ISSN 1346-5597

National Institute of Informatics

NII Technical Report

Completeness of Pointer Program Verification by Separation Logic

Makoto Tatsuta, Wei-Ngan Chin, and Mahmudul Faisal Al Ameen

NII-2009-013E
June 2009

URL: http://www.nii.ac.jp/ Tokyo 101-8430, JAPAN +81-3-4212-2000 FAX: +81-3-3556-1916

Completeness of Pointer Program Verification by Separation Logic

Makoto Tatsuta
National Institute of Informatics
2-1-2 Hitotsubashi, 101-8430 Tokyo, Japan
tatsuta@nii.ac.jp

Wei-Ngan Chin
Department of Computer Science
National University of Singapore
chinwn@comp.nus.edu.sg

Mahmudul Faisal Al Ameen
Department of Informatics
Graduate University for Advanced Studies
2-1-2 Hitotsubashi, 101-8430 Tokyo, Japan
alameen@nii.ac.jp

Abstract

Reynolds' separation logical system for pointer program verification is investigated. This paper proves its completeness theorem as well as the expressiveness theorem of Peano arithmetic language for the system under the standard interpretation. This paper also introduces the predicate that represents the next new cell, and proves the completeness and the soundness of the extended system under deterministic semantics.

1 Introduction

Program verification for while programs has been intensively studied [1]. However, pointer program verification has not been fully studied, since there was difficulty to design an appropriate logical system for asserted pointer programs. Reynolds gave a breakthrough for it by using separation logic [10] and proposed a new logical system for pointer program verification. It enables us concise specification of program properties and a manageable proof system. Separation logic is successful in a theoretical sense as well as a practical sense. By using separation logic, some pointer program verification systems have been implemented [8, 2]. For example, the system in [8] automatically proved the correctness of a pointer version of the quick sort program in a second.

One of the most important theoretical questions for a verification system is its completeness. The soundness of a system guarantees that if the correctness of a program is proved in the system, then the program will indeed run correctly. The soundness of those existing practical systems has been proved. However, it does not mean the system can prove all correct programs are correct, that is, there is a possibility that some programs are not proved to be correct by the system even though they are indeed correct. The completeness is the converse of the soundness. The completeness of the system guarantees that if a program runs correctly, then the system surely proves the program is correct. The completeness of the system shows how powerful the system is. If the completeness for the system is proved, we do not have to worry about theoretical power of the system. If we have the completeness for the core of the system with some restrictions, we know what kind of limitation this system has, and how to use and improve the system.

Our contributions are: (1) the completeness theorem of separation logic for the first time, (2) the expressiveness theorem of Peano arithmetic for the separation logic, and (3) the predicate that represents the next new cell for completeness under deterministic semantics.

We will prove completeness by extending the original completeness results for while programs [1] to pointer programs and separation logic. We choose backwards reasoning in Reynolds' system for logical rules. Main difficulty is proving the expressiveness theorem. We will also prove completeness for deterministic semantics as well.

The expressiveness theorem says that Peano arithmetic with the standard interpretation is expressive for the separation logic. That is, the weakest precondition of every program is definable in Peano-arithmetical language. This result is obtained by coding the heap information as well as the store information by natural numbers, and simulating program executions as well as the truth of assertions by using Peano arithmetic. At first sight, the expressiveness may look trivial, but it is indeed a subtle problem and some pathological counterexamples are known [3]. We can have natural numbers n and m that encode the given store s and heap s, respectively. We will construct the formula Heap(m) that exactly specifies the current heap

so that Heap(m) is true at the current state (s, h) if and only if the number m encodes the heap h. We can construct a Peano-arithmetical formula HEval$_A(m)$ for a separation logic assertion A such that A is true at (s, h) if and only if HEval$_A(m)$ is true at s when the number m represents the heap h. For program execution, we will have a Peano-arithmetical formula Exec$_{P,\overrightarrow{x}}(n_1, n_2)$ for a given program P and its variables $\overrightarrow{x}$ such that the execution of P at the state r_1 terminates with the state r_2 if and only if Exec$_{P,\overrightarrow{x}}(n_1, n_2)$ is true where the numbers n_1 and n_2 represent the states r_1 and r_2 respectively. Combining Heap(m), HEval$_A(m)$, and Exec$_{P,\overrightarrow{x}}(n_1, n_2)$, we will define the weakest precondition for each given program P and assertion A. Though this result relies on heavy number coding of programs and assertions, we will present detailed definitions in order to provide a fundamental framework for completeness results to support future systems with a wider class of programs and assertions.

Our completeness theorem only shows relative completeness. That is, it assumes all true assertions are available in the system. This is a best possible completeness for pointer program verification for a similar reason to that for while program verification discussed in [4].

Parts of a completeness proof have been discussed in [6]. They showed the precondition of their backwards axiom for every atomic statement gives the weakest precondition for the statement. However, they only proved completeness for programs without if-statements nor while-statements, and they did not prove the completeness theorem itself.

When we take deterministic semantics instead of nondeterministic semantics, the axiom for allocation in [10] becomes insufficient for completeness. In order to have a complete system for deterministic semantics, we introduce the predicate New(e) that represents the new cell consumed by the next allocation, and strengthen the axiom by using this predicate. Under deterministic semantics, we will show the completeness theorem of the resulting system as well as its soundness theorem.

Our long-term aim is proving completeness of the core of existing practical verification systems for pointer programs. This paper will give the first step for this purpose. Since our system in this paper is simple and general, our completeness theorem can be applied to those systems in order to show the completeness of their core systems. This paper will also provide a starting point for completeness theorems in extensions with richer programming languages and assertion languages such as recursive procedure calls and inductive definitions.

Section 2 defines our programming language and our assertion language. Their semantics is given in Section 3. Section 4 gives a logical system for proving asserted programs, and Section 5 shows our completeness theorem as well as our soundness theorem. Section 6 proves the expressiveness theorem. Section 7 discusses applying our results to the verification system implemented in [8]. Section 8 gives the conclusion.

2 Languages

This section defines our programming language and our assertion language. Our language is obtained from Reynolds' paper [10].

We first define our base language, which will be used later for both our programming language and our assertion language. It is essentially a first-order language for Peano arithmetic. We call its formula a base formula. We will use i, j, k, l, m, n for natural numbers. Our base language is defined as follows:

Variables $x, y, z, w, \ldots$.

Constants $c ::= 0, 1, \mathrm{null}$.

Function symbol $+, \times$.

We have no propositional constants.

Predicate symbols $p ::= =, <$.

Terms $t ::= x | c | f(t, \ldots, t)$.

Base formulas $A ::= p(t, \ldots, t) | \neg A | A \wedge A | A \vee A | A \to A | \forall x A | \exists x A$.

$p(t_1, \ldots, t_n)$ means the predicate p holds for terms $t_1, \ldots, t_n$. The other formula constructions mean usual logical connectives. We will sometimes write the number n to denote the term $1 + (1 + (1 + \ldots (1 + 0)))$ (n times of $1+$).

Our programming language is an extension of while programs to pointers. It is the same as that of Reynolds [10]. Its expressions are terms of the base language. That is,

Expressions $e ::= x | 0 | 1 | \mathrm{null} | e + e | e \times e$.

Expressions mean natural numbers or pointers. null means the null pointer.

Its boolean expressions are quantifier-free base formulas. That is,

Boolean expressions $b ::= e = e | e < e | \neg b | b \wedge b | b \vee b | b \to b$.

Boolean expressions are used as conditions in a program.

Programs are defined by:

Programs $P ::= x := e | \mathrm{if}\ (b)\ \mathrm{then}\ (P)\ \mathrm{else}\ (P) |$
$\mathrm{while}\ (b)\ \mathrm{do}\ (P) | P; P |$
$x := \mathrm{cons}(e, e) | x := [e] | [e] := e | \mathrm{dispose}(e)$.

The statement $x = \mathrm{cons}(e_1, e_2)$ allocates two new consecutive memory cells, put e_1 and e_2 in the cells, and put the address into x. The statement $x := [e]$ looks up the content of the memory cell at the address e and put it into x. The statement $[e_1] := e_2$ changes the content of the memory cell at the address e_1 by e_2. The statement dispose(e) deallocates the memory cell at the address e.

Our assertion language is a first-order language extended by the separating conjunction $*$ and the separating implication $-\!\!*$. We will sometimes call its formula an assertion. Its variables, constants, function symbols, and terms are the same as those of the base language. Our assertion language is defined as follows:

Predicate symbols $=, <, \mapsto$.

Propositional constant emp.

Formulas $A ::= \mathrm{emp} | e = e | e < e | e \mapsto e | \neg A | A \wedge A | A \vee A | A \to A | \forall x A | \exists x A | A * A | A -\!\!* A$.

emp means the current heap is empty. $e_1 \mapsto e_2$ means the current heap has only one cell at the address e_1 and its content is e_2. $A * B$ means the current heap can be split into some two disjoint heaps such that A holds at one heap and B holds at the other heap. $A \mathrel{-\!\!*} B$ means that for any heap disjoint from the current heap such that A holds at the heap, B holds at the new heap obtained from the current heap and the heap by combining them.

FV(A) is defined as the set of free variables in A. FV(e) and FV(P) are similarly defined. FV($O_1, \ldots, O_n$) is defined as FV(O_1) $\cup \ldots \cup$ FV(O_n) when O_i is a formula, an expression, or a program.

We use vector notation to denote a sequence. For example, $\overrightarrow{e}$ denotes the sequence $e_1, \ldots, e_n$ of expressions.

3 Semantics

The semantics of our programming language and our assertion language is defined in this section. Our semantics is the same as that in Reynolds' paper [10] except the following simplification: (1) values are natural numbers, (2) addresses are non-zero natural numbers, and (3) null is 0.

The set N is defined as the set of natural numbers. The set Vars is defined as the set of variables in the base language. The set Locs is defined as the set $\{n \in N | n > 0\}$.

For sets S_1, S_2, $f : S_1 \to S_2$ means that f is a map from S_1 to S_2. $f : S_1 \to_{fin} S_2$ means that f is a finite map from S_1 to S_2, that is, there is a finite subset S_1' of S_1 and $f : S_1' \to S_2$. Dom(f) denotes the domain of the map f. $p(S)$ denotes the powerset of the set S. For a set $S \subseteq A$ and a map $f : A \to B$, we define $f(S) = \{f(x) | x \in S\}$. For a map $f : A \to B$ and a subset $C \subseteq A$, the map $f|_C : C \to B$ is defined by $f|_C(x) = f(x)$ for $x \in C$.

A store is defined as a map from Vars $\to N$, and denoted by s. A heap is defined as a finite map from Locs $\to_{fin} N$, and denoted by h. A value is a natural number. An address is a positive natural number. The null pointer is 0. A store assigns a value to each variable. A heap assigns a value to an address in its finite domain.

The store $s[x_1 := n_1, \ldots, x_k := n_k]$ is defined by s' such that $s'(x_i) = n_i$ and $s'(y) = s(y)$ for $y \notin \{x_1, \ldots, x_k\}$. The heap $h[m_1 := n_1, \ldots, m_k := n_k]$ is defined by h' such that $h'(m_i) = n_i$ and $h'(y) = h(y)$ for $y \in$ Dom(h) $- \{m_1, \ldots, m_k\}$. The store $s[x_1 := n_1, \ldots, x_k := n_k]$ is the same as s except values for the variables $x_1, \ldots, x_n$. The heap $h[m_1 := n_1, \ldots, m_k := n_k]$ is the same as h except the contents of the memory cells at the addresses $m_1, \ldots, m_k$.

We will write $h = h_1 + h_2$ when Dom(h) = Dom(h_1) $\cup$ Dom(h_2), Dom(h_1) $\cap$ Dom(h_2) $= \phi$, $h(x) = h_1(x)$ for $x \in$ Dom(h_1), and $h(x) = h_2(x)$ for $x \in$ Dom(h_2). The heap h is divided into the two disjoint heaps h_1 and h_2 when $h = h_1 + h_2$.

A state is defined as (s, h). The set States is defined as the set of states. The state for pointer program is specified by the store and the heap, since pointer programs manipulate memory heaps as well as variable assignments.

Definition 3.1 We define the semantics of our base language by the standard model of natural numbers and $[\![\text{null}]\!] = 0$. That is, we suppose $[\![0]\!] = 0$, $[\![1]\!] = 1$, $[\![+]\!] = +$, $[\![\times]\!] = \times$, $[\![=]\!] = (=)$, and $[\![<]\!] = (<)$. For a store s, an expression e, and a base formula A, according to the interpretation of a first-order language, the meaning $[\![e]\!]_s$ is defined as a natural number and the meaning $[\![A]\!]_s$ is defined as true or false. $[\![e]\!]_s$ and $[\![A]\!]_s$ are the value of e under the store s, and the truth value of A under the store s, respectively.

