
Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

**Technical Report No. NJU-SEG-2013-IJ-005**

**2013-IJ-005**

# Scope Logic: An Extension to Hoare Logic for Pointers and Recursive Data Structures

Jianhua Zhao, Xuandong Li

Theoretical Aspects of Computing–ICTAC 2013

# Scope Logic: An Extension to Hoare Logic for Pointers and Recursive Data Structures *

Zhao Jianhua and Li Xuandong

State Key Laboratory of Novel Software Technology
Dept. of Computer Sci. and Tech. Nanjing University
Nanjing, Jiangsu, P.R. China 210093
{zhaojh,lxd}@nju.edu.cn



**Abstract.** This paper presents an extension to Hoare Logic for pointer program verification. The main observation leading to this logic is that the value of an expression $e$ depends only on the contents stored in a finite set of memory units. This set can be specified using another expression (called the memory scope of $e$) constructed syntactically from $e$. A set of construction rules are given in this paper for expressions which may contain recursive functions (predicates). It is also observed that the memory scope of $e$ is a super set of the memory scope of the memory scope of $e$. Based on this, local reasoning can be supported using assertion variables which represent arbitrary assertions. Program-point-specific expressions are used to specify the relations between different program points. Another feature of this logic is that for formulas with no user-defined functions, the weakest-preconditions can be calculated w.r.t. assignments.

## 1 Introduction

Hoare Logic[1] can not deal with pointer programs because of pointer alias, i.e. many pointers may refer to one memory location. Some extensions to Hoare Logic have been made to deal with pointers or shared mutable data structures [2][3][4]. Among them, Separation Logic is the most successful one. In that logic, the separation-conjunction connective $*$ is introduced to specify that two assertions hold on disjoint subheaps respectively. Based on this, heap-manipulation programs can be specified and verified. An important advantage of Separation Logic is that it supports local reasoning. However, Separation Logic is counter-intuitive to some extent. This may cause some difficulties to software engineers. For example, a programmer may use $\texttt{isList}(p) \wedge \texttt{isList}(q)$ to specify that both $p$ and $q$ point to lists. However, in Separation Logic, it also means that $\texttt{isList}(p)$ and $\texttt{isList}(q)$ hold for the exact same heap, which implies that $p$ and $q$ point to the same list. It is also difficult to use many existing logic tools designed for conventional first order logic because of the new logical connective and the new semantic of conventional connectives.

* This paper is supported by the Chinese National 863 Project, NO.2011AA010103

Z. Liu, J. Woodcock, and H. Zhu (Eds.): ICTAC 2013, LNCS 8049, pp. 409–426, 2013.

Weakest precondition calculation is useful for code verification. Using the weakest precondition calculation, a program specification $\{p\}\ s\ \{q\}$ can be reduced into a logical formula $p \Rightarrow \mathsf{WP}(q, s)$, where $\mathsf{WP}(q, s)$ means the weakest precondition of $s$ for $q$. However, the weakest precondition calculation in Separation Logic is hard to deal with using conventional logic tools, because of the separating implication $-\!\!*$ and the quantifications in the preconditions.

This paper presents another extension to Hoare Logic for verification of pointer programs with recursive data structures. This logic use conventional logical connectives only. Program states are specified by FOL formulas augmented with user-defined recursive functions.

The main observation leading to this logic is that the value of an expression (or a formula) $e$, which may contain recursively defined functions, depends only on the contents stored in a finite set of memory units. We present a set of rules to syntactically construct an expression (called the *memory scope* of $e$, denoted $\mathfrak{M}(e)$) to express this set. The value of $e$ keeps unchanged if no memory unit in this set is modified by program statements. Another important property of the memory scopes is that a memory scope expression is the super-set of its memory scope. Based on this, our logic supports local reasoning using assertion variables.

Besides establishing that some properties hold at a given program state, people are also interested in how the values of variables and recursive data structures are changed by the program. Using program-point-specific expressions, we can specify and verify relations between different program points (states). Weakest precondition calculation is also supported in our logic for a large set of formulas w.r.t. assignment statements, using program-point-specific expressions.

This paper is organized as follows. We first describe the syntax of programs and specifications in Section 2. A set of axioms are introduced in Section 3 to model memory access and layout in pointer programs. In Section 4, we introduce the concept of memory scopes. The rules to syntactically construct memory scopes are given in this section. Two important properties about memory scopes are also discussed. The axioms and proof rules about program statements are given in Section 5. In Section 6, the weakest precondition calculation of assignments are discussed. Section 7 discusses how to support local reasoning using assertion variables. A brief description of our supporting tool is given in Section 8. Section 9 concludes this paper.

## 2 The Syntax of Programs and Specifications

### 2.1 The Type Systems and Expressions

The small program language used in this paper is strongly typed. Each expression has a static type. The following types and their operators can be used in programs. Their meanings are similar to those in the C language.

1. The integer type (**int**) and the boolean type (**bool**). Operators of these basic types can be used in programs.

2. Array types. Let $t$ be a type and $c$ be a positive integer constant, $\mathbf{ARR}(t, c)$ is an array type. Given an expression $e$ with type $\mathbf{ARR}(t, c)$ and an integer-typed expression $e_i$, $e[e_i]$ is an expression with type $t$. It means the $e_i^{th}$ elements of $e$ if $0 \leq e_i < c$.
3. Record types. Let $t_1, \ldots, t_k$ be types, and $n_1, n_2, \ldots, n_k$ be $k$ different names, $\mathbf{REC}((n_1, t_1), \ldots, (n_k, t_k))$ is a *record-types*. Let $e$ be an expression of this record type, $e.n_i$ is an expression with type $t_i$. It means the field $n_i$ of $e$.
4. Pointer types. Let $t$ be a type, $\mathbf{P}(t)$ is a pointer type. **Ptr** is the super type of all pointer types. The symbol **nil** is used to represent the null pointer. Let $e$ be an expression with type $\mathbf{P}(t)$.
   - The type of $*e$ is $t$.
   - If $t$ is a record type $\mathbf{REC}((n_1, t_1), \ldots, (n_k, t_k))$, $e \to n_i$ and $(*e).n_i$ are expressions with type $t_i$. These two expressions are equivalent.
   - If $t$ is an array type $\mathbf{ARR}(t, c)$, $(*e)[e_i]$ is an expression with type $t$ if $e_i$ is an expression with type **int**.

We can also define user-defined types using the form *name* := *type*. In such a definition, $\mathbf{P}(name)$ can appear in the right-hand to define recursive data types.

*Example 1.* The following is the definition of the node type for binary trees.

$$\texttt{Node} = \mathbf{REC}((l, \mathbf{P}(\texttt{Node})), (r, \mathbf{P}(\texttt{Node})), (K, \mathbf{int}))$$

Let $v$ be a program variable with type `Node`, the expression $v.l \to K$ represents the field $K$ of the left-child of $v$. □

### 2.2 The Syntax of Program Statements

The small program language has three kinds of primitive statements (**skip**, assignment, and memory allocation) and three kinds of control-flow statements (sequence, selection, and repetition). The syntax of statements is as follows.

$$st ::= \texttt{skip} \mid e_1 := e_2 \mid e := \texttt{alloc}(t) \mid st; st \mid \mathbf{if}\ (e)\ st\ \mathbf{else}\ st \mid \mathbf{while}\ (e)\ st$$

The statement `skip` does nothing. The statement $e := \texttt{alloc}(t)$ allocates a new $t$-typed memory block, and stores the reference to this block into the memory unit referred by L-value of $e$. The statement $e_1 := e_2$ stores the value of $e_2$ into the memory unit referred by L-value of $e_1$. The semantic of control-flow statements are same as those in C. For an assignment $e_1 := e_2$, the type of $e_1$ and $e_2$ must be the same and must be **int**, **bool**, or a pointer type. For a memory allocation statement $e := \texttt{alloc}(t)$, the type of $e$ must be $\mathbf{P}(t)$. For **while**-statements and **if**-statements, the type of $e$ must be **bool**.

*Example 2.* A program is depicted in Fig. 1. The first two lines declare program variables `k`, `root` and `pt` respectively with type **int** and $\mathbf{P}(\texttt{Node})$, where `Node` is the type defined in Example 1. This program searches a binary search tree for a node of which the field $K$ equals `k`. The program variable `pt` is **nil** if no such node is found, otherwise it points to the node in the tree. □

```
int k;
P(Node) root, pt;
pt:=root;
while (pt ≠ nil ∧ pt→K ≠ k)
        if (k < pt → K )
           pt := pt → l
        else
           pt := pt → r;
```

**Fig. 1.** A program

### 2.3 The Syntax of Formulas and Specifications

**The Syntax of Formulas.** All program expressions with type **bool** can be used as formulas. For example, $\mathsf{p} \to K \geq 0$ is a formula. Besides, formulas can also use the operators associated with some abstract types, like finite sets (**SetOf**($t$)) and finite lists (**ListOf**($t$)). User-defined recursive functions can also be used in formulas. There are also some new kinds of expressions (formulas) as follow.

