Siberian Division of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

I. V. Maryasov

THE MIXED AXIOMATIC SEMANTICS METHOD

Preprint
160

Novosibirsk 2011

This work represents a mixed axiomatic semantics method suggested for C-light program verification. This semantics has the unambiguity of inference (i. e. one and only one inference rule is allowed at each step) and also has special variants of inference rules for the same program statement which are applied depending on the variable type and static information. This approach allows significant simplification of verification conditions.

This work was partially supported by Russian Foundation for Basic Research under grant 11-01-00028.

Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова

И. В. Марьясов

МЕТОД СМЕШАННОЙ АКСИОМАТИЧЕСКОЙ СЕМАНТИКИ

Препринт
160

Новосибирск 2011

В данной работе представлен метод смешанной аксиоматической семантики, предлагаемый для верификации C-light программ. Данная семантика обладает однозначностью вывода (т. е. на каждом шаге допустимо одно и только одно правило вывода), а также имеет специальные варианты правил вывода для одной и той же программной конструкции, которые применяются в зависимости от типа переменной и дополнительной статической информации. Такой подход позволяет существенно упрощать условия корректности верифицируемых программ.

Работа частично поддержана грантом РФФИ № 11-01-00028.

1. INTRODUCTION

The two-level C program verification scheme [1, 3, 5–11] is applied to C-light language. It is a powerful subset of ISO C [4] language. The valid base types of C-light are void, integer types, real types and enumerations. The derived types are pointers, arrays, structures and functions. C-light covers all C statements. There are two limitations on C-light statements: all `case`-labels and the `default` label of a `switch` statement must be at the same level; a jump into a block by a `goto` statement is prohibited. The expression evaluation order is fixed strictly in C-light. The arguments of operators and functions are evaluated right-to-left and initializing expressions lists are evaluated left-to-right.

The peculiarity of C-light is that it has formal operational semantics.

To verify C-light programs, we translate them into C-kernel programs. C-kernel language is a subset of C-light. It has axiomatic semantics. All expressions in C-kernel are in a normal form. The number of side effects is reduced to minimum and the operators with control points (for example, logical operators) are absent. A normalized expression does not contain conditional operators, comma operators, logical operators, simple and compound assignments, increment and decrement operators. The declaration lists are allowed only in function declarations, any other declaration defines one object exactly. The initializers contain only normalized expressions. All static objects' names are unique.

The C-kernel statements are the following: expression statements, `if` statements with a mandatory else branch and normalized condition, while statements with a normalized condition, jump statements `goto` and return, and compound statements.

This work represents a mixed operational semantics of C-light, a mixed axiomatic semantics of C-kernel and a proof of the consistency theorem. This semantics has an unambiguity of inference (i. e. one and only one inference rule is allowed at each step of analysis). The word "mixed" means that it has special inference rules which are applied depending on the information in annotations about the program construction. In many cases this allows verification conditions to be simplified. For example, we have a special inference rule for the assignment operator in the case when there is no aliasing.

2. THE MODIFIED OPERATIONAL SEMANTICS OF C-LIGHT

A formal definition of C-light was given in terms of operational semantics in Plotkin style [10]. A state of the abstract C-light machine (ACM) is a map which defines the following meta-variables:

1. MD — a variable of the type $\mathsf{Locations} \to \mathsf{CTypes}$.
2. Val — a variable of the type $\mathsf{CTypes} \times \mathsf{LogTypeSpecs}$.
3. Variables of the type Names.

The meta-variable MD defines the values stored in the memory. Locations is the set of addresses and CTypes is the union of all valid C-light types.

Names is a set of all program names and LogTypeSpecs is a set of logical names of types which are logical representations of abstract names of types.

The meta-variable Val stores the value and its type returning by a function or by an expression.

The set of all states is denoted by States. Let Greek letters σ, τ with or without indices stand for states. $\mathsf{MD}_\sigma(\mathsf{c})$ is a short form of $\sigma(\mathsf{MD})(\mathsf{c})$.

If the value of a variable is not used by pointers, we will call it a non-shared variable. All other variables will be called shared variables.

Let us introduce ACM parameters. The first group consists of functions the definition of which depends on a compiler.

The function mem(x) returns the address of a shared variable x:

$$\mathsf{mem}\colon \mathsf{ID} \to \mathsf{Locations}.$$

If we want to express that x is equal to 5, we should write $\mathsf{MD}(\mathsf{mem}(\mathsf{x})) = 5$. In the case when x is a non-shared variable, the function mem is undefined. And if we want to express that x is equal to 5, we should write $\mathsf{x} = 5$.

The function mb(e, id) evaluates the address of an array member e[id] or a structure member e.id:

$$\mathsf{mb}\colon \mathsf{ID} \times (\mathsf{N} \cup \mathsf{ID}) \to \mathsf{Locations}.$$

The function $\mathsf{cast}(\mathsf{e}, \tau, \tau')$ transforms a value of the type τ into a value of the type τ'.

C-light language inherits operations from C. In order to define evaluation of these operations symbolically, the functions UnOpSem and BinOpSem [10] are used.

The function $\mathsf{UnOpSem}(\bullet, \mathsf{v}, \tau)$ returns the result of applying the unary operation $\bullet$ to the value v of the type τ. The result has the form $\mathsf{Val}(\mathsf{v}', \tau')$ where v' is the value and τ' is its type.

The function $\mathsf{BinOpSem}(\bullet, \mathsf{v}_1, \tau_1, \mathsf{v}_2, \tau_2)$ returns the result of applying the binary operation $\bullet$ to the values v_1 and v_2 of types τ_1 and τ_2 respectively. The result has the form $\mathsf{Val}(\mathsf{v}', \tau')$, where v' is the value and τ' is its type.

The second group consists of functions which can be evaluated directly from the source code.

The function std juxtaposes an identifier with the type connected with it in the typedef-declaration, structure or enumeration declaration.

The function id(e) returns the identifier of the object which is declared in the declaration e.

The function tp(e) returns the type of the object which is declared in the declaration e.

The function labels(S) returns the number of labels at the top level of the statement sequence S except for case and default labels.

The function storage(e) returns the storage-class specifier of the declaration e:

- storage(e) = auto, if e contains the specifier auto or register;
- storage(e) = static, if e contains the specifier static;
- storage(e) = static, if e does not contain storage-class specifiers and e is at the external program level;
- storage(e) = auto, if e does not contain storage-class specifiers and e is not at the external program level.

A number of special abstract functions are also used to define how the abstract C-light machine works.

The function retType gets a logical type τ as an argument and returns the type τ_0, if τ has the form $\tau_1 \times \ldots \times \tau_n \to \tau_0$ or it returns τ otherwise.

The function defaultValue(τ, strg) returns the default value for the type τ in compliance with the storage-class specifier strg:

- defaultValue(τ, static) = default value for the type τ;
- defaultValue(τ, auto) = ω.

The function TypeofNewVal(τ) returns the type of a memory cell value created by a new operation for the derived type τ:

- TypeofNewVal(array(τ, k)) = τ;
- TypeofNewVal(struct(s, (τ_1, m_1), ..., (τ_n, m_n))) = struct s.

The function findex(τ) returns the first index of the derived type τ:

- $\text{findex}(\tau) = 0$ if τ is an array;
- $\text{findex}(\text{struct}(s, (\tau_1, l_1), \ldots, (\tau_k, l_k))) = l_1$.

The function $\text{next}(\tau, l)$ returns the index of the derived type τ which follows immediately after the index l:

- $\text{next}(\text{array}(\tau, k), l) = l + 1$ if $l < k - 1$;
- $\text{next}(\text{array}(\tau, k), l) = \omega$ if $l \geq k - 1$;
- $\text{next}(\text{struct}(s, (\tau_1, l_1), \ldots, (\tau_k, l_k)), l_i) = l_{i+1}$ if $i < k$;
- $\text{next}(\text{struct}(s, (\tau_1, l_1), \ldots, (\tau_k, l_k)), l_i) = \omega$ if $i \geq k$.

The function $\text{itype}(\tau, i)$ returns the type of the derived type τ element with the index i:

- $\text{itype}(\tau[k], i) = \tau$;
- $\text{itype}(\text{struct}(s, (\tau_1, l_1), \ldots, (\tau_k, l_k)), i) = \tau_i$.

