

# A Program Logic for Clike Programming Language

## Technique Report

Zhaopeng Li   Yiyun Chen   Zhenting Zhang

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 program logic for Clike is presented, which is a fragment of separation logic. Clike 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 and a touchstone certifying compiler. 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 has been proposed by our laboratory. It is a compiler from PointerC language to x86 assembly language. PointerC is a safe C-like im-

perative 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.

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, cyclic 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 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, tree and DAG.

The program logic is a fragment of separation logic. The representation style is oriented from Smallfoot.

This report presents a program logic for Clike. The rest is organized as following: section 2 presents the assertion language; section 3 presents its inference rules; section 4, a verification condition generator is introduced. And section 5 introduces the design of a prover for verification condition.

### 2. Assertion Language SAL

In this section, the separation assertion language (SAL) is introduced, the syntax is shown in Figure 1. Program variable are implicitly universal-quantified in assertions. And we use hatted variables to implicitly denote existentially quantified variables. That is,  $\mathbb{A}$  is a shorthand for

$$\forall id_1, \dots, id_n. \exists \hat{x}_1, \hat{x}_2, \dots, \hat{x}_n. \mathbb{A}.$$

Currently, the supported build-in predicates are as following: array for heap-allocated array, list/lseg for singly-linked list, dlist/dlseg for doubly-linked list, cdlist for circular doubly-linked list and tree for binary tree.

#### 2.1 Type System for SAL

Data structure types with build-in predicates supported in SAL is shown in Figure 2. That is, if one wants to specify the program using build-in predicates, the parameters must be typed with corresponding data structure types.

Suppose  $\Gamma$  is the typing context of the corresponding Clike program. After type checking the function, the typing context  $\Gamma$

(Assertion)	$\mathbb{A} ::= (\Pi \wedge \Sigma) \mid (\Pi \wedge \Sigma) \vee \mathbb{A}$
(Pure Formula)	$\Pi ::= \text{true} \mid \text{false} \mid \mathbb{B} \wedge \Pi$
(Spatial Formula)	$\Sigma ::= \text{emp} \mid \mathbb{H} * \Sigma$
(Boolean Predicates)	$\mathbb{B} ::= aexp_1 \text{ relop } aexp_2$
(Heap Predicates)	$\mathbb{H} ::= aexp \mapsto_{\tau} aexp_1 \mid [aexp, aexp_1] \mapsto_{\tau} aexp_2$ $\mid aexp \mapsto_{\tau} \{id_1 : aexp_1, id_2 : aexp_2, \dots, id_n : aexp_n\}$ $\mid \text{array}_{\tau}(aexp, aexp_1, aexp_2)$ $\mid \text{tree}(aexp) \mid \text{list}(aexp) \mid \text{lseg}(aexp_1, aexp_2)$ $\mid \text{dlist}(aexp_1, aexp_2) \mid \text{dlseg}(aexp_1, aexp_2, aexp_3, aexp_4)$ $\mid \text{cdlist}(aexp_1, aexp_2)$
(Expressions)	$aexp ::= (aexp) \mid \text{inconst} \mid \text{null} \mid \text{res} \mid \text{id} \mid \text{old}(\text{id}) \mid \hat{v} \mid aexp_1 \text{ biop } aexp_2$
(Relations)	$\text{relop} ::= == \mid != \mid >= \mid > \mid < \mid <=$
(Binary Operations)	$\text{binop} ::= + \mid - \mid *$

Figure 1. The Syntax of Assertion Language SAL

<pre> struct list{   int data;   struct list* next; } struct dlist{   int data;   struct dlist* next;   struct dlist* prev; } struct tree{   int data;   struct tree* left;   struct tree* right; } </pre>	$\overline{\Gamma \vdash \text{false}}$ (TR_PURE_FALSE) $\boxed{\Gamma \vdash \mathbb{B}}$ $\frac{\Gamma \vdash \mathbb{B}}{\Gamma \vdash \neg \mathbb{B}}$ (TR_BOOL_NOT) $\frac{\Gamma \vdash aexp_1 : \tau_1 \quad \Gamma \vdash aexp_2 : \tau_2 \quad \tau_1 \cong \tau_2}{\Gamma \vdash aexp_1 \text{ op } aexp_2}$ (TR_BOOL_EQUALITY) where $op \in \{==, !=\}$
--	--

Figure 2. The Structures with Build-in Supported Predicates

will be used to type check the specifications of the function : pre-/post-conditions and loop invariants. Programs with ill-typed assertions will be rejected by the type checker of Clite.