1. A free variable $x$. It is used in expressions of the form $\lambda x.e_1[e_2]$ or $\forall x \in e_1.e_2$ and the right-hand of function definitions.
2. The reference operator &. Given an expression $e$, $\&e$ gives the L-value (address) of $e$. Here, $e$ must be a program variable, $*e_1$, $e_1.n$, $e_1 \to n$, or $e_1[e_2]$ for some expressions $e_1$ and $e_2$.
3. Conditional expressions $e_0?e_1 : e_2$. Here $e_0$ is called the *guard* of this expression. The type of $e_0$ must be **bool**, and $e_1$ and $e_2$ must have the same type. The type of this expression is the type of $e_1$(or $e_2$). If $e_0$ evaluates to **true**, the value of $e_0?e_1 : e_2$ is that of $e_1$; otherwise, the value is that of $e_2$.
4. Universal quantifier over a set $\forall x \in e_1.e_2$. The type of $x$, $e_1$, $e_2$ must be $t$, **SetOf**($t$) and **bool** respectively for some $t$. The variable $x$ can only appear in $e_2$. The expression $\forall x \in e_1.e_2$ means that for all elements $x$ in $e_1$, $e_2$ holds.
5. Set-image expressions $\lambda x.e_1[e_2]$. The type of $e_2$ must be **SetOf**($t$) for some $t$. The $t$-typed free variable $x$ can only appear in $e_1$. Let $t'$ be the type of $e_1$, the type of $\lambda x.e_1[e_2]$ is **SetOf**($t'$). This expression means the set $\{e_1 | x \in e_2\}$.
6. Union expression $\bigcup e$. The type of $e$ must be **SetOf**(**SetOf**($t$)) for some $t$. The type of $\bigcup e$ is **SetOf**($t$). $\bigcup e$ means the set $\{x | \exists s.(x \in s \wedge s \in e)\}$.
7. Program-point-specific expressions $e@i$. It is required that $e$ contains no free variables. Such expressions are treated as special constant symbols in our logic. The next sub-subsection will give more details.

*Example 3.* Three recursive functions are defined in Fig. 2. $\mathsf{NodeSet}(x)$ yields the node set of the binary tree with root node $x$. The function $\mathsf{isHBST}(x)$ asserts that $x$ is the root of a binary search tree. $\mathsf{KeySet}(x)$ yields the set of keys stored in the binary search tree.

The formula $\forall x \in \mathsf{NodeSet}(\mathsf{root} \to l).(x \to K < \mathsf{root} \to K)$ says that all the keys in the left-subtree is less than the key in the root node. The formula $\&\mathsf{pt} \to K \in (\lambda x.(\&x \to K)[\mathsf{NodeSet}(\mathsf{root})])$ says that the address of $\mathsf{pt} \to K$ is in the set of addresses of the field $K$ of the nodes in the tree. From the axiom REC-2 presented later, this formula is equivalent to $\mathsf{pt} \in \mathsf{NodeSet}(\mathsf{root})$. □

$$
\begin{array}{l}
\mathsf{NodeSet}(x : \mathbf{P}(\mathtt{Node})) : \mathbf{SetOf}(\mathbf{P}(\mathtt{Node})) \\
\quad \triangleq (x = \mathbf{nil})?\ \emptyset : (\{x\} \cup \mathsf{NodeSet}(x \to l) \cup \mathsf{NodeSet}(x \to r)) \\
\mathsf{isHBST}(x : \mathbf{P}(\mathtt{Node})) : \mathbf{bool} \\
\quad \triangleq (x = \mathbf{nil})?\mathbf{true} : \mathsf{isHBST}(x \to l) \wedge \mathsf{isHBST}(x \to r) \wedge \\
\qquad\qquad\qquad\qquad \forall y \in \mathsf{NodeSet}(x \to l).(y \to K < x \to K) \wedge \\
\qquad\qquad\qquad\qquad \forall y \in \mathsf{NodeSet}(x \to r).(y \to K > x \to K) \\
\mathsf{KeySet}(x : \mathbf{P}(\mathtt{Node})) : \mathbf{SetOf}(\mathbf{int}) \triangleq \lambda x.(x \to K)[\mathsf{NodeSet}(x)]
\end{array}
$$

**Fig. 2.** A set of recursive functions

**The Syntax of Specifications.** In our logic, specifications and verifications are written in the proof-in-code form. Formulas are written at program points, which are places before and after program statements. For a sequential statement $s_1; s_2$, the point after $s_1$ is just the point before $s_2$. All the program points are uniquely numbered. A program goes through program points during its execution. A formula at a program point means that each time when the program goes to this point, the formula is evaluated to **true**.

When we concern only one statement $s$ of the program under verification, the specification can be written as the following Hoare-triple.

$$\{i : P\}\ s\ \{j : Q\}$$

Here, $i$ and $j$ are respectively the program point numbers before and after $s$. We can write the specification as $\{P\}\ s\ \{Q\}$ if the point numbers are irrelevant.

A program point $j$ is said to *dominates* a point $i$ if the program must go through the point $j$ before it goes to the point $i$. For the language used in this paper, $j$ dominates $i$ if one of the following conditions holds. (I) $j = i$ or there is a point $k$ such that $j$ dominates $k$ and $k$ dominates $i$; (II) $j$ is before a statement $s$ and $i$ is a point in $s$ or the point after $s$.

Given two program points $i$ and $j$ such that $j$ dominates $i$, we can write $e@j$ at the program point $i$. It represents the value of $e$ evaluated at the point $j$ when the program was at the point $j$ the last time.

At any program point $i$, a program-point-specific expression $e@i$ equals to $e$ if $e$ is meaningful at this point. Because each program point is either before or after a statement, the following axiom PST specifies this property. In this axiom, $e$ and $e'$ represent two arbitrary expressions. It is required that $e$ and $e'$ are meaningful respectively at the point $i$ and $j$.

$$\text{(PST)} \qquad \{i : e = e@i\}\ s\ \{j : e' = e'@j\}$$

Program-point-specific expressions should be viewed as a naming convention for constant symbols. At a point $i$ other than $j$, a program-point-specific expression $e@j$ is treated as a constant symbol. We should not infer properties from the structure of $e@j$.

*Example 4.* The program points, together with some formulas, of the program in Fig. 1 are depicted in Fig. 3. The entrance program point and the exit point are respectively 1 and 10. The formula $\mathsf{isHBST}(\mathsf{root})$ at point 1 is the precondition of this program, while the formula at point 10 is the postcondition.

The formula at point 8 says that $\mathsf{k}$ is in the key set of the right sub-tree of $\mathsf{p}$ evaluated at point 7 if and only if $\mathsf{k}$ is in the key set of the initial binary tree.

At point 6, the property $\mathsf{pt} = (\mathsf{pt} \to l)@5$ holds because of the assignment $\mathsf{pt} := \mathsf{pt} \to l$. However, it does not imply $\mathsf{pt} = \mathsf{pt}@5 \to l$. To prove this property, we should prove that $(\mathsf{pt} \to l)@5 = \mathsf{pt}@5 \to l$ holds at point 5 using the axiom PST. This formula is not affected by the assignment, so it also holds at point 6. Now $\mathsf{pt} = (\mathsf{pt} \to l)@5$ and $(\mathsf{pt} \to l)@5 = \mathsf{pt}@5 \to l$ imply $\mathsf{pt} = \mathsf{pt}@5 \to l$ at 6.

