Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science
Master's Program in Computer Science

Master's thesis

Permutation semantics of separation logic

submitted by
Murat Baktiev

on December 1, 2006

Supervisor
Prof. Dr. Gert Smolka

Advisor
Dr. Jan Schwinghammer

Reviewers
Prof. Dr. Gert Smolka
Prof. Dr. Holger Hermanns

Statement

Hereby I confirm that thesis is my own work and that I have documented all sources used

Saarbruecken, December 01, 2006

_____

Declaration of Consent
Hereby I agree that my thesis will be made available through the library of the Computer Science Department.

Saarbruecken, December 01, 2006

_____

Abstract

Separation logic is a recent extension of Hoare logic, developed by O'Hearn, Reynolds and others, where the separation conjunction $p * q$ describes a disjoint partition of the heap. With this connective, a frame rule can be stated that facilitates local reasoning about programs using shared mutable data structures in the heap.

Originally, soundness of the logic was proved with respect to a non-deterministic memory allocation model. However, in some contexts, this "artificial" non-determinism interferes with other language features. I develop an alternative semantics of separation logic, based on deterministic memory allocation.

Contents

List of Tables

Acknowledgments

Thanks to: Gert Smolka, Jan Schwinghammer, Holger Hermanns, my family, Hristofor Boev.

Chapter 1

Introduction

1.1 Separation Logic

In the end of the 60-s Hoare developed a logic for reasoning about the correctness of the programs written in a simple imperative *while*-language[4]. Since then the logic has been applied by many researchers to various more complex languages, like: machine code[6], object-oriented, [2], [14], procedural [10] languages. Central point in this logic is a notion of *Hoare triple*, $\{p\}c\{q\}$, which describes the state change induced by a piece of a program - running a command c in a state, where assertion p is true, results in a state, where assertion q holds. Command c is of *while*-language and it changes (unless it is **skip**) the *state* of the program, represented by *Store*,which is a mapping of variables to values, and described by assertions.

Separation logic is an extension of Hoare logic for reasoning about programs that use shared mutable data structures, usually called pointer programs. It was developed by joint efforts of John Reynolds[12], Peter O'Hearn, Hongseok Yang[8] and others in the first years of this century based on early ideas of Burstall [3] and Pym and O'Hearn's logic of Bunched Implications, [7]. Program state in programming language with heap operations is represented now not only by *Store* but by pair *Store* x *Heap*, and new commands and assertions were added to reflect these changes. Most importantly - *separating conjunction*: $*$, which asserts that its subformulas hold for disjoint parts of the heap. This connective solves the problem with aliasing -when a memory cell is pointed from different parts of a program, and its alteration affects all of them- appearing for example in a rule of constancy, replacing it with *Frame rule*. This rule is vital for *local reasoning* - possibility to concentrate only on those parts of the memory which are actually accessed by the procedure, as O'Hearn/Yang named it-the *footprint* [17], being sure that the rest part of the memory will remain unmodified.

The relatively young logic has already been successfully used by many researchers in practice: Yang's proof of the Schorr-Waite algorithm[16], Torp-Smith's proof of Cheney algorithm[13], various extensions of this logic by different authors for concurrency, higher-order languages, e.g. see [9], [5].

1.2 Description of the Problem

Having all these advantages nevertheless the usual semantics of the separation logic is not always suitable for extensions of the logic and in some cases shows to be inconvenient. One particular problem which we will try to solve in this work is about non-deterministic behaviour of memory allocation. Besides the requirement for the cells to be previously inactive and consequtive, the choice of locations is indeterminate. This problem was mentioned by several researchers:

> *"Previous work on modularity, simulation and refinement in separation logic has run into some technical difficulties associated with the non-deterministic treatment of allocation..."*
>
> Nick Benton[1]

> *"...in the presence of higher-order store where we have to solve recursive domain equations we found the use of (countable) non-determinism quite challenging (for instance, programs would no longer denote ω-continuous functions)."*
>
> Bernhard Reus, Jan Schwinghammer[11]

1.3 Permutation semantics

Yang/O'Hearn in their work "Semantic basis for local reasoning" [17] pointed out the reason for relying on a non-deterministic semantics and the way to solve this problem:

> *"Non-deterministic allocation is used to force a program proof not to depend on details of how the allocator might work (**In a language without address arithmetic, we could use invariance under location renaming rather than non-determinism in allocation to ensure this sort of independence**)"*

and here we will implement these ideas and prove the logic to be still sound.

We will check for O'Hearn's instructions and first in a Chapter 2 we give a language without address arithmetic, then in Chapter 3 exploit renaming of locations, in Chapter 4 we provide the logic for our language and in the next chapter prove the soundness of this logic.

In order to restrict address arithmetic we step back to the earlier versions of separation logic, rather than use Reynolds' version, and divide *Values* into classes of *Integers, Atoms* and *Locations*.

We change non-deterministic memory allocation operator **cons** to deterministic operator **new**, which will allocate the minimal inactive cell. The other commands will remain unchanged.

Having such a language we define the operation of *permutation*, which is a bijective function from *Locations* to *Locations* and prove a **permutation theorem**:

Running a consistently renamed program on a correspondingly renamed heap leads to the same result, up to a renaming of locations in the final heap.

Permutation: $\pi : Loc \to Loc$

$$\langle c; \sigma \rangle \rightsquigarrow \sigma' \Rightarrow \exists \pi'. \langle c; \pi \bullet \sigma \rangle \rightsquigarrow \sigma'' \wedge \sigma' = \pi' \bullet \sigma''$$

Chapter 2

Language and Semantics

As it was said in the introduction, we will follow the direction pointed by O'Hearn, and first of all we need a language without address arithmetic. For that reason we will step back from Reynolds' version of the separation logic[12] and treat *Integers*, *Locations* and *Atoms* as distinct kinds of *Values*.

2.1 Language Syntax

Programming language we use is the version of simple imperative while language extended to reflect changes in the state model (heap commands).
There are three syntactic categories: **booleans**, **expressions** and **commands**: Table 2.1. Commands are usual commands of *while*-language: **skip** - no operation, $c_0; c_1$ - sequencing, conditional, while loop and assignment; and the last four commands allow operations on the heap.

