A Local Shape Analysis Based on Separation Logic : Detailed Presentation and Soundness Proof

Amin Timany *Bart Jacobs*

Report CW 661, May 2014

KU Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

A Local Shape Analysis Based on Separation Logic : Detailed Presentation and Soundness Proof

Amin Timany *Bart Jacobs*

Report CW 661, May 2014

Department of Computer Science, KU Leuven

Abstract

Shape analysis is a static analysis of the source code of a program to determine shapes and manipulations of the dynamically allocated data structures at each point which that program can reach in an execution. In this report, we give a detailed presentation and soundness proof of a shape analysis method which uses separation logic to represent program memory.

A Local Shape Analysis Based on Separation Logic : Detailed Presentation and Soundness Proof *

Amin Timany
amin.timany@cs.kuleuven.be

Bart Jacobs
bart.jacobs@cs.kuleuven.be

Computer Science Department
K. U. Leuven

1 Introduction

Shape analysis is a static analysis of the source code of a program to determine shapes and manipulations of the dynamically allocated data structures at each point which that program can reach in an execution.

To do so, the analysis presented in this report computes (an overapproximation of) the set of states that the program can possibly be in, at each point in the program. This is done by computing all possible states that the program can be in, before execution of each statement of the program, i.e., that statement's *invariant*.

Here, we use a denotational style semantics to define semantics of programs. In denotational semantics, we consider a domain set $\mathcal{D}$. The semantics of statements (also that of the whole program) is then defined as a function on that domain; i.e., the semantics of a statement c is $[\![c]\!] : \mathcal{D} \to \mathcal{D}$. Here, we use the power set, $2^{\mathfrak{S}}$, of a set of states, $\mathfrak{S}$, as the domain. As a result, given the invariant of a statement, computing the invariant of the next statement is straightforward. We simply apply the statement's semantics to its invariant and get the invariant of the next statement [1]. We assume that the set of all states that the program can possibly be in before execution of the program begins (program's precondition) is provided together with the program being analyzed. The analysis then computes invariants of all program statements and also the postcondition of the whole program.

As the set of states that the program can be in is generally not computable, accurate computation of invariants is impossible. Therefore, we compute a sound overapproximation of them. We compute an overapproximation so that we have a guarantee that whenever a state is part of an invariant, it is also represented as part of the overapproximation of that invariant computed by the analysis. Here, the word *sound* is put to emphasize the fact that if a program has a reachable faulty state, so does the overapproximation. This assures that whenever the analysis indicates that a program is free of memory faults it is indeed the case [2].

*Based on [DOY06], a paper with the same title by Dino Distefano, Peter W O'Hearn, and Hongseok Yang. Here, we give a rigorous presentation and soundness proof of the shape analysis approach they presented.

[1] In case the next statement is a loop, the result is not the invariant of the next statement (the loop) but, as we shall see, can be used to compute the invariant of that loop

[2] The other direction, of course, does not hold (the analysis can indicate that a program is unsafe while it is not) as a sound and complete analysis is non-computable as mentioned above

The term "program state", as repeatedly referred to above, is generic and can cover a wide range of possibilities. The states of a program may simply be representations of its store (assigning values to program variables) and heap or they can be describing properties about data structures residing in the heap such as sortedness of a linked list, it being the reverse of another linked list, etc. In this regard, there are shape analyses that can be used to obtain a wide range of information about the programs they analyze, e.g., those presented in [LRS04, OCDY06, CR08, CRN07]. In this report, we consider a very simple programming language, *SimpleLang*, in which all program variables are assumed to be pointers pointing to heap cells. Heap cells, then, are themselves considered as pointers pointing to other heap cells. In other words, this programming language can be used to write programs manipulating singly-linked lists where the data field(s) of the linked lists are abstracted away, i.e., each linked list cell is considered to only point to the next cell in the linked list.

As we will discuss shortly, we will define three semantics for this language. One is a low-level semantics considered as the reference semantics. This semantics simply represents states as stores and heaps, respectively, a mapping from program variables to heap cells and a partial mapping from heap cells to where they point to (in case allocated). The other two are high level semantics, representing states with a variant of symbolic heaps. Symbolic heaps are a special form of separation logic formulas defined for symbolic execution [BCO05]. This representation uses (separation logic) predicates to indicate fragments of heap containing acyclic linked list segments of length at least one. Choosing to represent states in this symbolic way, not only does it allow to confirm lack of memory faults from a successful analysis but also gives information about different linked lists residing in the heap and their cyclicities.

In the rest of this report, we describe the syntax of *SimpleLang* and define three different semantics for programs expressed in this language, namely, *concrete* semantics, *symbolic* semantics and *abstract* semantics.

In concrete semantics (Sem_κ), we use a low level representation of a store and a heap to describe states. In other words, a state is a pair of a heap and a store, or an error state representing a state where a memory fault has occurred. Heap cells are represented by natural numbers, their memory address. A store is represented by a mapping that maps program variables to heap cells or nil (the null pointer), and a heap is represented by a partial mapping that maps heap cells to heap cells or nil if they are allocated. Concrete semantics is the basic semantics defined to correspond to a low level interpretation of the language.

In symbolic semantics (Sem_σ), program states are defined with help of symbols. We use nil and program variables as symbols to stand for themselves, i.e., the null pointer and where program variables point respectively. We, in addition, introduce primed variables, to represent heap cells that are (possibly) not pointed to by program variables, or nil. Primed variables are by definition assumed to be existentially quantified in a state. A symbolic state then, is either an error state or consists of a pair of a spatial part and a pure part. The spatial part expresses spatial assertions, e.g., $Ls(x, y')$, which asserts that there is a linked list segment starting from the heap cell pointed to by program variable x to some other heap cell or nil as represented by primed variable y'. The pure part, on the other hand, expresses equalities between symbols, e.g., asserting y' equals nil.

In abstract semantics (Sem_α), on the other hand, we introduce an abstraction that is a mapping from symbolic states to (abstract) symbolic states. The abstract semantics, then, is symbolic semantics after which the abstraction is applied to the resulting states.

We will establish a modeling relation between concrete and symbolic states and show

	Sem_κ	Sem_σ	Sem_α
States	pairs of heaps and stores or error	pairs of spatial and pure (equality) assertions or error	Abstracted symbolic states
Overapproximates	–	Sem_κ	Sem_κ
$\lvert$ States $\rvert$	Infinite	Infinite	Finite
Computing Invariants	Non-computational	Non-computational	Computational

Table 1: Summary of properties of concrete, symbolic and abstract semantics

$$
\begin{array}{rcl}
x & \in & \mathit{Vars} \\
\mathit{stm} & ::= & \textbf{new}(x) \mid \textbf{free}(x) \mid \mathit{exp} := \mathit{exp} \mid \textbf{while}(\mathit{bexp})\ \{\mathit{stm}\} \mid \mathit{stm}; \mathit{stm} \\
\mathit{exp\text{-}s} & ::= & x \mid \textbf{nil} \\
\mathit{exp} & ::= & \mathit{exp\text{-}s} \mid x.\textbf{next} \\
\mathit{bexp} & ::= & \mathit{exp\text{-}s} == \mathit{exp\text{-}s} \mid \mathit{exp\text{-}s} \mathrel{!}= \mathit{exp\text{-}s}
\end{array}
$$

Figure 1: The grammar of the programming language

that symbolic semantics and abstract semantics are both sound overapproximations of concrete semantics. Furthermore, we show that the set of abstract states is finite which means abstract semantics can be used to compute invariants and ultimately for shape analysis. Table 1 shows a summary of properties of concrete, symbolic and abstract semantics and their relation.

In the rest of this report we assume that the reader is familiar with basic notions of lattice and domain theory, e.g., complete lattices, the fact that the power set of any set with subset relation forms a complete lattice, (Scott-)continuous functions, etc.

2 The Programming Language

Here, we define the simple language (*SimpleLang*) that we are going to use throughout the rest of the report. This language is minimally designed for the purpose of formalizing concepts presented in this report. *SimpleLang* can express programs that manipulate singly-linked lists where data field(s) are abstracted away, i.e., linked lists only have a "next" field. Here, we give the syntax of *SimpleLang* and discuss its general semantics.

2.1 Syntax

The BNF grammar of *SimpleLang* is given in Figure 1. In this figure, the words in bold are keywords and Vars is the set of program variables. A program is simply a statements (represented as stm in the grammar). In the sequel, we use $x, y, z, \ldots$ to refer to program variables. In addition, we use e, E, b and c (possibly indexed) to represent simple expressions ($\mathit{exp\text{-}s}$ in grammar), expressions (exp in grammar), boolean expressions (bexp in grammar) and statements (stm in grammar), respectively.

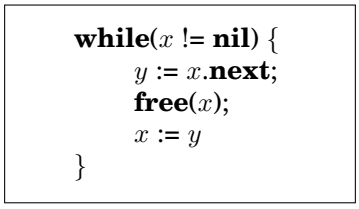

Figure 2: Dispose program: a program disposing a list

All program variables are assumed to be pointers that can be **nil** (the null pointer) or pointing to some memory location corresponding to a structure (in the sense of C structures) that has a next field. A program variable can be followed by a dot and the **next** keyword which indicates accessing the next field of the structure pointed to by that variable. Keywords **new** and **free** respectively allocate and deallocate memory cells. The **new**(x) statement changes the value of variable x so that it points to the newly allocated memory cell and the **free**(x) statement requires variable x to be pointing to some allocated memory cell which will be deallocated after its execution.

Effectively, this language can express programs that represent manipulations of singly linked data structures, e.g., cyclic or acyclic linked lists. As an example, Figure 2 depicts a program that disposes a list whose head is pointed to by x.

2.2 General Semantics

Here, we give a general account of semantics, we discuss how a semantics for *SimpleLang*, independent of the set of states, is defined. We use denotational semantics to define the semantics of programs. Assuming $\mathfrak{S}$ is the set of states, we define the domain of interpretation as $2^{\mathfrak{S}}$ (the power set of $\mathfrak{S}$). Hence, for a statement c, the semantics of c is a function $[\![c]\!] : 2^{\mathfrak{S}} \to 2^{\mathfrak{S}}$. In defining the semantics for a basic statement c (allocation, deallocation and assignment), we define a preliminary semantics $[\![c]\!]^\dagger : \mathfrak{S} \to 2^{\mathfrak{S}}$ and define the actual semantics of c based on that.

$$[\![c]\!](S) = \bigcup_{st \in S} [\![c]\!]^\dagger(st)$$

The semantics of the composition of statements is then defined as the composition of the semantics functions of those statements. In other words,

$$[\![c_1; c_2]\!] = [\![c_2]\!] \circ [\![c_1]\!]$$

Given the semantics of the basic statements and the composition of statements, we define the semantics of the while loops, independent of the set of states chosen. To do so, we use the function

$$\mathit{filter} : \mathit{bexp} \to \mathfrak{S} \to \mathfrak{S}$$

which, given a boolean expression (bexp) and a set of states, filters out all states that are not compatible with the given boolean expression.

Assuming $S_0 \subseteq \mathfrak{S}$ is the set of states before a while loop **while**$(b)\{c\}$, we define the function $f_{S_0}(S)$ as follows:

$$f_{S_0}(S) = S_0 \cup [\![c]\!] \circ \mathit{filter}(b)(S)$$

f_{S_0}, is the function that assuming that states in S are some states that the program can be in before the while loop (e.g., resulting from previous iterations of the loop), gets the new set

of states that the program can be in before the loop. In other words, considering S_0 as a set of states that we already know that the program can be in before entering the while loop, it evaluates the effect of one iteration of the while loop on members of S and considers the resulting states to also be part of the states that the program can be in before entering the while loop. In particular, assuming S_0 is the set of states that the program can be in before entering the while loop for the first time, $f^{n+1}_{S_0}(\varnothing)$ is the set of states that the program can be in before the while loop assuming the loop has already been evaluated up to n times. Given the definitions above, we can see that:

$$\begin{aligned} f_{S_0}(\varnothing) &= S_0 \\ f^2_{S_0}(\varnothing) &= S_0 \cup [\![c]\!] \circ \mathit{filter}(b)(S_0) \\ f^3_{S_0}(\varnothing) &= S_0 \cup [\![c]\!] \circ \mathit{filter}(b)(S_0) \cup ([\![c]\!] \circ \mathit{filter}(b))^2(S_0) \\ &\vdots \end{aligned}$$

Hence, the set of all states that the program can be in before the while loop is:

$$\varnothing \cup f_{S_0}(\varnothing) \cup f^2_{S_0}(\varnothing) \cup f^3_{S_0}(\varnothing) \cup \ldots \tag{1}$$

On the other hand, our domain, $2^{\mathfrak{S}}$, together with $\subseteq$ relation forms a complete lattice. Moreover, it is obvious, according to the definition of semantics as given above, that $[\![c]\!]$ for any basic statement c is continuous. For composition of statements, we will shortly discuss that the semantics defined below for the while loops (Formula 2, below) is also continuous, which shows continuousness of the semantics of the body of while loops; even if they contain other while loops. Hence, according to Kleene's fixpoint theorem, the Formula 1, above, is the least fixpoint of the function f_{S_0}. Therefore, we define the semantics of while loops as follows:

$$[\![\textbf{while}(b)\{c\}]\!](S) = \mathit{filter}(\neg b)(lfix(\lambda S'.S \cup [\![c]\!] \circ \mathit{filter}(b)(S'))) \tag{2}$$

where, $\neg b$ is simply the negation of b, i.e., the result of swapping "==" and "! =" in b.

Moreover, as alluded to earlier, the semantics definition of the while loops given above, is continuous. This we can see from the fact that:

$$\forall \mathcal{X} \subseteq 2^{\mathfrak{S}}.\ [\![\textbf{while}(b)\{c\}]\!](\bigcup_{X \in \mathcal{X}} X) = \bigcup_{X \in \mathcal{X}} [\![\textbf{while}(b)\{c\}]\!](X)$$

This means that the definition above for the semantics of the while loops can be used to give an inductively defined semantics for all of *SimpleLang*.

It is worth noting that the $f^n_{S_0}(\varnothing)$ is increasing with respect to $\subseteq$, i.e.,

$$\forall n \in \mathbb{N}.\ f^n_{S_0}(\varnothing) \subseteq f^{n+1}_{S_0}(\varnothing)$$

This can be easily shown with an induction over n. Hence, one can try to compute the fixpoint of the semantics of the while loop by computing $f_{S_0}(\varnothing), f^2_{S_0}(\varnothing), \ldots$ until reaching the least fixpoint, i.e., some n for which $f^{n+1}_{S_0}(\varnothing) = f^n_{S_0}(\varnothing)$. Although, in the general case, there is no guarantee that there is such a finite n for which $f^n_{S_0}(\varnothing)$ is the least fixpoint. However, for the case of abstract semantics, as we will discuss, since the set of states (and hence the domain which is its power set) is finite, such a finite n always exists. Throughout the rest of this report, we show example programs for which we compute the fixpoint using this approach and also discuss some examples for which no finite n exists such that $f^n_{S_0}(\varnothing)$ is the least fixpoint.

