Completeness of Separation Logic with Inductive Definitions for Program Verification

Makoto Tatsuta[1] and Wei-Ngan Chin[2]

[1] National Institute of Informatics,
2-1-2 Hitotsubashi, 101-8430 Tokyo, Japan
tatsuta@nii.ac.jp
[2] Department of Computer Science,
National University of Singapore,
13 Computing Drive, Singapore 117417, Singapore
chinwn@comp.nus.edu.sg

Abstract. This paper extends Reynolds' separation logical system for pointer-based while program verification by adding inductive definitions. Inductive definitions give us a great advantage for verification, since they enable us for example, to formalize linked lists and to support the lemma reasoning mechanism. This paper proves its completeness theorem that states that every true asserted program is provable in the logical system. In order to prove its completeness, this paper shows an expressiveness theorem that states the weakest precondition of every program and every assertion can be expressed by some assertion.

1 Introduction

Reynolds proposed a new logical system based on separation logic for pointer program verification [17]. It enables us to have a 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 [13,2].

Inductive definitions in logical systems to formalize properties of programs have been studied widely, for example, in [7,15,18,11]. Inductive definitions play an important role in formalizing properties of programs in logical systems. Many important data structures such as lists and trees are naturally represented in logical systems by using inductive definitions, since they are recursively defined by nature. Specifications and properties of programs can be formally represented in a natural way in a logical system with the help of inductive definitions.

Combining with separation logic, inductive definitions give a verification system a general mechanism to formalize recursive data structures such as linked lists and circular doubly-linked lists. Instead of manually adding these data structures one by one in an ad hoc way, the system uniformly formalizes all these recursive data structures once we have inductive definitions in the system. Some properties called lemmas in [14] are important for program verification. In our system, every lemma statement corresponding to each recursive data structure is

D. Giannakopoulou and G. Salaün (Eds.): SEFM 2014, LNCS 8702, pp. 20–34, 2014.

also generated automatically from the description of the recursive data structure, and the consistency of the system is automatically preserved.

One of the most important theoretical questions for a verification system is its completeness [1,5,9,12]. 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 a system shows how powerful the system is.

Our contributions are: (1) an extension of separation logic for pointer while program verification by adding inductive definitions, (2) the completeness theorem of separation logic with inductive definitions for pointer while programs, and (3) the expressiveness theorem of the separation logic with inductive definitions for pointer while programs.

We will prove the completeness by extending the completeness results of separation logic for pointer while programs given in [19] to assertions with inductive definitions. The main challenge is proving the expressiveness theorem.

We say that a logical system with the standard model is expressive for programs, if the weakest precondition of every program is definable in the logical system. At first sight, the expressiveness may look trivial, but it is indeed a subtle problem and some pathological counterexamples are known [3].

The expressiveness theorem for Peano arithmetic and separation logic was proved in [19] based on the following idea. We code 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. The idea uses natural numbers to encode the current store s and heap h, respectively. The store s is coded by a list of values in distinguished variables. We can construct a heapcode translation $\mathrm{HEval}_A(m)$ of an assertion A. $\mathrm{HEval}_A(m)$ is a pure formula such that A is true at (s, h) if and only if $\mathrm{HEval}_A(m)$ is true at s when the number m represents the heap h.

We will extend the expressiveness proof in [19] to inductive definitions. Since our system is proof-theoretically strictly stronger than the system in [19] because of inductive definitions [16], we did not know a possibility of this extension. The key in our proof of the expressiveness theorem for inductive definitions is to observe that if A is an inductively defined predicate, we can define $\mathrm{HEval}_A(m)$ by using another inductively defined predicate. This idea is a similar direction to the solutions used in an extension of type theory to inductive definitions [7,15], and an extension of realizability interpretations to inductive definitions [18].

An extension of bunched implications with inductive definitions was studied in [4]. Our assertion language is included in it, but ours is more specific for the aim of pointer program verification. They discussed only an assertion language and did not discuss asserted programs.

Recently the completeness of separation logic was actively studied [5,9,12]. However, the case of a predicate logic with inductive definitions has not been investigated yet, since [5] and [9] discussed only propositional logic, and [12] studied only a system without inductions.

Our long-term aim is proving completeness of the core of existing practical verification systems for pointer programs. This paper will give a step for this purpose. In order to analyze a verification system with built-in recursive data structures and their properties such as the lemma reasoning mechanism in [14], the separation logic with inductive definitions is indispensable. 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.

Section 2 defines our programming language and our assertion language, and gives examples of inductive definitions. 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 gives a proof sketch of the expressiveness theorem. Section 7 is the conclusion.

2 Languages

This section defines our programming language and our assertion language. Our language is obtained from Reynolds' paper [17] by adding inductive definitions to the assertion language.

Our programming language is an extension of while programs to pointers. It is the same as that of Reynolds [17].

We have variables $x, y, z, w, \ldots$, and constants $0, 1, \mathrm{null}$. Its expressions are defined as follows.

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

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

Its boolean expressions are propositional formulas defined as follows.

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

Boolean expressions are used as conditions in a program.