The declaration

$$e = e';$$

is in a normal form, if e' is an initializer in a fully bracketed form which does not contain designators and the object which is declared in e is of a full type.

For example, the declarations

```
int y[4][3] = {{1, 3, 5}, {2, 4, 6}, {3, 5, 7}};
```

and

```
int y[4][3] = {1, 3, 5, 2, 4, 6, 3, 5, 7};
```

contain equivalent compound initializers but only the first one is in the normal form.

The declaration

$$d = d';$$

is the normal form of the declaration

e = e';

if

1. d = d'; is in the normal form.
2. These declarations initialize the same object with the same value.

According to [4], any declaration can be transformed to its normal form. The Boolean function isNform(d = d'; , e = e';) takes the value true if and only if the declaration

d = d';

is the normal form of the declaration

e = e';.

Let A = (A_1, A_2). The functions fst(A) and snd(A) are the projections: fst(A) = A_1, and snd(A) = A_2.

The function family type evaluates the type of C-light constructions. It includes three functions.

The function type(u) evaluating the type of the constant u, is defined by the following axioms which are split into three groups.

The first group of nine axioms defines numerical constant types in a standard way (octal and hexadecimal constants are not considered for short). As is usual for signed integers, the key word signed is omitted:

- type(n) is one of {int, long int, long long int};
- type(nU) is one of {unsigned int, unsigned long int, unsigned long long int};
- type(nL) is one of {long int, long long int};
- type(nUL) is one of {unsigned long int, unsigned long long int};
- type(nLL) = long long int;
- type(nULL) = unsigned long long int;
- type([n_1].[n_2][E [±] n_3]) = double;
- type([n_1].[n_2][E [±] n_3]F) = float;
- type([n_1].[n_2][E [±] n_3]L) = long double.

The words “one of” mean that the first type is selected which is enough to store the constant. The square brackets stand for optional elements.

The second group of four axioms defines the character and string constant types:

- type('c') = int;
- type(L'c_1c_2') = unsigned short;
- type("$c_1...c_n$") = array(char, n + 1);
- type(L"$c_1...c_n$") = array(unsigned short, n / 2 + 1).

The first axiom reflects the C language feature: character constants are of integer type (though strings are arrays of type char). In this semantics, simple constants consisting of more than one symbol or containing non-standard escape-sequences (i. e. not corresponding to any one-byte symbol) are not considered because their values depend on complier implementation.

The third group consists of the axiom for a null pointer:

$$\text{type(NULL)} = \text{void}^*.$$

The function type(•, τ_1, τ_2) evaluates the type of the value returned by the operator • in the case when it has the arguments of types τ_1 and τ_2. The mandatory argument τ_2 is used for a binary and conditional operator. Semantics of this function is defined according to ISO C standard [4].

Let x be a variable and c be a constant. The function type(e) evaluates the type of the expression e:

- type(x) = τ if the declaration τ x exists;
- type(c) = type(c);
- type(• e) = type(•, (type(e));
- type(e_1 • e_2) = type(•, type(e_1), type(e_2));
- type(e_1 ? e_2 : e_3) = type(? :, type(e_2), type(e_3));
- type((e)) = type(e);
- type(e(e_1, ..., e_n)) = τ if type(e) = $\tau_1 \times ... \times \tau_n \rightarrow \tau$.

The function logtype(τ, MD) converts the abstract name of the type τ into the corresponding logical type name according to the value of meta-variable MD:

$$\text{logtype}(\tau, \text{MD}) = \tau',$$

where τ' is the logical type name corresponding to the abstract name of the type τ.

The function addr(e, MD) evaluates the address of the expression e according to the MD value:

- addr(e, MD) = mem(e), if e is a shared variable;
- addr(e, MD) = ω, if e is a constant or a non-shared variable;
- addr(e[e'], MD) = mb(e, val(e', MD));
- addr(e[e'], MD) = ω, if e is a non-shared array;
- addr(e.m, MD) = mb(val(e, MD), val(m, MD));
- addr(e.m, MD) = ω, if e is a non-shared structure;
- addr(&e, MD) = ω;
- addr(*e, MD) = val(e, MD).

The function val(e, MD) returns the value of the expression e according to the values of the meta-variable MD:

- val(e, MD) = MD(mem(e)), if e is a shared variable;
- val(e, MD) = e, if e is a constant or a non-shared variable;
- val(e[e'], MD) = MD(mb(e, val(e', MD)));
- val(e[e'], MD) = e[val(e', MD)], if e is the non-shared array;
- val(e.m, MD) = MD(mb(val(e, MD), val(m, MD)));
- val(e.m, MD) = e.m, if e is a non-shared structure;
- val(&e, MD) = addr(e, MD);
- val(*e, MD) = MD(val(e, MD));
- val((τ) e, MD) = cast(val(e, MD), type(e), fst(logtype(τ, MD)));
- val(• e, MD) = UnOpSem(•, val(e, MD), type(e)) , where • is a non-logical unary operation;
- val(e • e', MD) = BinOpSem(•, val(e, MD), type(e), val(e', MD), type(e')) where • is the non-logical binary operation;
- val(• e, MD) = • (val(e, MD)) , where • is a logical unary operation;
- val(e • e', MD) = val(e, MD) • val(e', MD) , where • is a logical binary operation.

The function logval(e, MD) analyses the expression e and performs its infiltration if it is of the Boolean type.

- logval(e, MD) = val(e, MD), if e is Boolean;
- logval(e, MD) = (cast(val(e, MD), type(e), int) $\neq$ 0), if e is not Boolean.

Let τ_1 be a simple type and τ_2 be a derived type. The function init(τ, e, MD) initializes the cell of the type τ in accordance with the initialization specifier e and the value of the meta-variable MD modifying it. The initialization specifier can be either an initializer or a storage-class specifier (if there is no initializer). In the last case initialization is performed by default. The function init is defined in the following way:

- init(τ_1, storage, MD) = (MD, defaultValue(τ_1, storage));
- init(τ_1 (v, τ), MD) = (MD, cast(v, τ, τ_1));
- init(τ_1, e, MD) = (MD, cast(val(e, MD), type(e), τ_1));
- init(τ_2, e, MD) = (updv(MD, nc, τ_2, e), nc), where MD(nc) = ω.

The function updv(MD, nc, τ, e) modifies the meta-variable MD. This modification specifies the values of the allocated cells and defines the objects of nested derived types (if any) by calling the function init. Thus the functions updv and init are mutually recursive:

- updv(MD, nc, τ, e) = updv(MD, nc, τ, e, findex(τ));
- updv(MD, nc, τ, storage, l) =
  updv(upd(MD', mb(nc, l), Val'), nc, τ, storage, next(l)),
  if l $\neq$ ω and (MD', Val') = init(itype(τ, l), storage, MD);
- updv(MD, nc, τ, {e_1, …, e_k}, l) =
  updv(upd(MD', mb(nc, l), Val'), nc, τ, {e_2, …, e_k}, next(l)),
  if l $\neq$ ω and (MD', Val') = init(itype(τ, l), e_1, MD);
- updv(MD, nc, τ, { }, l) =
  updv(upd(MD', mb(nc, l), Val'), nc, τ, { }, next(l)),
  if l $\neq$ ω and (MD', Val') = init(itype(τ, l), static, MD);
- updv(MD, nc, τ, e, ω) = MD.

The configuration of the abstract C-light machine is a pair $\langle$P, $\sigma\rangle$, where P is a program and σ is a state.

The axioms of operational semantics have the form

$$\langle A, \sigma\rangle \rightarrow \langle B, \sigma'\rangle$$

It means that one execution step of a program fragment A started in the state σ leads to the state σ' and B is the fragment remained to execute. All semantics rules have the form

$$\frac{P_1, ..., P_n}{\langle A, \sigma\rangle \rightarrow \langle B, \sigma'\rangle}$$

It means that, when the conditions P_1, ..., P_n hold, we can pass from the first configuration to the second. Thus program execution in operational semantics leads to configuration chains which can be infinite in the general case:

$$\langle S_1, \sigma_1\rangle \rightarrow \langle S_2, \sigma_2\rangle \rightarrow ... \rightarrow \langle S_i, \sigma_i\rangle \rightarrow ...$$

The configuration $\langle S_1, \sigma_1\rangle$ is called an initial configuration. If the chain is finite and $\langle S_n, \sigma_n\rangle$ is the last configuration of this chain, then we say that execution of the fragment S_1 started in the state σ_1 leads to the final configuration $\langle S_n, \sigma_n\rangle$. If we designate the transitive reflexive closure of the relation $\rightarrow$ as $\rightarrow^*$, then in the general case such execution can be designated as

$$\langle S_1, \sigma_1\rangle \rightarrow^* \langle S_n, \sigma_n\rangle.$$

The empty program fragment will be denoted by ε. The empty fragment can be both an empty program and empty expression.

