brotherston et al. [BFPG14] considered the satisfiability of a fragment of Separation Logic with user-define inductive predicates. After that work, Brotherston et al. [BGKR16] have shown that the model checking of that logic is EXPTIME-complete in the general case.

Le et al. [LTSC17] gave a decision procedure for satisfiability of a fragment of Separation Logic including inductive predicates with shape and arithmetic properties. It requires that the formula derived from inductive predicates could be stretched. That means the arithmetic part of the predicate is a conjunction of periodic constraints and closure of the union of these conjunctions can be represented by some semilinear sets, i.e., sets definable in Presburger arithmetic. Our spatial part has inductive predicates with formulas in fragment of array logic T_W. Thus, one solution is to check the sequence constraints in SLMA obey the restrictions in [LTSC17] or not.

5.3.4 Entailment Checking

The problem of validity of an entailment between formulas in SLMA is important for the static analysis method because it allows to order abstract values represented by SLMA formulas. Therefore, an abstract value a subsumes (is bigger than in the lattice of abstract values) a' if the entailment $a' \Rightarrow a$ is valid. Like for satisfiability, the validity of entailment may be dealt by sound methods when the problem is not decidable or the decision procedure has high complexity. If the sound method responds "yes" then the entailment is valid, otherwise nothing can be said (the entailment may be valid or invalid).

For the logic SLMA, I conjecture that the problem of checking validity of entailment is undecidable and I compare this result with the existing results for logics derived from Separation Logic. The next chapter contains a sound procedure for checking entailment between abstract values encoded by a formula in SLMA.

5.3.4.1 Entailment of Pure Part

Like for satisfiability, the array property fragment T_A^Z of array logic is the starting point of our study. Bradley et al. [Bra07] have shown that the entailment checking is decidable in T_A^Z because the validity of entailment can be translated into a satisfiability checking, as usual. However, the pure part of our logic combines the array property fragment over the content of sequences with sequences constraints which are outside the fragment. The conjecture is that the entailment checking of pure parts is not decidable for SLMA.

5.3.4.2 Entailment of Spatial Part

The entailment problem for the symbolic heap fragment of Separation Logic with only list segment inductive definition has been shown decidable in [BCO05a, CYO01,CHO+11,NPR11]. In particular, Cook et al. [CHO+11] prove that the satisfiability/entailment problem can be solved in polynomial time. Piskac et al. [PWZ13] show that the Boolean closure of this fragment can be translated to a decidable fragment of first-order logic, and in this way they prove that the satisfiability/entailment problem can be decided in **NP**/**co-NP**. Furthermore, they consider the problem of combining SL formulas with constraints on data using the Nelson-Oppen theory combination framework [NO79]. Adding constraints on data to SL formulas is considered also in Qiu et al. [QGŞM13]. The fragment of overlaid nested lists defined in [ESS13] has also a decidable entailment problem. Our operator $\Supset$ is less powerful than the one proposed in the fragment but our fragment includes data constraints, which is outside the scope of the fragment in [ESS13].

Iosif et al. [IRS13] also introduce a decidable fragment of SL with inductive definitions that can describe more complex data structures than the fragment presented in this work, including, e.g., trees with parent pointers or trees with linked leaves. The mentioned work reduces the entailment problem to Monadic Second Order Logic on graphs with a bounded tree width, resulting in a multiple-exponential complexity. They do not deal with data constraints.

In conclusion, our logic fragment combines in the spatial part inductive predicates with data constraints that are out of the scope of the existing procedures. I conjecture that the problem is undecidable for our fragment and focus on sound procedure for checking entailment based on graph homeomorphism like in [ESS13].

CHAPTER 6

Logic-based Abstract Domain

In this chapter, we present the abstract domain designed for the static analysis of SDMA. Our static analyzer is built based on the *abstract interpretation* [CC77b,CC79] framework which is used for constructing sound approximations for semantics of the programs. The *concrete semantics* precisely describes the executions of the programs. But the concrete semantics is not computable in general. The concrete semantics can be approximated by some *abstract semantics* on which we can reason on the properties we are interested in. The *sound approximations* represent the correspondence between concrete and abstract semantics and guarantee that properties proved within the abstract semantics hold in the corresponding concrete executions of the programs.

The properties inferred by our analyzer are specified using subformulas of the logic SLMA introduced in Chapter 5. The abstract domain is a cofibered product of an extended symbolic heap domain and existing domains, i.e., numerical domains [CH78,Min01b] and data words domain [BDE^{+}10]. The partial order between abstract values is defined by using an entailment between formulas.

This chapter is structured as follows. In § 6.1, we recall the principles of abstract interpretation. We also present existing abstract domains which are used in our domain and we explain the techniques used to combine multiple domains. § 6.3 defines our abstract domain we designed for analyzing SDMA. The domain is defined in a stepwise way. The operators on the abstract domain are presented in § 6.3.7.

6.1 Abstract Interpretation Preliminaries

We present a simple imperative program languages with fixed data types in Section 6.1.1 and give its syntax and concrete semantics. Then we recall in Section 6.1.2 the general definitions of abstract interpretation theory and explain how it computes approximations of concrete semantics of this language. Section 6.1.3 introduces some existing abstract domains and the techniques for combining different domains.

6.1.1 A Simple Imperative Language

We consider the simple imperative program and its syntax presented in Figure 6.1. It has no procedures and it could be seen as a subset of the C programming language. This language is used only for describing the application of abstract interpretation and it will be extended in the next section.

$prog$	$::=$	$stmt$	program
$stmt$	$::=$	$loc := exp$	assignment
	$\mid$	`skip`	skip
	$\mid$	**`if`** exp **`then`** $stmt$ **`else`** $stmt$	condition
	$\mid$	**`while`** exp **`do`** $stmt$ **`done`**	loop
	$\mid$	$stmt; stmt$	sequence
loc	$::=$	x	variables ($x \in$ PVar)
exp	$::=$	i	value $i \in \mathbb{Z}$
	$\mid$	loc	read location
	$\mid$	$exp \oplus exp$	arithmetic
$\oplus$	$::=$	$+ \mid - \mid \times \mid /$	arithmetic operators

Figure 6.1: Syntax of a simple imperative language

6.1.1.1 Syntax

Recall that PVar denotes the set of program variables in a program. The variables have only one type: machine integers **`int`**. We denote by $\mathbb{Z}$ the set of all integers. A program $prog$ in this language is a sequence of *statements*. A statement $stmt$ could be an assignment, a skip instruction, a condition branching or a loop. The left-value loc of an assignment evaluates to a memory location given by a program variable. An expression exp could be a constant, a left-value or an arithmetic expression. This language does not support Boolean type. In the condition branching, the expression is only compared with zero, thus, non-zero integers represent **`true`** ($exp \neq 0$) and zero represents **`false`** ($exp = 0$) respectively.

evaluation of locations

$$\begin{array}{rcl}
\mathbf{EvalL}[\![loc]\!] & : & \mathbb{S} \rightarrow \mathsf{Pvar} \\
\mathbf{EvalL}[\![x]\!](\sigma) & \triangleq & x
\end{array}$$

evaluation of expressions

$$\begin{array}{rcl}
\mathbf{EvalE}[\![exp]\!] & : & \mathbb{S} \rightarrow \mathbb{Z} \\
\mathbf{EvalE}[\![v]\!](\sigma) & \triangleq & v \\
\mathbf{EvalE}[\![x]\!](\sigma) & \triangleq & \sigma(x) \\
\mathbf{EvalE}[\![e_1 \oplus e_2]\!](\sigma) & \triangleq & \mathbf{EvalE}[\![e_1]\!](\sigma) \oplus \mathbf{EvalE}[\![e_2]\!](\sigma)
\end{array}$$

condition tests

$$\begin{array}{rcl}
\mathbf{Guard}[\![.]\!] & : & \mathcal{P}(\mathbb{S}) \rightarrow \mathcal{P}(\mathbb{S}) \\
\mathbf{Guard}[\![exp]\!](S) & \triangleq & \{\sigma \in S \mid \mathbf{EvalE}[\![exp]\!]\sigma \neq 0\} \\
\mathbf{Guard}[\![!exp]\!](S) & \triangleq & \{\sigma \in S \mid \mathbf{EvalE}[\![exp]\!]\sigma = 0\}
\end{array}$$

concrete transformers

$$\mathbf{Post}[\![.]\!] \quad : \quad \mathcal{P}(\mathbb{S}) \rightarrow \mathcal{P}(\mathbb{S})$$

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathbf{Post}[\![stmt]\!](S) \triangleq \text{match } stmt \text{ with:}} \\
\mid \texttt{skip} & \rightarrow & S \\
\mid loc := exp & \rightarrow & \{\sigma[\mathbf{EvalL}[\![loc]\!](\sigma) \leftarrow \mathbf{EvalE}[\![exp]\!](\sigma)] \mid \sigma \in S\} \\
\mid s_1; s_2 & \rightarrow & \mathbf{Post}[\![s_2]\!](\mathbf{Post}[\![s_1]\!](S)) \\
\mid \mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 & \rightarrow & \mathbf{Post}[\![s_1]\!](\mathbf{Guard}[\![e]\!](S)) \cup \mathbf{Post}[\![s_2]\!](\mathbf{Guard}[\![!e]\!](S)) \\
\mid \mathbf{while}\ e\ \mathbf{do}\ s\ \mathbf{done} & \rightarrow & \mathbf{Guard}[\![!e]\!](\mathbf{lfp}^{\subseteq}(\lambda S_i.S \cup \mathbf{Post}[\![s]\!](\mathbf{Guard}[\![e]\!](S_i))))
\end{array}$$

Figure 6.2: Concrete semantics of a simple imperative language

6.1.1.2 Concrete Semantics

The concrete semantics of the simple imperative language is defined in Figure 6.2. The program states are evaluations of variables and updated by statements. A *concrete state* of a program is defined as a store mapping program variables to their values ($\sigma \in \mathbb{S} \triangleq \mathsf{PVar} \rightarrow \mathbb{Z}$). The evaluation of a location, denoted by $\mathbf{EvalL}[\![loc]\!] : \mathbb{S} \rightarrow \mathsf{Pvar}$, maps a location expression to a memory location which is represented by a program variable. The evaluation of an expression is $\mathbf{EvalE}[\![exp]\!] : \mathbb{S} \rightarrow \mathbb{Z}$ which maps an expression to a value. The condition test $\textbf{Guard}[\![exp]\!] : \mathcal{P}(\mathbb{S}) \to \mathcal{P}(\mathbb{S})$ filters out the concrete states that do not satisfy the condition expressed by exp. The concrete transformers of the programs stated by statements is denoted by $\textbf{Post}[\![stmt]\!] : \mathcal{P}(\mathbb{S}) \to \mathcal{P}(\mathbb{S})$.

Fixpoint Computation: The transition of a loop statement is represented by a function $\boldsymbol{F} : \mathcal{P}(\mathbb{S}) \to \mathcal{P}(\mathbb{S})$ which is monotonic and continuous. The concrete states collected by transformer $\textbf{Post}[\![\texttt{while}\ e\ \texttt{do}\ s\ \texttt{done}]\!](S)$ is theleast fix-points of $\boldsymbol{F}$ where $\boldsymbol{F}(S_i) = S \cup \textbf{Post}[\![s]\!](\textbf{Guard}[\![e]\!](S_i))$. Moreover, the fixpont gives the loop invariant. To get the semantics of the loop, the fixpoint must be filtered by the loop exit condition $!e$.

We denote by $\textbf{lfp}^{\subseteq}\boldsymbol{F}$ the set of least fix-points of $\boldsymbol{F}$ with respect to $\subseteq$. Tarski theorem [Tar55] shows the existence of the fix-points of $\boldsymbol{F}$ while Kleene theorem [KdBdGZ52] gives a way to compute the fix-points. Given the transfinite induction:

$$\begin{cases} \boldsymbol{F}^0(X) \triangleq X \\ \boldsymbol{F}^{n+1}(X) \triangleq \boldsymbol{F}(\boldsymbol{F}^n(X)), \end{cases}$$

where $\boldsymbol{F}^i(X)$ denotes by $\boldsymbol{F}'s$ iteration on X, we have $\boldsymbol{F}^b(X) \triangleq \bigcup_{a<b} \boldsymbol{F}^a(X)$ where b is a limit ordinal.

Theorem 6.1 (Kleene fixpoint theorem)**.** *Given a partial ordered set* $(\mathcal{P}(\mathbb{S}), \subseteq)$ *and a monotonic and continuous function* $\boldsymbol{F} : \mathcal{P}(\mathbb{S}) \to \mathcal{P}(\mathbb{S})$*, for any* $X \in \mathcal{P}(\mathbb{S})$*, then*

$$\textbf{lfp}^{\subseteq}\boldsymbol{F} = \bigcup\{\boldsymbol{F}^n(\varnothing) | n \in \mathbb{N}\}$$

■

In general, the concrete semantics defined above is infinite and not computable. In the following section, we describe an example design an abstraction of the concrete semantic in the abstract interpretation framework.

6.1.2 Abstract Interpretation

The abstract interpretation framework [CC77b,CC79] provides a way to build a correspondence between two semantics of programs. The set of concrete states of the program forms a partially ordered set $(\mathcal{P}(\mathbb{S})), \subseteq)$, i.e., the *concrete domain* (*To simply the description, we assume that the concrete domain is the power-set of concrete states and the partial order between states is the subset relation in this section.*

It is not always the case). Each program point's information is carried by the element in $\mathcal{P}(\mathbb{S})$. If $s \subseteq c$, then c carries more information than s. By using the abstract interpretation framework, verifiers can focus on their own abstract properties instead of encoding program properties uniformly by using means which have restrictions. The properties that verifiers want to express on the program states are specified as elements of some *abstract domain* $(D^\sharp, \sqsubseteq)$. The abstract domain is designed with a set of *abstract operators* manipulating the properties, i.e., *abstract transformers*.

6.1.2.1 Abstract Domain

Let $C = (\mathcal{P}(\mathbb{S}), \subseteq, \cup, \cap, \varnothing, \mathbb{S})$ be the lattice of concrete program states representing the concrete domain. The *least* and *greatest* concrete elements are the empty set $\varnothing$ and the set of all program states $\mathbb{S}$. And let $A = (D^\sharp, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ be the abstract domain that approximates the concrete domain. The least and greatest elements in the abstract domain are called *bottom* and *top*, represented by $\bot$ and $\top$ respectively. We build the correspondence between the concrete domain and the abstract domain by defining the necessary functions: *abstraction* and *concretization*.

Abstraction function: An abstraction is a mathematical relationship between two semantics (domains). The *soundness* of the abstraction states that any program property proved to hold in the abstract domain also holds in the concrete one. Generally, however, not all properties provable in the concrete semantics can be proved in the abstract semantics, which is *incompleteness* and causes false alarms. Give the concrete domain C and the abstract domain A, an abstraction function converts concrete elements into abstract ones: $\alpha : \mathcal{P}(\mathbb{S}) \to D^\sharp$ and $\alpha(S)$ is a sound approximation of the concrete set S. Note that the abstraction function is monotonic, that is:

$$\forall S_1, S_2 \in \mathcal{P}(\mathbb{S}) \cdot S_1 \subseteq S_2 \Rightarrow \alpha(S_1) \sqsubseteq \alpha(S_2)$$

Concretization function: Similarly, we can also consider a reverse function of the abstraction function, which is called concretization function. A concretization function $\gamma : D^\sharp \to \mathcal{P}(\mathbb{S})$ converts abstract elements into concrete ones. The concretization function is also monotonic, i.e., it maps more approximate abstract elements to more approximate concrete elements. Its monotonicity is formalized as follows:

$$\forall d_1^\sharp, d_2^\sharp \in D^\sharp \cdot d_1^\sharp \sqsubseteq d_2^\sharp \Rightarrow \gamma(d_1^\sharp) \subseteq \gamma(d_2^\sharp)$$

Two special elements $\perp$ and $\top$ in the abstract domain, are mapped to an empty set ($\gamma(\perp) = \varnothing$) and the set of all program states ($\gamma(\top) = \mathbb{S}$) respectively.

Galois connection: The abstraction and the concretization functions are "inverse" of one another and form a *Galois connection* [CC79] between concrete and abstract domains. The Galois connection is a pair of monotonic functions ($\alpha : \mathcal{P}(\mathbb{S}) \to D^\sharp, \gamma : D^\sharp \to \mathcal{P}(\mathbb{S})$) between C and A such that:

$$\forall S \in \mathcal{P}(\mathbb{S}), d^\sharp \in D^\sharp \cdot \alpha(S) \sqsubseteq d^\sharp \Leftrightarrow S \subseteq \gamma(d^\sharp)$$

We denote by $(\mathcal{P}(\mathbb{S}), \langle\alpha, \gamma\rangle, D^\sharp)$ the Galois connection linking domain C and domain A. The existence of a Galois connection ensures that any concrete property $S \in \mathcal{P}(\mathbb{S})$ has a *best abstraction* in $D^\sharp$ and this best abstraction is given by α

Example 3 (The interval abstract domain)**.** The interval abstraction consists in inferring, for each variable, an upper and a lower bound on its possible values. It was introduced early by Cousot and Cousot [CC76] and it is used to specify properties like the absence of arithmetic overflow or out-of-bound array access. The abstract element in the interval domain $(D_I^\sharp, \sqsubseteq_I, \sqcup_I, \sqcap_I, \perp_I, \top_I)$ is defined as follows (where I is the set of integers):

$$\begin{aligned} D_i^\sharp &\triangleq (\mathsf{Pvar} \to \mathsf{I}\,) \cup \{\perp_I\} \\ &\text{where } \mathsf{I} \triangleq \{[a, b] \mid a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{+\infty\}, a \leq b\} \end{aligned}$$

Its abstraction and concretization functions are defined as:

$$\gamma_I(d^\sharp) \triangleq \begin{cases} \varnothing & \text{if } d^\sharp = \perp_I \\ \{\sigma \in \mathbb{S} \mid \forall X.\sigma(X) \in d^\sharp(X)\} & \text{if } d^\sharp \neq \perp_I \;\; (X \in \mathsf{Pvar}) \end{cases}$$

$$\alpha_I(S) \triangleq \begin{cases} \perp_I & \text{if } S = \varnothing \\ \lambda X.[\mathbf{min}\{\sigma(X) | \sigma \in S\}, \mathbf{max}\{\sigma(X) | \sigma \in S\}] & \text{is } S \neq \varnothing \end{cases}$$

△

6.1.2.2 Abstract Operators

The abstract operators include the lattice operators (i.e., abstract join $\sqcup$, abstract inclusion checking $\sqsubseteq$) and the abstract transformers which over-approximate the computation of program statements.

Lattice operations: In the abstract domain $D^\sharp$, the abstract join $\sqcup : D^\sharp \times D^\sharp \to D^\sharp$ computes a least upper bound of two abstract elements and over-approximates the concrete union $\cup$ of the concrete domain. The soundness of the abstract join is defined as follows:

$$\forall d_1^\sharp, d_2^\sharp \in D^\sharp \cdot \gamma(d_1^\sharp) \cup \gamma(d_2^\sharp) \subseteq \gamma(d_1^\sharp \sqcup d_2^\sharp)$$

Example 4 (Abstract join in the interval domain)**.** The abstract join $\sqcup_I$ in the interval domain is defined as follows:

$$d_1^\sharp \sqcup_I d_2^\sharp \triangleq \begin{cases} d_1^\sharp & \text{if } d_2^\sharp = \bot_I \\ d_2^\sharp & \text{if } d_1^\sharp = \bot_I \\ \lambda X.[\mathbf{min}(d_1^\sharp(X), d_2^\sharp(X)), \mathbf{max}(d_1^\sharp(X), d_2^\sharp(X))] & \text{otherwise} \end{cases}$$

It selects the minimal left bound and the maximal right bound between two intervals and generates a new interval, e.g., $[1,2] \sqcup_I [3,4]$ is equal to $[1,4]$. $\triangle$

The abstract inclusion checking operator $\sqsubseteq$ checks the order between abstract elements and it over-approximates the concrete inclusion checking operator $\subseteq$. The soundness of $\sqsubseteq$ is defined as follows:

$$\forall d_1^\sharp, d_2^\sharp \in D^\sharp \cdot d_1^\sharp \sqsubseteq d_2^\sharp \Rightarrow \gamma(d_1^\sharp) \subseteq \gamma(d_2^\sharp)$$

Example 5 (Inclusion checking in the interval domain)**.** The inclusion checking $\sqsubseteq_I$ in the interval domain is defined as follows:

$$d_1^\sharp \sqsubseteq_I d_2^\sharp \Leftrightarrow \begin{cases} d_1^\sharp = \bot_I & \text{or} \\ \forall X.d_1^\sharp(X) \subseteq d_2^\sharp(X)) & \text{if } d_1^\sharp, d_2^\sharp \neq \bot_I \end{cases}$$

$\triangle$

Abstract transformers: The *abstract transformers* over-approximate the concrete program semantics, i.e., condition tests and statements. Each program statement in the concrete domain has a corresponding abstract transformer in the abstract domain. Given the concrete domain C and the abstract domain A, which are linked by a Galois connection $(\mathcal{P}(\mathbb{S}), \langle\alpha, \gamma\rangle, D^{\sharp})$. Let $F : \mathcal{P}(\mathbb{S}) \to \mathcal{P}(\mathbb{S})$ be a concrete transformer function on $\mathcal{P}(\mathbb{S})$. The corresponding abstract transformer of F is $F^{\sharp} : D^{\sharp} \to D^{\sharp}$ satisfying the following condition for soundness:

$$\forall d^{\sharp} \in D^{\sharp} \cdot (F \circ \gamma)(d^{\sharp}) \subseteq (\gamma \circ F^{\sharp})(d^{\sharp})$$

It means that the result of a computation step performed in the abstract domain represents an over-approximation of the corresponding computation step performed in the concrete domain. The above condition can be replaced by the following condition using the abstraction function:

$$\forall S \in \mathcal{P}(\mathbb{S}) \cdot (\alpha \circ F)(S) \sqsubseteq (F^{\sharp} \circ \alpha)(S)$$

For the concrete condition test $\mathbf{Guard}[\![exp]\!]$ shown in Figure 6.2, its corresponding abstract condition test is $\mathbf{Guard}^{\sharp}[\![exp]\!] : D^{\sharp} \to D^{\sharp}$. It over-approximates $\mathbf{Guard}[\![exp]\!]$. The soundness of the abstract condition test is defined as follows:

$$\forall d^{\sharp} \in D^{\sharp} \cdot \mathbf{Guard}[\![exp]\!](\gamma(d^{\sharp})) \subseteq \gamma(\mathbf{Guard}^{\sharp}[\![exp]\!](d^{\sharp}))$$

Another abstract transformer $\mathbf{Assign}^{\sharp}[\![loc := exp]\!] : D^{\sharp} \to D^{\sharp}$ is defined for assignment $\mathbf{Post}[\![loc := exp]\!]$ in the concrete domain. The soundness of it is defined as follows:

$$\forall d^{\sharp} \in D^{\sharp} \cdot \mathbf{Post}[\![loc := exp]\!](\gamma(d^{\sharp})) \subseteq \gamma(\mathbf{Assign}^{\sharp}[\![loc := exp]\!](d^{\sharp}))$$

Abstract fix-points: In the concrete domain, the concrete transformer of the loop is denoted by function $\boldsymbol{F}$. We cannot ensure that the loop is always terminating because the sequence $(\boldsymbol{F}^n(\varnothing))_{n\geq 0}$ may not eventually stabilize. We assume the corresponding abstract transformer of $\boldsymbol{F}$ is $\boldsymbol{F}^{\sharp} : D^{\sharp} \to D^{\sharp}$. Tarski theorem [Tar55] ensures that the least fix-point of the abstract transformer $\boldsymbol{F}^{\sharp}$ is an over-approximation (an abstraction) of the least fix-point of the corresponding concrete transformer $\boldsymbol{F}$. In the abstract domain, the termination for the computation of abstract fix-points should be guaranteed. Also, if the computation is too slow to terminate, the analysis should accelerate it.

To accelerate the convergence by an extrapolation operator in a loop, we usually design a *widening operator* $\nabla : D^\sharp \times D^\sharp \to D^\sharp$ [CC76,CC77a,CC79,CH78]. When computing the abstract fix-points, the abstract join is usually replaced by a widening.

Definition 6.1 (Widening operator)**.** Given the abstract domain $(D^\sharp, \sqsubseteq)$, its widening operator is a function $\nabla : D^\sharp \times D^\sharp \to D^\sharp$ such that

- $\forall d_1^\sharp, d_2^\sharp, \in D^\sharp \cdot d_1^\sharp \sqsubseteq (d_1^\sharp \nabla d_2^\sharp)$ and $d_2^\sharp \sqsubseteq (d_1^\sharp \nabla d_2^\sharp)$,
- for any increasing chains $(d_0^\sharp \sqsubseteq^\sharp d_1^\sharp \sqsubseteq^\sharp ... \sqsubseteq^\sharp d_n^\sharp \sqsubseteq^\sharp ...)$ from $D^\sharp$, the new increasing chain defined as:

  $$e_1^\sharp \quad \triangleq \quad d_0^\sharp, \quad e_{n+1}^\sharp \quad \triangleq \quad e_n^\sharp \nabla d_{n+1}^\sharp.$$

  stabilizes in a finite number of iterations which means there exists $m \geq 0$ such that $e_{m+1}^\sharp = e_m^\sharp$.

■

Example 6 (Widening in the interval domain)**.** The widening operator in the interval domain ensures termination by replacing unstable lower bounds with $-\infty$ and upper bounds with $+\infty$, so that intervals cannot grow indefinitely. It is defined as follows:

$$d_1^\sharp \nabla_I d_2^\sharp \triangleq \begin{cases} d_1^\sharp & \text{if } d_2^\sharp = \bot_I \\ d_2^\sharp & \text{if } d_1^\sharp = \bot_I \\ \lambda X. d_1^\sharp(X) \nabla_I d_2^\sharp(X) & \text{otherwise} \end{cases}$$

$$\text{where } [a,b] \nabla_I [c,d] \triangleq \left[\begin{cases} a & \text{if } a \leq c \\ -\infty & \text{otherwise} \end{cases}, \begin{cases} b & \text{if } b \geq d \\ +\infty & \text{otherwise} \end{cases} \right]$$

△

6.1.3 Existing Abstract Domains

Numeric abstract domains: To compute different numerical properties of numerical program variables like integers and reals, many numerical abstract domains have been proposed. [CC77b] introduces *signs abstract domain* and *intervals domain* that can infer variables sign and bounds respectively. [Min01a] presents a DBM numerical domain that allows to represent invariants of the form $(x - y \leq c)$ and $(\pm x \leq c)$, where x and y are variables and c is an integer or real constant. Another one, the *octagons abstract domain* introduced in [Min01b] can describe invariants of the form $\pm v_1 \pm v_2 \leqslant c$ (v_1 and v_2 are numerical variables and c is a numeric constant). Intervals and octagons are two restrictions of *polyhedra abstract domain* [CH78] that manipulates systems of linear inequalities between program variables $\Sigma_{i=1}^{n}(a_i \cdot x_i \leq n)$. It can express complex invariants at the cost of a very high complexity. Each numerical domain focuses on one kind of properties and has its own expressiveness and computational complexity. The open library Apron [JM09] provides implementations of existing numerical abstract domains, and it gives a uniform, rich and domain-independent API.

Combination of abstract domains: To infer more precise properties of the program, the abstract interpretation framework supplies several ways to design a complex abstract domain by combining some simpler abstract domains. [CCF13] gives a survey of existing product operators on abstract domains. The conjunction of properties expressed in different abstract domains are performed by the *reduced product* of domains [CC79]. [Ven96] introduce cofibered domains, and [CLCVH00] gives open product framework. In this paper, we use cofibered-product to combine domains.