Programs are defined by:

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

The statement $x := \mathrm{cons}(e_1, e_2)$ allocates two new consecutive memory cells, puts e_1 and e_2 in the cells, and puts the address into x. The statement $x := [e]$ looks up the content of the memory cell at the address e and puts 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 $\mathrm{dispose}(e)$ deallocates the memory cell at the address e. We will sometimes write the number n to denote the term $1 + (1 + (1 + \ldots (1 + 0)))$ (n times of 1+). We will use i, j, k, l, m, n for natural numbers.

Our assertion language is a first-order language extended with inductive definitions and the separating conjunction $*$ and the separating implication $-\!\!*$. It is an extension of assertions in [17,19] with inductive definitions. Our assertion language is defined as follows: Terms are the same as the expressions of our programming language and denoted by t. We have predicate symbols $=, <, \mapsto$, a predicate constant emp, and predicate variables $X, Y, \ldots$. We assume that when we have a predicate variable X we also have a predicate variable $\tilde{X}$.

Open formulas $A ::= \mathrm{emp} \mid e = e \mid e < e \mid e \mapsto e \mid X(t, \ldots, t) \mid \neg A \mid A \wedge A \mid$

$$A \vee A \mid A \to A \mid \forall x A \mid \exists x A \mid (\mu X.\lambda x \ldots x.A)(t, \ldots, t) \mid A * A \mid A -\!\!* A.$$

We assume that X occurs in A only positively for $(\mu X.\lambda x_1 \ldots x_n.A)(t_1, \ldots, t_n)$. The positivity is defined in a standard manner as follows. We define the set $\mathrm{FPV}_+(A)$ of positive predicate variables and the set $\mathrm{FPV}_-(A)$ of negative predicate variables for A in a standard way. We say that X occurs only positively in A when $X \notin \mathrm{FPV}_-(A)$.

We define $\mathrm{FPV}(A)$ as $\mathrm{FPV}_+(A) \cup \mathrm{FPV}_-(A)$. We call an open formula A a formula if $\mathrm{FPV}(A) = \emptyset$. We will sometimes call a formula an assertion. We call an open formula pure when the open formula does not contain emp, $e_1 \mapsto e_2$, $A * B$, or $A -\!\!* B$.

The open formula $(\mu X.\lambda x_1 \ldots x_n.A)(t_1, \ldots, t_n)$ means the inductively defined predicate $\mu X.\lambda x_1 \ldots x_n.A$ holds for $t_1, \ldots, t_n$. The predicate $\mu X.\lambda x_1 \ldots x_n.A$ denotes the least predicate X such that $A \leftrightarrow X(x_1, \ldots, x_n)$. An open formula may contain some predicate variables. The meaning of an open formula depends on the meaning of its predicate variables. A formula does not contain any predicate variables, and its meaning is determined in an ordinary way. For an assertion, we will use only a formula, since it does not contain any free predicate variables.

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. $A -\!\!* B$ means that for any heap disjoint from the current heap such that A holds at the heap, B holds at the new heap obtained by combining the current heap and the heap. Note that $X(t_1, \ldots, t_n)$ may depend on the current heap since X could take emp or $e_1 \mapsto e_2$. The other formula constructions mean ordinary logical connectives.

$\mathrm{FV}(A)$ is defined as the set of free variables in A. $\mathrm{FV}(e)$ and $\mathrm{FV}(P)$ are similarly defined. $\mathrm{FV}(O_1, \ldots, O_n)$ is defined as $\mathrm{FV}(O_1) \cup \ldots \cup \mathrm{FV}(O_n)$ when O_i is an open formula, an expression, or a program. $A \leftrightarrow B$ is defined as $(A \to B) \wedge (B \to A)$. We use $A[x := t]$ for a standard substitution without variable capture.

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

Example 1 (linked lists). The predicate that characterizes singly linked lists is formalized by using inductive definitions as follows.

$$\mathrm{Node}(x, y, z) = x \mapsto y * x + 1 \mapsto z,$$
$$\mathrm{LL} = \mu X.\lambda xy.(x = \mathrm{null} \wedge y = 0 \vee \exists zw(\mathrm{Node}(x, z, w) * X(w, y - 1))).$$

$\mathrm{LL}(p, n)$ means that there is a singly linked list pointed by p such that its length is n.

LL is the least predicate that satisfies

$$\mathrm{LL}(p, n) \leftrightarrow p = \mathrm{null} \wedge n = 0 \vee \exists xq(\mathrm{Node}(p, x, q) * \mathrm{LL}(q, n - 1)).$$

$\mathrm{LL}(p, n)$ formalizes the predicate `p::ll<n>` given in [14]. They added their lemma properties in an ad hoc way for proof search. In our system, those properties are derived by the above general principle.

Example 2 (circular doubly-linked lists). Let

$$\begin{aligned}
&\mathrm{Node2}(x, y, z, w) = x \mapsto y * x + 1 \mapsto z * x + 2 \mapsto w,\\
&\mathrm{DSN} = \mu X.\lambda xyzwv.(x = w \wedge z = v \wedge y = 0\\
&\qquad \vee \exists y'w'(\mathrm{Node2}(x, y', z, w') * X(w', y - 1, x, w, v))),\\
&\mathrm{DCL}(x, y) = (x = \mathrm{null} \wedge y = 0 \vee\\
&\qquad \exists zw(\exists u\mathrm{Node2}(x, u, z, w) * \mathrm{DSN}(w, y - 1, x, x, z))).
\end{aligned}$$

$\mathrm{DSN}(q, s, p, n, t)$ means that there is a doubly linked list pointed by q such that its length is s, the previous pointer in the first element is p, the next pointer in the last element is n, and t points the last element. $\mathrm{DCL}(p, s)$ means that there is a circular doubly-linked list of length s pointed by p.