$x, y \in Variables$	$[e]$ – *contents of the heap at the location of* e	
$b ::=$	$\mathbf{false} \mid e_0 \leq e_1 \mid \neg b \mid b_0 \vee b_1$	
$e ::=$	$x \mid n \mid e_0 + e_1 \mid e_0 - e_1 \mid e_0 \times e_1$	
$c ::=$	$\mathbf{skip} \mid c_0;\ c_1 \mid \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1 \mid \mathbf{while}\ b\ \mathbf{do}\ c \mid x := e$	
	$\mid x := \mathbf{new}\ e$	*allocation*
	$\mid x := [y]$	*lookup*
	$\mid [x] := e'$	*mutation*
	$\mid \mathbf{dispose}\ e$	*deallocation*

Table 2.1: Language syntax

2.2 Domains

$Values = Integers \biguplus Atoms \biguplus Locations \biguplus Booleans$
Set of locations *Loc* isomorphic to *Naturals*

We make *Loc* isomorphic to *Naturals* because we will need an ordered set of locations with minimal element, to get our deterministic allocation.

Variables, countably infinite ranged over $x, y...$

Store is a function from variables to values. *Heap* is a function from finite set of locations to values. State is a pair of *Store* and *Heap*.
$Store = Variables \to Values$
$Heap = \bigcup_{A \subseteq \text{Locations}}^{fin} (A \to \text{Values})$

$State = Store \times Heap$.

2.3 Language Semantics

Denotional semantics of the booleans and expressions is the same as in Hoare logic, but with arithmetic operations on non-integers (which are addresses) yielding **undefined**, to limit address arithmetic: Table 2.2, Table 2.3.
$[\![e]\!]_s$ is the evaluation of expression e in a state with store s. Expressions depend only upon the *Store*.

$[\![b]\!] : Store \to Booleans \cup \{\textbf{undefined}\}$
$[\![\textbf{false}]\!]_s = false$
$[\![b_0 \vee b_1]\!]_s = [\![b_0]\!]_s \vee [\![b_1]\!]_s$
$[\![\neg b]\!]_s = \neg [\![b]\!]_s$
$$[\![e_0 \leq e_1]\!]_s = \begin{cases} [\![e_0]\!]_s \leq [\![e_1]\!]_s, & \text{if both parts yield integer} \\ \textbf{undefined}, & \text{otherwise} \end{cases}$$

Table 2.2: Language semantics: Booleans

Operational semantics of the original commands of the *while*-language usually considered by Hoare logic is slightly changed, since *State* is extended to contain *Heap*: Table 2.4. The triple $\langle c, s, h \rangle$, or pair $\langle s, h \rangle$ - where c is a command, s - store, h - heap, - is called *configuration*. There is a transition relation - $\rightsquigarrow$ on configurations. Configurations can be:

$$[\![e]\!] : Stores \to Values \cup \{\textbf{undefined}\}$$
$$[\![x]\!]_s = s(x)$$
$$[\![n]\!]_s = n$$
$$[\![e_0 + e_1]\!]_s = \begin{cases} [\![e_0]\!]_s + [\![e_1]\!]_s, & \text{if both parts yield integer} \\ \textbf{undefined}, & \text{otherwise} \end{cases} \quad \text{,same for } \times, -$$

Table 2.3: Language semantics: Expressions

1. terminal, $\tau \equiv \langle s, h \rangle$ or **abort**
2. non-terminal: $\langle c, s, h \rangle$

Thus $\langle c, s, h \rangle \rightsquigarrow \langle s', h' \rangle$ reads as: *"Running a command c from the state s, h terminates in the state s', h'"*.

$(f \mid x \mapsto y)$ means that the function f is the same, but with x mapping to y. Later we will use σ as an abbreviation of $\langle s, h \rangle$.

$$\langle \textbf{skip}, s, h \rangle \rightsquigarrow s, h$$

$$\frac{\langle c_0, s, h \rangle \rightsquigarrow s'', h'' \qquad \langle c_1, s'', h'' \rangle \rightsquigarrow \tau}{\langle c_0; c_1, s, h \rangle \rightsquigarrow \tau} \qquad \frac{\langle c_0, s, h \rangle \rightsquigarrow \textbf{abort}}{\langle c_0; c_1, s, h \rangle \rightsquigarrow \textbf{abort}}$$

$$\frac{[\![b]\!]_s = true \qquad \langle c_0, s, h \rangle \rightsquigarrow \tau}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, s, h \rangle \rightsquigarrow \tau} \qquad \frac{[\![b]\!]_s = false \qquad \langle c_1, s, h \rangle \rightsquigarrow \tau}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, s, h \rangle \rightsquigarrow \tau}$$

$$\frac{[\![b]\!]_s = false}{\langle \textbf{while } b \textbf{ do } c, s, h \rangle \rightsquigarrow s, h}$$

$$\frac{[\![b]\!]_s = true \qquad \langle c, s, h \rangle \rightsquigarrow s'', h'' \qquad \langle \textbf{while } b \textbf{ do } c, s'', h'' \rangle \rightsquigarrow s', h'}{\langle \textbf{while } b \textbf{ do } c, s, h \rangle \rightsquigarrow s', h'}$$

$$\frac{}{\langle x := e, s, h \rangle \rightsquigarrow s', h} \quad , s' = (s | x \mapsto [\![e]\!]_s)$$

Table 2.4: Semantics of commands: While Commands

Semantics of the heap commands is given in the Table 2.5. Notation $h \cdot h'$, means the composition of two disjoint heaps h and h'.

Only one command of the *while* - allocation - is changed compared to Reynolds' version. Now it allows to allocate and initialize a single cell, which will be minimal inactive cell in the heap. Notice *minimal*, this makes the allocation deterministic. *lookup*

alloc	$\overline{\langle x := \mathbf{new}\ e, s, h\rangle \rightsquigarrow s', h'}$	$, s' = (s\|x \mapsto l), \quad h' = (h \cdot [l \mapsto [\![e]\!]_s])$ and $l = min(Loc - dom(h))$
lookup	$\dfrac{[\![e]\!]_s \notin dom(h)}{\langle x := [e], s, h\rangle \rightsquigarrow \mathbf{abort}}$	
	$\dfrac{[\![e]\!]_s \in dom(h)}{\langle x := [e], s, h\rangle \rightsquigarrow s', h}$	$, s' = (s\|x \mapsto h([\![e]\!]_s))$
mutate	$\dfrac{[\![e]\!]_s \notin dom(h)}{\langle [e] := e', s, h\rangle \rightsquigarrow \mathbf{abort}}$	
	$\dfrac{[\![e]\!]_s \in dom(h)}{\langle [e] := e', s, h\rangle \rightsquigarrow s, h'}$	$, h' = (h\|[\![e]\!]_s \mapsto [\![e']\!]_s)$
dealloc	$\dfrac{[\![e]\!]_s \notin dom(h)}{\langle \mathbf{dispose}\ e, \sigma\rangle \rightsquigarrow \mathbf{abort}}$	
	$\dfrac{[\![e]\!]_s \in dom(h)}{\langle \mathbf{dispose}\ e, s, h\rangle \rightsquigarrow s, h'}$	$, h = (h_0 \cdot [[\![e]\!]_s \mapsto -])$ and $h' = h_0$

Table 2.5: Semantics of commands: Heap Operations

reads the value at the location of e and writes it to x. *mutate* updates the location e with value of e'. *dispose* deallocates location e from the heap.
All commands result in **abort** when trying to access inactive (not in the heap) location. Additionally any command aborts if boolean or expression used is **undefined**.

