

The Clike Programming Language Specification

Technique Report

Zhaopeng Li Yiyun Chen Zhong Zhuang Simin Yang

Department of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui 230026, China
Software Security Lab., Suzhou Institute for Advanced Study, University of Science and Technology of China,
Suzhou, Jiangsu 215123, China
zpli@mail.ustc.edu.cn

Abstract

Clike is a C-like programming languages. In this article, the definition of Clike is presented. It is the source language for the CComp certifying compiler. Some features of C language are not directly supported due to the use of separation logic (as the program logic) and the consideration of proof generation.

Keywords Software Safety, Hoare Logic, Separation Logic, Proof-Carrying Code, Certifying Compiler

1. Introduction

C language is widely used in system and application software development. One of the characteristics of C language is allowing low-level access to memory by converting machine addresses to typed pointers. Pointers to data improve performance for repetitive operations such as traversing list and tree structures. Because pointers allow largely unprotected access to memory addresses, there are risks associated with using them.

Since pointer programs are hard to write correctly and difficult to reason about, many efforts have been made to figure out methods to provide safe C programs by program verification. Necula and Lee proposed Proof-Carrying Code [1] and a touchstone certifying compiler [2]. Touchstone is a compiler from a type-safe subset of the C language to optimized DEC Alpha machine code. The novel feature of the compiler is that it contains a certifier that automatically checks the type safety and memory safety of any assembly language program produced by the compiler. The result of the certifier is either a formal proof of type safety or a counterexample pointing to a potential violation of the type system by the assembly-language target program.

There is no pointer type and dynamic storage management in the subset of C language supported by touchstone, and also only type and memory safety can be guaranteed. It is urgent to support more features and provide more expressive annotations to specify more properties of the programs. A PLCC certifying compiler [3] has been proposed by our laboratory. It is a compiler from PointerC language [5] to x86 assembly language. PointerC is a safe C-like imperative language, which offers C style syntactic features.

The limitation of PointerC is that pointer arithmetic and address-of operator are forbidden. One novel feature of PointerC, comparing with other type-safe dialects of C, is that it offers explicit memory management, and the memory safety is guaranteed by a logic system pointer logic [4].

Pointer logic is an extension to Hoare logic, using different access path sets to denote different heap objects. At each program point, precise pointer information must be provided or calculated in order to support the verification. The latest version of PLCC certifying compiler supports programs that manipulate data structures such as singly-linked list, doubly-linked list, circular doubly-linked list and tree. The verification conditions are proved by pointer logic theorem prover. Some kind of proof is generated but not proof terms which can be checked by the proof assistant such as Coq.

In order to provide strict proof terms automatically, In order to support more data structures and provide strict proof terms, we designed Clike and the certifying compiler CComp. CComp is planned to support programs which manipulate data structures such as singly-linked list, doubly-linked list and tree. The program logic is a fragment of separation logic [6, 7].

This report presents the specification of Clike, a C-like programming language. The rest is organized as following: section 2 presents the syntax; section 3 presents its formal type system; section 4 introduces the operational semantics.

2. Lexical Rules of Clike

In Clike, some lexical rules are as following:

- Identifiers : An identifier is a sequence of letter, digit and underscore, and it must start with a letter or underscore.
For example, the following are legal identifier:
`_list, x1, tree_node.`
- Numbers: Only integer numbers are supported.
- Operators: Operators are listed in Figure 2.
- Key words: The identifiers written in bold font in Figure 1 are reserved key words. And **main** is a key word for the name of main function. **rettype** is also a key word for type checking the return value of functions.
So they are :
bool else false for free if int main malloc null rettype return sizeof struct true while void.
- Comments: Clike supports the comments styles both in C and Java programming languages. There are two styles for writing comments. The first kind of comments start with `/*`, and end with `*/`. And the second kind starts with `//` and lasts to the end of this line.