The assignment operator. Let τ'' be a type different from array and e be a shared variable. The rule for a simple assignment has the form:

$$\frac{\langle e_0, \sigma\rangle \rightarrow^* \langle Val(v', \tau'), \sigma'\rangle, \langle e, \sigma'\rangle \rightarrow^* \langle Val(v'', \tau''), \sigma''\rangle, \quad v = cast(v', \tau', \tau''),\ c = addr(v'')}{\langle e = e_0, \sigma\rangle \rightarrow \langle Val(v, \tau''), \sigma''(MD \leftarrow upd(MD, c, v))\rangle}$$

If e is a non-shared variable, the rule has the form:

$$\frac{\langle e_0, \sigma\rangle \rightarrow^* \langle Val(v', \tau'), \sigma'\rangle, \langle e, \sigma'\rangle \rightarrow^* \langle Val(e, \tau''), \sigma''\rangle, v = cast(v', \tau', \tau'')}{\langle e = e_0, \sigma\rangle \rightarrow \langle Val(v, \tau''), \sigma''(e \leftarrow v)\rangle}$$

The rule for a compound assignment operator has the form:

$$\frac{\begin{array}{c}\langle e_0, \sigma\rangle \to^* \langle Val(v', \tau'), \sigma'\rangle, \langle e, \sigma'\rangle \to^* \langle Val(v'', \tau''), \sigma''\rangle,\\ v = cast(fst(BinOpSem(\bullet, v'', \tau'', v', \tau')),\\ snd(BinOpSem(\bullet, v'', \tau'', v', \tau')), \tau'')\end{array}}{\langle e \bullet\!= e_0, \sigma\rangle \to \langle Val(v, \tau''), \sigma''(MD \leftarrow upd(MD, addr(v''), v))\rangle}$$

Variables and constants. Semantics of a variable and constant evaluation is defined by one axiom:

$$\langle x, \sigma\rangle \to \langle Val(val(x, MD_\sigma), type(x)), \sigma\rangle$$

Access to the elements of compound types. Let τ' be the integer type. The rule for access to the elements of an array has the form:

$$\frac{\langle e_0, \sigma\rangle \to^* \langle Val(v, \tau'), \sigma'\rangle, \langle e, \sigma'\rangle \to^* \langle Val(c, \tau^*), \tau''), \sigma''\rangle}{\langle e[e_0], \sigma\rangle \to \langle Val(MD_{\sigma''}(mb(MD_{\sigma''}(c), v)), \tau), \sigma''\rangle}$$

The rule for selection of a structure field has the form:

$$\frac{\langle e, \sigma\rangle \to^* \langle Val(v, \tau), \sigma'\rangle}{\langle e.m, \sigma\rangle \to \langle Val(MD_{\sigma'}(mb(v, m)), \tau), \sigma'\rangle}$$

Indirection operator. The rule for the indirection operator has the form:

$$\frac{\langle e, \sigma\rangle \to^* \langle Val(v, \tau^*), \sigma'\rangle}{\langle {}^*e, \sigma\rangle \to \langle Val(MD_{\sigma'}(v), \tau), \sigma'\rangle}$$

Address operator. The rule for the address operator has the form:

$$\frac{\langle e, \sigma\rangle \to^* \langle Val(v, \tau), \sigma'\rangle}{\langle \&e, \sigma\rangle \to \langle Val(addr(v), \tau^*), \sigma'\rangle}$$

Cast operators. The rule for the cast operator has the form:

$$\frac{\langle e, \sigma\rangle \to^* \langle Val(v, \tau'), \sigma'\rangle}{\langle (\tau)e, \sigma\rangle \to \langle Val(cast(v, \tau', \tau), \tau), \sigma'\rangle}$$

Comma operator. Assume that an expression e does not contain the comma operator at the top level. The rule for the comma operator has the form:

$$\frac{\langle e, \sigma\rangle \to^{*} \langle v, \sigma'\rangle}{\langle e, e', \sigma\rangle \to \langle e', \sigma'\rangle}$$

Logical operators. Let τ be a scalar type. Expressions containing logical operators AND and OR can be evaluated incompletely. For their evaluation, an auxiliary construction OrAnd is introduced.

The rule for AND has the form:

$$\frac{\langle e_1, \sigma\rangle \to^{*} \langle \mathrm{Val}(v, \tau), \sigma'\rangle,\ \mathrm{cast}(v, \tau, \mathrm{int}) = 0}{\langle e_1 \mathbin{\&\&} e_2, \sigma\rangle \to \langle \mathrm{Val}(0, \mathrm{int}), \sigma'\rangle}$$

$$\frac{\langle e_1, \sigma\rangle \to^{*} \langle \mathrm{Val}(v, \tau), \sigma'\rangle,\ \mathrm{cast}(v, \tau, \mathrm{int}) \neq 0}{\langle e_1 \mathbin{\&\&} e_2, \sigma\rangle \to \langle \mathrm{OrAnd}(e_2), \sigma'\rangle}$$

The rule for OR has the form:

$$\frac{\langle e_1, \sigma\rangle \to^{*} \langle \mathrm{Val}(v, \tau), \sigma'\rangle,\ \mathrm{cast}(v, \tau, \mathrm{int}) \neq 0}{\langle e_1 \mathbin{||} e_2, \sigma\rangle \to \langle \mathrm{Val}(1, \mathrm{int}), \sigma'\rangle}$$

$$\frac{\langle e_1, \sigma\rangle \to^{*} \langle \mathrm{Val}(v, \tau), \sigma'\rangle,\ \mathrm{cast}(v, \tau, \mathrm{int}) = 0}{\langle e_1 \mathbin{||} e_2, \sigma\rangle \to \langle \mathrm{OrAnd}(e_2), \sigma'\rangle}$$

Semantics of OrAnd operator is defined by two rules:

$$\frac{\langle e, \sigma\rangle \to^{*} \langle \mathrm{Val}(v, \tau), \sigma'\rangle,\ \mathrm{cast}(v, \tau, \mathrm{int}) = 0}{\langle \mathrm{OrAnd}(e), \sigma\rangle \to \langle \mathrm{Val}(0, \mathrm{int}), \sigma'\rangle}$$

$$\frac{\langle e, \sigma\rangle \to^{*} \langle \mathrm{Val}(v, \tau), \sigma'\rangle,\ \mathrm{cast}(v, \tau, \mathrm{int}) \neq 0}{\langle \mathrm{OrAnd}(e), \sigma\rangle \to \langle \mathrm{Val}(1, \mathrm{int}), \sigma'\rangle}$$

Increment and decrement operators. The prefix increment and decrement operators are defined by the following axioms:

$$\langle {+}{+}e, \sigma\rangle \to \langle e \mathrel{+}= 1, \sigma\rangle$$
$$\langle {-}{-}e, \sigma\rangle \to \langle e \mathrel{-}= 1, \sigma\rangle$$

Let τ be a type different from array. The rules for postfix increment and decrement operators have the form:

$$\frac{\langle e, \sigma\rangle \to^{*} \langle \mathrm{Val}(v, \tau^{*}), \sigma'\rangle}{\begin{array}{c}\langle e{+}{+}, \sigma\rangle \to \langle \mathrm{Val}(v, \tau),\\ \sigma'(\mathrm{MD} \leftarrow \mathrm{upd}(\mathrm{MD}_{\sigma'}, \mathrm{addr}(v), \mathrm{fst}(\mathrm{BinOpSem}(+, v, \tau, 1, \mathrm{int}))))\rangle\end{array}}$$

$$\frac{\langle e, \sigma\rangle \to^{*} \langle \mathrm{Val}(v, \tau^{*}), \sigma'\rangle}{\begin{array}{c}\langle e{-}{-}, \sigma\rangle \to \langle \mathrm{Val}(v, \tau),\\ \sigma'(\mathrm{MD} \leftarrow \mathrm{upd}(\mathrm{MD}_{\sigma'}, \mathrm{addr}(v), \mathrm{fst}(\mathrm{BinOpSem}(-, v, \tau, 1, \mathrm{int}))))\rangle\end{array}}$$