Besides the typing judgements in Clite[1], some additional judgements are needed as following:

$\Gamma \vdash \mathbb{A}$	<i>Assertion <math>\mathbb{A}</math> is well-typed</i>
$\Gamma \vdash \Pi$	<i>Pure formula <math>\Pi</math> is well-typed</i>
$\Gamma \vdash \mathbb{B}$	<i>Bool Predicate <math>\mathbb{B}</math> is well-typed</i>
$\Gamma \vdash \Sigma : \Gamma'$	<i>Spatial formula <math>\Sigma</math> is well-typed</i>
$\Gamma \vdash (aexp : \tau)$	<i>Expressions <math>e</math> in assertions is well-typed</i>
$\Gamma \vdash (aexp : \tau) : \Gamma'$	<i>Expressions <math>e</math> in assertions is well-typed</i>

And in last two judgements, the typing context after type checking is changed into  $\Gamma'$  because types of hatted variables may be added.

$\boxed{\Gamma \vdash \mathbb{A}}$ $\frac{\Gamma \vdash \Pi \wedge \Sigma \quad \Gamma \vdash \mathbb{A}}{\Gamma \vdash (\Pi \wedge \Sigma) \vee \mathbb{A}}$ (TR_ASSERTION) $\frac{\Gamma \vdash \Sigma : \Gamma' \quad \Gamma' \vdash \Pi}{\Gamma \vdash \Pi \wedge \Sigma}$ (TR_FORMULA) $\boxed{\Gamma \vdash \Pi}$ $\frac{\Gamma \vdash \mathbb{B} \quad \Gamma \vdash \Pi}{\Gamma \vdash \mathbb{B} \wedge \Pi}$ (TR_PURE_CONJ) $\overline{\Gamma \vdash \text{true}}$ (TR_PURE_TRUE)	$\frac{\Gamma \vdash \text{retype} : \tau' \quad \tau \cong \tau'}{\Gamma \vdash \text{res} : \tau}$ (TR_EXP_RES) $\boxed{\Gamma \vdash \Sigma : \Gamma'}$ $\overline{\Gamma \vdash \text{emp} : \Gamma}$ (TR_SIGMA_EMP) $\frac{\Gamma \vdash \mathbb{H} : \Gamma_1 \quad \Gamma_1 \vdash \Sigma : \Gamma'}{\Gamma \vdash \mathbb{H} * \Sigma : \Gamma'}$ (TR_SIGMA_STAR) $\frac{\Gamma \vdash (aexp : \tau*) : \Gamma_1 \quad \Gamma_1 \vdash (aexp_1 : \tau) : \Gamma'}{\Gamma \vdash aexp \mapsto_{\tau} aexp_1 : \Gamma'}$ (TR_SPATIAL_MAPSTOSIM) $\frac{\tau* \cong s* \quad \Gamma \vdash s : (id_1 : \tau_1, id_2 : \tau_2, \dots, id_n : \tau_n) \quad \Gamma \vdash (aexp : \tau*) : \Gamma_0 \quad \forall i. \Gamma_{i-1} \vdash (aexp_i : \tau_i) : \Gamma_i (1 \leq i \leq n)}{\Gamma \vdash aexp \mapsto_{\tau} \{id_1 : aexp_1, id_2 : aexp_2, \dots, id_n : aexp_n\} : \Gamma_n}$ (TR_SPATIAL_MAPSTOREC) $\frac{\Gamma \vdash aexp : \tau[] \quad \Gamma \vdash aexp_1 : \text{int} \quad \Gamma \vdash (aexp_2 : \tau) : \Gamma'}{\Gamma \vdash [aexp, aexp_1] \mapsto_{\tau} aexp_2 : \Gamma'}$ (TR_SPATIAL_MAPSTOARR)
--	--

$$\frac{\Gamma \vdash (aexp : \tau[]) : \Gamma_0 \quad \forall i. \Gamma_{i-1} \vdash (aexp_i : \mathbf{int}) : \Gamma_i (1 \leq i \leq 2)}{\Gamma \vdash \mathbf{array}_\tau(aexp, aexp_1, aexp_2) : \Gamma_2} \text{(TR\_SPATIAL\_ARRAY)}$$

$$\frac{\Gamma \vdash (aexp : \mathbf{tree}^*) : \Gamma'}{\Gamma \vdash \mathbf{tree}(aexp) : \Gamma'} \text{(TR\_SPATIAL\_TREE)}$$

$$\frac{\Gamma \vdash (aexp : \mathbf{list}^*) : \Gamma'}{\Gamma \vdash \mathbf{list}(aexp) : \Gamma'} \text{(TR\_SPATIAL\_LIST)}$$