(Program)	<i>program</i>	::=	<i>structdeflist vardeclist fundef fundeflist</i>
(Structure Declare List)	<i>structdeflist</i>	::=	ϵ <i>structdef structdeflist</i>
(Structure Declare)	<i>structdef</i>	::=	struct <i>id</i> { <i>vardec vardeclist</i> }
(Variable Definition List)	<i>vardeclist</i>	::=	ϵ <i>vardec vardeclist</i>
(Variable Definition)	<i>vardec</i>	::=	<i>type idlist</i>
(Identifier List)	<i>idlist</i>	::=	<i>id</i> <i>id</i> , <i>idlist</i>
(Function Definition List)	<i>fundeflist</i>	::=	ϵ <i>fundef fundeflist</i>
(Function Definition)	<i>fundef</i>	::=	<i>type id(paramlist) body</i> void <i>id(paramlist) body</i>
(Parameter List)	<i>paramlist</i>	::=	ϵ <i>typeidlist</i>
(Type Identifier List)	<i>typeidlist</i>	::=	<i>type id</i> <i>type id</i> , <i>typeidlist</i>
(Function Body)	<i>body</i>	::=	{ <i>vardeclist stmlist</i> }
(Type)	<i>type</i>	::=	bool int struct <i>id</i> <i>type</i> [<i>intconst</i>] <i>type</i> *
(Statement List)	<i>stmlist</i>	::=	ϵ <i>stm stmlist</i>
(Statement)	<i>stm</i>	::=	<i>lval=exp</i> ; <i>lval=(type*) malloc(sizeof(type));</i> <i>lval=id(aparam);</i> <i>id(aparam);</i> free (<i>exp</i>); return <i>exp</i> ; return ; if (<i>exp</i>) <i>block</i> if (<i>exp</i>) <i>block</i> else <i>block</i> while (<i>exp</i>) <i>block</i> for (<i>stm; exp; stm</i>) <i>block</i>
(Statement Block)	<i>block</i>	::=	<i>stm</i> { <i>stmlist</i> }
(Expression)	<i>exp</i>	::=	(<i>exp</i>) <i>const</i> <i>lval</i> <i>unop exp</i> <i>exp biop exp</i>
(Actual Parameter List)	<i>aparam</i>	::=	ϵ <i>explist</i>
(Expression List)	<i>explist</i>	::=	<i>exp</i> <i>exp</i> , <i>explist</i>
(L-value Expression)	<i>lval</i>	::=	(<i>lval</i>) <i>id</i> <i>*lval</i> <i>lval[exp]</i> <i>lval.id</i> <i>lval->id</i>
(Constant)	<i>const</i>	::=	<i>intconst</i> true false null
(Unary Operator)	<i>unop</i>	::=	- !
(Binary Operator)	<i>biop</i>	::=	+ - * / % && == != < <= > >=
(Integer Constant)	<i>intconst</i>	::=	[0-9][0-9]+
(Identifier)	<i>id</i>	::=	[_A-Za-z][0-9_A-Za-z]*

Figure 1. The Syntax of Clike Programming Language

Operators	Associativity	Sort	Meaning
()	Left	None	Function Call
[] -> .	Left	None	Index, indirect selection, direct selection
! - *	Right	Unary	Not, negation, pointer de-reference
* / %	Left	Binary	Multiplication, division and modulus on integers
+ -	Left	Binary	Plus, subtract on integers
< <= > >=	Left	Binary	Comparison between integers
== !=	Left	Binary	Comparison between integers or pointers
&&	Left	Binary	Logic connective AND between booleans
	Left	Binary	Logic connective OR between booleans
=	Right	Binary	Assignment

Figure 2. The Precedence and Association of Operators in Clike

3. Syntax of Clike

The syntax of Clike is shown in Figure 1.

A Program in Clike is composed of structure definitions (*struct-deflist*), global variable declarations (*vardeclist*) and function definitions (*fundeflist*).

3.1 Structure Definition

Structure definitions defines zero or more structures. Each structure (*structdef*) contains a name (*id*) and structure body. Structure body contains names of structure fields and their types. Structures may be directly recursively defined. (Here by the word "directly", we mean that types of structure fields may be pointer types pointing to itself.)

3.2 Global Variable Definition

Global variable declaration (*vardeclist*) is composed of a type followed by a list of variables and then a semicolon. And variables are separated by comma if there are several variables.

3.3 Function Definition

Function definition list contains zero or more function definitions. Each function definition contains a return type, a function name, a formal argument list (*paramlist*) and a function body (*body*). The return type of a function may be void, which indicates nothing would be returned. Also note that an argument list may be empty. A function body contains local variable declarations and a list of statements (*stm*).

3.4 Statements

The basic syntactic form of statement is assignment (which also includes function call assignment). Other statements include function call/return, conditional branch, while-loop, for-loop and memory allocation/deallocation.

3.5 Types

Clike has a rich bunch of types. Basic types include integer (**int**) and boolean (**bool**). Structured types include structure, array and pointer. Variables of structure or array type may not be assigned to others as a whole. Condition expressions used in branch, loop statements are supposed to be of **int** or **bool** type. If the expression is of **int** type, the control flow will be determined by zero/non-zero value of it.

3.6 Operators

The address-of operator is forbidden in Clike, so all pointers point into the heap. In Clike, the associativity and precedence of operators are listed in Figure 2.

3.7 Sample Programs

Some typical program examples have been developed, some of them could be found under the test directory on our svn server. These programs could be compiled by the current version CComp compiler and run on x86/Linux.

4. Type System

Aa type system may be defined as a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. The process of verifying and enforcing the constraints of types is type checking.