Conditional operator. Let τ_0 be a scalar type. The rules for a conditional operator have the form:

$$\frac{\begin{array}{c}\tau = \mathrm{type}(?:, \mathrm{type}(e_1), \mathrm{type}(e_2)),\\ \langle e_0, \sigma\rangle \to^{*} \langle \mathrm{Val}(v, \tau_0), \sigma'\rangle, \mathrm{cast}(v, \tau_0, \mathrm{int}) \neq 0\end{array}}{\langle e_0 \;?\; e_1 : e_2, \sigma\rangle \to \langle (\tau)\, e_1, \sigma'\rangle}$$

$$\frac{\begin{array}{c}\tau = \mathrm{type}(?:, \mathrm{type}(e_1), \mathrm{type}(e_2)),\\ \langle e_0, \sigma\rangle \to^{*} \langle \mathrm{Val}(v, \tau_0), \sigma'\rangle, \mathrm{cast}(v, \tau_0, \mathrm{int}) = 0\end{array}}{\langle e_0 \;?\; e_1 : e_2, \sigma\rangle \to \langle (\tau)\, e_2, \sigma'\rangle}$$

Other unary operators. The unary operators which do not have their own rules are defined by a general rule:

$$\frac{\langle e, \sigma\rangle \to^{*} \langle \mathrm{Val}(v, \tau), \sigma'\rangle}{\langle \bullet e, \sigma\rangle \to \langle \mathrm{Val}(\mathrm{UnOpSem}(\bullet, v, \tau)), \sigma'\rangle}$$

Other binary operators. The binary operators which do not have their own rules are defined by a general rule:

$$\frac{\langle e_2, \sigma_0\rangle \to^{*} \langle \mathrm{Val}(v_2, \tau_2), \sigma_1\rangle, \langle e_1, \sigma_1\rangle \to^{*} \langle \mathrm{Val}(v_1, \tau_1), \sigma_2\rangle}{\langle e_1 \bullet e_2, \sigma\rangle \to \langle \mathrm{Val}(\mathrm{BinOpSem}(\bullet, v_1, \tau_1, v_2, \tau_2)), \sigma_2\rangle}$$

Variable declarations. Semantics of a variable declaration is defined by three rules: one for a declaration without initialization and two for a declaration with an initializer.

Let e; be a variable declaration without a specifier enum and containing a single declarator without initializer. The rule for a variable declaration without an initializer has the form:

$$\frac{\tau \in \mathrm{logtype}(e, MD_\sigma),\ MD_\sigma(nc) = \bot,\\ (MD', V) \in \mathrm{init}(\tau, \mathrm{storage}(e), \mathrm{upd}(MD_\sigma, nc, \omega))}{\langle e;\ , \sigma\rangle \to \langle \varepsilon, \sigma(MD \leftarrow \mathrm{upd}(MD', nc, \mathrm{fst}(V)))\rangle}$$

Let e = e'; be a variable declaration not including a specifier enum and containing a single declarator with an initializer. The rules for a variable declaration with an initializer have the form:

$$\frac{\mathrm{isNform}(d = d';, e = e';),\ \langle \mathrm{computeInit}(d'), \sigma\rangle \to \langle \mathrm{Val}(v), \sigma'\rangle,\ v \neq \omega,\\ MD_\sigma(nc) = \bot,\ \tau \in \mathrm{logtype}(d, MD_\sigma),\ (MD', V) \in \mathrm{init}(\tau, v, \mathrm{upd}(MD_\sigma, nc, \omega))}{\langle e = e';\ , \sigma\rangle \to \langle \varepsilon, \sigma(MD \leftarrow \mathrm{upd}(MD', nc, \mathrm{fst}(V)))\rangle}$$

$$\frac{\mathrm{isNform}(d = d';, e = e';),\ \langle \mathrm{computeInit}(d'), \sigma\rangle \to \langle \mathrm{Val}(\omega), \sigma'\rangle}{\langle e = e';\ , \sigma\rangle \to \langle \mathrm{Val}(\omega), \sigma'\rangle}$$

The auxiliary construction computeInit(e) evaluates all expressions' values entering the initializer e left-to-right:

$$\frac{\langle \mathrm{computeInit}(e_1), \sigma_0\rangle \to \langle \mathrm{Val}(v_1), \sigma_1\rangle,\ v_1 \neq \omega,\\ \ldots\\ \langle \mathrm{computeInit}(e_k), \sigma_{k-1}\rangle \to \langle \mathrm{Val}(v_k), \sigma_k\rangle,\ v_k \neq \omega}{\langle \mathrm{computeInit}(\{e_1, \ldots, e_k\}), \sigma_0\rangle \to \langle \mathrm{Val}(\{v_1, \ldots, v_k\}), \sigma_k\rangle}$$

$$\frac{\langle \mathrm{computeInit}(e_1), \sigma_0\rangle \to \langle \mathrm{Val}(v_1), \sigma_1\rangle,\ v_1 \neq \omega,\\ \ldots\\ \langle \mathrm{computeInit}(e_m), \sigma_{m-1}\rangle \to \langle \mathrm{Val}(v_m), \sigma_m\rangle,\ v_m \neq \omega,\ m < k,\\ \langle \mathrm{computeInit}(e_{m+1}), \sigma_m\rangle \to \langle \mathrm{Val}(v_k), \sigma_k\rangle,\ v_{m+1} = \omega}{\langle \mathrm{computeInit}(\{e_1, \ldots, e_k\}), \sigma_0\rangle \to \langle \mathrm{Val}(\omega), \sigma_k\rangle}$$

Let the initializer e be not in brackets. Then

$$\langle \mathrm{computeInit}(e), \sigma\rangle \to \langle e, \sigma\rangle$$

Variable declarations with the specifier enum. Semantics of a variable declaration with the specifier enum is defined as follows:

$$\langle \text{storage enum x \{e\} e'}, \sigma\rangle \rightarrow \langle \text{storage enum x \{e,\} e'}, \sigma\rangle,$$

if e is not ended by a comma.

$$\langle \text{storage enum \{e\} e'}, \sigma\rangle \rightarrow \langle \text{storage enum \{e,\} e'}, \sigma\rangle,$$

if e is not ended by a comma.

$$\langle \text{storage enum x \{e,\} e'}, \sigma\rangle \rightarrow \langle \text{storage enum x e, e'}, \sigma\rangle$$

$$\frac{\text{enum(x, ...)} \in \text{logtype(enum\{e\}}, MD_\sigma)}{\langle \text{storage enum \{e, \} e'}, \sigma\rangle \rightarrow \langle \text{storage x e, e'}, \sigma\rangle}$$

Type declaration. The axiom for a type declaration has the form:

$$\langle \text{typedef e}, \sigma\rangle \rightarrow \langle \varepsilon, \sigma\rangle$$

Function declaration. The rule for a function declaration has the form:

$$\frac{\tau' \in \text{logtype}(\tau\ f(\tau_1\ x_1, \ldots, \tau_n\ x_n), MD_\sigma),\ MD_\sigma(nc) = \bot}{\langle \tau\ f(\tau_1\ x_1, \ldots, \tau_n\ x_n)\{S\}, \sigma\rangle \rightarrow \langle \varepsilon, \sigma(MD \leftarrow \text{upd}(MD_\sigma, nc, (f, [x_1, \ldots, x_n], S)))\rangle}$$

Labeled statement. Besides regular labels, the statements can be labeled by the labels case and default.