Because point 5 does not dominate point 8, the formula $\mathsf{k} \in \mathsf{KeySet}(\mathsf{root})@5$ can not appear at point 8. □

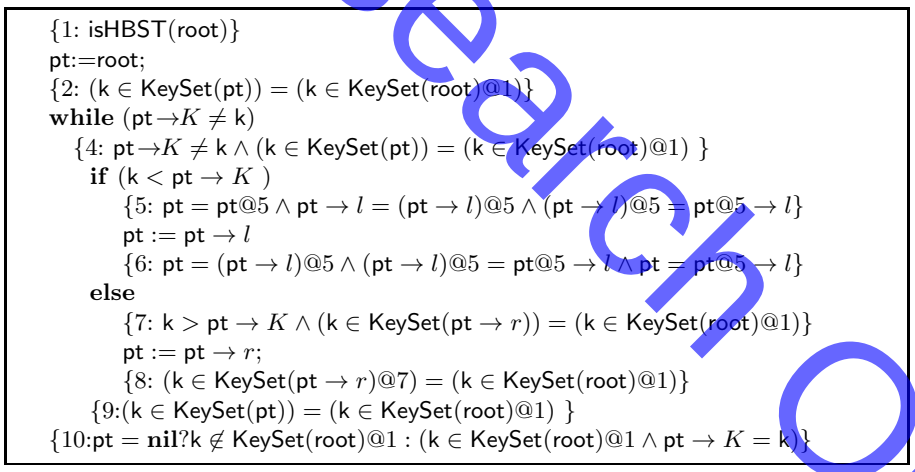


**Fig. 3.** A proof-in-code specification

## 3 The Memory Model and the Axioms about Memory Access Operators

In this section, we describe the memory model on which the programs execute. The memory consists of a set of addressed memory units. Each memory unit has a unique address and stores an integer, a boolean value, or a pointer. So the memory can be viewed as a map from addresses to **int**, **bool**, or **Ptr**.

Composite type data (either arrays or records) are stored in memory blocks. Each memory block is composed of sub-blocks and/or memory units for its component data. Each memory block has also a unique address. However, the memory model does not directly map block addresses to values. Instead, the block addresses are used to derive the addresses of its sub-blocks or units. Given the address $r$ of a memory block, the address of its components can be derived using expressions $\&r \to n$ (if $r$ refers to a record block and $n$ is a field name) or $\&(*r)[i]$ (if $r$ refers to an array block and $i$ is an integer). The values of $\&r \to n$ and $\&(*r)[i]$ depend only on the values of $r$ and $i$. They are irrelevant to the contents stored in the memory block.

*Example 5.* Suppose that a memory block with address $p$ stores a Node-typed data. This block is composed of three memory units for the fields $l$, $r$, and $K$. The addresses of these units are respectively $\&p \to l$, $\&p \to r$ and $\&p \to K$. □

| | | | |
|---|---|---|---|
| (DEREF-REF) | $*\&e = e$ | (REF-DEREF) | $e \neq \mathbf{nil} \Rightarrow \& * e = e$ |
| (PVAR-1) | $\&v \neq \mathbf{nil}$ | (PVAR-2) | $\&v_1 \neq \&v_2$ |
| (PVAR-3) | $\&v \neq \&r \to n$ | (PVAR-4) | $\&v \neq \&a[i]$ |
| (REC-1) | $r \neq \mathbf{nil} \Rightarrow \&r \to n \neq \mathbf{nil}$ | | |
| (REC-2) | $(r_1 \to n = r_2 \to n) \Leftrightarrow (r_1 = r_2)$ | (REC-3) | $r_1 \to n_1 \neq r_2 \to n_2$ |
| (ARR-1) | $a \neq \mathbf{nil} \wedge (0 \leq i < c) \Rightarrow \&((*a)[i]) \neq \mathbf{nil}$ | | |
| (ARR-2) | $(\&((*a_1)[i_1]) = \&((*a_2)[i_2])) \Leftrightarrow (a_1 = a_2 \wedge i_1 = i_2 \wedge 0 \leq i_1, i_2 < c)$ | | |
| (ARR-REC) | $\&a[i] \neq \&r \to n$ | | |

In these axioms, the type of $r, r_1, r_2$ are pointers to some record type and $n$, $n_1$ and $n_2$ are field names such that $n_1$ and $n_2$ are different. The type of $a, a_1, a_2$ are $\mathbf{ARR}(t, c)$ for some $t$ and $c$. $i, i_1, i_2$ are integers. The expression $e$ in DEREF-REF must be of the form $v, *e_1, e_1.n, e_1 \to n$ or $e_1[e_2]$.

**Fig. 4.** The axioms for memory layout and memory access

The axioms depicted in Fig. 4 are used to specify the addressing operator &, the memory access operator $*$, and the memory layouts for composite types.

The operators & and $*$ are inverse to each other. This is described by the axioms DEREF-REF and REF-DEREF.

Each program variable is assigned a unique memory block (or memory unit). Furthermore, the memory block (unit) is not a component of any other blocks. So we have the axioms PVAR-1, PVAR-2, PVAR-3 and PVAR-4.

Given a non-nil reference to a composite block, all the references to its sub-blocks or units are non-nil. So we have the axioms REC-1 and ARR-1. The axioms REC-2 and ARR-2 say that different components of a composite block

has different addresses. The axioms REC-3 and ARR-REC say that a component block/unit uniquely belongs to at most one enclosing memory block.

These axioms can be used to simplify expressions containing the addressing operator &. For example, the formula $\&\mathsf{pt} \rightarrow K \in (\lambda x.(\&x \rightarrow K)[\mathsf{NodeSet}(\mathsf{root})])$ can be simplified to an equivalent formula $\mathsf{pt} \in \mathsf{NodeSet}(\mathsf{root})$.

## 4 Memory Scopes of Expressions and Functions

### 4.1 Memory Scopes of Expressions

An expression $e$ may have different values before/after the execution of a program statement. However, the value of $e$ depends only on the contents stored in a finite set of memory units. This set can be expressed using another expression, called the *memory scope* of $e$, denoted as $\mathfrak{M}(e)$. We now show that $\mathfrak{M}(e)$ can be constructed syntactically.

If $e$ is of the form $f(e_1, \ldots, e_n)$, where $f$ is a function/operator other than $*, \&, [\,], ., \rightarrow$, the memory scope $\mathfrak{M}(e)$ is $\mathfrak{M}(e_1) \cup \ldots \cup \mathfrak{M}(e_n) \cup \mathfrak{M}(f)(e_1, \ldots, e_n)$, where $\mathfrak{M}(f)$ is the function to compute the memory scopes of applications of $f$.

1. If $f$ is an algebraic operator (e.g. +,-,...), a boolean operator, or other abstract operators, $\mathfrak{M}(f)$ is a constant function which always yields $\emptyset$.
2. If $f$ is a user-defined function (predicate), the definition of $\mathfrak{M}(f)$ can be derived syntactically from the definition of $f$, see next subsection.

The memory-scope-construction rules for other kinds of expressions are given in Fig. 5. The third column is used in the proof of an important property about memory scopes presented in Subsection 4.3.

Note that the memory scope of $e@i$ is $\emptyset$. The reason is that $e@i$ is viewed as a constant symbol of which the value is irrelevant to the current program state.

*Example 6.* Given a type $\mathbf{ARR}(\mathbf{ARR}(\mathbf{REC}((\mathbf{int}, f1), (\mathbf{int}, f2))), 100), 100)$ and a program variable $a$ of this type. The memory scope of $a[i][j].f1$ is constructed as follow.

$$\mathfrak{M}(a[i][j].f1) = \mathfrak{M}(\&a[i][j]) \cup \{\&a[i][j].f1\} = \mathfrak{M}(\&a[i]) \cup \{\&j\} \cup \{\&a[i][j].f1\}$$
$$= \mathfrak{M}(\&a) \cup \{\&i\} \cup \{\&j\} \cup \{\&a[i][j].f1\} = \{\&i\} \cup \{\&j\} \cup \{\&a[i][j].f1\}$$

This means that the value of $a[i][j].f1$ keeps unchanged if the contents stored in the memory units $\&i$, $\&j$, and $\&a[i][j].f1$ are not modified. □

### 4.2 Memory Scope of User-Defined Functions

Given a user-defined function $f$, we abuse the notation $\mathfrak{M}$ and use $\mathfrak{M}(f)$ to denote the name of the memory scope function of $f$. The formal parameters of $\mathfrak{M}(f)$ is the same as those of $f$. The return type of $\mathfrak{M}(f)$ is $\mathbf{SetOf}(\mathbf{Ptr})$. Intuitively speaking, $\mathfrak{M}(f)(x_1, \ldots, x_n)$ yields the set of memory units accessed during the evaluation of $f(x_1, \ldots, x_n)$. Let $f(x_1, \ldots, x_n) \triangleq e$ be the definition of $f$, the definition of $\mathfrak{M}(f)$ is as follow.