$$\frac{\Gamma \vdash (aexp_1 : \mathbf{list}^*) : \Gamma_1 \quad \Gamma_1 \vdash (aexp_2 : \mathbf{list}^*) : \Gamma'}{\Gamma \vdash \mathbf{lseg}(aexp_1, aexp_2) : \Gamma'} \text{(TR\_SPATIAL\_LSEG)}$$

$$\frac{\Gamma \vdash (aexp_1 : \mathbf{dlist}^*) : \Gamma_1 \quad \Gamma_1 \vdash (aexp_2 : \mathbf{dlist}^*) : \Gamma'}{\Gamma \vdash \mathbf{dlseg}(aexp_1, aexp_2) : \Gamma'} \text{(TR\_SPATIAL\_DLSEG)}$$

$$\frac{\Gamma \vdash (aexp_1 : \mathbf{dlist}^*) : \Gamma_0 \quad \forall i. \Gamma_{i-1} \vdash (aexp_i : \mathbf{dlist}^*) : \Gamma_i (1 \leq i \leq 3)}{\Gamma \vdash \mathbf{dlseg}(aexp, aexp_1, aexp_2, aexp_3) : \Gamma_3} \text{(TR\_SPATIAL\_DLSEG)}$$

$$\frac{\Gamma \vdash (aexp_1 : \mathbf{dlist}^*) : \Gamma_1 \quad \Gamma_1 \vdash (aexp_2 : \mathbf{dlist}^*) : \Gamma'}{\Gamma \vdash \mathbf{cdlist}(aexp_1, aexp_2) : \Gamma'} \text{(TR\_SPATIAL\_CDLIST)}$$

$$\boxed{\Gamma \vdash (aexp : \tau) : \Gamma'}$$

$$\frac{\Gamma \vdash aexp : \tau}{\Gamma \vdash ((aexp) : \tau) : \Gamma} \text{(TR\_ASSERTEXP\_EXP)}$$

$$\frac{\Gamma \vdash \mathbf{intconst} : \mathbf{int}}{\Gamma \vdash (\mathbf{intconst} : \mathbf{int}) : \Gamma} \text{(TR\_ASSERTEXP\_INTC)}$$

$$\frac{\tau \cong \mathbf{NS}^*}{\Gamma \vdash (\mathbf{null} : \tau) : \Gamma} \text{(TR\_ASSERTEXP\_NULL)}$$

$$\frac{\Gamma \vdash \mathbf{res} : \tau}{\Gamma \vdash (\mathbf{res} : \tau) : \Gamma} \text{(TR\_ASSERTEXP\_RES)}$$

$$\frac{\Gamma \vdash \mathbf{id} : \tau}{\Gamma \vdash (\mathbf{old}(\mathbf{id}) : \tau) : \Gamma} \text{(TR\_ASSERTEXP\_OLD)}$$

$$\frac{\Gamma \vdash \mathbf{id} : \tau}{\Gamma \vdash (\mathbf{id} : \tau) : \Gamma} \text{(TR\_ASSERTEXP\_ID)}$$

$$\frac{}{\Gamma, \hat{v} : \tau \vdash (\hat{v} : \tau) : \Gamma} \text{(TR\_ASSERTEXP\_HATVAR\_IN)}$$

$$\frac{\hat{v} \notin \mathbf{dom}(\Gamma)}{\Gamma \vdash (\hat{v} : \tau) : \Gamma, \hat{v} : \tau} \text{(TR\_ASSERTEXP\_HATVAR\_IN)}$$

$$\frac{\Gamma \vdash (aexp_1 : \tau) : \Gamma_1 \quad \Gamma_1 \vdash (aexp_2 : \tau) : \Gamma'}{\Gamma \vdash (aexp_1 \mathit{op} aexp_2 : \tau) : \Gamma'} \text{(TR\_ASSERTEXP\_OP)}$$

where  $op \in \{+, -, *\}$ .