Chapter 3

Permutation

Now that we have a language with deterministic memory allocation it is time to introduce the location renaming. With allocation becoming deterministic we need another method to ensure that a program proof does not depend on details of allocator's work, and location renaming is such a method. Location renaming is the result of applying a permutation π, which is just a bijective function on Loc:

$\pi : Loc \to Loc$

This function lifted to a value, store or a heap consistently renames all locations that appear inside. Having such a function is not enough yet, we need to prove *permutation theorem*, which says that "running a consistently renamed program on a correspondingly renamed heap leads to the same result, up to renaming of locations in the final heap". This means that it is always possible to find such a permutation, which when applied to the state resulting from running a command in a renamed(by another permutation) state would give us the original resulting state as if there was no permutation. This way we get the invariance under location renaming.

3.1 Definition

Permutation is a *bijective* function $\pi : Loc \to Loc$. :
Since $Values = Integers \uplus Locations \uplus Atoms \uplus Boolean$ we can lift it to $Values, Stores$ and $Heaps$ as follows:

- $\pi^{val} : Values \to Values$

$$\pi^{val}(v) = \begin{cases} v & v \in Int \\ \pi(v) & v \in Loc \\ v & v \in Atom \\ v & v \in Boolean \end{cases}$$

As expected permutation on values has effect only when the value is a location.

- lift to $\pi^s : Stores \to Stores$ where $Stores : Var \to Values$
  $(\pi^s(s))(x) = (\pi^{val}(s(x))$
- lift to $\pi^h : Heaps \to Heaps$ where $Heaps : Loc \rightharpoonup Values$
  $(\pi^h(h))(l) = (\pi^{val}(h(\pi^{-1\ val}(l)))$

We then define the action $\pi \bullet \sigma$ of a permutation π on a configuration σ by:

$$\pi \bullet \sigma = (\pi^s(s), \pi^h(h))$$

Later we will drop superscripts if clear from context or add them if needed.

3.2 Permutation Theorem

In this section we will prove permutation theorem, vital for having invariance under location renaming:

Theorem 3.1 (Permutation) *For all c, σ, σ' and π holds:*

1. $\langle c, \sigma\rangle \rightsquigarrow \sigma' \quad \Rightarrow \quad \exists \pi'. \langle c, \pi \bullet \sigma\rangle \rightsquigarrow \sigma'' \wedge \sigma' = \pi' \bullet \sigma''$
2. $\langle c, \sigma\rangle \rightsquigarrow \textbf{abort} \quad \Rightarrow \quad \langle c, \pi \bullet \sigma\rangle \rightsquigarrow \textbf{abort}$

Before proving this theorem we'll prove some useful lemmas.

Lemma 3.1 $\forall \pi, e, s.\ [\![e]\!]_s \neq \textbf{undefined} \Rightarrow [\![e]\!]_{\pi^s(s)} = \pi^{val}([\![e]\!]_s)$

Proof: By structural induction on e we have:

1. $e = n$
   $[\![n]\!]_{\pi^s(s)} = n = \pi^{val}([\![n]\!]_s) \qquad$ *by Def. of* $[\![n]\!]_s$ *and* π^{val}
2. $e = x$
   $[\![x]\!]_{\pi^s(s)} = \pi^{val}(s(x)) = \pi^{val}([\![x]\!]_s) \qquad$ *by Def. of* $[\![x]\!]_s$ *and* $\pi^s(s)$ *and* π^{val}

3. $e = e_0 + e_1$ (cases for $-, \times$ analogously)
   $[\![e_0 + e_1]\!]_{\pi^s(s)} = [\![e_0]\!]_{\pi^s(s)} + [\![e_1]\!]_{\pi^s(s)}$ *by Def. of +*
   $= \pi^{val}([\![e_0]\!]_s) + \pi^{val}([\![e_1]\!]_s) = \pi^{val}([\![e_0 + e_1]\!]_s)$ *by IH*

□

Lemma 3.2

1. $\pi' \bullet (\pi \bullet \sigma) = (\pi' \circ \pi) \bullet \sigma$
2. $id \bullet \sigma = \sigma$

Proof:

1.
$$\begin{array}{rll} \pi' \bullet (\pi \bullet \sigma) = & \pi' \bullet (\pi^s(s), \pi^h(h)) & \textit{by Def. of } \pi \bullet \sigma \\ = & (\pi'^s(\pi^s(s)), \pi'^h(\pi^h(h))) & \textit{by Def. of } \pi \bullet \sigma \\ = & ((\pi' \circ \pi)^s(s), (\pi' \circ \pi)^h)(h) & \textit{Function composition} \\ = & (\pi' \circ \pi) \bullet \sigma & \textit{by Def. of} \pi \bullet \sigma \end{array}$$

□

2. *id* is an identity function , therefore so are id^{val}, id^s, id^h:
   $id \bullet \sigma = (id^s(s), id^h(h)) = s, h = \sigma$ □

Lemma 3.3 $dom(\pi^h(h)) = \{\pi(l) \mid l \in dom(h)\} = \{l' \mid \pi^{-1}(l) \in dom(h)\}$

Proof:

$$\begin{array}{rll} dom(\pi^h(h)) & = \{l \in Loc \mid \pi(h(\pi^{-1}(l))) \text{ is defined}\} & \textit{by Def.} \\ & = \{l \in Loc \mid h(\pi^{-1}(l)) \text{ is defined }\} & \pi \textit{ total} \\ & = \{l \in Loc \mid \pi^{-1}(l) \in dom(h)\} & \textit{by Def.} \end{array}$$

□

Now we have everything to prove the theorem. We prove it by structural induction on derivations of $\langle c, \sigma \rangle \rightsquigarrow \sigma'$, walking through each command. For the first part we need to provide such a function π', that when it is applied to the permutated state its result is the original state. In most cases such a function is simply inverse of original permutation. But the case with allocation operator is more tricky, since it allocates the minimal inactive cell in the heap and there is no guarantee that after permutation the minimal cell will remain to be the minimal.
In this part we consider only the cases which do not **abort**.