4.1 Abstract Types

In Clike, type checking occurs compile-time (a static check). During type checking, an type context (Γ) is maintained. Variables are

associated with types. These types need not be the same as the concrete syntax written in program text. Instead abstract types are used in type checking. The production rules for abstract types are:

$$\begin{aligned} T^b &::= \mathbf{int} \mid \mathbf{bool} \mid s \mid \mathbf{array}[N, T^b] \mid T^b * \\ T^d &::= T^b \mid T^b \times T^d \\ T^a &::= \mathbf{void} \mid T^d \\ T^r &::= \mathbf{void} \mid T^b \\ T &::= \mathbf{NS} * \mid \mathbf{unit} \mid \mathbf{void} \mid T^d \mid T^a \rightarrow T^r \end{aligned}$$

In the following text, the symbols $\tau^b, \tau^d, \tau^a, \tau^r$ and τ stand for the expressions with these types.

The symbol s is meta variables stands for structure names. A structure definition includes a list of fields (field name and type). There exists a special symbols \mathbf{NS}^* , which is the type of **null**, it is compatible with any pointer type (the concept of compatibility is given in the next section). **void** denotes that the return type of a function without any return value. **unit** stands for the type of statements.

4.2 Typing Context

The typing context Γ can be produced by the following rule:

$$\begin{aligned} \Gamma &::= \cdot \mid \Gamma, x : \tau^b \mid \Gamma, f : \tau^a \rightarrow \tau^r \\ &\mid \Gamma, s : (l_1 : \tau_1^b, l_2 : \tau_2^b, \dots, l_n : \tau_n^b) \mid \Gamma, \mathbf{rettype} : \tau^r \end{aligned}$$

where Γ includes variables, function names, type names and function return type **rettype** and their type information. **rettype** is a reserved key word, so other identifiers should not be identical to it. The function returns no value, if the type of **rettype** is void.

$\Gamma(s)$ says that the structure name s appears in Γ and $\Gamma(s)(l) : \tau$ denotes Γ includes a structure s and the field l of s has type τ .

Generally speaking, a typing context Γ is assumed to be well-formed when it appears on the left hand side of a judgement. In all of the following judgements, Γ, τ, e and S denote typing context, type, expression and statement respectively.

$$\begin{aligned} \Gamma \vdash \diamond & \quad \text{typing context } \Gamma \text{ is well-formed} \\ \Gamma \vdash \tau & \quad \text{type } \tau \text{ is well-formed} \\ \Gamma \vdash e : \tau & \quad \text{expression } e \text{ with type } \tau \text{ is well-typed} \\ \Gamma \vdash S : \mathbf{unit} & \quad \text{statement } S \text{ is well-typed} \\ \Gamma \vdash \mathbf{rettype} : \tau^r & \quad \text{rettype with type } \tau^r \text{ is well-typed} \end{aligned}$$

4.3 Well-formed Types

Since not all types produced by the above syntactic rules are legal, we must give well-formed rules for types.

Note: in the following definitions, \mathbf{NS}^* is not a well-formed type.

$$\boxed{\Gamma \vdash \tau}$$

$$\frac{}{\Gamma \vdash \mathbf{int}} \quad (\mathbf{TINT})$$

$$\frac{}{\Gamma \vdash \mathbf{bool}} \quad (\mathbf{TBOOL})$$

$$\frac{}{\Gamma, s : (l_1 : \tau_1^b, l_2 : \tau_2^b, \dots, l_n : \tau_n^b) \vdash s} \quad (\mathbf{TSTR})$$

In the rule **TSTR**, $\Gamma, s : (l_1 : \tau_1^b, l_2 : \tau_2^b, \dots, l_n : \tau_n^b)$ is assumed to be valid here. Conditions such as l_i should not duplicate, or

τ_i^b should have type T^b will be considered in the typing context extension rule.

$$\frac{\Gamma \vdash \tau^b}{\Gamma \vdash \tau^b_*} \quad (\text{TPTR})$$

$$\frac{\Gamma \vdash \tau^b \text{ } N \text{ is integer constant}}{\Gamma \vdash \tau^b[N]} \quad (\text{TARRAY})$$

$$\frac{\Gamma \vdash \tau^b \quad \Gamma \vdash \tau^d}{\Gamma \vdash \tau^b \times \tau^d} \quad (\text{TPRODUCT})$$

$$\frac{}{\Gamma \vdash \text{void}} \quad (\text{TVOID})$$

TPRODUCT rule describes the well-formedness of function arguments type.

$$\frac{\Gamma \vdash \tau^a \quad \Gamma \vdash \tau^r}{\Gamma \vdash \tau^a \rightarrow \tau^r} \quad (\text{TFUN})$$

TFUN gives the well-formedness conditions for function type. Note that both function argument and its return value could be empty (the **void** type).