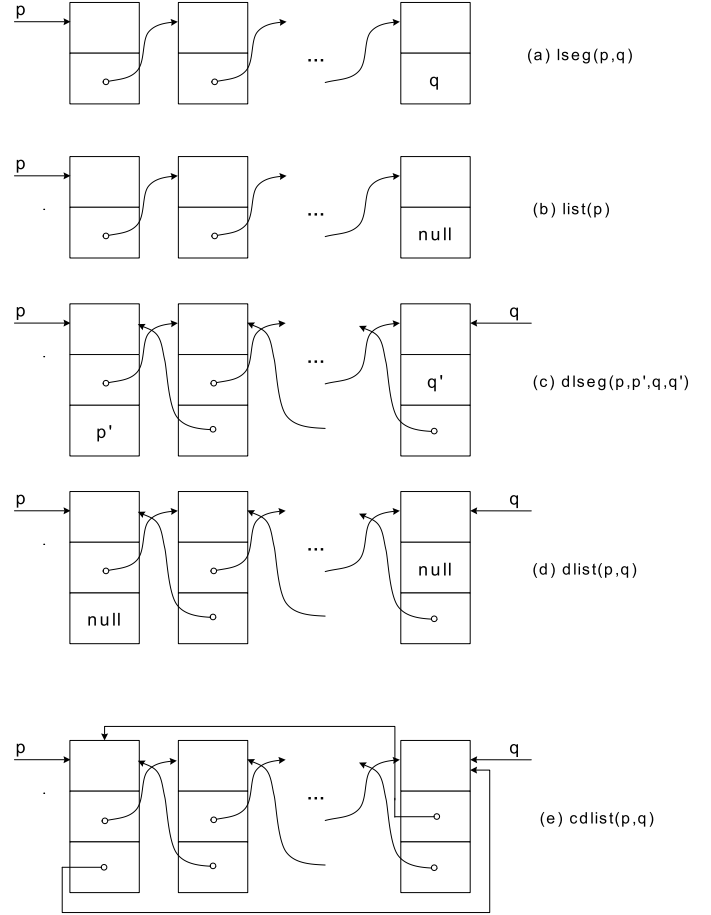


Figure 4. List Segment and List

## 2.2 Build-in Predicates

To use the build-in predicates support, the program must be typed and specified using corresponding types and predicates. And the prover can unfold or fold the inductive definitions using build-in inference rules for these predicates.

Next, the inductive definitions for supported data structures will be introduced.

### 2.2.1 Singly-linked List

A singly-linked list in Clike program must be typed with `struct list*`.

## 3. Specification Language

Specifications describe the behavior of programs, functions and statements. In this section, we will define the syntax and semantics of specifications. Then inference rules will be presented for proving valid specifications.

Each function has one pre-condition and one post-condition, which are assertions in SAL. We use  $\Delta$  as context of function specifications, that is,  $\Delta$  is a map from function identifier to its specification. The following judgements:

$$\begin{aligned}
\text{array}_\tau(\mathbf{a}, \mathbf{l}, \mathbf{h}) &\triangleq (\mathbf{l}=\mathbf{h} \wedge \text{emp}) \vee (\mathbf{h}>\mathbf{l} \wedge [\mathbf{a}, \mathbf{l}] \mapsto_\tau \hat{v} * \text{array}_\tau(\mathbf{a}, \mathbf{l}+\mathbf{1}, \mathbf{h})) \\
\text{lseg}(\mathbf{p}, \mathbf{q}) &\triangleq (\mathbf{p}=\mathbf{q} \wedge \text{emp}) \vee (\mathbf{p} \mapsto_{\text{list}} \{data : \hat{d}, next : \hat{r}\} * \text{lseg}(\hat{r}, \mathbf{q})) \\
\text{list}(\mathbf{p}) &\triangleq \text{lseg}(\mathbf{p}, \text{null}) \\
\text{tree}(\mathbf{p}) &\triangleq (\mathbf{p}=\text{null} \wedge \text{emp}) \vee (\mathbf{p} \mapsto_{\text{tree}} \{data : \hat{d}, left : \hat{l}, right : \hat{r}\} * \text{tree}(\hat{l}) * \text{tree}(\hat{r})) \\
\text{dlseg}(\mathbf{p}, \mathbf{p}', \mathbf{q}, \mathbf{q}') &\triangleq (\mathbf{p}=\mathbf{q} \wedge \mathbf{p}'=\mathbf{q}' \wedge \text{emp}) \vee (\mathbf{p} \mapsto_{\text{dlist}} \{data : \hat{d}, next : \hat{r}, prev : \mathbf{p}'\} * \text{dlseg}(\hat{r}, \mathbf{p}, \mathbf{q}, \mathbf{q}')) \\
\text{dlist}(\mathbf{p}, \mathbf{q}') &\triangleq \text{dlseg}(\mathbf{p}, \text{null}, \text{null}, \mathbf{q}') \\
\text{cdlist}(\mathbf{p}, \mathbf{q}') &\triangleq \text{dlseg}(\mathbf{p}, \mathbf{q}', \mathbf{p}, \mathbf{q}')
\end{aligned}$$

**Figure 3.** Inductive Definitions for Build-in Predicates

$$\begin{array}{ll}
\Gamma; \Delta \vdash \text{program } ok & \text{Program is well-formed} \\
\Gamma; \Delta \vdash \text{fundeflist } ok & \text{Function definition list is well-formed} \\
\Gamma; \Delta \vdash \text{fundef} & \text{Function is well-formed} \\
\Gamma; \Delta; f \vdash \{\mathbb{A}_{pre}\} \text{stm} \text{list} \{\mathbb{A}_{post}\} & \text{Function body of } f \text{ is well-formed}
\end{array}$$

### 3.1 Inference Rules

The inference rules are shown in Figure 5 and Figure 6.