The way the semantics of the while loops are defined here is not the conventional way in denotational semantics. There, usually the fixpoint is applied to the semantics function itself; i.e., a lattice of (partial) semantics functions is defined for the while loop together with a completion operation on those partial functions (adding to the domain of the definition of the partial semantics functions). The semantics of the loops, then, is defined as the least fixpoint of this completion operation over the lattice of partial functions. Here, the choice to define the semantics of the while loops differently, is so that the fixpoint is applied to sets of states rather than (partial) semantics functions.

Here, we have shown a way to define the semantics of the language using the preliminary semantics for basic statements and the filter function, independent of the set of states chosen. Hence, in the rest of the report, in order to define a semantics, we simply define the set of states, the preliminary semantics for basic statements and the filter function for that semantics. Furthermore, to define the preliminary semantics of basic statements, we define a relation of the form $st, c \Rightarrow^{\dagger} st'$ where c is a basic statement and $st, st' \in \mathfrak{S}$. We then derive the preliminary semantics as follows:

$$[\![c]\!](S) = \{st' \mid st \in \mathfrak{S} \wedge st, c \Rightarrow^{\dagger} st'\}$$

3 Concrete Semantics

In order to give a basic definition of the behavior of programs written in our simple language, we define a denotational semantics called concrete semantics (Sem_κ). A state of Sem_κ is either an error state $\top_\kappa$ or a simple representation of program memory, i.e., a mapping from program variables to heap cells and a representation of the program heap.

Definition 3.1 (Concrete States)**.** Let $Vars$ be the set of program variables. Then, we define concrete program heaps, concrete stores and concrete states as follows:

- A heap $h \in \mathbb{H}_\kappa$ is a partial function $h : \mathbb{N} \rightharpoonup Values$
- A store $s \in \mathbb{ST}_\kappa$ is a function $s : Vars \to Values$
- The set of concrete states is $\mathfrak{S}_\kappa = (\mathbb{H}_\kappa \times \mathbb{ST}_\kappa) \cup \{\top_\kappa\}$

Where, $Values = \mathbb{N} \cup \{nil\}$. Furthermore, for a heap $h \in \mathbb{H}_\kappa$, we use $dom(h)$ to refer to all natural numbers for which h is defined. Where $a \in \mathbb{N}$ is a natural number for which $h(a)$ is undefined, we write $h(a) = \perp$. ■

Note that, nil and memory location 0 are different; the former is the null pointer while the latter is a valid memory address that can be allocated to and accessed, just like any other memory address.

To depict concrete states, we use a table to represent heap and store. As an example, the following table represents a concrete state $(h, s) \in \mathfrak{S}_\kappa$ of the program depicted in Figure 2. In this state, program variables x and y are respectively pointing to heap cell 10 and nil; while heap cell 10 points to heap cell 13 which in turn points to nil, i.e., x points to a linked list of length 2.

h		s	
address	value	variable	value
10	13	x	10
13	nil	y	nil

As we explained earlier, we only need to define the preliminary semantics for basic statements and the filter function and we do this by establishing a relation $\Rightarrow^{\dagger}_{\kappa}$ on statements and states.

In the sequel, we use $f[a \mapsto b]$ to denote the result of updating function f such that it maps a to b. In particular, $f[a \mapsto \bot]$ denotes the updating of partial function f such that it has no value for a.

$$f[a \mapsto b](d) \quad = \quad \begin{cases} b & \text{if } d = a \\ f(d) & \text{otherwise} \end{cases}$$

Definition 3.2. The rules defining the relation $\Rightarrow^{\dagger}_{\kappa}$ used to define the preliminary concrete semantics for basic program statements are as follows:

$$\frac{\text{any basic statement } c}{\top_\kappa, c \Rightarrow^{\dagger}_{\kappa} \top_\kappa} \qquad \frac{l \in \mathbb{N} \setminus dom(h) \text{ and } n \in Values}{(h,s), \textbf{new}(x); \Rightarrow^{\dagger}_{\kappa} (h[l \mapsto n], s[x \mapsto l])}$$

$$\frac{s(x) \in dom(h)}{(h,s), \textbf{free}(x) \Rightarrow^{\dagger}_{\kappa} (h[s(x) \mapsto \bot], s)} \qquad \frac{s(x) \notin dom(h)}{(h,s), \textbf{free}(x) \Rightarrow^{\dagger}_{\kappa} \top_\kappa}$$

$$\frac{}{(h,s), x := e \Rightarrow^{\dagger}_{\kappa} (h, s[x \mapsto val_\kappa(s,e)])} \qquad \frac{s(y) \in dom(h)}{(h,s), x := y.\textbf{next} \Rightarrow^{\dagger}_{\kappa} (h, s[x \mapsto h(s(y))])}$$

$$\frac{s(y) \notin dom(h)}{(h,s), x := y.\textbf{next} \Rightarrow^{\dagger}_{\kappa} \top_\kappa} \qquad \frac{s(x) \in dom(h)}{(h,s), x.\textbf{next} := e \Rightarrow^{\dagger}_{\kappa} (h[s(x) \mapsto val_\kappa(s,e)], s)}$$

$$\frac{s(x) \notin dom(h)}{(h,s), x.\textbf{next} := e \Rightarrow^{\dagger}_{\kappa} \top_\kappa} \qquad \frac{s(x), s(y) \in dom(h)}{(h,s), x.\textbf{next} := y.\textbf{next} \Rightarrow^{\dagger}_{\kappa} (h[s(x) \mapsto h(s(y))], s)}$$

$$\frac{s(x) \notin dom(h) \text{ or } s(y) \notin dom(h)}{(h,s), x.\textbf{next} := y.\textbf{next} \Rightarrow^{\dagger}_{\kappa} \top_\kappa}$$

■

Where $val_\kappa(s,e)$ for e, a simple expression, and $s \in \mathbb{ST}_\kappa$ is the concrete valuation of e under s. It is defined as follows:

$$val_\kappa(s,e) = \begin{cases} s(e) & \text{if } e \in Vars \\ nil & \text{if } e = nil \end{cases}$$

Definition 3.3. The concrete filter function $filter_\kappa$ is defined as follows:

$$\begin{array}{lcl} filter_\kappa(e_1 == e_2)(S) & = & \{(h,s) \in S \mid val_\kappa(s,e_1) = val_\kappa(s,e_2)\} \cup \{\top_\kappa \mid \top_\kappa \in S\} \\ filter_\kappa(e_1 != e_2)(S) & = & \{(h,s) \in S \mid val_\kappa(s,e_1) \neq val_\kappa(s,e_2)\} \cup \{\top_\kappa \mid \top_\kappa \in S\} \end{array}$$

■

This concludes the definition of concrete semantics. As an example, Figure 4 depicts a computation of the least fixpoint for the while loop of the dispose program depicted in Figure 2 by assuming an initial set S_0 of states that the program can be in before entering the loop. It also shows the result of the semantics of the while loop of this program applied to S_0.

On the other hand, Figure 5 shows that the least fixpoint of the semantics of the while loop in the program depicted in Figure 3 is not obtainable with iterations of computing $f^n_{S_0}(\varnothing)$.

```
while(nil == nil) {
    new(x);
}
```

Figure 3: Infinite allocation: a program that allocates heap cells indefinitely

$S_0 =$

h		s	
addr.	val.	var.	val.
10	13	x	10
13	nil	y	nil

$[\![dispose]\!]_\kappa(S_0) =$

h		s	
addr.	val.	var.	val.
		x	nil
		y	nil

n	$f^n_{S_0}(\varnothing)$
0	$\varnothing$

1

h		s	
addr.	val.	var.	val.
10	13	x	10
13	nil	y	nil

2

h		s		,	h		s	
addr.	val.	var.	val.		addr.	val.	var.	val.
10	13	x	10	,	13	nil	x	13
13	nil	y	nil				y	13

3

h		s		,	h		s		,	h		s	
addr.	val.	var.	val.		addr.	val.	var.	val.		addr.	val.	var.	val.
10	13	x	10	,	13	nil	x	13	,			x	nil
13	nil	y	nil				y	13				y	nil

Figure 4: Computation of the least fixpoint of the while loop semantics for the dispose program (Figure 2) in the concrete semantics. In this case, $f^3_{S_0}$ is the least fixpoint. In addition, $[\![dispose]\!]_\kappa$ is the concrete semantics of the whole dispose program (while loop).

$S_0 =$

h		s	
addr.	val.	var.	val.
		x	10

n	$f^n_{S_0}(\varnothing)$
0	$\varnothing$
1	$\{$ h: addr. | val. — ; s: var. | val. — x | 10 $\}$
2	$\{$ h: addr. | val. — 1 | nil ; s: var. | val. — x | 1 , h: addr. | val. — 1 | 0 ; s: var. | val. — x | 1 , h: addr. | val. — 2 | nil ; s: var. | val. — x | 2 , $\ldots$ $\}$
3	$\{$ h: addr. | val. — 1 | nil ; 2 | 0 ; s: var. | val. — x | 2 , h: addr. | val. — 1 | 0 ; 10 | nil ; s: var. | val. — x | 10 , h: addr. | val. — 2 | nil ; 10 | 13 ; s: var. | val. — x | 10 , $\ldots$ $\}$
$\vdots$	$\vdots$

Figure 5: Computation of the least fixpoint of the while loop semantics for the infinite allocation program (Figure 3) in the concrete semantics. In this case, there *does not* exist an $n \in \mathbb{N}$ such that $f^n_{S_0}$ is the least fixpoint.

4 Symbolic Semantics

In this section we introduce symbolic semantics for programs of *SimpleLang*. Symbolic states are high-level representations of program memory where actual addresses of cells are represented by symbols. Since we are using a symbolic representation of memory, we do not need stores and need only to represent heaps. To do so, we use a variant of symbolic heaps. Symbolic heaps are special separation logic formulas presented in [BCO05] to be used for symbolic execution. A symbolic heap is a separation logic formula consisting of conjunctions of a spatial formulas and pure assertions (assertions on equalities of symbolic expressions). Here, we define symbolic states to be pairs of separation logic formulas of spatial assertions and pure parts which represent equalities by explicit representation of equivalence classes over symbols.

The spatial assertions we use for the spatial part of symbolic states express facts of the form $x \mapsto y$ which signifies that the memory location indicated by symbol x represents a heap cell pointing to symbol y or $Ls(x, y)$ which signifies the fact that there is a linked list segment in the heap starting from memory location represented by symbol x ending in symbol y. We denote the set of symbols with Sym and it is defined as follows:

$$\mathsf{Sym} = \mathit{Vars} \cup \mathit{Vars}' \cup \{\mathit{nil}\}$$

where, Vars is the set of programs, they stand for memory locations pointed to by them, Vars' is a countably infinite set of primed variables $x', y', \ldots$ that represent some memory location (potentially) not pointed to by program variables and nil stands for the null pointer.

Definition 4.1 (Symbolic States)**.** Symbolic states are defined as follows:

– A spatial part $\Sigma \in \mathbb{SP}_\sigma$ is a separation logic formula defined below:

$$\begin{array}{lll} \varsigma_1, \varsigma_2 & \in & \mathsf{Sym} \\ \Phi & ::= & Ls(\varsigma_1, \varsigma_2) \mid \varsigma_1 \mapsto \varsigma_2 \mid \mathit{Junk} \mid \mathit{emp} \mid \Phi * \Phi \end{array}$$

– A pure part $\Pi \in \mathbb{PR}_\sigma$ is a set $\Pi : 2^{2^{\mathsf{Sym}}}$ of equivalence classes of symbols

– The set of symbolic states is $\mathfrak{S}_\sigma = (\mathbb{SP}_\sigma \times \mathbb{PR}_\sigma) \cup \{\top_\sigma\}$

Furthermore, we use $\varsigma_1 =_\Pi \varsigma_2$ to stand for the fact that $\varsigma_1 = \varsigma_2$ or ς_1 and ς_2 belong to the same equivalence class in Π and use $\varsigma_1 \neq_\Pi \varsigma_2$ as its negation. In cases where we do not distinguish between Ls and $\mapsto$, we use capital letter P (possibly indexed) to denote them, i.e., we implicitly assume that $P \in \{Ls, \mapsto\}$. ■

We assume that the primed variables that appear in a symbolic state are all existentially quantified. That is to say, by a symbolic state $(\Sigma, \Pi) \in \mathfrak{S}_\sigma$ in which primed variables $x'_1, \ldots, x'_n$ appear, we mean:

$$\exists x'_1, \ldots, x'_n.\ (\Sigma, \Pi)$$

In the above definition, $*$ is the separating conjunction. We later discuss the relation between concrete and symbolic semantics and give the exact interpretation of separating conjunction. For now, suffice it to say that a separation logic formula $A * B$ means that the heap of the program can be divided into two disjoint subheaps (no address is given a value in both subheaps) such that one is represented by A and the other is represented by B. A separation logic formula with no separating conjunction (a single instance of a predicate) is called a (heap or memory) chunk. As the intuitive meaning of separating conjunction explained here suggests, the separating conjunction (similarly to classical logic's conjunction connective) is commutative and associative. Therefore, in the sequel we consider two spatial parts of symbolic states equal if they only differ in the order of chunks. For further reading on separation logic, refer to the seminal paper [Rey] by J. C. Reynolds, where he introduces separation logic.

Intuitively, the heap chunk $Ls(\varsigma_1, \varsigma_2)$ represents (part of) a heap where a non-empty (i.e., $\varsigma_1 \neq \varsigma_2$) acyclic linked list in the heap starting in the memory location represented by ς_1 ending in the memory location (if $\varsigma_2 \neq \mathit{nil}$) represented by ς_2. The heap chunk $\varsigma_1 \mapsto \varsigma_2$ represents (part of) a heap where memory location represented by ς_1 points to memory location (if $\varsigma_2 \neq \mathit{nil}$) represented by ς_2. The heap chunk Junk, on the other hand, represents (a part of) heap that contains at least one allocated memory location. This chunk simply signifies memory leaks. Finally, the emp chunk represents (a part of) heap that is empty, i.e., no memory location is allocated. As emp chunk represents an empty (part of) heap, in the sequel, we assume two spatial parts Σ and $\Sigma * \mathit{emp}$ equal.

As we discussed earlier, all primed variables are implicitly assumed to be existentially quantified. Therefore, we consider two symbolic states equal if they are equal up to renaming of primed variables. That is, in addition to the fact that spatial parts are equal up to reordering of their chunks and addition of emp chunks.