DSN is the least predicate that satisfies

$$\begin{aligned}
&\mathrm{DSN}(q, s, p, n, t) \leftrightarrow\\
&q = n \wedge p = t \wedge s = 0 \vee \exists rs'(\mathrm{Node2}(q, s', p, r) * \mathrm{DSN}(r, s - 1, q, n, t)).
\end{aligned}$$

Since it is the least, we can also show that one of their lemma properties

$$\mathrm{DSN}(q, s, p, n, t) \wedge s > 0 \leftrightarrow \exists r(\mathrm{DSN}(q, s - 1, p, t, r) * \exists x\mathrm{Node2}(t, x, r, n))$$

is true.

$\mathrm{DCL}(p, s)$ formalizes the predicate `p::dcl(s)`, $\mathrm{DSN}(r, s, p, n, t)$ formalizes `r::dseqN<s,p,n,t>`, and the last equivalence formula formalizes their lemma given in [14].

Example 3 (linked list segments). The predicate that characterizes linked list segments is formalized by using inductive definitions as follows.

$$\mathrm{LS} = \mu X.\lambda xy.(x = y \wedge \mathrm{emp} \vee \exists v(\exists u\mathrm{Node}(x, u, v) * X(v, y)) \wedge x \neq y).$$

$\mathrm{LS}(x, p)$ means that the heap is a linked list segment such that x points the first cell and p is the next pointer in the last cell.

LS is the least predicate that satisfies

$$\mathrm{LS}(x, p) \leftrightarrow x = p \wedge \mathrm{emp} \vee \exists v(\exists u\mathrm{Node}(x, u, v) * \mathrm{LS}(v, p)) \wedge x \neq p.$$

By this general principle we can show that

$$\mathrm{LS}(x, p) * \mathrm{Node}(p, a, b) \leftrightarrow \exists q(\mathrm{LS}(x, q) * \mathrm{LS}(q, p) * \mathrm{Node}(p, a, b))$$

is true.

$\mathrm{LS}(E, F)$ formalizes the following predicate $\mathrm{ls}(E, F)$ given in [2], where we represent $E \mapsto [f_1 : x, f_2 : y]$ by $\mathrm{Node}(E, x, y)$. These formulas are also true in our system.

$$\mathrm{ls}(E, F) \leftrightarrow (E = F \wedge \mathrm{emp}) \vee (E \neq F \wedge \exists y.E \mapsto [n : y] * \mathrm{ls}(y, F)),$$
$$\mathrm{ls}(E_1, E_2) * \mathrm{ls}(E_2, E_3) * E_3 \mapsto [\rho] \rightarrow \mathrm{ls}(E_1, E_3) * E_3 \mapsto [\rho].$$

3 Semantics

The semantics of our programming language and our assertion language is defined in this section. Our semantics is obtained by combining a standard semantics for natural numbers and inductive definitions and a semantics for programs and assertions given in Reynolds' paper [17] except the following simplification: (1) values are natural numbers, (2) addresses are non-zero natural numbers, and (3) null is 0. We call our model the standard model.

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

For sets S_1, S_2, $f : S_1 \rightarrow S_2$ means that f is a function from S_1 to S_2. $f : S_1 \rightarrow_{fin} S_2$ means that f is a finite function from S_1 to S_2, that is, there is a finite subset S_1' of S_1 and $f : S_1' \rightarrow S_2$. $\mathrm{Dom}(f)$ denotes the domain of the function f. We use $\emptyset$ and $p(S)$ to denote the empty set and the powerset of the set S respectively. For a function $f : A \rightarrow B$ and a subset $C \subseteq A$, the function $f|_C : C \rightarrow B$ is defined by $f|_C(x) = f(x)$ for $x \in C$.

A function $f : p(S) \rightarrow p(S)$ is called monotone if $f(X) \subseteq f(Y)$ for all $X \subseteq Y$. It is well-known that a monotone function has its least fixed point. The least fixed point of f is denoted by $\mathrm{lfp}(f)$.

A store is defined as a function from Vars $\rightarrow N$, and denoted by s. A heap is defined as a finite function from Locs $\rightarrow_{fin} N$, and denoted by h. We will write Heaps for the set of heaps. 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 $y \in \mathrm{Dom}(h) - \{m_1, \ldots, m_k\}$. 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 sometimes write $\emptyset$ for the empty heap whose domain is empty.

We will write $h = h_1 + h_2$ when $\mathrm{Dom}(h) = \mathrm{Dom}(h_1) \cup \mathrm{Dom}(h_2)$, $\mathrm{Dom}(h_1) \cap \mathrm{Dom}(h_2) = \emptyset$, $h(x) = h_1(x)$ for $x \in \mathrm{Dom}(h_1)$, and $h(x) = h_2(x)$ for $x \in \mathrm{Dom}(h_2)$. 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 programming language.