Inference rules for well-formed programs and functions are shown in Figure 7. Auxiliary inference rules are shown in Figure 8.

$$\frac{\text{etoa}(C_1 \vee \dots \vee C_n, \Pi, \Sigma)}{\text{etoa}(C_1, \Pi, \Sigma) \vee \dots \vee \text{etoa}(C_n, \Pi, \Sigma)} \text{ (ETOA\_DNF)}$$

$$\frac{\text{etoa}(B_1 \wedge \dots \wedge B_n, \Pi, \Sigma) \quad \vdash B_i \bowtie B'_i (1 \leq i \leq n)}{B'_1 \wedge \dots \wedge B'_n \wedge \Pi \wedge \Sigma} \text{ (ETOA\_CONJ)}$$

$$\frac{}{\vdash \text{true} \bowtie \text{true}} \text{ (ETOA\_TRUE)}$$

$$\frac{}{\vdash \text{false} \bowtie \text{false}} \text{ (ETOA\_FALSE)}$$

$$\frac{\text{biop} \in \{==, !=, >=, >, <, <=\}}{\vdash \text{exp}_1 \text{ biop } \text{exp}_2 \bowtie \text{exp}_1 \text{ biop } \text{exp}_2} \text{ (ETOA\_BIN\_EXP)}$$

$$\frac{}{\vdash !( \text{exp}_1 == \text{exp}_2 ) \bowtie ( \text{exp}_1 != \text{exp}_2 )} \text{ (ETOA\_NOT\_EQ)}$$

$$\frac{}{\vdash !( \text{exp}_1 != \text{exp}_2 ) \bowtie ( \text{exp}_1 == \text{exp}_2 )} \text{ (ETOA\_NOT\_NOTEQ)}$$

$$\frac{}{\vdash !( \text{exp}_1 >= \text{exp}_2 ) \bowtie ( \text{exp}_1 < \text{exp}_2 )} \text{ (ETOA\_NOT\_GE)}$$

$$\frac{}{\vdash !( \text{exp}_1 <= \text{exp}_2 ) \bowtie ( \text{exp}_1 > \text{exp}_2 )} \text{ (ETOA\_NOT\_LE)}$$

$$\frac{}{\vdash !( \text{exp}_1 > \text{exp}_2 ) \bowtie ( \text{exp}_1 <= \text{exp}_2 )} \text{ (ETOA\_NOT\_GT)}$$

$$\frac{}{\vdash !( \text{exp}_1 < \text{exp}_2 ) \bowtie ( \text{exp}_1 >= \text{exp}_2 )} \text{ (ETOA\_NOT\_LT)}$$

$$\frac{}{\vdash !\text{true} \bowtie \text{false}} \text{ (ETOA\_NOT\_TRUE)}$$

$$\frac{}{\vdash !\text{false} \bowtie \text{true}} \text{ (ETOA\_NOT\_FALSE)}$$

## 4. Proof System

We have designed a verification condition generator (VCGen for short) for annotated Clike programs. The verification conditions (VC for short) are then input into the automated theorem prover.

### 4.1 Verification Condition

VCS are generated by VCGen using the inference rules shown in Section 3. The general form of VC is as following:

$$\mathbb{A}_1 \Rightarrow \mathbb{A}_2$$

### 4.2 Proof Rules

The rules for VC proving are listed below.

$$\frac{\Pi \wedge \Sigma \Rightarrow \Pi' \quad \Pi \wedge \Sigma \Rightarrow \Sigma'}{\Pi \wedge \Sigma \Rightarrow \Pi' \wedge \Sigma'} \text{ (PF\_VC\_SIMPL)}$$

$$\frac{\Pi \Rightarrow \Pi'}{\Pi \wedge \Sigma \Rightarrow \Pi'} \text{ (PF\_PI\_PI)}$$

$$\frac{\Pi \Rightarrow \Pi'}{\Pi \wedge \Sigma \Rightarrow \Pi'} \text{ (PF\_PI\_SGPI)}$$

## References

- [1] Zhaopeng Li, Yiyun Chen, Zhong Zhuang, Simin Yang. The Clike Programming Language Specification. Technique Report. April, 2009.