4.4 Type Compatibility

The most rudimentary operation during type checking is comparison of type compatibility: when variables with different types are allowed to be used in assignment and parameter passing.

For arbitrary two types τ_1 and τ_2 , if they are compatible, we denote it as $\tau_1 \cong \tau_2$. Especially, if s_1 and s_2 are both identifiers, $s_1 \equiv s_2$ means that the two strings are identical. This convention illustrates our choice of name equivalent, but not structural equivalent. (Note that our choice is the same as that of the C programming language.) The rules for type compatibility are presented as the following.

$$\boxed{\Gamma \vdash \tau_1 \cong \tau_2}$$

$$\frac{}{\Gamma \vdash \text{int} \cong \text{int}} \quad (\text{TCMP_INT})$$

$$\frac{}{\Gamma \vdash \text{bool} \cong \text{bool}} \quad (\text{TCMP_BOOL})$$

$$\frac{\Gamma \vdash s_1 \quad \Gamma \vdash s_2 \quad s_1 \equiv s_2}{\Gamma \vdash s_1 \cong s_2} \quad (\text{TCMP_STRUCT})$$

$$\frac{\Gamma \vdash \tau_1[N_1] \quad \Gamma \vdash \tau_2[N_2] \quad \Gamma \vdash \tau_1 \cong \tau_2 \quad N_1 = N_2}{\Gamma \vdash \tau_1[N_1] \cong \tau_2[N_2]} \quad (\text{TCMP_ARRAY})$$

$$\frac{}{\Gamma \vdash \text{NS}^* \cong \text{NS}^*} \quad (\text{TCMP_NS})$$

$$\frac{\Gamma \vdash \tau_1^b \cong \tau_2^b}{\Gamma \vdash \tau_1^b_* \cong \tau_2^b_*} \quad (\text{TCMP_PTR})$$

$$\frac{\Gamma \vdash \tau^b}{\Gamma \vdash \tau^b_* \cong \text{NS}^*} \quad (\text{TCMP_PTR_NS})$$

$$\frac{\Gamma \vdash \tau^b}{\Gamma \vdash \text{NS}^* \cong \tau^b_*} \quad (\text{TCMP_NS_PTR})$$

$$\frac{\Gamma \vdash \tau_1^b \cong \tau_2^b \quad \Gamma \vdash \tau_1^d \cong \tau_2^d}{\Gamma \vdash \tau_1^b \times \tau_1^d \cong \tau_2^b \times \tau_2^d} \quad (\text{TCMP_PRODUCT})$$

4.5 Typing Context Extension

During type checking, typing context needs to be extended. For example, variable or function declarations can be added into the typing context. And the extension is governed by the following rules:

$$\boxed{\Gamma \vdash \diamond}$$

$$\frac{}{\emptyset_\Gamma \vdash \diamond} \quad (\text{TCEXT_EMP})$$

$$\frac{\Gamma \vdash \diamond \quad \Gamma \vdash \tau^b \quad x \notin \text{dom}(\Gamma)}{\Gamma, x : \tau^b \vdash \diamond} \quad (\text{TCEXT_VAR})$$

$$\frac{\Gamma \vdash \diamond \quad \Gamma \vdash \tau^b \quad \text{rettype} \notin \text{dom}(\Gamma)}{\Gamma, \text{rettype} : \tau^b \vdash \diamond} \quad (\text{TCEXT_RETTYPE})$$

$$\frac{\Gamma \vdash \diamond \quad \Gamma \vdash \tau^a \rightarrow \tau^r \quad f \neq \text{main} \quad f \notin \text{dom}(\Gamma)}{\Gamma, f : \tau^a \rightarrow \tau^r \vdash \diamond} \quad (\text{TCEXT_FUNCTION})$$

$$\frac{\Gamma \vdash \diamond \quad \Gamma \vdash \tau^r \quad \text{main} \notin \text{dom}(\Gamma)}{\Gamma, \text{main} : \text{void} \rightarrow \tau^r \vdash \diamond} \quad (\text{TCEXT_MAIN})$$

Note that the rule for the main function requires the argument type must be void, which means that the main function has no parameter.

$$\frac{\Gamma \vdash \diamond \quad s \notin \text{dom}(\Gamma) \quad \forall l_i, l_j. l_i \neq l_j (1 \leq i, j \leq n \text{ and } i \neq j) \quad \Gamma \vdash \tau_i^b \text{ or } \Gamma, s \vdash \tau_i^b \cong \text{Ptr}(s) (1 \leq i \leq n)}{\Gamma, s : (l_1 : \tau_1^b, l_2 : \tau_2^b, \dots, l_n : \tau_n^b) \vdash \diamond} \quad (\text{TCEXT_STRUCT})$$

where $\text{Ptr}(s)$ denotes a pointer type that eventually points to structure s (no matter how many pointers it goes through), and the rules for $\text{Ptr}(s)$ are:

$$\frac{\Gamma, s \vdash \tau \cong s^*}{\Gamma, s \vdash \tau : \text{Ptr}(s)} \quad (\text{PTR_S})$$

$$\frac{\Gamma, s \vdash \tau \cong \tau' * \quad \Gamma, s \vdash \tau' : \text{Ptr}(s)}{\Gamma, s \vdash \tau : \text{Ptr}(s)} \quad (\text{PTR_PTR_S})$$