We define the semantics $[\![e]\!]_s$ of our expressions e and the semantics $[\![A]\!]_s$ of our boolean expressions A under the variable assignment s by the standard model of natural numbers and $[\![\text{null}]\!] = 0$. For example, $[\![e]\!]_s$ is defined by induction on e by $[\![x]\!]_s = s(x)$, $[\![0]\!]_s = 0$, $[\![1]\!]_s = 1$, $[\![\text{null}]\!]_s = 0$, $[\![e_1 + e_2]\!]_s = [\![e_1]\!]_s + [\![e_2]\!]_s$, and so on. $[\![A]\!]_s$ is defined in a similar way.

For a program P, its meaning $[\![P]\!]$ is defined as a function 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, since there are no possible resulting states in this case. Our semantics is nondeterministic since the cons statement may choose a fresh cell address and we do not allow renaming of memory addresses. $[\![P]\!]$ is defined by induction on P as follows:

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

Definition 3.2. 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 predicate variable assignment σ is a function that maps a predicate variable X with arity n to a subset of $N^n \times \text{Heaps}$. Since an open formula A may contain free predicate variables, in order to give the meaning of A, we will use a predicate variable assignment for the meaning of free predicate variables in A. The predicate variable assignment $\sigma[X_1 := S_1, \ldots, X_n := S_n]$ is defined by σ' such that $\sigma'(X_i) = S_i$ and $\sigma'(Y) = \sigma(Y)$ for $Y \notin \{X_1, \ldots, X_n\}$. We will sometimes write $\emptyset$ for the constant predicate variable assignment such that $\emptyset(X)$ is the empty set for all X.

In order to define $[\![A]\!]_{(s,h)}$ for a formula, we first define $[\![A]\!]^{\sigma}_{(s,h)}$ for an open formula by induction on A as follows:

$[\![\mathrm{emp}]\!]^{\sigma}_{(s,h)} = \mathrm{true}$ if $\mathrm{Dom}(h) = \emptyset$,
$[\![e_1 = e_2]\!]^{\sigma}_{(s,h)} = ([\![e_1]\!]_s = [\![e_2]\!]_s)$,
$[\![e_1 < e_2]\!]^{\sigma}_{(s,h)} = ([\![e_1]\!]_s < [\![e_2]\!]_s)$,
$[\![e_1 \mapsto e_2]\!]^{\sigma}_{(s,h)} = \mathrm{true}$ if $\mathrm{Dom}(h) = \{[\![e_1]\!]_s\}, h([\![e_1]\!]_s) = [\![e_2]\!]_s$,
$[\![X(\overrightarrow{t})]\!]^{\sigma}_{(s,h)} = \mathrm{true}$ if $([\![\overrightarrow{t}]\!]_s, h) \in \sigma(X)$,
$[\![\neg A]\!]^{\sigma}_{(s,h)} = (\mathrm{not}\ [\![A]\!]^{\sigma}_{(s,h)})$,
$[\![A \wedge B]\!]^{\sigma}_{(s,h)} = ([\![A]\!]^{\sigma}_{(s,h)}\ \mathrm{and}\ [\![B]\!]^{\sigma}_{(s,h)})$,
$[\![A \vee B]\!]_{(s,h)^{\sigma}} = ([\![A]\!]^{\sigma}_{(s,h)}\ \mathrm{or}\ [\![B]\!]^{\sigma}_{(s,h)})$,
$[\![A \to B]\!]^{\sigma}_{(s,h)} = ([\![A]\!]^{\sigma}_{(s,h)}\ \mathrm{implies}\ [\![B]\!]^{\sigma}_{(s,h)})$,
$[\![\forall x A]\!]^{\sigma}_{(s,h)} = \mathrm{true}$ if $[\![A]\!]^{\sigma}_{(s[x:=n],h)} = \mathrm{true}$ for all $n \in N$,
$[\![\exists x A]\!]^{\sigma}_{(s,h)} = \mathrm{true}$ if $[\![A]\!]^{\sigma}_{(s[x:=n],h)} = \mathrm{true}$ for some $n \in N$,
$[\![A * B]\!]^{\sigma}_{(s,h)} = \mathrm{true}$ if $h = h_1 + h_2$,
$\quad [\![A]\!]^{\sigma}_{(s,h_1)} = [\![B]\!]^{\sigma}_{(s,h_2)} = \mathrm{true}$ for some h_1, h_2,
$[\![A \mathrel{-\!\!*} B]\!]^{\sigma}_{(s,h)} = \mathrm{true}$ if $h_2 = h_1 + h$ and
$\quad [\![A]\!]^{\sigma}_{(s,h_1)} = \mathrm{true}$ imply $[\![B]\!]^{\sigma}_{(s,h_2)} = \mathrm{true}$ for all h_1, h_2,
$[\![(\mu X.\lambda\overrightarrow{x}.A)(\overrightarrow{t})]\!]^{\sigma}_{(s,h)} = \mathrm{true}$ if $([\![\overrightarrow{t}]\!]_s, h) \in \mathrm{lfp}(F)$ where
$\quad n$ is the length of $\overrightarrow{x}$,
$\quad F : p(N^n \times \mathrm{Heaps}) \to p(N^n \times \mathrm{Heaps})$,
$\quad F(S) = \{(\overrightarrow{l}, h) \mid [\![A]\!]^{\sigma[X:=S]}_{(s[\overrightarrow{x}:=\overrightarrow{l}],h)} = \mathrm{true}\}$.