**Figure 8.** Auxiliary inference rules

$$\boxed{\Gamma; \Delta; f \vdash \{\mathbb{A}\} \text{stm} \text{list}\{\mathbb{A}'\}}$$

$$\frac{\Gamma; \Delta; f \vdash \{id == exp[\hat{x}/id] \wedge \Pi[\hat{x}/id] \wedge \Sigma[\hat{x}/id]\} \text{stm} \text{list}\{\mathbb{A}\} \quad \hat{x} \text{ is fresh}}{\Gamma; \Delta; f \vdash \{\Pi \wedge \Sigma\} id = exp; \text{stm} \text{list}\{\mathbb{A}\}} \quad (\text{IR\_ASSIGN\_SIMP})$$

$$\frac{\Gamma; \Delta; f \vdash \{id == aexp[\hat{x}/id] \wedge \Pi[\hat{x}/id] \wedge (\Sigma * id_1 \mapsto_{\tau} \{\dots, id_2 : aexp, \dots\})[\hat{x}/id]\} \text{stm} \text{list}\{\mathbb{A}\} \quad \hat{x} \text{ is fresh}}{\Gamma; \Delta; f \vdash \{\Pi \wedge \Sigma * id_1 \mapsto_{\tau} \{\dots, id_2 : aexp, \dots\} id = id_1 \rightarrow id_2; \text{stm} \text{list}\{\mathbb{A}\}} \quad (\text{IR\_ASSIGN\_STRUCT2ID})$$

$$\frac{\Gamma; \Delta; f \vdash \{id == aexp[\hat{x}/id] \wedge \Pi[\hat{x}/id] \wedge (\Sigma * [id_1, exp] \mapsto_{\tau} aexp)[\hat{x}/id]\} \text{stm} \text{list}\{\mathbb{A}\} \quad \hat{x} \text{ is fresh}}{\Gamma; \Delta; f \vdash \{\Pi \wedge \Sigma * [id_1, exp] \mapsto_{\tau} aexp\} id = id_1[exp]; \text{stm} \text{list}\{\mathbb{A}\}} \quad (\text{IR\_ASSIGN\_ARRAY2ID})$$

$$\frac{\Gamma; \Delta; f \vdash \{\Pi \wedge \Sigma * id \mapsto_{\tau} \{\dots, id_1 : exp, \dots\}\} \text{stm} \text{list}\{\mathbb{A}\}}{\Gamma; \Delta; f \vdash \{\Pi \wedge \Sigma * id \mapsto_{\tau} \{\dots, id_1 : aexp, \dots\} id \rightarrow id_1 = exp; \text{stm} \text{list}\{\mathbb{A}\}} \quad (\text{IR\_ASSIGN\_STRUCT})$$

$$\frac{\Gamma; \Delta; f \vdash \{\Pi \wedge \Sigma * [id, exp] \mapsto_{\tau} exp_1\} \text{stm} \text{list}\{\mathbb{A}\}}{\Gamma; \Delta; f \vdash \{\Pi \wedge \Sigma * [id, exp] \mapsto_{\tau} aexp\} id[exp] = exp_1; \text{stm} \text{list}\{\mathbb{A}\}} \quad (\text{IR\_ASSIGN\_ARRAY})$$

$$\frac{\Gamma; \Delta; f \vdash \{\Pi[\hat{x}/id] \wedge \Sigma[\hat{x}/id] * id \mapsto_s \{id_1 : \hat{y}_1, id_2 : \hat{y}_2, \dots, id_n : \hat{y}_n\}\} \text{stm} \text{list}\{\mathbb{A}\} \quad \Gamma \vdash s : (id_1 : \tau_1, id_2 : \tau_2, \dots, id_n : \tau_n) \quad \hat{x}, \hat{y}_i (1 \leq i \leq n) \text{ are fresh}}{\Gamma; \Delta; f \vdash \{\Pi \wedge \Sigma\} id = \text{malloc}(\text{sizeof}(\text{struct } s)); \text{stm} \text{list}\{\mathbb{A}\}} \quad (\text{IR\_MALLOC\_STRUCT})$$

$$\frac{\Gamma; \Delta; f \vdash \{\Pi \wedge \Sigma\} \text{stm} \text{list}\{\mathbb{A}\}}{\Gamma; \Delta; f \vdash \{\Pi \wedge \Sigma * exp \mapsto_{\tau} \{id_1 : aexp_1, id_2 : aexp_2, \dots, id_n : aexp_n\}\} \text{free}(exp); \text{stm} \text{list}\{\mathbb{A}\}} \quad (\text{IR\_FREE})$$