Definition 4.2 (Symbolic State Equality)**.** Let $st_1, st_2 \in \mathfrak{S}_\sigma$ be two symbolic states. Then, st_1 and st_2 are equal ($st_1 = st_2$) if and only if

$$st_1 = st_2 = \top_\sigma$$

or

$$st_1 = (\Sigma_1, \Pi_1) \text{ and } st_2 = (\Sigma_2, \Pi_2) \text{ such that } st_1 = \hat{r}(st_2)$$

for some bijection $r : Vars' \to Vars'$. Where $\hat{r}(\Sigma, \Pi)$ is the simultaneous renaming of primed variables in Σ and Π according to r and the equality between spatial parts is considered up to reordering of their chunks and addition of emp chunks.

In the sequel, we will call such a renaming that makes two symbolic states equal their *equalizer renaming*. ■

Note that the definition above is obviously reflexive, symmetric and transitive, i.e., it is an equivalence relation.

Definition 4.3 (Consistency of Symbolic States)**.** A non-error symbolic state $(\Sigma, \Pi) \in \mathfrak{S}_\sigma$ is consistent (written as $(\Sigma, \Pi) \not\models \mathit{false}$) if *none* of the following holds:

I. There is a Σ' such that $\Sigma = Ls(\varsigma_1, \varsigma_2) * \Sigma'$ or $\Sigma = \varsigma_1 \mapsto \varsigma_2 * \Sigma'$ and $\varsigma_1 =_\Pi nil$

II. There a Σ' such that $\Sigma = P_1(\varsigma_1, \varsigma_2) * P_2(\varsigma_3, \varsigma_4) * \Sigma'$ and $\varsigma_1 =_\Pi \varsigma_3$

III. There is a Σ' such that $\Sigma = Ls(\varsigma_1, \varsigma_2) * \Sigma'$ such that $\varsigma_1 =_\Pi \varsigma_2$

■

Moreover, for $e_1, e_2 \in Vars \cup \{nil\}$, we need to be able to determine whether a non-error symbolic state $(\Sigma, \Pi) \in \mathfrak{S}_\sigma$ entails $e_1 = e_2$ or $e_1 \neq e_2$. For the case of equality, $e_1 = e_2$, we simply see if $e_1 =_\Pi e_2$. On the other hand, for the case of inequality, the entailment might be drawn from the spatial part. In particular, we say (Σ, Π) entails $e_1 \neq e_2$ if adding equality of e_1 and e_2 to it makes it inconsistent. This is formally defined in the following.

Definition 4.4 (Equalities and Inequalities with Respect to Symbolic States)**.** Let $(\Sigma, \Pi) \in \mathfrak{S}_\sigma$ be a symbolic state and $e_1, e_2 \in Vars \cup \{nil\}$. Then, the entailment of equality and inequality of e_1 and e_2 under (Σ, Π), written as $(\Sigma, \Pi) \vdash e_1 = e_2$ and $(\Sigma, \Pi) \vdash e_1 \neq e_2$ respectively, are defined as follows.

$$(\Sigma, \Pi) \vdash e_1 = e_2 \quad \text{if} \quad e_1 =_\Pi e_2$$
$$(\Sigma, \Pi) \vdash e_1 \neq e_2 \quad \text{if} \quad (\Sigma, \Pi[e_1 :=: e_2]) \vdash \mathit{false}$$

where, $\Pi[\varsigma_1 :=: \varsigma_2]$, for two symbols ς_1 and ς_2, is obtained by uniting the equivalence classes of ς_1 and ς_2 if they exist or otherwise adding a new equivalence class $\{\varsigma_1, \varsigma_2\}$. ■

In order to evaluate statements that access pointers x (e.g., **free**(x);), we need to explicitly have a points-to chunk, $x \mapsto \varsigma$ for some symbol ς. Although, this is not always the case when x represents an allocated heap cell; e.g., we have an $Ls(x, \varsigma)$ chunk instead. Therefore, we define a rearrangement function which, given a symbolic state, $st \in \mathfrak{S}_\sigma$, and a program variable x, gives a set of symbolic states all of which are of the form $(\Sigma' * x \mapsto \varsigma, \Pi')$ for some ς if x represents an allocated heap cell in st or gives $\{\top_\sigma\}$, otherwise. The rearrangement function is formally defined as follows.

Definition 4.5 (Rearrangement)**.** Let $st \in \mathfrak{S}_\sigma$ be a symbolic heap and $x \in Vars$ be a program variable. Then, the rearrangement of st to reveal x (written as $Rearr(st, x)$) is defined as follows:

$$Rearr(st, x) = \begin{cases} \{(\Sigma * x \mapsto \varsigma_2, \Pi)\} & \text{if } st = (\Sigma * \varsigma_1 \mapsto \varsigma_2, \Pi) \text{ for some } \varsigma_1, \varsigma_2 \in \mathsf{Sym} \\ & \quad \text{and } \varsigma_1 =_\Pi x \\ \{(\Sigma * x \mapsto \varsigma_2, \Pi), & \text{if } st = (\Sigma * Ls(\varsigma_1, \varsigma_2), \Pi) \text{ for } \varsigma_1, \varsigma_2 \in \mathsf{Sym}, \\ \quad (\Sigma * x \mapsto x' * Ls(x', \varsigma_2), \Pi)\} & \quad \varsigma_1 =_\Pi x \text{ and } x' \text{ is a fresh primed variable} \\ \{\top_\sigma\} & \text{otherwise} \end{cases}$$

■

Next, we define the preliminary symbolic semantics for basic statements and define the filter function for symbolic semantics.

Definition 4.6 (Preliminary Symbolic Semantics for Basic Statements)**.** The preliminary symbolic semantics for basic statements is defined as follows:

$$\frac{\text{any basic statement } c}{\top_\sigma, c \Rightarrow^\dagger_\sigma \top_\sigma} \qquad \frac{x', y' \text{ fresh primed variables}}{(\Sigma, \Pi), \textbf{new}(x) \Rightarrow^\dagger_\sigma (\Sigma[x'/x] * x \mapsto y', \Pi[x'/x])}$$

$$\frac{(\Sigma' * x \mapsto \varsigma, \Pi') \in \mathit{Rearr}((\Sigma, \Pi), x)}{(\Sigma, \Pi), \textbf{free}(x) \Rightarrow^\dagger_\sigma (\Sigma', \Pi')} \qquad \frac{\top_\sigma \in \mathit{Rearr}((\Sigma, \Pi), x)}{(\Sigma, \Pi), \textbf{free}(x) \Rightarrow^\dagger_\sigma \top_\sigma}$$

$$\frac{x' \text{ fresh primed variable}}{(\Sigma, \Pi), x := e \Rightarrow^\dagger_\sigma (\Sigma[x'/x], (\Pi[x'/x])[x :=: e[x'/x]])}$$

$$\frac{x' \text{ fresh primed variable}, (\Sigma' * y \mapsto \varsigma, \Pi') \in \mathit{Rearr}((\Sigma, \Pi), y)}{(\Sigma, \Pi), x := y.\textbf{next} \Rightarrow^\dagger_\sigma ((\Sigma' * y \mapsto \varsigma)[x'/x], (\Pi'[x'/x])[x :=: \varsigma[x'/x]])}$$

$$\frac{\top_\sigma \in \mathit{Rearr}((\Sigma, \Pi), y)}{(\Sigma, \Pi), x := y.\textbf{next} \Rightarrow^\dagger_\sigma \top_\sigma}$$

$$\frac{(\Sigma' * x \mapsto \varsigma, \Pi') \in \mathit{Rearr}((\Sigma, \Pi), x)}{(\Sigma, \Pi), x.\textbf{next} := e \Rightarrow^\dagger_\sigma (\Sigma' * x \mapsto e, \Pi')} \qquad \frac{\top_\sigma \in \mathit{Rearr}((\Sigma, \Pi), x)}{(\Sigma, \Pi), x.\textbf{next} := e \Rightarrow^\dagger_\sigma \top_\sigma}$$

$$\frac{(\Sigma' * x \mapsto \varsigma_1 * y \mapsto \varsigma_2, \Pi') \in \mathit{Rearr}((\Sigma, \Pi), x, y)}{(\Sigma, \Pi), x.\textbf{next} := y.\textbf{next} \Rightarrow^\dagger_\sigma (\Sigma' * x \mapsto \varsigma_2 * y \mapsto \varsigma_2, \Pi')}$$

$$\frac{\top_\sigma \in \mathit{Rearr}((\Sigma, \Pi), x, y)}{(\Sigma, \Pi), x.\textbf{next} := y.\textbf{next} \Rightarrow^\dagger_\sigma \top_\sigma}$$

where $A[\varsigma_1/\varsigma_2]$ is replacement of ς_2 with ς_1 in A and $\mathit{Rearr}(st, x, y)$ is short for

$$\bigcup_{st' \in \mathit{Rearr}(st, x)} \mathit{Rearr}(st', y)$$

■

Definition 4.7 (Symbolic Filter Function)**.** The symbolic filter function is defined as follows:

$$\mathit{filter}_\sigma(e_1 == e_2)(S) = \{(\Sigma, \Pi[e_1 :=: e_2]) \mid (\Sigma, \Pi) \in S \wedge (\Sigma, \Pi) \not\models e_1 \neq e_2\} \cup \{\top_\sigma \mid \top_\sigma \in S\}$$

$$\mathit{filter}_\sigma(e_1 \mathrel{!}= e_2)(S) = \{(\Sigma, \Pi) \mid (\Sigma, \Pi) \in S \wedge (\Sigma, \Pi) \not\models \mathit{false} \wedge (\Sigma, \Pi) \not\models e_1 = e_2\} \cup \{\top_\sigma \mid \top_\sigma \in S\}$$

■

Figure 6 shows the symbolic computation of the least fixpoint of the semantics of the while loop of the dispose program, depicted in Figure 2, starting in a set of states consisting of a single symbolic state where x points to a linked list ending in nil. In this case, there does not exist an $n \in \mathbb{N}$ such that $f^n_{S_0}(\varnothing)$ is the least fixpoint. This is due to the introduction of new primed variables which causes the pure part to grow indefinitely.

On the other hand, Figure 7 shows the symbolic computation of the least fixpoint of the semantics of the while loop of the infinite allocation program, depicted in Figure 3, starting in a set of states consisting of a single symbolic state where the heap is empty and there are no pure assertions. Evidently, in the case of symbolic semantics, similarly to the case of concrete semantics, the least fixpoint of the while loop in this program is not obtainable by iterative computation of $f^n_{S_0}(\varnothing)$.

$$S_0 = \{(Ls(l, nil), \varnothing)\}$$

n	$f^n_{S_0}(\varnothing)$
0	$\varnothing$
1	$\{(Ls(x, nil), \varnothing)\}$
2	$\{(Ls(x, nil), \varnothing), (emp, \{\{y, nil, x\}\}), (Ls(x'_1, nil), \{\{y, x'_1, x\}\})\}$
3	$\{(Ls(l, nil), \varnothing), (emp, \{\{n, nil, l\}\}), (Ls(x'_1, nil), \{\{x, x'_1, y\}\}),$ $(emp, \{\{x'_3, x'_2, x'_5\}, \{y, nil, x\}\}), (Ls(x'_4, nil), \{\{x, x'_4, x\}, \{x'_3, x'_2, x'_5\}\})\}$
$\vdots$	$\vdots$

Figure 6: Computation of the least fixpoint of the while loop semantics for the dispose program (Figure 2) in the symbolic semantics. In this case, there *does not* exist an $n \in \mathbb{N}$ such that $f^n_{S_0}$ is the least fixpoint.

$$S_0 = \{(emp, \varnothing)\}$$

n	$f^n_{S_0}(\varnothing)$
0	$\varnothing$
1	$\{(emp, \varnothing)\}$
2	$\{(emp * x \mapsto x'_1, \varnothing)\}$
3	$\{(emp * x \mapsto x'_3 * x'_2 \mapsto x'_1, \varnothing)\}$
4	$\{(emp * x \mapsto x'_5 * x'_4 \mapsto x'_3 * x'_2 \mapsto x'_1, \varnothing)\}$
$\vdots$	$\vdots$

Figure 7: Computation of the least fixpoint of the while loop semantics for the infinite allocation program (Figure 3) in the symbolic semantics. In this case, there *does not* exist an $n \in \mathbb{N}$ such that $f^n_{S_0}$ is the least fixpoint.

5 Overapproximation of Concrete Semantics by Symbolic Semantics

In this section, we show that the symbolic semantics is a sound overapproximation of the concrete semantics. Before we can establish this result though, we must define what constitutes a symbolic state representing a concrete state.

We say a concrete state (h, s) models a symbolic state (Σ, Π), if there is a $s' : \mathit{Vars}' \to \mathit{Values}$ such that the values assigned by s and s' are compatible with the equalities of Π and s and s' can be used to translate the symbols in Σ such that Σ holds in h.

Definition 5.1 (Modeling Relation)**.** Let (h, s) and (Σ, Π) be a concrete and symbolic state respectively. Then, we say (h, s) models (Σ, Π) (written as $(h, s) \vDash (\Sigma, \Pi)$) if there exists a $s' : \mathit{Vars}' \to \mathit{Values}$ such that:

$$\forall \varsigma_1, \varsigma_2 \in \mathsf{Sym}.\ \varsigma_1 =_\Pi \varsigma_2 \to \mathit{val}_\sigma(s, s', \varsigma_1) = \mathit{val}_\sigma(s, s', \varsigma_2)$$

and

$$(h, s) \overset{s'}{\vDash} \Sigma$$

where $\mathit{val}_\sigma(s, s', \varsigma)$ is defined as follows:

$$\mathit{val}_\sigma(s, s', \varsigma) = \begin{cases} s(\varsigma) & \text{if } \varsigma \in \mathit{Vars} \\ s'(\varsigma) & \text{if } \varsigma \in \mathit{Vars}' \\ \mathit{nil} & \text{if } \varsigma = \mathit{nil} \end{cases}$$

Furthermore, $(h, s) \overset{s'}{\vDash} \Sigma$ is defined inductively as follows:

$(h, s) \overset{s'}{\vDash} \mathit{emp}$ if and only if $\mathit{dom}(h) = \varnothing$

$(h, s) \overset{s'}{\vDash} \mathit{Junk}$ if and only if $\mathit{dom}(h) \neq \varnothing$

$(h, s) \overset{s'}{\vDash} \varsigma_1 \mapsto \varsigma_2$ if and only if $\varsigma_1 \neq \mathit{nil} \wedge h(a) = \begin{cases} \mathit{val}_\sigma(s, s', \varsigma_2) & \text{if } a = \mathit{val}_\sigma(s, s', \varsigma_1) \\ \bot & \text{otherwise} \end{cases}$

$(h, s) \overset{s'}{\vDash} \mathit{Ls}(\varsigma_1, \varsigma_2)$ if and only if $\varsigma_1 \neq \mathit{nil} \wedge \mathit{val}_\sigma(s, s', \varsigma_1) \neq \mathit{val}_\sigma(s, s', \varsigma_2) \wedge$
$((h, s) \overset{s'}{\vDash} \varsigma_1 \mapsto \varsigma_2 \vee (h, s) \overset{s''}{\vDash} (\varsigma_1 \mapsto z' * \mathit{Ls}(z', \varsigma_2)))$
for some fresh z' and $s'' = s'[z' \mapsto h(\mathit{val}_\sigma(s, s', \varsigma_1))]$

$(h, s) \overset{s'}{\vDash} F * G$ if and only if $h = h_1 \uplus h_2$ where $(h_1, s) \overset{s'}{\vDash} F \wedge (h_2, s) \overset{s'}{\vDash} G$

where, $h_1 \uplus h_2$ is disjoint union of h_1 and h_2 and is undefined if $\mathit{dom}(h_1) \cap \mathit{dom}(h_2) \neq \varnothing$. ■

Given the modeling relation, we define the the set of concrete states represented by a set of symbolic states as follows.