Definition 3.2 We define the semantics of our programming language. For a program P, its meaning $[\![P]\!]$ is defined as a map from States $\cup \{\text{abort}\}$ to $p(\text{States} \cup \{\text{abort}\})$. We will define $[\![P]\!](r_1)$ as the set of all the possible resulting states after the execution of P with the initial state r_1 terminates. In particular, if the execution of P with the initial state r_1 does not terminate, we will define $[\![P]\!](r_1)$ as the empty set. $[\![P]\!]$ is defined by induction on P as the smallest set satisfying the following:

$[\![P]\!](\text{abort}) = \{\text{abort}\}$,
$[\![x := e]\!]((s,h)) = \{(s[x := [\![e]\!]_s], h)\}$,
$[\![\text{if } (b) \text{ then } (P_1) \text{ else } (P_2)]\!]((s,h)) =$
  $[\![P_1]\!]((s,h))$ if $[\![b]\!]_s = \text{true}$,
  $[\![P_2]\!]((s,h))$ otherwise,
$[\![\text{while } (b) \text{ do } (P)]\!]((s,h)) = \{(s,h)\}$ if $[\![b]\!]_s = \text{false}$,
  $[\![\text{while } (b) \text{ do } (P)]\!]([\![P]\!]((s,h)))$ otherwise,
$[\![P_1; P_2]\!]((s,h)) = [\![P_2]\!]([\![P_1]\!]((s,h)))$,
$[\![x := \text{cons}(e_1, e_2)]\!]((s,h)) =$
  $\{(s[x := n], h[n := [\![e_1]\!]_s, n+1 := [\![e_2]\!]_s]) |$
  $n > 0, n, n+1 \notin \text{Dom}(h)\}$,
$[\![x := [e]]\!]((s,h)) =$
  $\{(s[x := h([\![e]\!]_s)], h)\}$ if $[\![e]\!]_s \in \text{Dom}(h)$,
  $\{\text{abort}\}$ otherwise,
$[\![[e_1] := e_2]\!]((s,h)) =$
  $\{(s, h[[\![e_1]\!]_s := [\![e_2]\!]_s])\}$ if $[\![e_1]\!]_s \in \text{Dom}(h)$,
  $\{\text{abort}\}$ otherwise,
$[\![\text{dispose}(e)]\!]((s,h)) =$
  $\{(s, h|_{\text{Dom}(h) - \{[\![e]\!]_s\}})\}$ if $[\![e]\!]_s \in \text{Dom}(h)$,
  $\{\text{abort}\}$ otherwise.

Definition 3.3 We define the semantics of the assertion language. For an assertion A and a state (s, h), the meaning $[\![A]\!]_{(s,h)}$ is defined as true or false. $[\![A]\!]_{(s,h)}$ is the truth value of A at the state (s, h). $[\![A]\!]_{(s,h)}$ is defined by induc-

tion on A as follows:

$[\![\text{emp}]\!]_{(s,h)} = \text{true}$ if $\text{Dom}(h) = \phi$,
$[\![e_1 = e_2]\!]_{(s,h)} = ([\![e_1]\!]_s = [\![e_2]\!]_s)$,
$[\![e_1 < e_2]\!]_{(s,h)} = ([\![e_1]\!]_s < [\![e_2]\!]_s)$,
$[\![e_1 \mapsto e_2]\!]_{(s,h)} = \text{true}$
if $\text{Dom}(h) = \{[\![e_1]\!]_s\}, h([\![e_1]\!]_s) = [\![e_2]\!]_s$,
$[\![\neg A]\!]_{(s,h)} = (\text{not } [\![A]\!]_{(s,h)})$,
$[\![A \wedge B]\!]_{(s,h)} = ([\![A]\!]_{(s,h)} \text{ and } [\![B]\!]_{(s,h)})$,
$[\![A \vee B]\!]_{(s,h)} = ([\![A]\!]_{(s,h)} \text{ or } [\![B]\!]_{(s,h)})$,
$[\![A \to B]\!]_{(s,h)} = ([\![A]\!]_{(s,h)} \text{ implies } [\![B]\!]_{(s,h)})$,
$[\![\forall x A]\!]_{(s,h)} = \text{true}$
if $[\![A]\!]_{(s[x:=n],h)} = \text{true}$ for all $n \in N$,
$[\![\exists x A]\!]_{(s,h)} = \text{true}$
if $[\![A]\!]_{(s[x:=n],h)} = \text{true}$ for some $n \in N$,
$[\![A * B]\!]_{(s,h)} = \text{true}$ if $h = h_1 + h_2$,
$[\![A]\!]_{(s,h_1)} = [\![B]\!]_{(s,h_2)} = \text{true}$ for some h_1, h_2,
$[\![A \mathrel{-\!\!*} B]\!]_{(s,h)} = \text{true}$ if $h_2 = h_1 + h$ and
$[\![A]\!]_{(s,h_1)} = \text{true}$ imply $[\![B]\!]_{(s,h_2)} = \text{true}$
for all h_1, h_2.

We say A is true when if $[\![A]\!]_{(s,h)} = \text{true}$ for all (s, h).

Definition 3.4 For an asserted program $\{A\}P\{B\}$, its meaning $[\![\{A\}P\{B\}]\!]$ is defined as true or false. $[\![\{A\}P\{B\}]\!]$ is defined to be true if the following hold.

(1) for all (s, h), if $[\![A]\!]_{(s,h)} = \text{true}$, then $[\![P]\!]((s, h)) \neq \text{abort}$.

(2) for all (s, h) and (s', h'), if $[\![A]\!]_{(s,h)} = \text{true}$ and $[\![P]\!]((s, h)) \ni (s', h')$, then $[\![B]\!]((s', h')) = \text{true}$.

We say $\{A\}P\{B\}$ is true when $[\![\{A\}P\{B\}]\!] = \text{true}$. $\{A\}P\{B\}$ means abort-free partial correctness. $\{A\}P\{B\}$ implies partial correctness. It also implies that the execution of the program P with the initial state that satisfies A never aborts, that is, P does not access to any unallocated addresses during the execution.

4 Logical System

This section defines our logical system. It is the same as Reynolds' system presented in [10].

We will write $A[x := e]$ for the formula obtained from A by replacing x by e. We will write the formula $e \mapsto e_1, e_2$ to denote $(e \mapsto e_1) * (e + 1 \mapsto e_2)$.

Definition 4.1 Our logical system is defined by the following inference rules.

$$\frac{}{\{A[x := e]\}x := e\{A\}} \ (assignment)$$

$$\frac{\{A \wedge b\}P_1\{B\} \quad \{A \wedge \neg b\}P_2\{B\}}{\{A\}\text{if } (b) \text{ then } (P_1) \text{ else } (P_2)\{B\}} \ (if)$$

$$\frac{\{A \wedge b\}P\{A\}}{\{A\}\text{while } (b) \text{ do } (P)\{A \wedge \neg b\}} \ (while)$$

$$\frac{\{A\}P_1\{C\} \quad \{C\}P_2\{B\}}{\{A\}P_1; P_2\{B\}} \ (comp)$$

$$\frac{\{A_1\}P\{B_1\}}{\{A\}P\{B\}} \ (conseq) \quad (A \to A_1,\ B_1 \to B \text{ true})$$

$$\frac{}{\{\forall x'((x' \mapsto e_1, e_2) \mathrel{-\!\!*} A[x := x'])\}} \ (cons)$$
$$x := \text{cons}(e_1, e_2)\{A\} \quad (x' \notin \text{FV}(e_1, e_2, A))$$

$$\frac{}{\{\exists x'(e \mapsto x' * (e \mapsto x' \mathrel{-\!\!*} A[x := x']))\}} \ (lookup)$$
$$x := [e]\{A\} \quad (x' \notin \text{FV}(e, A))$$

$$\frac{}{\{(\exists x(e_1 \mapsto x)) * (e_1 \mapsto e_2 \mathrel{-\!\!*} A)\}} \ (mutation)$$
$$[e_1] := e_2\{A\} \quad (x \notin \text{FV}(e_1))$$

$$\frac{}{\{(\exists x(e \mapsto x)) * A\}\text{dispose}(e)\{A\}} \ (dispose)$$
$$(x \notin \text{FV}(e))$$

We say $\{A\}P\{B\}$ is provable and we write $\vdash \{A\}P\{B\}$, when $\{A\}P\{B\}$ can be derived by these inference rules.

We explain inference rules by the example of $\{\text{emp}\}x := \text{cons}(0, 0); [x + 1] := 3; \text{dispose}(x)\{x + 1 \mapsto 3\}$. This means that starting from the empty heap, if we allocate two cells at x, we put 3 in the cell at $x + 1$, and we deallocate the cell at x, then the resulting heap will have only one memory cell at $x + 1$ with its content 3. It is derived by using inference rules as follows. The rule $(cons)$ gives $\{\forall x'(x' \mapsto 0, 0 \mathrel{-\!\!*} x' \mapsto 0, 0)\}x := \text{cons}(0, 0)\{x \mapsto 0, 0\}$. Since $\text{emp} \to \forall x'(x' \mapsto 0, 0 \mathrel{-\!\!*} x' \mapsto 0, 0)$ is true, by the rule $(conseq)$, $\{\text{emp}\}x := \text{cons}(0, 0)\{x \mapsto 0, 0\}$ is provable. The rule $(mutation)$ gives $\{(\exists x'(x + 1 \mapsto x')) * (x + 1 \mapsto 3 \mathrel{-\!\!*} x \mapsto 0, 3)\}[x + 1] := 3\{x \mapsto 0, 3\}$. Since $x \mapsto 0, 0 \to ((\exists x'(x + 1 \mapsto x')) * (x + 1 \mapsto 3 \mathrel{-\!\!*} x \mapsto 0, 3))$ is true, by the rule $(conseq)$, $\{x \mapsto 0, 0\}[x+1] := 3\{x \mapsto 0, 3\}$ is provable. The rule $(dispose)$ gives $\{(\exists x'(x \mapsto x')) * x + 1 \mapsto 3\}\text{dispose}(x)\{x + 1 \mapsto 3\}$. Since $x \mapsto 0, 3 \to (\exists x'(x \mapsto x')) * x + 1 \mapsto 3$ is true, by the rule $(conseq)$, $\{x \mapsto 0, 3\}\text{dispose}(x)\{x + 1 \mapsto 3\}$ is provable. By using the rule $(comp)$ twice for $\{\text{emp}\}x := \text{cons}(0, 0)\{x \mapsto 0, 0\}$, $\{x \mapsto 0, 0\}[x + 1] := 3\{x \mapsto 0, 3\}$, and $\{x \mapsto 0, 3\}\text{dispose}(e)\{x + 1 \mapsto 3\}$, we finish showing $\{\text{emp}\}x := \text{cons}(0, 0); [x+1] := 3; \text{dispose}(x)\{x+1 \mapsto 3\}$ is provable.

5 Soundness and Completeness Theorems

Our main results are the following theorems. Our completeness theorem is new. For our soundness theorem we give a whole proof in the case study of our system along general ideas already discussed in [10].

Theorem 5.1 (Soundness) *If $\{A\}P\{B\}$ is provable, then $\{A\}P\{B\}$ is true.*

Theorem 5.2 (Completeness) *If $\{A\}P\{B\}$ is true, then $\{A\}P\{B\}$ is provable.*

Their proofs are given in the appendix. We only sketch the proofs here.

The soundness theorem is proved by induction on the given proof of $\{A\}P\{B\}$. Intuitively, we will show each inference rule preserves the truth.

For example, we discuss the rule $(comp)$. The rule is

$$\frac{\{A\}P_1\{C\} \quad \{C\}P_2\{B\}}{\{A\}P_1; P_2\{B\}}\ (comp)$$

If we know $\{A\}P_1\{C\}$ and $\{C\}P_2\{B\}$ are both true, we know $\{A\}P_1; P_2\{B\}$ is true, since the truth of $\{A\}P_1; P_2\{B\}$ denotes that from the state where A is true, if the execution of $P_1; P_2$ reaches some state, then B is true at the resulting state, and in the execution of $P_1; P_2$ we have some intermediate state after the execution of P_1 where C is true, with which we can use the assumptions for $\{A\}P_1\{C\}$ and $\{C\}P_2\{B\}$.

For another example, we discuss the $(mutation)$ rule. It is sufficient to show $\{(\exists x(e_1 \mapsto x)) * (e_1 \mapsto e_2 \mathrel{-\!\!*} A)\}[e_1] := e_2\{A\}$ is true. We assume $(\exists x(e_1 \mapsto x)) * (e_1 \mapsto e_2 \mathrel{-\!\!*} A)$ is true at the initial state (s, h) and the execution of $[e_1] := e_2$ reaches the final state (s', h'). We have to show A is true at (s', h'). By the first assumption, we know the heap h is divided into the disjoint heaps h_1 and h_2 such that $h = h_1 + h_2$, $\exists x(e_1 \mapsto x)$ is true at (s, h_1), and $e_1 \mapsto e_2 \mathrel{-\!\!*} A$ is true at (s, h_2). That is, the heap h is split into the first part h_1 which contains only one memory cell at the address e_1 and the second part. By the program execution, the first part is changed so that the content of the cell becomes e_2. Since the second assumption says $e_1 \mapsto e_2 \mathrel{-\!\!*} A$ is true at the second part of the heap, A is true at the resulting heap after the program execution.

Once we prove every rule preserves the truth, we know the conclusion of a given proof is true by applying it to each inference in the proof.

The completeness theorem is proved by induction on the program P. The goal is showing a given true asserted program is provable. Intuitively, we will reduce this goal to subgoals for smaller pieces of the given program that state true asserted subprograms of the given program are provable. If we show that for each program construction a true asserted program is provable by using the assumption that all the asserted subprograms are provable, we can say any given true asserted program is provable. The proof will use the expressiveness of our language, which is proved as Theorem 6.12 in the next section.

For example, we discuss the rule $(comp)$. Suppose $\{A\}P_1; P_2\{B\}$ is true. We have to construct a proof of $\{A\}P_1; P_2\{B\}$. In order to do that, we have to find some assertion C such that $\{A\}P_1\{C\}$ is true and $\{C\}P_2\{B\}$ is true. If we find the assertion C, since P_1 and P_2 are smaller pieces of the given program $P_1; P_2$, we can suppose $\{A\}P_1\{C\}$ and $\{C\}P_2\{B\}$ are both provable, and by the rule $(comp)$, we have a proof of $\{A\}P_1; P_2\{B\}$. In order to find the assertion C, we will use the expressiveness given by Theorem 6.12, to take the weakest precondition $\mathrm{W}_{P_2,B}(\overrightarrow{x})$ for P_2 and B as the assertion C.