Theorem 6.2 (Reduced product)**.** *Given two Galois connections* $(\mathcal{P}(\mathbb{S}), \langle \alpha_1, \gamma_1 \rangle, D_1^\sharp)$, *and* $(\mathcal{P}(\mathbb{S}), \langle \alpha_2, \gamma_2 \rangle, D_2^\sharp)$, *the reduction operator is defined as* $\sigma : (D_1^\sharp \times D_2^\sharp) \rightarrow (D_1^\sharp \times D_2^\sharp)$. *The reduced product of* $D_1^\sharp$ *and* $D_2^\sharp$ *is a complete lattice:* $(\sigma(D_1^\sharp \times D_2^\sharp), \sqsubseteq_1 \times \sqsubseteq_2, \sqcup_1 \times \sqcup_2, \sqcap_1 \times \sqcap_2, \bot_1 \times \bot_2, \top_1 \times \top_2)$. *And* $(\mathcal{P}(\mathbb{S}), \langle \sigma \circ \alpha, \gamma \rangle, \sigma(D_1^\sharp \times D_2^\sharp))$ *is a Galois connection of* $\mathcal{P}(\mathbb{S})$ *and* $\sigma(D_1^\sharp \times D_2^\sharp)$. ■

Definition 6.2 (Cofibered domain)**.** Given the concrete domain $(\mathcal{P}(\mathbb{S}), \subseteq)$ and its abstractions $(D^\sharp, \sqsubseteq_D)$ and $(E^\sharp, \sqsubseteq_E)$. The function $R : D^\sharp \rightarrow E^\sharp$ maps elements of $D^\sharp$ to elements of $E^\sharp$. Each element in $d_i^\sharp \in D^\sharp$ is the property over a set of variables, denoted by X. And $R(d_i^\sharp)$ is the property over a set of variables Y which includes X, i.e., $X \subseteq Y$. The element of the cofibered domain $(F^\sharp, \sqsubseteq_F)$ is a pair $(d_i^\sharp \mapsto e_i^\sharp)$ where $d_i^\sharp \in D^\sharp$ and $e_i^\sharp \in E^\sharp$. For any $(d_i^\sharp \mapsto e_i^\sharp)$, $(d_j^\sharp \mapsto e_j^\sharp) \in F^\sharp$, $(d_i^\sharp \mapsto e_i^\sharp) \sqsubseteq_F (d_j^\sharp \mapsto e_j^\sharp)$ if there is a morphism $f : d_i^\sharp \rightarrow d_j^\sharp$ such that $f(e_i^\sharp) \sqsubseteq_E e_j^\sharp$. ■

6.2 Programming Language

The programs coding the SDMA is a typed imperative programs manipulating memory regions. It is the extended version of the simple programming language described in Section 6.1. The programs include structure types and integers, pointers and pointer arithmetics, pointer casting, bit-wise operations, standard system routines for data segment manipulation (`sbrk`). String manipulation, arrays, pointers to functions, float arithmetics, concurrency constructs, are not present. Moreover, recursive functions are not used. We formalise this observation by fixing in Figure 6.3 an imperative programming language which captures the features used in the C code analysed.

6.2.1 Programming language syntax

$$
\begin{array}{rcll}
prog & ::= & stmt & \text{program} \\
stmt & ::= & cmd & \text{command} \\
 & \mid & \texttt{skip} & \text{skip} \\
 & \mid & \textbf{if}\ b\ \textbf{then}\ stmt\ \textbf{else}\ stmt & \text{condition} \\
 & \mid & \textbf{while}\ b\ \textbf{do}\ stmt\ \textbf{done} & \text{loop} \\
 & \mid & stmt; stmt & \text{sequence} \\
cmd & ::= & loc :=_t exp & \text{assignment} \\
 & \mid & loc :=_t \texttt{sbrk}(exp) & \text{system assignment} \\
loc & ::= & x & \text{variable} \\
 & \mid & loc.f & \text{field access} \\
 & \mid & *exp & \\
exp & ::= & loc & \\
 & \mid & \&loc & \text{reference} \\
 & \mid & v & \text{value} \\
 & \mid & exp \oplus exp & \text{arithmetic} \\
b & ::= & \textbf{true} \mid \textbf{false} \mid \neg b \mid b_1 \vee b_2 \mid b_1 \wedge b_2 & \text{boolean} \\
 & \mid & exp_1 \ltimes exp_2 & \\
\oplus & ::= & + \mid - \mid \times \mid / \mid \& \mid \sim \mid | \mid \ll \mid \gg & \text{binary operators} \\
\ltimes & ::= & < \mid \leq \mid == & \text{comparisons}
\end{array}
$$

Figure 6.3: Program syntax

A program *prog* in this language is a statement. A statement *stmt* could be a command cmd, a do-nothing statement `skip`, sequencing ...; ..., a condition branching **`if`**...**`then`**...**`else`**... or loop **`while`**...**`do`**...**`done`**. A command is either an assignment or a *system assignment*, i.e., data segment extension.

An assignment is specified by a location expression loc that names a memory address to update and an expression exp that is evaluated to yield the new content. We use $:=_t$ to replace the normal assignment operator. The type t gives the typing information of the content written at location loc and it is mainly used to encode C casting. Data segment extension is done using the system method `sbrk`; moreover, it returns the address (of type **`void`**$\star$) after the last byte included in the extended data segment. Notice that this address is page aligned, i.e., multiple of 8. Note that pages are generally much larger than 8 bytes (generally 4KB or 1MB). 8 bytes means only that it is aligned on memory bus width.

Types are either structure types, pointer types or basic types (left unspecified). The set of types is denoted by $\mathbb{T}$. Structure types are defined by a list of fields. The first field in the list determines the alignment constraints for the addresses at which the values of this type may be stored. Fields are treated as numerical offsets, so C field access loc`.`f is the address $a + f$ where a is the address denoted by location expression loc. A typing function τ maps fields and variables to their declaration type; it is extended naturally to location expressions.

Operators include binary arithmetic, bitwise (and, or, shift), and comparison operations. The bitwise operations are used to encode masking or extraction of least/most significant bits. The absence of recursive procedure calls allows us to consider intra-procedural analysis only and use inlining for procedure calls. However, our analysis may be extended to an inter-procedural analysis using, e.g., [BDES11].

6.2.2 Concrete Memory States

We consider a model of program configurations, formally defined in Figure 6.4, that abstracts some low level details of program states as follows.

The absence of recursive functions in the SDMA code allows us to model the program stack by a mapping ϵ, called *concrete environment*, that associates the program variable to the unique address where its value is stored. The set of all concrete environment mappings is denoted by $\mathbb{E}$.

The memory is modelled by a mapping h, called *heap*, which partially maps address of the memory to a value or sequence of values $\mathbb{V}^+$, e.g., the start address of a structure typed memory region is mapped to a sequences of values. The set of all heap mappings is denoted by $\mathbb{H}$. A concrete memory state m is a pair (ϵ, h). The set of memory states $\mathbb{M} \triangleq \mathbb{E} \times \mathbb{H}$ is the product of the set of concrete environments and the set of concrete heaps.

values	$v \in \mathbb{V}$	addresses	$\alpha \in \mathbb{A}$, $\mathbb{A} \subseteq \mathbb{V}$
fields	$f \in \mathbb{F}$	program variables	$x \in \mathsf{Pvar}$
types	$t \in \mathbb{T}$	typing	$\tau : \mathsf{Pvar} \cup \mathbb{F} \to \mathbb{T}$
memories	$m \in \mathbb{M} \triangleq \mathbb{E} \times \mathbb{H}$		
environments	$\epsilon \in \mathbb{E} \triangleq \mathsf{Pvar} \to \mathbb{A}$	heaps	$h \in \mathbb{H} \triangleq \mathbb{A} \rightharpoonup \mathbb{V}^+$

Figure 6.4: Memory model

In the following, we suppose that the domain $\mathbb{V}$ of values stored in the memory is $\mathbb{N}$, the natural numbers, equipped with the usual comparison, arithmetic, and bitwise operations. For the sub-domain of addresses, $\mathbb{A}$, the addition is done only between an address and a non address value. Also, the stored values are typed using types in $\mathbb{T}$ that represent either subsets of naturals (e.g., `size_t`), addresses (e.g., `HDR*`), or tuples of such types (e.g., `HDR`). The type `HDR` is predefined and its elements are labeled by fields, that belong to a finite set $\mathbb{F}$, and are typed using the typing function τ. Without loss of generality, we consider that `HDR` contains at least the three fields `size`, `fnx`, and `isfree`.

6.2.3 Concrete Program Semantics

We define the denotational semantics for this language shown in Figure 6.5.

We denote by $h[\alpha \leftarrow v]$ the heap obtained by the update of h at address α with value v. The evaluations of locations and expressions are denoted by $\mathcal{L}[\![\cdot]\!]$ and $\mathcal{E}[\![\cdot]\!]$ respectively. The function $\mathcal{L}[\![\cdot]\!]$ (resp. $\mathcal{E}[\![\cdot]\!]$) maps a location expression $loc \in \mathbb{Loc}$ (resp. expression $exp \in \mathbb{Exp}$) in a context of a given concrete memory state to an address (resp. value). The evaluation of the address operator $\mathcal{E}[\![\&loc]\!]$ is obtained by evaluation of location loc, i.e. $\mathcal{L}[\![loc]\!]$. The evaluation of field $loc.f$ is to evaluate location loc then do address computation. And the evaluation of deference $\mathcal{L}[\![\star exp]\!]$ is obtained by evaluation of expression exp, i.e. $\mathcal{E}[\![exp]\!]$.

Evaluation of locations $\mathcal{L}[\![.]\!] : \mathbb{M} \to \mathbb{A}$

$$\begin{array}{rcl}
\mathcal{L}[\![x]\!](\epsilon, h) & \triangleq & \epsilon(x) \\
\mathcal{L}[\![\star exp]\!](\epsilon, h) & \triangleq & \mathcal{E}[\![exp]\!](\epsilon, h) \\
\mathcal{L}[\![loc.f]\!](\epsilon, h) & \triangleq & \mathcal{L}[\![loc]\!](\epsilon, h) + f
\end{array}$$

Evaluation of expressions $\mathcal{E}[\![.]\!] : \mathbb{M} \to \mathbb{V}$

$$\begin{array}{rcl}
\mathcal{E}[\![v]\!](\epsilon, h) & \triangleq & v \\
\mathcal{E}[\![\&loc]\!](\epsilon, h) & \triangleq & \mathcal{L}[\![loc]\!](\epsilon, h) \\
\mathcal{E}[\![loc]\!](\epsilon, h) & \triangleq & h(\mathcal{L}[\![loc]\!](\epsilon, h)) \\
\mathcal{E}[\![e_1 \oplus e_2]\!](\epsilon, h) & \triangleq & \mathcal{E}[\![e_1]\!](\epsilon, h) \oplus \mathcal{E}[\![e_2]\!](\epsilon, h)
\end{array}$$

Condition tests Guard$[\![.]\!] : \mathcal{P}(\mathbb{M}) \to \mathcal{P}(\mathbb{M})$

$$\textbf{Guard}[\![b]\!](S) \triangleq \{(\epsilon, h) \in S \mid \mathcal{B}[\![b]\!](\epsilon, h) = \texttt{true}\}$$

Transformers Post$[\![.]\!] : \mathcal{P}(\mathbb{M}) \to \mathcal{P}(\mathbb{M})$

$$\begin{array}{lcl}
\multicolumn{3}{l}{\textbf{post}[\![stmt]\!]S \triangleq \text{match } stmt \text{ with:}} \\
\mid \texttt{skip} & \to & S \\
\mid loc :=_t exp & \to & \{(\epsilon, h[\mathcal{L}[\![loc]\!](\epsilon, h) \leftarrow_t \mathcal{E}[\![exp]\!](\epsilon, h)]) \mid (\epsilon, h) \in S\} \\
\mid loc :=_t \texttt{sbrk}(exp) & \to & \{(\epsilon, h[\mathcal{L}[\![loc]\!](\epsilon, h) \leftarrow_t \texttt{sbrk}(\mathcal{E}[\![exp]\!](\epsilon, h))]) \mid (\epsilon, h) \in S\} \\
\mid s_1; s_2 & \to & \textbf{Post}[\![s_2]\!](\textbf{Post}[\![s_1]\!](S)) \\
\mid \textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 & \to & (\textbf{Post}[\![s_1]\!](\textbf{Guard}[\![b]\!](S)) \cup (\textbf{Post}[\![s_2]\!](\textbf{Guard}[\![\neg b]\!](S)) \\
\mid \textbf{while } b \textbf{ do } s \textbf{ done} & \to & \textbf{Guard}[\![\neg b]\!](\textbf{lfp}^{\subseteq}(\lambda S_i.S \cup (\textbf{Post}[\![s]\!](\textbf{Guard}[\![b]\!](S_i)))))
\end{array}$$

Figure 6.5: Denotational semantics of the language

A condition test **Guard**$[\![b]\!] : \mathcal{P}(\mathbb{M}) \to \mathcal{P}(\mathbb{M})$ collects the set of concrete memory states in which b is **`true`**. The evaluation of the boolean expression $\mathcal{B}[\![b]\!]$ was left out since it is standard.

The concrete transformer for the statement *stmt* is defined as **Post**$[\![stmt]\!] : \mathcal{P}(\mathbb{M}) \to \mathcal{P}(\mathbb{M})$. It maps a set of initial concrete states before the execution of *stmt* to the set of reachable states after the execution. For example, the do-nothing statement `skip` does not update the memory states $S \in \mathcal{P}(\mathbb{M})$ before the execution, thus **Post**$[\![\texttt{skip}]\!](S) \triangleq S$. The special one is the transformer of a loop statement. **Post**$[\![\textbf{while } b \textbf{ do } s \textbf{ done}]\!](S)$ represents all concrete memory states that can be obtained from the finite iterations of **Post**$[\![s]\!] \circ$ **Guard**$[\![b]\!]$ on S and satisfies **Guard**$[\![\neg b]\!]$. In fact, the set of reachable states of the transformer for the loop can be obtained by fixpoint computation. We denoted by $\textbf{lfp}^{\subseteq}\textbf{F}$ the set of least-fixed points of the continuous function

$\mathbf{F} : \mathcal{P}(\mathbb{M}) \rightarrow \mathcal{P}(\mathbb{M})$. Thus, $\mathbf{Post}[\![\texttt{while } b \texttt{ do } s \texttt{ done}]\!](S) = \mathbf{lfp}^{\subseteq}\mathbf{F}$ where $\mathbf{F}(S_i) \triangleq S \cup \mathbf{Post}[\![p]\!](\mathbf{Guard}[\![b]\!](S_i))$.

```
1  typedef struct hdr_s {
2    struct hdr_s *fnx;
3    size_t size;
4    //@ghost bool isfree;
5  }  HDR;
6
7  static void *_hsta = NULL;
8  static void *_hend = NULL;
9  static HDR *frhd = NULL;
10 static size_t memleft;
11
12 void minit(size_t sz) {
13   size_t align_sz;
14   align_sz = (sz+sizeof(HDR)-1)
15             & ~(sizeof(HDR)-1);
16
17   _hsta = sbrk(align_sz);
18   _hend = sbrk(0);
19   frhd = _hsta;
20   frhd->size =
21         align_sz / sizeof(HDR);
22   frhd->fnx = NULL;
23   //@ghost frhd->isfree = true;
24
25   memleft = frhd->size;
26 }
```

(a) Globals

```
27 void* malloc(size_t nbytes) {
28   HDR *nxt, *prv;
29   size_t nunits =
30     (nbytes+sizeof(HDR)-1)/sizeof(HDR) + 1;
31   for (prv = NULL, nxt = frhd; nxt;
32        prv = nxt, nxt = nxt->fnx) {
33     if (nxt->size >= nunits) {
34       if (nxt->size > nunits) {
35         nxt->size -= nunits;
36         nxt += nxt->size;
37         nxt->size = nunits;
38       } else {
39         if (prv == NULL)
40           frhd = nxt->fnx;
41         else
42           prv->fnx = nxt->fnx;
43       }
44       memleft -= nunits;
45       //@ghost nxt->isfree = false;
46       return ((void*)(nxt + 1));
47     }
48   }
49   warning("Allocation Failed!");
50   return (NULL);
51 }
```

(b) Allocation

Figure 6.6: A piece of code of LA allocator

Example 7 (Evaluating an assignment)**.** We give an example of the assignment evaluation. We select the piece of code from LA allocator [Ald08] shown in Figure 6.6. The LA allocator tracks free chunks using a free list. The chunk header has two fields, `fnx` and `size`. The boundaries of the memory region managed by the LA allocator are given by the global variables, i.e., `_hsta` and `_hend`. To simplify the presentation, we added the ghost field `isfree`, to mark explicitly free chunks. Figure 6.6(b) is the implementation of the allocation method. It traverses the free list for searching a suitable candidate for allocating.

We consider the assignment at line 42 in Figure 6.6(b): `prv->fnx = nxt->fnx`. We assume the concrete state of program before line 42 is denoted by (ϵ_{42}, h_{42}). At this state, the structure of the memory managed by LA allocator is shown as the left part of Figure 6.7. The white chunk are the free chunks and the gray are the allocated ones. The heap list is specified by the

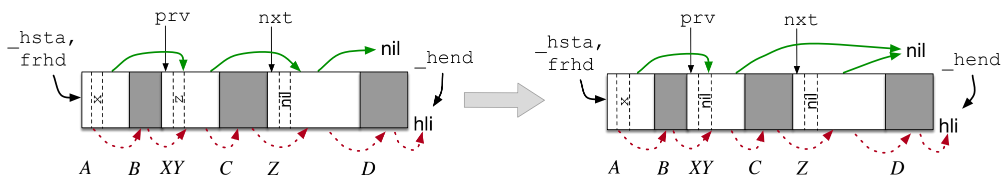


Figure 6.7: Concrete state change at line 45

dashed arrows and the green arrows form a free list. The program variables point to the chunks specified . The start addresses of the chunks are specified below the chunks. The program variable `prv` points to a chunk whose start address is X, i.e., $\epsilon_{42}(\texttt{prv}) = X$. The chunk has a field `fnx` and its address is Y. Before the assignment, the value stored in the `fnx` field is Z which is the start address of the chunk to which the program variable `nxt` points, i.e., $h_{42}(Y) = Z$. The evaluation of the assignment at line 42 proceeds as follows. We first evaluate the right-hand side:

$$\begin{aligned}\mathcal{E}[\![\texttt{nxt->fnx}]\!](\epsilon_{42}, h_{42}) &= h_{42}(\mathcal{L}[\![\texttt{nxt->fnx}]\!](\epsilon_x, h_{42}))\\ &= h_{42}(\mathcal{L}[\![\texttt{nxt}]\!](\epsilon_{42}, h_{42}) + \texttt{fnx})\\ &= h_{42}(\epsilon_{42}(\texttt{nxt}) + \texttt{fnx}) = \mathsf{nil}\end{aligned}$$

Then, we do location evaluation of the left-hand side of the assignment:

$$\begin{aligned}\mathcal{L}[\![\texttt{prv->fnx}]\!](\epsilon_{42}, h_{42}) &= \mathcal{L}[\![\texttt{prv}]\!](\epsilon_{42}, h_{42}) + \texttt{fnx}\\ &= \epsilon_{42}(\texttt{prv}) + \texttt{fnx}\\ &= X + \texttt{fnx} = Y\end{aligned}$$

After the execution of the assignment, the value stored at address Y, i.e.., the content of the `fnx` filed, is updated with nil. Thus, the green arrow (representing the points-to relation) out from the chunk that `prv` points to will point to nil : $h[Y \leftarrow \mathsf{nil}]$. $\triangle$

Concrete Domain: We define the concrete domain $\mathcal{M} = \mathcal{P}(\mathbb{M})$ to be the power set of concrete memory states. Domain $\mathcal{M}$ forms a complete lattice $(\mathcal{M}, \subseteq, \cup, \cap, \top, \bot)$ with subset containment $\subseteq$. The concrete joins and meets are set union $\cup$ and intersection $\cap$, respectively.

6.3 Abstract Domain Based on SLMA

In this section, we present our abstract domain used for analysing SDMA. The elements of the abstract domain are based on formulas of our logic SLMA introduced in Chapter 5. The ordering of abstract elements is defined as a sound entailment checking procedure (in Chapter 5.3.4) between the corresponding formulas.

Recall that logical formulas in SLMA consist of two main parts, pure part and spatial part. The inductive predicates, specifying heap regions, are defined with not only numeric constraints, but also data words constraints (shown in Figure 5.2, page 74). Thus, we design an abstract domain ($\mathbb{M}^\sharp$) in a modular way, i.e., we use the individual domain for each type of formulas, then we combine these domains together. The abstract domain $\mathbb{M}^\sharp$ is an *extended symbolic heap domain* which is the combination of a *shape heap domain* ($\mathbb{G}^\sharp$) encoding the spatial part, a *numerical domain* ($\mathbb{N}^\sharp$) encoding the pure arithmetic formulas [CH78,Min01b] and a *data words domain* ($\mathbb{W}^\sharp$) encoding the sequence constraints [BDE+10]. These domains are connected by using cofibered product operator [Ven96,CR13].

6.3.1 Numerical Abstract Domain

To track the pure arithmetic constraints, we employ some existing numerical domain. For example, if we consider the constraints, $L_1 - L_2 \# t$ (where L_1 and L_2 are location variables, $\#$ is one of the comparison operators and t is the integer term), then we use the polyhedra domain[CH78].

Let $C_\mathbb{N}$ be a concrete lattice of the interpretation functions: $\mathbb{I}_\mathbb{N} : \mathsf{AVar} \cup \mathsf{IVar} \to \mathbb{V}$ and $C_\mathbb{N}$ is defined as follows:

$$C_\mathbb{N} = (\mathcal{P}(\mathbb{I}_\mathbb{N}), \subseteq, \cup, \cap, \varnothing, \mathbb{I}_\mathbb{N}).$$

We define an abstract domain of $C_\mathbb{N}$ which is the numerical domain used in our analyzer, that is

$$\mathcal{N}^\sharp = \langle \mathbb{N}^\sharp, \sqsubseteq^\mathcal{N}, \sqcap^\mathcal{N}, \sqcup^\mathcal{N}, \top^\mathcal{N}, \bot^\mathcal{N} \rangle$$

where the elements of $\mathbb{N}^\sharp$ are constraints over location variables AVar and integer variables IVar, $\sqsubseteq^\mathcal{N}$ is the order relation between formulas, $\sqcap^\mathcal{N}$ and $\sqcup^\mathcal{N}$ are meet and join operators respectively. The concretization of the domain is defined as $\gamma_\mathbb{N} : \mathbb{N}^\sharp \to \mathcal{P}(\mathbb{I}_\mathbb{N})$. It associates to a set of numerical constraints in $\mathbb{N}^\sharp$ the set of valuations $\mathsf{AVar} \cup \mathsf{IVar} \to \mathbb{V}$ that satisfy the constraints. The widening operation of the domain is denoted by $\nabla^{\mathcal{N}}$.

The numerical domain could be a parameter of the analysis and the cost of the operations in the underlying numerical domain plays an important role in analysis. We assume that the numerical domain provides lattice operations (e.g., $\sqsubseteq^{\mathcal{N}}, \sqcap^{\mathcal{N}}$) and abstract transformers such as abstract condition tests, abstract transformers for assignments, projection.

6.3.2 Data Words Domain

The next abstract domain is built on top of sequence constrains. A sequence variable $W \in \mathsf{SVar}$ in SLMA can be also called as a data word. A sequence variable is mapped to a sequence of values by the interpretation function $\mathbb{I}_{\mathbb{W}} : \mathsf{SVar} \to \mathbb{V}^*$. Notice that the sequence constraints also contain numerical constraints, thus the concrete lattice of all interpretation functions is defined as follows:

$$C_{\mathbb{W}} = (\mathcal{P}(\mathbb{I}_{\mathbb{W}} \cup \mathbb{I}_{\mathbb{N}}), \subseteq, \cup, \cap, \varnothing, \mathbb{I}_{\mathbb{W}} \cup \mathbb{I}_{\mathbb{N}}).$$

We define a following abstract domain, called *data words domain*, for $C_{\mathbb{W}}$:

$$\mathcal{D}^{\sharp} = (\mathbb{W}^{\sharp}, \sqsubseteq^{\mathcal{W}}, \sqcap^{\mathcal{W}}, \sqcup^{\mathcal{W}}, \top^{\mathcal{W}}, \bot^{\mathcal{W}})$$

whose elements are conjunctions of some fragment of sequence formulas in SLMA which is expressive enough to capture the properties we need to infer during analysis of SDMA.

Table 6.1: Syntax of sequence formulas

$\Pi_{\mathbb{W}}^{=}$	$::=$	$W = w \mid \Pi_{\mathbb{W}}^{=} \wedge \Pi_{\mathbb{W}}^{=}$	quantifier-free formula
$\Pi_{\mathbb{W}}^{\forall}$	$::=$	$\forall X \in W \cdot A_G \Rightarrow A_U \mid \Pi_{\mathbb{W}}^{\forall} \wedge \Pi_{\mathbb{W}}^{\forall}$	quantified formula
w	$::=$	$\epsilon \mid [X] \mid W \mid w.w$	sequence terms
$\Pi_{\mathbb{W}}$	$::=$	$\Pi_{\mathbb{W}}^{=} \wedge \Pi_{\mathbb{W}}^{\forall} \wedge A$	abstract value in $\mathbb{W}^{\sharp}$

6.3.2.1 Abstract Elements

The syntax of the sequence constraints is defined in Table 5.1 (page 80). For readability, we recall it in Table 6.1.

The first component of a sequence constraint $\Pi_{\mathbb{W}} \in \mathbb{W}^{\sharp}$ is a conjunction of a quantifier-free part, denoted by $\Pi_{\mathbb{W}}^{=}$, which is a conjunction of quantifier-free formulas $W = w$. The quantifier-free part of a sequence constraint contains only the constraints $W_H = w$ and $W_F = w'$ where W_H and W_F are special variables representing the full sequence of start addresses of chunks in the heap and free list levels respectively. The second component is a quantified part, denoted by $\Pi_{\mathbb{W}}^{\forall}$, which is a conjunction of universally quantified formulas of the form $\forall X \in W \cdot A_G \Rightarrow A_U$ where the guard A_G is a constraint over the value of X and A_U is an arithmetical constraint over data values and terms. The formulas in A_U are formulas of the numerical domain $\mathcal{N}^{\sharp}$. In the universal constraints, only one variable is quantified and the formulas in A_G are fixed.The third component is the numerical formulas over field access terms (e.g., $\mathcal{F}_{\texttt{size}}(X)$) and location constraints which belong to the numerical domain.

Some properties of the addresses in the sequence variables are captured by the inductive predicates specifying the heap list or the free list. For example, the predicate $\mathsf{hls}(X;Y)[W]$ constrains the consecutive values in W to satisfy the relation $\mathcal{F}_{\texttt{fnx}}(X) = Y$. Also, property like eager coalescing policy is encoded in the hlsc predicate and does not require universally quantified formulas.

The $\Pi_{\mathbb{W}}^{\forall}$ part contains properties involving universally quantified formulas and is useful for describing fit policies. For example, the first-fit policy specified by the following formula which is the post-condition of the loop traversing a free-list (specified by the predicate fls) to find the first suitable chunk:

$$\begin{aligned}
&\mathsf{fls}(A;B)[W_1] * \mathsf{fck}(C;D) * \mathsf{fls}(E;F)[W_2] \\
&\quad \wedge \mathcal{F}_{\texttt{size}}(C) \geq s \\
&\quad \wedge \forall X \in W_1 \cdot \mathcal{F}_{\texttt{size}}(X) < s \\
&\quad \wedge W_F = W_1.[C].W_2
\end{aligned}$$

where s is the size requested for allocation and C is the address of the first fitting chunk. Recall that the next-fit policy is a special fit-policy. Suppose the position at which the traversal begins over the list is denoted by N, the next-fit policy is specified by replacing the start location A with N in the above formula. Similarly, after traversing a free list specified by fls to find the best suitable chunk, the post-condition obtained with the best-fit policy is specified by the following formula:

$$\begin{aligned}&\mathsf{fls}(A;B)[W_1] * \mathsf{fck}(C;D) * \mathsf{fls}(E;F)[W_2]\\ &\quad\wedge \mathcal{F}_{\mathtt{size}}(C) \geq s\\ &\quad\wedge \forall X \in W_1, W_2 \cdot \mathcal{F}_{\mathtt{size}}(X) \geq s \Rightarrow \mathcal{F}_{\mathtt{size}}(X) \geq \mathcal{F}_{\mathtt{size}}(C)\\ &\quad\wedge W_F = W_1.[C].W_2\end{aligned}$$

Considering all the properties over the structure of the memory and the content, we need the universal formulas of the form: $\forall X \in W \cdot A_G \Rightarrow A_U$ where A_G belongs to the fixed set of constrains (i.e., $\mathcal{G} = \{\mathcal{F}_{\mathtt{isfree}}(X)\#c\}$ where c is a constant and $\#$ stands for the comparison operator), A_U is a conjunction of linear constraints over function application on X or free variables, free variables and constants.

The concretization function of elements of $\mathbb{W}^\sharp$ is defined as $\gamma_{\mathbb{W}} : \mathbb{W}^\sharp \rightarrow \mathcal{P}(\mathbb{I}_{\mathbb{W}} \cup \mathbb{I}_{\mathbb{N}})$. The concretization of sequence constraints in $\mathbb{W}^\sharp$ is defined according to the semantics of these formulas. Because $\Pi_{\mathbb{W}}^{\forall}$ may include free numerical variable (chunk addresses or integers), the concretisation of such formula is composed of interpretation of sequence variables and of numerical variables.

Precisely, we denote by $\Pi_{\mathbb{W}}(\mathcal{W}, \mathcal{G})$ a sequence constraint with universally quantified constraints over a set of sequence variables $\mathcal{W}$ and $\mathcal{G}$ is a set of guard forms.