Definition 5.2 (Representation Function)**.** The representation function $\gamma : 2^{\mathfrak{S}_\sigma} \to 2^{\mathfrak{S}_\kappa}$ is the function that maps a set of symbolic states to the set of concrete states represented by them.

$$\gamma(S) = \begin{cases} \{(h, s) \in \mathfrak{S}_\kappa \mid (\Sigma, \Pi) \in S \wedge (h, s) \vDash (\Sigma, \Pi)\} & \text{if } \top_\sigma \notin S \\ \mathfrak{S}_\kappa & \text{otherwise} \end{cases}$$

■

In the following, we show that a symbolic state (Σ, Π) is inconsistent, i.e., $(\Sigma, \Pi) \vdash \mathit{false}$ if and only if there are no concrete states modeling it. Furthermore, we show that whenever two symbolic states are equal, a concrete state either models both of them or none of them. This means that $(\Sigma_1, \Pi_1) = (\Sigma_2, \Pi_2)$ implies, $\gamma(\{(\Sigma_1, \Pi_1)\}) = \gamma(\{(\Sigma_2, \Pi_2)\})$.

Theorem 5.3. Let (Σ, Π) be a symbolic state, then $(\Sigma, \Pi) \vdash \mathit{false}$ if and only if there is no concrete state (h, s) such that $(h, s) \models (\Sigma, \Pi)$. ■

Proof. We prove each side separately as follows:

$\Rightarrow$ $(\Sigma, \Pi) \vdash \mathit{false}$, then $\forall (h, s) \in \mathfrak{S}_\kappa.\ (h, s) \not\models (\Sigma, \Pi)$:
$(\Sigma, \Pi) \vdash \mathit{false}$ if at least one of the conditions I, II or III in 4.3 holds. We show that for each of these cases, (Σ, Π) has no model.

- I: There is a Σ' such that $\Sigma = Ls(\varsigma_1, \varsigma_2) * \Sigma'$ or $\Sigma = \varsigma_1 \mapsto \varsigma_2 * \Sigma'$ and we have $\varsigma_1 =_\Pi \mathit{nil}$. If $(h, s) \models (\Sigma, \Pi)$, there must exist an $s' : \mathit{Vars}' \to \mathit{Values}$ such that $\mathit{val}_\sigma(s, s', \varsigma_1) = \mathit{val}_\sigma(s, s', \mathit{nil}) = \mathit{nil}$ and we should have $\mathit{val}_\sigma(s, s', \varsigma_1) = \mathit{nil} \in \mathit{dom}(h)$ which is impossible as $h : \mathbb{N} \rightharpoonup \mathit{Values}$.
- II: There are two chunks $P_1(\varsigma_1, \varsigma_2)$ and $P_2(\varsigma_3, \varsigma_4)$ such that $\varsigma_1 =_\Pi \varsigma_3$. This means, there must be s' and a subheap h' such that $\mathit{dom}(h') \subseteq \mathit{dom}(h)$ and $(h', s) \overset{s'}{\models} P_1(\varsigma_1, \varsigma_2) * P_2(\varsigma_3, \varsigma_4)$. Thus, we should have $h' = h'_1 \uplus h'_2$ such that $(h'_1, s) \overset{s'}{\models} P_1(\varsigma_1, \varsigma_2)$ and $(h'_2, s) \overset{s'}{\models} P_2(\varsigma_3, \varsigma_4)$. This can not be as we have
$$\mathit{val}_\sigma(s, s', \varsigma_1) = \mathit{val}_\sigma(s, s', \varsigma_3) \in \mathit{dom}(h'_1) \cap \mathit{dom}(h'_2)$$
which contradicts the fact that $h' = h'_1 \uplus h'_2$, as it is undefined.
- III: There is a chunk $Ls(\varsigma_1, \varsigma_2)$ such that $\varsigma_1 =_\Pi \varsigma_2$. Then, there must be an s' and a subheap h' such that $\mathit{dom}(h') \subseteq \mathit{dom}(h)$ and $(h', s) \models Ls(\varsigma_1, \varsigma_2)$ which in turn means, we should have $\mathit{val}_\sigma(s, s', \varsigma_1) \neq \mathit{val}_\sigma(s, s', \varsigma_2)$ which is a contradiction (see Definition 5.1).

$\Leftarrow$: $\forall (h, s) \in \mathfrak{S}_\sigma.\ (h, s) \not\models (\Sigma, \Pi)$, then $(\Sigma, \Pi) \vdash \mathit{false}$:
Instead of a direct proof, we show that the contrapositive holds. Namely, we show $(\Sigma, \Pi) \not\vdash \mathit{false} \to \exists (h, s) \in \mathfrak{S}_\kappa.\ (h, s) \models (\Sigma, \Pi)$:

As for any heap h, it holds that $h = h \uplus h_\perp$ where $\mathit{dom}(h_\perp) = \varnothing$, whenever $(h, s) \models (\Sigma, \Pi)$, also $(h, s) \models (\Sigma * \mathit{emp}, \Pi)$. Therefore, in the rest of the proof we assume that there are no emp chunks in the spatial part.

Let $\Pi = \{\pi_0, \ldots, \pi_n\}$ then, we define $s : \mathit{Vars} \to \mathit{Values}$ and $s' : \mathit{Vars}' \to \mathit{Values}$ as follows:

$$s(x) = \begin{cases} i & \text{if } x \in \pi_i \wedge \mathit{nil} \notin \pi_i \\ \mathit{nil} & \text{if } \{x, \mathit{nil}\} \subseteq \pi_i \\ \mathcal{U}(x) & \text{otherwise} \end{cases} \qquad s'(x') = \begin{cases} i & \text{if } x' \in \pi_i \wedge \mathit{nil} \notin \pi_i \\ \mathit{nil} & \text{if } \{x', \mathit{nil}\} \subseteq \pi_i \\ \mathcal{U}(x') & \text{otherwise} \end{cases}$$

where $\mathcal{U} : \mathit{Vars} \cup \mathit{Vars}' \to \{n+m+1, n+m+2, \ldots\}$ is an injective mapping that maps each program variable or primed variable to some unique natural number greater than $n + m$, where m is the number of Junk chunks in Σ. Furthermore, let $h : \mathbb{N} \rightharpoonup \mathit{Values}$ be defined as follows:

$$h(a) = \begin{cases} \mathit{val}_\sigma(s, s', \varsigma_2) & \text{if } a = \mathit{val}_\sigma(s, s', \varsigma_1) \wedge \Sigma = P(\varsigma_1, \varsigma_2) * \Sigma' \\ n + i & \text{if } a = n + i \wedge 1 \leq i < m \\ \perp & \text{otherwise} \end{cases}$$

It is easy to see that $(h,s) \overset{s'}{\vDash} (\Sigma,\Pi)$, thus, we omit the proof here. □

Theorem 5.4. Let (Σ_1,Π_1) and (Σ_2,Π_2) be two symbolic states such that $(\Sigma_1,\Pi_1) = (\Sigma_2,\Pi_2)$. Then a concrete state either models both or neither of them. In other words,

$$\text{if } (\Sigma_1,\Pi_1) = (\Sigma_2,\Pi_2) \text{ then } \left(\forall (h,s) \in \mathfrak{S}_\kappa.\ (h,s) \vDash (\Sigma_1,\Pi_1) \text{ if and only if } (h,s) \vDash (\Sigma_2,\Pi_2)\right)$$

■

Proof. To prove this theorem, we assume $(\Sigma_1,\Pi_1) = (\Sigma_2,\Pi_2)$ and show that for an arbitrary concrete state $(h,s) \in \mathfrak{S}_\kappa$, $(h,s) \vDash (\Sigma_1,\Pi_1)$ then $(h,s) \vDash (\Sigma_2,\Pi_2)$. The other side (i.e., $(h,s) \vDash (\Sigma_2,\Pi_2)$ then $(h,s) \vDash (\Sigma_1,\Pi_1)$) follows through a similar reasoning and is therefore omitted.

Let $r : \mathit{Vars}' \to \mathit{Vars}'$ be the equalizer renaming of (Σ_1,Π_1) and (Σ_2,Π_2) and $s' : \mathit{Vars}' \to \mathit{Values}$ be such that $(h,s) \overset{s'}{\vDash} (\Sigma_1,\Pi_1)$. Then, $(h,s) \overset{s'\circ r}{\vDash} (\Sigma_2,\Pi_2)$.

This simply follows from the fact that

$$\forall x' \in \mathit{Vars}'.\ \mathit{val}_\sigma(s,s',x') = \mathit{val}_\sigma(s,s' \circ r, r(x'))$$

Which can be easily seen from the definition of $\mathit{val}_\sigma(s,s',\varsigma)$, for a symbol ς, in Definition 5.1. □

In preparation for the main result here, i.e., symbolic semantics being a sound overapproximation of concrete semantics, we show that the set of concrete states represented by a set of symbolic states is preserved under replacement of a symbol with a fresh primed variable. Furthermore, we show that the rearrangement to reveal a program variable is an overapproximation, i.e., if $(h,s) \vDash (\Sigma,\Pi)$ then $(h,s) \in \gamma(\mathit{Rearr}((\Sigma,\Pi),x))$ for any $x \in \mathit{Vars}$.

Lemma 5.5. Let (Σ,Π) be a symbolic state, $\varsigma \in \mathsf{Sym}$ be a symbol and $x' \in \mathit{Vars}'$ be a primed variable that does not appear in (Σ,Π) . Then,

$$\gamma(\{(\Sigma,\Pi)\}) \subseteq \gamma(\{(\Sigma,\Pi)[x'/\varsigma]\})$$

■

Proof. Let $s'_1 : \mathit{Vars}' \to \mathit{Values}$ and $s'_2 : \mathit{Vars}' \to \mathit{Values}$ be two valuation functions for primed variables such that:

$$s'_2(y') = \begin{cases} \mathit{val}_\sigma(s,s'_1,\varsigma) & \text{if } y' = x' \\ s'_1(y') & \text{otherwise} \end{cases}$$

We show that for any $(h,s) \in \mathfrak{S}_\kappa$,

$$(h,s) \overset{s'_1}{\vDash} (\Sigma,\Pi) \to (h,s) \overset{s'_2}{\vDash} (\Sigma,\Pi)[x'/\varsigma]$$

To show this, we only need to show that for any symbol $\varsigma_1 \in \mathsf{Sym}$,

$$\mathit{val}_\sigma(s,s'_1,\varsigma_1) = \mathit{val}_\sigma(s,s'_2,\varsigma_1[x'/\varsigma])$$

This, on the other hand, is obvious from the definition of s'_2 above. □

Lemma 5.6. Let $st \in \mathfrak{S}_\sigma$ be a symbolic state and $x \in \mathit{Vars}$ be a program variable. Then,

$$\gamma(\{st\}) \subseteq \gamma(\mathit{Rearr}(st,x))$$

■

Proof. If $st = \top_\sigma$, $\gamma(Rearr(\top_\sigma, x)) = \gamma(\{\top_\sigma\}) = \mathfrak{S}_\kappa$. If $st = (\Sigma, \Pi)$, we can have three cases:

- There is a Σ' such that $\Sigma = \varsigma_1 \mapsto \varsigma_2 * \Sigma'$ for some ς_2 and $\varsigma_1 =_\Pi x$. In this case, $Rearr((\Sigma, \Pi), x) = \{(x \mapsto \varsigma_2 * \Sigma', \Pi')\}$. In addition, if $(h, s) \vDash (\Sigma, \Pi)$, there is an s', h_1, h_2 such that $h = h_1 \uplus h_2$ and $(h_1, s) \overset{s'}{\vDash} (\varsigma_1 \mapsto \varsigma_2, \Pi)$ and $(h_2, s) \overset{s'}{\vDash} (\Sigma', \Pi)$. On the other hand, $\varsigma_1 =_\Pi x$ and hence $val_\sigma(s, s', x) = val_\sigma(s, s', \varsigma_1)$. Thus, $(h_1, s) \overset{s'}{\vDash} (x \mapsto \varsigma_2, \Pi)$ and as a result
$$(h, s) \vDash (x \mapsto \varsigma_2 * \Sigma', \Pi')$$
- There is a Σ' such that $\Sigma = Ls(\varsigma_1, \varsigma_2) * \Sigma'$ for some ς_2 and $\varsigma_1 =_\Pi x$. In this case,
$$Rearr((\Sigma, \Pi), x) = \{\underbrace{(x \mapsto \varsigma_2 * \Sigma', \Pi)}_{st_1}, \underbrace{(x \mapsto x' * Ls(x', \varsigma_2) * \Sigma', \Pi)}_{st_2}\}$$
According to Definition 5.1 and a reasoning similar to that of the previous case, if $(h, s) \vDash (\Sigma, \Pi)$, then, either $(h, s) \vDash st_1$ or $(h, s) \vDash st_1$.
- None of the above hold. In this case, $\gamma(Rearr((\Sigma, \Pi), x)) = \gamma(\{\top_\sigma\}) = \mathfrak{S}_\kappa$.

□

Lemma 5.7 (Symbolic Semantics of Basic Statements Overapproximates Concrete Semantics)**.** Let c be a basic statement. Then,
$$\forall S \subseteq \mathfrak{S}_\sigma.\ [\![c]\!]_\kappa(\gamma(S)) \subseteq \gamma([\![c]\!]_\sigma(S))$$

■

Proof. Since semantics functions and representation function, γ, are both continuous, we simply need to show that
$$\forall st \in \mathfrak{S}_\sigma.\ [\![c]\!]_\kappa(\gamma(\{st\})) \subseteq \gamma([\![c]\!]_\sigma(\{st\}))$$
If $st = \top_\sigma$,
$$\gamma([\![c]\!]_\sigma(\{\top_\sigma\})) = \gamma(\top_\sigma) = \mathfrak{S}_\kappa$$
If $st = (\Sigma, \Pi)$,

- $c = \textbf{new}(x)$;

  For $st \in [\![c]\!]_\sigma(\{(\Sigma, \Pi)\})$, we have $st = (x \mapsto y' * \Sigma[x'/x], \Pi[x'/x])$ where x' and y' are some fresh primed variables.