The expressiveness theorem will also be necessary for the case for $\{A\}$while (b) do $(P)\{B\}$ in order to find the intermediate assertion C such that $A \to C$, $\{C \wedge b\}P\{C\}$, and $C \wedge \neg b \to B$ are true.

For another example, we discuss the rule $(mutation)$. Suppose $\{A\}[e_1] := e_2\{B\}$ is true. We have to construct a proof of $\{A\}[e_1] := e_2\{B\}$. For this purpose, it is sufficient to show $A \to ((\exists x(e_1 \mapsto x)) * (e_1 \mapsto e_2 \mathrel{-\!\!*} B))$ is true, since we have the axiom $\{(\exists x(e_1 \mapsto x)) * (e_1 \mapsto e_2 \mathrel{-\!\!*} B)\}[e_1] := e_2\{B\}$ and by the rule $(conseq)$ with $A \to ((\exists x(e_1 \mapsto x)) * (e_1 \mapsto e_2 \mathrel{-\!\!*} B))$, we have a proof of $\{A\}[e_1] := e_2\{B\}$. In order to show $A \to ((\exists x(e_1 \mapsto x)) * (e_1 \mapsto e_2 \mathrel{-\!\!*} B))$ is true, we assume A is true at (s, h) and we will show $(\exists x(e_1 \mapsto x)) * (e_1 \mapsto e_2 \mathrel{-\!\!*} B)$ is true at (s, h). Since $\{A\}[e_1] := e_2\{B\}$ is true, the execution of the statement $[e_1] := e_2$ from (s, h) does not return the abort. Hence h contains the memory cell at the address e_1. Since B is true at the heap after the content of the cell at the address e_1 is changed to e_2, the assertion $(e_1 \mapsto e_2 \mathrel{-\!\!*} B)$ is true for the heap obtained from h by removing the cell at the address e_1. Hence $(\exists x(e_1 \mapsto x)) * (e_1 \mapsto e_2 \mathrel{-\!\!*} B)$ is true at (s, h).

6 Expressiveness Theorem

This section proves the expressiveness theorem. We will first define the base formulas $\mathrm{Store}_{\overrightarrow{x}}(m)$ and $\mathrm{Heap}(m)$ for describing the current store and the current heap respectively. Next we will provide the base formulas $\mathrm{EEval}_{e,\overrightarrow{x}}(n, k)$ and $\mathrm{BEval}_{A,\overrightarrow{x}}(n)$, which express the meaning of the expression e and the base formula A respectively. Then we will define the base formula $\mathrm{HEval}_A(m)$ for expressing the meaning of the assertion A at the heap by m. By using it, we will define the base formula $\mathrm{Eval}_{A,\overrightarrow{x}}(n, m)$, which expresses the meaning of the assertion A. We will also define the base formula $\mathrm{Exec}_{P,\overrightarrow{x}}(n, m)$ for the meaning of the program P. Finally we will define the base formula $W_{P,A}(\overrightarrow{x})$ for the weakest precondition of the program P and the assertion A, and we will prove the expressiveness theorem that states $W_{P,A}(\overrightarrow{x})$ indeed expresses the weakest precondition.

We assume a standard surjective pairing function on natural numbers. For natural numbers n, m, we will write (n, m) to denote the number that represents the pair of n and m. We also assume a standard surjective coding of a sequence of natural numbers by a natural number. We will write $\langle n_1, \ldots, n_k \rangle$ for the number that represents the sequence $n_1, \ldots, n_k$. When the number n represents a sequence, $\mathrm{Lh}(n)$ and $(n)_i$ denote the length of the sequence and the i-th element of the sequence respectively.

The following predicates for handling sequences are known to be definable in the language of Peano arithmetic. $\mathrm{Pair}(k,n,m)$ is defined to hold if k is the number that represents the pair of n and m. $\mathrm{Lh}(n,k)$ is defined to hold if k is the length of the sequence represented by n. That is, $\mathrm{Lh}(\langle n_1,\ldots,n_k\rangle,k)$ holds. $\mathrm{Elem}(n,i,k)$ is defined to hold if k is the i-th element in the sequence represented by n. That is, $\mathrm{Elem}(\langle n_1,\ldots,n_k\rangle,i-1,n_i)$ holds. Note that i ranges over $\{0,1,\ldots,k-1\}$.

We code the piece of the store s for variables $x_1,\ldots,x_k$ by the number $\langle n_1,\ldots,n_k\rangle$ where $s(x_i)=n_i$. We code the heap h by the number $\langle m_1,\ldots,m_k\rangle$ where $\mathrm{Dom}(h)=\{n_1,\ldots,n_k\}$ and m_i is the number that represents the pair of n_i and $h(n_i)$. We code the result of a program execution by coding abort and (s,h) by 0 and $k+1$ respectively where the piece of s is coded by n, h is coded by m, and k is the pair of numbers n and m. We allow doubled elements and elements for the address 0 in a sequence for simplicity when we use the sequence to represent a heap. For example, $\langle(0,5),(1,3),(1,3)\rangle$ as well as $\langle(1,3)\rangle$ represents the heap h where $\mathrm{Dom}(h)=\{1\}$ and $h(1)=3$.

We say A is true at (s,h) when $[\![A]\!]_{(s,h)}=\mathrm{true}$. $A\leftrightarrow B$ is defined as $(A\rightarrow B)\wedge(B\rightarrow A)$.

0 sometimes denotes the store that maps every variable to 0, that is, $0(x)=0$ for all variables x. ϕ sometimes denotes the empty heap, that is, $\phi(x)$ is undefined for all $x\in N$.

We define the following base formulas.

$$\begin{aligned}&\mathrm{Lesslh}(i,n)=\exists x(\mathrm{Lh}(n,x)\wedge i<x),\\&\mathrm{Addseq}(k,n,m)=\exists x(\mathrm{Lh}(n,x)\wedge\mathrm{Lh}(m,x+1))\wedge\\&\quad\mathrm{Elem}(m,0,k)\wedge\\&\quad\forall yx(\mathrm{Lesslh}(y,n)\wedge\mathrm{Elem}(n,y,x)\\&\quad\rightarrow\mathrm{Elem}(m,y+1,x)).\end{aligned}$$

$\mathrm{Lesslh}(i,n)$ means $i<lh(n)$. $\mathrm{Addseq}(k,n,m)$ means $\langle k\rangle\cdot n=m$ where $\cdot$ denotes the concatenation of sequences.

Definition 6.1 We define the base formulas $\mathrm{Store}_{x_1,\ldots,x_n}(m)$ and $\mathrm{Heap}(m)$.

$$\begin{aligned}&\mathrm{Store}_{x_1,\ldots,x_n}(m)=\mathrm{Lh}(m,n)\wedge\mathrm{Elem}(m,0,x_1)\wedge\ldots\\&\quad\wedge\mathrm{Elem}(m,n-1,x_n),\\&\mathrm{Lookup}(m,l,k)=\exists yz(\mathrm{Lesslh}(y,m)\wedge y\neq0\wedge\\&\quad\mathrm{Elem}(m,y,z)\wedge\mathrm{Pair}(z,l,k)),\\&\mathrm{Heap}(m)=\\&\quad\forall xy(\mathrm{Lookup}(m,x,y)\leftrightarrow(x\mapsto y*0=0)).\end{aligned}$$

$\mathrm{Store}_{x_1,\ldots,x_n}(\langle m_1,\ldots,m_n\rangle)$ means $s(x_i)=m_i$ where s is the current store. $\mathrm{Lookup}(m,l,k)$ means $h(l)=k$ where m represents the heap h. $\mathrm{Heap}(\langle(l_1,n_1),\ldots,(l_k,n_k)\rangle)$ means $\mathrm{Dom}(h)=\{l_1,\ldots,l_k\}$ and $h(l_i)=n_i$ where h is the current heap.

Definition 6.2 We define the base formulas $\mathrm{EEval}_{e,\overrightarrow{x}}(n,k)$ for the expression e and $BEval_{A,\overrightarrow{x}}(n)$ for the base formula A where we suppose $\overrightarrow{x}$ includes $\mathrm{FV}(e)$ and $\mathrm{FV}(A)$ respectively.

$$\begin{aligned}&\mathrm{EEval}_{e,\overrightarrow{x}}(n,k)=\exists\overrightarrow{x}(\mathrm{Store}_{\overrightarrow{x}}(n)\wedge e=k),\\&\mathrm{BEval}_{A,\overrightarrow{x}}(n)=\exists\overrightarrow{x}(\mathrm{Store}_{\overrightarrow{x}}(n)\wedge A).\end{aligned}$$

$\mathrm{EEval}_{e,\overrightarrow{x}}(n,k)$ means $[\![e]\!]_s=k$ where n represents the store s. $\mathrm{BEval}_{A,\overrightarrow{x}}(n)$ means $[\![A]\!]_s=\mathrm{true}$ where n represents the store s.

We define the following base formulas.

$$\begin{aligned}&\mathrm{Pair2}(z,x,y)=\exists w(z=w+1\wedge\mathrm{Pair}(w,x,y)),\\&\mathrm{Domain}(k,m)=\exists y\mathrm{Lookup}(m,k,y),\\&\mathrm{Separate}(m,m_1,m_2)=\forall x(\exists y(\mathrm{Elem}(m,y,x))\leftrightarrow\\&\quad\exists y(\mathrm{Elem}(m_1,y,x)\vee\mathrm{Elem}(m_2,y,x)))\wedge\\&\quad\forall x_1x_2y_1y_2(\mathrm{Lookup}(m_1,x_1,y_1)\wedge\\&\quad\mathrm{Lookup}(m_2,x_2,y_2)\rightarrow x_1\neq x_2).\end{aligned}$$

$\mathrm{Domain}(k,m)$ means $k\in Dom(h)$ where m represents the heap h. $\mathrm{Separate}(m,m_1,m_2)$ means $h=h_1+h_2$ where m, m_1, and m_2 represent the heaps h, h_1, and h_2 respectively.

Definition 6.3 We define the base formula $\mathrm{HEval}_A(x)$ for the assertion A by induction on A.

$$\begin{aligned}&\mathrm{HEval}_A(m)=A\qquad(A\text{ is a base formula}),\\&\mathrm{HEval}_{\mathsf{emp}}(m)=\neg\exists xy\mathrm{Lookup}(m,x,y),\\&\mathrm{HEval}_{e_1\mapsto e_2}(m)=e_1>0\wedge\\&\quad\forall xy(\mathrm{Lookup}(m,x,y)\leftrightarrow x=e_1\wedge y=e_2),\\&\mathrm{HEval}_{\neg A}(m)=\neg\mathrm{HEval}_A(m),\\&\mathrm{HEval}_{A\wedge B}(m)=\mathrm{HEval}_A(m)\wedge\mathrm{HEval}_B(m),\\&\mathrm{HEval}_{A\vee B}(m)=\mathrm{HEval}_A(m)\vee\mathrm{HEval}_B(m),\\&\mathrm{HEval}_{A\rightarrow B}(m)=\mathrm{HEval}_A(m)\rightarrow\mathrm{HEval}_B(m),\\&\mathrm{HEval}_{\forall xA}(m)=\forall x\mathrm{HEval}_A(m),\\&\mathrm{HEval}_{\exists xA}(m)=\exists x\mathrm{HEval}_A(m),\\&\mathrm{HEval}_{A*B}(m)=\exists y_1y_2(\mathrm{Separate}(m,y_1,y_2)\wedge\\&\quad\mathrm{HEval}_A(y_1)\wedge\mathrm{HEval}_B(y_2)),\\&\mathrm{HEval}_{A\text{—}\!*B}(m)=\forall y_1y_2(\mathrm{HEval}_A(y_2)\wedge\\&\quad\mathrm{Separate}(y_1,m,y_2)\rightarrow\mathrm{HEval}_B(y_1)).\end{aligned}$$

$\mathrm{HEval}_A(m)$ means $[\![A]\!]_{(s,h)}=\mathrm{true}$ where s is the current store and m represents the heap h.

Definition 6.4 We define the base formula $\mathrm{Eval}_{A,\overrightarrow{x}}(n,m)$ for the assertion A. We suppose $\overrightarrow{x}$ includes $\mathrm{FV}(A)$.

$$\mathrm{Eval}_{A,\overrightarrow{x}}(n,m)=\exists\overrightarrow{x}(\mathrm{Store}_{\overrightarrow{x}}(n)\wedge\mathrm{HEval}_A(m)).$$

$\mathrm{Eval}_{A,\overrightarrow{x}}(n,m)$ means $[\![A]\!]_{(s,h)}=\mathrm{true}$ where n represents the store s and m represents the heap h.