4.6 Typing Rules for Expressions

Typing rules for expressions are straightforward.

$$\boxed{\Gamma \vdash e : \tau}$$

- Constants

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}} \quad (\text{TR_EXP_TRUE})$$

$$\frac{}{\Gamma \vdash \text{false} : \text{bool}} \quad (\text{TR_EXP_FALSE})$$

$$\frac{}{\Gamma \vdash \mathbf{null} : \mathbf{NS}^*} \quad (\text{TR_EXP_NULL})$$

$$\frac{}{\Gamma \vdash \mathbf{intconst} : \mathbf{int}} \quad (\text{TR_EXP_INT})$$

- L-value Expressions

$$\frac{}{\Gamma, x : \tau^b \vdash x : \tau^b} \quad (\text{TR_EXP_VAR})$$

$$\frac{\Gamma \vdash e : \tau^*}{\Gamma \vdash *e : \tau} \quad (\text{TR_EXP_DEREF})$$

$$\frac{\Gamma \vdash e : s \quad \Gamma \vdash \Gamma(s)(l) : \tau}{\Gamma \vdash e.l : \tau} \quad (\text{TR_EXP_SFIELD})$$

$$\frac{\Gamma \vdash e : \tau^* \quad \Gamma \vdash \Gamma(s)(l) : \tau'}{\Gamma \vdash e \rightarrow l : \tau'} \quad (\text{TR_EXP_PFIELD})$$

$$\frac{\Gamma \vdash e : \tau[N] \quad \Gamma \vdash l : \mathbf{int}}{\Gamma \vdash e[l] : \tau} \quad (\text{TR_EXP_ARRAY})$$

- Unary operator Expressions

$$\frac{\Gamma \vdash e : \mathbf{int}}{\Gamma \vdash -e : \mathbf{int}} \quad (\text{TR_EXP_INT_MINUS})$$

$$\frac{\Gamma \vdash e : \mathbf{bool}}{\Gamma \vdash !e : \mathbf{bool}} \quad (\text{TR_EXP_BOOL_NOT})$$

- Binary operator Expressions

$$\frac{\Gamma \vdash e_1 : \mathbf{int} \quad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 \mathit{binop} e_2 : \mathbf{int}} \quad (\text{TR_EXP_INT_ARITH})$$

where $\mathit{binop} \in \{+, -, *, /, \%\}$

$$\frac{\Gamma \vdash e_1 : \mathbf{int} \quad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 \mathit{binop} e_2 : \mathbf{bool}} \quad (\text{TR_EXP_INT_COMPARE})$$

where $\mathit{binop} \in \{==, !=, <, <=, >, >=\}$

$$\frac{\Gamma \vdash e_1 : \mathbf{bool} \quad \Gamma \vdash e_2 : \mathbf{bool}}{\Gamma \vdash e_1 \mathit{binop} e_2 : \mathbf{bool}} \quad (\text{TR_EXP_BOOL_ARITH})$$

where $\mathit{binop} \in \{\&\&, \|\}$

- Other Expressions

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (e) : \tau} \quad (\text{TR_EXP_BRACKET})$$

$$\frac{\Gamma \vdash e : \tau^b \quad \Gamma \vdash \mathit{explist} : \tau^d}{\Gamma \vdash e, \mathit{explist} : \tau^b \times \tau^d} \quad (\text{TR_EXP_EXPLIST})$$

$$\frac{}{\Gamma, \mathbf{rettype} : \tau^r \vdash \mathbf{rettype} : \tau^r} \quad (\text{TR_EXP_RETTYPE})$$

4.7 Typing Rules for Statements

- Assignment Statements

$$\frac{\Gamma \vdash \mathit{lval} : \tau_l^b \quad \Gamma \vdash \mathit{exp} : \tau_e^b \quad \Gamma \vdash \tau_l^b \cong \tau_e^b \quad \nexists \tau. \Gamma \vdash \tau_l^b \cong \tau^*}{\Gamma \vdash \mathit{lval} = \mathit{exp}; : \mathbf{unit}} \quad (\text{TR_STM_ASSIGN_SIM})$$

$$\frac{\Gamma \vdash \mathit{lval} : \tau_l^* \quad \Gamma \vdash \mathit{exp} : \tau_e^* \quad \Gamma \vdash \tau_l^* \cong \tau_e^*}{\Gamma \vdash \mathit{lval} = \mathit{exp}; : \mathbf{unit}} \quad (\text{TR_STM_ASSIGN_PTR})$$