$$\mathfrak{M}(f)(x_1, \ldots, x_n) \triangleq \mathfrak{M}(e)$$

| Expressions | Memory Scopes | Memory Scopes of Memory Scopes |
|---|---|---|
| a constant $c$ | $\emptyset$ | $\emptyset$ |
| free variable $x$ | $\emptyset$ | $\emptyset$ |
| $e@i$ | $\emptyset$ | $\emptyset$ |
| $\&v$ | $\emptyset$ | $\emptyset$ |
| $\& * e$ | $\mathfrak{M}(e)$ | $\mathfrak{M}^2(e)$ |
| $\&e_1[e_2]$ | $\mathfrak{M}(\&e_1) \cup \mathfrak{M}(e_2)$ | $\mathfrak{M}^2(\&e_1) \cup \mathfrak{M}^2(e_2)$ |
| $\&e.n$ | $\mathfrak{M}(\&e)$ | $\mathfrak{M}^2(\&e)$ |
| $\&e \to n$ | $\mathfrak{M}(e)$ | $\mathfrak{M}^2(e)$ |
| $v$ | $\{\&v\}$ | $\emptyset$ |
| $*e$ | $\{e\} \cup \mathfrak{M}(e)$ | $\mathfrak{M}(e) \cup \mathfrak{M}^2(e)$ |
| $e_1.n$ | $\{\&e_1.n\} \cup \mathfrak{M}(\&e_1)$ | $\mathfrak{M}(\&e_1) \cup \mathfrak{M}^2(\&e_1)$ |
| $e_1 \to n$ | $\{\&e_1 \to n\} \cup \mathfrak{M}(e_1)$ | $\mathfrak{M}(e_1) \cup \mathfrak{M}^2(e_1)$ |
| $e_1[e_2]$ | $\{\&e_1[e_2]\} \cup \mathfrak{M}(\&e_1) \cup \mathfrak{M}(e_2)$ | $\mathfrak{M}(\&e_1) \cup \mathfrak{M}(e_2) \cup \mathfrak{M}^2(\&e_1) \cup \mathfrak{M}^2(e_2)$ |
| $e_0?e_1 : e_2$ | $\mathfrak{M}(e_0) \cup (e_0?\mathfrak{M}(e_1) : \mathfrak{M}(e_2))$ | $\mathfrak{M}^2(e_0) \cup \mathfrak{M}(e_0) \cup (e_0?\mathfrak{M}^2(e_1) : \mathfrak{M}^2(e_2))$ |
| $e_1 \wedge e_2$ | $\mathfrak{M}(e_1) \cup (e_1?\mathfrak{M}(e_2) : \emptyset)$ | $\mathfrak{M}^2(e_1) \cup \mathfrak{M}(e_1) \cup (e_1?\mathfrak{M}^2(e_2) : \emptyset)$ |
| $e_1 \vee e_2$ | $\mathfrak{M}(e_1) \cup (e_1?\emptyset : \mathfrak{M}(e_2))$ | $\mathfrak{M}^2(e_1) \cup \mathfrak{M}(e_1) \cup (e_1?\emptyset : \mathfrak{M}^2(e_2))$ |
| $\lambda x.e_1[e_2]$ | $\mathfrak{M}(e_2) \cup \bigcup(\lambda x.\mathfrak{M}(e_1)[e_2])$ | $\mathfrak{M}^2(e_2) \cup \mathfrak{M}(e_2) \cup \bigcup(\lambda x.\mathfrak{M}^2(e_1)[e_2])$ |
| $\forall x \in e_2.e_1$ | $\mathfrak{M}(e_2) \cup \bigcup(\lambda x.\mathfrak{M}(e_1)[e_2])$ | $\mathfrak{M}^2(e_2) \cup \mathfrak{M}(e_2) \cup \bigcup(\lambda x.\mathfrak{M}^2(e_1)[e_2])$ |

NOTE: $\mathfrak{M}^2(e)$ is an abbreviation for $\mathfrak{M}(\mathfrak{M}(e))$

**Fig. 5.** The memory scope for different forms of expressions

*Example 7.* Let $\mathfrak{M}(\mathsf{NodeSet})$ be $\mathsf{MNS}$. According to the definition of $\mathsf{NodeSet}$ in Fig. 2, the definition of $\mathsf{MNS}$ is as follow.

$$\mathsf{MNS}(x) \triangleq (x = \mathbf{nil})?\emptyset : \{\&x \to l, \&x \to r\} \cup \mathsf{MNS}(x \to l) \cup \mathsf{MNS}(x \to r)$$

The above definition is equivalent to the following one.

$$\mathsf{MNS}(x) \triangleq (\lambda y.(\&y \to l)[\mathsf{NodeSet}(x)]) \cup (\lambda y.(\&y \to r)[\mathsf{NodeSet}(x)])$$

$\mathsf{KeySet}$ and $\mathsf{isHBST}$ have the same memory scope function $M$ defined as

$$M(x) \triangleq \begin{array}{l}(\lambda y.(\&y \to K)[\mathsf{NodeSet}(x)]) \cup (\lambda y.(\&y \to l)[\mathsf{NodeSet}(x)]) \\ \cup (\lambda y.(\&y \to r)[\mathsf{NodeSet}(x)])\end{array}$$

From the above, the memory scope of the formula $\mathsf{pt} \in \mathsf{NodeSet}(\mathsf{root})$ is

$$\{\&\mathsf{pt}, \&\mathsf{root}\} \cup (\lambda y.(\&y \to l)[\mathsf{NodeSet}(\mathsf{root})]) \cup (\lambda y.(\&y \to r)[\mathsf{NodeSet}(\mathsf{root})])$$

It means that the formula keeps unchanged if the values of $\mathsf{pt}$, $\mathsf{root}$, and the fields $l$ and $r$ of the tree nodes keep unchanged. □

### 4.3 Two Properties of Memory Scopes

This section presents two important properties about memory scopes.

**Theorem 1.** *Let $e$ be an arbitrary expression and $x_1, \ldots, x_n$ are free variables in $e$. Given an assignment to these free variables and two program states $s_1$, $s_2$ such that $s_1$ and $s_2$ agree on all the memory units in $\mathfrak{M}(e)$. The expression $e$ is evaluated to the same value at $s_1$ and $s_2$.*

Because of the space limitation, we just give a brief proof.

1. If there is no user-defined function in $e$, the above conclusion can be proved by an induction on the length of $e$.
2. If there are user-defined functions in $e$ but none of these functions are recursive, we can expand the function applications with their definitions to get an equivalent expression $e'$. There is no user-defined function in $e'$, and $\mathfrak{M}(e)$ is a superset of $\mathfrak{M}(e')$. From 1, the conclusion is proved.
3. If there are recursively user-defined functions in $e$. Let $f$ be such a function defined as $f(\ldots) \triangleq \mathsf{EXP}(f)$, where $\mathsf{EXP}(f)$ is an expression containing $f$. Suppose that $f$ recursively called itself $n$ times during the evaluation of $e$ at the state $s_1$, we can define $n$ functions, $f_0, f_1, \ldots, f_n$ as $f_0 \triangleq \perp$, $f_1 \triangleq \mathsf{EXP}(f_0)$, $\ldots$, $f_i \triangleq \mathsf{EXP}(f_{i-1})$, $\ldots$, $f_n \triangleq \mathsf{EXP}(f_{n-1})$, and replace $f$ in $e$ with $f_n$. The derived expression $e'$ equals to $e$ at the state $s_1$, and $\mathfrak{M}(e')$ is a subset of $\mathfrak{M}(e)$ at the state $s_1$. From 2, $e'$ has the same value at the states $s_1$ and $s_2$. It also can be proved that $e'$ and $e$ evaluates to the same value on $s_2$. So $e$ evaluates to the same value at the states $s_1$ and $s_2$.

**Theorem 2.** *Let $e$ be an arbitrary expression $e$ such that $e$ is meaningful at a state $s$ for an assignment to the free variables in $e$, $\mathfrak{M}(\mathfrak{M}(e)) \subseteq \mathfrak{M}(e)$ is evaluated to* ***true*** *at $s$.*

The brief proof is as follow. Here, we use $\mathfrak{M}^2(e)$ as an abbreviation for $\mathfrak{M}(\mathfrak{M}(e))$.