We define $[\![A]\!]_{(s,h)}$ for a formula A as $[\![A]\!]^{\emptyset}_{(s,h)}$. We say A is true when $[\![A]\!]_{(s,h)} = \mathrm{true}$ for all (s,h).

Note that in the definition of $[\![(\mu X.\lambda\overrightarrow{x}.A)(\overrightarrow{t})]\!]^{\sigma}_{(s,h)}$, since X appears only positively in A, F is a monotone function and there is the least fixed point of F.

Since the inductively defined predicates are interpreted by the least fixed points, we have the following lemma. We use $A[X := \lambda\overrightarrow{x}.C]$ to denote the formula obtained from A by replacing $X(\overrightarrow{t})$ by $C[\overrightarrow{x} := \overrightarrow{t}]$.

Lemma 3.3. Let μ be $\mu X.\lambda\overrightarrow{x}.A$.
(1) $A[X := \mu] \leftrightarrow \mu(\overrightarrow{x})$ is true.
(2) $\forall\overrightarrow{x}(A[X := \lambda\overrightarrow{x}.C] \to C) \to \forall\overrightarrow{x}(\mu(\overrightarrow{x}) \to C)$ is true for any formula C.

They are proved by using the definition of semantics.

The claim (1) means the folding and the unfolding of inductive definitions. The claim (2) means the inductively defined predicate is the least among C satisfying $\forall\overrightarrow{x}(A[X := \lambda\overrightarrow{x}.C] \to C)$.

Definition 3.4. For an asserted program $\{A\}P\{B\}$ with assertions A and B, its meaning is defined as true or false. $\{A\}P\{B\}$ is defined to be true if the following hold.
(1) for all (s,h), if $[\![A]\!]_{(s,h)} = \mathrm{true}$, then $[\![P]\!]((s,h)) \not\ni \mathrm{abort}$.
(2) for all (s,h) and (s',h'), if $[\![A]\!]_{(s,h)} = \mathrm{true}$ and $[\![P]\!]((s,h)) \ni (s',h')$, then $[\![B]\!]((s',h')) = \mathrm{true}$.

$\{A\}P\{B\}$ means abort-free partial correctness. It implies partial correctness in the standard sense. 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.

Examples. (1) $\{0 = 1\}\text{dispose}(1); [1] := 0\{0 = 1\}$ is true. Because there is no initial state that satisfies $0 = 1$.

(2) $\{\text{emp}\}[1] := 0\{0 = 0\}$ is false. Because the abort occurs at $[1] := 0$.

(3) $\{\text{emp}\}\text{while } (0 = 0) \text{ do } (x := 0); [1] := 0\{0 = 1\}$ is true. Because we do not reach $[1] := 0$ because of the infinite loop, and the abort does not occur.

4 Logical System

This section defines our logical system. It is an extension of Reynolds' system presented in [17] so that our assertion language is extended with inductive definitions.

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.

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

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

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

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

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

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

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

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

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

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.

Note that in the side condition ($A \to A_1$ true, $B_1 \to B$ true) of the rule (*conseq*), the truth means one in the standard model of natural numbers and inductive

definitions. Theoretically there are several interesting choices for the truth of this side condition [1]. Since we are interested in whether a given implementation of this logical system is indeed powerful enough in a real world, we choose the truth of the standard model. Hence the completeness of our system means completeness relative to all true formulas in the standard model of natural numbers and inductive definitions.

5 Soundness and Completeness Theorems

Our main results are the completeness theorem and the expressiveness theorem stated in this section. We will also show the soundness theorem. The soundness theorem is proved in a similar way to [17] and [19]. The completeness theorem is proved in a similar way to [19] if we have the expressiveness theorem. A proof of the completeness theorem requires the expressiveness theorem. Since our assertion language is extended with inductive definitions, the expressiveness theorem for our assertion language is really new. For this reason, the completeness result is also new. We will give only proof sketches of the soundness theorem and the completeness theorem.

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

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.

Definition 5.2. For a program P and an assertion A, the weakest precondition for P and A under the standard model is defined as the set $\{(s,h)|\forall r([\![P]\!]((s,h)) \ni r \to r \neq \text{abort} \wedge [\![A]\!]_r = \text{true})\}$.

Our proof of the completeness theorem will use the next expressiveness theorem, which will be proved in the next section.

Theorem 5.3 (Expressiveness). 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 defined in Definition 5.2 for P and A under the standard model.

Theorem 5.4 (Completeness). If $\{A\}P\{B\}$ is true, then $\{A\}P\{B\}$ is provable.