A statement labeled by a label L is executed either in a normal sequential program run or when it catches the exception Exc(gotoStart(L)) raised by the statement goto L:

$$\langle L\text{: } T, \sigma\rangle \rightarrow \langle T, \sigma\rangle$$

$$\langle \text{Exc(gotoStart(L)) } L\text{: } T, \sigma\rangle \rightarrow \langle T, \sigma\rangle$$

A statement labeled by the label case is executed either in a normal sequential program run or when it catches the exception Exc(switchStart(c, τ)) raised by the switch statement:

$$\langle \mathsf{case}\ e\colon T, \sigma\rangle \to \langle T, \sigma\rangle$$

$$\frac{\mathsf{cast}(\mathsf{val}(e, MD_\sigma), \mathsf{type}(e, MD_\sigma), \tau) = c}{\langle \mathsf{Exc}(\mathsf{switchStart}(c, \tau))\ \mathsf{case}\ e\colon T, \sigma\rangle \to \langle T, \sigma\rangle}$$

The statement labeled by the label default is executed either in a normal sequential program run or when it catches the exception Exc(defaultStart) raised by the auxiliary construction switchStop:

$$\langle \mathsf{default}\colon T, \sigma\rangle \to \langle T, \sigma\rangle$$

$$\langle \mathsf{Exc}(\mathsf{defaultStart})\ \mathsf{default}\colon T, \sigma\rangle \to \langle T, \sigma\rangle$$

The auxiliary construction switchStop(T) catches the exception Exc(switchStart(c)), raised by a switch statement with the body T in the case when none of case labels in T were matched with the control expression value, and starts executing the body T raising the exception Exc(defaultStart) first. Otherwise this instruction is ignored:

$$\langle \mathsf{Exc}(\mathsf{switchStart}(c))\ \mathsf{switchStop}(T)\ T', \sigma\rangle \to$$
$$\langle \mathsf{Exc}(\mathsf{defaultStart})\ T\ \mathsf{defaultStop}\ T', \sigma\rangle$$

$$\langle \mathsf{switchStop}(T), \sigma\rangle \to \langle \varepsilon, \sigma\rangle$$

The auxiliary construction defaultStop catches the exception Exc(defaultStart) raised by switchStop(T) in the case when T does not contain the label default. Otherwiese this instruction is ignored:

$$\langle \mathsf{Exc}(\mathsf{defaultStart})\ \mathsf{defaultStop}\ T, \sigma\rangle \to \langle T, \sigma\rangle$$

$$\langle \mathsf{defaultStop}, \sigma\rangle \to \langle \varepsilon, \sigma\rangle$$

Compound statement. When defining semantics of a block, it is necessary to take into consideration that the rules for a goto statement transfer the control only forward. When meeting goto L, we raise the exception Exc(gotoStart(L)) and skip the next statements until we meet the statement labeled by the label L which catches this exception.

But how can we restore the passed program part if control is transferred backward? The idea is the following. When transferring the control, we move always forward but at the end of each block we check if the label L belongs to this block. If yes, the goto L statement was met in the block body and the label is situated earlier in the text. In this case we jump to the beginning of the block and descend to the label L. Otherwise control is transferred to the covering block which will perform the same check.

To handle blocks of labels, th e auxiliary construction gotoStop(T) is used, where T is a sequence of block statements.

The axiom for the block has the form:

$$\langle \{T\}, \sigma\rangle \rightarrow \langle T \text{ gotoStop}(T), \sigma\rangle$$

The auxiliary construction gotoStop(T) catches the exception Exc(gotoStart(L)) raised by the statement goto L in the case when the label L is situated in the sequence of statements T. Otherwise this construction is ignored:

$$\frac{L \in \text{labels}(T)}{\begin{array}{c}\langle \text{Exc}(\text{gotoStart}(L)) \text{ gotoStop}(T)\ T', \sigma\rangle \rightarrow \\ \langle \text{Exc}(\text{gotoStart}(L))\ T \text{ gotoStop}(T)\ T', \sigma\rangle\end{array}}$$

$$\langle \text{gotoStop}(T), \sigma\rangle \rightarrow \langle \varepsilon, \sigma\rangle$$

Expression statement. Execution of an expression statement is reduced to evaluation of the corresponding expression:

$$\langle e;, \sigma\rangle \rightarrow \langle e, \sigma\rangle$$

Null statement. Null statement performs no actions:

$$\langle ;, \sigma\rangle \rightarrow \langle \varepsilon, \sigma\rangle$$

Selection statements. In a normal sequential program run, the selection statement execution is defined by the following rules:

$$\frac{\langle e, \sigma\rangle \rightarrow^{*} \langle \text{Val}(\omega), \sigma'\rangle}{\langle \text{if } (e)\ S_1 \text{ else } S_2, \sigma\rangle \rightarrow \langle \text{Val}(\omega), \sigma'\rangle}$$

$$\frac{\langle e, \sigma\rangle \to^* \langle \mathrm{Val}(v, \tau), \sigma'\rangle,\ \mathrm{cast}(\mathrm{val}(v), \tau, \mathrm{int}) \neq 0}{\langle \mathrm{if}\ (e)\ S_1\ \mathrm{else}\ S_2, \sigma\rangle \to \langle S_1, \sigma'\rangle}$$

$$\frac{\langle e, \sigma\rangle \to^* \langle \mathrm{Val}(v, \tau), \sigma'\rangle,\ \mathrm{cast}(\mathrm{val}(v), \tau, \mathrm{int}) = 0}{\langle \mathrm{if}\ (e)\ S_1\ \mathrm{else}\ S_2, \sigma\rangle \to \langle S_2, \sigma'\rangle}$$

$$\frac{\langle e, \sigma\rangle \to^* \langle \mathrm{Val}(\omega), \sigma'\rangle}{\langle \mathrm{if}\ (e)\ S, \sigma\rangle \to \langle \mathrm{Val}(\omega), \sigma'\rangle}$$

$$\frac{\langle e, \sigma\rangle \to^* \langle \mathrm{Val}(v, \tau), \sigma'\rangle,\ \mathrm{cast}(\mathrm{val}(v), \tau, \mathrm{int}) \neq 0}{\langle \mathrm{if}\ (e)\ S, \sigma\rangle \to \langle S, \sigma'\rangle}$$

$$\frac{\langle e, \sigma\rangle \to^* \langle \mathrm{Val}(v, \tau), \sigma'\rangle,\ \mathrm{cast}(\mathrm{val}(v), \tau, \mathrm{int}) = 0}{\langle \mathrm{if}\ (e)\ S, \sigma\rangle \to \langle \varepsilon, \sigma'\rangle}$$

In a normal sequential program run the statement switch (e) S evaluates the value of the expression e and raises the exception Exc(switchStart(c)) which could be caught by the statements from S labeled by the labels case and default in accordance with the rules for labeled statements. The use of this exception allows us not to search for the required branch in the switch statement and to jump to its body directly:

$$\frac{\langle e, \sigma\rangle \to^* \langle \mathrm{Val}(\omega), \sigma'\rangle}{\langle \mathrm{switch}\ (e)\ S, \sigma\rangle \to \langle \mathrm{Val}(\omega), \sigma'\rangle}$$

$$\frac{\langle e, \sigma\rangle \to^* \langle \mathrm{Val}(v, \tau), \sigma'\rangle}{\begin{array}{c}\langle \mathrm{switch}\ (e)\ S, \sigma\rangle \to \\ \langle \mathrm{Exc}(\mathrm{switchStart}((\mathrm{val}(v), \tau)))\ S\ \mathrm{switchStop}(S)\ \mathrm{gotoStop}(S)\ \mathrm{breakStop}, \sigma'\rangle\end{array}}$$

The auxiliary construction breakStop catches the exception Exc(breakStart(c)) raised by the break statement:

$$\langle \mathrm{Exc}(\mathrm{breakStart})\ \mathrm{breakStop}\ T, \sigma\rangle \to \langle T, \sigma\rangle$$