The constraint $\Pi_{\mathbb{W}}(\mathcal{W}, \mathcal{G})$ is defined in the following form:

$$\Pi_{\mathbb{W}}(\mathcal{W}, \mathcal{G}) = A(V) \wedge P_{\mathbb{W}} \wedge \Big(\bigwedge_{W_i \in \mathcal{W}} \Big(\bigwedge_{A_{G_i} \in \mathcal{G}} \forall X \in W_i \cdot A_{G_i}(W_i) \Rightarrow A_{U_i}(W_i, V)\Big)\Big)$$

where $P_{\mathbb{W}} = (W_H = w \wedge W_F = w')$ and $A(V)$ is a numerical constraint on the set of free variables V and function symbols.

6.3.2.2 Lattice Operators

Ideally, the order relation $\sqsubseteq^{\mathcal{W}}$ of the data words domain should be given by the logical implication. However, the entailment checking of the array logic fragment is undecidable which is already discussed in Section 5.3.4 (page 80). Assume that the array logic fragment is decidable, the complexity of checking the entailment could be also very high. Thus, designing a sound approximation of the logical implication is inevitable when we want to provide an efficient analysis. Due to the restricted syntax of data word constraints encoding the abstract values, we could define an efficient and sound ordering relation. This relation is a simplified version of the one defined in [Dra11].

Two elements in the domain $\mathbb{W}^\sharp$ are comparable if they have the same quantifier-free part. Given two sequence constraints shown as follows:

$$\Pi_{\mathbb{W}}(\mathcal{W},\mathcal{G}) = A(V) \wedge P_{\mathbb{W}} \wedge (\bigwedge_{W_i \in \mathcal{W}} (\bigwedge_{A_{G_i} \in \mathcal{G}} \forall X \in W_i \cdot A_{G_i}(W_i) \Rightarrow A_{U_i}(W_i, V))),$$

$$\Pi'_{\mathbb{W}}(\mathcal{W},\mathcal{G}) = A'(V) \wedge P'_{\mathbb{W}} \wedge (\bigwedge_{W_i \in \mathcal{W}} (\bigwedge_{A_{G_i} \in \mathcal{G}} \forall X \in W_i \cdot A_{G_i}(W_i) \Rightarrow A'_{U_i}(W_i, V))),$$

the order between them is defined as follows:

$$\Pi_{\mathbb{W}}(\mathcal{W},\mathcal{G}) \sqsubseteq^{\mathcal{W}} \Pi'_{\mathbb{W}}(\mathcal{W},\mathcal{G}) \text{ iff } P_{\mathbb{W}} = P'_{\mathbb{W}} \text{ and } \bigwedge_i (A_{U_i} \wedge A(V) \sqsubseteq^{\mathcal{N}} A'_{U_i} \wedge A'(V)).$$

The join operator $\Pi_{\mathbb{W}} \sqcup^{\mathcal{W}} \Pi'_{\mathbb{W}}$ is defined as follows:

$$\Pi_{\mathbb{W}} \sqcup^{\mathcal{W}} \Pi'_{\mathbb{W}} \triangleq \begin{cases} \top^{\mathcal{W}} & \text{if } P_{\mathbb{W}} \neq P'_{\mathbb{W}} \\ (A(V) \sqcup^{\mathcal{N}} A'(V)) \wedge P_{\mathbb{W}} \wedge P^{\forall}_{\mathbb{W}} & \text{otherwise} \end{cases}$$

$$\text{where } P^{\forall}_{\mathbb{W}} = \bigwedge_{W_i \in \mathcal{W}_i} (\bigwedge_{A_{G_i} \in \mathcal{G}} (\forall X \in W_i \cdot A_{G_i}(W_i) \Rightarrow A_{U_i} \sqcup^{\mathcal{N}} A'_{U_i})).$$

The meet and widening operator of the data words domain, denoted by $\sqcap^{\mathcal{W}}$ and $\nabla^{\mathcal{W}}$, are defined similarly by replacing $\sqcup^{\mathcal{N}}$ with $\sqcap^{\mathcal{N}}$ and $\nabla^{\mathcal{N}}$, respectively.

$$\Pi_{\mathbb{W}} \sqcap^{\mathcal{W}} \Pi'_{\mathbb{W}} \triangleq \begin{cases} \bot^{\mathcal{W}} & \text{if } P_{\mathbb{W}} \neq P'_{\mathbb{W}} \\ (A(V) \sqcap^{\mathcal{N}} A'(V)) \wedge P_{\mathbb{W}} \wedge P^{\forall}_{\mathbb{W}} & \text{otherwise} \end{cases}$$

$$\text{where } P^{\forall}_{\mathbb{W}} = \bigwedge_{W_i \in \mathcal{W}_i} (\bigwedge_{A_{G_i} \in \mathcal{G}} (\forall X \in W_i \cdot A_{G_i}(W_i) \Rightarrow A_{U_i} \sqcap^{\mathcal{N}} A'_{U_i})).$$

6.3.3 Shape Abstract Domain

We define the *shape abstract domain* $\mathbb{G}^\sharp$ which abstracts the shapes of concrete heaps specified by spatial formulas. Each abstract shape $\Sigma \in \mathbb{G}^\sharp$ over-approximates a set of concrete heaps. Let us recall the syntax of the spatial formulas Σ of SLMA. The general syntax of Σ is shown in Table 6.2.

We denote by $P_H(\overrightarrow{x})$ and $P_F(\overrightarrow{x})$ the predicates in the heap formulas and free list formulas respectively where $\overrightarrow{x}$ is the set of numerical and sequence variables over which the heaps are defined. In the shape abstract domain $\mathbb{G}^\sharp$, the spatial predicate is always defined with only one sequence variable, in some places, we simply put it in the brackets, e.g., $\mathsf{hls}(X;Y)[W]$.

Table 6.2: Syntax of spatial formulas

Σ	$::=$	$\Sigma_H \ni \Sigma_F$	spatial formulas
Σ_H	$::=$	$\mathsf{emp} \mid P_H(\overrightarrow{x}) \mid \Sigma_H * \Sigma_H$	heap formulas
Σ_F	$::=$	$\mathsf{emp} \mid P_F(\overrightarrow{x}) \mid \Sigma_F * \Sigma_F$	free list formulas

Graphical notation: For the sake of readability and to ease the manipulation of spatial formulas, we define the Gaifman graph representation for $\mathbb{G}^\sharp$. Figure 6.8 lists the basic graphical notations for spatial atoms. A node in the graph represents an address, e.g., each start address of the memory chunk or the start address of a sequence of bytes which can be specified by blk predicate is represented by a node. The constant null address is specified by a unique node $\#$. And the address right after the memory region is specified by a distinguished node hli.

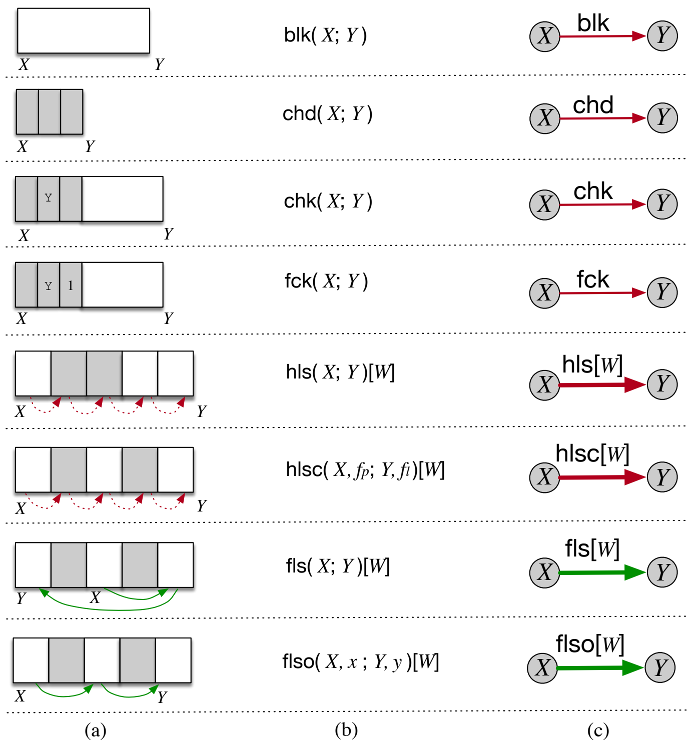


Figure 6.8: (a) concrete memory regions; (b) spatial atoms (c) graphical notations

The node may have outgoing edges. The edges of the graph describe disjoint memory regions and have several different types. They represent the neighbour relation of the heap list (e.g., next chunk, shown in red arrows), or the points-to relation of the free list (e.g., the `fnx` field, shown in green arrows). Edges connecting nodes are labeled with the spatial atoms with some parameters. The bold edges represent the memory region described by the summary predicates, e.g., hls, and the thin edges represent atomic predicates, e.g., chk. In some places, we use (X, P, Y) to denote the edge between node X and node Y where P is a non-summary atom and use $(X, P[W], Y)$ to represent the edge between node X and node Y where P is a summary predicate.

In the graph representation for the concrete memory state, there are two individual parts specifying the heap list and the free list respectively. They are linked using the composition operator $\Supset$.

Example 8 (Graph representation)**.** We present an example using the graph to represent the spatial formulas. In example 7, the concrete heap state before line 42 is shown as Figure 6.9(a) (Here, the shape abstract domain has no abstraction for program variables, thus, program variables are omitted in the abstract state).

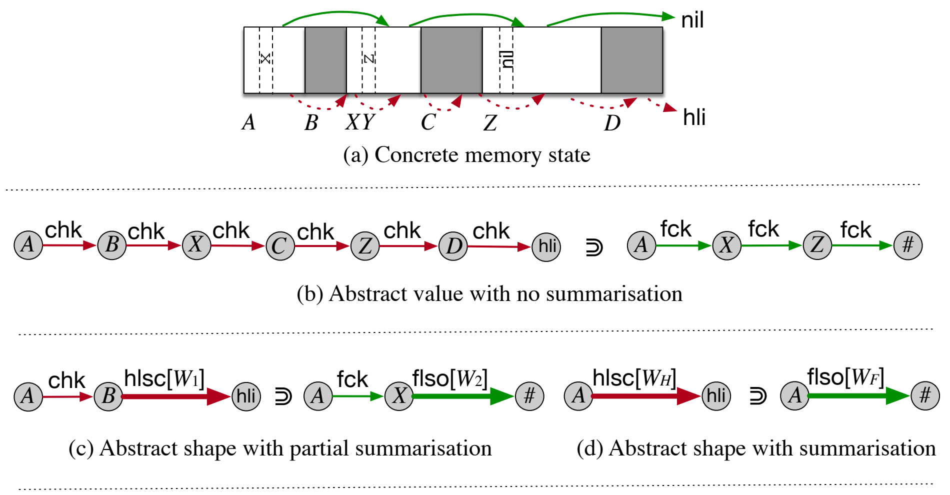

nil
X
Z
nil
hli
A B XY C Z D
(a) Concrete memory state
A chk B chk X chk C chk Z chk D chk hli ⋑ A fck X fck Z fck #
(b) Abstract value with no summarisation
A chk B hlsc[W1] hli ⋑ A fck X flso[W2] #
A hlsc[WH] hli ⋑ A flso[WF] #
(c) Abstract shape with partial summarisation
(d) Abstract shape with summarisation

Figure 6.9: Graph representation of an abstract heap

Figure 6.9(b) depicts the abstract shape with no summarisation where every start address of the memory chunk is represented by a node. A list with the arbitrary length $k (k \geq 2)$ can be summarized by an appropriate edge labeled by an inductive predicate. Thus, we can summarize a part of the concrete chunk list, e.g., an abstract shape with partial summarisation is specified as Figure 6.9(c).

$\triangle$

Concretization: We recall from the explanation of the logic the interpretation (or valuation) function $I \in \mathbb{I} \triangleq ((\mathsf{AVar} \cup \mathsf{IVar}) \rightharpoonup \mathbb{V}) \cup (\mathsf{SVar} \rightharpoonup \mathbb{V}^*)$ that maps logical variables (numerical variables, address variables and data words variables) to their values and the set of heaps $\mathbb{H} \triangleq \mathbb{A} \rightharpoonup \mathbb{V}^+$. Each shape constraint Σ corresponds to the set of concrete heaps defined under an interpretation I. The concretization $\gamma_{\mathbb{G}}(\Sigma)$ of Σ is the set of the concrete states, that satisfy $\exists \overrightarrow{x} \cdot \Sigma$ where $\overrightarrow{x}$ includes all logical variables in Σ . The concretization function $\gamma_{\mathbb{G}}$ has type $\gamma_{\mathbb{G}} : \mathbb{G}^\sharp \to \mathcal{P}(\mathbb{H} \times \mathbb{I})$. The concretization function for basic spatial predicates in $\mathbb{G}^\sharp$ is defined according to the semantics of these spatial predicates.

$$
\begin{array}{lcl}
\gamma_{\mathbb{G}}(\Sigma_H \Supset \Sigma_F) & \triangleq & \{(h, I) \mid (h, I) \in \gamma_{\mathbb{G}}(\Sigma_H) \land (\exists h' \subseteq h \; s.t \; (h', I) \in \gamma_{\mathbb{G}}(\Sigma_F) \\
& & \qquad \land \forall x \in \mathsf{dom}(h') \cdot h'(x)[\mathtt{isfree}] = 1\} \\
\gamma_{\mathbb{G}}(\Sigma_1 * \Sigma_2) & \triangleq & \{(h_1 \uplus h_2, I) \mid (h_1, I) \in \gamma_{\mathbb{G}}(\Sigma_1) \;\land (h_2, I) \in \gamma_{\mathbb{G}}(\Sigma_2) \\
& & \qquad \land \mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) = \varnothing\}
\end{array}
$$

6.3.4 Shape-Value Domain

The shape abstract domain $\mathbb{G}^\sharp$, corresponding to the spatial part, is quite weak since it only gives the abstraction of the separating memory cells. A separating shape graph $\Sigma \in \mathbb{G}^\sharp$ is defined over a set of logical variables, denoted by $\mathsf{LVar}(\Sigma)$, including a set of numerical variables $\mathsf{Num} \subseteq \mathsf{AVar} \cup \mathsf{IVar}$ (i.e., location variables and integer variables) and a set of sequence variables $\mathsf{Dw} \subseteq \mathsf{SVar}$ (i.e., data words variables). The shape abstraction should be enriched with the information over the values, e.g., the addresses and contents. We first enrich the shape abstract domain with the numerical domain, then enrich it again with the data words domain.

Enrich $\mathbb{G}^\sharp$ with a numerical domain: The numerical properties of the values stored in the heap specified by an abstract graph Σ are characterised by logical pure formulas over the set of numerical variables Num. These logical formulas expressing numerical properties are represented by a numerical domain $\mathbb{D}_{\mathbb{N}}$. The elements of $\mathbb{D}_{\mathbb{N}}$ concretize into sets of interpretations, satisfying value constraints, thus $\gamma_{\mathbb{D}_{\mathbb{N}}} : \mathbb{D}_{\mathbb{N}} \rightarrow \mathcal{P}(\mathsf{Num} \rightarrow \mathbb{V})$. We denote by $\mathbb{N}^\sharp$ the set of numerical properties corresponding to any abstract shape of heaps. Thus, any numerical constraints, denoted by $\Pi_{\mathbb{N}}$, over $\mathsf{AVar} \cup \mathsf{lVar}$ are elements of $\mathbb{N}^\sharp$ and the concretization of the domain is $\gamma_{\mathbb{N}} : \mathbb{N}^\sharp \rightarrow \mathcal{P}(\mathbb{I})$.

A way to combine the shape graph with a set of numerical constraints is to use product abstraction of the shape abstract domain $\mathbb{G}^\sharp$ and the numerical domain $\mathbb{N}^\sharp$, such as reduced product [CC79]. However, the numerical domain used to cover numerical properties depends on the shape graph. We use cofibered product operator to combine the shape abstract domain $\mathbb{G}^\sharp$ and the numerical domain $\mathbb{N}^\sharp$. We define the combined domain $\mathbb{G}^\sharp \rightrightarrows \mathbb{N}^\sharp$ (or *shape-numerical domain*) and its concretization: $\gamma_{\mathbb{G}^\sharp \rightrightarrows \mathbb{N}^\sharp} : (\mathbb{G}^\sharp \rightrightarrows \mathbb{N}^\sharp) \rightarrow \mathcal{P}(\mathbb{H} \times \mathbb{I})$ as follows:

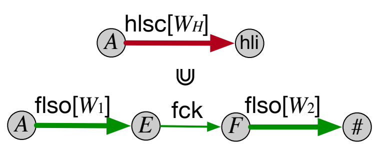
$$\begin{aligned} \mathbb{G}^\sharp \rightrightarrows \mathbb{N}^\sharp &\triangleq \{(\Sigma, \Pi_{\mathbb{N}}) \mid \Sigma \in \mathbb{G}^\sharp \wedge \Pi_{\mathbb{N}} \in \mathbb{N}^\sharp\} \\ \gamma_{\mathbb{G}^\sharp \rightrightarrows \mathbb{N}^\sharp}(\Sigma, \Pi_{\mathbb{N}}) &\triangleq \{(h, I) \mid (h, I) \in \gamma_{\mathbb{G}}(\Sigma) \ \wedge I \in \gamma_{\mathbb{N}}(\Pi_{\mathbb{N}})\} \end{aligned}$$

Enrich $\mathbb{G}^\sharp$ with a data words domain: Now we consider the constraints over sequence variables $\Pi_{\mathbb{W}}$. We define the *shape-value domain* $\mathbb{H}^\sharp = \mathbb{G}^\sharp \rightrightarrows (\mathbb{N}^\sharp \times \mathbb{W}^\sharp)$ (or *symbolic heap domain*) whose elements are tuples $\mathbb{h}^\sharp = (\Sigma, \Pi_{\mathbb{N}}, \Pi_{\mathbb{W}})$ where Σ is an abstract shape of the heaps, $\Pi_{\mathbb{N}}$ and $\Pi_{\mathbb{W}}$ are numerical and sequence constraints over the set of logical variables with which the shape defined (In the following content, we write the element of shape-value domain as (Σ, Π) if we do not distinguish numerical and sequence constraints). The shape-value domain and its concretization are defined as follows:

$$\begin{aligned} \mathbb{H}^\sharp &\triangleq \{(\Sigma, \Pi_{\mathbb{N}}, \Pi_{\mathbb{W}}) \mid \Sigma \in \mathbb{G}^\sharp \wedge \Pi_{\mathbb{N}} \in \mathbb{N}^\sharp \wedge \Pi_{\mathbb{W}} \in \mathbb{W}^\sharp\} \\ \gamma_{\mathbb{H}}(\Sigma, \Pi_{\mathbb{N}}, \Pi_{\mathbb{W}}) &\triangleq \{(h, I) \mid (h, I) \in \gamma_{\mathbb{G}}(\Sigma) \ \wedge I \in \gamma_{\mathbb{N}}(\Pi_{\mathbb{N}}) \wedge I \in \gamma_{\mathbb{W}}(\Pi_{\mathbb{W}})\} \end{aligned}$$

Definition 6.3 (Well-formed shape-value)**.** An abstract value of the shape-value domain $(\Sigma, \Pi_{\mathbb{N}}, \Pi_{\mathbb{W}}) \in \mathbb{H}^\sharp$ is *well-formed* if the sequence variable bound to each summary predicate of Σ is not constrained to be equal to a sequence term in $\Pi_{\mathbb{W}}$. ■

Example 9 (Abstract value of $\mathbb{H}^\sharp$)**.** The abstract value of $\mathbb{H}^\sharp$ extends the value of $\mathbb{G}^\sharp$ with the numerical and sequence constraints. For example, to capture allocation policies implemented by the `malloc` in Figure 6.6(b), we add numerical and sequence constraints. The first-fit policy obtained at line 37 of `malloc`, is specified by the value shown in Figure 6.10.



$$\begin{array}{l}(\mathcal{F}_{\texttt{size}}(E) \geq \text{nunits}) \wedge \\ (\forall X \in W_1 \cdot \mathcal{F}_{\texttt{size}}(X) < \text{nunits}) \wedge \\ (W_F = W_1.[E].W_2)\end{array}$$

Figure 6.10: An example of abstract value of $\mathbb{H}^\sharp$

$\triangle$

6.3.5 Extended Symbolic Heap Domain

The concrete memory states $\mathbb{M} \triangleq (\mathbb{E}, \mathbb{H})$ is composed of the set of concrete environments and concrete heaps. For concrete heaps $\mathbb{H}$, our shape-value domain gives an abstraction. Now we need to give the abstract counterpart to $\mathbb{E}$ to complete the abstraction for memory states.

We define the *abstract environment*, denoted by $\epsilon^\sharp \in \mathbb{E}^\sharp \triangleq \mathsf{PVar} \to \mathsf{LVar}$, which maps program variables to logical variables. The location variables are represented by nodes in the abstract graph, thus, an abstract environment is specified by arrows from program variables to the nodes, as is shown in Figure 6.11 which is the extended version of Figure 6.9.

The set of abstract memory states $\mathbb{M}^\sharp$ is represented by $\mathbb{E}^\sharp \times \mathbb{H}^\sharp$ and an abstract memory value $\mathbb{m}^\sharp$ is a pair $(\epsilon^\sharp, \mathbb{h}^\sharp)$ which consist of a set of mappings and a separating conjunction of spatial constraints, numerical constraints and sequence constraints. We define the *extended symbolic heap domain* $\mathbb{M}^\sharp = \mathbb{E}^\sharp \times \mathbb{H}^\sharp$ with its concretisation function $\gamma_{\mathbb{M}} : \mathbb{E}^\sharp \times \mathbb{H}^\sharp \to \mathcal{P}(\mathbb{E} \times \mathbb{H})$ as follows:

$$\begin{array}{rcl}\mathbb{M}^\sharp & \triangleq & \{(\epsilon^\sharp, \mathbb{h}^\sharp) \mid \epsilon^\sharp \in \mathbb{E}^\sharp \wedge \mathbb{h}^\sharp \in \mathbb{H}^\sharp\} \\ \gamma_{\mathbb{M}}(\epsilon^\sharp, \mathbb{h}^\sharp) & \triangleq & \{(I \circ \epsilon^\sharp, h) | (h, I) \in \gamma_{\mathbb{H}}(\mathbb{h}^\sharp)\}\end{array}$$

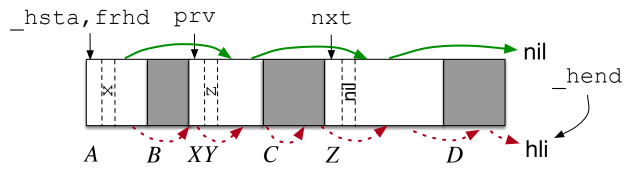
Summarisation: Recall that the shape abstract domain $\mathbb{G}^\sharp$ performs summarisation over separating memory cells without any restrictions. However, when we consider the program variables, summarisation has a restriction. In the abstraction $\mathbb{M}^\sharp$, the nodes which are not mapped by $\epsilon^\sharp$ are called *anonymous nodes*.

Only continuous anonymous nodes in the list can be summarised in a summary predicate. Thus, the nodes in the abstract shape with summarisation are either distinguished nodes or nodes pointed by program variables. The number of the nodes in the abstract shape is *finite* due to the finite set of program variables.

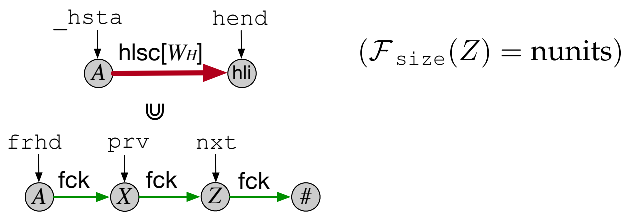
Definition 6.4. (**Graphical heap**). The shape of an abstract heap in the domain $\mathbb{M}^\sharp$ is represented by a tuple $g^\sharp = (\mathsf{N}, \mathsf{E})$ where:

- $\mathsf{N} \subseteq \mathsf{LVar}$ is a set of nodes which includes the distinguished nodes nil (we also denote by $\sharp$ the node of nil) and hli representing the null address and the heap limitation, respectively;
- $\mathsf{E} \subseteq (\mathsf{N} \times \mathbb{P} \times \mathsf{N}) \cup (\mathsf{PVar} \times \{*\} \times \mathsf{N})$ is a set of edges connecting nodes and points-to edges associating nodes to pointer variables. An edge is represented by a tuple, e.g., $e = (n_1, p, n_2)$ is an edge connecting node n_1 and node n_2 and it is labelled by a spatial atom $p \in \mathbb{P}$ and $e = (x, *, n_x)$ is a points-to edge representing that the program variable x points to the node n_x. Obviously, a points-to edge $(p, *, X)$ in Σ corresponds to the element in the abstract environment $\{p \mapsto X\} \in \epsilon^\sharp$. For simplicity, we denote by $p(X; Y)$ and $p[W](X; Y)$ the unsummarized edge and summary edge.

■



(a) Concrete memory state



(b) Abstract value with summarisation

Figure 6.11: Abstract value of $\mathbb{M}^\sharp$

Example 10 (Abstract value of $\mathbb{M}^\sharp$)**.** The abstract value of $\mathbb{M}^\sharp$ extends the value of $\mathbb{H}^\sharp$ with the abstract environment mappings. An example is shown in Figure 6.11 which is the extended value of the one in Example 8. The abstract value has a graph Σ and numerical and sequence constraints. $\triangle$

Disjunctive abstraction: The abstraction $\mathbb{M}^\sharp$ is not enough to express the abstract value of each program point, because a program point may collect several ingoing values which can not be joined. Thus a program point may have more than one element of $\mathbb{M}^\sharp$. Also, when comes a loop which traverses a list and the precondition is given in the summarized form, then an *unfolding* operation is called. The detailed definition of unfolding is given in the next chapter describing abstract domain operations. Given a summarized abstract value $\mathbb{m}^\sharp$, $\texttt{Unfold}_{\mathbb{M}}$ generates a finite set of disjunctive abstract elements, $\{\mathbb{m}^\sharp_0, \mathbb{m}^\sharp_2, ..., \mathbb{m}^\sharp_{n-1}\}$. To support disjunctive abstractions, we define an abstract domain $\mathbb{A}^\sharp$ with the concretization $\gamma_{\mathbb{A}} : \mathbb{A}^\sharp \to \mathcal{P}(\mathbb{M}^\sharp)$. The abstract value at each program point is a set of elements of $\mathbb{M}^\sharp$.

$$\begin{array}{rcl} \mathbb{A}^\sharp & \triangleq & \mathcal{P}_{\text{fin}}(\mathbb{M}^\sharp) \\ \gamma_{\mathbb{A}}(\mathbb{a}^\sharp) & \triangleq & \bigcup\{\gamma_{\mathbb{M}}(\mathbb{m}^\sharp) \mid \mathbb{m}^\sharp \in \mathbb{a}^\sharp\} \end{array}$$

6.3.6 Abstract Value

We summarize the restrictions of the abstract elements and the internal representation of the abstract value of $\mathbb{A}^\sharp$.

Restriction: Values in $\mathbb{A}^\sharp$ are a restricted form of logic formulas. More precisely, $\mathbb{A}^\sharp$ includes a special value for $\top$ and finite elements of the form:

$$A^\sharp ::= \bigcup_{0 \le i \le n} \{(\epsilon^\sharp_i, (\Sigma_i, \Pi_{\mathbb{N}_i}, \Pi_{\mathbb{W}_i}))\} \tag{6.1}$$

where $\epsilon^\sharp_i \in \mathbb{E}^\sharp$ is an abstract environment mapping program variables to symbolic location variables, $\Pi_{\mathbb{N}_i}$ includes arithmetic constraints allowed by $\mathbb{N}^\sharp$. Furthermore, the usage of sequence variables in Σ_i and $\Pi_{\mathbb{W}_i}$ is restricted as follows:

R$_1$: A sequence variable is bound to exactly one list segment atom in Σ_i; thus Σ_i defines an injection between list segment atoms and sequence variables.

R$_2$: $\Pi_{\mathbb{W}_i}$ contains only the sequence constraints $W_H = w$ and $W_F = w'$, where W_H and W_F are special variables representing the full sequence of start addresses of chunks in the heap resp. free list levels.

In addition, the universal constraints in the pure formulas $\Pi_{\mathbb{N}_i}$ and $\Pi_{\mathbb{W}_i}$ are restricted such that, in any formula $\forall X \in W \cdot A_G \Rightarrow A_U$:

R$_3$: A_G and A_U use only terms where X appears inside a field access $\mathcal{F}_\star(X)$.

R$_4$: A_G has one of the forms: $\mathcal{F}_{\texttt{size}}(X) \# i$, $\mathcal{F}_{\texttt{size}}(X) \ - c\# \ i$, $\mathcal{F}_{\texttt{isfree}}(X) = i$.

These restrictions still permit to specify SDMA policies like first-fit and besides enable an efficient inference of universal constraints.

Internal representation: To ease the manipulation of extended spatial formulas $\langle \epsilon^\sharp, \mathbb{h}^\sharp \rangle$, we use their Gaifman graph representation, like in Figure 6.11: nodes represent symbolic locations variables and labeled edges represent the spatial atoms in $\mathbb{h}^\sharp$ or mappings in $\epsilon^\sharp$. The universal formulas are represented by a map binding each pair (W, A_G) built from a sequence variable and some guard A_G to a numerical abstract value.