$$\frac{\Delta(id_{fun}) = (\mathbb{A}_{pre}, \mathbb{A}_{post}, (arg_1, \dots, arg_n)) \quad \Pi \wedge \Sigma \Rightarrow \Pi_r \wedge \mathbb{A}_{pre}[exp_1/arg_1][\dots/\dots][exp_n/arg_n] * \Sigma_r \quad \Gamma; \Delta; f \vdash \{\Pi_r \wedge \mathbb{A}_{post}[exp_1/old(arg_1)][\dots/\dots][exp_n/old(arg_n)] * \Sigma_r\} \text{stm} \text{list}\{\mathbb{A}\}}{\Gamma; \Delta; f \vdash \{\Pi \wedge \Sigma\} id_{fun}(exp_1, \dots, exp_n); \text{stm} \text{list}\{\mathbb{A}\}} \quad (\text{IR\_PROCALL})$$

$$\frac{\Delta(id_{fun}) = (\mathbb{A}_{pre}, \mathbb{A}_{post}, (arg_1, \dots, arg_n)) \quad \Pi \wedge \Sigma \Rightarrow \Pi_r \wedge \mathbb{A}_{pre}[exp_1/arg_1][\dots/\dots][exp_n/arg_n] * \Sigma_r \quad \Gamma; \Delta; f \vdash \{\Pi_r \wedge \mathbb{A}_{post}[\hat{x}/id][id/\text{res}][exp_1/old(arg_1)][\dots/\dots][exp_n/old(arg_n)] * \Sigma_{fr}\} \text{stm} \text{list}\{\mathbb{A}\} \quad \hat{x} \text{ is fresh}}{\Gamma; \Delta; f \vdash \{\Pi \wedge \Sigma\} id = id_{fun}(exp_1, \dots, exp_n); \text{stm} \text{list}\{\mathbb{A}\}} \quad (\text{IR\_FUNCALL})$$

$$\frac{\Delta(f) = (\mathbb{A}_{pre}, \mathbb{A}_{post}, (arg_1, \dots, arg_n)) \quad \Pi \wedge \Sigma \Rightarrow \mathbb{A}_{post}}{\Gamma; \Delta; f \vdash \{\Pi \wedge \Sigma\} \text{return}; \text{stm} \text{list}\{\mathbb{A}\}} \quad (\text{IR\_RETURN\_VOID})$$

$$\frac{\Delta(f) = (\mathbb{A}_{pre}, \mathbb{A}_{post}, (arg_1, \dots, arg_n)) \quad \text{res} == exp \wedge \Pi \wedge \Sigma \Rightarrow \mathbb{A}_{post}}{\Gamma; \Delta; f \vdash \{\Pi \wedge \Sigma\} \text{return } exp; \text{stm} \text{list}\{\mathbb{A}\}} \quad (\text{IR\_RETURN\_VAL})$$

$$\frac{\Gamma; \Delta; f \vdash \{\text{et} \text{oa}(\text{DNF}(exp), \Pi, \Sigma)\} \text{stm} \text{list}_1 \text{stm} \text{list}_2\{\mathbb{A}\} \quad \Gamma; \Delta; f \vdash \{\text{et} \text{oa}(\text{DNF}(!(\text{exp})), \Pi, \Sigma)\} \text{stm} \text{list}_2\{\mathbb{A}\}}{\Gamma; \Delta; f \vdash \{(\Pi \wedge \Sigma)\} \text{if}(exp)\{\text{stm} \text{list}_1\} \text{stm} \text{list}_2\{\mathbb{A}\}} \quad (\text{IR\_IF})$$

$$\frac{\Gamma; \Delta; f \vdash \{\text{et} \text{oa}(\text{DNF}(exp), \Pi, \Sigma)\} \text{stm} \text{list}_1 \text{stm} \text{list}\{\mathbb{A}\} \quad \Gamma; \Delta; f \vdash \{\text{et} \text{oa}(\text{DNF}(!(\text{exp})), \Pi, \Sigma)\} \text{stm} \text{list}_2 \text{stm} \text{list}\{\mathbb{A}\}}{\Gamma; \Delta; f \vdash \{(\Pi \wedge \Sigma)\} \text{if}(exp)\{\text{stm} \text{list}_1\} \text{else}\{\text{stm} \text{list}_2\} \text{stm} \text{list}\{\mathbb{A}\}} \quad (\text{IR\_IF\_ELSE})$$