Part 1.

$$\langle c, \sigma \rangle \rightsquigarrow \sigma' \quad \Rightarrow \quad \exists \pi'. \langle c, \pi \bullet \sigma \rangle \rightsquigarrow \sigma'' \wedge \sigma' = \pi' \bullet \sigma''$$

Proof:

1. **skip:**
   $\langle \textbf{skip}, \sigma \rangle \rightsquigarrow \sigma$
   $\langle \textbf{skip}, \pi \bullet \sigma \rangle \rightsquigarrow \pi \bullet \sigma$

   Let $\pi' := \pi^{-1}$, then

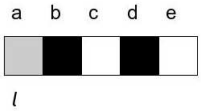
   $$\begin{array}{rcll} \pi' \bullet \sigma'' = & \pi' \bullet (\pi \bullet \sigma) & & by\ def.\ of\ \sigma'' \\ = & (\pi' \circ \pi) \bullet \sigma & & by\ Lemma\ 2 \\ = & (\pi^{-1} \circ \pi) \bullet \sigma & & by\ def.\ of\ \pi' \\ = & Id \bullet \sigma & & by\ def.\ of\ Id \\ = & \sigma & & by\ Lemma\ 2 \end{array}$$

   □

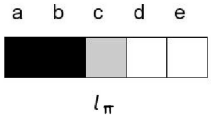
2. **new:**
   $\langle x := \textbf{new}\ e, s, h \rangle \rightsquigarrow (s|x \mapsto l), (h * [l \mapsto [\![e]\!]_s])$
   $\langle x := \textbf{new}\ e, \pi^s(s), \pi^h(h) \rangle \rightsquigarrow (\pi^s(s)|x \mapsto l_\pi), (\pi^h(h) * [l_\pi \mapsto [\![e]\!]_{\pi^s(s)}])$

   $\sigma' = (s|x \mapsto l), (h * [l \mapsto [\![e]\!]_s])$

   Consider the original resulting heap looks as follows, with l(grey square) marking the minimal available(white square) location:

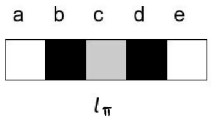
   And let's suppose the heap after applying π is as follows, with l_π marking the new minimal available location:

   $\sigma'' = (\pi^s(s)|x \mapsto l_\pi), (\pi^h(h) * [l_\pi \mapsto [\![e]\!]_{\pi^s(s)}])$

   $\sigma' = \pi' \bullet \sigma''$

   Let π be such that original l after applying π was mapped **not** to the minimal free location, then if we just take $\pi' = \pi^{-1}$ and apply it to σ'' heap will be something like this, with l being not minimal:



which is surely wrong.
In that case we can repair π' by taking it to be $\pi' = \pi^{-1}[l_\pi := l; \pi(l) := l_\pi]$. That is the same π^{-1}, but with cells mapped uncorrectly swapped to fix the problem: l_π, the newly allocated cell, is mapped to l, the location which would be allocated in original not renamed heap, and $\pi(l)$, the location original would-be-allocated cell is renamed to is mapped to l_π.

□

3. **lookup:**
$\langle x := [e], s, h\rangle \rightsquigarrow (s|x \mapsto h([\![e]\!]_s)), h$
$\langle x := [e], \pi^s(s), \pi^h(h)\rangle \rightsquigarrow (\pi^s(s)|x \mapsto \pi^h(h)([\![e]\!]_{\pi^s(s)})), \pi^h(h)$

$\sigma' = (s|x \mapsto h([\![e]\!]_s)), h$
$\sigma'' = (\pi^s(s)|x \mapsto \pi^h(h)([\![e]\!]_{\pi^s(s)})), \pi^h(h)$

Let $\pi' := \pi^{-1}$, then

$$\begin{array}{rcll} \pi' \bullet \sigma'' = & \pi' \bullet (\pi \bullet \sigma) & & by\ def.\ of\ \sigma'' \\ = & (\pi' \circ \pi) \bullet \sigma & & by\ Lemma\ 2 \\ = & (\pi^{-1} \circ \pi\bullet)\sigma & & by\ def.\ of\ \pi' \\ = & Id \bullet \sigma & & by\ def.\ of\ Id \\ = & \sigma & & by\ Lemma\ 2 \end{array}$$

□

4. **mutation:**
$\langle [e] := e', s, h\rangle \rightsquigarrow s, (h|[\![e]\!]_s \mapsto [\![e']\!]_s)$

$\langle [e] := \pi \bullet e', \pi^s(s), \pi^h(h)\rangle \rightsquigarrow \pi^s(s), (\pi^h(h)|[\![e]\!]_{\pi^s(s)} \mapsto [\![e']\!]_{\pi^s(s)})$

$\sigma' = s, (h|[\![e]\!]_s \mapsto [\![e']\!]_s)$
$\sigma'' = \pi^s(s), (\pi^h(h)|[\![e]\!]_{\pi^s(s)} \mapsto [\![e']\!]_{\pi^s(s)})$

Let $\pi' := \pi^{-1}$, then

$$\begin{array}{rcll} \pi' \bullet \sigma'' = & \pi' \bullet (\pi \bullet \sigma) & & by\ def.\ of\ \sigma'' \\ = & (\pi' \circ \pi) \bullet \sigma & & by\ Lemma\ 2 \\ = & (\pi^{-1} \circ \pi) \bullet \sigma & & by\ def.\ of\ \pi' \\ = & Id \bullet \sigma & & by\ def.\ of\ Id \\ = & \sigma & & by\ Lemma\ 2 \end{array}$$

□

5. **dispose:**
   $\langle \textbf{dispose}\ e, s, h\rangle \rightsquigarrow s, h - [\![e]\!]_s$

   $\langle \textbf{dispose}\ e, \pi^s(s), \pi^h(h)\rangle \rightsquigarrow \pi^s(s), \pi^h(h) - [\![e]\!]_{\pi^s(s)}$

   $\sigma' = s, h - [\![e]\!]_s$
   $\sigma'' = \pi^s(s), \pi^h(h) - [\![e]\!]_{\pi^s(s)}$
   Let $\pi' := \pi^{-1}$, then

$$\begin{array}{rcll} \pi' \bullet \sigma'' = & \pi' \bullet (\pi \bullet \sigma) & & by\ def.\ of\ \sigma'' \\ = & (\pi' \circ \pi) \bullet \sigma & & by\ Lemma\ 2 \\ = & (\pi^{-1} \circ \pi) \bullet \sigma & & by\ def.\ of\ \pi' \\ = & Id \bullet \sigma & & by\ def.\ of\ Id \\ = & \sigma & & by\ Lemma\ 2 \end{array}$$