6.3.7 Lattice Operators

6.3.7.1 Ordering

The partial order $\sqsubseteq^{\mathcal{A}}$ in $\mathbb{A}^\sharp$ is defined using a sound procedure inspired by [BDES12, ESW15]. For any two non trivial abstract values $A^\sharp, B^\sharp \in \mathbb{A}^\sharp$, $A^\sharp \sqsubseteq^{\mathcal{A}} B^\sharp$ if for each value $(\epsilon^\sharp_i, (\Sigma_i, \Pi_{\mathbb{N}_i}, \Pi_{\mathbb{W}_i})) \in A^\sharp$ there exists a value $(\epsilon^\sharp_j, (\Sigma_j, \Pi_{\mathbb{N}_j}, \Pi_{\mathbb{W}_j})) \in B^\sharp$ such that:

- there is a graph isomorphism between the Gaifman graphs of spatial formula at each level of abstraction from Σ_i to Σ_j; this isomorphism is defined by a bijection $\Psi : \mathrm{img}(\epsilon^\sharp_i) \to \mathrm{img}(\epsilon^\sharp_j)$ between symbolic location variables and a bijection Ω between sequence variables. Thus, $\Sigma_i[\Psi][\Omega] = \Sigma_j$,
- for each sequence constraint $W = w$ in $\Pi_{\mathbb{W}_i}$, $\Omega(W) = \Omega(w)$ is a sequence constraint in $\Pi_{\mathbb{W}_j}$,

- $\Psi(\Pi_{\mathbb{N}_i}) \sqsubseteq^{\mathcal{N}} \Pi_{\mathbb{N}_j}$,
- for each W defined in Σ_i and for each universal constraint $\forall X \in W \cdot A_G \Rightarrow A_U$ in $\Pi_{\mathbb{W}_i}$, then $\Pi_{\mathbb{W}_j}$ contains a universal constraint on $W' = \Omega(W)$ of the form $\forall X \in W' \cdot A_G \Rightarrow A'_U$ such that $\Psi(\Pi_{\mathbb{N}_i} \wedge A_U) \sqsubseteq^{\mathcal{N}} A'_U$.

The following theorem is a consequence of restrictions on the syntax of formulas used in the abstract values.

Theorem 6.3 ($\sqsubseteq^{\mathcal{A}}$ soundness)**.** *If* $A^\sharp \sqsubseteq^{\mathcal{A}} B^\sharp$ *then* $\gamma_{\mathbb{A}}(A^\sharp) \subseteq \gamma_{\mathbb{A}}(B^\sharp)$. ■

Proof. Notice that the final abstract domain $\mathbb{A}^\sharp$ is the combination of several domains. The soundness of the partial order of the abstract domain $\mathbb{A}^\sharp$ is guaranteed by the soundness of each partial order in each component domain. □

6.3.7.2 Join

Given two non-trivial abstract values, $A^\sharp$ and $B^\sharp$, their join is computed by joining the pure parts of bindings with isomorphic shape graphs [BDES11]. Formally, for each pair of values $(\epsilon_i^\sharp, (\Sigma_i, \Pi_{\mathbb{N}_i}, \Pi_{\mathbb{W}_i}) \in A^\sharp$ and $(\epsilon_j^\sharp, (\Sigma_j, \Pi_{\mathbb{N}_j}, \Pi_{\mathbb{W}_j})) \in B^\sharp$ such that there is a graph isomorphism defined by Ψ and Ω between $\langle \epsilon_i^\sharp, \Sigma_i \rangle$ and $\langle \epsilon_j^\sharp, \Sigma_j \rangle$, we define their join to be the mapping $\{(\epsilon_j^\sharp, (\Sigma_j, \Pi_{\mathbb{N}}, \Pi_{\mathbb{W}}))\}$ where $\Pi_{\mathbb{N}}$ and $\Pi_{\mathbb{W}}$ are defined by:

- $\Pi_{\mathbb{W}}$ has the same sequence constraints as $B^\sharp$, i.e., $\Pi_{\mathbb{W}} \triangleq \Pi_{\mathbb{W}_j}$,
- $\Pi_{\mathbb{N}} \triangleq \Psi(\Pi_{\mathbb{N}_i}) \sqcup^{\mathcal{N}} \Pi_{\mathbb{N}_j}$,
- for each W sequence variable in $\mathsf{dom}(\Omega)$ and for each type of constraint A_G, then $\Pi_{\mathbb{W}}$ contains the formula $\forall X \in \Omega(W) \cdot A_G \Rightarrow \Psi(A_{U,i}) \sqcup^{\mathcal{N}} A_{U,j}$ where $A_{U,i}$ (resp. $A_{U,j}$) is the constraint bound to W (resp. $\Omega(W)$) in $\Pi_{\mathbb{W}_i}$ (resp. $\Pi_{\mathbb{W}_j}$) for guard A_G or $\top$ if no such constraint exists.

The join of two values with non-isomorphic spatial parts is the union of the two values. Then, $(A^\sharp \sqcup^{\mathcal{A}} B^\sharp)$ computes the join of values in $A^\sharp$ with each values in $B^\sharp$. Intuitively, the operator collects the disjuncts of $A^\sharp$ and $B^\sharp$ but replaces the disjuncts with isomorphic spatial parts by one disjunct which maps the spatial part to the join of the pure parts. Two universal constraints are joined when they concern the same sequence variables and guard A_G since $\big((\forall w \in W \cdot A_G \Rightarrow A_1) \vee (\forall w \in W \cdot A_G \Rightarrow A_2)\big) \Rightarrow \big(\forall w \in W \cdot A_G \Rightarrow (A_1 \vee A_2)\big)$.

Theorem 6.4 ($\sqcup^{\mathcal{A}}$ soundness)**.** *For any* $A^\sharp, B^\sharp \in \mathbb{A}^\sharp$, $\gamma_{\mathbb{A}}(A^\sharp) \cup \gamma_{\mathbb{A}}(B^\sharp) \subseteq \gamma_{\mathbb{A}}(A^\sharp \sqcup^{\mathcal{A}} B^\sharp)$. ■

6.3.7.3 Cardinality

The number of elements in an abstract memory value $A^\sharp \in \mathbb{A}^\sharp$ increases during the symbolic execution by the introduction of new existential variables keeping track of the created chunks. Although the analysis stores only values with linear shape of lists (other shapes are signalled as an error state), the number of linear shapes is exponential in the number of nodes, in general. Keeping small the number of addresses used in the shape part of heap and free list level is very important for the efficiency of the analysis. Moreover, to some of these addresses are bound feature variables which blow-up with a constant factor the variables of the pure part.

We make the compromise to keep in the shape parts only k addresses which are used in the shape atoms but are not aliased by program variables, also called *anonymous addresses*. k is a parameter of the analysis and it is usually small, e.g., 2 or 3. It especially influences the quality of values in the $\mathbb{W}^\sharp$ domain, i.e., the universally quantified formulas. An abstract memory state $A^\sharp \in \mathbb{A}^\sharp$ is k-*normalised* if all its conjuncts contain shape graphs with less than k anonymous addresses bound by predicate atoms.

We avoid this abstract memory value explosion by eliminating existential variables using the transformation rules that replace sub-formulas by predicates, an operation classically called *predicate folding* (its definition is detailed in the next chapter). This operation uses lemmas (1)–(4), as discussed in Section 5.3.2 (page 77). The application of the lemma follows a heuristics that searches to replace sets of atoms including an address referenced by a program variable and some anonymous addresses by an unique predicate. If an abstract memory value can not be normalised, it is replaced soundly by $\top$. At the heap list level, the failure to fold abstract values may correspond to a memory leak and a warning is issued. Also, the normalisation may lead to an empty pure constraint, in which case the disjunct representing the abstract value is removed. An empty list of disjuncts is equivalent to $\bot$.

6.3.7.4 Widening

The widening operator $\nabla^{\mathcal{A}} : \mathbb{A}^\sharp \times \mathbb{A}^\sharp \to \mathbb{A}^\sharp$ in $\mathbb{A}^\sharp$ should ensure that both shapes and data invariants are stable after finitely many iterations. The domain of abstract values is bounded by an exponential on the number of pointer program variables local to SDMA methods which is small in general, e.g., ≤ 3 in our benchmark. However, the domain of pure formulas used in the image of abstract values is not bounded because of integer constants. Therefore a widening operation for the combined domain $\mathbb{A}^\sharp$ can be obtained by simply using the widening operators ($\nabla^{\mathcal{N}}$ and $\nabla^{\mathcal{W}}$) instead of the joins in the data domains ($\sqcup^{\mathcal{N}}$ and $\sqcup^{\mathcal{W}}$).

Given two abstract values, $A^\sharp$ and $B^\sharp$ and for each $(\epsilon_i^\sharp, (\Sigma_i, \Pi_{\mathbb{N}_i}, \Pi_{\mathbb{W}_i}) \in A^\sharp$ and $(\epsilon_j^\sharp, (\Sigma_j, \Pi_{\mathbb{N}_j}, \Pi_{\mathbb{W}_j})) \in B^\sharp$ such that there is a graph isomorphism defined by Ψ and Ω between $\langle \epsilon_i^\sharp, \Sigma_i \rangle$ and $\langle \epsilon_j^\sharp, \Sigma_j \rangle$, the widening operator $\nabla^{\mathcal{A}}$ is defined as follows:

$$A^\sharp \, \nabla^{\mathcal{A}} \, B^\sharp \triangleq \begin{cases} A^\sharp & \text{if } B^\sharp = \bot \\ B^\sharp & \text{if } A^\sharp = \bot \\ \bigcup\limits_{0 \leq i \leq n} \{(\epsilon_j^\sharp, \Sigma_i, (\Psi(\Pi_{\mathbb{N}_i}) \nabla^{\mathcal{N}} \Pi_{\mathbb{N}_j}), (\Omega(\Pi_{\mathbb{W}_i}) \nabla^{\mathcal{W}} \Pi_{\mathbb{W}_j})\} & \text{otherwise} \end{cases}$$

CHAPTER 7

Static Analysis Operations and Algorithm

In this chapter, we define the main abstract operations in static analysis of SDMA, i.e., the abstract transformers for program statements, such as assignment and condition tests. As mentioned in Section 6.3.5, the important abstract operation is *unfolding* of summary predicates in order to materialize the points-to atoms which allows to mutate the heap. The novel aspect of our proposal is the use of a *hierarchical unfolding* of predicate depending on the level (heap list or free list) required by the program statements. The analyser determines the level that should be materialized and concretizes the summary edge until the targeted edge.

This chapter is structured as follows. In § 7.1, we describe the abstract postcondition of the language we considered. § 7.2 presents the abstract operations in the underlying domains and defines the hierarchical unfolding and folding operations in our analysis. The abstract transformers for program assignments and condition tests are presented in § 7.3. The analysis algorithm and the experimental results are detailed in § 7.4 and § 7.5 presents related work and concludes static analysis of SDMA.

7.1 Abstract Postcondition

We now describe how program statements are interpreted using our abstract domain $\mathbb{A}^\sharp$ to yield a computable over-approximation of the set of reachable concrete memory states.

Figure 7.1 defines the abstract postconditions corresponding to the concrete postconditions shown in Figure 6.5. The abstract transformer for the program statement $stmt$ is denoted by $\mathbf{Post}^\sharp[\![stmt]\!] : \mathbb{A}^\sharp \to \mathbb{A}^\sharp$. Let $S^\sharp$ be a set of abstract memory states. The simplest statement is the do-nothing statement `skip` which does not update the memory state, thus its abstract transformer is defined as $\mathbf{Post}^\sharp[\![\texttt{skip}]\!](S^\sharp) \triangleq S^\sharp$ which is the identity.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{Post}^\sharp[\![stmt]\!](S^\sharp) \triangleq \text{match } stmt \text{ with:}} \\
\mid \texttt{skip} & \to & S^\sharp \\
\mid loc :=_t exp & \to & \mathbf{Norm}(\bigcup\{\mathbf{Assign}^\sharp_{\mathbb{M}}(loc, exp, t, \mathbb{m}^\sharp) \mid \mathbb{m}^\sharp \in S^\sharp\}) \\
\mid loc :=_t \texttt{sbrk}(exp) & \to & \mathbf{Norm}(\bigcup\{\mathbf{Assign}^\sharp_{\texttt{sbrk}}(loc, exp, t, \mathbb{m}^\sharp) \mid \mathbb{m}^\sharp \in S^\sharp\}) \\
\mid s_1; s_2 & \to & \mathbf{Post}^\sharp[\![s_2]\!](\mathbf{Post}^\sharp[\![s_1]\!](S^\sharp)) \\
\mid \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 & \triangleq & \mathbf{Post}^\sharp[\![s_1]\!](\mathbf{Guard}^\sharp[\![b]\!](S^\sharp)) \\
 & & \quad \sqcup^{\mathcal{A}} \mathbf{Post}^\sharp[\![s_2]\!](\mathbf{Guard}^\sharp[\![\neg b]\!](S^\sharp)) \\
\mid \texttt{while } b \texttt{ do } s \texttt{ done} & \to & \mathbf{Guard}^\sharp[\![\neg b]\!](\mathbf{Fp}(\lambda S^\sharp_i.\text{let } S^\sharp_p = (S^\sharp_i \sqcup^{\mathcal{A}} \\
 & & \quad \mathbf{Post}^\sharp[\![s]\!](\mathbf{Guard}^\sharp[\![b]\!](S^\sharp_i))) \text{ in} \\
 & & \quad \text{if } S^\sharp_p \cong S^\sharp_i \text{ then } S^\sharp_p \nabla^{\mathcal{A}} S^\sharp_i \text{ else } S^\sharp_p)(S^\sharp))
\end{array}
$$

Figure 7.1: Abstract postconditions

Recall that in our programming language, $loc \in \mathbb{Loc}$ and $exp \in \mathbb{Exp}$ are location and value expressions respectively. For the assignment $loc :=_t exp$, its abstract transformer is denoted by $\mathbf{Assign}^\sharp_{\mathbb{M}}$ and is formally defined in Section 7.3.

The abstract transformer $\mathbf{Assign}^\sharp_{\mathbb{M}}(loc, exp, t, \mathbb{m}^\sharp)$ calls two basic operations: evaluations of a location loc and of an expression exp over the abstract state $\mathbb{m}^\sharp$, denoted by $\mathbf{EvalL}^\sharp[\![loc]\!]\mathbb{m}^\sharp$ and $\mathbf{EvalE}^\sharp[\![exp]\!]\mathbb{m}^\sharp$, respectively. Then, it manipulates the abstract memory state by calling $\mathbf{Mutate}^\sharp_{\mathbb{M}}$ operation. These operations are provided by the interface of the domain $\mathbb{M}^\sharp$.The normalization function **Norm** normalizes the abstract value by using the folding operation. If in the abstract state $\mathbb{m}^\sharp$, the heap that has to be mutated is summarized, then the evaluation transformers $\mathbb{m}^\sharp$ concretize the summarized edges, i.e., preform unfolding operation first. All of these operations used in $\mathbf{Assign}^\sharp_{\mathbb{M}}$ are detailed in Section 7.2.

For the assignment $loc :=_t \texttt{sbrk}(exp)$, which makes a system call to `sbrk` function, its abstract transformer is represented by $\mathbf{Assign}^{\sharp}_{\texttt{sbrk}}$ and is formally defined in Section 7.3.1.4.

As shown in Figure 7.1, the abstract transformer for the conditional branching statement collects two abstract states which are postconditions of each branch using the abstract join operation. The abstract condition test is defined by $\mathbf{Guard}^{\sharp}$. It is based on the condition test in domain $\mathbb{M}^{\sharp}$ which is denoted by $\mathbf{Guard}^{\sharp}_{\mathbb{M}}$.

For the loop statement, we denote by $\mathbf{Fp}(F)(S)$ a function that computes the fix-point of F reached by successive iterations of F starting at a state S. The termination of the iteration is guaranteed by the widening operation $\nabla^{\mathcal{A}}$.

7.2 Hierarchical Unfolding and Folding

In this section, we define the basic abstract operations (like unfolding, folding, decomposing, etc) in the shape domain $\mathbb{G}^{\sharp}$, data words domain $\mathbb{W}^{\sharp}$ and shape-value domain $\mathbb{H}^{\sharp}$. We assume that the numerical domain used provides some abstract operations. Based on these abstract operations, we define the hierarchical unfolding and folding operations, that performs these operations at the appropriate abstraction level, i.e., heap list or free list.

7.2.1 Abstract Operations in Shape Domain $\mathbb{G}^{\sharp}$

7.2.1.1 Unfolding and Folding Summary

Predicate unfolding: We define *predicate unfolding* operation, denoted by $\mathbf{UnfoldList}^{\sharp}_{\mathbb{G}}$, which takes an abstract shape Σ and a summary predicate in Σ that should be unfolded, and returns a disjunction of shapes and data constraints. More precisely, given a summary predicate $P(\overrightarrow{x})[W]$ in the abstract shape Σ such that the predicate is defined by (for sake of readability, we applied the substitution of formal parameters in the predicate definition with the actual ones):

$$P(\overrightarrow{x})[W] \triangleq \bigvee_{0\le i\le n} \exists \overrightarrow{y_i} \cdot \Sigma_i \wedge \Pi_{\mathbb{N}_i} \wedge \Pi_{\mathbb{W}_i},$$

(where $\overrightarrow{y_i}$ is a vector of fresh variables in each disjunct) the unfolding of the predicate $P(\overrightarrow{x})[W]$ is defined as follows:

$$\mathbf{UnfoldList}^{\sharp}_{\mathbb{G}}(\Sigma, P(\overrightarrow{x})[W], X, W') \triangleq \bigvee_{0\le i\le n} (((\Sigma \setminus P(\overrightarrow{x})[W]) * \Sigma_i), \Pi_{\mathbb{N}_i}, \Pi_{\mathbb{W}_i}).$$

According to the definitions of the summary predicates (Table 5.2, page 74), in each disjunct, the generated sequence constraint $\Pi_{\mathbb{W}_i}$ is either $W = \epsilon$ or $W = [X].W'$ where X, W are fresh logical variables in $\overrightarrow{y_i}$. Therefore, the sequence variable W may be substituted by the sequence expression on the right to obtain an abstract value where data word part is of the form required by the general definition.

Predicate folding: In order to obtain loop invariants and normalize the abstract values, the analysis should reconstruct the summaries. We define the basic folding operation in the shape domain to construct a summary predicate from the separating parts.

Let us fix an abstract shape Σ with a sub-formula specified as $p(X;Y) * P(Y, \overrightarrow{y})[W]$ where $p \in \{\mathsf{chk}, \mathsf{fck}\}$ is a chunk or a free chunk atom, $P \in \{\mathsf{hls}, \mathsf{hlsc}, \mathsf{fls}, \mathsf{flso}\}$ is a summary atom, X, Y and $\overrightarrow{y}$ stand for the parameters according to the definitions of the predicates. If p is an atom matching P's definition and the node Y is the start of the edge labeled by $P[W]$ (e.g., chk for hls because one case of hls definition is $\mathsf{chk}(X;Y) * \mathsf{hls}(Y;Z)[W]$), then the sub-formula can be folded using the folding operation $\textbf{FoldList}^{\sharp}_{\mathbb{G}}$ defined as follows:

$$\textbf{FoldList}^{\sharp}_{\mathbb{G}}(\Sigma, p(X;Y)*P(Y,\overrightarrow{y})[W], W') \triangleq (\Sigma \backslash (p(X;Y)*P(\overrightarrow{y})[W]))*P(X,\overrightarrow{y})[W'].$$

If the sub-formula in Σ is specified as $P(\overrightarrow{y}, X)[W] * p(X;Y)$, it can be also folded and the folding is defined similar to $\textbf{FoldList}^{\sharp}_{\mathbb{G}}$. The folding operation $\textbf{FoldList}^{\sharp}_{\mathbb{G}}$ is called by the folding operation in the shape-value domain and cooperates with other operations manipulating numerical constraints. It is applied only when some conditions are satisfied which is explained in Section 7.2.3.

Example 11 (Predicate folding)**.** An example of folding hls which has three cases is shown in Figure 7.2. The first one folds two separating chunks $\mathsf{chk}(X;Y) * \mathsf{chk}(Y;Z)$ into a heap list segment $\mathsf{hls}(X;Z)[W]$ where W is a fresh sequence variable. It is a special case of the second one because $\mathsf{hls}(Y;Z)[W]$ could be $\mathsf{chk}(Y;Z)$ if the list has only one chunk. The second one transforms $\mathsf{chk}(X;Y) * \mathsf{hls}(Y;Z)[W]$ into $\mathsf{hls}(X;Z)[W']$. The node Y is removed and a new summary edge is added between node X and node Z. The third one folds $\mathsf{hls}(X;Y)[W] * \mathsf{chk}(Y;Z)$ into $\mathsf{hls}(X;Z)[W']$. The fresh variables used in the folded edge will be used to replaces some variables in the abstract value when doing the folding in the shape-value domain. △

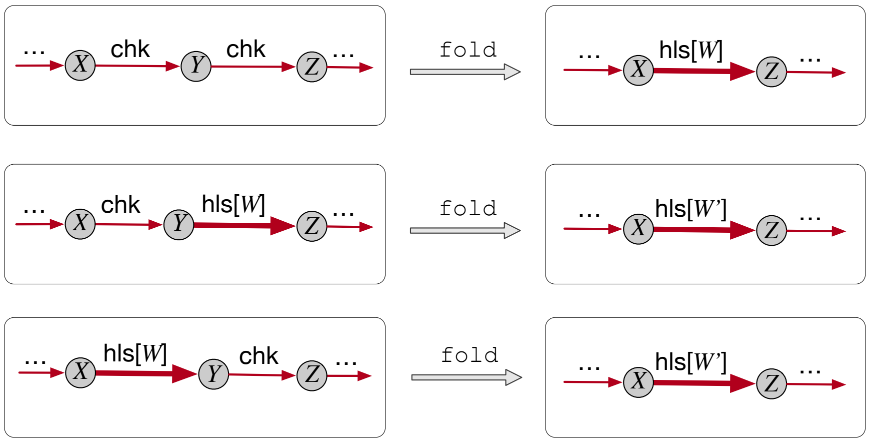


Figure 7.2: Example of **FoldList**$^\sharp_{\mathbb{G}}$

7.2.1.2 Decomposing and Composing Summary

Decomposition: The *decomposition operation* applies the segment decomposition lemma (in Section 5.3.2, page 77). Given a spacial value Σ with a summary $P(\overrightarrow{x})[W]$, the decomposition operation decomposes the summary $P(\overrightarrow{x})[W]$ into two smaller parts and generates new pure constraints. It is denoted by **DecompList**$^\sharp_{\mathbb{G}}$ and defined as follows:

$$\mathbf{DecompList}^\sharp_{\mathbb{G}}(\Sigma, P(\overrightarrow{x})[W], W_1, W_2) \triangleq (\Sigma^n, \Pi^n_{\mathbb{N}}, \Pi^n_{\mathbb{W}})$$

where

$$\Sigma^n = (\Sigma \setminus P(\overrightarrow{x})[W]) * P(\overrightarrow{y})[W_1] * P(\overrightarrow{z})[W_2] \text{ and } \Pi^n_{\mathbb{W}} = (W = W_1.W_2),$$

W_1, W_2 are fresh sequence variables, $\overrightarrow{y}, \overrightarrow{z}$ are vectors of fresh variables representing the parameters, and $\Pi^n_{\mathbb{N}}$ is the numerical constraint over the fresh variables in $\overrightarrow{y} \cup \overrightarrow{z}$.

Composition: Two continuous separating summaries can be composed into one summary using the *composition operation* **ComposeList**$^\sharp_{\mathbb{G}}$ which is the reverse of decomposition. Its definition is shown as follows:

$$\mathbf{ComposeList}^\sharp_{\mathbb{G}}(\Sigma, P(\overrightarrow{x})[W_1]*P(\overrightarrow{y})[W_2], \overrightarrow{z}, W) \triangleq (\Sigma \setminus (P(\overrightarrow{x})[W_1]*P(\overrightarrow{y})[W_2]))*P(\overrightarrow{z})[W]$$

where $\overrightarrow{z}$ is a vector of fresh variables and W is a new fresh sequence variable. This composition operation briefly manipulates the shape and can not be used independently. It is applied only in the composition operation in the shape-value domain and the test on the pure constraints is required before applying.

Example 12 (Decomposing and composing summary hls)**.** An example of decomposing and composing is shown in Figure 7.3. The summary edge $\mathsf{hls}[W](X;Y)$ is split and a new node is added between X and Y by the decomposing operation $\mathbf{DecompList}^\sharp_{\mathbb{G}}$. The composition operation $\mathbf{ComposeList}^\sharp_{\mathbb{G}}$ removes the middle node connecting two summary edges.

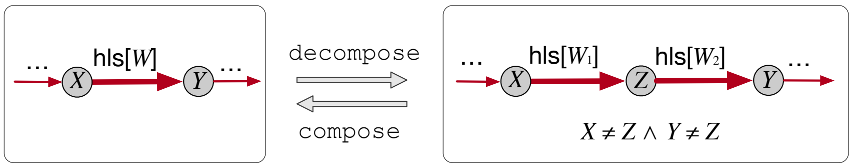


Figure 7.3: Example of $\mathbf{DecompList}^\sharp_{\mathbb{G}}$ and $\mathbf{ComposeList}^\sharp_{\mathbb{G}}$

$$\mathbf{DecompList}^\sharp_{\mathbb{G}}(\Sigma, \mathsf{hls}(X;Y)[W], W_1, W_2) =$$
$$((\Sigma \setminus \mathsf{hls}(X;Y)[W]) * (\mathsf{hls}(X;Z)[W_1] * \mathsf{hls}(Z;Y)[W_2]), X \neq Z \wedge Z \neq Y),$$
$$\mathbf{ComposeList}^\sharp_{\mathbb{G}}(\Sigma, \mathsf{hls}(X;Z)[W_1] * \mathsf{hls}(Z;Y)[W_2], W) =$$
$$(\Sigma \setminus \{\mathsf{hls}(X;Z)[W_1] * \mathsf{hls}(Z;Y)[W_2]\}) * \mathsf{hls}(X;Y)[W].$$

△

7.2.1.3 Unfolding and Folding a Chunk

Unfolding a chunk: Recall that there are some predicates, like chk, fck specifying a chunk in the heap list and free list respectively, are not summary predicates. The edges labeled by these predicates can be unfolded to materialize a chunk head edge chd when the program statements mutate the value of fields in a chunk. We define the following unfolding operation for edge chk based on its definition:

$$\mathbf{UnfoldChk}^\sharp_{\mathbb{G}}(\Sigma, \mathsf{chk}(X;Y), Z) \triangleq ((\Sigma \setminus \mathsf{chk}(X;Y)) * \mathsf{chd}(X;Z) * \mathsf{blk}(Z;Y), \Pi^n_{\mathbb{N}})$$

where Z is a fresh sequence variable and $\Pi^n_{\mathbb{N}}$ is the generated pure constraint (i.e., $(\mathcal{F}_{\texttt{size}}(X) \times \texttt{sizeof(HDR)} = Y - X)$). Unfolding a chunk only works on heap list abstraction, i.e., $\mathbf{UnfoldChk}^\sharp_{\mathbb{G}}$ only manipulates Σ_H in Σ.

Given a shape $\Sigma_H \Supset \Sigma_F$, to unfold a free chunk edge $\mathsf{fck}(X;Y)$ in Σ_F, it requires that the node X is also in Σ_H, i.e., there is an edge $\mathsf{chk}(X;Z)$ in Σ_H.

If Σ_H is summarized such that if there is no corresponding chunk is in Σ_H, then Σ_H should be decomposed (to materialize the node X) and unfolded (to materialize the edge $\mathsf{chk}(X; Z)$) first. Then we remove the edge $\mathsf{fck}(X; Y)$ in Σ_F and unfold $\mathsf{chk}(X; Z)$ in Σ_H using operation $\textbf{UnfoldChk}^\sharp_{\mathbb{G}}$. This situation is explained later in Section 7.2.4 which describes the hierarchical unfolding operation.

Folding a chunk: A sub-formula $\mathsf{chd}(X; Z) * \mathsf{blk}(Z; Y)$ in Σ_H can be folded by operation $\textbf{FoldChk}^\sharp_{\mathbb{G}}$ which is defined as follows:

$$\textbf{FoldChk}^\sharp_{\mathbb{G}}(\Sigma, \mathsf{chd}(X; Z) * \mathsf{blk}(Z; Y)) \triangleq (\Sigma \setminus \mathsf{chd}(X; Z) * \mathsf{blk}(Z; Y)) * \mathsf{chk}(X; Y)$$

Similarly, these operations over shape should work with operations on pure part when performed in the shape-value domain and can not be applied independently.