1. If $e$ contains no user-defined function symbols, from the table in Fig. 5, we can prove this theorem by an induction on the length of $e$.
2. Let $f$ be a user-defined function defined as $f(x_1, \ldots, x_n) \triangleq e'$ and $e'$ contains no user-defined functions. $\mathfrak{M}(f(e_1, \ldots, e_n))$ is $\mathfrak{M}(e_1) \cup \ldots \cup \mathfrak{M}(e_n) \cup \mathfrak{M}(f)(e_1, e_2, \ldots, e_n)$; $\mathfrak{M}^2(f(e_1, \ldots, e_n))$ is $\mathfrak{M}^2(e_1) \cup \ldots \cup \mathfrak{M}^2(e_n) \cup \mathfrak{M}(e_1) \cup \ldots \cup \mathfrak{M}(e_n) \cup \mathfrak{M}^2(f)(e_1, e_2, \ldots, e_n)$. Note that $\mathfrak{M}(f)$ and $\mathfrak{M}(f)$ are respectively defined as $\mathfrak{M}(f)(x_1, \ldots, x_n) \triangleq \mathfrak{M}(e')$ and $\mathfrak{M}^2(f)(x_1, \ldots, x_n) \triangleq \mathfrak{M}^2(e')$. From 1, we can prove $\mathfrak{M}^2(f(e_1, \ldots, e_n)) \subseteq \mathfrak{M}(f(e_1, \ldots, e_n))$. So the theorem holds if the functions in $e$ are not defined with other user-defined functions.
3. We can prove by an induction that $\mathfrak{M}^2(f(e_1, \ldots, e_n)) \subseteq \mathfrak{M}(f(e_1, \ldots, e_n))$ holds for a user-defined non-recursive function $f$ based on 2. So the theorem holds for expressions containing non-recursive functions.
4. Now we prove the case of recursive functions. From the definition of $\mathfrak{M}$, we have the following fact: let $f$ be a user-defined function symbol in an expression $e$, the functions $f$, $\mathfrak{M}(f)$ and $\mathfrak{M}^2(f)$ are applied to same real parameters in $e$, $\mathfrak{M}(e)$ and $\mathfrak{M}^2(e)$. Furthermore, in $\mathfrak{M}(e)$ and $\mathfrak{M}^2(e)$, the counterparts of the conditional sub-expressions in $e$ have the same guard. So during the evaluation of $e$, $\mathfrak{M}(e)$ and $\mathfrak{M}^2(e)$, $f$ recursively call itself if and only if $\mathfrak{M}(f)$ and $\mathfrak{M}^2(f)$ call themselves.

Let $f$, $\mathfrak{M}(f)$ and $\mathfrak{M}^2(f)$ be functions respectively defined as $f(x_1,\ldots,x_n) \triangleq e'$, $\mathfrak{M}(f)(x_1,\ldots,x_n) \triangleq \mathfrak{M}(e')$ and $\mathfrak{M}^2(f)(x_1,\ldots,x_n) \triangleq \mathfrak{M}^2(e')$. Suppose that $f$ recursively calls itself for $n$ times during the evaluation of $e$ on a state $s$, the fact above means that $\mathfrak{M}(f)$ and $\mathfrak{M}^2(f)$ also recursively call themselves for $n$ times during the evaluation of $\mathfrak{M}(e)$ and $\mathfrak{M}^2(e)$. So we can introduce $n$ new functions $f_0, f_1, \ldots, f_n$ defined as $f_0 \triangleq \bot$, $f_1 \triangleq \mathsf{EXP}(f_0)$, ..., $f_i \triangleq \mathsf{EXP}(f_{i-1})$, ..., $f_n \triangleq \mathsf{EXP}(f_{n-1})$, where $\mathsf{EXP}(f_i)$ means the expression derived by replacing $f$ with $f_i$ in $e'$. It can be proved that $f(e_1,\ldots,e_n) = f_n(e_1,\ldots,e_n)$, $\mathfrak{M}(f)(e_1,\ldots,e_n) = \mathfrak{M}(f_n)(e_1,\ldots,e_n)$, and $\mathfrak{M}^2(f)(e_1,\ldots,e_n) = \mathfrak{M}^2(f_n)(e_1,\ldots,e_n)$ on the state $s$. Because $f_i$s are not recursive, we prove that $\mathfrak{M}^2(f(e_1,\ldots,e_n)) \subseteq \mathfrak{M}(f(e_1,\ldots,e_n))$. So the theorem holds for expressions containing recursive functions.

In our logic, we use the following axiom to describe this property.

(SCOPE-SHRINK) $\mathfrak{M}(\mathfrak{M}(e)) \subseteq \mathfrak{M}(e)$ *Note: e must be meaningful*

This axiom is important for local reasoning. We will discuss this in Section 7.

## 5 The Axioms and Proof Rules of Program Statements

In this section, we present the axioms and proof rules to specify the effects of program statements. There are three axioms for primitive statements and three proof rules for control flow statements. They are all presented in Fig. 6.

For an assignment $e_1 := e_2$, let $i, j$ respectively be the program points before/after this statement. It is required that $\&e_1$ evaluates to a non-nil pointer at $i$. At the program point $j$, the memory unit referred by $\&e_1$ stores the value of $e_2$ evaluated at $i$. Furthermore, if a formula holds at the point $i$, and $\&e_1$ is not in the memory scope of this formula, the formula still holds at the point $j$. This is specified by the axiom ASSIGN.

*Example 8.* Considering the assignment $\mathsf{pt} := \mathsf{pt} \to l$ in Fig. 3. Let **Prop** be the formula $\mathsf{pt}@5 \neq \mathbf{nil} \wedge (\mathsf{pt} \to l)@5 = \mathsf{pt}@5 \to l \wedge \&\mathsf{pt} = (\&\mathsf{pt})@5$. $\mathfrak{M}(\mathbf{Prop})$ is $\{\&(\mathsf{pt}@5) \to l\}$. Substituting $\rho$ with **Prop** in the axiom ASSIGN, we have

$\{5 : \mathbf{Prop} \wedge (\&\mathsf{pt} \notin \{\&(\mathsf{pt}@5) \to l\}) \wedge (\&\mathsf{pt} \neq \mathbf{nil})\}$ $\mathsf{pt} := \mathsf{pt} \to l$
$\{6 : \mathbf{Prop} \wedge *((\&\mathsf{pt})@5) = (\mathsf{pt} \to l)@5\}$

From the axioms PVAR-1, PVAR-3, PST, and the proof rules CONSEQ and CONJ, we have $\{5 : \mathsf{pt} \neq \mathbf{nil}\}$ $\mathsf{pt} := \mathsf{pt} \to l$ $\{6 : \mathsf{pt} = \mathsf{pt}@5 \to l\}$. □

For an allocation statement $e_1 := \mathsf{alloc}(t)$, let $i, j$ respectively be the program points before/after this statement. It is required that $\&e_1$ evaluates to a non-nil pointer at the point $i$. After the execution, the memory unit referred by $(\&e_1)@i$ stores a reference to a newly allocated memory block. This memory block is unreachable at the point $i$. So $*((\&e_1)@i) \notin e_2@i$ holds at the point $j$ for any expressions $e_2$ if $e_2$ is meaningful at $i$. This allocation statement modifies only the memory unit referred by $(\&e_1)@i$ and the memory block newly allocated (this block is unreachable at the point $i$). If an assertion $\rho$ holds at the point $i$

and $(\&e_1)@i$ is not in the memory scope of $\rho$, $\rho$ still holds at the point $j$. This is specified by the axiom ALLOC. In this axiom, $\texttt{Init}(x)$ is an abbreviation for the assertion that all the pointers stored in the block referred by $x$ are set to **nil**. For example, if the type of $x$ is $\mathbf{P}(\mathsf{Node})$, $\texttt{Init}(x)$ is $x \to l = \mathbf{nil} \wedge x \to r = \mathbf{nil}$.

*Example 9.* Considering the statement $\mathsf{t} := \mathsf{alloc}(\mathsf{Node})$. From the axiom ALLOC, substituting $\rho$ and $e_2$ respectively with $\mathsf{isHBST}(\mathsf{rt}) \wedge (\&\mathsf{t} = (\&\mathsf{t})@i)$ and $\mathsf{NodeSet}(\mathsf{rt})$, we have

$$\begin{array}{l}\{i : \mathsf{isHBST}(\mathsf{rt}) \wedge (\&\mathsf{t} = (\&\mathsf{t})@i) \wedge \&\mathsf{t} \notin \mathfrak{M}(\mathsf{isHBST}(\mathsf{rt})) \wedge \&\mathsf{t} \neq \mathbf{nil}\} \\ \quad \mathsf{t} := \mathsf{alloc}(\mathsf{Node}); \\ \{j : \mathsf{isHBST}(\mathsf{rt}) \wedge (\&\mathsf{t} = (\&\mathsf{t})@i) \wedge ((*((\&\mathsf{t})@i) \notin \mathsf{NodeSet}(\mathsf{rt})@i) \\ \qquad \wedge (*((\&\mathsf{t})@i) \neq \mathbf{nil}) \wedge (*((\&\mathsf{t})@i) \to l = \mathbf{nil}) \wedge (*((\&\mathsf{t})@i) \to r = \mathbf{nil})\}\end{array}$$