□

6. **assignment**:
   $\langle x := e, s, h\rangle \rightsquigarrow (s|x \mapsto [\![e]\!]_s), h$
   $\langle x := e, \pi^s(s), \pi^h(h)\rangle \rightsquigarrow (\pi^s(s)|x \mapsto [\![e]\!]_{\pi^s(s)}), \pi^h(h)$

   $\sigma' = (s|x \mapsto [\![e]\!]_s), h$
   $\sigma'' = (\pi^s(s)|x \mapsto [\![e]\!]_{\pi^s(s)}), \pi^h(h)$
   Let $\pi' := \pi^{-1}$, then

$$\begin{array}{rcll} \pi' \bullet \sigma'' = & \pi' \bullet (\pi \bullet \sigma) & & by\ def.\ of\ \sigma'' \\ = & (\pi' \circ \pi) \bullet \sigma & & by\ Lemma\ 2 \\ = & (\pi^{-1} \circ \pi) \bullet \sigma & & by\ def.\ of\ \pi' \\ = & Id \bullet \sigma & & by\ def.\ of\ Id \\ = & \sigma & & by\ Lemma\ 2 \end{array}$$

□

   At this point we have proven all the base cases.

7. **conditional**:
   $\langle \textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1, s, h\rangle \rightsquigarrow s', h'$
   $\langle \textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1, \pi^s(s), \pi^h(h)\rangle \rightsquigarrow s'', h''$

$\sigma' = s', h'$
$\sigma'' = s'', h''$

Depending on b, either c_0 or c_1 is executed:

- case b *true*
  $\langle c_0, s, h\rangle \rightsquigarrow s', h'$
  $\langle c_0, \pi^s(s), \pi^h(h)\rangle \rightsquigarrow s'', h''$
  By induction hypothesis we can conclude that π' exists.
- case b *false*
  analogously.

□

while, *sequencing* are proved the same way.

Part2.

$$\langle c, \sigma\rangle \rightsquigarrow \textbf{abort} \quad \Rightarrow \quad \langle c, \pi \bullet \sigma\rangle \rightsquigarrow \textbf{abort}$$

8. **abort** cases:
   abort arises when command tries to access inactive memory ($[\![e]\!]_s \notin dom(h)$). To show
   $$(c, \pi^s(s), \pi^h(h)) \rightsquigarrow \textbf{abort}$$
   it suffices to show: $[\![e]\!]_{\pi^s(s)} \notin dom(\pi^h(h))$ whenever $[\![e]\!]_s \notin dom(h)$

   Suppose $[\![e]\!]_s \notin dom(h)$ and $[\![e]\!]_{\pi^s(s)} \in dom(\pi^h(h))$, then

   (a) $[\![e]\!]_{\pi^s(s)} = \pi(l)$, for some $l \in dom(h)$, then *by Lemma 3.3*

   (b) $\pi^{-1\ val}([\![e]\!]_{\pi^s(s)}) = \pi^{-1\ val}(\pi^{val}(l)) = l$ *by Lemma 3.1*
   $l = [\![e]\!]_s$. *contradiction*
   Therefore $[\![e]\!]_{\pi^s(s)} \notin dom(\pi^h(h))$

□

Chapter 4

Logic

In this section we will provide the separation logic for our language with deterministic semantics. We change almost nothing here compared to Reynolds. Separation logic contains all of the boolean expressions, classical logic plus adds special assertions to describe the heap. For the purposes of checking the type correctness we added another two predicates, testing whether expression is boolean or is defined.
Hoare triples specifying the program will be interpreted according to the partial correctness.

4.1 Assertions

Assertions of our logic are usual formulae of separation logic(without separating implication):

$p, q, r ::= b \mid p \Rightarrow q \mid \forall x.p$	
$\mid \mathbf{emp}$	*empty heap*
$\mid e \mapsto f$	*singleton heap*
$\mid p * q$	*separating conjunction*
$\mid isbool(e)$	*e is boolean*
$\mid isdef(e)$	*e is defined*

Table 4.1: Assertions

Meaning of an assertion depends on both store and heap. When an assertion p is **true** in some state σ we say that *the state satisfies the assertion* and write:

$$\sigma \models p$$

Abbreviations are as usual: $e_0 = e_1 : \ e_0 \leq e_1 \wedge e_1 \leq e_0$; $\exists x.p : \neg\forall x.\neg p$; $x \mapsto - : \exists v.x \mapsto v$.
The meaning of heap assertions:

- $isbool(e)$ is **true** only when e is boolean:

$$\sigma \models isbool(e) \Leftrightarrow [\![e]\!]_s \in boolean$$

- $isdef(e)$ is **true** only when e is defined:

$$\sigma \models isdef(e) \Leftrightarrow [\![e]\!]_s \neq \textbf{undefined}$$

- **emp** is **true** only when the current heap is empty:

$$\sigma \models \textbf{emp} \Leftrightarrow dom(h) = \oslash$$

- $e \mapsto f$ is **true** only when the current heap contains one element at the location e with the value f:

$$\sigma \models e \mapsto f \Leftrightarrow dom(h) = \{[\![e]\!]_s\} \text{ and } h([\![e]\!]_s) = [\![e']\!]_s$$

- $\mathbf{p} * \mathbf{q}$ is **true** only when the current heap can be split into two disjoint parts and for each of them p and q hold respectively:

$$s, h \models p * q \Leftrightarrow \exists h_0, h_1. \quad \begin{array}{l} h_0 \perp h_1 \text{ and } h_0 \cdot h_1 = h \text{ and} \\ s, h_0 \models p \text{ and } s, h_1 \models q \end{array}$$

  Two heaps are disjoint if they have disjoint domains:

$$dom(h_0) \cap dom(h_1) = \oslash$$

  We write $h_0 \perp h_1$ to show that h_0 is disjoint from h_1, and $h_0 \cdot h_1$ to denote the heap combined from these disjoint parts.