  On the other hand, if $(h, s) \vDash (\Sigma, \Pi)$, i.e., $(h, s) \in \gamma(\{(\Sigma, \Pi)\})$, then, for any $st' \in [\![c]\!]_\kappa(h, s)$, we have,
  $$st' = (h[m \mapsto n], s[x \mapsto m])$$
  for some $m \notin dom(h)$ and $n \in \mathbb{N}$. Furthermore, $h[m \mapsto n] = h \uplus h'$ where,
  $$h'(x) = \begin{cases} n & \text{if } x = m \\ \bot & \text{otherwise} \end{cases}$$
  According to Lemma 5.5, $(h, s) \vDash (\Sigma[x'/x], \Pi[x'/x])$. On the other hand, since x does not appear in $(\Sigma[x'/x], \Pi[x'/x])$, we have
  $$(h, s[x \mapsto m]) \vDash (\Sigma[x'/x], \Pi[x'/x])$$

In addition,

$$(h', s[x \mapsto m]) \vDash (x \mapsto y', \Pi[x'/x])$$

Thus,

$$(h[m \mapsto n], s[x \mapsto m]) \vDash (x \mapsto y' * \Sigma[x'/x], \Pi[x'/x])$$

Consequently,

$$(h[m \mapsto n], s[x \mapsto m]) \in \gamma([\![c]\!]_\sigma(\{(\Sigma, \Pi)\}))$$

– $c = \textbf{free}(x)$;

If $Rearr((\Sigma, \Pi), x) = \{\top_\sigma\}$, it trivially holds.

Otherwise, for $(h, s) \vDash (\Sigma, \Pi)$, According to Lemma 5.6, $(h, s) \in \gamma(Rearr((\Sigma, \Pi), x))$. Hence, there is a $(\Sigma' * x \mapsto \varsigma, \Pi) \in Rearr((\Sigma, \Pi), x)$ such that

$$(h, s) \vDash (x \mapsto \varsigma * \Sigma', \Pi)$$

Therefore, $h = h_1 \uplus h_2$ such that $(h_1, s) \vDash (x \mapsto \varsigma, \Pi)$ and $(h_2, s) \vDash (\Sigma' * \Pi)$. Thus, for $(h[s(x) \mapsto \bot], s) \in [\![c]\!]_\kappa(h, s)$,

$$(h[s(x) \mapsto \bot], s) = (h_2, s) \in \gamma(\Sigma' * \Pi)$$

Consequently,

$$(h[s(x) \mapsto \bot], s) \in \gamma([\![c]\!]_\sigma(\{(\Sigma, \Pi)\}))$$

– $c = x := e$;

Let x' be a fresh primed variable and (h, s) be a concrete state such that $(h, s) \vDash (\Sigma, \Pi)$. According to Lemma 5.5, we have $(h, s) \vDash (\Sigma[x'/x], \Pi[x'/x])$. Here, we consider two cases, first if x and e are the same (an assignment of x to itself!) and second if x and e are two different simple expressions.

If x and e are the same variables, then we have $s[x \mapsto val_\kappa(s, e)]) = s$ and

$$(\Sigma[x'/x], (\Pi[x'/x])[x :=: (y[x'/x])]) = (\Sigma[x'/x], \Pi[x'/x])$$

If x and e are two different symbols, we have $(e[x'/x]) = e$ and thus, we have to show

$$(h, (s[x \mapsto val_\kappa(s, e)])) \vDash (\Sigma[x'/x], (\Pi[x'/x])[x :=: e])$$

To see this, observe that

$$(h, (s[x \mapsto val_\kappa(s, e)])) \vDash (\Sigma[x'/x], \Pi[x'/x])$$

Notice that x does not appear in $(\Sigma[x'/x], \Pi[x'/x])$ and hence the value of $s(x)$ is irrelevant. Thus, there is an s' such that

$$(h, (s[x \mapsto val_\kappa(s, e)])) \overset{s'}{\vDash} (\Sigma[x'/x], \Pi[x'/x])$$

On the other hand, since

$$\forall \varsigma_1, \varsigma_1 \in \mathsf{Sym}.\ \varsigma_1 =_\Pi \varsigma_2 \to val_\sigma(s, s', \varsigma_1) = val_\sigma(s, s', \varsigma_2)$$

we have

$$(h, (s[x \mapsto val_\kappa(s, e)])) \overset{s'}{\vDash} (\Sigma[x'/x], (\Pi[x'/x])[x :=: e])$$

Consequently,

$$(h, (s[x \mapsto val_\kappa(s, e)])) \in \gamma([\![c]\!]_\sigma(\{(\Sigma, \Pi)\}))$$

– $c = x.\textbf{next} := e$;

If $Rearr((\Sigma, \Pi), x) = \{\top_\sigma\}$, it trivially holds.

Otherwise, for any $(h, s) \vDash (\Sigma, \Pi)$, according to Lemma 5.6, there is a $(\Sigma' * x \mapsto \varsigma, \Pi) \in Rearr((\Sigma, \Pi), x)$, such that

$$(h, s) \vDash (\Sigma' * x \mapsto \varsigma, \Pi)$$

On the other hand, since $(h, s) \vDash (\Sigma' * x \mapsto \varsigma, \Pi)$, we have $h = h_1 \uplus h_2$, such that $(h_1, s) \vDash (\Sigma', \Pi)$ and $(h_2, s) \vDash (x \mapsto \varsigma, \Pi)$. Therefore, since $(h_2[x \mapsto val_\kappa(e)], s) \vDash (x \mapsto e, \Pi)$, we have

$$(h[x \mapsto val_\kappa(s, e)], s) \vDash (\Sigma' * x \mapsto e, \Pi) \in [\![c]\!]_\sigma(\{(\Sigma, \Pi)\})$$

Consequently,

$$(h[x \mapsto val_\kappa(s, e)], s) \in \gamma([\![c]\!]_\sigma(\{(\Sigma, \Pi)\}))$$

– $c = x := y.\textbf{next}$;

If $Rearr((\Sigma, \Pi), y) = \{\top_\sigma\}$, it trivially holds.

Otherwise, for any $(h, s) \vDash (\Sigma, \Pi)$, according to Lemma 5.6, we have

$$(h, s) \in \gamma(Rearr((\Sigma, \Pi), y))$$

Hence, there is a $(\Sigma' * y \mapsto \varsigma, \Pi) \in Rearr((\Sigma, \Pi), y)$ such that

$$(h, s) \vDash (\Sigma' * y \mapsto \varsigma, \Pi)$$

Now, let x' be a fresh primed variable. Then, according to Lemma 5.5, $(h, s) \vDash ((\Sigma' * y \mapsto \varsigma)[x'/x], \Pi[x'/x])$.

Here, we should consider two cases, first if ς and x are the same (y's next cell is x) and second if ς and x are two different symbols.

If x and ς are the same variables, then, since $h(s(y)) = s(x)$, we have

$$s[x \mapsto h(s(y))] = s$$

and

$$((\Sigma'' * y \mapsto \varsigma)[x'/x], ((\Pi[x'/x])[x :=: (\varsigma[x'/x])])) = ((\Sigma'' * y \mapsto \varsigma)[x'/x], \Pi[x'/x])$$

Thus,

$$(h, s[x \mapsto h(s(y))]) \in \gamma([\![c]\!]_\sigma(\{(\Sigma, \Pi)\}))$$

On the other hand, if x and ς are two different symbols, since $(h, s) \vDash ((\Sigma' * y \mapsto \varsigma)[x'/x], \Pi[x'/x])$, we have

$$(h, s[x \mapsto h(s(y))]) \vDash ((\Sigma' * y \mapsto \varsigma)[x'/x], \Pi[x'/x])$$

Notice that x does not appear in $((\Sigma' * y \mapsto \varsigma)[x'/x], \Pi[x'/x])$ and thus the value of $s(x)$ is irrelevant. Thus, there is a s' such that

$$(h, (s[x \mapsto h(s(y))])) \overset{s'}{\vDash} (\Sigma[x'/x], \Pi[x'/x])$$

On the other hand, since

$$\forall \varsigma_1, \varsigma_2 \in \mathsf{Sym}.\ \varsigma_1 =_{\Pi} \varsigma_2 \rightarrow val_\sigma(s, s', \varsigma_1) = val_\sigma(s, s', \varsigma_2)$$

we have,

$$\forall \varsigma_1, \varsigma_2 \in \mathsf{Sym}.\ \varsigma_1 =_{(\Pi[x'/x])[x:=:\varsigma]} \varsigma_2 \rightarrow val_\sigma((s[x \mapsto s(y)]), s', \varsigma_1) = val_\sigma((s[x \mapsto s(y)]), s', \varsigma_2)$$

and as a result,

$$(h, (s[x \mapsto s(y)])) \overset{s'}{\vDash} (\Sigma[x'/x], (\Pi[x'/x])[x :=: \varsigma])$$

Consequently,

$$(h, (s[x \mapsto s(y)])) \in \gamma([\![c]\!]_\sigma(\{(\Sigma, \Pi)\}))$$

– $c = x.\mathbf{next} := y.\mathbf{next}$;

If $Rearr((\Sigma, \Pi), x, y) = \{\top_\sigma\}$, it trivially holds.

Otherwise, for any $(h, s) \vDash (\Sigma, \Pi)$, according to Lemma 5.6, there is

$$(\Sigma' * x \mapsto \varsigma_1 * y \mapsto \varsigma_2, \Pi') \in Rearr((\Sigma, \Pi), x, y)$$

such that $(h, s) \vDash (\Sigma' * x \mapsto \varsigma_1 * y \mapsto \varsigma_2, \Pi')$.

On the other hand, since $(h, s) \vDash (\Sigma' * x \mapsto \varsigma_1 * y \mapsto \varsigma_2, \Pi)$, we have $h = h_1 \uplus h_2 \uplus h_3$, such that

$$(h_1, s) \vDash (\Sigma', \Pi),\ \ (h_2, s) \vDash (x \mapsto \varsigma_1, \Pi)\ \text{ and }\ (h_3, s) \vDash (y \mapsto \varsigma_2, \Pi)$$

Therefore, we have

$$(h[x \mapsto h(s(y))], s) \vDash (\Sigma' * x \mapsto \varsigma_2 * y \mapsto \varsigma_2, \Pi)$$

Consequently,

$$(h[x \mapsto h(s(y)))], s) \in \gamma([\![c]\!]_\sigma(\{(\Sigma, \Pi)\}))$$

□

After proving overapproximation results for basic statements, we show overapproximation of the filter function before proving the final result, i.e., overapproximation for the whole of *SimpleLang*.

Lemma 5.8. Let $S \subseteq \mathfrak{S}_\sigma$ be a set of symbolic states and $cond$ be a loop condition of the form $e1$ == e_1 or e_1 != e_2 for some simple expressions e_1 and e_2. Then,

$$\mathit{filter}_\kappa(cond)(\gamma(S)) \subseteq \gamma(\mathit{filter}_\sigma(cond)(S))$$

■

Proof. Since the representation function, γ, and both filter functions are continuous, the results for singletons naturally extend to all sets. In other words, we only need to show that,

$$\forall st \in \mathfrak{S}_\sigma.\ \mathit{filter}_\kappa(cond)(\gamma(\{st\})) \subseteq \gamma(\mathit{filter}_\sigma(cond)(\{st\}))$$

If $st = \top_\sigma$, $\gamma(\mathit{filter}_\sigma(cond)(\{st\})) = \mathfrak{S}_\kappa$. On the other hand, if $st = (\Sigma, \Pi)$ and $(\Sigma, \Pi) \vdash \mathit{false}$, according to Theorem 5.3 and Definition 5.2, $\gamma(\{(\Sigma, \Pi)\}) = \varnothing$ and thus,

$$\mathit{filter}_\kappa(cond)(\gamma(\{(\Sigma, \Pi)\})) = \varnothing$$

Therefore, in the rest of the proof we consider $st = (\Sigma, \Pi)$ such that $(\Sigma, \Pi) \nvdash \mathit{false}$. We proceed by case analysis on $cond$.

Case: $cond$ = 'e_1 == e_2'

If $(h,s) \in \mathit{filter}_\kappa(cond)(\gamma(\{(\Sigma,\Pi)\}))$, we have $(h,s) \vDash (\Sigma,\Pi)$ and $val_\kappa(s,e_1) = val_\kappa(s,e_2)$. As a result, $(h,s) \vDash (\Sigma,\Pi[e_1 :=: e_2])$. This, according to Theorem 5.3 and Definition 4.4, shows that $(\Sigma,\Pi) \nvdash e_1 \neq e_2$. Hence,

$$(\Sigma,\Pi[e_1 :=: e_2]) \in \mathit{filter}_\sigma(cond)(\Sigma,\Pi)$$

Consequently,

$$(h,s) \in \gamma(\mathit{filter}_\sigma(cond)(\{st\}))$$

Case: $cond$ = 'e_1 != e_2'

If $(h,s) \in \mathit{filter}_\kappa(cond)(\gamma(\{(\Sigma,\Pi)\}))$, we have $(h,s) \vDash (\Sigma,\Pi)$ and $val_\kappa(s,e_1) \neq val_\kappa(s,e_2)$. According to Definition 4.4, we have $(\Sigma,\Pi) \nvdash e_1 = e_2$. Hence,

$$(\Sigma,\Pi) \in \mathit{filter}_\sigma(cond)(\Sigma,\Pi)$$

Consequently,

$$(h,s) \in \gamma(\mathit{filter}_\sigma(cond)(\{st\}))$$

□

Finally, we can state and prove the main theorem of this section, i.e., the correspondence of semantics.

Theorem 5.9 (Symbolic Semantics Overapproximates Concrete Semantics)**.** Let c be any statement or block of statements. Then,

$$\forall S \subseteq \mathfrak{S}_\sigma.\ [\![c]\!]_\kappa(\gamma(S)) \subseteq \gamma([\![c]\!]_\sigma(S))$$

■

Proof. We prove this theorem by structural induction on the structure of statements:

Base Case: The statement c is a basic statement. This is already proven in Lemma 5.7.