Iteration statements. In these rules we use the auxiliary construction continueStop(e, S), which catches the exception Exc(continueStart) raised by the continue statement and, besides, checks the condition e when leaving the loop body S:

$$\frac{\langle e, \sigma\rangle \to^{*} \langle \mathrm{Val}(\omega), \sigma'\rangle}{\langle \mathrm{continueStop}(e, T)\ T', \sigma\rangle \to \langle \mathrm{Val}(\omega), \sigma'\rangle}$$

$$\frac{\langle e, \sigma\rangle \to^{*} \langle \mathrm{Val}(\omega), \sigma'\rangle}{\langle \mathrm{Exc}(\mathrm{continueStart})\ \mathrm{continueStop}(e, T)\ T', \sigma\rangle \to \langle \mathrm{Val}(\omega), \sigma'\rangle}$$

$$\frac{\langle e, \sigma\rangle \to^{*} \langle \mathrm{Val}(v, \tau), \sigma'\rangle,\ \mathrm{cast}(\mathrm{val}(v), \tau, \mathrm{int}) = 0}{\langle \mathrm{continueStop}(e, T)\ T', \sigma\rangle \to \langle T', \sigma'\rangle}$$

$$\frac{\langle e, \sigma\rangle \to^{*} \langle \mathrm{Val}(v, \tau), \sigma'\rangle,\ \mathrm{cast}(\mathrm{val}(v), \tau, \mathrm{int}) = 0}{\langle \mathrm{Exc}(\mathrm{continueStart})\ \mathrm{continueStop}(e, T)\ T', \sigma\rangle \to \langle T', \sigma'\rangle}$$

$$\frac{\langle e, \sigma\rangle \to^{*} \langle \mathrm{Val}(v, \tau), \sigma'\rangle,\ \mathrm{cast}(\mathrm{val}(v), \tau, \mathrm{int}) \neq 0}{\langle \mathrm{continueStop}(e, T)\ T', \sigma\rangle \to \langle T\ \mathrm{gotoStop}(T)\ \mathrm{continueStop}(e, T)\ T', \sigma'\rangle}$$

$$\frac{\langle e, \sigma\rangle \to^{*} \langle \mathrm{Val}(v, \tau), \sigma'\rangle,\ \mathrm{cast}(\mathrm{val}(v), \tau, \mathrm{int}) \neq 0}{\begin{array}{c}\langle \mathrm{Exc}(\mathrm{continueStart})\ \mathrm{continueStop}(e, T)\ T', \sigma\rangle \to \\ \langle T\ \mathrm{gotoStop}(T)\ \mathrm{continueStop}(e, T)\ T', \sigma'\rangle\end{array}}$$

In a normal sequential program run, execution of the loop body statements is reduced to execution of the construction continueStop:

$$\langle \mathrm{while}\ (e)\ S, \sigma\rangle \to \langle \mathrm{continueStop}(e, S)\ \mathrm{breakStop}, \sigma\rangle$$

$$\langle \mathrm{do}\ S\ \mathrm{while}\ (e), \sigma\rangle \to \langle S\ \mathrm{gotoStop}(S)\ \mathrm{continueStop}(e, S)\ \mathrm{breakStop}, \sigma\rangle$$

$$\begin{array}{c}\langle \mathrm{for}\ (e_1; e_2; e_3)\ S, \sigma\rangle \to \\ \langle e_1;\ \mathrm{if}\ (!e_2)\ \mathrm{break};\ S\ \mathrm{gotoStop}(S)\ \mathrm{continueStop}((e_3, e_2), S)\ \mathrm{breakStop}, \sigma\rangle\end{array}$$

Jump statements. Jump statements raise exceptions of the form Exc(...). These exceptions are intercepted by another language construction by analogy with the exception processing mechanism:

$$\langle \text{goto L;}, \sigma\rangle \to \langle \text{Exc(gotoStart(L))}, \sigma\rangle$$

$$\langle \text{break;}, \sigma\rangle \to \langle \text{Exc(breakStart}, \sigma\rangle$$

$$\langle \text{continue;}, \sigma\rangle \to \langle \text{Exc(continueStart}, \sigma\rangle$$

$$\langle \text{return;}, \sigma\rangle \to \langle \text{Exc(returnStart)}, \sigma\rangle$$

$$\frac{\langle \text{e}, \sigma\rangle \to^{*} \langle \text{Val(v}, \tau), \sigma'\rangle}{\langle \text{return e;}, \sigma\rangle \to \langle \text{Exc(returnStart(v}, \tau)), \sigma'\rangle}$$

$$\frac{\langle \text{e}, \sigma\rangle \to^{*} \langle \text{Val}(\omega), \sigma'\rangle}{\langle \text{return e;}, \sigma\rangle \to \langle \text{Val}(\omega), \sigma'\rangle}$$

The beauty of this approach is that there is no need to search for a program point to jump to when transferring control. Such a search is difficult to formalize. It is much easier to ignore the next statements until control is in the required point or in a program position where the required point could be easily found.

Declarator sequences. Declarator chains are divided into separated declarations by the following rules:

$$\frac{\tau' \in \text{logtype}(\tau, \text{MD}_\sigma)}{\langle \text{storage } \tau\ e_1, \ldots, e_n;, \sigma\rangle \to \langle \text{storage } \tau'\ e_1; \ldots; \text{storage } \tau'\ e_n;, \sigma\rangle}$$

$$\frac{\tau' \in \text{logtype}(\tau, \text{MD}_\sigma)}{\langle \tau\ e_1, \ldots, e_n;, \sigma\rangle \to \langle \tau'\ e_1; \ldots; \tau'\ e_n;, \sigma\rangle}$$

Statement sequences. Let S be a statement or an auxiliary construction, T be a non-empty statement and an auxiliary construction sequence. The rule for a statement sequence has the form:

$$\frac{\langle S, \sigma\rangle \to \langle S', \sigma'\rangle}{\langle S\ T, \sigma\rangle \to \langle S'\ T, \sigma'\rangle}$$

Exceptional values infiltration axioms. These axioms are applied every time when no other axiom or auxiliary construction could be applied. This is not a meta-condition which sorts the application of C-light operational semantics axioms and inference rules in the whole, i. e. these axioms are only a convenient abbreviation for those axiom groups, the elements of which are defined by specialization of T, namely, by the definition of the first operator in the sequence T. Because the number of C-light statements is finite, these groups define the finite number of axioms:

$$\langle \mathsf{Exc}(e)\ E\ T, \sigma\rangle \rightarrow \langle \mathsf{Exc}(e)\ T, \sigma'\rangle$$

$$\langle \mathsf{Val}(\omega)\ E\ T, \sigma\rangle \rightarrow \langle \mathsf{Val}(\omega)\ T, \sigma'\rangle$$

$$\langle \mathsf{Val}(v, \tau)\ E\ T, \sigma\rangle \rightarrow \langle \mathsf{Val}(v, \tau)\ T, \sigma'\rangle$$

The program. The rule for the program Prgm(T) consisting of the declaration sequence T with one dedicated function main has the form:

$$\frac{\langle T, \sigma\rangle \rightarrow^{*} \langle \varepsilon, \sigma'\rangle,\ MD_{\sigma'}(\&main) = (params, S)}{\langle \mathsf{Prgm}(T), \sigma\rangle \rightarrow \langle \{S\}\ returnStop\ Cast(\tau_{main}), \sigma'\rangle}$$

where τ_{main} is the type of the value returning by the function main. In our approach the arguments of the function main are considered as global variables, the values of which are defined by the initial state σ.

3. THE MIXED AXIOMATIC SEMANTICS MHSC OF C-KERNEL LANGUAGE

An environment E is a tuple (f, τ, B, bid, lab), where f is the name of a current function which returns a value of the type τ, B is its body, bid is a unique identifier of the current block and lab is the label of a goto statement or the identifier of the function body block. In all other cases lab = ω. We assume that the function main is the current function for the whole program.

A specification $SP = (SP_{fun}, SP_{lab})$ defines the information about function preconditions and postconditions and about the invariants of labelled statements.

The function specification SP_{fun} is a pair (SP_{pre}, SP_{post}) of maps SP_{pre} and SP_{post} called the specifications of the function pre- and postconditions, respectively.

The specification of the function preconditions SP_{pre} is a map from the names of the function f to functions P_f of meta-variables. An assertion $P_f(MD)$ is called a precondition of the function f.

The specification of the function postconditions SP_{post} is a map from the names of the function f to functions Q_f of meta-variables. An assertion $Q_f(MD)$ is called a postcondition of the function f.

The specification of labels is a map from labels to assertions. An assertion $SP_{lab}(L)$ is called an invariant of the label L.

A formula Ψ of the assertion language can contain the specification of function preconditions SP_{pre} and the specification of function postconditions SP_{post}. Let $SP_{fun} \models \Psi$ denote that Ψ is true in any state, when semantics of the maps SP_{pre} and SP_{post} is fixed and $SP_{pre} = fst(SP_{fun})$, $SP_{post} = snd(SP_{fun})$.

Let P, Q, and R denote assertions, A and S stand for statement sequences, the environment $E = (f, \tau, B, cur, \omega)$, and a program block identifier is Prgm.

We define the mixed axiomatic semantics MHSC (Mixed Hoare System for C) of C-kernel language as a Hoare triples calculus [2] with an environment. We write $E, SP \vdash_{MHSC} \{P\}\ S\ \{Q\}$ if the triple {P} S {Q} is deducible in the MHSC system.