7.2.1.4 Decomposing and Composing a Memory Block

Program statements involving pointer arithmetic, e.g., an assignment `p = q + c` (where `p`, `q` are pointer variables and c is a constant), manipulate the memory at the level of blocks of bytes. The abstract transformers of these statements call a low-level abstract operation $\textbf{DecompBlk}^\sharp_{\mathbb{G}}$ which only works on the heap list level. It splits a raw memory region, specified by $\mathsf{blk}(X; Y)$ in the abstract shape Σ into two blocks and generates a new numerical constraint. $\textbf{DecompBlk}^\sharp_{\mathbb{G}}(\Sigma, \mathsf{blk}(X; Y))$ is defined using the corresponding decomposing lemma of the predicate blk:

$$\textbf{DecompBlk}^\sharp_{\mathbb{G}}(\Sigma, \mathsf{blk}(X; Y), Z) \triangleq ((\Sigma \setminus \mathsf{blk}(X; Y) * \mathsf{blk}(X; Z) * \mathsf{blk}(Z; Y), \Pi^n_{\mathbb{N}})$$

where Z is a fresh location variable and $\Pi^n_{\mathbb{N}}$ is a numerical constraint over X, Y, Z, i.e., $X \le Z \le Y$.

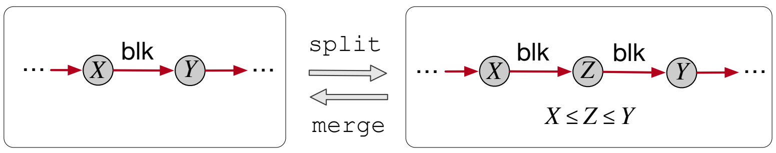


Figure 7.4: Example of $\textbf{DecompBlk}^\sharp_{\mathbb{G}}$ and $\textbf{ComposeBlk}^\sharp_{\mathbb{G}}$

The reverse operation of $\mathbf{DecompBlk}^{\sharp}_{\mathbb{G}}$ is the composing operation $\mathbf{ComposeBlk}^{\sharp}_{\mathbb{G}}$ which merges two contiguous blocks. This operation is often used in analysing SDMA algorithms with eager coalescing policy because they often coalesce adjacent memory chunks. An example of $\mathbf{DecompBlk}^{\sharp}_{\mathbb{G}}$ and $\mathbf{ComposeBlk}^{\sharp}_{\mathbb{G}}$ is shown in Figure 7.4.

7.2.2 Abstract Operations in Data Words Domain $\mathbb{W}^{\sharp}$

7.2.2.1 Unfolding Operation

The unfolding operation, denoted by $\mathbf{Unfold}^{\sharp}_{\mathbb{W}}$, in the data words domain shall collaborate with the unfolding operation provided by the shape domain, and there is an important restriction: *unfolding operation* $\mathbf{Unfold}^{\sharp}_{\mathbb{W}}$ *only unfolds a sequence variable in a sequence constraint which is a part of a well-formed shape-value.*

The operation $\mathbf{Unfold}^{\sharp}_{\mathbb{W}}(\Pi_{\mathbb{W}}, W, x, W')$ takes the sequence variable W bound to the summary predicate needed to be unfolded in the sequence constraint $\Pi_{\mathbb{W}}$, then replaces a sequence W in $\Pi_{\mathbb{W}}$ with the concatenation of its head and its tail. The head of w is represented by $[x]$ and its tail is by W' where x and W' are fresh variables. We denote by "_" in the returned value by $\mathbf{Unfold}^{\sharp}_{\mathbb{W}}$ if $\mathbf{Unfold}^{\sharp}_{\mathbb{W}}$ does not return any new numerical constraints. The unfolding operation is defined as follows:

$$\mathbf{Unfold}^{\sharp}_{\mathbb{W}}(\Pi_{\mathbb{W}}, W, x, W') \triangleq \begin{cases} (\_\,, \mathbf{Proj}^{\sharp}_{\mathbb{W}}(\Pi_{\mathbb{W}}, \{W\}) & \text{if } W = \epsilon \\ (\_\,, \Pi_{\mathbb{W}}) & \text{if } W = [X] \\ (\Pi^{n}_{\mathbb{N}}, \Pi^{n}_{\mathbb{W}}) & \text{otherwise} \end{cases}$$

where $\mathbf{Proj}^{\sharp}_{\mathbb{W}}$ is the projection operation. When w is the empty term, the returned value of $\mathbf{Unfold}^{\sharp}_{\mathbb{W}}$ is a sequence constraint $\mathbf{Proj}^{\sharp}_{\mathbb{W}}(\Pi_{\mathbb{W}}, \{W\})$ which is the result of projecting out W in $\Pi_{\mathbb{W}}$. And when w is a singleton, then the unfolding returns the original sequence constraint. In the third case, the new constraints $(\Pi^{n}_{\mathbb{N}}, \Pi^{n}_{\mathbb{W}})$ are obtained by

1. creating a numerical constraint $\Pi^{n}_{\mathbb{N}}$ on the head of W (represented by x) obtained from A_U in each universal constraint on W, i.e., adding $A_U(x)$ if $A_G(x)$ is true;

2. replacing $W_L = w$ ($L \in \{H, F\}$) in $\Pi_{\mathbb{W}}$ with a new sequence constraint $W_L = w[[x].W'/W]$;

3. transferring all the universal constraints on W to W' in $\Pi_{\mathbb{W}}$.

The operation which unfolds a sequence W at the end and generates a sequence constraint of the form $W = W'.[x]$ is defined similarly.

7.2.2.2 Splitting Operation

The transformers for the decomposition of summary predicates generate sequence constraints of the form $W = W_1.W_2$. This constraint is used to transform the associated data word abstract value. We define the corresponding abstract transformer for splitting a sequence in the data words domain.

The transformer $\textbf{Split}^{\sharp}_{\mathbb{W}}(\Pi_{\mathbb{W}}, W, W_1, W_2)$ splits a sequence in w into two parts represented by the fresh sequence variables, W_1 and W_2. It is defined as follows:

$$\textbf{Split}^{\sharp}_{\mathbb{W}}(\Pi_{\mathbb{W}}, W, W_1, W_2) \triangleq \begin{cases} \textbf{Proj}^{\sharp}_{\mathbb{W}}(\Pi_{\mathbb{W}}, \{W\}) & \text{if } w = \epsilon \\ \Pi_{\mathbb{W}} & \text{if } w = [X] \\ \Pi^{n}_{\mathbb{W}} & \text{otherwise} \end{cases}$$

where in the third case the new sequence constraint $\Pi^{n}_{\mathbb{W}}$ is obtained by

1. replacing $W_L = w$ ($L \in \{H, F\}$) in $\Pi_{\mathbb{W}}$ with a new sequence constraint $W_L = w[W_1.W_2/W]$;
2. adding universal quantified sequence constraints on W_1 and W_2 computed from the universal constraints of W for each guard $A_G \in \mathcal{G}$, i.e., $\forall X \in W_1 \cdot A_G \Rightarrow A_U$, $\forall X \in W_2 \cdot A_G \Rightarrow A_U$.

7.2.2.3 Concatenation Operation

Given a sequence constraint $\Pi_{\mathbb{W}}$ with two sequences represented by two sequence variables W_1, W_2, suppose for each guard form $A_{G_i} \in \mathcal{G}$ they are constrained by the following universal constraints:

$$\forall X \in W_1 \cdot A_{G_i} \Rightarrow A_{U_i}, \forall X \in W_2 \cdot A_{G_i} \Rightarrow A_{U_j}.$$

If W_1 or W_2 is a singleton, i.e., $[X]$, then for each guard $A_{G_i} \in \mathcal{G}$, if $\Pi_{\mathbb{W}} \Rightarrow A_{G_i}(X)$, then we propagate the following universal constraints over the singleton in $\Pi_{\mathbb{W}}$:

$$\forall y \in [X], A_{G_i} \Rightarrow A_{U_k}$$

W_1 and W_2 can be concatenated by using the concatenation operation $\textbf{Concat}^\sharp_{\mathbb{W}}(\Pi_{\mathbb{W}}, W_1, W_2, W)$ where W is a fresh sequence variable. It generates a new sequence constraint which is obtained by

1. adding a new universal constraint over W, i.e., $\bigwedge_{A_{G_i} \in \mathcal{G}} \forall X \in W_1 \cdot A_{G_i} \Rightarrow A_{U_i} \sqcup^{\mathcal{N}} A_{U_j}$;
2. replacing $W_L = w$ in $\Pi_{\mathbb{W}}$ with a new sequence constraint $W_L = w[W/W_1.W_2]$;
3. projecting out W_1 and W_2 in $\Pi_{\mathbb{W}}$ by calling operation $\textbf{Proj}^\sharp_{\mathbb{W}}$.

7.2.3 Abstract Operations in Shape-value Domain $\mathbb{H}^\sharp$

Because the values in the shape-value domain $\mathbb{H}^\sharp$ contain elements in the shape domain $\mathbb{G}^\sharp$ and data words domain $\mathbb{W}^\sharp$, most of abstract operations in $\mathbb{H}^\sharp$ require careful coordination between operations provided by $\mathbb{G}^\sharp$ and $\mathbb{W}^\sharp$.

7.2.3.1 Unfolding and Folding Summary

Unfolding: The unfolding operation over summary predicates in the shape-value domain $\mathbb{H}^\sharp$ takes an element $(\Sigma \rightrightarrows \Pi)$ and yields a disjunction of such elements $(\Sigma_0 \rightrightarrows \Pi_0) \vee (\Sigma_1 \rightrightarrows \Pi_1) ... \vee (\Sigma_n \rightrightarrows \Pi_n)$. It uses the unfolding operation $\textbf{UnfoldList}^\sharp_{\mathbb{G}}$ to manipulate the shape and the unfolding operation $\textbf{Unfold}^\sharp_{\mathbb{W}}$ to unfold the data word. The operation $\textbf{UnfoldList}^\sharp_{\mathbb{G}}$ cooperates with the operation $\textbf{Unfold}^\sharp_{\mathbb{W}}$. We denote by $\textbf{UnfoldList}^\sharp_{\mathbb{H}}$ the unfolding operation in the shape-value domain $\mathbb{H}^\sharp$. It returns a new shape which is the result of unfolding of the shape and a new numerical constraint and a new sequence constraint obtained from unfolding of the data word. The unfolding operation $\textbf{Unfold}_{\mathbb{H}}$ is defined as follows:

$$\textbf{UnfoldList}^\sharp_{\mathbb{H}}((\Sigma, \Pi_{\mathbb{N}}, \Pi_{\mathbb{W}}, P(\overrightarrow{x})[W], X, W') \triangleq \bigvee_{0 \le i \le n} (\Sigma_i, \Pi_{\mathbb{N}} \sqcap^{\mathcal{N}} \Pi_{\mathbb{N}_i} \sqcap^{\mathcal{N}} \Pi^n_{\mathbb{N}_i}, \Pi^n_{\mathbb{W}_i})$$

where

$$\begin{aligned} \bigvee_{0 \le i \le n} (\Sigma_i, \Pi_{\mathbb{N}_i}, \Pi_{\mathbb{W}_i}) &= \textbf{UnfoldList}^\sharp_{\mathbb{G}}(\Sigma, P(\overrightarrow{x})[W], X, W'), \\ \bigvee_{0 \le i \le n} (\Pi^n_{\mathbb{N}_i}, \Pi^n_{\mathbb{W}_i}) &= \textbf{Unfold}^\sharp_{\mathbb{W}}(\Pi_{\mathbb{W}}, W, X, W'). \end{aligned}$$

Each sequence constraint $\Pi^n_{\mathbb{W}_i}$ returned by $\textbf{Unfold}^\sharp_{\mathbb{W}}$ contains the sequence constraint $\Pi_{\mathbb{W}_i}$ generated by $\textbf{UnfoldList}^\sharp_{\mathbb{G}}$.

Example 13 (Unfolding a heap list)**.** We show an example of unfolding a heap list hls in Figure 7.5. The first disjunct in the unfolded value is the case when the shape is unfolded into emp. In this case, the universal constraints over W_H is removed. The second disjunct contains an unfolded shape $\mathsf{chk}(A; B) * \mathsf{hls}(B; \mathsf{hli})[W']$ and the new pure constraints. $\triangle$

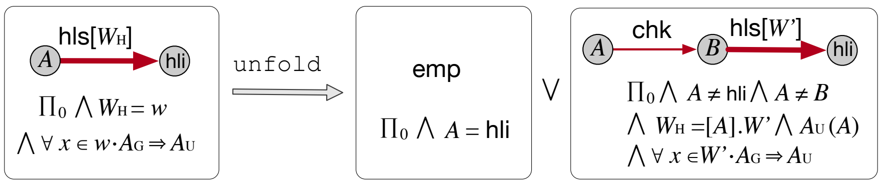

hls[W_H]
A
hli
$\Pi_0 \wedge W_H = w$
$\wedge \forall x \in w \cdot A_G \Rightarrow A_U$
unfold
emp
$\Pi_0 \wedge A = \mathsf{hli}$
$\vee$
chk
hls[W']
A
B
hli
$\Pi_0 \wedge A \neq \mathsf{hli} \wedge A \neq B$
$\wedge W_H = [A].W' \wedge A_U(A)$
$\wedge \forall x \in W' \cdot A_G \Rightarrow A_U$

Figure 7.5: Example of unfolding a heap list

Theorem 7.1 (Soundness of unfolding $\mathbf{UnfoldList}^\sharp_{\mathbb{H}}$)**.** *If* $\mathbf{UnfoldList}^\sharp_{\mathbb{H}}$ *transforms* $(\Sigma, \Pi_{\mathbb{N}}, \Pi_{\mathbb{W}})$ *into a finite number of disjuncts* $(\Sigma_0, \Pi_{\mathbb{N}_0}, \Pi_{\mathbb{W}_0}) \vee (\Sigma_1, \Pi_{\mathbb{N}_1}, \Pi_{\mathbb{W}_1}) \ldots \vee (\Sigma_n, \Pi_{\mathbb{N}_n}, \Pi_{\mathbb{W}_n})$*, then* $\gamma_{\mathbb{H}}(\Sigma, \Pi_{\mathbb{N}}, \Pi_{\mathbb{W}}) \subseteq \bigcup_{0 \le i \le n} \gamma_{\mathbb{H}}(\Sigma_i, \Pi_{\mathbb{N}_i}, \Pi_{\mathbb{W}_i})$. $\blacksquare$

Folding: Given a shape-value $(\Sigma \rightrightarrows \Pi)$ and a sub-formula $p(\overrightarrow{x}) * P(\overrightarrow{v})[W]$ in Σ which should be folded, $\mathbf{FoldList}^\sharp_{\mathbb{H}}$ calls $\mathbf{FoldList}^\sharp_{\mathbb{G}}$ to fold the shape and $\mathbf{Concat}^\sharp_{\mathbb{W}}$ to concatenate words. Precisely, $p(\overrightarrow{x})$ is a chunk, could be $\mathsf{chk}(X; Y)$ or $\mathsf{fck}(X; Y)$ and the node Y should be the start of the summary edge specified by $P(Y, \overrightarrow{y})[W]$. $\mathbf{Concat}^\sharp_{\mathbb{W}}$ in fact concatenates the singleton $[X]$ and the sequence W. Recall that the transformation lemma specified as follows:

$$p(X; Y) * P(Y, \vec{y})[W] \wedge (\exists \overrightarrow{u} \cdot \Pi^\Delta_{\mathbb{N}} \wedge \Pi^\Delta_{\mathbb{W}}) \Rightarrow P(X, \vec{y})[W']$$

where $\Pi^\Delta_{\mathbb{N}}$ and $\Pi^\Delta_{\mathbb{W}} = (W' = [X].W)$ are numerical and sequence constraints the same as constraints in the definition of predicate P. Evidently, the folding operation can be applied to fold $p(X; Y) * P(Y, \vec{y})[W]$ in the value (Σ, Π) only if the condition is satisfied, i.e., $\Pi \Rightarrow \Pi^\Delta_{\mathbb{N}} \wedge \Pi^\Delta_{\mathbb{W}}$. The definition of $\mathbf{FoldList}^\sharp_{\mathbb{H}}$ is given as follows:

$$\mathbf{FoldList}^\sharp_{\mathbb{H}}((\Sigma, \Pi_{\mathbb{N}}, \Pi_{\mathbb{W}}), p(X; Y) * P(Y, \overrightarrow{y})[W], W') \triangleq (\Sigma^n, \Pi^n_{\mathbb{N}}, \Pi^n_{\mathbb{W}})$$

where

$$\begin{array}{rcl} \Sigma^n & = & \mathbf{FoldList}^\sharp_{\mathbb{G}}(\Sigma, p(X; Y) * P(Y, \overrightarrow{y})[W], W'), \\ \Pi^n_{\mathbb{N}} & = & \mathbf{Proj}^\sharp_{\mathbb{N}}(\Pi_{\mathbb{N}}, \{Y\}), \\ \Pi^n_{\mathbb{W}} & = & \mathbf{Concat}^\sharp_{\mathbb{W}}(\Pi_{\mathbb{W}}, [X], W, W'). \end{array}$$

Σ^n is the new shape returned by $\mathbf{FoldList}^\sharp_{\mathbb{G}}$, $\Pi^n_{\mathbb{N}}$ is the new numerical constraint which is the result of projecting out Y in $\Pi_{\mathbb{N}}$, and $\Pi^n_{\mathbb{W}}$ is the new sequence constraint returned by $\mathbf{Concat}^\sharp_{\mathbb{W}}$. Precisely, the sub-formula $p(X;Y) * P(Y,\overrightarrow{y})[W]$ is folded as a new summary $P(X,\overrightarrow{y})[W']$ in Σ where W' is the fresh variable. The fresh sequence variable W' is also used to represent the concatenation when calling operation $\mathbf{Concat}^\sharp_{\mathbb{W}}$.

7.2.3.2 Decomposing and Composing Summary

The decomposition operation in the shape-value domain $\mathbb{H}^\sharp$ is defined using the decomposition operation $\mathbf{DecompList}^\sharp_{\mathbb{G}}$ provided by the shape domain $\mathbb{G}^\sharp$ and the splitting operation $\mathbf{Split}^m_{\mathbb{W}}$ of $\mathbb{W}^\sharp$. Its definition is given as follow:

$$\mathbf{DecompList}^\sharp_{\mathbb{H}}((\Sigma,\Pi_{\mathbb{N}},\Pi_{\mathbb{W}}),P(\overrightarrow{x})[W],W_1,W_2) \triangleq (\Sigma^n,\Pi_{\mathbb{N}}\sqcap^{\mathcal{N}}\Pi^n_{\mathbb{N}}\sqcap^{\mathcal{N}}\Pi^m_{\mathbb{N}},\Pi_{\mathbb{W}}\sqcap^{\mathcal{W}}\Pi^m_{\mathbb{W}})$$

where

$$\begin{array}{lcl}
(\Sigma^n,\Pi^n_{\mathbb{N}},\Pi^n_{\mathbb{W}}) & = & \mathbf{DecompList}^\sharp_{\mathbb{G}}(\Sigma,P(\overrightarrow{x})[W],W_1,W_2),\\
(\Pi^m_{\mathbb{N}},\Pi^m_{\mathbb{W}}) & = & \mathbf{Split}^\sharp_{\mathbb{W}}(\Pi_{\mathbb{W}},W,W_1,W_2).
\end{array}$$

Similarly, the composition operation $\mathbf{ComposeList}^\sharp_{\mathbb{H}}$ in the domain $\mathbb{H}^\sharp$ is defined using the composition operation $\mathbf{ComposeList}^\sharp_{\mathbb{G}}$ in $\mathbb{G}^\sharp$ and the concatenation operation $\mathbf{Concat}^\sharp_{\mathbb{W}}$ in $\mathbb{W}^\sharp$. Let us fix an abstract value $(\Sigma,\Pi_{\mathbb{N}},\Pi_{\mathbb{W}})$ where there are two separated segments in Σ, denoted by $P(\overrightarrow{x})[W_1] * P(\overrightarrow{y})[W_2]$, and $\Pi_{\mathbb{W}}$ contains a sequence constraint with the subword $W_1.W_2$. Recall that the transformation lemma specified as follows:

$$P(\overrightarrow{x})[W_1] * P(\vec{y})[W_2] \wedge (\exists\overrightarrow{u}\cdot\Pi^\Delta_{\mathbb{N}} \wedge \Pi^\Delta_{\mathbb{W}}) \Rightarrow P(\vec{z})[W]$$

where $\Pi^\Delta_{\mathbb{N}}$ and $\Pi^\Delta_{\mathbb{W}} = (W = W_1.W_2)$ are numerical and sequence constraints. The composition operation can be applied if $\Pi_{\mathbb{N}} \wedge \Pi_{\mathbb{W}} \Rightarrow \Pi^\Delta_{\mathbb{N}} \wedge \Pi^\Delta_{\mathbb{W}}$. The composition operation is defined as follows:

$$\mathbf{ComposeList}^\sharp_{\mathbb{H}}((\Sigma,\Pi_{\mathbb{N}},\Pi_{\mathbb{W}}),P(\overrightarrow{x})[W_1] * P(\overrightarrow{y})[W_2],W) \triangleq (\Sigma^n,\Pi^n_{\mathbb{N}},\Pi^n_{\mathbb{W}})$$

where

$$\begin{array}{lcl}
\Sigma^n & = & \mathbf{ComposeList}^\sharp_{\mathbb{G}}(\Sigma,P(\overrightarrow{x})[W_1] * P(\overrightarrow{y})[W_2],W),\\
\Pi^n_{\mathbb{N}} & = & \mathbf{Proj}^\sharp_{\mathbb{N}}(\Pi_{\mathbb{N}},\{\overrightarrow{x},\overrightarrow{y}\}) \wedge \Pi_{\mathbb{N}}(\overrightarrow{z}),\\
\Pi^n_{\mathbb{W}} & = & \mathbf{Concat}^\sharp_{\mathbb{W}}(\Pi_{\mathbb{W}},W_1,W_2,W),
\end{array}$$

and $\overrightarrow{z}$ and W are fresh variables in the new composed segment.

7.2.3.3 Unfolding and Folding Chunk

The unfolding operation over a heap chunk in the shape-value domain is denoted by $\mathbf{UnfoldChk}^\sharp_{\mathbb{H}}$ which only works on the heap list abstraction. It manipulates the shape and generates numerical constraint by calling operation $\mathbf{UnfoldChk}^\sharp_{\mathbb{G}}$. Given an abstract value $(\Sigma, \Pi_{\mathbb{N}}, \Pi_{\mathbb{W}})$ and the chunk $\mathsf{chk}(X;Y)$ in Σ that should be unfolded, the definition of $\mathbf{UnfoldChk}^\sharp_{\mathbb{H}}$ is defined as follows:

$$\mathbf{UnfoldChk}^\sharp_{\mathbb{H}}((\Sigma, \Pi_{\mathbb{N}}, \Pi_{\mathbb{W}}), \mathsf{chk}(X;Y), Z) \triangleq (\Sigma^n, \Pi_{\mathbb{N}} \sqcap^{\mathcal{N}} \Pi^n_{\mathbb{N}}, \Pi_{\mathbb{W}})$$

where

$$(\Sigma^n, \Pi^n_{\mathbb{N}}) = \mathbf{UnfoldChk}^\sharp_{\mathbb{G}}(\Sigma, \mathsf{chk}(X;Y), Z)$$

and Z is a fresh variable. The value returned by $\mathbf{UnfoldChk}^\sharp_{\mathbb{H}}$ is composed of 1). the new shape Σ^n, 2). a meet between the original numerical constraint and the new numerical constraint $\Pi^n_{\mathbb{N}}$ returned by $\mathbf{UnfoldChk}^\sharp_{\mathbb{G}}$, and 3). the original sequence constraint. The procedure of unfolding free chunk $\mathbf{UnfoldFck}^\sharp_{\mathbb{H}}$ is defined similarly to the operation $\mathbf{UnfoldFck}^\sharp_{\mathbb{G}}$.

Given a shape-value $((\Sigma_H \Supset \Sigma_F) \rightrightarrows \Pi)$ and $\mathsf{chd}(X;Z) * \mathsf{blk}(Z;Y)$ is a sub-formula in Σ_H, if Π satisfies the data constraints in the definition of chk, i.e., $\Pi \Rightarrow \mathcal{F}_{\mathtt{size}}(X) \times \mathtt{sizeof(HDR)} = Y - X)$, then $\mathsf{chd}(X;Z) * \mathsf{blk}(Z;Y)$ can be folded as a chunk $\mathsf{chk}(X;Y)$ in Σ_H by using the operation $\mathbf{FoldChk}^\sharp_{\mathbb{H}}$ which is defined as follows:

$$\mathbf{FoldChk}^\sharp_{\mathbb{H}}((\Sigma, \Pi_{\mathbb{N}}, \Pi_{\mathbb{W}}), \mathsf{chd}(X;Z) * \mathsf{blk}(Z;Y)) \triangleq (\Sigma^n, \Pi^n_{\mathbb{N}}, \Pi_{\mathbb{W}})$$

where

$$\begin{array}{rcl} \Sigma^n & = & \mathbf{FoldChk}^\sharp_{\mathbb{G}}(\Sigma, \mathsf{chd}(X;Z) * \mathsf{blk}(Z;Y)), \\ \Pi^n_{\mathbb{N}} & = & \mathbf{Proj}^\sharp_{\mathbb{N}}(\Pi_{\mathbb{N}}, \{Z\}). \end{array}$$

Notice that $\Pi_{\mathbb{W}}$ is not changed and the new numerical constraint is obtained by projecting out Z in $\Pi_{\mathbb{N}}$.

In fact, the $\mathsf{chd}(X;Z) * \mathsf{blk}(Z;Y)$ can be folded as a free chunk $\mathsf{fck}(X;Y)$ if the pure constraints in fck's definition are satisfied. The free chunk is added in Σ_F. This situation is explained in Section 7.2.5 which describes the hierarchical folding operation.

7.2.3.4 Decomposing and Composing a Memory Block

In the shape-value domain, the decomposing operation $\mathbf{DecompBlk}^\sharp_{\mathbb{H}}$ updates the shape value and the numerical constraint. The new shape is the result of $\mathbf{DecompBlk}^\sharp_{\mathbb{G}}$ and the new numerical constraint is the meet between the old numerical constraint and the constraint returned by $\mathbf{DecompBlk}^\sharp_{\mathbb{G}}$. The definition of $\mathbf{DecompBlk}^\sharp_{\mathbb{H}}$ is shown as follows:

$$\mathbf{DecompBlk}^\sharp_{\mathbb{H}}((\Sigma, \Pi_{\mathbb{N}}, \Pi_{\mathbb{W}}), \mathsf{blk}(X; Y), Z) \triangleq (\Sigma^n, \Pi_{\mathbb{N}} \sqcap^{\mathcal{N}} \Pi^n_{\mathbb{N}}, \Pi_{\mathbb{W}})$$

where $(\Sigma^n, \Pi^n_{\mathbb{N}}) = \mathbf{DecompBlk}^\sharp_{\mathbb{G}}(\Sigma, \mathsf{blk}(X; Y), Z)$. The composition operation is defined similarly:

$$\mathbf{ComposeBlk}^\sharp_{\mathbb{H}}((\Sigma, \Pi_{\mathbb{N}}, \Pi_{\mathbb{W}}), \mathsf{blk}(X; Y) * \mathsf{blk}(Y; Z)) \triangleq (\Sigma^n, \Pi^n_{\mathbb{N}}, \Pi_{\mathbb{W}})$$

where

$$\begin{array}{rcl} \Sigma^n & = & \mathbf{ComposeBlk}^\sharp_{\mathbb{G}}(\Sigma, \mathsf{blk}(X; Y) * \mathsf{blk}(Y; Z)), \\ \Pi^n_{\mathbb{N}} & = & \mathbf{Proj}^\sharp_{\mathbb{N}}(\Pi_{\mathbb{N}}, \{Y\} \wedge X < Z. \end{array}$$

7.2.4 Hierarchical Unfolding

Order of predicates: Given a summary $\Sigma_H \Supset \Sigma_F$ and the information we want to expose (e.g., the access of a field), the analysis needs to decide on which level of abstraction the unfolding should be performed. For example, for a summary atom representing a free list segment, the read access to the `size` field requires unfolding the summary at the free list level to obtain a free chunk. However, if the `size` field is written, the free chunk fck shall be instantiated at the heap list level into a chunk and finally the chunk decomposed to obtain the chunk header storing the `size` field.

We define the following partial order $\prec_P$ between predicates $\mathsf{blk} \prec_P \mathsf{chd} \prec_P \mathsf{chk} \prec_P \mathsf{fck} \prec_P \{\mathsf{hls}, \mathsf{hlsc}, \mathsf{fls}, \mathsf{flso}\}$ which intuitively corresponds to an increasing degree of specialisation. We denote by $Q \preceq_P P$ if $Q \prec_P P$ or $Q = P$.