Inductive Case1: The statement c is of the form $c_1; c_2$ and, as induction hypothesis, we have:

$$\forall S \subseteq \mathfrak{S}_\sigma.\ [\![c_1]\!]_\kappa(\gamma(S)) \subseteq \gamma([\![c_1]\!]_\sigma(S)) \text{ and } \forall S \subseteq \mathfrak{S}_\sigma.\ [\![c_2]\!]_\kappa(\gamma(S)) \subseteq \gamma([\![c_2]\!]_\sigma(S))$$

From the definition of semantics of composition of statements, $[\![c_1;c_2]\!] = [\![c_2]\!] \circ [\![c_1]\!]$, it follows that:

$$\forall S \subseteq \mathfrak{S}_\sigma.\ [\![c_1;c_2]\!]_\kappa(\gamma(S)) \subseteq \gamma([\![c_1;c_2]\!]_\sigma(S))$$

Inductive Case2: The statement c is of the form **while**$(b)\{c'\}$ and, as induction hypothesis, we have:

$$\forall S \subseteq \mathfrak{S}_\sigma.\ [\![c']\!]_\kappa(\gamma(S)) \subseteq \gamma([\![c']\!]_\sigma(S))$$

Assuming $S \subseteq \mathfrak{S}_\sigma$ is a set of symbolic states, we show that:

$$[\![\textbf{while}(b)\{c'\}]\!]_\kappa(\gamma(S)) \subseteq \gamma([\![\textbf{while}(b)\{c'\}]\!]_\sigma(S))$$

Assuming that $f_{\kappa,\gamma(S)}$ and $f_{\sigma,S}$ are as defined for general semantics of the while loops, respectively for concrete and symbolic semantics, we have:

$$[\![\textbf{while}(b)\{c'\}]\!]_\kappa(\gamma(S)) = \mathit{filter}_\kappa(\neg b)(\bigcup_{n\in\mathbb{N}} f^n_{\kappa,\gamma(S)})$$

and

$$[\![\textbf{while}(b)\{c'\}]\!]_\sigma(S) = \mathit{filter}_\sigma(\neg b)(\bigcup_{n\in\mathbb{N}} f^n_{\sigma,S})$$

By a simple induction on n, using induction hypothesis and Lemma 5.8, we can see that:

$$\forall n \in . \ f^n_{\kappa,\gamma(S)} \subseteq f^n_{\sigma,S}$$

Thus,

$$\bigcup_{n\in\mathbb{N}} f^n_{\kappa,\gamma(S)} \subseteq \bigcup_{n\in\mathbb{N}} f^n_{\sigma,S}$$

On the other hand, according to Lemma 5.8, we have:

$$\mathit{filter}_\sigma(\neg b)(\bigcup_{n\in\mathbb{N}} f^n_{\sigma,S}) \subseteq \mathit{filter}_\kappa(\neg b)(\bigcup_{n\in\mathbb{N}} f^n_{\kappa,\gamma(S)})$$

Consequently,

$$[\![\textbf{while}(b)\{c'\}]\!]_\sigma(S) \subseteq [\![\textbf{while}(b)\{c'\}]\!]_\kappa(\gamma(S))$$

□

As a corollary, we get that whenever the concrete execution of the program, starting in the empty memory, indicates an error in the program, so does the symbolic execution, starting in the empty memory.

Corollary 5.10. For a statement (program) c,

$$\text{if } \ \top_\kappa \in [\![c]\!]_\kappa(h_\varepsilon, s_\varepsilon) \ \text{ then } \ \top_\sigma \in [\![c]\!]_\sigma(emp, \varnothing)$$

where,

$$h_\varepsilon(a) = \bot \ \text{ and } \ s_\varepsilon(x) = nil$$

■

6 Abstract Semantics

In this section we discuss an abstraction that maps symbolic states to abstract symbolic states. Abstract semantics then is simply symbolic semantics having abstraction applied to their results.

We define the abstraction in two phases; in the first phase, we remove all primed variables from the pure part and in the second phase, we contract the spatial part by collapsing consecutive chains of linked list segments (if possible) and gathering all memory leaks into a single Junk predicate.

Definition 6.1 (Removing Primed Variables from Pure Part)**.** Let (Σ, Π) be a symbolic state and $\pi \in \Pi$ be an equivalence class in the pure part. The removal of primed variables appearing in π, denoted as $rp_\pi(\Sigma, \Pi)$ is defined as follows:

$$rp_\pi(\Sigma, \Pi) = (\Sigma[\underbrace{rep(\pi), \ldots, rep(\pi)}_{n \text{ times}}/x'_1, \ldots, x'_n], (\Pi \setminus \{\pi\}) \cup \xi(\pi))$$

where

$$rep(\pi) = \begin{cases} x \in \pi \setminus \{x'_1, \ldots, x'_n\} & \text{if } \pi \setminus \{x'_1, \ldots, x'_n\} \neq \varnothing \\ x'_i & \text{for some } 1 \leq i \leq n \end{cases}$$

and

$$\xi(\pi) = \begin{cases} \{\pi \setminus \{x'_1, \ldots, x'_n\}\} & \text{if } |\pi \setminus \{x'_1, \ldots, x'_n\}| \geq 1 \\ \varnothing & \text{otherwise} \end{cases}$$

and $\{x'_1, \ldots, x'_n\} \subseteq \pi$ is the set of all primed variables in π and $[b_1, \ldots, b_n / a_1, \ldots, a_n]$ denotes simultaneous replacement of a_1 by $b1$, $\ldots$, a_n by b_n.

In the sequel we will use $rp_\Pi(\Sigma, \Pi)$ defined as

$$rp_\Pi(\Sigma, \Pi) = rp_{\pi_1}(\ldots(rp_{\pi_m}(\Sigma, \Pi))\ldots) \text{ where } \Pi = \{\pi_1, \ldots, \pi_m\}$$

to be the result of removal of all primed variables from the pure part of a symbolic heap. ■

Definition 6.2 (Spatial Abstraction Rules)**.** The following are the rules for the abstraction of the spatial part of a symbolic state.

$$\frac{x' \notin Vars'(\Sigma, \Pi) \qquad \varsigma \in \mathsf{Sym}}{(\Sigma * P(x', \varsigma), \Pi) \overset{\mathcal{A}}{\rightsquigarrow} (\Sigma \cup Junk, \Pi)} \quad (Garbage1)$$

$$\frac{x', y' \notin Vars'(\Sigma, \Pi)}{(\Sigma * P_1(x', y') * P_2(y', x'), \Pi) \overset{\mathcal{A}}{\rightsquigarrow} (\Sigma \cup Junk, \Pi)} \quad (Garbage2)$$

$$\frac{y' \notin Vars'(\Sigma, \Pi) \cup \{\varsigma_1, \varsigma_2\} \qquad \varsigma_1, \varsigma_2 \in \mathsf{Sym} \qquad \varsigma_2 =_\Pi nil}{(\Sigma * P_1(\varsigma_1, y') * P_2(y', \varsigma_2), \Pi) \overset{\mathcal{A}}{\rightsquigarrow} (\Sigma * Ls(\varsigma_1, nil), \Pi)} \quad (Abs1)$$

$$\frac{y' \notin Vars'(\Sigma, \Pi) \cup \{\varsigma_1, \varsigma_2, \varsigma_3, \varsigma_4\} \quad \varsigma_1, \varsigma_2, \varsigma_3, \varsigma_4 \in \mathsf{Sym} \quad \varsigma_2 =_\Pi \varsigma_3}{(\Sigma * P_1(\varsigma_1, y') * P_2(y', \varsigma_2) * P_3(\varsigma_3, \varsigma_4), \Pi) \overset{\mathcal{A}}{\rightsquigarrow} (\Sigma * Ls(\varsigma_1, \varsigma_2) * P_3(\varsigma_3, \varsigma_4), \Pi)} \quad (Abs2)$$

Where,

$$\Sigma \cup Junk = \begin{cases} \Sigma & \text{if } \Sigma \text{ has a } Junk \text{ chunk} \\ \Sigma * Junk & \text{otherwise} \end{cases}$$

and $Vars'(\Sigma, \Pi)$ is the set of all primed variables that appear in Σ or Π. ■

The rule $Garbage1$ simply collects all memory chunks that are not pointed to by any program variables directly or indirectly – as x' does not appear in Σ or Π. The rule $Garbage2$ handles the cases where there is a cycle that is not reachable directly or indirectly from program variables (cyclic memory leak).

The rule $Abs1$ and $Abs2$ both merge linked list segments (or direct links) that form a continuous linked list. The fact that we require the end of the contracting linked list to be nil or already allocated in a separate portion of the heap is to make sure that the segments being merged don't form a cyclic linked list. Indeed if the end of the linked list segments is allocated in another separate portion of the memory or is nil, it can't be pointing to the middle or beginning of that linked list segment and it is thus safe to merge them to a single *acyclic* linked list segment.

Before we formally define the abstraction operation, we show that the set of rules for abstraction of the spatial part are strongly normalizing, i.e., there are no infinite chains of reduction.

Lemma 6.3 (Strong Normalization of Spatial Abstraction)**.** Let (Σ_0, Π_0) be a symbolic state, then, there does not exist an infinite sequence $(\Sigma_0, \Pi_0), (\Sigma_1, \Pi_1), \ldots$ such that $\forall i \in \mathbb{N}.\ (\Sigma_i, \Pi_i) \overset{\mathcal{A}}{\rightsquigarrow} (\Sigma_{i+1}, \Pi_{i+1})$. ■

Proof. It suffices to note that any of these rules decreases the collective number of Ls and $\mapsto$ chunks in the spatial part of the given symbolic states. □

In the sequel, we use $\overset{\mathcal{A}}{\rightsquigarrow}{}^*$ to denote the reflexive transitive closure of $\overset{\mathcal{A}}{\rightsquigarrow}$ and we use $(\Sigma, \Pi) \overset{\mathcal{A}}{\not\rightsquigarrow}$ to indicate that (Σ, Π) is a normal form, i.e., none of the rules of spatial abstraction are applicable to (Σ, Π).

Before, defining the abstraction operation, we define the set of abstract states. That is, the set of all consistent symbolic states that have no primed variables in their pure part and none of the abstraction rules are applicable to them, together with the symbolic error state, $\top_\sigma$.

Definition 6.4 (Abstract States)**.** The set of abstract states $\mathfrak{S}_\alpha$ is defined as follows:

$$\mathfrak{S}_\alpha = \{(\Sigma, \Pi) \in \mathfrak{S}_\sigma \mid (\Sigma, \Pi) \not\models \mathit{false} \wedge (rp_\Pi(\Sigma, \Pi)) = (\Sigma, \Pi) \wedge (\Sigma, \Pi) \overset{\mathcal{A}}{\not\rightsquigarrow}\} \cup \{\top_\sigma\}$$

■

The rewriting rules of Definition 6.2 form a rewriting relation, $\overset{\mathcal{A}}{\rightsquigarrow}$. Since this rewrite system is terminating, we can extract a function out of this relation that maps each symbolic state to one of its normal forms. This can simply be done by fixing an order on applications of the rules. In our implementation, we simply go through abstraction rules from top to bottom and for each of them keep reapplying them until they are no longer applicable. This process is repeated until none of the rules are applicable. In the sequel, we assume $\mathfrak{red}_\mathcal{A} : \mathfrak{S}_\sigma \to \mathfrak{S}_\sigma$ is such a function.

Definition 6.5 (Abstraction)**.** The abstraction function, denoted by $\mathfrak{Abs} : 2^{\mathfrak{S}_\sigma} \to 2^{\mathfrak{S}_\alpha}$ is defined as follows:

$$\mathfrak{Abs}(S) = \{(\Sigma', \Pi') \mid (\Sigma, \Pi) \in S \wedge \mathfrak{red}_\mathcal{A}(rp_\Pi(\Sigma, \Pi)) = (\Sigma', \Pi') \wedge (\Sigma', \Pi') \not\models \mathit{false}\} \cup \{\top_\sigma \mid \top_\sigma \in S\}$$

■

Abstract semantics is simply symbolic semantics where abstraction is applied to its result.

Definition 6.6 (Abstract Semantics)**.** Let c be a basic statement, then the abstract semantics of c denoted by $[\![c]\!]_\alpha : 2^{\mathfrak{S}_\alpha} \to 2^{\mathfrak{S}_\alpha}$ is defined as follows:

$$[\![c]\!]_\alpha(S) = \mathfrak{Abs}([\![c]\!]_\sigma S)$$

■

Figure 8 shows the abstract computation of the least fixpoint of the semantics of the while loop of the dispose program, depicted in Figure 2, starting in a set of states consisting of a single abstract state where x points to a linked list ending in nil.

Figure 9 shows the abstract computation of the least fixpoint of the semantics of the while loop for the infinite allocation program, depicted in Figure 3, starting in a set of states consisting of a single abstract state where the heap is empty and there are no pure assertions. Contrary to the case of symbolic semantics and concrete semantics, the least fixpoint of the while loop in this program, in abstract semantics, is obtainable by iterative computation of $f^n_{S_0}(\varnothing)$.

$$S_0 = \{(Ls(x, nil), \varnothing)\} \quad [\![dispose]\!]_\alpha(S_0) = \{(emp, \{y, nil, x\})\}$$

n	$f^n_{S_0}(\varnothing)$
0	$\varnothing$
1	$\{(Ls(x, nil), \varnothing)\}$
2	$\{(Ls(x, nil), \varnothing), (emp, \{y, nil, x\}), (Ls(y, nil), \{y, x\})\}$

Figure 8: Computation of the least fixpoint of the while loop semantics for the dispose program (Figure 2) in the abstract semantics. In this case, $f^2_{S_0}$ is the least fixpoint. In addition, $[\![dispose]\!]_\alpha$ is the abstract semantics of the whole dispose program (while loop).

$$S_0 = \{(emp, \varnothing)\} \quad [\![infalloc]\!]_\alpha(S_0) = \varnothing$$

n	$f^n_{S_0}(\varnothing)$
0	$\varnothing$
1	$\{(emp, \varnothing)\}$
2	$\{(emp * x \mapsto x'_1, \varnothing)\}$
3	$\{(emp * x \mapsto x'_3 * Junk, \varnothing)\}$

Figure 9: Computation of the least fixpoint of the while loop semantics for the infinite allocation program (Figure 3) in the abstract semantics. In this case, $f^3_{S_0}$ is the least fixpoint. In addition, $[\![infalloc]\!]_\alpha$ is the abstract semantics of the whole dispose program (while loop).

7 Overapproximation of Concrete Semantics by Abstract Semantics

To show that abstract semantics is a sound overapproximation of concrete semantics, we first show that abstraction is an overapproximation.