We define the following base formulas.

$$\begin{aligned}&\mathrm{New2}(n,m)=n>0\wedge\neg\mathrm{Domain}(n,m)\wedge\\&\quad\neg\mathrm{Domain}(n+1,m),\\&\mathrm{ChangeStore}_{x_0,\ldots,x_n,x_i}(m_1,k,m_2)=\\&\quad\mathrm{Lh}(m_1,n+1)\wedge\mathrm{Lh}(m_2,n+1)\wedge\\&\quad\forall yx(y<n+1\wedge y\neq i\wedge\mathrm{Elem}(m_1,y,x)\rightarrow\\&\quad\mathrm{Elem}(m_2,y,x))\wedge\mathrm{Elem}(m_2,i,k),\end{aligned}$$

$$\begin{aligned}&\text{ChangeHeap}(m_1,l,k,m_2) = \forall xy(x \neq l \rightarrow\\ &\quad (\text{Lookup}(m_1,x,y) \leftrightarrow \text{Lookup}(m_2,x,y)))\\ &\quad \wedge \text{Lookup}(m_2,l,k),\\ &\text{EqHeap}(m_1,m_2) =\\ &\quad \forall xy(\text{Lookup}(m_1,x,y) \leftrightarrow \text{Lookup}(m_2,x,y)).\end{aligned}$$

$\text{New2}(n,m)$ means n is the address of free cells in h where m represents the heap h. That is, the address n can be used by the next $x := \text{cons}(e_1,e_2)$ statement. $\text{ChangeStore}_{x_0,\ldots,x_n,x_i}(m_1,k,m_2)$ means m_2 represents the store $s[x_i := k]$ where m_1 represents the store s. $\text{ChangeHeap}(m_1,l,k,m_2)$ means m_2 represents the heap $h[l := k]$ where m_1 represents the heap h. $\text{EqHeap}(m_1,m_2)$ means m_1 and m_2 represent the same heap.

We define the following base formula.

$$\begin{aligned}&\text{EqResult}(n_1,n_2) = n_1 = n_2 \vee n_1 > 0 \wedge n_2 > 0 \wedge\\ &\quad \exists xy_1y_2(\text{Pair2}(n_1,x,y_1) \wedge \text{Pair2}(n_2,x,y_2)\\ &\quad \wedge \text{EqHeap}(y_1,y_2)).\end{aligned}$$

We say the number n represents the result r if $r = \text{abort}$ and $n = 0$ or $r = (s,h)$ and $n = (m,k)+1$ where m represents the store s and k represents the heap h. $\text{EqResult}(n_1,n_2)$ means n_1 and n_2 represent the same result.

Definition 6.5 We define the base formula $\text{Exec}_{P,\overrightarrow{x}}(n,m)$ by induction on the program P in Figure 1. $\text{Exec}_{P,\overrightarrow{x}}(n_1,n_2)$ means $[\![P]\!](r_1) \ni r_2$ where n_1 and n_2 represent r_1 and r_2 respectively and n_1 and n_2 contain store information for variables $\overrightarrow{x}$.

We define the following abbreviations. Note that they are not formulas.

$$\begin{aligned}&\text{Storecode}_{x_1,\ldots,x_n}(m,s) = \text{Lh}(m,n) \wedge\\ &\quad \forall i < n(\text{Elem}(m,i,s(x_{i+1}))),\\ &\text{Heapcode}(m,h) =\\ &\quad \forall ln(h(l) = n \leftrightarrow \text{Lookup}(m,l,n)),\\ &\text{Result}_{\overrightarrow{x}}(n,r) = n = 0 \wedge r = \text{abort} \vee\\ &\quad n > 0 \wedge \exists shyz(r = (s,h) \wedge \text{Pair2}(n,y,z) \wedge\\ &\quad \text{Storecode}_{\overrightarrow{x}}(y,s) \wedge \text{Heapcode}(z,h)).\end{aligned}$$

$\text{Storecode}_{x_1,\ldots,x_n}(m,s)$ means the number m is the code that represents the store s for variables $x_1,\ldots,x_n$. $\text{Heapcode}(m,h)$ means the number m is the code that represents the heap h. $\text{Result}_{\overrightarrow{x}}(n,r)$ means the number n represents the result r.

The next lemma shows that the base formulas $\text{EEval}_{e,\overrightarrow{x}}(n,k)$, $\text{BEval}_{A,\overrightarrow{x}}(n)$, $\text{HEval}_A(m)$, $\text{Eval}_{A,\overrightarrow{x}}(n,m)$, and $\text{Exec}_{P,\overrightarrow{x}}(n_1,n_2)$ actually have the meaning we explained above.

Lemma 6.6 *(1) $\text{EEval}_{e,\overrightarrow{x}}(n,k)$ is true if and only if $\exists s(\text{Storecode}_{\overrightarrow{x}}(n,s) \wedge [\![e]\!]_s = k)$.*

(2) $\text{BEval}_{A,\overrightarrow{x}}(n)$ is true if and only if $\exists s(\text{Storecode}_{\overrightarrow{x}}(n,s) \wedge [\![A]\!]_s = \text{true})$.

(3) $\text{Heapcode}(m,h) \rightarrow [\![\text{HEval}_A(m)]\!]_s = [\![A]\!]_{(s,h)}$.

(4) $\text{Eval}_{A,\overrightarrow{x}}(n,m)$ is true if and only if $\exists sh(\text{Storecode}_{\overrightarrow{x}}(n,s) \wedge \text{Heapcode}(m,h) \wedge [\![A]\!]_{(s,h)} = \text{true})$.

(5) $\text{Exec}_{P,\overrightarrow{x}}(n_1,n_2)$ is true if and only if $\exists r_1r_2(\text{Result}_{\overrightarrow{x}}(n_1,r_1) \wedge [\![P]\!](r_1) \ni r_2 \wedge \text{Result}_{\overrightarrow{x}}(n_2,r_2))$.

Its proof is given in the appendix. They can be straightforwardly proved by using their definitions.

Definition 6.7 For a program P and and an assertion A, the weakest precondition for P and A under the standard interpretation is defined as the set $\{(s,h) | \forall r([\![P]\!]((s,h)) \ni r \rightarrow r \neq \text{abort} \wedge [\![A]\!]_r = \text{true})\}$.

Definition 6.8 We define the base formula $\text{W}_{P,A}(\overrightarrow{x})$ for the program P and the assertion A. We fix some sequence $\overrightarrow{x}$ of the variables in $\text{FV}(P,A)$.

$$\begin{aligned}&\text{W}_{P,A}(\overrightarrow{x}) = \forall xyzw(\text{Store}_{\overrightarrow{x}}(x) \wedge \text{Heap}(y) \wedge\\ &\quad \text{Pair2}(z,x,y) \wedge \text{Exec}_{P,\overrightarrow{x}}(z,w) \rightarrow w > 0 \wedge\\ &\quad \exists y_1z_1(\text{Pair2}(w,y_1,z_1) \wedge \text{Eval}_{A,\overrightarrow{x}}(y_1,z_1))).\end{aligned}$$

$\text{W}_{P,A}(\overrightarrow{x})$ means the weakest precondition for P and A. That is, $\text{W}_{P,A}(\overrightarrow{x})$ gives the weakest assertion W such that $\{W\}P\{A\}$ is true. Note that all the free variables in $\text{W}_{P,A}(\overrightarrow{x})$ are $\overrightarrow{x}$ and they appear only in $\text{Store}_{\overrightarrow{x}}(x)$.

For a set V of variables, $s =_V s'$ is defined to hold if $s(x) = s'(x)$ for all $x \in V$. $(s,h) =_V (s',h')$ is defined to hold if $s =_V s'$ and $h = h'$.

The next lemma will be used in the proof of Lemma 6.10. It shows that the store information involved in the execution of P is only the information for the variables actually appearing in P.

Lemma 6.9 *Suppose $s =_{FV(P)} s'$.*

(1) If $[\![P]\!]((s,h)) \ni$ abort, then $[\![P]\!]((s',h)) \ni$ abort.

(2) If $[\![P]\!]((s,h)) \ni (s_1,h_1)$, then $[\![P]\!]((s',h)) \ni (s_1',h_1)$ where $s_1' =_{FV(P)} s_1$ and $s_1'(x) = s'(x)$ for $x \notin FV(P)$.

Proof. They are prove by induction on P. □.

The next lemma says that $\text{W}_{P,A}(\overrightarrow{x})$ indeed describes the weakest precondition for P and A.

Lemma 6.10 *(1) $\{\text{W}_{P,A}(\overrightarrow{x})\}P\{A\}$ is true.*

(2) If $[\![P]\!]((s,h)) \ni r$ implies $r \neq$ abort and $[\![A]\!]_r =$ true for all r, then $[\![\text{W}_{P,A}(\overrightarrow{x})]\!]_{(s,h)} =$ true.

(3) If $\{A\}P\{B\}$ is true, then $A \rightarrow \text{W}_{P,B}(\overrightarrow{x})$ is true.

Its proof is given in the appendix. Lemmas 6.6 and 6.9 are used there.

Definition 6.11 We say our assertion language is expressive for our programming language under the standard interpretation, when for every program P and assertion A, there is a formula W such that $[\![W]\!]_{(s,h)} = \text{true}$ if and only if (s,h) is in the weakest precondition for P and A under the standard interpretation.

$$
\begin{aligned}
&\mathrm{Exec}_{x:=e,\,\overrightarrow{x}}(n_1,n_2) = (n_1 = 0 \to n_2 = 0) \wedge \\
&\quad (n_1 > 0 \to \exists y_1 z_1 y_2 z_2 w(\mathrm{Pair2}(n_1,y_1,z_1) \wedge \mathrm{EEval}_{e,\,\overrightarrow{x}}(y_1,w) \wedge \mathrm{ChangeStore}_{\overrightarrow{x},x}(y_1,w,y_2) \\
&\quad \wedge \mathrm{EqHeap}(z_1,z_2) \wedge \mathrm{Pair2}(n_2,y_2,z_2))), \\
&\mathrm{Exec}_{x:=\mathrm{cons}(e_1,e_2),\,\overrightarrow{x}}(n_1,n_2) = (n_1 = 0 \to n_2 = 0) \wedge \\
&\quad (n_1 > 0 \to \exists y_1 z_1 y_2 z_2 w w_1 w_2(\mathrm{Pair2}(n_1,y_1,z_1) \wedge \mathrm{EEval}_{e_1,\,\overrightarrow{x}}(y_1,w_1) \wedge \mathrm{EEval}_{e_2,\,\overrightarrow{x}}(y_2,w_2) \\
&\quad \wedge \mathrm{New2}(w,z_1) \wedge \mathrm{ChangeStore}_{\overrightarrow{x},x}(y_1,w,y_2) \wedge \\
&\quad \forall xy(x \neq w \wedge x \neq w+1 \to (\mathrm{Lookup}(z_1,z,y) \leftrightarrow \mathrm{Lookup}(z_2,x,y))) \wedge \\
&\quad \mathrm{Lookup}(z_2,w,w_1) \wedge \mathrm{Lookup}(z_2,w+1,w_2) \wedge \mathrm{Pair2}(n_2,y_2,z_2))), \\
&\mathrm{Exec}_{x:=[e],\,\overrightarrow{x}}(n_1,n_2) = (n_1 = 0 \to n_2 = 0) \wedge \\
&\quad (n_1 > 0 \to \exists y_1 z_1 y_2 z_2 w w_1(\mathrm{Pair2}(n_1,y_1,z_1) \wedge \mathrm{EEval}_{e,\,\overrightarrow{x}}(y_1,w) \wedge \\
&\quad (\neg\mathrm{Domain}(w,z_1) \to n_2 = 0) \wedge \\
&\quad (\mathrm{Domain}(w,z_1) \to \mathrm{Lookup}(z_1,w,w_1) \wedge \mathrm{ChangeStore}_{\overrightarrow{x},x}(y_1,w_1,y_2) \wedge \\
&\quad \mathrm{EqHeap}(z_1,z_2) \wedge \mathrm{Pair2}(n_2,y_2,z_2)))), \\
&\mathrm{Exec}_{[e_1]:=e_2,\,\overrightarrow{x}}(n_1,n_2) = (n_1 = 0 \to n_2 = 0) \wedge \\
&\quad (n_1 > 0 \to \exists y_1 z_1 z_2 w_1 w_2(\mathrm{Pair2}(n_1,y_1,z_1) \wedge \mathrm{EEval}_{e_1,\,\overrightarrow{x}}(y_1,w_1) \wedge \mathrm{EEval}_{e_2,\,\overrightarrow{x}}(y_1,w_2) \wedge \\
&\quad (\neg\mathrm{Domain}(w_1,z_1) \to n_2 = 0) \wedge \\
&\quad (\mathrm{Domain}(w_1,z_1) \to \mathrm{ChangeHeap}(z_1,w_1,w_2,z_2) \wedge \mathrm{Pair2}(n_2,y_1,z_2)))), \\
&\mathrm{Exec}_{\mathrm{dispose}(e),\,\overrightarrow{x}}(n_1,n_2) = (n_1 = 0 \to n_2 = 0) \wedge \\
&\quad (n_1 > 0 \to \exists y_1 z_1 z_2 w(\mathrm{Pair2}(n_1,y_1,z_1) \wedge \mathrm{EEval}_{e,\,\overrightarrow{x}}(y_1,w) \wedge \\
&\quad (\neg\mathrm{Domain}(w,z_1) \to n_2 = 0) \wedge \\
&\quad (\mathrm{Domain}(w,z_1) \to \forall xy(\mathrm{Lookup}(z_1,x,y) \wedge x \neq w \leftrightarrow \mathrm{Lookup}(z_2,x,y)) \wedge \mathrm{Pair2}(n_2,y_1,z_2)))), \\
&\mathrm{Exec}_{\mathrm{if}\ (b)\ \mathrm{then}\ (P_1)\ \mathrm{else}\ (P_2),\,\overrightarrow{x}}(n_1,n_2) = (n_1 = 0 \to n_2 = 0) \wedge \\
&\quad (n_1 > 0 \to \exists xy(\mathrm{Pair2}(n_1,x,y) \wedge (\mathrm{BEval}_{b,\,\overrightarrow{x}}(x) \to \mathrm{Exec}_{P_1,\,\overrightarrow{x}}(n_1,n_2)) \\
&\quad \wedge (\neg\mathrm{BEval}_{b,\,\overrightarrow{x}}(x) \to \mathrm{Exec}_{P_2,\,\overrightarrow{x}}(n_1,n_2)))), \\
&\mathrm{Exec}_{\mathrm{while}\ (b)\ \mathrm{do}\ (P),\,\overrightarrow{x}}(n_1,n_2) = (n_1 = 0 \to n_2 = 0) \wedge \\
&\quad (n_1 > 0 \to \exists wz(\mathrm{Lh}(w,z+1) \wedge \mathrm{Elem}(w,0,n_1) \wedge \exists w_1(\mathrm{Elem}(w,z,w_1) \wedge \mathrm{EqResult}(w_1,n_2)) \wedge \\
&\quad \forall w_1(w_1 < z \to \exists z_1 z_2 w_2 w_3(\mathrm{Elem}(w,w_1,z_1) \wedge \mathrm{Elem}(w,w_1+1,z_2) \wedge \\
&\quad z_1 > 0 \wedge \mathrm{Pair2}(z_1,w_2,w_3) \wedge \mathrm{BEval}_{b,\,\overrightarrow{x}}(w_2) \wedge \mathrm{Exec}_{P,\,\overrightarrow{x}}(z_1,z_2)))) \\
&\quad \wedge (n_2 > 0 \to \exists yz(\mathrm{Pair2}(n_2,y,z) \wedge \neg\mathrm{BEval}_{b,\,\overrightarrow{x}}(y)))), \\
&\mathrm{Exec}_{P_1;P_2,\,\overrightarrow{x}}(n_1,n_2) = \exists z(\mathrm{Exec}_{P_1,\,\overrightarrow{x}}(n_1,z) \wedge \mathrm{Exec}_{P_2,\,\overrightarrow{x}}(z,n_2)).
\end{aligned}
$$