From the axioms DEREF-REF, PVAR-1, PVAR-3, PST, and the proof rules CONSEQ and CONJ, this specification can be simplified to

$$\begin{array}{l}\{i : \mathsf{isHBST}(\mathsf{rt})\} \quad \mathsf{t} := \mathsf{alloc}(\mathsf{Node}); \\ \{j : \mathsf{isHBST}(\mathsf{rt}) \wedge (\mathsf{t} \notin \mathsf{NodeSet}(\mathsf{rt})@i) \wedge (\mathsf{t} \neq \mathbf{nil}) \wedge (\mathsf{t} \to l = \mathbf{nil}) \wedge (\mathsf{t} \to r = \mathbf{nil})\}\end{array}$$

□

The axiom for the `skip` statement, the proof rules for control-flow statements, the consequence rule, and the conjunction rule are depicted in Fig. 6. They are same as the ones in Hoare Logic.

$$\text{(SKIP)} \quad \{q\}\ \texttt{skip}\ \{q\} \qquad\qquad \text{(ASSIGN)} \quad \begin{array}{l}\{i : \rho \wedge (\&e_1 \notin \mathfrak{M}(\rho)) \wedge (\&e_1 \neq \mathbf{nil})\} \\ \quad e_1 := e_2 \\ \{j : \rho \wedge (*((\&e_1)@i) = e_2@i)\}\end{array}$$

$$\text{(ALLOC)} \quad \begin{array}{l}\{i : \rho \wedge (\&e_1 \notin \mathfrak{M}(\rho)) \wedge (\&e_1 \neq \mathbf{nil})\} \\ \quad e_1 := \mathsf{alloc}(t) \\ \{j : \rho \wedge (*((\&e_1)@i) \neq \mathbf{nil}) \wedge (*((\&e_1)@i) \notin e_2@i) \wedge \texttt{Init}(*((\&e_1)@i))\}\end{array}$$

$$\boxed{\text{IF}}\ \frac{\{p \wedge e\}s_1\{q\} \quad \{p \wedge \neg e\}s_2\{q\}}{\{p\}\ \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2\ \{q\}} \qquad \boxed{\text{WHILE}}\ \frac{\{p \wedge e\}\ s\ \{p \wedge (e \vee \neg e)\}}{\{p\}\ \ \texttt{while}\ (e)\ s\ \{\neg e \wedge p\}}$$

$$\boxed{\text{SEQ}}\ \frac{\{p\}s_1\{q\} \quad \{q\}s_2\{r\}}{\{p\}\ s_1; s_2\ \{r\}}$$

$$\boxed{\text{CONSEQ}}\ \frac{\{p\}s\{q\} \quad p' \Rightarrow p \quad q \Rightarrow q'}{\{p'\}\ s\ \{q'\}} \qquad \boxed{\text{CONJ}}\ \frac{\{p\}\ s\ \{q\} \quad \{p'\}\ s\ \{q'\}}{\{p \wedge p'\}\ s\ \{q \wedge q'\}}$$

**Fig. 6.** The axioms and proof rules for program statements

## 6 The Weakest-Preconditions of Assignments

In this section, we will show how to compute the weakest precondition of an assignment for a postcondition that contains no user-defined function.

| $e$ | $\mathsf{WP}(e)$, the expression equivalent to $e$ before $e_1 = e_2$ |
|---|---|
| a const or a quantified variable | $e$ |
| $e@k$ for some point $k$ $(k \neq j)$ | $e@k$ |
| $\&v$ | $\&v$ |
| $*e'$ | $(\mathsf{WP}(e') \neq (\&e_1)@i)?(*\mathsf{WP}(e')) : (e_2@i)$ |
| $v$ | $(\mathsf{WP}(\&v) \neq (\&e_1)@i)?(*\mathsf{WP}(\&v)) : (e_2@i)$ |
| $e'.n$ | $(\mathsf{WP}(\&e'.n) \neq (\&e_1)@i)?(*\mathsf{WP}(\&e'.n)) : (e_2@i)$ |
| $e' \to n$ | $(\mathsf{WP}(\&e' \to n) \neq (\&e_1)@i)?(*\mathsf{WP}(\&e' \to n)) : (e_2@i)$ |
| $e'[e'']$ | $(\mathsf{WP}(\&e'[e'']) \neq (\&e_1)@i)?(*\mathsf{WP}(\&e'[e''])) : (e_2@i)$ |
| $e'$ op $e''$ | $\mathsf{WP}(e')$ op $\mathsf{WP}(e'')$ |
| op $e'$ (op *is not* $*$) | op $\mathsf{WP}(e')$ |
| $e_0?e' : e''$ | $\mathsf{WP}(e_0)?\mathsf{WP}(e') : \mathsf{WP}(e'')$ |
| $\&(e'.n)$ | $\&(\mathsf{WP}(\&e') \to n)$ |
| $\&(e' \to n)$ | $\&(\mathsf{WP}(e') \to n)$ |
| $\&(e'[e''])$ | $\&((*\mathsf{WP}(\&e'))[\mathsf{WP}(e'')])$ |
| $\lambda x.e'[e'']$ | $\lambda x.\mathsf{WP}(e')[\mathsf{WP}(e'')]$ |
| $\forall x \in e'.e''$ | $\forall x \in \mathsf{WP}(e').\mathsf{WP}(e'')$ |

**Fig. 7.** The rules to construct $\mathsf{WP}(e)$

Given an assignment $e_1 := e_2$, and $i$, $j$ be the program points before/after this assignment respectively. The program state at $i$ is different only with the state at $j$ on the memory unit $(\&e_1)@i$. For any address $x$ of a memory unit, the value of $((x \neq (\&e_1)@i)? * x : e_2@i)$ at the point $i$ equals to the value of $*x$ at the point $j$.

For an arbitrary formula $e$, we can construct the weakest precondition of the assignment $e_1 := e_2$ for $e$ according to rules depicted in Fig. 7. The basic idea of these rules is that for each expression $e$ with an L-value, we first construct an expression $\mathsf{WP}(\&e)$, the value of which at $i$ equals to the value of $\&e$ at $j$; then $\mathsf{WP}(e)$ is constructed as $((\mathsf{WP}(\&e) \neq (\&e_1)@i)? * \mathsf{WP}(\&e) : e_2@i)$. From the discussion above, the value of $\mathsf{WP}(e)$ at $i$ equals to the value of $*\&e$ (equivalent to $e$ by the axiom DEREF-REF) at $j$.

By an induction on the length of the expression $e$, we can prove that the value of $e$ at $j$ equals to the value of $\mathsf{WP}(e)$ at $i$. So a formula $e$ holds at the point $j$ if and only if $\mathsf{WP}(e)$ holds at $i$. Theorem 3 says that $\mathsf{WP}(e)$ is a precondition of $e_1 := e_2$ for $e$ in Scope Logic.

**Theorem 3.** *Given an assignment $e_1 := e_2$, and $i, j$ be respectively the program points before/after this assignment. Let $e$ be a formula containing no user-defined functions and no program-point-specific sub-expression of the form $e@j$, the following specification can be proved in Scope Logic.*

$$\{\mathsf{WP}(e) \wedge \&e_1 \neq \mathbf{nil}\}\ e_1 := e_2\ \{e\}$$

*Proof.* From the axiom ASSIGN, we have

$$\{\mathsf{WP}(e) \wedge \&e_1 \notin \mathfrak{M}(\mathsf{WP}(e)) \wedge \&e_1 \neq \mathbf{nil}\}\ e_1 := e_2 \{\mathsf{WP}(e) \wedge *((\&e_1)@i) = e_2@i\}$$

By a mathematical induction on the length of $e$, we can show that $\&e_1 \notin \mathfrak{M}(\mathsf{WP}(e))$ holds at the point $i$ for any $e$. So we have

$$\{\mathsf{WP}(e) \wedge \&e_1 \neq \mathbf{nil}\}\ e_1 := e_2\ \{\mathsf{WP}(e) \wedge *((\&e_1)@i) = e_2@i\}$$

Because $(*((\&e_1)@i) = e_2@i)$ implies $\forall x.(((x = (\&e_1)@i)?e_2@i : *(x)) = *x)$, we can prove that $\mathsf{WP}(e) \wedge (*((\&e_1)@i) = e_2@i) \Rightarrow e$ holds at the point $j$. From the proof rule CONSEQ, we have $\{\mathsf{WP}(e) \wedge \&e_1 \neq \mathbf{nil}\}\ e_1 := e_2\ \{e\}$. QED.