$$\frac{\Gamma \vdash \mathit{lval} : \tau_l^b \quad \Gamma \vdash \tau_l^b \cong \tau_a^b}{\Gamma \vdash \mathit{lval} = (\tau_a^b) \mathbf{malloc}(\mathbf{sizeof}(\tau_a^b)); : \mathbf{unit}} \quad (\text{TR_STM_ASSIGN_MALLOC})$$

$$\frac{\Gamma \vdash \mathit{lval} : \tau_l^b \quad \Gamma \vdash f : \tau_1^d \rightarrow \tau_e^b \quad \Gamma \vdash \mathit{explist} : \tau_e^b \quad \Gamma \vdash \tau_l^b \cong \tau_e^b \quad \Gamma \vdash \tau_1^d \cong \tau_2^d}{\Gamma \vdash \mathit{lval} = f(\mathit{explist}); : \mathbf{unit}} \quad (\text{TR_STM_ASSIGN_FUN})$$

$$\frac{\Gamma \vdash \mathit{lval} : \tau_l^b \quad \Gamma \vdash f : \mathbf{void} \rightarrow \tau_e^b \quad \Gamma \vdash \tau_l^b \cong \tau_e^b}{\Gamma \vdash \mathit{lval} = f(); : \mathbf{unit}} \quad (\text{TR_STM_ASSIGN_FUNVOID})$$

- If-Else Statements

$$\frac{\Gamma \vdash \mathit{exp} : \tau \quad \tau \in \{\mathbf{bool}, \mathbf{int}\} \quad \Gamma \vdash \mathit{block} : \mathbf{unit}}{\Gamma \vdash \mathbf{if}(\mathit{exp}) \mathit{block} : \mathbf{unit}} \quad (\text{TR_STM_IF})$$

$$\frac{\Gamma \vdash \mathit{exp} : \tau \quad \tau \in \{\mathbf{bool}, \mathbf{int}\} \quad \Gamma \vdash \mathit{block}_1 : \mathbf{unit} \quad \Gamma \vdash \mathit{block}_2 : \mathbf{unit}}{\Gamma \vdash \mathbf{if}(\mathit{exp}) \mathit{block}_1 \mathbf{else} \mathit{block}_2 : \mathbf{unit}} \quad (\text{TR_STM_IFELSE})$$

- Function Call/Return Statements

$$\frac{\Gamma \vdash e : \tau^*}{\Gamma \vdash \mathbf{free}(e); : \mathbf{unit}} \quad (\text{TR_STM_FREE})$$

$$\frac{\Gamma \vdash f : \tau^d \rightarrow \tau^r \quad \Gamma \vdash \mathit{explist} : \tau_e^d \quad \Gamma \vdash \tau_e^d \cong \tau_e^d}{\Gamma \vdash f(\mathit{explist}); : \mathbf{unit}} \quad (\text{TR_STM_FUN})$$

$$\frac{\Gamma \vdash f : \mathbf{void} \rightarrow \tau^r}{\Gamma \vdash f(); : \mathbf{unit}} \quad (\text{TR_STM_FUNVOID})$$

$$\frac{\Gamma \vdash \mathit{exp} : \tau_e^r \quad \Gamma \vdash \mathbf{rettype} : \tau^r \quad \Gamma \vdash \tau_e^r \cong \tau^r}{\Gamma \vdash \mathbf{return}(\mathit{exp}); : \mathbf{unit}} \quad (\text{TR_STM_RETURN_VAL})$$

$$\frac{\Gamma \vdash \mathbf{rettype} : \mathbf{void}}{\Gamma \vdash \mathbf{return}; : \mathbf{unit}} \quad (\text{TR_STM_RETURN})$$

- Loop Statements

$$\frac{\Gamma \vdash \mathit{exp} : \tau \quad \tau \in \{\mathbf{bool}, \mathbf{int}\} \quad \Gamma \vdash \mathit{block} : \mathbf{unit}}{\Gamma \vdash \mathbf{while}(\mathit{exp}) \mathit{block} : \mathbf{unit}} \quad (\text{TR_STM_WHILE})$$

$$\frac{\Gamma \vdash \text{exp} : \tau \quad \tau \in \{\text{bool}, \text{int}\} \quad \Gamma \vdash \text{stm}_1 : \text{unit} \quad \Gamma \vdash \text{stm}_2 : \text{unit} \quad \Gamma \vdash \text{block} : \text{unit}}{\Gamma \vdash \text{for}(\text{stm}_1; \text{exp}; \text{stm}_2) \text{ block} : \text{unit}} \quad (\text{TR_STM_FOR})$$