Figure 1. Definition of $\mathbf{Exec}_{P,\vec{x}}$

Theorem 6.12 (Expressiveness) *Our assertion language is expressive for our programming language under the standard interpretation.*

Proof. Since Lemma 6.10 (1) and (2) show $\mathrm{W}_{P,A}(\overrightarrow{x})$ defines the weakest precondition for P and A under the standard interpretation, the weakest precondition is definable in our language. □

7 Application to Real System

In this section, we discuss possibility of applying our theoretical results to the system implemented in [8].

Our results in this paper will give a starting point to prove completeness of the core system of existing practical systems for pointer program verification. Our current target system is the system presented by [8]. We will compare that system and our system and discuss possibility of applying our results to that system.

We can list the difference between them as follows. For the programming languages, (1) recursive procedure calls, (2) data types such as int, bool, float, and void, (3) user defined data types such as lists and trees. For the assertion languages, (4) user defined predicates and their lemmas, (5) two sorts (pointers and numbers), (6) Presburger arithmetic, (7) restricted disjunction normal forms, (8) approximation to pure logic.

Recursive procedure calls will be the next challenge. They have been intensively studied for Hoare's logic for while programs [1]. Combining those results in while programs together with our results, our framework will enable a completeness result for an extension with recursive procedure calls in separation logic.

Data types such as int, bool, float, and void are not essential difference. Our results can be extended to them by small modification.

In order to handle user defined data types that are needed for building data structures, such as lists and trees, we have to extend our programming language and assertion language. They are coded by $\text{cons}(e_1, e_2)$, so the soundness theorem will be proved straightforwardly. We have to prove the completeness theorem again since it depends on rules for user defined data types, but the key ideas in our results remains applicable.

User defined predicates and their lemmas gives logical difference, that is, the proof theoretic strength is changed by them. However, by extending our assertion languages with generalized inductive definitions [11], our results can immediately show the completeness theorem for the extension, since we assume all true assertions in the theorem.

A two-sorted system is used in their programming language and assertion language. In general, a many-sorted system is often used in programming languages and assertion languages. For example, each pointer type is typically distinct from the number type. In contrast, our system is based on a one-sorted system where a pointer is also a number. However, our results can directly apply to a two-sorted system by assigning a number to each location and coding a pointer by a unique number.

Only Presburger arithmetic is allowed in their system. Since our results rely on intensive coding by Peano arithmetic, our results do not directly apply to a system with Presburger arithmetic. Moreover, [1] showed that a system with Presburger arithmetic is inherently incomplete. Our results can show completeness results of their system extended with Peano arithmetic.

Only restricted disjunction normal forms are allowed in their systems, so that (a) conjunction of heap information is not allowed, and (b) heap information must be positive, that is, not negated. On the other hand, we used those to express the assertion $\text{Heap}(m)$ to prove the expressive theorem. In order to apply our results to their system, we have to extend their system with more flexible assertions. With those assertions, we will have to prove the completeness theorem again since it depends on the assertion language, but the key ideas in our results remains applicable.

Approximation to pure logic, called XPure, is used in their system. Its purpose is efficient automatic verification and this approximation has been proved to be sound, but it is essentially not complete. Our results can apply to the core system obtained from their system by removing the approximation.

8 Deterministic Semantics

This section studies deterministic semantics and shows completeness and soundness under deterministic semantics.

Reynolds assumed his program semantics is nondeterministic. On the other hand, some real system runs deterministically. In this section, we will discuss deterministic program semantics and show its completeness.

The nondeterminism in his semantics came from the $x := \text{cons}(e_1, e_2)$ statement. Its execution finds new free cells in memory space and allocates them. The choice of the free cells is not specified. On the other hand, in our deterministic semantics, the choice is specified. For simplicity, we suppose the execution chooses the first free cells in memory space, that is, the $x := \text{cons}(e_1, e_2)$ statement allocates the address n that is the smallest natural number such that $n > 0$ and $n, n+1$ are not used in the heap.

In deterministic semantics, the asserted program $\{1 \mapsto 5\}x := \text{cons}(0,0); y = \text{cons}(0,0)\{x = 2 \wedge y = 4\}$ becomes true. On the other hand, we will have only a restricted frame rule

$$\frac{\{A\}P\{B\}}{\{A * C\}P\{B * C\}}$$

with the additional side condition stating that the program P does not contain any allocation statements $x := \text{cons}(e_1, e_2)$.

We define the deterministic semantics of our program language. For a program P, its meaning $[\![P]\!]$ is defined as a partial map from States$\cup\{\text{abort}\}$ to States$\cup\{\text{abort}\}$. When the execution of P with the initial state r_1 terminates with the resulting state r_2, we will define $[\![P]\!](r_1) = r_2$. If the execution of P with the initial state r_1 does not terminate, $[\![P]\!](r_1)$ becomes undefined. $[\![P]\!]$ is defined by induction on P as the smallest partial map satisfying the following:

$[\![P]\!](\text{abort}) = \text{abort}$,
$[\![x := e]\!]((s,h)) = (s[x := [\![e]\!]_s], h)$,
$[\![\text{if } (b) \text{ then } (P_1) \text{ else } (P_2)]\!]((s,h)) =$
  $[\![P_1]\!]((s,h))$ if $[\![b]\!]_s = \text{true}$,
  $[\![P_2]\!]((s,h))$ otherwise,
$[\![\text{while } (b) \text{ do } (P)]\!]((s,h)) = (s,h)$ if $[\![b]\!]_s = \text{false}$,
  $[\![\text{while } (b) \text{ do } (P)]\!]([\![P]\!]((s,h)))$ otherwise,
$[\![P_1; P_2]\!]((s,h)) = [\![P_2]\!]([\![P_1]\!]((s,h)))$,
$[\![x := \text{cons}(e_1, e_2)]\!]((s,h)) =$
  $(s[x := n], h[n := [\![e_1]\!]_s, n+1 := [\![e_2]\!]_s])$,
  where n is the smallest number such that
  $n > 0$ and $n, n+1 \notin \text{Dom}(h)$,
$[\![x := [e]]\!]((s,h)) =$
  $(s[x := h([\![e]\!]_s)], h)$ if $[\![e]\!]_s \in \text{Dom}(h)$,
  abort otherwise,
$[\![[e_1] := e_2]\!]((s,h)) =$
  $(s, h[[\![e_1]\!]_s := [\![e_2]\!]_s])$ if $[\![e_1]\!]_s \in \text{Dom}(h)$,
  abort otherwise,
$[\![\text{dispose}(e)]\!]((s,h)) =$
  $(s, h|_{\text{Dom}(h)-\{[\![e]\!]_s\}})$ if $[\![e]\!]_s \in \text{Dom}(h)$,
  abort otherwise.

The difference from the semantics in Definition 3.2 comes from the meaning of the statement $x := \text{cons}(e_1, e_2)$. In the above definition, the resulting state after the execution

of the statement $x := \mathrm{cons}(e_1, e_2)$ is uniquely determined by using the smallest number n.

Our assertion language is extended by adding the predicate symbol New so that its formulas are defined by

Formulas $A ::= \mathrm{New}(e)|\mathrm{emp}|e = e|e < e|e \mapsto e|\neg A|$
$A \wedge A|A \vee A|A \to A|\forall xA|\exists xA|A * A|A \mathrel{-\!\!*} A.$

$\mathrm{New}(e)$ means to hold if and only if e is the address of the first free cell in memory space. That is, $x = e$ holds after the next $x := \mathrm{cons}(e_1, e_2)$ statement.

Our logical system is obtained from the system defined in Definition 4.1 by changing the $(cons)$ rule as follows:

$$\frac{}{\{\exists x'(\mathrm{New}(x') \wedge ((x' \mapsto e_1, e_2) \mathrel{-\!\!*} A[x := x']))\}\ x := \mathrm{cons}(e_1, e_2)\{A\} \quad (x' \notin \mathrm{FV}(e_1, e_2, A))}\ (cons)$$

We have the completeness theorem as well as the soundness theorem for this extension.

Theorem 8.1 (Soundness) *If $\{A\}P\{B\}$ is provable in the system with New(e), then $\{A\}P\{B\}$ is true under deterministic semantics.*

Theorem 8.2 (Completeness) *If $\{A\}P\{B\}$ is true under deterministic semantics, then $\{A\}P\{B\}$ is provable in the system with New(e).*

They are proved in a similar way to Theorems 5.1 and 5.2 respectively.

9 Conclusion

We have shown the completeness theorem of the pointer program verification system by separation logic presented by Reynolds. For this purpose, we have also proved the expressiveness theorem of Peano-arithmetical language for the system under the standard interpretation. We have also discussed deterministic semantics for the system, introduced the predicate that represents the next new cell, and shown the soundness and the completeness of the system under deterministic semantics.

Future work will be proving the completeness of the core of the system presented in [8] by extending our results. We will have to extend our system to support recursive procedure calls with call-by-value parameters and inductive definitions. We will also have to restrict our system from a one-sorted system to a two-sorted system. On the other hand, the core of their system will have to be designed so that the assertion language of the core system will include Peano arithmetical language and the core system does not contain the approximation to pure logic. Consequently we will know the minimum assertion language for the completeness of their system, which could be theoretically complete as well as practically efficient.

Another future work will be proving completeness results of various extensions of our system such as recursive procedure calls with call-by-name parameters and global variables, which have been intensively analyzed for while programs in several papers [1, 5, 7].

References

[1] K.R. Apt, Ten Years of Hoare's Logic: A Survey — Part I, *ACM Transactions on Programming Languages and Systems* 3 (4) (1981) 431–483.

[2] J. Berdine, C. Calcagno, and P.W. O'Hearn, Symbolic Execution with Separation Logic, In: Proceedings of the Third Asian Symposium on Programming Languages and Systems (APLAS2005), *Lecture Notes in Computer Science* 3780 (2005) 52-68.

[3] J.A. Bergstra and J.V. Tucker, Expressiveness and the Completeness of Hoare's Logic, *Journal Computer and System Sciences* 25 (3) (1982) 267–284.

[4] S.A. Cook, Soundness and completeness of an axiom system for program verification, *SIAM Journal on Computing* 7 (1) (1978) 70–90.

[5] J.Y. Halpern, A good Hoare axiom system for an ALGOL-like language, In: *Proceedings of 11th ACM symposium on Principles of programming languages (POPL84)* (1984) 262–271.

[6] S. Ishtiaq and P.W. O'Hearn, BI as an Assertion Language for Mutable Data Structures, In: *Proceedings of 28th ACM Symposium on Principles of Programming Languages (POPL2001)* (2001) 14–26.

[7] B. Josko, On expressive interpretations of a Hoare-logic for Clarke's language L4, In: Proceedings of 1st Annual Symposium of Theoretical Aspects of Computer Science (STACS 84), *Lecture Notes in Computer Science* 166 (1984) 73–84.

[8] H. H. Nguyen, C. David, S.C. Qin, and W.N. Chin, Automated Verification of Shape and Size Properties Via Separation Logic, In: Proceedings of 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2007), *Lecture Notes in Computer Science* 4349 (2007) 251–266.

[9] H.H. Nguyen and W.N. Chin, Enhancing Program Verification with Lemmas, In: Proceedings of 20th International Conference on Computer Aided Verification (CAV 2008), *Lecture Notes in Computer Science* 5123 (2008) 355-369.

[10] J.C. Reynolds, Separation Logic: A Logic for Shared Mutable Data Structures, In: *Proceedings of Seventeenth Annual IEEE Symposium on Logic in Computer Science (LICS2002)* (2002) 55–74.