This theorem says that a given asserted program $\{A\}P\{B\}$ is true (defined in Section 3), then this is provable (defined in Section 4). Note that it is relative completeness in the sense that our logical system assumes all true formulas in the standard model of natural numbers and inductive definitions. This is the best possible completeness for pointer program verification for a similar reason to that for while program verification discussed in [6].

We sketch 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.

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 5.3, to take the weakest precondition for P_2 and B as the assertion C.

6 Proof Sketch of Expressiveness Theorem

This section gives a sketch of proofs of the expressiveness theorem (Theorem 5.3). We extend the expressiveness proof given in [19] to inductive definitions. We assume the readers of this section have knowledge of [19] and [20].

In order to show the expressiveness theorem, we have to construct a formula that expresses the weakest precondition for given a program P and a formula A. We will follow the technique used in [19] and [20]. The main technique is to translate separation logic into ordinary first-order logic by coding a heap by a natural number and simulating a separation-logic formula by a pure formula produced by its translation. First we translate a separation-logic formula A into a pure formula $\mathrm{HEval}_A(m)$ such that A is true at the current heap h if and only if $\mathrm{HEval}_A(m)$ is true where m is a natural number that represents the current heap h. We say m is a code of h. Secondly we give a pointer program P a semantics $\mathrm{Exec}_P((n_1, m_1), (n_2, m_2))$ that manipulates the code of the current heap instead of the current heap itself. We will define the pure formula $\mathrm{Exec}_P((n_1, m_1), (n_2, m_2))$ such that when the current heap is represented by m_1, if we execute P, then the current heap is changed into some heap represented by m_2. Finally the weakest precondition for P and A is described by a formula $\mathrm{W}_{P,A}$ that transforms the current heap into its heap code m_1, requires $\mathrm{Exec}_P((n_1, m_1), (n_2, m_2))$ for executing P, and requires $\mathrm{HEval}_A(m_2)$ for enforcing A at the resulting heap m_2. This formula $\mathrm{W}_{P,A}$ proves our expressiveness theorem.

Since our assertions include inductive definitions, it is non-trivial to make this technique work for our system. In particular, the main challenge is to define a translation scheme HEval_A for assertions of form A that contain inductive definitions. This section shows it is actually possible. Similar problems occurred in type theory and realizability interpretations. An extension of type theory to inductive definitions was solved in [7] and [15], and an extension of realizability interpretations to inductive definitions was solved in [18]. Their ideas were to use another inductive definition for translating a given inductive definition. Our solution will be similar to these ideas.

We will define a heapcode translation $\mathrm{HEval}_A(m)$ of an assertion A such that $\mathrm{HEval}_A(m)$ is a pure formula for expressing the meaning of A at the heap coded by m. The main question is how to define $\mathrm{HEval}_A(m)$ for inductively defined predicates. To answer this question, we will show that we can define $\mathrm{HEval}_{(\mu X.\lambda \overrightarrow{x}.A)(\overrightarrow{t})}(m)$ as $(\mu \tilde{X}.\lambda \overrightarrow{x} y.\mathrm{HEval}_A(y) \wedge \mathrm{IsHeap}(y))(\overrightarrow{t}, m)$ by using another inductively defined predicate $\mu \tilde{X}.\lambda \overrightarrow{x} y.\mathrm{HEval}_A(y) \wedge \mathrm{IsHeap}(y)$, and we will also show that this satisfies a desired property (Lemma 6.8).

Semantics for Pure Formulas

When we simulate an inductively defined separation-logic formula by some inductively defined pure formula, in order to avoid complications, we introduce the semantics of pure formulas, which does not depend on a heap. This semantics has the same meaning as our semantics defined in Section 3, and is a standard semantics for pure formulas with inductive definitions, for example, given in [18,16].

Definition 6.1. For a store s, and a pure formula A, according to the standard interpretation of a first-order language with inductive definitions, the meaning $[\![A]\!]_s$ is defined as true or false. $[\![A]\!]_s$ is the truth value of A under the store s.

A pure predicate variable assignment σ is a function that maps a predicate variable of arity n to a subset of N^n. The pure predicate variable assignment $\sigma[X_1 := S_1, \ldots, X_n := S_n]$ and the pure constant predicate variable assignment $\emptyset$ are defined in a similar way to Section 3.

In order to define $[\![A]\!]_s$ for a pure formula A, we first define $[\![A]\!]^\sigma_s$ for a pure open formula A as follows. We give only interesting cases.

$$[\![X(\overrightarrow{t})]\!]^\sigma_s = \text{true if } [\![\overrightarrow{t}]\!]_s \in \sigma(X),$$
$$[\![(\mu X.\lambda \overrightarrow{x}.A)(\overrightarrow{t})]\!]^\sigma_s = \text{true if } [\![\overrightarrow{t}]\!]_s \in \mathrm{lfp}(F) \text{ where } n \text{ is the length of } \overrightarrow{x},$$
$$F : p(N^n) \to p(N^n),$$
$$F(S) = \{\overrightarrow{l} \mid [\![A]\!]^{\sigma[X:=S]}_{s[\overrightarrow{x}:=\overrightarrow{l}]} = \text{true}\}.$$

We define $[\![A]\!]_s$ for a pure formula A as $[\![A]\!]^\emptyset_s$.