For each program statement $stmt$ and each pointer variable `x` in $stmt$, an atom $P(X; \ldots)$ with $\epsilon^\sharp(\mathrm{x}) = X$ is transformed using the unfolding transformers to obtain the atom $Q(X; \ldots)$ such that Q is the maximal predicate satisfying $Q \preceq_P P$ and:

- if $stmt$ reads in `x` the fields of `HDR`, then $Q \preceq_P \mathsf{fck}$,
- if $stmt$ assigns `x.isfree` or `x.fnx`, then $Q \preceq_P \mathsf{chk}$,
- if $stmt$ mutates `x` using pointer arithmetics or assigns `x.size`, then $Q \preceq_P \mathsf{chd}$.

Hierarchical unfolding: We illustrate the hierarchical unfolding, $\textbf{Unfold}^\sharp_{\mathbb{H}}$, by using the program in Figure 6.6(b), the procedure of allocation. For simplicity, we omit the pure constraints in the presented values. Before the traversal loop of free list (at line 34), the abstract graph is given as a summary, i.e., the heap list and the free list are described by hlsc and flso respectively, as shown in the first frame in Figure 7.6.

The procedure aims to find a suitable free chunk by traversing the free list. During the first iteration, the summary edge flso is unfolded to materialise a chunk to which the pointer `nxt` points using $\textbf{UnfoldList}^\sharp_{\mathbb{H}}$ operation as shown in the 2nd frame in Figure 7.6. The edge $\mathsf{fck}(X; X')$ is also enough for statements reading the fields of the chunk X. When the size of X does not satisfy the condition, the pointer `nxt` points to the next free chunk by executing the assignment `nxt = nxt->fnx`, i.e., `nxt` points to X', the start of the summary $\mathsf{flso}(X', ...)[W_1]$. Then the summary $\mathsf{flso}(X', ...)[W_1]$ is unfolded to materialise a chunk. The abstract shape is in the 3rd frame shown in Figure 7.6.

We assume that in the i-th iteration (4th frame), the pointer `prv` points to the chunk Y and the pointer `nxt` points to Z which is the start of the summary $\mathsf{flso}(Z, ...)[W_3]$ in Σ_F. The free chunks between X and Y are folded as $\mathsf{flso}(X, ...)[W_2]$. The next step is to read the field `size` of Z. It requires to unfold the summary $\mathsf{flso}(Z, ...)[W_3]$. Thus, $\mathsf{flso}(Z, ...)[W_3]$ is split into two separated parts, i.e., $\mathsf{fck}(Z; Z') * \mathsf{flso}(Z', ...)[W_4]$.

Let us assume that the size of chunk Z is larger than `nunits` (5th frame). Then the next step is to compute the post-image of the next statement at line 38, i.e., `nxt->size-=nunits`. The symbolic location Z shall be the root a chd predicate. Thus, the free chunk at $\mathsf{fck}(Z, Z')$ is instantiated in the heap list by

1. decomposing and then unfolding the summary edge hls using $\textbf{DecompList}^\sharp_{\mathbb{H}}$ and $\textbf{UnfoldList}^\sharp_{\mathbb{H}}$, and by
2. unfolding the chunk chk to materialise the chd edge using the operation $\textbf{UnfoldChk}^\sharp_{\mathbb{H}}$.

The unfolding of chk requires to remove the fck atom from Z in the free list because its definition is not more satisfied at the free list abstraction level.

The next assignment, `nxt+=nxt->size` involving pointer arithmetic, does not require to transform the predicate rooted in Z because it is already chd.

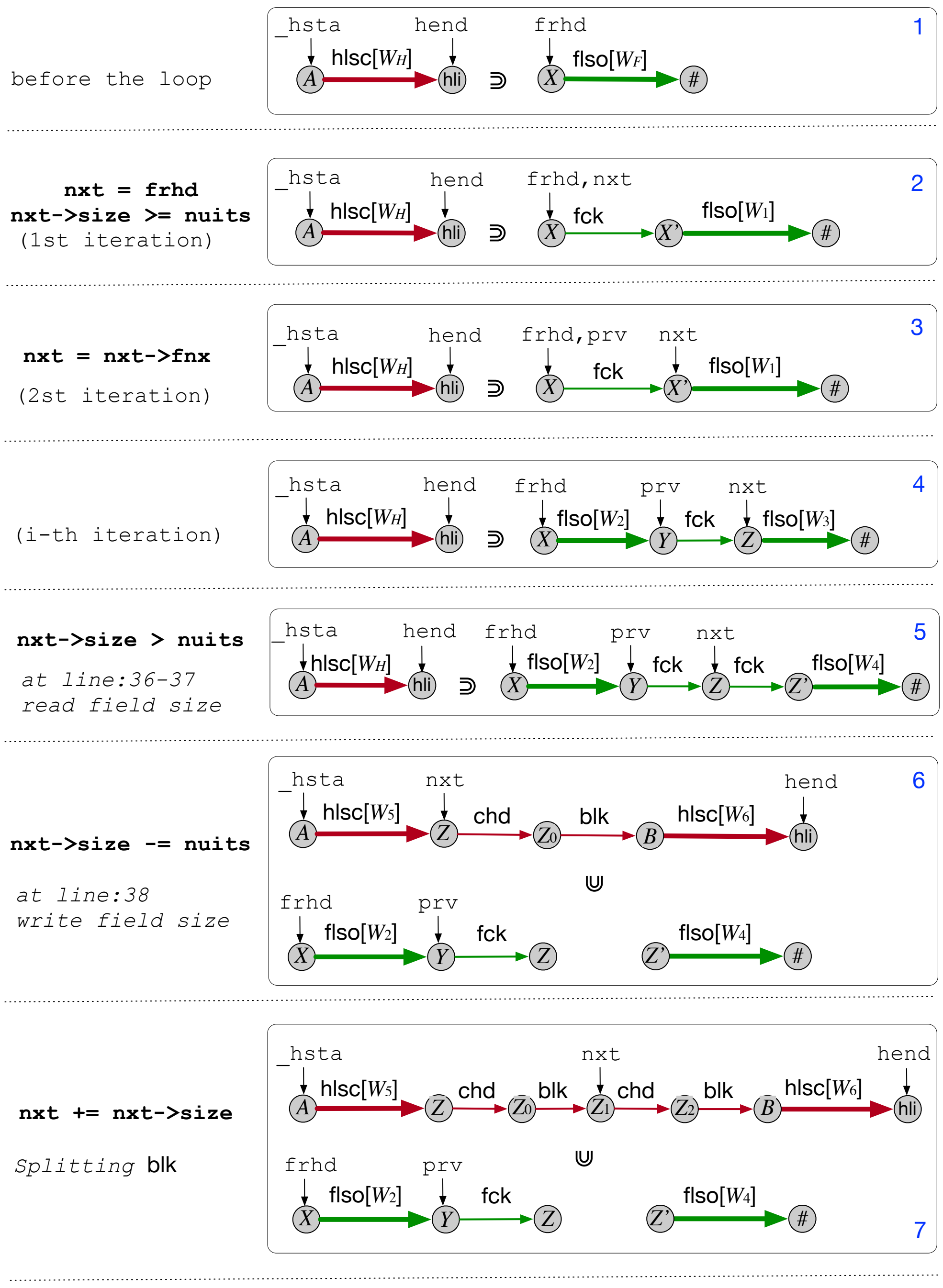


Figure 7.6: Hierarchical unfolding at line 38

Instead, the transformer adds a new symbolic location Z_1 in the heap list level by using $\mathbf{DecompBlk}^{\sharp}_{\mathbb{G}}$ operation.

If Z_1 goes beyond the limit of the block of the chunk starting at Z (i.e., outside the interval $[Z_0, B)$ in Figure 7.6), the analysis signals a chunk breaking. Otherwise, the blk atom from X is split using decomposing operation $\mathbf{DecompBlk}^{\sharp}_{\mathbb{G}}$ to insert Z_1. The result is given in the top part of Figure 7.7. The abstract transformer for the assignment involving pointer arithmetic is detailed in Section 7.3.1.

7.2.5 Hierarchical Folding

To reduce the size of abstract values, the abstract transformers finish their computation on an abstract value $(\epsilon^{\sharp}, \Sigma, \Pi)$ by eliminating the symbolic locations which are anonymous nodes in Σ. The elimination uses folding and composing operations to replace sub-formulas using these variables by one predicate atom. The graph representation eases the computation of sub-formulas matching the left part of a folding operations. More precisely, the elimination process ($\mathbf{Fold}^{\sharp}_{\mathbb{H}}$) has the following steps.

First, it searches sequences of sub-formula of the form $\mathsf{chd}(X_0; X_1) * \mathsf{blk}(X_1; X_2) * \ldots * \mathsf{blk}(X_{n-1}; X_n)$ where none of X_i $(1 \leq i < n)$ is in $\mathrm{img}(\epsilon^{\sharp})$. Such sub-formulas are folded into $\mathsf{chk}(X_0; X_n)$ if the pure part of the abstract value implies $X_0\texttt{.size} \times \texttt{sizeof(HDR)} = X_n - X_0$ (see Table 5.2) using $\mathbf{ComposeBlk}^{\sharp}_{\mathbb{H}}$ and $\mathbf{FoldChk}^{\sharp}_{\mathbb{H}}$ operations. We use the variable elimination ($\mathbf{Proj}^{\sharp}_{\mathbb{N}}$) provided by the numerical domain $\mathbb{N}^{\sharp}$ to project out $\{X_1, \ldots, X_{n-1}\}$ from the pure part. An example is shown in the top of Figure 7.7, i.e., the shape value is obtained by folding $\mathsf{chd}(Z; Z_0) * \mathsf{blk}(Z_0; Z_1)$ (a sub-formula in the value shown at the bottom of Figure 7.6) into a chunk denoted by $\mathsf{fck}(Z; Z_1)$.

Furthermore, if a chunk $\mathsf{chk}(X; Y)$ is in Σ_H and the pure part implies $\mathcal{F}_{\texttt{isfree}}(X) = 1$, then the chunk atom (and its start address) is promoted as fck to the free list level. That means a free chunk edge should be added into Σ_F. An example is shown in Figure 7.6 (in the 2nd frame), the free chunk $\mathsf{fck}(Z, Z')$ is added in Σ_F.

The next step of the hierarchical folding is to search sequences of sub-formula in Σ_H of the form $\mathsf{chk}(X_0; X_1) * \mathsf{chk}(X_1; X_2) * \ldots * \mathsf{chk}(X_{n-1}; X_n)$ where none of X_i $(1 \leq i < n)$ is in $\mathrm{img}(\epsilon^{\sharp})$ that is X_i is an anonymous node. By using $\mathbf{FoldList}^{\sharp}_{\mathbb{H}}$ operation, such sub-formulas are folded into summary edges

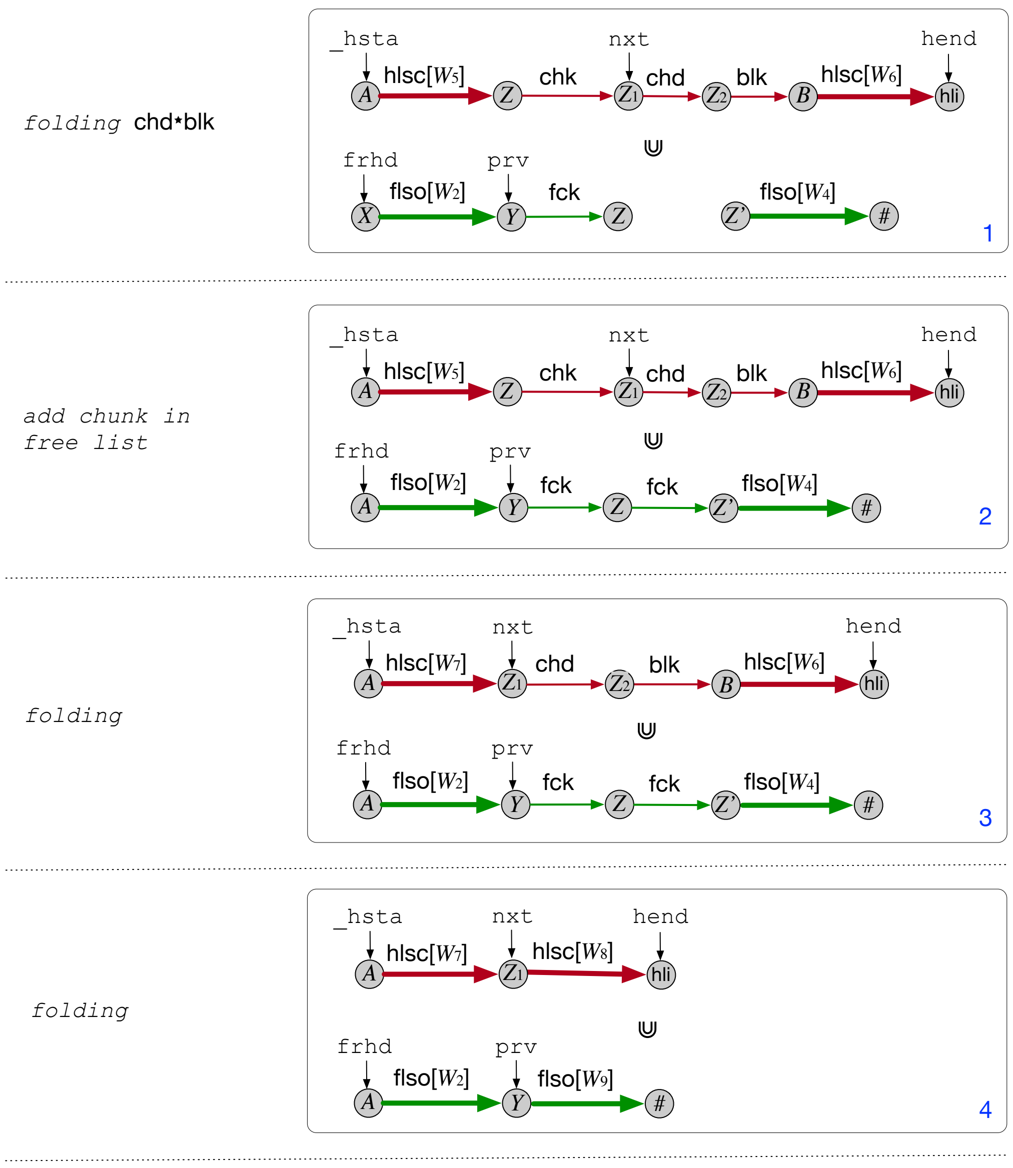


Figure 7.7: Hierarchical folding after line 38

$P_H(X_0,...)[W_i]$ where P_H could be hls or hlsc depending on the type of heap list used in the SDMA and the pure constraints.

Then, hierarchical folding searches the sub-formula of the form $\mathsf{chk}(X_0;X_1) * P_H(X_1,...)[W]$ (or $P_H(...,X_0)[W] * \mathsf{chk}(X_0;X_1)$ in Σ_H) and searches the sub-formula of the form $\mathsf{fck}(X_0;X_1) * P_F(X_1,...)[W]$ (or $P_F(...,X_0)[W] * \mathsf{fck}(X_0;X_1)$) in Σ_F where P_F is fls or flso and checks the pure constraints in pure part. By using the composition operation $\textbf{ComposeList}^\sharp_{\mathbb{H}}$ and concatenation operation $\textbf{Concat}^\sharp_{\mathbb{W}}$, such sub-formulas are folded into a totally summarized value. An example is shown in Figure 7.7, the value in the third frame is obtained by unfolding chunk $\mathsf{chk}(Z;Z_1)$ inside the segment $\mathsf{hlsc}(A,...,Z)[W_7]$. The final shape shown in the 4th frame is obtained after performing unfolding in heap list and free list.

7.3 Abstract Transformers

In this section, we define the abstract transformers for assignments and condition tests. Table 7.1 summaries the basic abstract operations provided by the domains which are used in the following content.

Numerical domain $\mathbb{N}^\sharp$

$\textbf{Guard}^\sharp_{\mathbb{N}}$	:	$\mathbb{Bexp} \times \mathbb{N}^\sharp \to \mathbb{N}^\sharp$
$\textbf{Assign}^\sharp_{\mathbb{N}}$	:	$\mathsf{LVar} \times \mathbb{V} \times \mathbb{N}^\sharp \to \mathbb{N}^\sharp$
$\textbf{Proj}^\sharp_{\mathbb{N}}$	:	$\mathbb{N}^\sharp \times \mathcal{P}(\mathsf{LVar}) \to \mathbb{N}^\sharp$

Data words domain $\mathbb{W}^\sharp$

$\textbf{Unfold}^\sharp_{\mathbb{W}}$	:	$\mathbb{W}^\sharp \times \mathbb{W}^\sharp \times \mathcal{P}(\mathsf{LVar}) \to \mathbb{W}^\sharp \times \mathbb{N}^\sharp$
$\textbf{Split}^\sharp_{\mathbb{W}}$	:	$\mathbb{W}^\sharp \times \mathbb{W}^\sharp \times \mathcal{P}(\mathsf{LVar}) \to \mathbb{W}^\sharp$
$\textbf{Concat}^\sharp_{\mathbb{W}}$	:	$\mathbb{W}^\sharp \times \mathbb{W}^\sharp \times \mathcal{P}(\mathsf{LVar}) \to \mathbb{W}^\sharp$
$\textbf{Proj}^\sharp_{\mathbb{W}}$	:	$\mathbb{W}^\sharp \times \mathcal{P}(\mathsf{LVar}) \to \mathbb{W}^\sharp$

Shape domain $\mathbb{G}^\sharp$

$\textbf{UnfoldList}^\sharp_{\mathbb{G}}$	:	$\mathbb{G}^\sharp \times \mathsf{E} \times \mathcal{P}(\mathsf{LVar}) \to \mathcal{P}(\mathbb{G}^\sharp \times \mathbb{N}^\sharp \times \mathbb{W}^\sharp)$
$\textbf{FoldList}^\sharp_{\mathbb{G}}$	:	$\mathbb{G}^\sharp \times \mathsf{E} \times \mathsf{LVar} \to \mathbb{G}^\sharp$
$\textbf{DecompList}^\sharp_{\mathbb{G}}$	:	$\mathbb{G}^\sharp \times \mathsf{E} \times \mathcal{P}(\mathsf{LVar}) \to \mathbb{G}^\sharp \times \mathbb{N}^\sharp \times \mathbb{W}^\sharp$
$\textbf{ComposeList}^\sharp_{\mathbb{G}}$	:	$\mathbb{G}^\sharp \times \mathsf{E} \times \mathsf{LVar} \to \mathbb{G}^\sharp$
$\textbf{UnfoldChk}^\sharp_{\mathbb{G}}$	:	$\mathbb{G}^\sharp \times \mathsf{E} \times \mathsf{LVar} \to \mathbb{G}^\sharp \times \mathbb{N}^\sharp$

$\mathbf{FoldChk}^\sharp_{\mathbb{G}}$	:	$\mathbb{G}^\sharp \times \mathsf{E} \to \mathbb{G}^\sharp$
$\mathbf{DecompBlk}^\sharp_{\mathbb{G}}$	:	$\mathbb{G}^\sharp \times \mathsf{E} \times \mathsf{LVar} \to \mathbb{G}^\sharp \times \mathbb{N}^\sharp$
$\mathbf{ComposeBlk}^\sharp_{\mathbb{G}}$	:	$\mathbb{G}^\sharp \times \mathsf{E} \to \mathbb{G}^\sharp$
		Shape-value domain $\mathbb{H}^\sharp$
$\mathbf{UnfoldList}^\sharp_{\mathbb{H}}$	:	$\mathbb{H}^\sharp \times \mathsf{E} \times \mathcal{P}(\mathsf{LVar}) \to \mathcal{P}(\mathbb{H}^\sharp)$
$\mathbf{FoldList}^\sharp_{\mathbb{H}}$	:	$\mathbb{H}^\sharp \times \mathsf{E} \times \mathsf{LVar} \to \mathbb{H}^\sharp$
$\mathbf{DecompList}^\sharp_{\mathbb{H}}$	:	$\mathbb{H}^\sharp \times \mathsf{E} \times \mathcal{P}(\mathsf{LVar}) \to \mathbb{H}^\sharp$
$\mathbf{ComposeList}^\sharp_{\mathbb{H}}$	:	$\mathbb{H}^\sharp \times \mathsf{E} \times \mathsf{LVar} \to \mathbb{H}^\sharp$
$\mathbf{UnfoldChk}^\sharp_{\mathbb{H}}$	:	$\mathbb{H}^\sharp \times \mathsf{E} \times \mathsf{LVar} \to \mathbb{H}^\sharp$
$\mathbf{FoldChk}^\sharp_{\mathbb{H}}$	:	$\mathbb{H}^\sharp \times \mathsf{E} \to \mathbb{H}^\sharp$
$\mathbf{DecompBlk}^\sharp_{\mathbb{H}}$	:	$\mathbb{H}^\sharp \times \mathsf{E} \times \mathsf{LVar} \to \mathbb{H}^\sharp$
$\mathbf{ComposeBlk}^\sharp_{\mathbb{H}}$	:	$\mathbb{H}^\sharp \times \mathsf{E} \to \mathbb{H}^\sharp$
		Extended symbolic heap domain $\mathbb{M}^\sharp$
$\mathbf{EvalL}^\sharp_{\mathbb{M}}$	:	$\mathbb{Loc} \times \mathbb{M}^\sharp \to \mathsf{E} \times \mathsf{E}^?$
$\mathbf{EvalE}^\sharp_{\mathbb{M}}$	:	$\mathbb{Exp} \times \mathbb{M}^\sharp \to \mathsf{LVar} \times \mathsf{E}^?$
$\mathbf{Mutate}^\sharp_{\mathbb{M}}$	:	$\mathsf{E} \times \mathsf{LVar} \times \mathbb{M}^\sharp \to \mathbb{M}^\sharp$
$\mathbf{Sbrk}^\sharp_{\mathbb{M}}$	:	$\mathsf{E} \times \mathsf{LVar} \times \mathbb{M}^\sharp \to \mathbb{M}^\sharp$
$\mathbf{Assign}^\sharp_{\mathbb{M}}$	:	$\mathbb{Loc} \times \mathbb{Exp} \times \mathbb{T} \times \mathbb{M}^\sharp \to \mathcal{P}(\mathbb{M}^\sharp)$
$\mathbf{Assign}^\sharp_{\texttt{sbrk}}$	:	$\mathbb{Loc} \times \mathbb{Exp} \times \mathbb{M}^\sharp \to \mathbb{M}^\sharp$
$\mathbf{Guard}^\sharp_{\mathbb{M}}$	:	$\mathbb{Bexp} \times \mathbb{M}^\sharp \to \mathcal{P}(\mathbb{M}^\sharp)$
$\mathbf{Unfold}^\sharp_{\mathbb{M}}$	:	$\mathbb{M}^\sharp \times \mathcal{P}(\mathsf{E}) \times \mathcal{P}(\mathsf{LVar}) \to \mathcal{P}(\mathbb{M}^\sharp)$
		Combined abstract domain $\mathbb{A}^\sharp$
$\mathbf{Assign}^\sharp_{\mathbb{A}}$	:	$\mathbb{Loc} \times \mathbb{Exp} \times \mathbb{T} \times \mathbb{A}^\sharp \to \mathbb{A}^\sharp$
$\mathbf{Assign}^\sharp_{\texttt{sbrk}}$	:	$\mathbb{Loc} \times \mathbb{Exp} \times \mathbb{A}^\sharp \to \mathbb{A}^\sharp$
$\mathbf{Guard}^\sharp$	:	$\mathbb{Bexp} \times \mathbb{A}^\sharp \to \mathbb{A}^\sharp$

Table 7.1: Abstract operations of domains

7.3.1 Assignments

Recall that the abstract transformers for the assignment $loc :=_t exp$ and the system assignment $loc :=_t \texttt{sbrk}(exp)$ are denoted by $\mathbf{Assign}^\sharp_{\mathbb{M}}$ and $\mathbf{Assign}^\sharp_{\texttt{sbrk}}$, respectively. Given an assignment with an abstract value, $\mathbf{Assign}^\sharp_{\mathbb{M}}$ and $\mathbf{Assign}^\sharp_{\texttt{sbrk}}$ should compute a sound post-condition for the assignment.

Let $\mathbb{m}^\sharp$ be the abstract value representing the abstract state in the input of transformer. We assume that the numerical domain provides the transformer $\mathbf{Assign}^\sharp_{\mathbb{N}}$ for the assignment on scalar variables. For assignment on pointer type variables, before manipulating the abstract value, the abstract transformers $\mathbf{Assign}^\sharp_{\mathbb{M}}$ and $\mathbf{Assign}^\sharp_{\texttt{sbrk}}$ call two operations: location evaluation and expression evaluation to expose the locations changed or read by the assignment.

7.3.1.1 Location and Expression Evaluations

Location evaluation: Given a location expression loc and an abstract state $(\epsilon^\sharp, \Sigma, \Pi)$, the aim of location evaluation operation, denoted by $\mathbf{EvalL}^\sharp_{\mathbb{M}}$, is to locate an edge in Σ that includes the left value location denoted by loc. The operation is defined as follows:

$$\mathbf{EvalL}^\sharp_{\mathbb{M}} : \mathbb{Loc} \times \mathbb{M}^\sharp \rightarrow \mathsf{E} \times \mathsf{E}^?.$$

Recall that E represents the set of edges in the abstract heap which is defined in Section 6.3.5 and $\mathsf{E}^?$ is an optional value. If the location denoted by loc appears in an edge which is not summarized, the value returned by $\mathbf{EvalL}^\sharp_{\mathbb{M}}$ can be a points-to edge (e.g., $(p, *, X)$ represents program variable p points to node X), or an edge labelled by an unsummarized predicate atom, (e.g., in the second frame of Figure 7.6, the location `nxt->fnx` is evaluated to an edge (X, fck, X')).

If the location expression operation involves exposing a memory cell summarized by an edge, then $\mathbf{EvalL}^\sharp_{\mathbb{M}}$ returns the information for unfolding, i.e., the summary edge. The unfolding operation shall be performed on the summarized edge to materialize the needed edge. For example, in the abstract state presented in the third frame of Figure 7.6, to evaluate the location `nxt->fnx`, $\mathbf{EvalL}^\sharp_{\mathbb{M}}$ returns the summary edge $(X', \mathsf{flso}[W_1], \mathsf{nil})$ which should be unfolded. Recall that the unfolding operation returns a disjunction of abstract values. The analysis will evaluate loc again on each disjunct generated by the unfolding.

Expression evaluation: The expression evaluation operation $\mathbf{EvalE}^{\sharp}_{\mathbb{M}}$ evaluates an expression to a location value or a numerical variable. Similarly, the unfolding operation is required if the node denoted by the expression is included in a summary edge and a node should be materialized in the shape. In this case, $\mathbf{EvalE}^{\sharp}_{\mathbb{M}}$ returns the information, i.e., an edge for unfolding.

$$\mathbf{EvalE}^{\sharp}_{\mathbb{M}} : \mathbb{Exp} \times \mathbb{M}^{\sharp} \rightarrow \mathsf{LVar} \times \mathsf{E}^{?}$$

Algorithm 3: Algorithm of $\mathbf{Assign}^{\sharp}_{\mathbb{M}}$

```
input  : an assignment $loc :=_t exp$ and an abstract value $m \in \mathbb{M}^{\sharp}$
output : a new abstract value

begin
    $(l^{\sharp}, el) \leftarrow \mathbf{EvalL}^{\sharp}_{\mathbb{M}}(loc, m)$;                ▷ evaluate a location
    $(v^{\sharp}, ev) \leftarrow \mathbf{EvalE}^{\sharp}_{\mathbb{M}}(exp, m)$;              ▷ evaluate an expression
    if $(el \cup ev)$ is empty then
        $\mathbb{m}^{\sharp} \leftarrow \mathbf{Mutate}^{\sharp}_{\mathbb{M}}(l^{\sharp}, v^{\sharp}, m)$;          ▷ mutate the value
        return Norm($\mathbb{m}^{\sharp}$);                            ▷ return normalised value
    else
        $R \leftarrow \varnothing$;
        foreach (disjunct $u$ in $\mathbf{Unfold}^{\sharp}_{\mathbb{M}}(m, el \cup ev)$) do
            $(l^{\sharp}, el) \leftarrow \mathbf{EvalL}^{\sharp}_{\mathbb{M}}(loc, u)$;
            $(v^{\sharp}, ev) \leftarrow \mathbf{EvalE}^{\sharp}_{\mathbb{M}}(exp, u)$;
            $\mathbb{m}^{\sharp} \leftarrow \mathbf{Mutate}^{\sharp}_{\mathbb{M}}(l^{\sharp}, v^{\sharp}, u)$;
            $R \leftarrow R \cup \{\mathbb{m}^{\sharp}\}$;
        end
        return Norm($\bigvee_{a_i \in R} a_i$);
    end
end
```

Transformer function for assignment: The algorithm of $\mathbf{Assign}^{\sharp}_{\mathbb{M}}$ is described in Algorithm 3. The first case is for a fully exposed value. The location and expression evaluation operations directly return evaluated values and no unfolding is needed.