Lemma 7.1 (Overapproximation of Removal of Primed Variables)**.** Let (Σ, Π) be a symbolic state. Then,

$$\forall (h, s) \in \mathfrak{S}_\sigma.\ (h, s) \vDash (\Sigma, \Pi) \to (h, s) \vDash pr_\Pi(\Sigma, \Pi)$$

■

Proof. Assume that $pr_\pi(\Sigma, \Pi) = (\Sigma', \Pi')$ for some $\pi \in \Pi$. We show that if $(h, s) \vDash (\Sigma, \Pi)$, then $(h, s) \vDash (\Sigma', \Pi')$. Assume that $\{x'_1, \ldots, x'_n\} \subseteq \pi$ is the set of all primed variables in π and $rep(\pi)$ is as in Definition 6.1. In addition, let $ren : \mathsf{Sym} \to \mathsf{Sym}$ be as follows:

$$ren(\varsigma) = \begin{cases} rep(\pi) & \text{if } \varsigma = x'_i \text{ for some } 1 \le i \le n \\ \varsigma & \text{otherwise} \end{cases}$$

We show that, for any $s' : Vars' \to Values$,

$$(h, s) \overset{s'}{\vDash} (\Sigma, \Pi) \to (h, s) \overset{s'}{\vDash} pr_\pi(\Sigma, \Pi)$$

First, note that for all $\varsigma_1, \varsigma_2 \in \mathsf{Sym}$,

$$\varsigma_1 =_{\Pi'} \varsigma_2 \to \varsigma_1 =_\Pi \varsigma_2 \to val_\sigma(s, s', \varsigma_1) = val_\sigma(s, s', \varsigma_2)$$

Second, for any chunk A of Σ, $A' = A[\underbrace{rep(\pi), \ldots, rep(\pi)}_{n \text{ times}}/x'_1, \ldots, x'_n]$ and subheap h' of h, we have,

$$(h, s) \overset{s'}{\models} (A, \Pi) \rightarrow (h, s) \overset{s'}{\models} (A', \Pi')$$

There are four possibilities for A; $Junk$, emp, $\varsigma_1 \mapsto \varsigma_2$ and $Ls(\varsigma_1, \varsigma_2)$. In the first two cases, $A' = A$. According to Definition 5.1, the the case for Ls, follows form the case for $\mapsto$. Therefore, we consider the case where $A = \varsigma_1 \mapsto \varsigma_2$. Due to Definition 5.1, $(h, s) \overset{s'}{\models} (\varsigma_1 \mapsto \varsigma_2, \Pi)$, then,

$$h(a) = \begin{cases} val_\sigma(s, s', \varsigma_2) & \text{if } a = val_\sigma(s, s', \varsigma_1) \\ \bot & \text{otherwise} \end{cases}$$

On the one hand, $A' = ren(\varsigma_1) \mapsto ren(\varsigma_2)$. Thus, since $\forall \varsigma \in \mathsf{Sym}.\ \varsigma =_\pi ren(\varsigma)$, we have

$$\forall \varsigma \in \mathsf{Sym}.\ val_\sigma(s, s', \varsigma) = val_\sigma(s, s', ren(\varsigma))$$

Therefore, $(h, s) \overset{s'}{\models} (A', \Pi')$. □

Lemma 7.2 (Overapproximation of Spatial Abstraction)**.** Let (Σ, Π) and (Σ', Π') be two symbolic states such that $(\Sigma, \Pi) \overset{\mathcal{A}}{\rightsquigarrow} (\Sigma', \Pi')$, then,

$$\forall (h, s) \in \mathfrak{S}_\kappa.\ (h, s) \models (\Sigma, \Pi) \rightarrow (h, s) \models (\Sigma', \Pi')$$

■

Proof. We consider all the four cases. In what follows, we assume $(h, s) \overset{s'}{\models} (\Sigma, \Pi)$.

*Garbage*1. This can be if $\Sigma = \Sigma_1 * P(x', \varsigma)$ for x' not appearing in Σ_1. In this case, there must be two concrete heaps h' and h'' such that $h = h' \uplus h''$ and $(h', s) \overset{s'}{\models} (\Sigma_1, \Pi)$ and $(h'', s) \overset{s'}{\models} (P(x', \varsigma), \Pi)$ which means $dom(h'') \neq \varnothing$.

Here, there are two cases to consider $\Sigma_1 = \Sigma_2 * Junk$ or Σ_1 has no $Junk$ chunk. let $h' = h'_1 \uplus h'_2$ such that in the first case $(h'_1, s) \models (\Sigma_2, \Pi)$ and $dom(h'_2) \neq \varnothing$ and in the second case, $h'_1 = h'$ and $dom(h'_2) = \varnothing$.

In both cases, $dom(h'_2 \uplus h'') \neq \varnothing$ and thus, we have $(h, s) \models (\Sigma_1 \cup Junk, \Pi)$. Note that since $h = (h'_1 \uplus h'_2) \uplus h''$, $dom(h'_2) \cap dom(h'') = \varnothing$ and thus $h'_2 \uplus h''$ is not undefined.

*Garbage*2. Proof is very similar to the previous case and is thus omitted.

*Abs*1. This can be if $\Sigma = \Sigma_1 * P_1(\varsigma_1, y') * P_2(y', \varsigma_2)$ for $\varsigma_1, \varsigma_2 \in \mathsf{Sym}$ and y' neither appear in Σ_1 nor in Π nor is it equal to ς_1 or ς_2 and we have $\varsigma_2 =_\Pi nil$. We prove that if $(h, s) \models (\Sigma, \Pi)$ then $(h, s) \models (\Sigma_1 * Ls(\varsigma_1, nil), \Pi)$.

Since ς_1 is allocated in (Σ, Π) and $val_\sigma(s, s', \varsigma_2) = nil$, $val_\sigma(s, s', \varsigma_2) \neq val_\sigma(s, s', \varsigma_1)$. On the other hand, $h = h_1 \uplus h_2 \uplus h_3$ such that $(h_1, s) \overset{s'}{\models} (\Sigma_1, \Pi)$, $(h_2, s) \overset{s'}{\models} (P_1(\varsigma_1, y'), \Pi)$ and $(h_3, s) \overset{s'}{\models} (P_2(y', \varsigma_2), \Pi)$. Therefore, $(h_2 \uplus h_3, s) \overset{s'}{\models} (Ls(\varsigma_1, \varsigma_2), \Pi)$ and hence $(h, s) \overset{s'}{\models} (\Sigma_1 * Ls(\varsigma_1, nil), \Pi)$.

*Abs*2. This case follows from a reasoning very similar to the previous case. The only thing to note is that the reason for having $val_\sigma(s, s', \varsigma_2) \neq val_\sigma(s, s', \varsigma_1)$ is different. Namely, it is due to the fact that $\varsigma_2 =_\Pi \varsigma_3$ and therefore $val_\sigma(s, s', \varsigma_2) = val_\sigma(s, s', \varsigma_3)$ and since ς_1 and ς_3 are allocated in disjoint parts of the heap, we know they can not be corresponding to the same address and thus $val_\sigma(s, s', \varsigma_2) \neq val_\sigma(s, s', \varsigma_1)$.

□

Theorem 7.3 (Abstraction Overapproximates)**.** Let $S \subseteq \mathfrak{S}_\sigma$ be a set of symbolic states, then:

$$\gamma(S) \subseteq \gamma(\mathfrak{Abs}(S))$$

■

Proof. Since $\mathfrak{Abs}$ is a continuous function, we need only to show:

$$\forall st \in \mathfrak{S}.\ \gamma(\{st\}) \subseteq \gamma(\mathfrak{Abs}(\{st\}))$$

If $st = \top_\sigma$, $\gamma(\mathfrak{Abs}(\{st\})) = \mathfrak{S}_\kappa$. Otherwise, if $st = (\Sigma, \Pi)$, let's assume

$$\mathfrak{red}_{\mathcal{A}}(rp_\Pi(\Sigma, \Pi)) = (\Sigma', \Pi')$$

In this case, according to Lemma 7.1 and Lemma 7.2,

$$\gamma(\{(\Sigma, \Pi)\}) \subseteq \gamma(\{(\Sigma', \Pi')\})$$

Therefore, according to Theorem 5.3, $(\Sigma', \Pi') \vdash \mathit{false}$, only if $(\Sigma, \Pi) \vdash \mathit{false}$. Hence,

$$\gamma(\{st\}) \subseteq \gamma(\mathfrak{Abs}(\{st\}))$$

□

Finally, the main result of this section, the overapproximation of concrete semantics by symbolic semantics, is given below.

Theorem 7.4 (Abstract Semantics Overapproximates Concrete Semantics)**.** Let c be any statement or block of statements. Then,

$$\forall S \subseteq \mathfrak{S}_\alpha.\ [\![c]\!]_\kappa(\gamma(S)) \subseteq \gamma([\![c]\!]_\alpha(S))$$

■

Proof. This follows using a similar reasoning as in the proof of Theorem 5.9. The only difference being the proof of the base case, i.e., the case of basic statements, follows from Lemma 5.7 together with Theorem 7.3. □

As an important corollary, we get that whenever the concrete execution of the program, starting in the empty memory, indicates an error in the program, so does the abstract execution, starting in the empty memory.

Corollary 7.5. For a statement (program) c,

$$\text{if } \top_\kappa \in [\![c]\!]_\kappa(h_\varepsilon, s_\varepsilon) \text{ then } \top_\sigma \in [\![c]\!]_\alpha(emp, \varnothing)$$

where,

$$h_\varepsilon(a) = \bot \text{ and } s_\varepsilon(x) = nil$$

■

8 Termination of Analysis with Abstract Semantics

In order to show termination of fixpoint computation – as fixpoints are used for the computation of semantics of while loops –, we show that the set of abstract states is finite. We define a notion of reduced-ness and show that a symbolic state is reduced if and only if it is not changed under abstraction. Then, we show that the set of reduced symbolic states is finite.

In order to formalize the concept of reduced-ness, we first define paths in symbolic states and define some properties of symbols in symbolic states.

Definition 8.1 (Paths, Cycles, Length and Reachability)**.** Let (Σ, Π) be a symbolic state, then, a path in (Σ, Π) is a sequence $\varsigma_0, \varsigma_1, \ldots, \varsigma_{n-1}, \varsigma_n$ such that

$$\forall 0 \leq i \leq n.\ \varsigma_i \in \mathsf{Sym}$$

$$\forall 1 \leq i \leq n.\ \exists \varsigma', \varsigma'' \in \mathsf{Sym}.\ \varsigma_{i-1} =_\Pi \varsigma' \wedge \varsigma_i =_\Pi \varsigma'' \wedge \Sigma\varsigma', \varsigma'')$$

A cycle is a path $\varsigma_0, \ldots, \varsigma_n$ if $\varsigma_0 =_\Pi \varsigma_n$.

In addition, the length of a path (or cycle) $\varsigma_0, \ldots, \varsigma_n$, n here, is the syntactical length of a path (or cycle), i.e., we don't distinguish $\mapsto$ and Ls.

For $\varsigma_0, \varsigma_n \in \mathsf{Sym}$, we say ς_n is reachable from ς_0 in (Σ, Π), if there is a path $\varsigma_0, \varsigma_1, \ldots, \varsigma_{n-1}, \varsigma_n$ in (Σ, Π). ■

Definition 8.2 (Shared, Internal, Possibly Dangling and Pointing to Possibly Dangling Symbols)**.** Let (Σ, Π) be a symbolic state, then,

> ***Shared Symbols:*** A symbol ς is a shared symbol, if $\Sigma = \Sigma' * P_1(\varsigma_1, \varsigma_2) * P_2(\varsigma_3, \varsigma_4)$ such that $\varsigma =_\Pi \varsigma_2 =_\Pi \varsigma_4$.
>
> ***Internal Symbols:*** A symbol ς is an internal node of a cycle in (Σ, Π) if and only if it is not shared.
>
> ***Possibly Dangling Symbols:*** A symbol ς is a possibly dangling symbol if $\varsigma \neq_\Pi nil$ and
>
> $$\exists \varsigma_1, \varsigma_2 \in \mathsf{Sym}.\ \Sigma = \Sigma' * P(\varsigma_1, \varsigma_2) \wedge \varsigma_2 =_\Pi \varsigma$$
>
> and
>
> $$\nexists\ \varsigma_3, \varsigma_4 \in \mathsf{Sym}.\ \Sigma' = \Sigma'' * P'(\varsigma_3, \varsigma_4) \wedge \varsigma_3 =_\Pi \varsigma$$
>
> ***Pointing to Possibly Dangling Symbols:*** A symbol ς points to a possibly dangling symbol if $\Sigma = \Sigma' * P(\varsigma_1, \varsigma_2)$ for some $\varsigma_1, \varsigma_2 \in \mathsf{Sym}$ such that $\varsigma =_\Pi \varsigma_1$ and ς_2 is a possibly dangling symbol.
>
> ■

Definition 8.3 (Reduced Symbolic States)**.** Let (Σ, Π) be a symbolic state, then, (Σ, Π) is reduced if and only if

1. There are no primed variables appearing in Π.
2. Every primed variable appearing in Σ is reachable from some program variable.
3. If a primed variable x' appears in Σ, then, at least one of the followings hold
   (a) x' is shared

(b) x' is the internal node of a cycle of length exactly 2

(c) x' is pointing to a possibly dangling variable

(d) x' is possibly dangling

■

Lemma 8.4. Let (Σ, Π) be a symbolic state such that $(\Sigma, \Pi) \not\vdash \mathit{false}$. Then, (Σ, Π) is reduced if and only if $(\Sigma, \Pi) \overset{\mathcal{A}}{\not\rightarrow}$. ■

Proof. We prove 'if' and 'only if' parts as follows:

$\Rightarrow$. If (Σ, Π) is a reduced, we show that $(\Sigma, \Pi) \overset{\mathcal{A}}{\not\rightarrow}$. As (Σ, Π) is reduced, there are no primed variables in Π and thus, $rp_\Pi(\Sigma, \Pi) = (\Sigma, \Pi)$. Furthermore, we show that if any of the spatial abstraction rules is applicable in (Σ, Π), (Σ, Π) is not reduced which contradicts our assumption.

$\mathit{Garbage}1$: If $\mathit{Garbage}1$ is applicable in (Σ, Π), we should have $\Sigma = \Sigma' * P(x', \varsigma)$ for some $\varsigma \in \mathsf{Sym}$ such that $x' \notin \mathit{Vars}'(\Sigma', \Pi)$. In such a case, x' is a primed variable not reachable from any program variable which contradicts condition 2 of Definition 8.3.

$\mathit{Garbage}2$: If $\mathit{Garbage}2$ is applicable in (Σ, Π), we should have $\Sigma = \Sigma' * P_1(x', y') * P_2(y', x')$ for some $x', y' \in \mathit{Vars}'$ such that $x', y' \notin \mathit{Vars}'(\Sigma', \Pi)$. In such a case, x' and y' are primed variables not reachable from any program variable which contradicts condition 2 of Definition 8.3.