4.2 Partial Correctness

Assertions and commands are combined into *Hoare triples* to reason about programs:

$$\{p\}c\{q\}$$

p is a *precondition*, q is a *postcondition*, c is a command. A triple describes how the execution of a command changes the state of the program, from state satisfying the

precondition to the state satisfying the postcondition. In our work we are going to consider *partial correctness* interpretation of Hoare triples, i.e. we don't take into account requirement for the command to necessarily terminate, which would give us a *total correctness*.
We say that the command is *safe*, if it does not abort: $\langle c, \sigma \rangle \not\rightsquigarrow$ **abort**, and that Hoare triple is true $\models \{p\}c\{q\}$, iff in all states satisfying precondition executing command is safe and the resulting state satisfies the postcondition:

$$\models \{p\}c\{q\} \Leftrightarrow \forall \sigma. \quad \begin{array}{l} \sigma \models p \Rightarrow \langle c, \sigma \rangle \text{ is safe, and} \\ \langle c, \sigma \rangle \rightsquigarrow \sigma' \Rightarrow \sigma' \models q \end{array}$$

4.3 Inference rules

Inference rules for Hoare triples are given in Table 4.2.

skip

$$\frac{}{\{p\}\textbf{skip}\{p\}}$$

assign

$$\frac{}{\{p[e/x]\}x := e\{p\}}$$

seq

$$\frac{\{p\}c_0\{t\} \qquad \{t\}c_1\{q\}}{\{p\}c_0; c_1\{q\}}$$

cond

$$\frac{\{p \wedge b\}c_0\{q\} \qquad \{p \wedge \neg b\}c_1\{q\}}{\{p \wedge isbool(b)\}\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1\{q\}}$$

while

$$\frac{\{p \wedge b\}c\{p\}}{\{p\}\textbf{while} \wedge \textbf{isbool}(\textbf{b})\ b \textbf{ do } c\{\neg b \wedge p\}}$$

weak

$$\frac{p \Rightarrow s \qquad \{s\}c\{t\} \qquad t \Rightarrow q}{\{p\}c\{q\}}$$

alloc

$$\frac{}{\{\textbf{emp} \wedge isdef(e)\}x := \textbf{new } e\{x \mapsto e\}}$$

mutate

$$\frac{}{\{e \mapsto -\}[e] := e'\{e \mapsto e'\}}$$

lookup

$$\frac{}{\{x = y \wedge (e \mapsto z)\}x := [e]\{x = z \wedge (e[x := y] \mapsto z)\}}$$

dispose

$$\frac{}{\{e \mapsto -\}\textbf{dispose } e\{\textbf{emp}\}}$$

frame

$$\frac{\{p\}c\{q\}}{\{p * r\}c\{q * r\}} \qquad , mod(c) \cap fv(r) = \{\}$$

Table 4.2: Inference rules

Rules remain the same as in separation logic. *Skip* doesn't change anything. The *assignment* axiom states that after the assignment any predicate holds for the variable that was previously true for the right-hand side of the assignment. $p[e/x]$ denotes the assertion p in which all free occurrences of the variable x have been replaced with the expression e. As an example:

$$\{y + 1 = 42 \wedge y = 5\}x := y + 1\{0 \leq x\}$$

Sequencing allows composing commands if the postcondition of the first one mathes precondition of the second. *Conditional* allows executing one of two commands depending on some condition. *While* loops a command depending on condition. *Weakening* strengthens the precondition and weakens the postcondition.

Allocation: starting in a state with empty heap execution ends in a state with single cell allocated. *Mutation*: starting in a state where heap contains a single active cell and store with a variable pointing to this memory cell changes the content of this cell. *Lookup*: assigns to the variable x the content of the heap cell specified by expression e. *Dispose*: starting in a state with single cell allocated and pointed to by variable, ends in a state with an empty heap.

The most important is the Frame rule. It has as a side condition the requirement for free variables of r, $fv(r)$, not to be modified by c, $mod(c)$ is the set of variables updated by c. The rule is the key to *local reasoning* about the heap. Local reasoning reflects the informal intuition of programmers who usually concentrate only on relevant resources used by part of the program:

> To understand how a program works, it should be possible for reasoning and specification to be confined to the cells that the program actually accesses. The value of any other cell will automatically remain unchanged.[8]

Frame rule allows deriving global versions of the previously described rules, e.g. from local mutation:

$$\frac{}{\{x \mapsto -\}[x] := e\{x \mapsto e\}}$$

to global:

$$\frac{}{\{(x \mapsto -) * r\}[x] := e\{(x \mapsto e) * r\}}$$

Lemma 4.1

$\forall p. \forall s, s',\ h.\ s(x) = s'(x)\ for\ all\ fv(p) \Rightarrow ((s,h) \models p \Rightarrow (s',h) \models p)$

Proof by induction on p.

Lemma 4.2

$\forall p. \forall s, h, \pi.\ (s, h) \models p \Rightarrow (\pi^s(s), \pi^h(h)) \models p$

Proof by induction on p.

Chapter 5

Soundness

Soundness of our logic is proved by rule induction. Proof will be split into three parts: Hoare rules, Heap rules and the most important - frame rule.

1. **Traditional Hoare rules**

   (a) **skip**
   To show: $\models \{p\}\mathbf{skip}\{p\}$

   1)Assume $\sigma \models p$
   2)By rule for skip: $\langle \mathbf{skip}, \sigma \rangle \rightsquigarrow \sigma$

   $\Rightarrow \models \{p\}\mathbf{skip}\{p\}$

   (b) **assignment**
   To show: $\models \ \{p[e/x]\}x := e\{p\}$

   1)Assume $s, h \models p[e/x]$
   2)By rule for assignment: $\langle x := e, s, h \rangle \rightsquigarrow s \mid x \mapsto [\![e]\!]_s, h$
   3)Using lemma 6.9 from [15] $s \mid x \mapsto [\![e]\!]_s, h \models \ p$

   $\Rightarrow \models \ \{p[e/x]\}x := e\{p\}$

   (c) **sequencing**
   To show: $\dfrac{\models \{p\}c_0\{t\} \qquad \models \{t\}c_1\{q\}}{\models \{p\}c_0; c_1\{q\}}$

1) Assume $\models \{p\}c_0\{t\}; \models \{t\}c_1\{q\}$
2) Suppose $\sigma \models p$
3) Then $\langle c_0, \sigma\rangle \rightsquigarrow \sigma'$ and from 1) $\sigma' \models t$

4) Then $\langle c_1, t\rangle \rightsquigarrow \sigma''$ and from 1) $\Rightarrow\ \sigma'' \models q$

$$\Rightarrow \frac{\models \{p\}c_0\{t\} \qquad \models \{t\}c_1\{q\}}{\models \{p\}c_0; c_1\{q\}}$$

(d) **conditional**

To show: $\dfrac{\models \{p \wedge b\}c_0\{q\} \qquad \models \{p \wedge \neg b\}c_1\{q\}}{\models \{p \wedge \mathit{isbool}(b)\}\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1\{q\}}$

1) Assume $\models \{p \wedge b\}c_0\{q\}; \models \{p \wedge \neg b\}c_1\{q\}$
2) Suppose $\sigma \models p \wedge \mathit{isbool}(b)$
then 3) if $\sigma \models b$ then 1) and $\langle c_0, \sigma\rangle \rightsquigarrow \sigma' \models q$ or
if $\sigma \models \neg b$ then 1) and $\langle c_1, \sigma\rangle \rightsquigarrow \sigma'' \models q$

$$\Rightarrow \frac{\models \{p \wedge b\}c_0\{q\} \qquad \models \{p \wedge \neg b\}c_1\{q\}}{\models \{p\}\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1\{q\}}$$