[11] M. Tatsuta, Program synthesis using realizability, *Theoretical Computer Science* 90 (1991) 309–353.

Appendix

A Proof of Theorem 5.1

Lemma A.1 $[\![A]\!]_s = [\![A]\!]_{(s,h)}$ *for a base formula A where the left-hand side is the semantics for the base language and the right-hand side is the semantics for the assertion language.*

Proof. By induction on b. □

Proof of Theorem 5.1.

By induction on the proof. We consider cases according to the last rule.

Case $(assignment)$. Let P be $x := e$. Assume $[\![A[x := e]]\!]_{(s,h)} =$ true. Let n be $[\![e]\!]_s$. By the definition of $[\![x := e]\!]$, we have $[\![P]\!]((s,h)) = \{(s_1, h)\}$ where $s_1 = s[x := n]$. Since $[\![A[x := e]]\!]_{(s,h)} = [\![A]\!]_{(s_1,h)}$, we have $[\![A]\!]_{(s_1,h)} =$ true. Hence $\{A[x := e]\}x := e\{A\}$ is true.

Case (if). Assume $[\![A]\!]_{(s,h)} =$ true and $[\![P]\!]((s,h)) \ni r$. We will show $r \neq$ abort and $[\![B]\!]_r =$ true.

Case 1. $[\![b]\!]_s =$ true. By Lemma A.1, we have $[\![A \wedge b]\!]_{(s,h)} =$ true. By the definition we have $r \in [\![P]\!]((s,h)) = [\![P_1]\!]((s,h))$. By induction hypothesis for the first assumption, $\{A \wedge b\}P_1\{B\}$ is true. Hence we have $r \neq$ abort and $[\![B]\!]_r =$ true.

Case 2. $[\![b]\!]_s =$ false. $r \neq$ abort and $[\![B]\!]_r =$ true are similarly proved to Case 1.

Hence $\{A\}$if (b) then (P_1) else $(P_2)\{B\}$ is true.

Case $(while)$. Assume $[\![A]\!]_s =$ true and [\![while (b) do (P)]\!]$((s,h)) \ni r$. We will show $r \neq$ abort and $[\![A \wedge \neg b]\!]_r =$ true.

Case 1. $[\![b]\!]_s =$ true. We have some r_1 such that $[\![P]\!]((s,h)) \ni r_1$ and [\![while (b) do (P)]\!]$(r_1) \ni r$.

By Lemma A.1, we have $[\![b]\!]_{(s,h)} =$ true. Hence $[\![A \wedge b]\!]_{(s,h)} =$ true. By induction hypothesis, $\{A \wedge b\}P\{A\}$ is true. By this, $r_1 \neq$ abort and $[\![A]\!]_{r_1} =$ true.

By induction hypothesis of the induction of the definition of [\![while (b) do (P)]\!], from [\![while (b) do (P)]\!]$(r_1) \ni r$ and $[\![A]\!]_{r_1} =$ true, we have $r \neq$ abort and $[\![A \wedge \neg b]\!]_r =$ true.

Case 2. $[\![b]\!]_s =$ false. We have [\![while (b) do (P)]\!]$((s,h)) = \{(s,h)\}$. Since $r = (s,h)$, we have $r \neq$ abort. By Lemma A.1, we have $[\![b]\!]_{(s,h)} =$ false. Hence we have $[\![A \wedge \neg b]\!]_r =$ true.

Therefore $\{A\}$while (b) do $(P)\{A \wedge \neg b\}$ is true.

Case $(comp)$. Assume $[\![A]\!]_{(s,h)} =$ true and $[\![P_1; P_2]\!]((s,h)) \ni r$. We will show $r \neq$ abort and $[\![B]\!]_r =$ true.

By the definition, $[\![P_1; P_2]\!]((s,h)) = [\![P_2]\!]([\![P_1]\!]((s,h)))$. We have r_1 such that $[\![P_1]\!]((s,h)) \ni r_1$ and $[\![P_2]\!](r_1) \ni r$. By induction hypothesis for the first assumption, $\{A\}P_1\{C\}$ is true. Hence we have $r_1 \neq$ abort and $[\![C]\!]_{r_1} =$ true. By induction hypothesis for the second assumption, $\{C\}P_2\{B\}$ is true. Since $r \in [\![P_2]\!](r_1)$, we have $r \neq$ abort and $[\![B]\!]_r =$ true. Hence $\{A\}P_1; P_2\{B\}$ is true.

Case $(conseq)$. Assume $[\![A]\!]_{(s,h)} =$ true and $[\![P]\!]((s,h)) \ni r$. We will show $r \neq$ abort and $[\![B]\!]_r =$ true. By the side condition $[\![A \to A_1]\!] =$ true, we have $[\![A_1]\!]_{(s,h)} =$ true. By induction hypothesis, $\{A_1\}P\{B_1\}$ is true. Hence $r \neq$ abort and $[\![B_1]\!]_r =$ true. By the side condition $[\![B_1 \to B]\!] =$ true, we have $[\![B]\!]_r =$ true. Hence $\{A\}P\{B\}$ is true.

Case $(cons)$. Assume $[\![\forall x'(x' \mapsto e_1, e_2 \mathrel{-\!\!*} A[x := x'])]\!]_{(s,h)} =$ true and $[\![x := \mathrm{cons}(e_1, e_2)]\!]((s,h)) \ni r$. We will show $r \neq$ abort and $[\![A]\!]_r =$ true.

Let P be $x := \mathrm{cons}(e_1, e_2)$, n_1 be $[\![e_1]\!]_s$, and n_2 be $[\![e_2]\!]_s$. By the definition of $[\![P]\!]$, $r \neq$ abort and $r = (s_1, h_1)$ where $s_1 = s[x := n]$ and $h_1 = h[n := n_1, n+1 := n_2]$ for some n such that $n > 0$ and $n, n+1 \notin \mathrm{Dom}(h)$. By $[\![\forall x'(x' \mapsto e_1, e_2 \mathrel{-\!\!*} A[x := x'])]\!]_{(s,h)} =$ true, we have $[\![x' \mapsto e_1, e_2 \mathrel{-\!\!*} A[x := x']]\!]_{(s',h)} =$ true where $s' = s[x' := n]$. Let $h_2 = \phi[n := n_1, n+1 := n_2]$. We have $h_1 = h + h_2$. Then $[\![x' \mapsto e_1, e_2]\!]_{(s',h_2)} =$ true since $x' \notin \mathrm{FV}(e_1, e_2)$. Since $[\![x' \mapsto e_1, e_2 \mathrel{-\!\!*} A[x := x']]\!]_{(s',h)} =$ true, we have $[\![A[x := x']]\!]_{(s',h_1)} =$ true. Hence $[\![A]\!]_{(s_1,h_1)} =$ true, since $x' \notin \mathrm{FV}(A)$.

Case $(lookup)$. Assume $[\![\exists x'((e \mapsto x') * ((e \mapsto x') \mathrel{-\!\!*} A[x := x']))]\!]_{(s,h)} =$ true and $[\![x := [e]]\!]((s,h)) \ni r$. We will show $r \neq$ abort and $[\![A]\!]_r =$ true.

Let n be $[\![e]\!]_s$. From the assumption, we have some n_1 such that $[\![(e \mapsto x') * (e \mapsto x' \mathrel{-\!\!*} A[x := x'])]\!]_{(s',h)} =$ true where $s' = s[x' := n_1]$. Hence we have h_1, h_2 such that $h = h_1 + h_2$, $[\![e \mapsto x']\!]_{(s',h_1)} =$ true, and $[\![e \mapsto x' \mathrel{-\!\!*} A[x := x']]\!]_{(s',h_2)} =$ true. Hence $[\![A[x := x']]\!]_{(s',h)} =$ true. Since $x' \notin \mathrm{FV}(e)$, $[\![e]\!]_{s'} = [\![e]\!]_s = n$. Hence $\mathrm{Dom}(h_1) = \{n\}$ and $h_1(n) = n_1$. Since $n \in \mathrm{Dom}(h)$, $r \neq$ abort and $r = (s_1, h)$ where $s_1 = s[x := n_1]$. Since $[\![A[x := x']]\!]_{(s',h)} = [\![A]\!]_{(s_1,h)}$ by $x' \notin \mathrm{FV}(A)$, we have $[\![A]\!]_r =$ true.

Case $(mutation)$. Assume $[\![(\exists x(e_1 \mapsto x)) * (e_1 \mapsto e_2 \mathrel{-\!\!*} A)]\!]_{(s,h)} =$ true and $[\![[e_1] := e_2]\!]((s,h)) \ni r$. We will show $r \neq$ abort and $[\![A]\!]_r =$ true.

Let n be $[\![e_1]\!]_s$ and n_2 be $[\![e_2]\!]_s$. By the assumption, we have h_1, h_2 such that $h = h_1 + h_2$, $[\![\exists x(e_1 \mapsto x)]\!]_{(s,h_1)} =$ true, and $[\![e_1 \mapsto e_2 \mathrel{-\!\!*} A]\!]_{(s,h_2)} =$ true. Hence we have some n_1 such that $[\![e_1 \mapsto x]\!]_{(s',h_1)} =$ true where $s' = s[x := n_1]$. Since $[\![e_1]\!]_{s'} = [\![e_1]\!]_s = n$ by $x \notin \mathrm{FV}(e_1)$, we have $\mathrm{Dom}(h_1) = \{n\}$ and $h_1(n) = n_1$. Since $n \in \mathrm{Dom}(h)$, $r \neq$ abort

and $r = (s, h')$ where $h' = h[n := n_2]$. Let h'_1 be $\phi[n := n_2]$. Then $h' = h'_1 + h_2$. $[\![e_1 \mapsto e_2]\!]_{(s,h'_1)}$ = true. By $[\![e_1 \mapsto e_2 \mathrel{-\!\!*} A]\!]_{(s,h_2)}$ = true, we have $[\![A]\!]_{(s,h')}$ = true.

Case $(dispose)$. Assume $[\![(\exists x(e \mapsto x)) * A]\!]_{(s,h)}$ = true and $[\![\mathrm{dispose}(e)]\!]((s,h)) \ni r$. We will show $r \neq$ abort and $[\![A]\!]_r$.

Let n be $[\![e]\!]_s$. By the assumption, we have h_1, h_2 such that $h = h_1 + h_2$, $[\![\exists x(e \mapsto x)]\!]_{(s,h_1)}$ = true, and $[\![A]\!]_{(s,h_2)}$ = true. Hence we have some n_1 such that $[\![e \mapsto x]\!]_{(s',h_1)}$ = true where $s' = s[x := n_1]$. Since $[\![e]\!]_{s'} = [\![e]\!]_s = n$ by $x \notin \mathrm{FV}(e)$, $\mathrm{Dom}(h_1) = \{n\}$ and $h_1(n) = n_1$. Since $n \in \mathrm{Dom}(h)$, $r \neq$ abort and $r = (s, h'_2)$ where $h'_2 = h|_{\mathrm{Dom}(h)-\{n\}}$. Hence $h_2 = h'_2$. Therefore $[\![A]\!]_r$ = true. □

B Proof of Theorem 5.2

Proof. We will show that, for all A, P, B, if $\{A\}P\{B\}$ is true, then $\vdash \{A\}P\{B\}$ by induction on P. We consider cases according to P.

Case $x := e$. We will show $A \to B[x := e]$ is true. Assume $[\![A]\!]_{(s,h)}$. Let n be $[\![e]\!]_s$. We have $[\![P]\!]((s,h)) = \{(s_1, h)\}$ where $s_1 = s[x := n]$. Since $\{A\}P\{B\}$ is true, $[\![B]\!]_{(s_1,h)}$ = true. Since $[\![B]\!]_{(s_1,h)} = [\![B[x := e]]\!]_{(s,h)}$, we have $[\![B[x := e]]\!]_{(s,h)}$ = true. Hence $A \to B[x := e]$ is true.

By applying $(conseq)$ to $[\![A \to B[x := e]]\!]$ = true and $\vdash \{B[x := e]\}x := e\{B\}$ from $(assignment)$, we have $\vdash \{A\}P\{B\}$.

Case if (b) then (P_1) else (P_2).

We will show $\{A \wedge b\}P_1\{B\}$ is true. Assume $[\![A \wedge b]\!]_{(s,h)}$ = true and $[\![P_1]\!]((s,h)) \ni r$. We will show $r \neq$ abort and $[\![B]\!]_r$ = true. By Lemma A.1, $[\![b]\!]_s = [\![b]\!]_{(s,h)}$ = true. Hence $[\![P]\!]((s,h)) = [\![P_1]\!]((s,h)) \ni r$. Since $\{A\}P\{B\}$ is true and $[\![A]\!]_{(s,h)}$ = true, $r \neq$ abort and $[\![B]\!]_r$ = true. Hence $\{A \wedge b\}P_1\{B\}$ is true.

Similarly $\{A \wedge \neg b\}P_2\{B\}$ is true.

By induction hypothesis for P_1 and P_2, we have $\vdash \{A \wedge b\}P_1\{B\}$ and $\vdash \{A \wedge \neg b\}P_2\{B\}$. By (if), we have $\vdash \{A\}P\{B\}$.

Case while (b) do (P_1). Let P be while (b) do (P_1) and C be $\mathrm{W}_{P,B}(\overrightarrow{x})$.