- Other cases

$$\frac{}{\Gamma \vdash \epsilon : \text{unit}} \quad (\text{TR_STMLIST_EMP})$$

$$\frac{\Gamma \vdash \text{stm} : \text{unit} \quad \Gamma \vdash \text{stmlist} : \text{unit}}{\Gamma \vdash \text{stm} \text{ stmlist} : \text{unit}} \quad (\text{TR_STMLIST_STM})$$

$$\frac{\Gamma \vdash \text{stmlist} : \text{unit}}{\Gamma \vdash \{\text{stmlist}\} : \text{unit}} \quad (\text{TR_STM_BLOCK})$$

4.8 Typing Rules for Function Definition

- Function Definition.

$$\frac{\Gamma \vdash f : \text{void} \rightarrow \tau^r \quad \Gamma, \text{rettype} : \tau^r \vdash \diamond \quad \Gamma, \text{rettype} : \tau^r \vdash \text{body} : \text{unit}}{\Gamma \vdash \tau^r f() \{\text{body}\} : \text{unit}} \quad (\text{TR_FUNDEF_NOPARAM})$$

$$\frac{\Gamma \vdash f : \tau_1 \times \tau_2 \times \dots \times \tau_n \rightarrow \tau^r \quad \Gamma, p_1 : \tau_1, p_2 : \tau_2, \dots, p_n : \tau_n, \text{rettype} : \tau^r \vdash \diamond \quad \Gamma, p_1 : \tau_1, p_2 : \tau_2, \dots, p_n : \tau_n, \text{rettype} : \tau^r \vdash \text{body} : \text{unit}}{\Gamma \vdash \tau^r f(\tau_1 p_1, \tau_2 p_2, \dots, \tau_n p_n) \{\text{body}\} : \text{unit}} \quad (\text{TR_FUNDEF_PARAM})$$

- Function Body.

$$\frac{\Gamma, x_1 : \tau, x_2 : \tau, \dots, x_n : \tau \vdash \diamond \quad \Gamma, x_1 : \tau, x_2 : \tau, \dots, x_n : \tau \vdash \text{vardeclist} \text{ stmlist} : \text{unit}}{\Gamma \vdash \tau x_1, x_2, \dots, x_n; \text{vardeclist} \text{ stmlist} : \text{unit}} \quad (\text{TR_FUNDEF_BODY})$$

$$\frac{\Gamma \vdash \text{stmlist} : \text{unit}}{\Gamma \vdash \epsilon \text{ stmlist} : \text{unit}} \quad (\text{TR_FUNDEF_BODY_VDECEMP})$$

4.9 Typing Rules for Program

- Program (Top Rule).

$$\frac{\emptyset \vdash \text{program} : \text{unit}}{\vdash \text{program} : \text{unit}} \quad (\text{TR_PROG})$$

where $\text{program} \equiv \text{structdeflist} \text{ vardeclist} \text{ fundef} \text{ fundeflist}$

- Structure Definition.

$$\frac{\Gamma, s : (l_1 : \tau_1, l_2 : \tau_2, \dots, l_n : \tau_n) \vdash \diamond \quad \Gamma, s : (l_1 : \tau_1, l_2 : \tau_2, \dots, l_n : \tau_n) \vdash \text{program} : \text{unit}}{\Gamma \vdash \text{struct} s\{\tau_1 l_1; \tau_2 l_2; \dots; \tau_n l_n;\} \text{program} : \text{unit}} \quad (\text{TR_PROG_STRUCTDEF})$$

where $\text{program} \equiv \text{structdeflist} \text{ vardeclist} \text{ fundef} \text{ fundeflist}$

$$\frac{\Gamma \vdash \text{vardeclist} \text{ fundef} \text{ fundeflist} : \text{unit}}{\Gamma \vdash \epsilon \text{ vardeclist} \text{ fundef} \text{ fundeflist} : \text{unit}} \quad (\text{TR_PROG_STRUCTDEF_EMP})$$

- Global Variable Declaration.

$$\frac{\forall i. x_i \notin \text{dom}(\Gamma) (1 \leq i \leq n) \quad \Gamma, x_1 : \tau^b, x_2 : \tau^b, \dots, x_n : \tau^b \vdash \text{vardeclist} \text{ fundef} \text{ fundeflist} : \text{unit}}{\Gamma \vdash \tau^b x_1, x_2, \dots, x_n; \text{vardeclist} \text{ fundef} \text{ fundeflist} : \text{unit}} \quad (\text{TR_PROG_VARDEC})$$

$$\frac{\Gamma \vdash \text{fundef} \text{ fundeflist} : \text{unit}}{\Gamma \vdash \epsilon \text{ vardeclist} \text{ fundef} \text{ fundeflist} : \text{unit}} \quad (\text{TR_PROG_VARDEC_EMP})$$

- Function Definition List.

$$\frac{\Gamma \vdash \text{fundef} : \text{unit} \quad \Gamma \vdash \text{fundeflist} : \text{unit}}{\Gamma \vdash \text{fundef} \text{ fundeflist} : \text{unit}} \quad (\text{TR_PROG_FUNDEF_LIST})$$

$$\frac{}{\Gamma \vdash \epsilon : \text{unit}} \quad (\text{TR_PROG_FUNDEF_EMP})$$

5. Test Cases

Currently CComp is planned to support programs which manipulate data structures such as singly-linked list, doubly-linked list and tree. The program logic is a fragment of separation logic.