To get the unambiguity of inference, we use forward tracing [1, 3, 6]. We deduce the verification conditions by eliminating left statements.

The deducibility of the program Prgm(T) consisting of a declaration sequence T is denoted by $E \vdash_{MHSC} \{P\}\ Prgm(T)\ \{Q\}$.

Let symbols ' and " near the variable identifier stand for the introduction of a new variable.

Function call. The rule for a function denoted by an expression e_0 in the case when it returns a value has the form:

$$\frac{\begin{array}{c} E, SP \vdash \{\exists Val'\ \alpha \wedge (P'(x_1 \leftarrow cast(val(e_1, MD'), type(e_1), \tau_1), \ldots, \\ x_n \leftarrow cast(val(e_n, MD'), type(e_n), \tau_n)) \Rightarrow \\ Q'(x_1 \leftarrow cast(val(e_1, MD'), type(e_1), \tau_1), \ldots, \\ x_n \leftarrow cast(val(e_n, MD'), type(e_n), \tau_n)))\}\ A;\ \{Q\} \end{array}}{E, SP \vdash \{P\}\ \mathbf{e = e_0(e_1, \ldots, e_n);}\ A;\ \{Q\}} \qquad (3.1)$$

where SP_{fun} is the specification of the function e_0;
$P' = fst(SP_{fun}(val(e_0, MD')))(MD', Val')$ is the precondition of the function e_0;
$Q' = snd(SP_{fun}(val(e_0, MD')))(MD', Val')$ is the postcondition of the function e_0;
$x_1, \ldots, x_n$ are formal parameters of the function e_0 of the types $\tau_1, \ldots, \tau_n$ respectively;

$\alpha = \exists MD'\ P(MD \leftarrow MD')(Val \leftarrow Val') \wedge MD = upd(MD', addr(val(e, MD'), MD'), cast(fst(Val'), snd(Val'), type(e)))$ if e is a shared variable;
$\alpha = \exists e'\ P(e \leftarrow e')(Val \leftarrow Val') \wedge e = cast(fst(Val'), snd(Val'), type(e))$ if e is a non-shared variable;
$\alpha = \exists v'\ P(v \leftarrow v')(Val \leftarrow Val') \wedge v = upd(v', cast(val(i, MD), type(i), int), cast(fst(Val), snd(Val), type(v)))$ if $e = v[i]$ is a non-shared array member;
$\alpha = \exists s'\ P(s \leftarrow s')(Val \leftarrow Val') \wedge s = upd(s', t, cast(fst(Val), snd(Val), type(s)))$ if $e = s.t$ is a non-shared structure field;
MD', e', v', s' are fresh variables of corresponding types.

The inference rule for a function denoted by an expression e_0 in the case when it does not return a value has the form:

$$\frac{\begin{array}{c} E, SP \vdash \{P \wedge (P'(x_1 \leftarrow cast(val(e_1, MD), type(e_1), \tau_1), \ldots, \\ x_n \leftarrow cast(val(e_n, MD), type(e_n), \tau_n)) \Rightarrow \\ Q'(x_1 \leftarrow cast(val(e_1, MD), type(e_1), \tau_1), \ldots, \\ x_n \leftarrow cast(val(e_n, MD), type(e_n), \tau_n)))\}\ A;\ \{Q\} \end{array}}{E, SP \vdash \{P\}\ \mathbf{e_0(e_1, \ldots, e_n);}\ A;\ \{Q\}} \tag{3.2}$$

Assignment operator. Assume that an expression e_0 does not contain function calls and cast operators.

$$\frac{\begin{array}{c} E, SP \vdash \{\exists MD'\ P(MD \leftarrow MD') \wedge MD = upd(MD', addr(val(e, MD'), \\ MD'), cast(val(e_0, MD'), type(e_0), type(e)))\}\ A;\ \{Q\} \end{array}}{E, SP \vdash \{P\}\ \mathbf{e = e_0;}\ A;\ \{Q\}} \tag{3.3}$$

where e is a shared variable.

In the case when e is a non-shared variable, the rule has the form:

$$\frac{\begin{array}{c} E, SP \vdash \{\exists e'\ P(e \leftarrow e') \wedge \\ e = cast(val(e_0(e \leftarrow e'), MD), type(e_0), type(e))\}\ A;\ \{Q\} \end{array}}{E, SP \vdash \{P\}\ \mathbf{e = e_0;}\ A;\ \{Q\}} \tag{3.4}$$

If $e = v[i]$ is a non-shared array member, then

$$\frac{\begin{gathered}E, SP \vdash \{\exists v'\ P(v \leftarrow v') \wedge v = upd(v', val(i, MD),\\ cast(val(e_0(v \leftarrow v'), MD), type(e_0), type(v[i])))\}\ A;\ \{Q\}\end{gathered}}{E, SP \vdash \{P\}\ \mathbf{v[i] = e_0;}\ A;\ \{Q\}} \qquad (3.5)$$

If $e = s.t$ is a non-shared structure field, then

$$\frac{\begin{gathered}E, SP \vdash \{\exists s'\ P(s \leftarrow s') \wedge s = upd(s', t, cast(val(e_0(s \leftarrow s'), MD),\\ type(e_0), type(s.t)))\}\ A;\ \{Q\}\end{gathered}}{E, SP \vdash \{P\}\ \mathbf{s.t = e_0;}\ A;\ \{Q\}} \qquad (3.6)$$

Variable declarations. The inference rule for the declaration of a shared variable without an initializer has the form:

$$\frac{\begin{gathered}E, SP \vdash \{\exists MD'\ \exists nc\ \exists \tau\ \exists V\ \exists MD''\ P(MD \leftarrow MD') \wedge\\ \tau = logtype(tp(e), MD') \wedge MD'(nc) = \omega \wedge\\ (MD'', V) = init(\tau, storage(e), MD')) \wedge\\ MD = upd(MD'', nc, V)\}\ A\ \{Q\}\end{gathered}}{E, SP \vdash \{P\}\ \mathbf{e;}\ A;\ \{Q\}} \qquad (3.7)$$

For a non-shared variable e, the rule has the form:

$$\frac{E, SP \vdash \{\exists e'\ P(e \leftarrow e') \wedge e = defaultValue(\tau, storage)\}\ A\ \{Q\}}{E, SP \vdash \{P\}\ \mathbf{storage}\ \tau\ \mathbf{e;}\ A;\ \{Q\}} \qquad (3.8)$$

If a non-shared array v is defined in a declaration, then

$$\frac{E, SP \vdash \{\exists v'\ P(v \leftarrow v') \wedge v = ((dV, \ldots, dV), \ldots (dV, \ldots, dV))\}\ A\ \{Q\}}{E, SP \vdash \{P\}\ \mathbf{storage}\ \tau\mathbf{[n_1, \ldots, n_k]\ v;}\ A;\ \{Q\}} \qquad (3.9)$$

where $dV = defaultValue(\tau, storage)$.

In the case when a non-shared structure s is defined, the rule has the form:

$$\frac{\begin{gathered}E, SP \vdash \{\exists s'\ P(s \leftarrow s') \wedge s = (defaultValue(\tau_1, storage), \ldots,\\ defaultValue(\tau_n, storage))\}\ A\ \{Q\}\end{gathered}}{E, SP \vdash \{P\}\ \mathbf{storage\ struct\ s\ \{\tau_1\ t_1;\ \ldots;\ \tau_n\ t_n;\};}\ A;\ \{Q\}} \qquad (3.10)$$

The inference rule for a shared variable with an initializer has the form:

$$\frac{\begin{array}{c}E, SP \vdash \{\exists MD' \; \exists nc \; \exists \tau \; \exists V \; \exists MD'' \; P(MD \leftarrow MD') \wedge \\ \tau = logtype(tp(e), MD') \wedge MD'(nc) = \omega \wedge \\ (MD'', V) = init(\tau, e_0, MD')) \wedge \\ MD = upd(MD'', nc, V)\} \; A \; \{Q\}\end{array}}{E, SP \vdash \{P\} \; \mathbf{e = e_0;} \; A; \{Q\}} \tag{3.11}$$