We will show $\{C \wedge b\}P_1\{C\}$ is true. Assume $[\![C \wedge b]\!]_{(s,h)}$ = true and $[\![P_1]\!]((s,h)) \ni r$. We will show $r \neq$ abort and $[\![C]\!]_r$ = true. By Lemma A.1, we have $[\![b]\!]_s = [\![b]\!]_{(s,h)}$ = true. By the definition $[\![P]\!]$, $[\![P]\!]((s,h)) \supseteq [\![P]\!](r)$. Since $\{C\}P\{B\}$ is true by Lemma 6.10 (1), from $[\![C]\!]_{(s,h)}$ = true, we have $[\![P]\!]((s,h)) \not\ni$ abort. Hence $r \neq$ abort. Assume $[\![P]\!](r) \ni r'$. Since $r' \in [\![P]\!](r) \subseteq [\![P]\!]((s,h))$ and $[\![C]\!]_{(s,h)}$ = true, we have $r' \neq$ abort and $[\![B]\!]_{r'}$ = true. Hence $[\![P]\!](r) \ni r'$ implies $r' \neq$ abort and $[\![B]\!]_{r'}$ = true for all r'. By Lemma 6.10 (2), we have $[\![C]\!]_r$ = true. Hence $\{C \wedge b\}P_1\{C\}$ is true.

By induction hypothesis for P_1, we have $\vdash \{C \wedge b\}P_1\{C\}$.

By Lemma 6.10 (3) with $\{A\}P\{B\}$, $A \to C$ is true.

We will show $C \wedge \neg b \to B$ is true. Assume $[\![C \wedge \neg b]\!]_{(s,h)}$ = true. We will show $[\![B]\!]_{(s,h)}$ = true. By Lemma A.1, $[\![\neg b]\!]_s = [\![\neg b]\!]_{(s,h)}$ = true. Hence $[\![P]\!]((s,h)) = \{(s,h)\}$. Since $\{C\}P\{B\}$ is true by Lemma 6.10 (1), from $[\![C]\!]_{(s,h)}$ = true and $[\![P]\!]((s,h)) = \{(s,h)\}$, we have $[\![B]\!]_{(s,h)}$ = true. Hence $C \wedge \neg b \to B$ is true.

Since $\vdash \{C \wedge b\}P_1\{C\}$, by $(while)$, we have $\vdash \{C\}P\{C \wedge \neg b\}$. Since $A \to C$ and $C \wedge \neg b \to B$ are true, by $(conseq)$, we have $\vdash \{A\}P\{B\}$.

Case $P_1; P_2$. Let P be $P_1; P_2$ and C be $\mathrm{W}_{P_2,B}(\overrightarrow{x})$. By Lemma 6.10 (1), $\{C\}P_2\{B\}$ is true.

We will show $\{A\}P_1\{C\}$ is true. Assume $[\![A]\!]_{(s,h)}$ = true and $[\![P_1]\!]((s,h)) \ni r$. We will show $r \neq$ abort and $[\![C]\!]_r$ = true. Since $\{A\}P\{B\}$ is true, $[\![P]\!]((s,h)) \not\ni$ abort. Since $[\![P]\!]((s,h)) \supseteq [\![P_2]\!](r)$ by the definition of $[\![P]\!]$, $r \neq$ abort. We will show $[\![C]\!]_r$ = true. Assume $[\![P_2]\!](r) \ni r_1$. We will show $r_1 \neq$ abort and $[\![B]\!]_{r_1}$ = true. Then $[\![P]\!]((s,h)) \ni r_1$. Since $[\![A]\!]_{(s,h)}$ = true and $\{A\}P\{B\}$ is true, we have $r_1 \neq$ abort and $[\![B]\!]_{r_1}$ = true. Hence $[\![P_2]\!](r) \ni r_1$ implies $r_1 \neq$ abort and $[\![B]\!]_{r_1}$ = true for all r_1. By Lemma 6.10 (2), $[\![C]\!]_r$ = true. Hence $\{A\}P_1\{C\}$ is true.

By induction hypothesis for P_1 and P_2, we have $\vdash \{A\}P_1\{C\}$ and $\vdash \{C\}P_2\{B\}$. By $(comp)$, we have $\vdash \{A\}P\{B\}$.

Case $x := \mathrm{cons}(e_1, e_2)$. Let $x' \notin \mathrm{FV}(e_1, e_2, B)$, and C be $\forall x'(x' \mapsto e_1, e_2 \mathrel{-\!\!*} B[x := x'])$.

We will show $A \to C$ is true. Assume $[\![A]\!]_{(s,h)}$ = true. Let $n_1 = [\![e_1]\!]_s$ and $n_2 = [\![e_2]\!]_s$. Fix n. Let $s' = s[x' := n]$.

We will show $[\![x' \mapsto e_1, e_2 \mathrel{-\!\!*} B[x := x']]\!]_{(s',h)}$ = true. Assume $[\![x' \mapsto e_1, e_2]\!]_{(s',h'_1)}$ = true and $h'_1 + h$ exists. We have $n > 0$ and $n, n+1 \notin \mathrm{Dom}(h)$. Let $h_1 = \phi[n := n_1, n+1 := n_2]$. Since $[\![e_1]\!]_{s'} = n_1$ and $[\![e_2]\!]_{s'} = n_2$ by $x' \notin \mathrm{FV}(e_1, e_2)$, we have $h'_1 = h_1$. Let $s_1 = s[x := n]$. Since $\{A\}P\{B\}$ is true and $[\![P]\!]((s,h)) \ni (s_1, h + h_1)$, $[\![B]\!]_{(s_1,h+h_1)}$ = true. Since $[\![B[x := x']]\!]_{(s',h+h'_1)} = [\![B]\!]_{(s_1,h+h'_1)}$ by $x' \notin \mathrm{FV}(B)$, we have $[\![B[x := x']]\!]_{(s',h+h'_1)}$ = true. Therefore $[\![x' \mapsto e_1, e_2 \mathrel{-\!\!*} B[x := x']]\!]_{(s',h)}$ = true.

Hence $[\![x' \mapsto e_1, e_2 \mathrel{-\!\!*} B[x := x']]\!]_{(s',h)}$ = true for all n. Hence $[\![\forall x'(x' \mapsto e_1, e_2 \mathrel{-\!\!*} B[x := x'])]\!]_{(s,h)}$ = true. Hence $A \to C$ is true.

Since $\vdash \{C\}x := \mathrm{cons}(e_1, e_2)\{B\}$ by $(cons)$ and $A \to C$ is true, we have $\vdash \{A\}x := \mathrm{cons}(e_1, e_2)\{B\}$ by $(conseq)$.

Case $x := [e]$. Let $x' \notin \mathrm{FV}(e, B)$, C be $\exists x'(e \mapsto x' * (e \mapsto x' \mathrel{-\!\!*} B[x := x']))$, and P be $x := [e]$.

We will show $A \to C$. Assume $[\![A]\!]_{(s,h)} = \text{true}$. We will show $[\![C]\!]_{(s,h)} = \text{true}$.

Let n be $[\![e]\!]_s$. Since $\{A\}P\{B\}$ is true, $[\![P]\!]((s,h)) \not\equiv \text{abort}$. Hence $n \in \mathrm{Dom}(h)$. Let $h(n) = n_1$. We have $[\![P]\!]((s,h)) = \{(s_1, h)\}$ and $[\![B]\!]_{(s_1,h)} = \text{true}$ where $s_1 = s[x := n_1]$. Let $h_1 = h|_{\{n\}}$, $h_2 = h|_{\mathrm{Dom}(h)-\{n\}}$, and $s' = s[x' := n_1]$. Then $h = h_1 + h_2$.

We have $[\![e \mapsto x']\!]_{(s',h_1)} = \text{true}$ since $[\![e]\!]_{s'} = [\![e]\!]_s = n$ by $x' \notin \mathrm{FV}(e)$.

We will show $[\![e \mapsto x' \mathrel{-\!\!*} B[x := x']]\!]_{(s',h_2)} = \text{true}$. Assume $[\![e \mapsto x']\!]_{(s',h_1')} = \text{true}$ and $h_2 + h_1'$ exists. We have $h_1 = h_1'$. Hence $h_1' + h_2 = h$. From $[\![B]\!]_{(s_1,h)} = [\![B[x := x']]\!]_{(s',h)}$ by $x' \notin \mathrm{FV}(B)$ and $[\![B]\!]_{(s_1,h)} = \text{true}$, we have $[\![B[x := x']]\!]_{(s',h_2+h_1')} = \text{true}$. Hence $[\![e \mapsto x' \mathrel{-\!\!*} B[x := x']]\!]_{(s',h_2)} = \text{true}$.

Combining them, we have $[\![e \mapsto x' * (e \mapsto x' \mathrel{-\!\!*} B[x := x'])]\!]_{(s',h)} = \text{true}$. Hence $[\![C]\!]_{(s,h)} = \text{true}$. Hence $A \to C$ is true.

By $(lookup)$, $\vdash \{C\}P\{B\}$. Since $A \to C$ is true, by $(conseq)$, we have $\{A\}P\{B\}$.

Case $[e_1] := e_2$. Let $x \notin \mathrm{FV}(e_1)$, P be $[e_1] := e_2$, and C be $(\exists x(e_1 \mapsto x)) * (e_1 \mapsto e_2 \mathrel{-\!\!*} B)$.

We will show $A \to C$ is true. Assume $[\![A]\!]_{(s,h)} = \text{true}$. We will show $[\![C]\!]_{(s,h)} = \text{true}$. Let $n_1 = [\![e_1]\!]_s$ and $n_2 = [\![e_2]\!]_s$. Since $\{A\}P\{B\}$ is true, $[\![P]\!]((s,h)) \not\equiv \text{abort}$. Hence $n_1 \in \mathrm{Dom}(h)$. Then $[\![P]\!]((s,h)) = \{(s, h_1)\}$ and $[\![B]\!]_{(s,h_1)} = \text{true}$ where $h_1 = h[n_1 := n_2]$. Let $h_2 = h|_{\{n_1\}}$ and $h_3 = h|_{\mathrm{Dom}(h)-\{n_1\}}$. We have $h = h_2 + h_3$. Let $s' = s[x := h(n_1)]$. Since $[\![e_1]\!]_{s'} = [\![e_1]\!]_s = n_1$ by $x \notin \mathrm{FV}(e_1)$, we have $[\![e_1 \mapsto x]\!]_{(s',h_2)} = \text{true}$. Hence $[\![\exists x(e_1 \mapsto x)]\!]_{(s,h_2)} = \text{true}$.

We will show $[\![e_1 \mapsto e_2 \mathrel{-\!\!*} B]\!]_{(s,h_3)} = \text{true}$. Assume $[\![e_1 \mapsto e_2]\!]_{(s,h_2')} = \text{true}$ and $h_3 + h_2'$ exists. Then $h_2' = \phi[n_1 := n_2]$ and $h_2' + h_3 = h_1$. Since $[\![B]\!]_{(s,h_1)} = \text{true}$, we have $[\![B]\!]_{(s,h_2'+h_3)} = \text{true}$. Hence $[\![e_1 \mapsto e_2]\!]_{(s,h_2')} = \text{true}$ implies $[\![B]\!]_{(s,h_2'+h_3)} = \text{true}$ for all h_2'. Hence $[\![e_1 \mapsto e_2 \mathrel{-\!\!*} B]\!]_{(s,h_3)} = \text{true}$.

Combining them, we have $[\![C]\!]_{(s,h)} = \text{true}$. Hence $A \to C$ is true.

By $(mutation)$, $\vdash \{C\}P\{B\}$. Since $A \to C$ is true, by $(conseq)$, we have $\vdash \{A\}P\{B\}$.

Case dispose(e). Let $x \notin \mathrm{FV}(e)$, C be $(\exists x(e \mapsto x)) * B$, and P be dispose(e).

We will show $A \to C$ is true. Assume $[\![A]\!]_{(s,h)} = \text{true}$. We will show $[\![C]\!]_{(s,h)} = \text{true}$. Let $n = [\![e]\!]_s$. Since $\{A\}P\{B\}$ is true, $[\![P]\!]((s,h)) \not\equiv \text{abort}$. Hence $n \in \mathrm{Dom}(h)$. Hence $[\![P]\!]((s,h)) = \{(s, h_1)\}$ and $[\![B]\!]_{(s,h_1)} = \text{true}$ where $h_1 = h|_{\mathrm{Dom}(h)-\{n\}}$. Let $n_1 = h(n)$, $h_2 = \phi[n := n_1]$, and $s' = s[x := n_1]$. We have $h = h_1 + h_2$. Since $[\![e]\!]_{s'} = [\![e]\!]_s = n$ by $x \notin \mathrm{FV}(e)$, we have $[\![e \mapsto x]\!]_{(s',h_2)} = \text{true}$. Hence $[\![\exists x(e \mapsto x)]\!]_{(s,h_2)} = \text{true}$. Hence $[\![C]\!]_{(s,h)} = \text{true}$. Hence $A \to C$ is true.

By $(dispose)$, $\vdash \{C\}P\{B\}$. Since $A \to C$ is true, by $(conseq)$, we have $\vdash \{A\}P\{B\}$. □

C Proof of Lemma 6.6

Proof. (1) This is similarly proved to (2).

(2) The left-hand side is equivalent to $\forall s[\![\exists \overrightarrow{x}(\mathrm{Store}_{\overrightarrow{x}}(n) \wedge A)]\!]_s = \text{true}$. It is equivalent to $\exists s[\![\mathrm{Store}_{\overrightarrow{x}}(n) \wedge A]\!]_s = \text{true}$. Hence it is equivalent to $\exists s([\![\mathrm{Store}_{\overrightarrow{x}}(n)]\!]_s = \text{true} \wedge [\![A]\!]_s = \text{true})$. Since $[\![\mathrm{Store}_{\overrightarrow{x}}(n)]\!]_s = \text{true}$ is equivalent to $\mathrm{Storecode}_{\overrightarrow{x}}(n, s)$, we have the claim.