The second one is the unfolding is needed to evaluate the location or the expression. Given an abstract value and a set of edges, the unfolding operation

$\mathbf{Unfold}^{\sharp}_{\mathbb{M}}$ selects an appropriate unfolding operation (shown in Table 7.1) provided by the shape-value domain $\mathbb{H}^{\sharp}$ for each edge which should be unfolded. Since unfolding may generate a disjunction of states, $\mathbf{Assign}^{\sharp}_{\mathbb{M}}$ performs the operations like the first case to reflect the assignment on each unfolded shape.

Theorem 7.2 (Soundness of $\mathbf{Assign}^{\sharp}_{\mathbb{M}}$)**.** *The abstract transformer* $\mathbf{Assign}^{\sharp}_{\mathbb{M}}$ *computes a sound postconditon for the assignment statement* $loc :=_t exp$*. If* $(\epsilon, h) \in \gamma_{\mathbb{M}}(\mathfrak{m}^{\sharp})$*, then*

$$(\epsilon, h[\mathcal{L}[\![loc]\!](\epsilon, h) \leftarrow_t \mathcal{E}[\![exp]\!](\epsilon, h)]) \in \gamma_{\mathbb{M}}(\mathbf{Assign}^{\sharp}_{\mathbb{M}}(loc, exp, t, \mathfrak{m}^{\sharp})).$$

■

The overall abstract transformer for assignment is denoted by $\mathbf{Assign}^{\sharp}_{\mathbb{A}}$ is defined based on $\mathbf{Assign}^{\sharp}_{\mathbb{M}}$. The definition is given as follows:

$$\mathbf{Assign}^{\sharp}_{\mathbb{A}}(loc, exp, t, S^{\sharp}) \triangleq \mathbf{Norm}(\bigcup\{\mathbf{Assign}^{\sharp}_{\mathbb{M}}(loc, exp, t, v) | v \in S^{\sharp}\})$$

Theorem 7.3 (Soundness of $\mathbf{Assign}^{\sharp}_{\mathbb{A}}$)**.** *The abstract transformer* $\mathbf{Assign}^{\sharp}_{\mathbb{A}}$ *computes a sound postconditon for the assignment statement* $loc :=_t exp$*. If* $(\epsilon, h) \in \gamma_{\mathbb{A}}(\mathfrak{m}^{\sharp})$*, then*

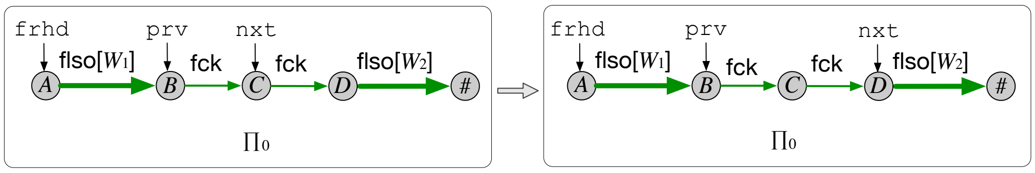
$$(\epsilon, h[\mathcal{L}[\![loc]\!](\epsilon, h) \leftarrow_t \mathcal{E}[\![exp]\!](\epsilon, h)]) \in \bigcup\{\gamma_{\mathbb{M}}(\mathfrak{m}^{\sharp}_u) \mid \mathfrak{m}^{\sharp}_u \in \mathbf{Assign}^{\sharp}_{\mathbb{A}}(loc, exp, t, \mathfrak{m}^{\sharp})\}.$$

■

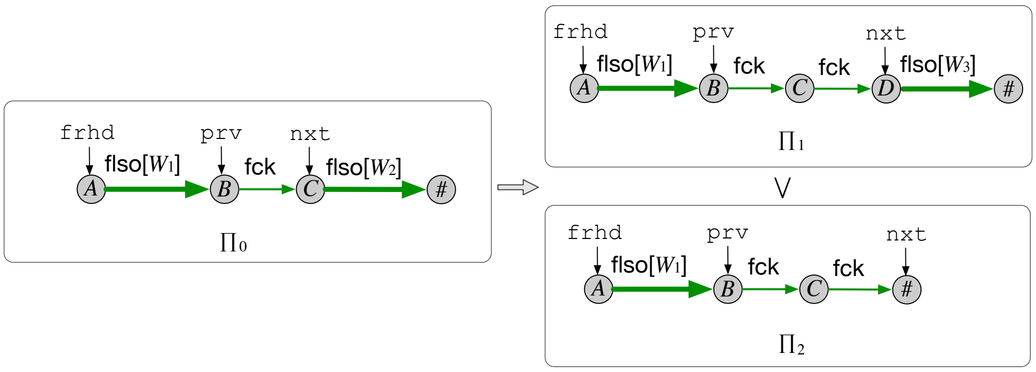
7.3.1.2 Assignment $loc := loc'$

Assignment with no folding: We select as running example the assignment (at line 32, `nxt:=nxt->fnx`) in the implementation of LA allocator shown in Figure 6.6 to illustrate the abstract transformer for this kind of assignment. In this paragraph, we illustrate the simplest case where none of the locations appearing in either side of the assignment are summarized. Suppose that the abstract state before the assignment is the one in Figure 7.8(a). Because this assignment only works on the free list level, for simplicity, we remove part of the abstract value concerning the heap list. The abstract state shown on the right of Figure 7.8(a) is obtained after applying the abstract transformer $\mathbf{Assign}^{\sharp}_{\mathbb{M}}$. $\mathbf{Assign}^{\sharp}_{\mathbb{M}}$ first calls the location evaluation which evaluates the location expression `nxt` to an edge in the shape. In Figure 7.8(a), the edge corresponding to the location expression is a points-to edge from variable `nxt` to node C. Then,

Assign$^\sharp_{\mathbb{M}}$ calls the expression evaluation to evaluate `nxt->fnx`. The first step is to replace the program variable `nxt` with a symbolic variable in the abstract shape by using the abstract environment $\epsilon^\sharp$. We obtain the node C. The next step is to access the `fnx` field of the node C. Recall that in the definition of $\mathsf{fck}(C; D)$, D represents the location stored in the `fnx` field of node C, thus the access of `fnx` field leads to exposing the fck edge. The expression evaluation yields the node D. Finally, the points-to edge is updated to point to the node specified by `nxt->fnx`, i.e., node D, by the mutation function **Mutate**$^\sharp_{\mathbb{M}}$.



(a) The abstract value is not summarized



(b) The edge is materialized by unfolding operation

Figure 7.8: Applying **Assign**$^\sharp_{\mathbb{M}}$ to assignment `nxt:=nxt->fnx`

Assignment over summary: Let us consider the case where the location in the assignment is summarized in the abstract value. The unfolding operation has to be called to materialize the desired edge. This situation appears for our running example when assignment `nxt:=nxt->fnx` is done on the abstract state on the left of Figure 7.8(b). The program variable `nxt` points to node C which is the start node of a summary edge labelled by the list segment atom $\mathsf{flso}[W_2]$. The expression evaluation fails when evaluating the right side of the assignment because the free chunk corresponding to the expression `nxt->fnx` is summarized as part of the list segment. Thus, the summary edge flso$[W_2]$ should be unfolded to materialize an edge for the next field. Recall that the unfolding operation generates a disjunction of abstract values. On the abstract values obtained by the unfolding, **Assign**$^\sharp_{\mathbb{M}}$ performs the evaluations again and now acts as in the previous paragraph. The abstract state obtained is shown on the right of Figure 7.8(b).

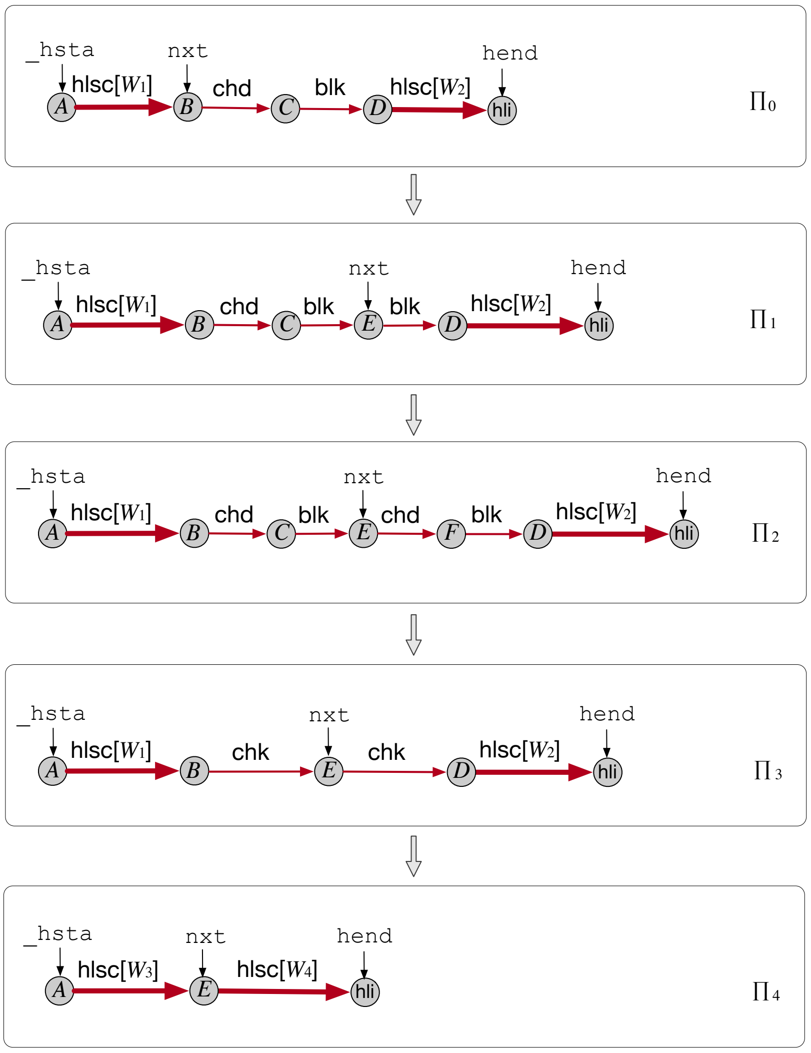


Figure 7.9: Applying **Assign**$^\sharp_{\mathbb{A}}$ to assignment `nxt+=nxt->size`

7.3.1.3 Assignment $loc := exp$

Assignments of this form are frequently used in SDMA to perform pointer arithmetics. We focus on one form of the assignment whose right-hand side is an expression that adds a positive integer to a location, e.g., $loc :=_t loc' + c$. We select as running example the assignment (at line 36, `nxt+=nxt->size`) from the implementation of LA allocator (shown in Figure 6.6) to explain the procedure. The assignment is normalized to `nxt:=nxt+nxt->size` at first. We detail the steps of the abstract transformer in the following.

We assume that the abstract state before the assignment has the form shown on the top of Figure 7.9. For simplicity, we remove the shape of the free list because we suppose that the hierarchical unfolding has instantiated the location pointed by `nxt` at the heap list level. More precisely, `nxt` points to a chunk represented by node B which as been unfolded into a chunk header (from node B to node C) and a body part (from node C to node D).

The evaluation of the left-hand side loc of the assignment is the same as in the previous section. We obtain the points-to edge in the shape for `nxt`. However, there is no explicit node in the shape when evaluating the right-hand side expression `nxt+nxt->size`.

The abstract transformer creates a new node in the shape to represent the location of `nxt+nxt->size`. The position at which the new node is added depends on the value of `nxt->size`. The different cases are tested in the analysis. In our running example, `nxt->size` is less than the size of chunk starting from C, a new node E is added between C and D. This step calls the abstract operation $\mathbf{DecompBlk}^\sharp_\mathbb{G}$ defined in Section 7.2.3.4 and yields a middle state as shown in the 2nd frame of Figure 7.9. Now `nxt` points to the node E. Between the node E and the node D is an edge labeled by the atom blk representing a raw memory region.

According to the type of `nxt`, the location to which `nxt` points should be the start address of a chunk. Thus, the raw memory region $\mathsf{blk}(E, D)$ is split into a chunk header and a chunk body and the new node F is added. Notice that E is a new free chunk after the assignment at line 37 (`nxt->size:=nuits`), thus it will be added into the free list. This communication between the two abstractions is explained in Section 7.2.5. The final abstract state on heap list level is obtained by calling the normalisation function.

Algorithm 4: Algorithm of $\mathbf{Assign}^\sharp_{\texttt{sbrk}}$

```
input  : an assignment $loc :=_t \texttt{sbrk}(exp)$ and an abstract value $v \in \mathbb{M}^\sharp$
output : a new abstract value

begin
    $(e, el) \leftarrow \mathbf{EvalL}^\sharp_{\mathbb{M}}(loc, m)$;          ▷ evaluate a location
    $(v^\sharp, er) \leftarrow \mathbf{EvalE}^\sharp_{\mathbb{M}}(exp, m)$;    ▷ evaluate an expression
    if $((el \cup er) = \varnothing)$ then
        $\mathbb{n}^\sharp \leftarrow \mathbf{Sbrk}^\sharp_{\mathbb{M}}(e, v^\sharp, m)$;   ▷ mutate the value
        return Norm($\mathbb{n}^\sharp$);                 ▷ return normalised value
    else
        $R \leftarrow \varnothing$;
        foreach (disjunct $u$ in $\mathbf{Unfold}^\sharp_{\mathbb{M}}(m, el \cup er)$) do
            $(e, el) \leftarrow \mathbf{EvalL}^\sharp_{\mathbb{M}}(loc, u)$;
            $(v^\sharp, er) \leftarrow \mathbf{EvalE}^\sharp_{\mathbb{M}}(exp, u)$;
            $\mathbb{n}^\sharp \leftarrow \mathbf{Sbrk}^\sharp_{\mathbb{M}}(e, v^\sharp, u)$;
            $R \leftarrow R \cup \{\mathbb{n}^\sharp\}$;
        end
        return Norm($\bigvee_{a_i \in R} a_i$);
    end
end
```

7.3.1.4 System Assignment: $loc := \texttt{sbrk}(exp)$

The essence of system assignment $loc := \texttt{sbrk}(exp)$ is to extend the data segment of the current process, so also the SDMA's memory region. The overall algorithm of its abstract transformer $\mathbf{Assign}^\sharp_{\texttt{sbrk}}$ is described in Algorithm 4. It calls the location evaluation and expression evaluation which is as same as $\mathbf{Assign}^\sharp_{\mathbb{M}}$.

The underlying operation used in $\mathbf{Assign}^\sharp_{\texttt{sbrk}}$ is $\mathbf{Sbrk}^\sharp_{\mathbb{M}}$. It takes an edge which is the evaluation of the location loc, an evaluation v of exp and an unfolded abstract value $(\epsilon^\sharp, \Sigma, \Pi)$ and then returns a new value. In most cases, the edge is a points-to edge, that is a program variable points to a node in the abstract graph, i.e., $\{loc \mapsto n\} \in \epsilon^\sharp$.

The algorithm for $\mathbf{Sbrk}^\sharp_{\mathbb{M}}$ is defined in Algorithm 5. There are two cases. When constraint on the value v implies it is equal to zero, it means that the memory is not extended. And that the points-to edge will be updated to point to the heap limitation address, i.e., the address right after the last address of

Algorithm 5: Algorithm of $\mathbf{Sbrk}^{\sharp}_{\mathbb{M}}$

```
input  : an unfolded abstract value $(\epsilon^\sharp, \Sigma, \Pi)$, an edge ( $\{loc \mapsto n\} \in \epsilon^\sharp$), a
         value $v \geq 0$,
output : a new abstract value

begin
    if $v = 0$ then
        $\epsilon^\sharp \leftarrow (\epsilon^\sharp \setminus \{loc \mapsto n\}) \cup \{loc \mapsto \mathsf{hli}\}$;      ▷ update points-to edge
        return $(\epsilon^\sharp, \Sigma, \Pi)$;
    else
        if $\exists X, P \cdot P(X; \mathsf{hli}) \in \Sigma$ then
            $Y \leftarrow \mathsf{hli}$;                                            ▷ create a new node
            $\epsilon^\sharp \leftarrow (\epsilon^\sharp \setminus \{loc \mapsto n\}) \cup \{loc \mapsto Y\}$;
            $\Sigma \leftarrow (\Sigma \setminus P(X; \mathsf{hli})) * P(X; Y) * \mathsf{blk}(Y; \mathsf{hli})$;   ▷ add a blk edge
            $\Pi \leftarrow \Pi[X/\mathsf{hli}] \wedge (\mathsf{hli} - X = v)$;              ▷ add a constraint
            return $(\epsilon^\sharp, \Sigma, \Pi)$;
        else
            $X \leftarrow \mathsf{hli}$;
            $\epsilon^\sharp \leftarrow (\epsilon^\sharp \setminus \{loc \mapsto n\}) \cup \{loc \mapsto X\}$;
            $\Sigma \leftarrow \Sigma * \mathsf{blk}(X; \mathsf{hli})$;
            $\Pi \leftarrow \Pi[X/\mathsf{hli}] \wedge (\mathsf{hli} - X = v)$;
            return $(\epsilon^\sharp, \Sigma, \Pi)$;
        end
    end
end
```

the memory region, denoted by node hli. When it is strictly greater than zero, the memory is extended and represented by adding a new edge in the shape. An example of applying $\mathbf{Sbrk}^{\sharp}_{\mathbb{M}}$ on the assignment (`p=sbrk(exp)`) is shown in Figure 7.10. The program variable `p` originally points to the node B. After the assignment, `p` points to the new node X. The edge node hli is replaced by a node X and a new edge $\mathsf{blk}(X, \mathsf{hli})$ is added. The new pure constraint Π_1 is obtained by substituting hli by X in Π and by adding the constraint $\mathsf{hli} - X = v$.

7.3.2 Condition Tests

The concrete semantics of a condition test $b \in \mathbb{Bexp}$ (e.g., a boolean expression or a linear inequality) is the function **Guard**$[\![.]\!]$ which inputs a set of concrete states S and returns the subset of S in which b evaluates to **true**. The corresponding

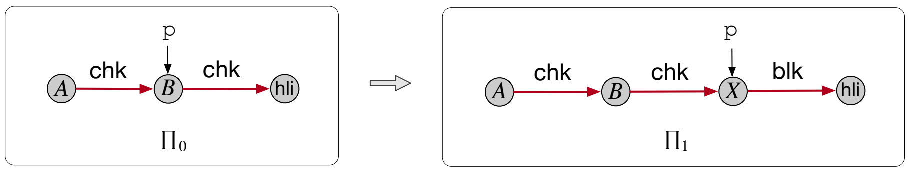


Figure 7.10: An example of applying $\mathbf{Sbrk}^\sharp_{\mathbb{M}}$ on assignment: `p:=sbrk(exp)`

abstract transformer, denoted by $\mathbf{Guard}^\sharp[\![.]\!]$ filters out abstract values in which b does not evaluates to **true**.

We first consider the case when the condition b is a constraint over only scalar variables, thus, it is not needed to check the spacial part of the abstract value. We assume that the numerical domain used provides the operation $\mathbf{Guard}^\sharp_{\mathbb{N}}$ which is the abstract condition test. The type of $\mathbf{Guard}^\sharp_{\mathbb{N}}$ is presented in Table 7.1. Given a boolean expression and a set of numerical constraint, $\mathbf{Guard}^\sharp_{\mathbb{N}}$ returns an abstract element in which b evaluates to **true**. In this case, the abstract transformer for condition tests is defined as follows:

$$\mathbf{Guard}^\sharp[\![b]\!](S^\sharp) \triangleq \{(\epsilon^\sharp_j, \Sigma_j, \Pi_j) \mid \Pi_j \in \mathbf{Guard}^\sharp_{\mathbb{N}}(b)(S^\sharp_{\mathbb{N}})\}$$

where $S^\sharp = \{(\epsilon^\sharp_0, \Sigma_0, \Pi_0), ..., (\epsilon^\sharp_n, \Sigma_n, \Pi_n)\}$ and $S^\sharp_{\mathbb{N}} = \{\Pi_0, ..., \Pi_n\}$.

When the condition b is the constraint over location variables, the transfer function will check the shape. Given a comparison $exp_1 \ltimes exp_2$, the first step is to call expression evaluation operation $\mathbf{EvalE}^\sharp_{\mathbb{M}}$ to evaluate exp_1 and exp_2 on an abstract value $\mathfrak{m}^\sharp = (\epsilon^\sharp, \Sigma, \Pi)$.

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{Guard}^\sharp[\![b]\!](S^\sharp) \triangleq \text{match } b \text{ with:}} \\
\mid \neg b & \rightarrow & \mathbf{Guard}^\sharp[\![\neg b]\!](S^\sharp) \\
\mid b_1 \wedge b_2 & \rightarrow & \mathbf{Guard}^\sharp[\![b_2]\!](\mathbf{Guard}^\sharp[\![b_1]\!](S^\sharp)) \\
\mid b_1 \vee b_2 & \rightarrow & \mathbf{Guard}^\sharp[\![b_1]\!](S^\sharp) \sqcup^{\mathcal{A}} \mathbf{Guard}^\sharp[\![b_2]\!](S^\sharp) \\
\mid exp_1 \ltimes exp_2 & \rightarrow & \bigcup\{\mathbf{Guard}^\sharp_{\mathbb{M}}(b, \mathfrak{m}^\sharp) \mid \mathfrak{m}^\sharp \in S^\sharp)\}
\end{array}$$

Figure 7.11: Abstract transformer of condition tests

Because $\mathbf{Guard}^\sharp_{\mathbb{M}}$ may need to perform the unfolding operation, $\mathbf{Guard}^\sharp_{\mathbb{M}}$ shall return a finite set of abstract values $\{\mathfrak{m}^\sharp_0, ..., \mathfrak{m}^\sharp_n\}$ filtered by the imput constraint.

Given a constraint $b = exp_1 \ltimes exp_2$, we denote by **el** and **er** the evaluated values of exp_1 and exp_2 returned by $\mathbf{EvalE}^\sharp_{\mathbb{M}}$ respectively and they may be obtained after unfolding the input value. We define the operator $\mathbf{Guard}^\sharp_{\mathbb{M}}$ which takes a constraint b, an abstract value $\mathbb{m}^\sharp = (\epsilon^\sharp, \Sigma, \Pi)$ then returns a new value. Precisely, in each element $\mathbb{m}^\sharp_i$, $\mathbf{Guard}^\sharp_{\mathbb{M}}$ propagates constraint Π_p in the pure part where Π_p is ($\mathbf{el} \ltimes \mathbf{er}$).

$$\mathbf{Guard}^\sharp_{\mathbb{M}}(b, \mathbb{m}^\sharp) \triangleq \begin{cases} \bigcup\{(\epsilon^\sharp_i, \Sigma_i, \Pi_i \wedge \Pi_p)\} & \text{if } \mathbb{m}^\sharp \text{ should be unfolded} \\ (\epsilon^\sharp, \Sigma, \Pi \wedge \Pi_p) & \text{otherwise} \end{cases}$$

Property 7.3.1 (Soundness of $\mathbf{Guard}^\sharp$)**.** *The abstract transformer of the condition test is sound, i.e., for all S and $S^\sharp$ such that $S \subseteq \gamma_{\mathbb{A}}(S^\sharp)$, we obtain*

$$\mathbf{Guard}[\![b]\!]S \subseteq \gamma_{\mathbb{A}}(\mathbf{Guard}[\![b]\!](S^\sharp))$$

7.4 Analysis Algorithm

We now describe the specific issues of the static analysis algorithm based on the hierarchical abstract domain presented in the last two chapters.

7.4.1 Main principles

The analysis algorithm consists of the following three steps. The first step targets on discovering the properties of the free and heap lists in order to select a suitable set of list segment predicates.

It consists of an inter-procedural and non relational *symbolic execution* of a correct client program like the one in Figure 7.12. The sets of reachable configurations are represented by abstract values of our domain built over the chunk and block atoms only, i.e., atoms using predicates fck, chk, chd, and blk. Thus, the heap and the free lists are completely unfolded.

```
int main(void) {
minit(1024);
void* p = malloc(20);
malloc(20);
mfree(p); p = NULL;
p = malloc(20);
malloc(20);
mfree(p); p = NULL;
return 0;
}
```

Figure 7.12: A client program

For example, the abstract value computed for the start location of method `malloc` (line 28 in Figure 6.6) when executing the client program in Figure 7.12 is built from four disjuncts whose shape part is given in Figure 7.13.

The client programs are chosen to reveal the heap list organisation (including chunk coalescing) and the shape of the free list. We don't employ the most general client or a client using an incorrect sequence of calls to the SDMA methods in order to speed-up this step and avoid configurations leading to error states that increases the size of abstract values.

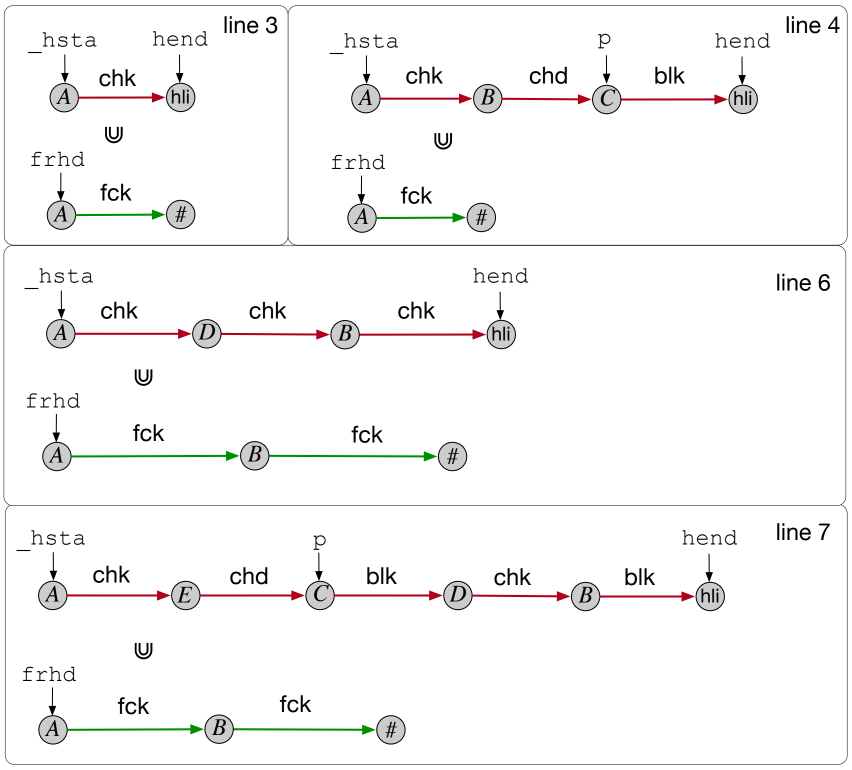


Figure 7.13: Spatial formulas at line 28 of allocation in 6.6

The second step transforms the abstract values computed by the previous step to obtain an abstract value representing a pre-condition of the SDMA method that constrains the global variables and the parameters of the method. For this, the variables of the client program (e.g., p in Figure 7.12) are projected out and folding lemmas are applied to obtain list atoms. For example, the transformation of the abstract value in Figure 7.13 leads to an abstract value with one disjunct whose spatial part is $\mathsf{hlsc}(A, 0; \mathsf{hli}, 0)[W_H] \Supset \mathsf{flso}(A, A; \mathsf{nil}, \mathsf{hli})[W_F]$. The resulting pre-condition is not the weakest one, but it is bigger than (as regards $\sqsubseteq$) the abstract value computed by the symbolic execution at this control location.

The third step does forward, non-relational abstract interpretation [CC77a] starting from the computed pre-conditions of each SDMA method. The analysis uses the widening operator to speed-up the convergence of the fix-point computation for program loops.

7.4.2 Experiments

We implemented the abstract domain and the analysis algorithm in Ocaml as a plug-in of the Frama-C platform [KKP+15]. We are using several modules of Frama-C, e.g., C parsing, abstract syntax tree transformations, and the fix-point computation. The data word domain uses as numerical join-lattice $\mathcal{N}$ the library of polyhedra with congruence constraints included in APRON [JM09]. To obtain precise numerical invariants, we manually transform program statements using bit-vector operations (e.g., line 16 of Figure 6.6(a)) into statements allowed by the polyhedra domain which over-approximate the original effect.