In order to show $[\![A]\!]_s = [\![A]\!]_{(s,h)}$ for a pure formula A, we need some preparation.

For a pure predicate variable assignment σ, we define a predicate variable assignment $\sigma \times \mathrm{Heaps}$ by $(\sigma \times \mathrm{Heaps})(X) = \sigma(X) \times \mathrm{Heaps}$. For a subset S of $N^n \times \mathrm{Heaps}$, we call the subset S H-independent when $S = S' \times \mathrm{Heaps}$ for some subset S' of N^n. For a predicate variable assignment σ, we call the predicate variable assignment σ H-independent when $\sigma(X)$ is H-independent for all X.

Lemma 6.2. Suppose A is pure.

(1) $\{(s,h) \mid [\![A]\!]^\sigma_{(s,h)} = \text{true}\}$ is H-independent if σ is H-independent.

(2) $[\![A]\!]^\sigma_s = [\![A]\!]^{\sigma \times \mathrm{Heaps}}_{(s,h)}$ for all heaps h.

Lemma 6.3. For a pure formula A, we have $[\![A]\!]_s = [\![A]\!]_{(s,h)}$ for any h.

Proof. By letting $\sigma = \emptyset$ in Lemma 6.2 (2). □

Heapcode Translation

We define a heapcode translation of an assertion A that is a pure formula and describes the meaning of A in terms of the heap code. This is based on the same idea in [19]. Our key idea is to find that it is possible to define $\mathrm{HEval}_A(x)$ for an inductively defined predicate A by using another inductively defined predicate.

Definition 6.4. We define the pure open formula $\mathrm{HEval}_A(x)$ for the open formula A by induction on A. We give only interesting cases.

$$\mathrm{HEval}_{X(\overrightarrow{t})}(m) = \tilde{X}(\overrightarrow{t}, m),$$
$$\mathrm{HEval}_{(\mu X.\lambda \overrightarrow{x}.A)(\overrightarrow{t})}(m) = (\mu \tilde{X}.\lambda \overrightarrow{x} y.\mathrm{HEval}_A(y) \wedge \mathrm{IsHeap}(y))(\overrightarrow{t}, m).$$

For a formula A, $\mathrm{HEval}_A(m)$ means $[\![A]\!]_{(s,h)} = \mathrm{true}$ where s is the current store and m represents the heap h. That is, we have $[\![\mathrm{HEval}_A(m)]\!]_s = [\![A]\!]_{(s,h)}$ if m represents the heap h. This will be formally stated in Lemma 6.8.

Note that in the definition of $\mathrm{HEval}_{(\mu X.\lambda \overrightarrow{x}.A)(\overrightarrow{t})}(m)$, since X appears only positively in A, $\tilde{X}$ appears only positively in $\mathrm{HEval}_A(y)$. We have $\mathrm{FPV}(\mathrm{HEval}_A(m)) = \{\tilde{X} | X \in \mathrm{FPV}(A)\}$. In particular, when $(\mu X.\lambda \overrightarrow{x}.A)(\overrightarrow{t})$ is a formula, $(\mu \tilde{X}.\lambda \overrightarrow{x} y.\mathrm{HEval}_A(y) \wedge \mathrm{IsHeap}(y))(\overrightarrow{t}, m)$ is also a formula.

Definition 6.5. We define the pure formula $\mathrm{Eval}_{A, \overrightarrow{x}}(n, m)$ for the assertion A. We suppose $\overrightarrow{x}$ includes $\mathrm{FV}(A)$.

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

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

Key Lemma

To utilize the heapcode translation defined just above, we need the key lemma that states that the semantics of a separation-logic formula equals the semantics of the corresponding pure formula obtained by the translation even if our system includes inductive definitions.

We define $(\_)^*$ for transforming semantics for heaps between that for heap codes.

Definition 6.6. We use $\mathrm{Heapcode}(m, h)$ to mean the number m is the code that represents the heap h. For $S \subseteq N^n \times \mathrm{Heaps}$, we define

$$S^* = \{(\overrightarrow{l}, m) \mid (\overrightarrow{l}, h) \in S, \mathrm{Heapcode}(m, h)\},$$

For a predicate assignment σ, we define σ^* by $\sigma^*(\tilde{X}) = \sigma(X)^*$.

The role of S^* is to give the semantics of the corresponding pure formula when S gives the semantics of a separation-logic formula.

In order to prove Lemma 6.8, we need the following key lemma, which is a generalization of Lemma 6.8 for open formulas.

Lemma 6.7 (Key Lemma). Suppose A is an open formula and $y \notin \mathrm{FV}(A)$. We have $\forall mh(\mathrm{Heapcode}(m,h) \to [\![\mathrm{HEval}_A(y)]\!]^{\sigma^*}_{s[y:=m]} = [\![A]\!]^{\sigma}_{(s,h)})$.

The next lemma shows that the pure formula $\mathrm{HEval}_A(m)$ actually has the meaning we explained above.

Lemma 6.8. Suppose A is a formula. We have $\mathrm{Heapcode}(m,h) \to [\![\mathrm{HEval}_A(m)]\!]_s = [\![A]\!]_{(s,h)}$.