(3) We suppose $\mathrm{Heapcode}(m, h)$. We will show $[\![\mathrm{HEval}_A(m)]\!]_s = \text{true} \leftrightarrow [\![A]\!]_{(s,h)} = \text{true}$ by induction on A. We consider cases according to A.

Case A is a base formula. We have $\mathrm{HEval}_A(m) = A$ and the claim holds.

Case $A = \text{emp}$. We have $\mathrm{HEval}_A(m) = \neg\exists xy\mathrm{Lookup}(m, x, y)$. Since $[\![\text{emp}]\!]_{(s,h)} = \text{true}$, $\mathrm{Dom}(h) = \phi$, and $\neg\exists xy\mathrm{Lookup}(m, x, y)$ are equivalent, we have the claim.

Case $A = e_1 \mapsto e_2$. Let k_i be $[\![e_i]\!]_s$. All of $[\![\mathrm{HEval}_A(m)]\!]_s = \text{true}$, $\forall k_1' k_2'(\exists m_1 m_2(m = m_1 \cdot \langle (k_1', k_2') \rangle \cdot m_2 \wedge k_1' > 0) \leftrightarrow k_1' = k_1 \wedge k_2' = k_2)$, $h = \phi[k_1 := k_2]$, and $[\![A]\!]_{(s,h)} = \text{true}$ are equivalent. Hence the claim holds.

Case $A = A_1 * A_2$.

From the left-hand side to the right-hand side. Assume $[\![\mathrm{HEval}_A(m)]\!]_s = \text{true}$. We will show $[\![A]\!]_{(s,h)} = \text{true}$. We have $[\![\mathrm{Separate}(m, y_1, y_2) \wedge \mathrm{HEval}_{A_1}(y_1) \wedge \mathrm{HEval}_{A_2}(y_2)]\!]_{s[y_1:=m_1, y_2:=m_2]} = \text{true}$ for some m_1, m_2. Then $[\![\mathrm{HEval}_{A_i}(m_i)]\!]_s = \text{true}$. We have h_i such that $\mathrm{Heapcode}(m_i, h_i)$. Then $h = h_1 + h_2$. By induction hypothesis with $[\![\mathrm{HEval}_{A_i}(m_i)]\!]_s = \text{true}$, we have $[\![A_i]\!]_{(s,h_i)} = \text{true}$. Hence $[\![A]\!]_{(s,h)} = \text{true}$.

From the right-hand side to the left-hand side. Assume $[\![A]\!]_{(s,h)} = \text{true}$. We will show $[\![\mathrm{HEval}_A(m)]\!]_s = \text{true}$. There are h_1, h_2 such that $h = h_1 + h_2$ and $[\![A_i]\!]_{(s,h_i)} = \text{true}$. We have m_1, m_2 such that $\mathrm{Heapcode}(m_i, h_i)$. Then $\mathrm{Separate}(m, m_1, m_2)$. By induction hypothesis for A_i, we have $[\![\mathrm{HEval}_{A_i}(m_i)]\!]_s = \text{true}$. Hence $[\![\mathrm{HEval}_A(m)]\!]_s = \text{true}$ by taking $y_1 = m_1$ and $y_2 = m_2$.

Case $A = A_1 \mathrel{-\!\!*} A_2$.

From the left-hand side to the right-hand side. Assume $[\![\mathrm{HEval}_A(m)]\!]_s = \text{true}$. We will show $[\![A]\!]_{(s,h)} = \text{true}$. Assume $[\![A_1]\!]_{(s,h_1)} = \text{true}$ and $h + h_1$ exists. We will show $[\![A_2]\!]_{(s,h+h_1)}$. We have m_1, m_2 such that

Heapcode(m_2, h_1), Heapcode$(m_1, h + h_1)$. By induction hypothesis for A_1, we have $[\![\text{HEval}_{A_1}(m_2)]\!]_s = \text{true}$. We also have Separate$(m_1, m, m_2)$. From the assumption, we have $[\![\text{HEval}_{A_2}(m_1)]\!]_s = \text{true}$. By induction hypothesis for A_2, we have $[\![A_2]\!]_{(s,h+h_1)} = \text{true}$.

From the right-hand side to the left-hand side. Assume $[\![A]\!]_{(s,h)} = \text{true}$. We will show $[\![\text{HEval}_A(m)]\!]_s = \text{true}$. Fix m_1, m_2 and assume $[\![\text{HEval}_{A_1}(m_2) \wedge \text{Separate}(m_1, m, m_2)]\!]_s = \text{true}$. We will show $[\![\text{HEval}_{A_2}(m_1)]\!]_s = \text{true}$. We have h_1, h_2 such that Heapcode(m_1, h_1), Heapcode(m_2, h_2). Then $h_1 = h + h_2$. By induction hypothesis for A_1, we have $[\![A_1]\!]_{(s,h_2)} = \text{true}$. From the assumption, we have $[\![A_2]\!]_{(s,h_1)} = \text{true}$. By induction hypothesis for A_2, we have $[\![\text{HEval}_{A_2}(m_1)]\!]_s = \text{true}$.

Cases $A = \neg A_1, A_1 \wedge A_2, A_1 \vee A_2, A_1 \to A_2, \forall x A_1, \exists x A_1$ are proved straightforwardly by using induction hypothesis.

(4) The right-hand side is equivalent to $\exists h(\text{Heapcode}(m, h) \wedge \exists s(\text{Storecode}_{\overrightarrow{x}}(n, s) \wedge [\![A]\!]_{(s,h)} = \text{true}))$. Since $[\![A]\!]_{(s,h)} = [\![\text{HEval}_A(m)]\!]_s$ under Heapcode(m, h) by (3), it is equivalent to $\exists h(\text{Heapcode}(m, h) \wedge \exists s(\text{Storecode}_{\overrightarrow{x}}(n, s) \wedge [\![\text{HEval}_A(m)]\!]_s = \text{true}))$. Hence $\exists s(\text{Storecode}_{\overrightarrow{x}}(n, s) \wedge [\![\text{HEval}_A(m)]\!]_s = \text{true})$. Since (2) shows $\text{BEval}_{\text{HEval}_{A(m)}, \overrightarrow{x}}(n)$ is equivalent to $\exists s(\text{Storecode}_{\overrightarrow{x}}(n, s) \wedge [\![\text{HEval}_A(m)]\!]_s = \text{true})$, it is equivalent to $\text{BEval}_{\text{HEval}_{A(m)}, \overrightarrow{x}}(n)$, that is, $\exists \overrightarrow{x}(\text{Store}_{\overrightarrow{x}}(n) \wedge \text{HEval}_A(m))$, which is the left-hand side by the definition of $\text{Eval}_{A, \overrightarrow{x}}$.

(5) By induction on P. □

D Proof of Lemma 6.10

Proof. (1) Assume $[\![\text{W}_{P,A}(\overrightarrow{x})]\!]_{(s,h)} = \text{true}$ and $[\![P]\!]((s, h)) \ni r$. We will show $r \neq \text{abort}$ and $[\![A]\!]_r = \text{true}$. We have n_1,n_2,n and m such that $\text{Result}_{\overrightarrow{x}}(n_1, (s, h))$, $\text{Result}_{\overrightarrow{x}}(n_2, r)$, and $\text{Pair2}(n_1, n, m)$. We have $[\![\text{Store}_{\overrightarrow{x}}(n)]\!]_{(s,h)} = \text{true}$, $[\![\text{Heap}(m)]\!]_{(s,h)} = \text{true}$, $\text{Storecode}_{\overrightarrow{x}}(n, s)$, and Heapcode$(m, h)$.

We will show $\exists r_1 r_2(\text{Result}_{\overrightarrow{x}}(n_1, r_1) \wedge [\![P]\!](r_1) \ni r_2 \wedge \text{Result}_{\overrightarrow{x}}(n_2, r_2))$. It is proved by taking $r_1 = (s, h)$ and $r_2 = r$. Therefore, by Lemma 6.6 (5), $\text{Exec}_{P, \overrightarrow{x}}(n_1, n_2)$ is true.

By letting $x = n, y = m, z = n_1, w = n_2$ in the definition of $\text{W}_{P,A}(\overrightarrow{x})$, from $[\![\text{W}_{P,A}(\overrightarrow{x})]\!]_{(s,h)} = \text{true}$, we have $n_2 > 0 \wedge \exists y_1 z_1(\text{Pair2}(n_2, y_1, z_1) \wedge \text{Eval}_{A, \overrightarrow{x}}(y_1, z_1))$. By $n_2 > 0$, $r \neq \text{abort}$. Let $r = (s_1, h_1)$. We have n', m' such that $\text{Pair2}(n_2, n', m')$ and $\text{Eval}_{A, \overrightarrow{x}}(n', m')$ is true. By Lemma 6.6 (4), we have s_1', h_1' such that $\text{Storecode}_{\overrightarrow{x}}(n', s_1') \wedge \text{Heapcode}(m', h_1') \wedge [\![A]\!]_{(s_1', h_1')} = \text{true}$. Since $\text{Storecode}_{\overrightarrow{x}}(n', s_1)$ and Heapcode(m', h_1), we have $s_1' =_{\overrightarrow{x}} s_1$ and $h_1' = h_1$. Hence $[\![A]\!]_{(s_1, h_1)} = \text{true}$, that is, $[\![A]\!]_r = \text{true}$.

(2) Assume $\forall r([\![P]\!]((s, h)) \ni r$ implies $r \neq \text{abort} \wedge [\![A]\!]_r = \text{true})$. We will show $[\![\text{W}_{P,A}(\overrightarrow{x})]\!]_{(s,h)} = \text{true}$. Fix n, m, n_1, n_2 and assume $\text{Store}_{\overrightarrow{x}}(n)$, Heap$(m)$, $\text{Pair2}(n_1, n, m)$, and $\text{Exec}_{P, \overrightarrow{x}}(n_1, n_2)$ are true at (s, h). We will show $n_2 > 0$ and $\exists y_1 z_1(\text{Pair2}(n_2, y_1, z_1) \wedge \text{Eval}_{A, \overrightarrow{x}}(y_1, z_1))$.

We have $\text{Result}_{\overrightarrow{x}}(n_1, (s, h))$. By Lemma 6.6 (5) with $\text{Exec}_{P, \overrightarrow{x}}(n_1, n_2)$, we have r_1', r_2' such that $\text{Result}_{\overrightarrow{x}}(n_1, r_1') \wedge [\![P]\!](r_1') \ni r_2' \wedge \text{Result}_{\overrightarrow{x}}(n_2, r_2')$. By $\text{Result}_{\overrightarrow{x}}(n_1, (s, h))$, we have s' such that $r_1' = (s', h)$ and $s =_{\overrightarrow{x}} s'$. If $r_2' = \text{abort}$, by Lemma 6.9 (1), $[\![P]\!]((s, h)) \ni \text{abort}$, which contradicts to the assumption. Hence $r_2' \neq \text{abort}$. Let (s_2', h_2) be r_2'. By Lemma 6.9 (2), $[\![P]\!]((s, h)) \ni (s_2, h_2)$ where $s_2' =_{\text{FV}(P)} s_2$. By the assumption, we have $[\![A]\!]_{(s_2, h_2)} = \text{true}$. We have $s_2' =_{\overrightarrow{x}} s_2$ since for all $y \in \overrightarrow{x} - \text{FV}(P)$, $s_2'(y) = s'(y) = s(y) = s_2(y)$. Hence $[\![A]\!]_{r_2'} = [\![A]\!]_{r_2} = \text{true}$. We have n', m' such that $\text{Pair2}(n_2, n', m')$. Then $\text{Storecode}_{\overrightarrow{x}}(n', s_2')$ and Heapcode(m', h_2). Since the right-hand side of Lemma 6.6 (4) holds by letting $s = s_2'$ and $h = h_2$, we have $\text{Eval}_{A, \overrightarrow{x}}(n', m')$. Hence $\exists y_1 z_1(\text{Pair2}(n_2, y_1, z_1) \wedge \text{Eval}_{A, \overrightarrow{x}}(y_1, z_1))$ by taking $y_1 = n'$ and $z_1 = m'$.

Therefore $\forall xyzw(\text{Store}_{\overrightarrow{x}}(x) \wedge \text{Heap}(y) \wedge \text{Pair2}(z, x, y) \wedge \text{Exec}_{P, \overrightarrow{x}}(z, w) \to$
$w > 0 \wedge \exists y_1 z_1(\text{Pair2}(w, y_1, z_1) \wedge \text{Eval}_{A, \overrightarrow{x}}(y_1, z_1)))$ is true at (s, h), that is, $[\![\text{W}_{P,A}(\overrightarrow{x})]\!]_{(s,h)} = \text{true}$.

(3) Assume $[\![A]\!]_{(s,h)} = \text{true}$. We will show $[\![\text{W}_{P,A}(\overrightarrow{x})]\!]_{(s,h)} = \text{true}$.

Assume $[\![P]\!]((s, h)) \ni r$. Since $\{A\}P\{B\}$ is true, $r \neq \text{abort}$ and $[\![B]\!]_r = \text{true}$. Hence we have $[\![P]\!]((s, h)) \ni r$ implies $r \neq \text{abort}$ and $[\![A]\!]_r = \text{true}$. By (2), we have $[\![\text{W}_{P,A}(\overrightarrow{x})]\!]_{(s,h)} = \text{true}$.

Hence $A \to \text{W}_{P,B}(\overrightarrow{x})$ is true. □