(e) **while-loop**

To show: $\dfrac{\models \{p \wedge b\}c\{p\}}{\models \{p \wedge \mathit{isbool}(b)\}\textbf{while } b \textbf{ do } c\{\neg b \wedge p\}}$

1) Assume $\models \{p \wedge b\}c\{p\}$
2) Suppose $\sigma \models p$
case $\sigma \models \neg b$:

$\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \rightsquigarrow \sigma$

$\sigma \models p \wedge \neg b$

case $\sigma \models b$:

$\langle c, \sigma\rangle \rightsquigarrow \sigma_3; \langle \textbf{while } b \textbf{ do } c, \sigma_3\rangle \rightsquigarrow \sigma_2$

3) 1)$\Rightarrow \sigma_3 \models p$
4)By induction $\sigma_2 \models p \wedge \neg b$

$$\Rightarrow \frac{\models \{p \wedge b\}c\{p\}}{\models \{p\}\textbf{while } b \textbf{ do } c\{\neg b \wedge p\}}$$

(f) **weakening**

To show: $\dfrac{\models p \Rightarrow s \qquad \models \{s\}c\{t\} \qquad \models t \Rightarrow q}{\models \{p\}c\{q\}}$

1) Assume $\models p \Rightarrow s$; $\models \{s\}c\{t\}$; $\models t \Rightarrow q$
2) Suppose $\sigma \models p$
3) From 1) and 2): $\sigma \models s$
4) From 1) and 3): $\Rightarrow \ \langle c, \sigma\rangle \rightsquigarrow \sigma' \models t$
5) From 1): $\Rightarrow \ \sigma' \models q$

$\Rightarrow \dfrac{\models p \Rightarrow s \qquad \models \{s\}c\{t\} \qquad \models t \Rightarrow q}{\models \{p\}c\{q\}}$

2. **Heap rules**

(a) **allocation**

To show: $\models \{\mathbf{emp} \wedge isdef(e)\}x := \mathbf{new}\ e\{x \mapsto e\}$

Suppose $s, h \models \mathbf{emp} \wedge \mathbf{isdef}(\mathbf{e})$
then $\langle x := \mathbf{new}\ e, s, h\rangle \rightsquigarrow \langle s \mid x : l, l \rightarrow [\![e]\!]_s\rangle$ and

$\langle s \mid x : l, l \rightarrow [\![e]\!]_s\rangle \models x \mapsto e$

(b) **mutation**

To show: $\models \{x \mapsto - \wedge isdef(e)\}[x] := e\{x \mapsto e\}$

Suppose $s, h \models x \mapsto - \wedge isdef(e)$
then $\langle [x] := e, s, h\rangle \rightsquigarrow \langle s, [\![x]\!]_s \rightarrow [\![e]\!]_s\rangle$ and

$\langle s, [\![x]\!]_s \rightarrow [\![e]\!]_s\rangle \models x \mapsto e$

(c) **lookup**

To show:$\models \{x = y \wedge (e \mapsto z) \wedge isdef(e)\}x := [e]\{x = z \wedge (e' \mapsto z)\}$, $e' = e[x := y]$

Lemma 5.1 $[\![e[x := y]]\!]_s = [\![e]\!]_{s[x:=s(y)]}$

Suppose $s, h \models x = y \wedge (e \mapsto z)$
then $\langle x := [e], s, h\rangle \rightsquigarrow \langle s|x \rightarrow h([\![e]\!]_s), h\rangle$ and

$\langle s|x \rightarrow h([\![e]\!]_s), h\rangle \models x = z \wedge (e' \mapsto z)$ *by Lemma 5.1*

(d) **deallocation**
To show:$\models \{e \mapsto -\}$**dispose** $e\{$**emp**$\}$

Suppose $s, h \models e \mapsto -$
then $\langle$**dispose** $e, s, h\rangle \rightsquigarrow (s,$ **emp**$)$ and

$(s,$ **emp**$) \models$ **emp**

3. **Frame rule**
To show: $\dfrac{\models \{p\}c\{q\}}{\models \{p * r\}c\{q * r\}}$, $mod(c) \cap fv(r) = \{\}$

Extension of permutation theorem

In order to prove the frame rule we will rephrase semantic property of safety monotonicity and the Frame property from[17] according to permutation theorem.

Lemma 5.2 (Safety monotonicity)

- *If $\langle c, s, h\rangle$ safe, and π- renaming and $(\pi^h(h)) \perp h'$,*
  then $\langle c, \pi^s(s), \pi^h(h) \cdot h'\rangle$ is safe

Proof: Safety monotonicity
Proof is similar to Part 2 of permutation theorem.
Suppose $\langle c, s, h\rangle$ safe and $\langle c, \pi^s(s), \pi^h(h) \cdot h'\rangle$ isn't. Then $[\![e]\!]_{\pi^s(s)} \notin dom(\pi^h(h) \cup dom(h'))$, and therefore also $[\![e]\!]_{\pi^s(s)} \notin dom(\pi^h(h))$.

(a) $[\![e]\!]_{\pi^s(s)} = \pi(l)$, for some $l \in dom(h)$, then *by Lemma 3.3*
(b) $\pi^{-1\ val}([\![e]\!]_{\pi^s(s)}) = \pi^{-1\ val}(\pi^{val}(l)) = l$ *by Lemma 3.1*
$l = [\![e]\!]_s$. *contradiction*

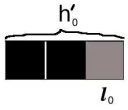
Contradiction. □

Lemma 5.3 (Frame property) .