Proof. By letting $\sigma = \emptyset$ in Lemma 6.7. □

Once HEval_A is defined and Lemma 6.8 is shown, we can construct the formula required in the expressiveness theorem in a similar way to [19]. Note that $\mathrm{Eval}_{A,\overrightarrow{x}}$ below is extended to inductive definitions. We will use $\mathrm{Pair2}(k,n,m)$ to mean that k represents the state (s,h) when n represents s and m represents h.

Definition 6.9. We define the formula $\mathrm{W}_{P,A}(\overrightarrow{x})$ for the program P and the assertion A. We fix some sequence $\overrightarrow{x}$ of the variables in $\mathrm{FV}(P,A)$.

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

$\mathrm{W}_{P,A}(\overrightarrow{x})$ means the weakest precondition for P and A. That is, $\mathrm{W}_{P,A}(\overrightarrow{x})$ gives the weakest assertion W such that $\{W\}P\{A\}$ is true. Note that all the free variables in $\mathrm{W}_{P,A}(\overrightarrow{x})$ are $\overrightarrow{x}$ and they appear only in $\mathrm{Store}_{\overrightarrow{x}}(x)$. This formula is the formula that describes the weakest precondition, and by this formula we can prove the expressiveness theorem (Theorem 5.3).

7 Conclusion

We have shown the completeness theorem of the pointer while program verification system which is an extension of Reynolds' separation logic with inductive definitions. For this purpose, we have also proved the expressiveness theorem of Peano arithmetic, the separation logic, and inductive definitions for pointer while programs under the standard model.

Future work would be to find a assertion language with inductive definitions that would be more suitable for automated deduction. For example, it would be interesting to find what syntactical condition guarantees that the claim (1) derives the claim (2) in Lemma 3.3. It would be also interesting to find a decidable fragment of a logical system with inductive definitions.

Another future work would 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 by several papers [1,8,10].

References

1. Apt, K.R.: Ten Years of Hoare's Logic: A Survey — Part I. ACM Transactions on Programming Languages and Systems 3(4), 431–483 (1981)
2. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic Execution with Separation Logic. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 52–68. Springer, Heidelberg (2005)
3. Bergstra, J.A., Tucker, J.V.: Expressiveness and the Completeness of Hoare's Logic. Journal Computer and System Sciences 25(3), 267–284 (1982)
4. Brotherston, J.: Formalised Inductive Reasoning in the Logic of Bunched Implications. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 87–103. Springer, Heidelberg (2007)
5. Brotherston, J., Villard, J.: Parametric Completeness for Separation Theories. In: Proceedings of POPL 2014, pp. 453–464 (2014)
6. Cook, S.A.: Soundness and completeness of an axiom system for program verification. SIAM Journal on Computing 7(1), 70–90 (1978)
7. Coquand, T., Paulin, C.: Inductively Defined Types. In: Martin-Löf, P., Mints, G. (eds.) COLOG 1988. LNCS, vol. 417, pp. 50–66. Springer, Heidelberg (1990)
8. Halpern, J.Y.: A good Hoare axiom system for an ALGOL-like language. In: Proceedings of POPL 1984, pp. 262–271 (1984)
9. Hou, Z., Clouston, R., Gore, R., Tiu, A.: Proof search for propositional abstract separation logics via labelled sequents. In: Proceedings of POPL 2014, pp. 465–476 (2014)
10. Josko, B.: On expressive interpretations of a Hoare-logic for Clarke's language L4. In: Fontet, M., Mehlhorn, K. (eds.) STACS 1984. LNCS, vol. 166, pp. 73–84. Springer, Heidelberg (1984)
11. Kimura, D., Tatsuta, M.: Call-by-Value and Call-by-Name Dual Calculi with Inductive and Coinductive Types. Logical Methods in Computer Science 9(1), Article 14 (2013)
12. Lee, W., Park, S.: A Proof System for Separation Logic with Magic Wand. In: Proceedings of POPL 2014, pp. 477–490 (2014)
13. Nguyen, H.H., David, C., Qin, S.C., Chin, W.N.: Automated Verification of Shape and Size Properties Via Separation Logic. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 251–266. Springer, Heidelberg (2007)
14. Nguyen, H.H., Chin, W.N.: Enhancing Program Verification with Lemmas. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 355–369. Springer, Heidelberg (2008)
15. Paulin-Mohring, C.: Extracting F_ω's programs from proofs in the Calculus of Constructions. In: Proceedings of POPL 1989, pp. 89–104 (1989)
16. Pohlers, W.: Proof Theory. Springer (2009)
17. Reynolds, J.C.: Separation Logic: A Logic for Shared Mutable Data Structures. In: Proceedings of LICS 2002, pp. 55–74 (2002)
18. Tatsuta, M.: Program synthesis using realizability. Theoretical Computer Science 90, 309–353 (1991)
19. Tatsuta, M., Chin, W.N., Al Ameen, M.F.: Completeness of Pointer Program Verification by Separation Logic. In: Proceeding of SEFM 2009, pp. 179–188 (2009)
20. Tatsuta, M., Chin, W.N., Al Ameen, M.F.: Completeness of Pointer Program Verification by Separation Logic. NII Technical Report, NII-2009-013E (2009)