Usually there are many conditional expressions and operators $\&$, $@i$ in $\mathsf{WP}(e)$. We can remove $@i$ in $\mathsf{WP}(e)$ using the axiom PST. Most of the operator $\&$ in $\mathsf{WP}(e)$ can be eliminated using the axioms in Section 3. For the conditional sub-expressions $(e' \neq (\&e_1)@i)? * e' : e_2@i$ in $\mathsf{WP}(e)$, we can automatically simplify them when $(e' \neq (\&e_1)@i)$ is either unsatisfiable or a tautology according to the axioms in Section 3.

*Example 10.* Here are some examples of the weakest precondition calculation.

1. Let $\{i : \}\ x := a+b\ \{j : x > y\}$ be an unfinished specification. Applying $\mathsf{WP}$, it yields $((\&x \neq (\&x)@i)? * (\&x) : (a + b)@i) > (((\&y \neq (\&x)@i)? * (\&y) : (a + b)@i)$. Simplifying it, we have $\{a + b > y\}\ x := a + b\ \{x > y\}$.
2. Let $\{i : \}\ a[a[2]] := 3\ \{a[a[2]] = 3\}$ be an unfinished specification, where $a$ is a program variable with type $\mathbf{ARR}(\mathbf{int}, 100)$. Applying $\mathsf{WP}$, it yields the precondition $((\&a[\mathsf{IND}] \neq (\&a[a[2]])@i)? * (\&a[\mathsf{IND}]) : 3@i) = 3$, where $\mathsf{IND}$ is the abbreviation for $(\&a[2] \neq (\&a[a[2]])@i)? * (\&a[2]) : 3@i$. The precondition can be automatically simplified to $((((2 \neq a[2])?a[2] : 3) \neq a[2])?a[(2 \neq a[2])?a[2] : 3] : 3) = 3$. Using an SMT solver, it can be easily proven that this formula is equivalent to $a[2] \neq 2 \vee a[3] = 3$. So we have $\{a[2] \neq 2 \vee a[3] = 3\}\ a[a[2]] := 3\ \ \{a[a[2]] = 3\}$.
3. Let $\{i : \}\ a[\mathsf{n}] := \mathsf{tmp}\ \{j : \forall x \in [0..\mathsf{n}].a[x] >= 0\}$ be an unfinished specification, $[0..\mathsf{n}]$ is the set of integers from 0 to $\mathsf{n}$. Applying $\mathsf{WP}$, we get $\forall x \in [0..(\&\mathsf{n} \neq (\&a[\mathsf{n}])@i? * (\&\mathsf{n}) : \mathsf{tmp}@i)].((\&a[x] \neq (\&a[\mathsf{n}])@i? * (\&a[x]) : \mathsf{tmp}@i) >= 0)$. Simplifying it, we have the following specification. $\{i : \forall x \in [0..\mathsf{n}].((x = n?\mathsf{tmp} : a[x]) >= 0)\}a[n] := \mathsf{tmp}\{j : \forall x \in [0..\mathsf{n}].a[x] >= 0\}$.

□

## 7 Supporting Local Reasoning

To support local reasoning, a specification should be in the following form.

$$\{\rho \wedge (\mathfrak{M}(\rho) \cap e = \emptyset) \wedge pre\}\ s\ \{\rho \wedge post\}$$

where $\rho$ is an assertion variable representing an arbitrary assertion, and $e$ is (an over-approximation of) the set of memory units modified by the statement $s$.

We say that such specifications are in the *memory-modification-bounded* form because the expression $e$ bounds the set of memory units modified by $s$. The assertion variable $\rho$ in the specification can be substituted with a formula $e'$. Simultaneously, the memory scope of $\rho$, i.e. $\mathfrak{M}(\rho)$, is substituted with $\mathfrak{M}(e')$. For example, in the proof of Theorem 4, $\rho$ and $\mathfrak{M}(\rho)$ are respectively substituted with $e' \wedge \rho \wedge (\mathfrak{M}(\rho) \cap e@i = \emptyset)$ and $\mathfrak{M}(e') \cup \mathfrak{M}(\rho) \cup \mathfrak{M}(\mathfrak{M}(\rho))$.

Theorem 4 shows that if we get a specification in the memory-modification-bounded form, we can expand the specification with a formula $e'$, if $\mathfrak{M}(e')$ is disjoint with the memory bound $e$.

**Theorem 4.** *Let $\rho$ be an assertion variable.*

$$\frac{\begin{array}{c} pre \wedge e' \Rightarrow \mathfrak{M}(e') \cap e = \emptyset \\ \{i : \rho \wedge (\mathfrak{M}(\rho) \cap e = \emptyset) \wedge pre\}\ s\ \{j : \rho \wedge post\} \end{array}}{\{i : \rho \wedge (\mathfrak{M}(\rho) \cap e = \emptyset) \wedge (e' \wedge pre)\}\ s\ \{j : \rho \wedge (e' \wedge post)\}}$$

*Proof.* First, we substitute $\rho$ in the second premise with $e' \wedge \rho \wedge (\mathfrak{M}(\rho) \cap e@i = \emptyset)$. The memory scope of this formula is $\mathfrak{M}(e') \cup \mathfrak{M}(\rho) \cup \mathfrak{M}(\mathfrak{M}(\rho))$. From the axiom SCOPE-SHRINK, it equals $\mathfrak{M}(e') \cup \mathfrak{M}(\rho)$. So we have

$$\begin{array}{l} \{(e' \wedge \rho \wedge (\mathfrak{M}(\rho) \cap e@i = \emptyset)) \wedge ((\mathfrak{M}(e') \cup \mathfrak{M}(\rho)) \cap e = \emptyset) \wedge pre\} \\ \quad s \\ \{(e' \wedge \rho \wedge (\mathfrak{M}(\rho) \cap e@i = \emptyset)) \wedge post\} \end{array}$$

From the first premise and the proof rules CONSEQ, PST, we have

$$\{\rho \wedge (\mathfrak{M}(\rho) \cap e = \emptyset) \wedge (e' \wedge pre)\}\ s\ \{\rho \wedge (e' \wedge post)\}$$

QED.

*Example 11.* Suppose a program **REORDER** reorders the nodes of a singly-linked list $p$, and the derived list is pointed to by $q$. The program modifies only the field *link* of the nodes. Let **MemBnd** be $(\{\&\mathsf{q}\} \cup \lambda x.(\&x \rightarrow link)[\mathsf{ListNodes(p)}])$. The specification can be written as

$$\begin{array}{l} \{i : \rho \wedge (\mathfrak{M}(\rho) \cap \mathbf{MemBnd} = \emptyset) \wedge \mathsf{isSList(p)}\} \quad \mathbf{REORDER} \\ \{j : \rho \wedge \mathsf{isSList(q)} \wedge \mathsf{ListNodes(q)} = \mathsf{ListNodes(p)}@i\} \end{array}$$

Let **Prop** be $\forall x \in (\mathsf{ListNodes(p)}@i).(x \rightarrow D > 0)$. $\mathfrak{M}(\mathbf{Prop})$ is $\lambda x.(\&x \rightarrow D)[\mathsf{ListNodes(p)}@i]$, which is disjoint with **MemBnd**. From Theorem 4, we have

$$\begin{array}{l} \{i : \rho \wedge (\mathfrak{M}(\rho) \cap \mathbf{MemBnd} = \emptyset) \wedge \mathbf{Prop} \wedge \mathsf{isSList(p)}\} \quad \mathbf{REORDER} \\ \{j : \rho \wedge \mathbf{Prop} \wedge \mathsf{isSList(q)} \wedge \mathsf{ListNodes(q)} = \mathsf{ListNodes(p)}@i\} \end{array}$$

Using the proof rules PST and CONSEQ, we have

$$\begin{array}{l} \{i : \rho \wedge (\mathfrak{M}(\rho) \cap \mathbf{MemBnd} = \emptyset) \wedge \mathbf{Prop} \wedge \mathsf{isSList(p)}\} \quad \mathbf{REORDER} \\ \{j : \rho \wedge \forall x \in (\mathsf{ListNodes(q)}).(x \rightarrow D > 0) \wedge \mathsf{isSList(q)} \wedge \\ \qquad \mathsf{ListNodes(q)} = \mathsf{ListNodes(p)}@i\} \end{array}$$

□

To apply the proof rules WHILE and SEQ, the post-conditions of sub-statements must also be in the form $\rho \wedge (\mathfrak{M}(\rho) \cap e = \emptyset) \wedge property$. Theorem 5 can be used to derive such post-conditions.