$\mathit{Abs}1$: If $\mathit{Abs}1$ applies in (Σ, Π), we should have $\Sigma = \Sigma' * P_1(\varsigma_1, y') * P_2(y', \varsigma_2)$ for $\varsigma_1, y', \varsigma_2 \in \mathsf{Sym}$ such that $y' \notin \mathit{Vars}'(\Sigma', \Pi) \cup \{\varsigma_1, \varsigma_2\}$ and $\varsigma_2 =_\Pi \mathit{nil}$.
Since $y' \notin \mathit{Vars}'(\Sigma, \Pi) \cup \{\varsigma_1, \varsigma_2\}$, y' is not shared. In addition, y' can not be on a cycle since there should be path from ς_2 to ς_1 which is impossible as $\varsigma_2 =_\Pi \mathit{nil}$ and thus we would have $(\Sigma, \Pi) \vdash \mathit{false}$ which contradicts our assumption.
On the other hand, since $\varsigma_2 =_\Pi \mathit{nil}$, ς_2 can not be a dangling symbol and thus y' can not be pointing to a dangling symbol. Moreover, y' can not itself be possibly dangling as it is appearing on the left hand side of $P_2(y', \varsigma_2)$.

$\mathit{Abs}2$: If $\mathit{Abs}2$ applies in (Σ, Π), we should have $\Sigma = \Sigma' * P_1(\varsigma_1, y') * P_2(y', \varsigma_2) * P_3(\varsigma_3, \varsigma_4)$ for $\varsigma_1, y', \varsigma_2, \varsigma_3, \varsigma_4 \in \mathsf{Sym}$ such that $y' \notin \mathit{Vars}'(\Sigma', \Pi) \cup \{\varsigma_1, \varsigma_2, \varsigma_3, \varsigma_4\}$ and $\varsigma_2 =_\Pi \varsigma_3$.
Since $y' \notin \mathit{Vars}'(\Sigma, \Pi) \cup \{\varsigma_1, \varsigma_2, \varsigma_3, \varsigma_4\}$ holds, y' is not shared. Moreover, since $\varsigma_2 =_\Pi \varsigma_3$, ς_2 can not be a dangling symbol and thus y' can not be pointing to a dangling symbol. On the other hand, y' can not itself be possibly dangling as it is appearing on the left hand side of $P_2(y', \varsigma_2)$.
In addition, y' can not be on a cycle of length exactly two. There are three ways that can result in y' to be part of a cycle of length two and we show that all three of these cases are impossible.

1. We have $\varsigma_2 =_\Pi \varsigma_1$ which is impossible given $(\Sigma, \Pi) \not\vdash \mathit{false}$ as $\varsigma_2 =_\Pi \varsigma_3$ and ς_3 is allocated in a portion of the heap that is disjoined from the portion where ς_1 is allocated.
2. We have $\Sigma' = \Sigma'' * P_0(\varsigma_5, \varsigma_6)$ for $\varsigma_6 =_\Pi y'$ and $\varsigma_5 =_\Pi \varsigma_2$. This is impossible as $\varsigma_5 =_\Pi \varsigma_5$ would result in $(\Sigma, \Pi) \vdash \mathit{false}$ which contradicts our assumption.
3. We have $y' =_\Pi \varsigma_4$. This is impossible as $\varsigma_4 =_\Pi y'$ would require y' appearing in Π which contradicts condition 1 of Definition 8.3 or have $\varsigma_4 = y'$ which contradicts $y' \notin \mathit{Vars}'(\Sigma', \Pi) \cup \{\varsigma_1, \varsigma_2, \varsigma_3, \varsigma_4\}$.

$\Leftarrow$. We assume that $(\Sigma, \Pi) \overset{\mathcal{A}}{\not\to}$ and show that (Σ, Π) is reduced. First, we observe that since no spatial abstraction rule changes Π, we have $rp_\Pi(\Sigma, \Pi) = (\Sigma, \Pi)$ which means there are no primed variables in Π. This justifies the first condition of Definition 8.3.

Let x' be a primed variable in Σ not reachable from any program variable. We show that at least one of the abstraction rules will apply which contradicts our assumption. We consider the following cases. In the following, we use left occurrence and right occurrence for referring to a symbol appearing in the first or second argument of a P chunk, respectively.

- The primed variable x' can not be only appearing in the left hand side of a P chunk. If there is only one such chunk, $Garbage1$ would be applicable. If there are more than one such chunks, (Σ, Π) would be inconsistent. Therefore, there always exists a chunk $P(\varsigma, x')$ in Σ for some $\varsigma \in \mathsf{Sym}$.
- We have x' is not part of any cycles. Let's assume that $\rho = x'_1, \ldots, x'_n, x'$ is a maximal path ending in x', i.e., there does not exist y' such that there is a path from y' to x'_1. Then, rule $Garbage1$ would be applicable as x'_1 can not be appearing in any chunk other that the one being part of ρ, otherwise, either consistency of (Σ, Π) or maximality of ρ is violated.
- We have x' is part of a cycle ρ. First note that ρ can not have any program variables or nil as it would contradict x' not being reachable by any program variables or consistency of (Σ, Π).

  Second, let us note that no symbol can be part of two different cycles in a consistent symbolic state. Therefore, if any of the primed variables y' of ρ is shared, it can only be pointed to by some primed variable (as otherwise, it would violate x' not being reachable by any program variables or consistency of (Σ, Π)). In such a case, a reasoning like that of the previous case can show that $Garbage1$ would be applicable to the beginning of such a path ending in y'.

  In addition, note that the length of the cycle must be at least 2. If the cycle is of length 1, i.e., we have $P(x', x')$, if x' is shared, we know that the beginning of the path pointing to it has $Garbage1$ applicable to it. If x' is not shared, since (Σ, Π) is consistent, rule $Garbage1$ would be applicable to $P(x', x')$.

  Hence, we assume that all primed variables of ρ are not shared and the length of the cycle is at least 2. As a result, no primed variable y' in ρ can be appearing in any chunk except for the chunks $P_1(x'_1, y')$ and $P_2(y', x'_2)$ such that x'_1, y', x'_2 is part of cycle ρ. Consequently, rule $Abs2$ would be applicable to $P_1(x'_1, y') * P_2(y', x'_2)$ as x'_2 is part of the cycle ρ and is hence equal to some allocated symbol under Π.

Now we consider the cases where $\Sigma = \Sigma' * P(x', \varsigma)$ for some $\varsigma \in \mathsf{Sym}$ or there is no such chunk $P(x', y)$ in Σ. We consider these two cases respectively as follows:

Case 1: $\Sigma = \Sigma' * P(x', \varsigma)$. If x' is shared, the condition 3.a of Definition 8.3 is satisfied. So in the rest of this case, we assume that x' is not shared.

If x' is part of a cycle, as mentioned earlier, the length of such a cycle must be at least 2. If x' is on a cycle of length 2, then the the condition 3.b of Definition 8.3 holds. If x' is on a cycle of length greater than 2, then $Abs2$ would be applicable which contradicts our assumption. Thus, in the rest of this case, we assume that x' is not part of any cycles.

Since x' is not part of any cycles, and x' is reachable from some program variable, we know that there must be a path $x, \varsigma_1, \ldots, \varsigma_n, x', \varsigma$, for some program variable x.

If the length of this path is greater than 2, ($n \geq 1$), as x' is assumed not be shared, the rule $Abs2$ would be applicable which is contradictory. Therefore, we assume that $\Sigma = \Sigma' * P_1(x, x') * P_2(x', \varsigma)$. In such a case, if $\varsigma =_\Pi nil$, $Abs1$ would be applicable which is contradictory. If $\varsigma =_\Pi \varsigma_2$ for some symbol ς_2 and ς_2 occurs on the left hand side of a P chunk, $Abs2$ would be applicable which is again contradictory. Thus, ς must be a dangling symbol, which implies that x' is pointing to a dangling symbol which is the condition 3.c of Definition 8.3.

Case 2: There is no Σ' such that $\Sigma = \Sigma' * P(x', \varsigma)$. In this case, x' only appears as the second argument of some P chunk. So let's assume $\Sigma = \Sigma' * P(\varsigma_1, x')$ for some $\varsigma_1 \in \mathsf{Sym}$. As x' does not appear at the left hand side of any P chunk, and there are no primed variables appearing in Π, nothing equal to it based on Π can be appearing as the left hand side of a P chunk. Consequently, x' must be a dangling symbol which is condition 3.d of Definition 8.3.

□

Now, in order to show that the set of consistent reduced symbolic states (which we just proved is the same as the set of abstract symbolic states) is finite, we show that if a consistent symbolic state is reduced, then, there is an upper bound on the number of primed variables that can be appearing in it.

Lemma 8.5 (Partitioning of Primed Variables)**.** Let (Σ, Π) be a consistent reduced symbolic state. Then, the set $\{X_s, (X_c \setminus X_s), (X_p \setminus X_s), (X_d \setminus X_s)\}$ where X_s, X_c, X_p and X_d are defined below, is a partitioning of primed variables appearing in Σ.

- $x' \in X_s$ if x' is shared
- $x' \in X_c$ if s' is the internal node of a cycle of length exactly 2
- $x' \in X_p$ if x' is pointing to a possibly dangling variable
- $x' \in X_d$ if x' is possibly dangling

■

Proof. Since (Σ, Π) is reduced, we obviously have $Vars'(\Sigma) = X_s \cup (X_c \setminus X_s) \cup (X_p \setminus X_s) \cup (X_d \setminus X_s)$. The only thing that remains to be shown is that these sets are pairwise disjoint.

Obviously,

$$X_s \cap (X_c \setminus X_s) = X_s \cap (X_p \setminus X_s) = X_s \cap (X_d \setminus X_s) = \varnothing$$

If any primed variable is part of a cycle, it can neither be dangling nor can it be pointing to a dangling symbol. Hence,

$$(X_c \setminus X_s) \cap (X_p \setminus X_s) = (X_c \setminus X_s) \cap (X_d \setminus X_s) = \varnothing$$

On the other hand, if a symbol is dangling, it can not be pointing to any (dangling) symbol. As a result,

$$(X_c \setminus X_s) \cap (X_d \setminus X_s) = \varnothing$$

□

Lemma 8.6 (Bound on Primed Variables of Consistent Reduced Symbolic States)**.** Let (Σ, Π) be a reduced symbolic state such that $(\Sigma, \Pi) \not\models \mathit{false}$, then the number of primed variables appearing in Σ is bound by $3n + 2$, where n is the number of program variables. ■

Proof. Let X_s, X_c, X_p and X_d be as defined in Lemma 8.5. Furthermore, let m_s, m_c, m_p and m_d be respectively the cardinalities of sets X_s, $(X_c \setminus X_s)$, $(X_p \setminus X_s)$ and $(X_d \setminus X_s)$.

Since, symbols appearing on the left hand sides of Ls and $\mapsto$ predicates must be unique (as $(\Sigma, \Pi) \not\models \mathit{false}$) and considering that members of m_d can not appear as the left hand side of any predicate, we have that

$$|\Sigma| \leq n + m_s + m_c + m_p + 1$$

where $|\Sigma|$ is the number of conjuncts of Σ. The 1 at the end is added for the sake of $Junk$ predicate. Furthermore, we do not consider emp chunks.

On the other hand, since all primed variables are reachable from some program variable, they must all be appearing on the right hand side of some Ls or $\mapsto$. Particularly members of m_s must appear at least twice (as they are shared). Hence,

$$2m_s + m_c + m_p + m_d \leq |\Sigma|$$

Altogether,

$$2m_s + m_c + m_p + m_d \leq n + m_s + m_c + m_p + 1$$

From which we can draw the conclusion that

$$m_s + m_d \leq n + 1 \tag{3}$$

For the upper bound of $m_c + m_p$, we can see that if $x' \in m_c \cup m_p$, there must be a chunk $P(\varsigma, x')$ in Σ for $\varsigma \in \mathsf{Sym}$. In such a case, ς can not be in m_c, m_p and m_d. If $\varsigma \in m_c$, then, ς is a primed variable and together with x' form a cycle of length 2 and are both not shared which means rule $Garbage2$ should be applicable which, according to Lemma 8.4, is a contradiction to the fact that (Σ, Π) is reduced. If $\varsigma \in m_p$, then ς should be pointing to a possibly dangling symbol but x' is already allocated (as it is part of a cycle or is pointing to a possibly dangling symbol) and can not be a possibly dangling symbol. If $\varsigma \in m_d$, then ς must be a possibly dangling symbol which means it can not appear on the left hand side of any Ls or $\mapsto$ predicate which contradicts our initial assumption that $P(\varsigma, x')$ is a chunk of Σ. Consequently, ς can only be a shared primed variable or a program variable. Hence,

$$m_c + m_p \leq n + m_s$$

From Equation 3, we get that $m_s \leq n + 1$ which means:

$$m_c + m_p \leq 2n + 1 \tag{4}$$

From Equation 3 and Equation 4, we have that

$$m_s + m_c + m_p + m_d \leq 3n + 2 \tag{5}$$

Which means that a consistent reduced symbolic state can have at most $3n + 2$ primed variables appearing in it. □

Theorem 8.7 (Finiteness of Abstract States)**.** Let n be the number of program variables, then, the number of reduced states is bounded by $2^{(2^{n+1})(16n^2+20n+7)}$.

Proof. Since there are no primed variables in the pure part of a reduced symbolic state, there can be at most $n + 1$ elements in each equivalence class. This means that we can get a very coarse bound of $2^{2^{n+1}}$ equivalence relations (pure parts).

On the other hand, from Lemma 8.6, we have that there can be at most $4n + 2$ symbols (primed and program variables) appearing on the left hand side of a Ls or $\mapsto$ predicate and at most $4n + 3$ symbols (primed and program variables together with nil) appearing on the right hand side. Therefore, there can be at most $16n^2 + 20n + 6 + 1$ chunks (one for $Junk$ predicate) that can possibly appear in spatial part. Therefore, there can be at most $2^{16n^2+20n+7}$ different spatial parts.

This means that in total there can be at most $2^{(2^{n+1})(16n^2+20n+7)}$ different reduced symbolic states. Hence, $\mathfrak{S}_\alpha$ is finite. □

Acknowledgements

This work was partially funded by EU FP7 FET-Open project ADVENT under grant number 308830.

References

[BCO05] Josh Berdine, C Calcagno, and P W O'Hearn. Symbolic Execution with Separation Logic. *Programming Languages and . . .* , 3780(Chapter 5):52–68, 2005.

[CR08] Bor-Yuh Evan Chang and Xavier Rival. *Relational inductive shape analysis*, volume 43. ACM, New York, New York, USA, January 2008.

[CRN07] Bor-Yuh Evan Chang, Xavier Rival, and George C Necula. Shape Analysis with Structural Invariant Checkers. *Static Analysis*, 4634(Chapter 24):384–401, 2007.

[DOY06] Dino Distefano, Peter W O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. pages 287–302, 2006.

[LRS04] Alexey Loginov, Thomas Reps, and Mooly Sagiv. Abstraction refinement for 3-valued-logic analysis. *Submitted for publication*, 2004.

[OCDY06] P W O'Hearn, C Calcagno, Dino Distefano, and Hongseok Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. pages 182–203, 2006.

[Rey] J C Reynolds. Separation logic: a logic for shared mutable data structures. In *17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE Comput. Soc.