Table 7.2: Benchmark of SDMA

DMA	*LOC*	*List pred.*	*Time (s)*	$\lvert a^\sharp \rvert$	$\lvert W_H \rvert / \lvert W_F \rvert$	*Invariants*
DKFF	176	hlsc, flso	0.05	25	8/5	first-fit, MIN_SIZE-size
DKBF	130	hlsc, flso	0.05	26	8/6	best-fit, MIN_SIZE-size
LA	181	hlsc, flso	0.07	25	8/5	first-fit, 0-size
DKNF	137	hlsc, flso	0.05	30	8/6	first-fit, MIN_SIZE-size
KR	284	hlsc, flso	2.8	32	8/6	first-fit, 0-size

We applied our analysis on the benchmark of SDMA in Table 7.2. All of them are less than 300 lines of code. The second column represents the types of heap and free lists used in the allocator. DKFF and DKBF are implementations of Algorithms A and B from Section 2.5 of [Knu73]. These SDMA keep an acyclic free list sorted by the start addresses of chunks. The deallocation does coalescing of successive free chunks. The allocation implements a first-fit resp. best-fit policy such that the fitting chunk is not split if the remaining free part is less than `MIN_SIZE` (variant proposed in [Knu73]). This property is expressed by the following sub-formula of the invariant "MIN_SIZE-size" (for MIN_SIZE = 8):

$$\forall X \in W_H \cdot X.\texttt{size} \geq 8 \tag{7.1}$$

which is inferred by our analysis. The first-fit policy is implied by an abstract value similar to the one described in Section 5.3.1 (page 77). The best-fit policy is implied by a value using the constraint:

$$\forall X \in W_i \cdot X\texttt{.size} \geq \texttt{rsz} \Rightarrow X\texttt{.size} \geq Y\texttt{.size} \tag{7.2}$$

where `rsz` is the requested size, Y is the symbolic address of the fitting chunk, and W_i represents a list segment around the fitting chunk. Recall that in the universal constraints, the type of guard is fixed in the abstract domain.

LA is our running example in Figure 6.6; it follows the same principles as DKFF, but get rid of the constraint for chunk splitting. For this case study, our analyser infers the "0-size" invariant, i.e., $\forall X \in W_H \cdot X\texttt{.size} \geq 4$ (=**sizeof**(HDR)). Notice that the code analysed fixes an obvious problem of the `malloc` method published in [Ald08]. DKNF implements the next-fit policy using the "roving pointer" technique proposed in [Knu73]: a global variable points to the chunk in the free list involved in the last allocation or deallocation; `malloc` searches for a fitting free chunk starting from this pointer. Thus, the next-fit policy is a first-fit from the roving pointer. DKNF is challenging because the roving pointer introduces a case splitting that increases the size (number of disjuncts) in abstract values. The KR allocator [KR88] keeps a circular singly linked free list, circularly sorted by the chunk start addresses; the start of the free list points to the last deallocated block. The circular shape of the list requires to keep track of the free chunk with the biggest start address and this increases the size of the abstract values.

The analysis times reported in Table 7.2 have been obtained on a 2.53 GHz Intel Core 2 Duo laptop with 4GB of RAM. They correspond to the total time of the three steps of the analysis starting from the client given in Figure 7.12. The universally quantified invariants inferred for SDMA policies are given in the last column. Columns $|a^\sharp|$ and $|W_H|/|W_F|$ provide the maximum number of disjuncts generated for an abstract value resp. the maximum number of predicate atoms in each abstraction level.

7.5 Related Work and Conclusion

Our analysis infers complex invariants of free list SDMA implementations due to the combination of two ingredients: the hierarchical representation of the shape of the memory region managed by the SDMA and an abstract domain for the numerical constraints based on universally quantified formulas. The abstract domain has a clear logical definition, which facilitates the use of the inferred invariants by other verification methods.

The proposed abstract domain extends previous works [CDOY06, BDES11, Dra11, LYP11, DES13]. We consider the SL fragment proposed in [CDOY06] to analyse programs using pointer arithmetic. We enrich this fragment in both spatial and pure formulas to infer a richer class of invariants. E.g., we add a heap list level to track properties like chunk overlapping and universal constraints to infer first-fit policy invariants.

The split of shape abstraction on levels is inspired by work on overlaid data structures [LYP11, DES13]. We consider here a specific overlapping schema based on set inclusion which is adequate for the class of SDMA we consider. We propose new abstract transformers which do not require user annotations like in [LYP11]. Another hierarchical analysis of shape and numeric properties has been proposed in [SR12]. They consider the analysis of linked data structures coded in arrays and track the shape of these data structures and not the organisation of the set of free chunks. Their approach is not based on logic and the invariants inferred on the content of list segments are simpler.

Our abstract domain includes a simpler version of the data word domain proposed in [BDES11, Dra11], since the universal constraints quantify only one position in the list. Several abstract domains have been defined to infer invariants over arrays, e.g., [GLAS09] for array sizes, [GMT08, HP08] for array content. These works infer invariants of different kind on array partitions and they can not be applied directly to sequences of addresses. Recently, [LR15] defined an abstract domain for the analysis of array properties and applies it to the Minix 1.1 SDMA which uses chunks of fixed size. A modular combination of shape and numerical domains has been proposed in [CR13]. We extend their proposal to combine shape domains with domains on sequences of integers. Precise analyses exist for low level code in C [Min06] or for binary code [BR06]. They efficiently track properties about pointer alignment and memory region separations, but can not infer shape properties.

CHAPTER 8

Conclusion

8.1 Summary

In the first part of this thesis, we first provide a complete hierarchy of models, published in [FS17, FSG$^+$17], for the full class of list based SDMA. The hierarchy is built based on the stepwise refinement method and in a modular way. Our set of specifications is complete for the techniques usually employed in the list based SDMA. The refinement strategy and the principles we propose allow to extend the hierarchy to specify other policies used in SDMA. We construct the models in the Rodin platform by using the Event-B specification language. Several projects report on the mechanical proofs using theorem provers of (partial) correctness of code for specific purpose SDMA, e.g., [MAY06, TKN07a, KEH$^+$09, HP09, Chl11]. Most of these works use Separation Logic [ORY01] which provides a scalable and expressive reasoning framework. Our work is complementary to these projects. We provide reusable and complete specifications for all list based SDMA by applying several refinement steps, while they focus on the verification of specifications for a particular SDMA code.

The second part of the thesis focuses we focus on the verification of SDMA implementations by static analysis has been considered, which has been considered in [CDOY06, LR15, FS16]. All these methods infer only some properties for particular allocators. Indeed, they employ fragments of Separation Logic or some logics over arrays which are not expressive enough to cover fully the invariants of the SDMA analysed (e.g., the fit policy). The formalization we considered for SDMA provides reference specifications to compare with the inferred ones, in a logic fragment more general than Separation Logic. It motivates the extension or the direct application of general purpose methods based on Separation Logic, e.g., [CDNQ12, QHL+14]. Thus, we design an analysis [FS16] based on abstract interpretation using abstract domains based on a fragment of Separation Logic that is able to infer some invariants proposed in Event-B models. The Separation Logic fragment we proposed, called SLMA, could express properties of SDMA concerning the shape of the memory data structures stored in the memory region and the data value stored inside the memory. The abstract domain is built based on the logical formulae of SLMA. We make intensive use of two important techniques defined for the design of abstract domains: the hierarchical abstraction of memory region managed by the SDMA and the product of abstract domains.

8.2 Discussion

Formalization in Event-B: Notice that we use Event-B specification language to construct models for sequential programs. Event-B is a formalism for developing and verifying systems using events. The obvious problem is the semantic gap between sequential requirements and guard-action style event model. In an Event-B model, each event used to specify a transition or a function in sequential program is an atomic operation. It is non-deterministically selected for execution if its guards are satisfied under a state. To obtain a sequential model, one way is to constrain execution of events in the model, i.e., include a deterministic scheduler of events. The sequential program can be generated from the Event-B model and there exist several translation tools, such as translating Event-B to C, or Java. However, these translation tools have no formal result stating that the translation preserves the semantics.

Combining abstract domains: To infer spatial and numerical properties of heap-manipulating programs, designing an abstract domain using formulae from fragments of Separation Logic is a trend in static analysis based on abstract interpretation. The different families of properties are captured by distinct domains. The unbound heap regions (e.g., list) are specified by inductive predicates in shape abstraction and the numerical properties over data values (e.g., integer) are described by numerical and data word domains. Providing the communication between different domains is a complex task which has been handled while designing the analyzer.

8.3 Future Work

This thesis focuses on sequential allocators, however there are lots of concurrent dynamic allocators used in the operating systems for multiprocessors architectures. An aspect that can be the object of further investigation is to formalize and verify such concurrent allocators.

One recent survey paper [DD15] summaries different approaches to verifying linearizability and gives a detailed comparison between them. One of techniques for proving linearizability is construction-based proof. Abrial et al. introduce a constructive approach [AC05] to linearizability by using Event-B modelling. They present a completely formal development of concurrent queue algorithms. Building models for concurrent dynamic memory allocators is a more complex task. The most important steps are to determine the atomic operations in the concurrent allocators and to formalize multiple processors in the models. Another approach is to use CIVL [SZL+15] which is a concurrency intermediate verification language. It provides a general concurrency model capable of representing programs in a variety of concurrency dialects. For static analysis, our fragment of Separation Logic has inductive predicates specifying distinct types of free list used in SDMA. They specify mainly singly-linked list. An extension of the logic is to add inductive predicates to describe doubly-linked list. Another direction for future work is the design of decision procedures for fragments of this logic.

List of Figures

List of Tables

List of Symbols

Specification for SDMA

$\mathsf{hst}, \mathsf{hli} \in \mathbb{N}$	limits of the memory	nil	null memory address
$H \subset \mathbb{N}$	set of all chunks	$F \subset \mathbb{N}$	set of free chunks
$\mathsf{chd} \in \mathbb{N}$	size of header	$\mathsf{cal} \in \mathbb{N}$	alignment
$[\mathsf{hst}, \mathsf{hli}[$	chunks domain	$\mathtt{MOD}$	modulo operation
fit	fitting function	$csz(c)$	size of a chunk c
$cst(c)$	status of a chunk c		
I_{ec}	early coalescing	I_{pc}	partial coalescing
$\{e\}$	singleton set	$\{e_1, e_2, ..., e_n\}$	set enumeration
$\varnothing$	empty set	$S \cup T$	set union
$S \cap T$	set intersection	$S \setminus T$	set difference
$S \times T$	cartesian product	$\mathcal{P}(S)$	powset
$e_1 \mapsto e_2$	ordered pair	$m..n$	interval
$\mathsf{dom}(r)$	domain	$\mathsf{ran}(r)$	range
r^{-1}	inverse	$r[S]$	relational image
$id(S)$	identity	$S \lhd r$	domain subtraction
$r \rhd T$	range subtraction	$r_1 \lhd\!\!\!+ r_2$	overriding
$q \circ p$	composition	$S \nrightarrow T$	partial function
$S \rightarrow T$	total function	$S \rightarrowtail\!\!\!\!\!\!+ T$	partial injection
$S \rightarrowtail T$	total injection	$S \twoheadrightarrow\!\!\!\!\!\!+ T$	partial surjection
$S \twoheadrightarrow T$	total surjection	$S \rightarrowtail\!\!\!\!\!\!\twoheadrightarrow T$	bijection

Symbolics in the Logic

ϕ	assertion	$\{\phi\}C\{\psi\}$	Hoare triple
SL	Separation Logic	$P * Q$	separating conjunction
$P -\!\!* Q$	separating implication	$\mathsf{dom}()$	domain of function
$h_1 \perp h_2$	disjoint heaps in SL	$h_1 \uplus h_2$	union of heaps
SLMA	name of SL fragment	PVar	program variables
AVar	location variables	IVar	integer variables
SVar	sequence variables	LVar	logical variables
$\vec{x}, \vec{y}$	vector of variables	X, Y	location variables
W	sequence variable	$\#$	comparison operators
$\mathbb{A}$	domain of addresses	$\mathbb{V}$	domain of values
$\mathbb{F}$	the set of fields	$\mathbb{V}^+$	non-empty sequence
$\mathbb{V}^*$	sequence of values	$w[i]$	indexed access
$\lvert\mathbb{V}^+\rvert$	length of sequence	ϵ	empty sequence
$[X]$	singleton	$.$	sequence concatenation
t	integer term	Δ	integer formula
$\mathcal{F}$	set of function symbols	$\mathcal{F}_f$	function symbol ($f \in \mathbb{F}$)
I	interpretation	h	heap
(I, h)	heap state	Σ	spatial formula
Σ_H	heap list	Σ_F	free list
Π	pure formula	$\Supset$	composition operator
$\Pi_{\mathbb{N}}$	numerical constraint	$\Pi_{\mathbb{W}}$	sequential constraint
$h_1 \circledast h_2$	disjoint heap in SLMA	blk	raw memory atom
chk	chunk atom	chd	chunk header atom
fck	free chunk atom	hls	heap list atom
hlsc	special heap list atom	fls	free list atom
flso	ordered free list atom	$\prec_P$	order of predicates
$\mathbb{P}$	set of predicates		

Arithmetic, Language, Abstract Interpretation

$\mathcal{P}$	power set	$\mathbb{N}$	set of natural integers
$\mathbb{Z}$	set of integers	$\oplus$	arithmetic operators
$\ltimes$	comparison operator	$A \to B$	functions from A to Bs
$\sqsubseteq$	partial order	$\sqcup$	join
$\sqcap$	meet	$\perp$	least element
$\top$	greatest element	$\mathbf{lfp}F$	least fixpoint of F

$\mathbb{Loc}$	set of locations	$\mathbb{Exp}$	set of expressions
$\mathbb{T}$	set of types	$\mathbb{Bexp}$	set of booleans
E	edges in shape	N	nodes in shape
α	abstraction	γ	concretization
$\mathbb{G}^\sharp$	shape abstract domain	$\gamma_\mathbb{G}$	concretization of $\mathbb{G}^\sharp$
$\mathbb{N}^\sharp$	numerical domain	$\gamma_\mathbb{N}$	concretization of $\mathbb{N}^\sharp$
$\mathbb{W}^\sharp$	data words domain	$\gamma_\mathbb{W}$	concretization of $\mathbb{W}^\sharp$
$\mathbb{H}^\sharp$	shape-value domain	$\gamma_\mathbb{H}$	concretization of $\mathbb{H}^\sharp$
$\mathbb{M}^\sharp$	extended heap domain	$\gamma_\mathbb{M}$	concretization of $\mathbb{M}^\sharp$
$\mathbb{A}^\sharp$	combined abstract domain	$\gamma_\mathbb{A}$	concretization of $\mathbb{A}^\sharp$
∇	widening operator	$\nabla^\mathcal{N}$	widening in $\mathbb{N}^\sharp$
$\nabla^\mathcal{W}$	widening in $\mathbb{W}^\sharp$	$\nabla^\mathcal{A}$	widening in $\mathbb{A}^\sharp$

Bibliography

[ABH$^+$10] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in event-b. *International journal on software tools for technology transfer*, 12(6):447–466, 2010.

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[AC05] Jean-Raymond Abrial and Dominique Cansell. Formal construction of a non-blocking concurrent queue algorithm (a case study in atomicity). *J. UCS*, 11(5):744–770, 2005.

[AH07] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to event-b. *Fundamenta Informaticae*, 77(1-2):1–28, 2007.

[Ald08] Leslie Aldridge. Memory allocation in C. *Embedded Systems Programming*, pages 35–42, August 2008.

[Bar04] Jonathan Bartlett. Inside memory management. http://www.ibm.com/developerworks/library/l-memory/sidefile.html, 2004.

[BC13] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[BCD$^+$11] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

[BCO05a] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. A decidable fragment of separation logic. In *FSTTCS*, volume 3328, pages 97–109. Springer, 2005.

[BCO05b] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, volume 3780, pages 52–68. Springer, 2005.

[BDE$^+$10] Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, Ahmed Rezine, and Mihaela Sighireanu. Invariant synthesis for programs manipulating lists with unbounded data. In *CAV*, volume 6174 of *LNCS*, pages 72–88. Springer, 2010.

[BDES11] Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. On inter-procedural analysis of programs with lists and data. In *PLDI*, pages 578–589. ACM, 2011.

[BDES12] Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. Accurate invariant checking for programs manipulating lists and arrays with infinite data. In *ATVA*, volume 7561 of *LNCS*, pages 167–182. Springer, 2012.

[BDM05] Gerth Stølting Brodal, Erik D Demaine, and J Ian Munro. Fast allocation and deallocation with an improved buddy system. *Acta Informatica*, 41(4-5):273–291, 2005.

[BFPG14] James Brotherston, Carsten Fuhs, Juan A Navarro Pérez, and Nikos Gorogiannis. A decision procedure for satisfiability in separation logic with inductive predicates. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, page 25. ACM, 2014.

[BGKR16] James Brotherston, Nikos Gorogiannis, Max Kanovich, and Reuben Rowe. Model checking for symbolic-heap separation logic with inductive predicates. In *ACM SIGPLAN Notices*, volume 51, pages 84–96. ACM, 2016.

[BKSM11] Ghassem Barootkoob, Ehsan Musavi Khaneghah, Mohsen Sharifi, and Seyedeh Leili Mirtaheri. Parameters affecting the functionality of memory allocators. In *Communication software and networks (iccsn), 2011 ieee 3rd international conference on*, pages 499–503. IEEE, 2011.

[BR06] Gogul Balakrishnan and Thomas W. Reps. Recency-abstraction for heap-allocated storage. In *SAS*, volume 4134 of *LNCS*, pages 221–239. Springer, 2006.

[Bra07] Aaron R Bradley. *Safety analysis of systems*. PhD thesis, Stanford University, 2007.

[BW12] Ralph-Johan Back and Joakim Wright. *Refinement calculus: a systematic introduction*. Springer Science & Business Media, 2012.

[BY08] Michael Butler and Divakar Yadav. An incremental development of the mondex system in event-b. *Formal Aspects of Computing*, 20(1):61–77, 2008.

[CC76] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming, Paris, France*. Dunod, 1976.

[CC77a] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[CC77b] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of generalized type unions. In *ACM SIGPLAN Notices*, volume 12, pages 77–94. ACM, 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282. ACM, 1979.

[CCF13] Agostino Cortesi, Giulia Costantini, and Pietro Ferrara. A survey on product operators in abstract interpretation. *arXiv preprint arXiv:1309.5146*, 2013.

[CDNQ12] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming*, 77(9):1006–1036, 2012.

[CDOY06] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. *Static Analysis*, pages 182–203, 2006.

[CGJ$^+$01] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. In *Informatics*, pages 176–194. Springer, 2001.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96. ACM, 1978.

[Chl11] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *ACM SIGPLAN Notices*, volume 46, pages 234–245. ACM, 2011.

[CHO$^+$11] Byron Cook, Christoph Haase, Joël Ouaknine, Matthew J. Parkinson, and James Worrell. Tractable reasoning in a fragment of separation logic. In *CONCUR*, volume 6901 of *LNCS*, pages 235–249. Springer, 2011.

[CLCVH00] Agostino Cortesi, Baudouin Le Charlier, and Pascal Van Hentenryck. Combinations of abstract domains for logic programming: Open product and generic pattern construction. *Science of Computer Programming*, 38(1-3):27–71, 2000.

[CR13] Bor-Yuh Evan Chang and Xavier Rival. Modular construction of shape-numeric analyzers. In *Semantics, Abstract Interpretation, and*

Reasoning about Programs, volume 129 of *EPTCS*, pages 161–185, 2013.

[CYO01] Cristiano Calcagno, Hongseok Yang, and Peter W O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *FSTTCS*, volume 1, pages 108–119. Springer, 2001.

[DD15] Brijesh Dongol and John Derrick. Verifying linearisability: A comparative survey. *ACM Computing Surveys (CSUR)*, 48(2):19, 2015.

[DES13] Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. Local shape analysis for overlaid data structures. In *SAS*, volume 7935 of *LNCS*, pages 150–171. Springer, 2013.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[DOY06] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *TACAS*, volume 3920, pages 287–302. Springer, 2006.

[Dra11] Cezara Dragoi. *Automated verification of heap-manipulating programs with infinite data*. PhD thesis, University Paris Diderot, 2011.

[DREB98] Willem-Paul De Roever, Kai Engelhardt, and Karl-Heinz Buth. *Data refinement: model-oriented proof methods and their comparison*, volume 47. Cambridge University Press, 1998.

[DSDR12] Dipti Diwase, Shraddha Shah, Tushar Diwase, and Priya Rathod. Survey report on memory allocation strategies for real time operating system in context with embedded devices. *International Journal of Engineering Research and Applications (IJERA) Vol*, 2:1151–1156, 2012.

[ESS13] Constantin Enea, Vlad Saveluc, and Mihaela Sighireanu. Compositional invariant checking for overlaid and nested linked lists. In *ESOP*, volume 7792 of *LNCS*, pages 129–148. Springer, 2013.

[ESW15] Constantin Enea, Mihaela Sighireanu, and Zhilin Wu. On automated lemma generation for separation logic with inductive definitions. In *ATVA*, LNCS, pages 80–96. Springer, 2015.

[Flo93] Robert W Floyd. Assigning meanings to programs. In *Program Verification*, pages 65–81. Springer, 1993.

[FS16] Bin Fang and Mihaela Sighireanu. Hierarchical shape abstraction for analysis of free-list memory allocators. In *Proceedings of International Symposium on Logic-based Program Synthesis and Transformation*, pages 151–167, 2016.

[FS17] Bin Fang and Mihaela Sighireanu. A refinement hierarchy for free list memory allocators. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management, ISMM 2017, Barcelona, Spain, June 18, 2017*, pages 104–114, 2017.

[FSG$^+$17] Bin Fang, Mihaela Sighireanu, Pu Geguang, Su Wen, Abrial Jean-Raymond, QIAO Lei, and YANG Mengfei. Formal modelling of list based dynamic memory allocators. *SCIENCE CHINA Information Sciences*, 2017.

[GLAS09] S. Gulwani, T. Lev-Ami, and S. Sagiv. A combination framework for tracking partition sizes. In *POPL*, pages 239–251. ACM, 2009.

[GMT08] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, pages 235–246. ACM, 2008.

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[HP08] N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *PLDI*, pages 339–348. ACM, 2008.

[HP09] Chris Hawblitzel and Erez Petrank. Automated verification of practical garbage collectors. In *ACM SIGPLAN Notices*, volume 44, pages 441–453. ACM, 2009.

[IRS13] Radu Iosif, Adam Rogalewicz, and Jiri Simacek. The tree width of separation logic with recursive definitions. In *CADE*, volume 7898 of *LNCS*, pages 21–38. Springer, 2013.

[JL96] Richard Jones and Rafael D Lins. Garbage collection: algorithms for automatic dynamic memory management. 1996.

[JM09] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, volume 5643 of *LNCS*, pages 661–667. Springer, 2009.

[KdBdGZ52] Stephen Cole Kleene, NG de Bruijn, J de Groot, and Adriaan Cornelis Zaanen. *Introduction to metamathematics*, volume 483. van Nostrand New York, 1952.

[KEH$^+$09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[Kin69] James C King. A program verifier. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1969.

[KKP$^+$15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. *FAC*, 27(3):573–609, 2015.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd Edition*. Addison-Wesley, 1973.

[KR88] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988.

[Lea12] Doug Lea. `dlmalloc` memory allocator. `ftp://gee.cs.oswego.edu/pub/misc/malloc.c`, 2012.

[LG96] Doug Lea and Wolfram Gloger. A memory allocator, 1996.

[LR15] Jiangchao Liu and Xavier Rival. Abstraction of arrays based on non contiguous partitions. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 282–299. Springer, 2015.

[LTSC17] Quang Loc Le, Makoto Tatsuta, Jun Sun, and Wei-Ngan Chin. A decidable fragment in separation logic with inductive predicates and arithmetic. 2017.

[LYP11] Oukseh Lee, Hongseok Yang, and Rasmus Petersen. Program analysis for overlaid data structures. In *CAV*, volume 6806 of *LNCS*, pages 592–608. Springer, 2011.

[MAY06] Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. Formal verification of the heap manager of an operating system using separation logic. In *ICFEM*, volume 4260, pages 400–419. Springer, 2006.

[McM93] Kenneth L McMillan. Symbolic model checking. In *Symbolic Model Checking*, pages 25–60. Springer, 1993.

[Min01a] Antoine Miné. A new numerical abstract domain based on difference-bound matrices. In *Programs as Data Objects*, pages 155–172. Springer, 2001.

[Min01b] Antoine Miné. The octagon abstract domain. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 310–319. IEEE, 2001.

[Min06] Antoine Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *LCTES*, pages 54–63. ACM, 2006.

[MLL09] Qaisar A Malik, Johan Lilius, and Linas Laibinis. Model-based testing using scenarios and event-b refinements. *Methods, Models and Tools for Fault Tolerance*, 5454:177–195, 2009.

[Mor87] Joseph M Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer programming*, 9(3):287–306, 1987.

[MPS17] Louis Mussat, Thibaut Pierre, and Denis Sabatier. Safety analysis of a cbtc system: A rigorous approach with event-b. In *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification: Second International Conference, RSSRail 2017, Pistoia, Italy, November 14-16, 2017, Proceedings*, volume 10598, page 148. Springer, 2017.

[MRCR04] Miguel Masmano, Ismael Ripoll, Alfons Crespo, and Jorge Real. TLSF: A new dynamic memory allocator for real-time systems. In *ECRTS*, pages 79–86. IEEE Computer Society, 2004.

[NO79] Greg Nelson and Derek C Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.

[NPR11] Juan Antonio Navarro Pérez and Andrey Rybalchenko. Separation logic+ superposition calculus= heap theorem prover. *ACM SIGPLAN Notices*, 46(6):556–566, 2011.

[NPW02] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[ORY01] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Computer science logic*, pages 1–19. Springer, 2001.

[Pay07] Hannes Payer. A survey of dynamic real-time memory management systems. 2007.

[PN77] James L Peterson and Theodore A Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, 1977.

[Pua02] Isabelle Puaut. Real-time performance of dynamic memory allocation algorithms. In *Real-Time Systems, 2002. Proceedings. 14th Euromicro Conference on*, pages 41–49. IEEE, 2002.

[PWZ13] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating separation logic using smt. In *CAV*, volume 8044 of *LNCS*, pages 773–789. Springer, 2013.

[QGŞM13] Xiaokang Qiu, Pranav Garg, Andrei Ştefănescu, and Parthasarathy Madhusudan. Natural proofs for structure, data, and separation. *ACM SIGPLAN Notices*, 48(6):231–242, 2013.

[QHL$^+$14] Shengchao Qin, Guanhua He, Chenguang Luo, Wei-Ngan Chin, and Hongli Yang. Automatically refining partial specifications for heap-manipulating programs. *Science of Computer Programming*, 82:56–76, 2014.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[RJLP03] Lukas Ruf, Claudio Jeker, Boris Lutz, and Bernhard Plattner. Topsy v3: A nodeos for network processors. In *2nd International Workshop on Active Network Technologies and Applications (ANTA 2003)*, Osaka, Japan, May 2003.

[SA17] Wen Su and Jean-Raymond Abrial. Aircraft landing gear system: approaches with event-b to the modeling of an industrial system. *International Journal on Software Tools for Technology Transfer*, 19(2):141–166, 2017.

[SAPF15] Wen Su, Jean-Raymond Abrial, Geguang Pu, and Bin Fang. Formal development of a real-time operating system memory manager. In *20th International Conference on Engineering of Complex Computer Systems, ICECCS 2015, Gold Coast, Australia, December 9-12, 2015*, pages 130–139, 2015.

[SBDL01] Aaron Stump, Clark W Barrett, David L Dill, and Jeremy Levitt. A decision procedure for an extensional theory of arrays. In *Logic in Computer Science, 2001. Proceedings. 16th Annual IEEE Symposium on*, pages 29–37. IEEE, 2001.

[SL90] Douglas R Smith and Michael R Lowry. Algorithm theories and design tactics. *Science of Computer programming*, 14(2-3):305–321, 1990.

[SR12] Pascal Sotin and Xavier Rival. Hierarchical shape abstraction of dynamic structures in static blocks. In *APLAS*, volume 7705 of *LNCS*, pages 131–147. Springer, 2012.

[SWC07] XiaoHui Sun, JinLin Wang, and Xiao Chen. An improvement of tlsf algorithm. In *Real-Time Conference, 2007 15th IEEE-NPSS*, pages 1–5. IEEE, 2007.

[SZL$^+$15] Stephen F. Siegel, Manchun Zheng, Ziqing Luo, Timothy K. Zirkel, Andre V. Marianiello, John G. Edenhofner, Matthew B. Dwyer, and Michael S. Rogers. CIVL: The concurrency intermediate verification language. In *SC15: International Conference for High Performance Computing, Networking, Storage and Analysis, Proceedings*, SC ’15, pages 61:1–61:12, Piscataway, NJ, USA, Nov 2015. IEEE Press.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955.

[TKN07a] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In *ACM SIGPLAN Notices*, volume 42, pages 97–108. ACM, 2007.

[TKN07b] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In *ACM SIGPLAN Notices*, volume 42, pages 97–108. ACM, 2007.

[Ven96] Arnaud Venet. Abstract cofibered domains: Application to the alias analysis of untyped programs. In *SAS*, volume 1145 of *LNCS*, pages 366–382. Springer, 1996.

[WIR83] Niklaus WIRTH. Program development by stepwise refinement. *Communications of the ACM*, 26(1):70–74, 1983.

[WJNB95a] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *IWMM*, volume 986 of *LNCS*, pages 1–116. Springer, 1995.

[WJNB95b] Paul R Wilson, Mark S Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Memory Management*, pages 1–116. Springer, 1995.

[ZYSL15] Yongwang Zhao, Zhibin Yang, David Sanán, and Yang Liu. Event-based formalization of safety-critical operating system standards: An experience report on arinc 653 using event-b. In *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, pages 281–292. IEEE, 2015.