**Theorem 5.** *Let $\rho$ be an assertion variable.*

$$\frac{\{i : \rho \wedge (\mathfrak{M}(\rho) \cap e = \emptyset) \wedge pre\}\ s\ \{j : \rho \wedge post\}}{\{i : \rho \wedge (\mathfrak{M}(\rho) \cap e = \emptyset) \wedge pre\}\quad s\quad \{j : \rho \wedge (\mathfrak{M}(\rho) \cap e@i = \emptyset) \wedge post\}}$$

*Proof.* Note that the memory scope of $\rho \wedge (\mathfrak{M}(\rho) \cap e@i = \emptyset)$ is $\mathfrak{M}(\rho) \cup \mathfrak{M}(\mathfrak{M}(\rho))$, which equals to $\mathfrak{M}(\rho)$ from the axiom SCOPE-SHRINK. So $\rho \wedge (\mathfrak{M}(\rho) \cap e@i = \emptyset)$ implies that its memory scope is disjoint with $e@i$. Substitute $\rho$ in the premises with $\rho \wedge (\mathfrak{M}(\rho) \cap e@i = \emptyset)$, and apply the proof rule PST, CONSEQ, we can get

$$\{i : \rho \wedge (\mathfrak{M}(\rho) \cap e = \emptyset) \wedge pre\}\ s\ \{j : \rho \wedge (\mathfrak{M}(\rho) \cap e@i = \emptyset) \wedge post\}$$

QED.

This theorem shows that $\mathfrak{M}(\rho)$ is still disjoint with $e@i$ on the post-state. We may replace $e@i$ with some other expressions $e'$ if $post \Rightarrow e' \subseteq e@i$.

*Example 12.* Let **ST** be $\mathsf{tmp} := \mathsf{p} \to link; \mathsf{p} \to link = \mathsf{q}; \mathsf{q} := \mathsf{p}; \mathsf{p} = \mathsf{tmp};$, which is the loop-body of the program that reverses a singly-linked list. Let MSet be $\{\&\mathsf{p}, \&\mathsf{q}, \&\mathsf{tmp}\} \cup \lambda x.(\&x \to link)[\mathsf{ListNodes(first)}@1]$, and Prop be

$$\mathsf{IsList(p)} \wedge \mathsf{IsList(q)} \wedge (\mathsf{ListNodes(p)} \cap \mathsf{ListNodes(q)} = \emptyset) \wedge$$
$$(\mathsf{ListNodes(p)} \cup \mathsf{ListNodes(q)} = \mathsf{ListNodes(first)}@1).$$

The following specification can be proved.

$$\{i : \rho \wedge (\mathfrak{M}(\rho) \cap \mathsf{MSet} = \emptyset) \wedge \mathsf{Prop}\}\ \mathbf{ST}\ \{j : \rho \wedge \mathsf{Prop} \wedge \mathsf{MSet}@i = \mathsf{MSet}\}$$

From Theorem 5 and the rule CONSEQ, we have

$$\{i : \rho \wedge (\mathfrak{M}(\rho) \cap \mathsf{MSet} = \emptyset) \wedge \mathsf{Prop}\}\ \mathbf{ST}\ \{j : \rho \wedge (\mathfrak{M}(\rho) \cap \mathsf{MSet} = \emptyset) \wedge \mathsf{Prop}\}$$

Now, $\rho \wedge (\mathfrak{M}(\rho) \cap \mathsf{MSet} = \emptyset) \wedge \mathsf{Prop}$ can be used as a loop invariant. □

## 8 The Tool

An interactive tool has been implemented to support code verification using Scope Logic. Users can input formulas at program points and then prove them. A formula holds at a point if (1) it is logically implied by other proved formulas in the same program point; (2) or the formula holds at the predecessor point(s) and it is not affected by program statement; (3) or it is a natural result of the program execution, e.g. the condition expression of an if-statement holds at the point before its then branch.

This tool also supports some automatic verification mechanisms like weakest precondition calculation and data-flow analysis techniques. However, assertion variables and local reasoning have not been supported yet.

Many examples, including the Schorre-Waite algorithm, several array-sorting algorithms, singly-linked list manipulations, binary search tree manipulations, and a topological sorting algorithm have been verified using this tool.

For more technical details of this tool, please visit the web page of this tool: http://seg.nju.edu.cn/SCL.html.

## 9 Related Works and Conclusions

In this paper, we present an extension to Hoare Logic for programs with pointers and recursive data structures. Formulas augmented with recursively defined functions (predicates) are used to deal with recursive data structures. The logic can specify and verify relations between different program points using program-point-specific expressions. Our logic also supports local reasoning, which is important for verification of real programs. The weakest precondition of assignments for postconditions containing no user-defined function is well supported.

In Separation Logic, a new logical connective $*$ (separation conjunction) is introduced to specify that two assertions assert properties of two disjoint heaplets. However, this new logical connective makes it difficult to use conventional logical tools and techniques, like SMT solvers. To solve this problem, implicit dynamic frame (IDF) [10] uses RAS to compute the footprint of an assertion (i.e. the locations required to be accessed). Though [9] presents a method to convert a certain fragment of Separation Logic specifications into representations in IDF so that they can be verified using conventional logic tools. However, recursive predicates (which are important to specify recursive data structures and their properties) are not supported yet. The memory scope symbol $\mathfrak{M}$ in Scope Logic is similar to RAS in IDF. The main difference is the way in which footprints (memory scopes) of recursive predicates are dealt with. RAS in IDF uses some axioms to specify the relation between the footprint functions and the corresponding predicates. These axioms refer to the global heap directly. In Scope Logic, the explicit definitions of memory scope functions can be syntactically constructed based on the predicate (or function) definitions using $\mathfrak{M}$. Using explicit definitions, people can do better on reasoning about memory scopes. For example, we can find that several predicates (functions) share same memory scope functions. Another benefit of using explicit definitions of memory scope functions is that the global heap is not referred in code verifications using Scope Logic. Verification conditions generated in [10] contain a global heap, many store operators on the heap, and universal quantifiers over the addresses. From our experience, it is difficult to verify such complicated formulas using SMT solvers.

In [8], memory unit sets relevant to assertions are specified using 'region's. For a recursively defined predicate, people must define its region together with the definition of the predicate. Ghost variables and fields are instrumented into programs such that assertions can refer regions explicitly. In Scope Logic, the memory scope of a recursive predicate (function) is treated as an intrinsic attribute of the predicate (function). Memory scopes (similar to 'region's) of assertions, expressions, (recursive) predicates and functions are constructed syntactically. One advantage of our method is that ghost variables and fields are avoided. Another advantage is that it is simpler to first define a recursive predicate (function) and then construct its memory scope function syntactically.

Our logic does not support memory-deallocation statements now. Another disadvantage is that we use the conventional FOL as the base logic, people must carefully avoid meaningless (not-welldefined) expressions in code verifications. To solve this problem, we will try to find a method to generate meaningful-conditions for

expressions. We will also try to extend our logic to deal with more sophisticated program structures like procedure definitions/calls, classes/objects, function pointers, etc.

## References

1. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)
2. Burstall, R.M.: Some techniques for proving correctness of programs which alter data structures. In: Machine Intelligence, vol. 7, pp. 23–50. Edinburgh University Press, Edinburgh (1972)
3. Cook, S.A., Oppen, D.C.: An assertion language for data structures. In: Conference Record of 2nd ACM Symposium on Priciples of Programming Languages, New York, pp. 160–166 (1975)
4. Morris, J.M.: A general axiom of assignment; assignment and linked data structures; a proof of the Schorr-Waite algorithm. In: Theoretical Foundations of Programming Methodology, pp. 25–51. D. Reidel, Dordrecht (1982)
5. Reynolds, J.C.: An overview of separation logic. In: Proceedings of Verified Software: Theories, Tools, Experiments 2005, Zurich, Switzerland, October 10-13 (2005) Revised Selected Papers and Discussions
6. Yang, H.: An example of local reasoning in BI pointer logic: The Schorr-Waite graph marking algorithm. In: Henglein, F., Hughes, J., Makholm, H., Niss, H. (eds.) SPACE 2001: Informal Proceedings of Workshop on Semantics, Program Analysis and Computing Environments for Memory Management, pp. 41–68. IT University of Copenhagen (2001)
7. Jones, C.B., Middelburg, C.A.: A typed logic of partial functions reconstructed classically. Acta Inform 31(5), 399–430 (1994)
8. Banerjee, A., Naumann, D.A., Rosenberg, S.: Regional logic for local reasoning about global invariants. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 387–411. Springer, Heidelberg (2008)
9. Parkinson, M.J., Summers, A.J.: The Relationship Between Separation Logic and Implicit Dynamic Frames. Logical Methods in Computer Science 8(3) (2012)
10. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames. ACM Trans. on Programgramming Language and Systems 34(1) (2012)