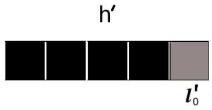
$$c \text{ is safe for } \sigma = (s, h_0) \text{ and } \langle c, s, h_0 \cdot h_1 \rangle \rightsquigarrow (s', h') \Rightarrow$$
$$\exists \pi, h'_0.\ \langle c, s, h_0 \rangle \rightsquigarrow (s'', h'_0) \text{ and } h' = (\pi^h(h'_0)) \cdot h_1,\ s' = \pi^s(s'')$$

Proof by structural induction:

(a) *allocation*
$\langle x := \mathbf{new}\ e, s, h_0 \rangle \rightsquigarrow \langle (s \mid x \mapsto l_0), (h_0 \cdot l_0 \mapsto [\![e]\!]_s)) \rangle$, l_0 - minimal



$\langle x := \mathbf{new}\ e, s, h_0 \cdot h' \rangle \rightsquigarrow \langle (s \mid x \mapsto l'_0), (h_0 \cdot h_1 \cdot l'_0 \mapsto [\![e]\!]_s)) \rangle$, l'_0 - minimal



$\pi = id \mid l_0 \mapsto l'_0, h'_0 = h_0 \cdot l'_0 \mapsto [\![e]\!]_s$ □

(b) *mutation*:
$\langle [x] := e, s, h_0 \rangle \rightsquigarrow \langle s, (h_0 \mid [\![x]\!]_s \mapsto [\![e]\!]_s) \rangle$
$\langle [x] := e, s, h_0 \cdot h_1 \rangle \rightsquigarrow \langle s, (h_0 \cdot h_1 \mid [\![x]\!]_s \mapsto [\![e]\!]_s) \rangle$

$\pi = id, h'_0 = h_0 \mid [\![x]\!]_s \mapsto [\![e]\!]_s$

□

(c) *lookup*:
$\langle x := [e], s, h_0 \rangle \rightsquigarrow \langle (s \mid x \mapsto h([\![e]\!]_s)), h_0 \rangle$
$\langle x := [e], s, h_0 \cdot h_1 \rangle \rightsquigarrow \langle (s \mid x \mapsto h([\![e]\!]_s)), h_0 \cdot h_1 \rangle$

$\pi = id, h'_0 = h_0$

□

(d) *dispose*:
$\langle \mathbf{dispose}\ e, s, h_0 \cdot [\![e]\!]_s \mapsto - \rangle \rightsquigarrow \langle s, h_0 \rangle$
$\langle \mathbf{dispose}\ e, s, h_0 \cdot [\![e]\!]_s \mapsto - \cdot h_1 \rangle \rightsquigarrow \langle s, h_0 \cdot h_1 \rangle$

$\pi = id, h'_0 = h_0$

Cases for **skip**, and *assignment* are similar to the last two rules, since they don't modify the heap.
□ Base cases are proven. Cases for *conditional, while* and *sequencing* are by induction.

□

Now back to the Frame rule:

Need to prove: $\dfrac{\models \{p\}c\{q\}}{\models \{p * r\}c\{q * r\}}$, $mod(c) \cap fv(r) = \{\}$

Suppose $s, h \models p * r$, i.e.:

$(s, h_0) \models p$ and $(s, h_1) \models r$
To show:

(a) $\langle c, s, h\rangle$ safe,

(b) $\langle c, s, h\rangle \rightsquigarrow \langle s', h'\rangle \Rightarrow s', h' \models q * r$,

Safety of $\langle c, s, h\rangle$ follows from safety monotonicity.

If $\langle c, (s, h_0 \cdot h_1)\rangle \rightsquigarrow (s', h')$, then $\exists \pi, h'_0.\langle c, (s, h_0)\rangle \rightsquigarrow \langle s'', h'_0\rangle$

$$\wedge h' = (\pi \bullet h'_0) \cdot h_1, \quad s' = \pi \bullet s'' \qquad \text{by Lemma 5.3}$$

$\Rightarrow (s', h'_0) \models q$ by assumption $\models \{p\}c\{q\}$

$\Rightarrow (\pi \bullet s'', \pi \bullet h'_0) \models q$ by Lemma 4.2

$\Rightarrow (\pi \bullet s', h') \models q * r$ by Lemma 4.1 and $mod(c) \cap fv(r) = \{\}$

□

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this work we have followed the direction stated by Yang/O'Hearn in [17] to walk around the non-determinancy in allocation. We confirmed that allocation is not necessarily should be non-deterministic, if we have a language without address arithmetic and invariance under location renaming. The central contribution of this work is a formal statement and proof of the permutation theorem. This proof with the given deterministic semantics shows that running a program on a correspondingly renamed heap leads to the same result, up to a renaming of locations in the final heap. We have shown that it is always possible to find the needed permutation, and in most cases it was simply the inverse of original permutation. The only problem was with allocation, since renaming could map the original minimal location to arbitrary cell, not necessarily being also minimal, and we have shown that suitable permutation is found in this case by "swapping" uncorrectly mapped locations. Next we gave the separation logic for our language and proved its soundness.

6.2 Future Work

As usual there are a lot of open ends to explore. Traditionally allocation was treated non-deterministically and so far most of works were done based on this approach. Now it would be interesting to apply what is currently done to our approach, like extending it to more expressive languages e.g. with procedures or shared-variable concurrency, or proving completeness of this logic. We also thought about deterministic allocation for more than one cell and found that it is still possible to find suitable permutation function, but the problem is that permutation does not preserve consequetive memory blocks, and this is left for future. Reynolds in [12] said that there is a hope to construct

a garbage collector in a situation where addresses are disjoint from integers, which is ours, although we didn't think about this yet, since allocation always chooses a minimal address we have an implicit memory reuse.

Bibliography

[1] Nick Benton, Noah Torp-Smith. Abstracting allocation: The new new thing. *SPACE*, strony 108–110, 2006.

[2] Cees Pierik Frank S. de Boer. How to cook a complete hoare logic for your pet oo language.

[3] Sidney L. Hantler, James C. King. An introduction to proving the correctness of programs. *ACM Comput. Surv.*, 8(3):331–353, 1976.

[4] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 26(1):53–56, 1983.

[5] Neel Krishnaswami. Separation logic for a higher-order typed language., 2006.

[6] Michael J.C. Gordon Magnus O. Myreen, Antony C. J. Fox. Hoare logic for arm machine code, 2006.

[7] P. O'Hearn, D. Pym. The logic of bunched implications, 1999.

[8] Peter O'Hearn, John Reynolds, Hongseok Yang. Local reasoning about programs that alter data structures. *Lecture Notes in Computer Science*, 2142:1–??, 2001.

[9] Peter W. O'Hearn. Resources, concurrency and local reasoning.

[10] Ernst-Rdiger Olderog. A characterization of hoare's logic for programs with pascal-like procedures, 1983.

[11] Bernhard Reus, Jan Schwinghammer. Separation logic for higher-order store. *Computer Science Logic (CSL'06)*, Lecture Notes in Computer Science. Springer, 2006.

[12] J. Reynolds. Separation logic: a logic for shared mutable data structures, 2002.

[13] Noah Torp-Smith. Proving correctness of a garbage collector via local reasoning marking algorithm, 2003.

[14] David von Oheimb. Hoare logic for java in isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.

[15] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA, 1993.

[16] H. Yang. An example of local reasoning in bi pointer logic: the schorr-waite graph marking algorithm, 2000.

[17] Hongseok Yang, Peter W. O'Hearn. A semantic basis for local reasoning. *Foundations of Software Science and Computation Structure*, strony 402–416, 2002.