5.1 Singly-linked List

Suppose the data type is declared as the following:

```
struct list{
  int data;
  struct list * next; }
```

For programs to manipulate singly-linked list, the followings are supported : traversal, creation, destroy, node insertion, node removal, reversal and etc.

5.1.1 List Traversal

In Figure 3, it is a program which traverses a singly-linked list and does nothing else.

```
/*@ true | list(p) @*/
void list.traversal(struct list * p) {
  struct list* tmp;
  tmp = p;
  /*@ true | lseg(p, tmp)*list(tmp) @*/
  while(tmp != NULL){
    tmp = tmp->next;
  }
  return;
}
/*@ true | list(p) @*/
```

Figure 3. test case for singly-linked list traversal

5.1.2 List Creation

In Figure 4, it is a program which creates a singly-linked list with n nodes. Due to space limit, $(\text{type}^*)\text{malloc}(\text{sizeof}(\text{type}))$ is abbreviated into $\text{malloc}(\text{sizeof}(\text{type}))$ in this and the following test cases.

5.1.3 List Deallocation

In Figure 5, it is a program frees all of the nodes of a singly-linked list.

```

/*@ n>=0 | emp @*/
struct list* list_create(int n) {
    struct list *p;
    struct list *l;
    int i;
    p = null;
    l = null;
    i = 0;
/*@ i<=n/\p==l | list(p) @*/
    while(i<n){
        l = malloc(sizeof(struct list));
        l->data = n-i;
        l->next = p;
        p = l;
        i = i+1;
    }
    return p;
}
/*@ true | list(res) @*/

```

Figure 4. test case for singly-linked list creation

```

/*@ true | list(p) @*/
void list_dealloc(struct list* p) {
    struct list *tmp;
    if(p==NULL)
        return;
    else {
        tmp = p;
/*@ p==tmp | list(p) @*/
        while(p!=NULL){
            tmp = p->next;
            free(p);
            p = tmp;
        }
        return;
    }
}
/*@ true | emp @*/

```

Figure 5. test case for singly-linked list deallocation

5.1.4 List Insertion

In Figure 6, it is a program inserts a node into a singly-linked list.

5.2 doubly-linked List

5.3 Tree

References

- [1] G. Necula. Proof-carrying code, In *Proc.24th ACM Symposium on Principles of Programming Languages*, pages 106-119, Jan 1997.
- [2] G. Necula, P. Lee. The Design and Implementation of a Certifying Compiler, In *Proceedings of the '98 Conference on Programming Language Design and Implementation*, Montreal, 1998.
- [3] Y. Y. Chen, L. Ge, B. J. Hua, Z. P. Li and C. Liu. Design of a Certifying Compiler Supporting Proof of Program Safety. In *Proceedings of 1st IEEE IFIP International Symposium on Theoretical Aspects of Software Engineering (TASE 2007)*, pages 117-126, June 2007.
- [4] Y. Y. Chen, L. Ge, B. J. Hua, Z. P. Li, C. Liu and Z. F. Wang. A Pointer Logic and Certifying Compiler. *Frontiers of Computer Science in China*.1(3):297-312, July 2007.
- [5] B. J. Hua, Y. Y. Chen, L. Ge, and Z. F. Wang. The PointerC

```

/*@ true | list(p) @*/
struct list* list_insert(struct list* p, int val) {
    struct list *tmp;
    if(p==NULL){
        tmp = malloc(sizeof(struct list));
        tmp->data = val;
        tmp->next = p;
        return tmp;
    }
    else {
        if(p->data<val){
            tmp = list_insert(p->next, val);
            p->next = tmp
            return p;
        }
        else{
            tmp = malloc(sizeof(struct list));
            tmp->data = val;
            tmp->next = p;
            return tmp;
        }
    }
}
/*@ true | list(res) @*/

```

Figure 6. test case for singly-linked list deallocation

programming language specification. (*Technical Report*) Available at: <http://sug.ustcsz.edu.cn/lss/doc/>.

- [6] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55-74, July 2002.
- [7] J. C. Reynolds. An introduction to separation logic. Lecture notes available at <http://www.cs.cmu.edu/jcr/>, Spring 2005.