$$\frac{\Gamma; \Delta; f \vdash \{\text{et} \text{oa}(\text{DNF}(exp), \mathbb{A}_{inv})\} \text{stm} \text{list}\{\mathbb{A}_{inv}\} \quad \Gamma; \Delta; f \vdash \{\text{et} \text{oa}(\text{DNF}(!(\text{exp})), \mathbb{A}_{inv})\} \text{stm} \text{list}_2\{\mathbb{A}\}}{\Gamma; \Delta; f \vdash \{(\Pi \wedge \Sigma)\} \text{while}[\mathbb{A}_{inv}](exp)\{\text{stm} \text{list}_1\} \text{stm} \text{list}_2\{\mathbb{A}\}} \quad (\text{IR\_WHILE})$$

$$\frac{\Gamma; \Delta; f \vdash \{\text{et} \text{oa}(\text{DNF}(exp), \mathbb{A}_{inv})\} \text{stm} \text{list}_1 \text{stm} \text{list}_2\{\mathbb{A}_{inv}\} \quad \Gamma; \Delta; f \vdash \{\text{et} \text{oa}(\text{DNF}(!(\text{exp})), \mathbb{A}_{inv})\} \text{stm} \text{list}_2\{\mathbb{A}\}}{\Gamma; \Delta; f \vdash \{\Pi \wedge \Sigma\} \text{for}[\mathbb{A}_{inv}](\text{stm} \text{list}_1; exp; \text{stm} \text{list}_2)\{\text{stm} \text{list}_1\} \text{stm} \text{list}_2\{\mathbb{A}\}} \quad (\text{IR\_FOR})$$

$$\frac{\mathbb{A} \Rightarrow \mathbb{A}'}{\Gamma; \Delta; f \vdash \{\mathbb{A}\} e\{\mathbb{A}'\}} \quad (\text{IR\_EMPTY\_STMTLIST})$$

Figure 5. Inference Rules of the Program Logic (a)

$$\boxed{\Gamma; \Delta; f \vdash \{\mathbb{A}\} \text{stmtlist}\{\mathbb{A}'\}}$$

$$\frac{\Gamma; \Delta; f \vdash \{\Pi \wedge \Sigma\} \text{stmtlist}\{\mathbb{A}'\} \quad \Gamma; \Delta \vdash \{\mathbb{A}\} \text{stmtlist}\{\mathbb{A}''\}}{\Gamma; \Delta \vdash \{(\Pi \wedge \Sigma) \vee \mathbb{A}\} \text{stmtlist}\{\mathbb{A}' \vee \mathbb{A}''\}} \quad (\text{IR\_CASE\_ANALYSIS})$$

$$\frac{\Gamma; \Delta; f \vdash \{\Pi \wedge \Sigma\} \text{stmtlist}\{\Pi' \wedge \Sigma'\}}{\Gamma; \Delta; f \vdash \{\Pi_r \wedge \Pi \wedge \Sigma * \Sigma_r\} \text{stmtlist}\{\Pi_r \wedge \Pi' \wedge \Sigma' * \Sigma_r\}} \quad (\text{IR\_FRAME\_RULE})$$

where no variable occurring free in  $\Pi_r \wedge \Sigma_r$  is modified by *stmtlist*.

**Figure 6.** Inference Rules of the Program Logic (b)

$$\boxed{\Gamma; \Delta \vdash \text{fundef ok}}$$

$$\frac{\Delta(f) = (\mathbb{A}_{pre}, \mathbb{A}_{post}, (arg_1, \dots, arg_n)) \quad \Gamma; \Delta; f \vdash \{\mathbb{A}_{pre}\} \text{stmtlist}\{\mathbb{A}_{post}\}}{\Gamma; \Delta \vdash \tau f(arg_1, \dots, arg_n) \{vardeclst \text{stmtlist}\} ok} \quad (\text{WF\_FUNCTION})$$

$$\boxed{\Gamma; \Delta \vdash \text{fundeflist ok}}$$

$$\frac{}{\Gamma; \Delta \vdash \epsilon ok} \quad (\text{WF\_FUNLIST\_NIL})$$

$$\frac{\Gamma; \Delta \vdash \text{fundef ok} \quad \Gamma; \Delta \vdash \text{fundeflist ok}}{\Gamma; \Delta \vdash \text{fundef fundeflist ok}} \quad (\text{WF\_FUNLIST\_CONS})$$

$$\boxed{\Gamma; \Delta \vdash \text{program ok}}$$

$$\frac{\Gamma; \Delta \vdash \text{fundeflist ok}}{\Gamma; \Delta \vdash \text{structdeflist vardeclst fundeflist ok}} \quad (\text{WF\_PROGRAM})$$

**Figure 7.** Well-formed Programs and Functions