For a non-shared variable e, the rule has the form:

$$\frac{E, SP \vdash \{\exists e' \; P(e \leftarrow e') \wedge e = e_0(e \leftarrow e')\} \; A \; \{Q\}}{E, SP \vdash \{P\} \; \mathbf{storage} \; \tau \; \mathbf{e = e_0;} \; A; \{Q\}} \tag{3.12}$$

If a non-shared array v is defined in a declaration, then

$$\frac{\begin{array}{c}E, SP \vdash \{\exists v' \; P(v \leftarrow v') \wedge \\ v = ((v_{0\ldots0}(v \leftarrow v'), \ldots, v_{0\ldots0 \; n1-1}(v \leftarrow v')), \ldots, \\ (v_{nk-1\ldots nk-1 \; 0}(v \leftarrow v'), \ldots, v_{nk-1\ldots nk-1}(v \leftarrow v')))\} \; A \; \{Q\}\end{array}}{\begin{array}{c}E, SP \vdash \{P\} \; \mathbf{storage} \; \tau\mathbf{[n_1, \ldots, n_k] \; v = \{\{v_{0\ldots0}, \ldots, v_{0\ldots0 \; n1-1}\},} \\ \mathbf{\ldots, \{v_{nk-1\ldots nk-1 \; 0}, \ldots, v_{nk-1\ldots nk-1}\}\};} \; A; \{Q\}\end{array}} \tag{3.13}$$

In the case when a non-shared structure s is defined, the rule has the form:

$$\frac{E, SP \vdash \{\exists s' \; P(s \leftarrow s') \wedge s = (v_1(s \leftarrow s'), \ldots, v_n(s \leftarrow s'))\} \; A \; \{Q\}}{\begin{array}{c}E, SP \vdash \{P\} \; \mathbf{storage \; struct \; s \; \{\tau_1 \; t_1 = v_1; \ldots; \tau_n \; t_n = v_n;\};} \\ A; \{Q\}\end{array}} \tag{3.14}$$

Type declaration.

$$\frac{E, SP \vdash \{\exists \tau \; P \wedge \tau = logtype(tp(e), MD)\} \; A; \{Q\}}{E, SP \vdash \{P\} \; \mathbf{typedef \; e;} \; A; \{Q\}} \tag{3.15}$$

Function declaration.

$$\frac{\begin{array}{c}E, SP \vdash \{\exists MD'\ \exists\tau'\ \exists nc\ P(MD \leftarrow MD')\ \wedge \\ \tau' = logtype(\tau\ f(\tau_1\ x_1, \ldots, \tau_n\ x_n), MD')\ \wedge \\ MD'(nc) = \omega \wedge MD = upd(MD', nc, (f, (x_1, \ldots, x_n), S))\}\ A;\ \{Q\}\end{array}}{E, SP \vdash \{P\}\ \tau\ \mathbf{f(\tau_1\ x_1, \ldots, \tau_n\ x_n)\ \{S\}}\ A;\ \{Q\}} \quad (3.16)$$

Labelled statement.

$$\frac{\begin{array}{c}(f, \tau, B, cur, L), SP \vdash \{P\}\ A;\ \{Q\} \\ (f, \tau, B, cur, \omega), SP \vdash \{SP_{lab}(L_1)\}\ S;\ A;\ \{Q\}\end{array}}{(f, \tau, B, cur, L), SP \vdash \{P\}\ \{SP_{lab}(L_1)\}\ \mathbf{L_1\!: S;}\ A;\ \{Q\}}\ L \neq L_1 \quad (3.17)$$

$$\frac{\begin{array}{c}SP_{fun} \models P \Rightarrow SP_{lab}(L) \\ (f, \tau, B, cur, \omega), SP \vdash \{SP_{lab}(L)\}\ S;\ A;\ \{Q\}\end{array}}{(f, \tau, B, cur, E_5), SP \vdash \{P\}\ \{SP_{lab}(L)\}\ \mathbf{L\!: S;}\ A;\ \{Q\}}\ \begin{array}{c}E_5 = \omega \\ \text{or} \\ E_5 = L\end{array} \quad (3.18)$$

Compound statement.

$$\frac{(f, \tau, B, id, E_5), SP \vdash \{P\}\ S;\ blockEnd(cur);\ A;\ \{Q\}}{(f, \tau, B, cur, E_5), SP \vdash \{P\}\ \mathbf{\{S\}_{id}}\ A;\ \{Q\}} \quad (3.19)$$

Let $x_1, \ldots, x_n$ be shared variables and $y_1, \ldots, y_k$ be non-shared variables declared in the block id.

$$\frac{\begin{array}{c}(f, \tau, B, cur, E_5), SP \vdash \{\exists MD'\ \exists y_1' \ldots \exists y_k' \\ P(MD \leftarrow MD')(y_1 \leftarrow y_1') \ldots (y_k \leftarrow y_k')\ \wedge \\ MD = upd(MD', \{mem(x_1), \ldots, mem(x_n)\}, \omega)\ \wedge \\ y_1 = \omega \wedge \ldots y_k = \omega\}\ A;\ \{Q\}\end{array}}{(f, \tau, B, id, E_5), SP \vdash \{P\}\ \mathbf{blockEnd(cur);}\ A;\ \{Q\}}\ id \neq E_5 \quad (3.20)$$

$$\frac{\begin{array}{c}SP_{fun} \vDash (\exists MD'\ \exists y_1'\ \ldots\ \exists y_k'\ P(MD \leftarrow MD')(y_1 \leftarrow y_1')\ \ldots\ (y_k \leftarrow y_k') \wedge \\ MD = upd(MD', \{mem(x_1), \ldots, mem(x_n)\}, \omega) \wedge \\ y_1 = \omega \wedge \ldots\ y_k = \omega) \Rightarrow SP_{post}(f)\end{array}}{(f, \tau, B, id(f), id(f)), SP \vdash \{P\}\ \mathbf{blockEnd(cur);}\ A; \{Q\}} \qquad (3.21)$$

Null statement.

$$\frac{E, SP \vdash \{P\}\ A; \{Q\}}{E, SP \vdash \{P\}\ \mathbf{;}\ A; \{Q\}} \qquad (3.22)$$

$$\frac{SP_{fun} \vDash P \Rightarrow Q}{(f, \tau, B, cur, \omega), SP \vdash \{P\}\ \mathbf{;}\ \{Q\}} \qquad (3.23)$$

The if statement.

$$\frac{\begin{array}{c}E, SP \vdash \{P \wedge logval(e)\}\ S_1; A; \{Q\} \\ E, SP \vdash \{P \wedge \neg logval(e)\}\ S_2; A; \{Q\}\end{array}}{E, SP \vdash \{P\}\ \mathbf{if\ (e)\ S_1\ else\ S_2;}\ A; \{Q\}} \qquad (3.24)$$

The while statement.

$$\frac{\begin{array}{c}SP_{fun} \vDash P \Rightarrow INV \\ E, SP \vdash \{INV \wedge logval(e)\}\ S; \{INV\} \\ E, SP \vdash \{INV \wedge \neg logval(e)\}\ A\ \{Q\}\end{array}}{E, SP \vdash \{P\}\ \{INV\}\ \mathbf{while\ (e)\ S;}\ A; \{Q\}} \qquad (3.25)$$

Jump statements.

$$\frac{(f, \tau, B, cur, L), SP \vdash \{P\}\ A; \{Q\}}{(f, \tau, B, cur, \omega), SP \vdash \{P\}\ \mathbf{goto\ L;}\ A; \{Q\}} \qquad (3.26)$$

$$\frac{(f, \tau, B, cur, L), SP \vdash \{P\}\ A; \{Q\}}{(f, \tau, B, cur, L), SP \vdash \{P\}\ \mathbf{T;}\ A; \{Q\}} \qquad (3.27)$$

where T is not a labelled statement and is not blockEnd(cur).

$$\frac{(\mathsf{f}, \tau, \mathsf{B}, \mathsf{cur}, \mathsf{id(f)}), \mathsf{SP} \vdash \{\mathsf{P}\}\ \mathsf{A}\ \{\mathsf{Q}\}}{(\mathsf{f}, \tau, \mathsf{B}, \mathsf{cur}, \omega), \mathsf{SP} \vdash \{\mathsf{P}\}\ \textbf{return;}\ \mathsf{A}\ \{\mathsf{Q}\}} \qquad (3.28)$$

$$\frac{(\mathsf{f}, \tau, \mathsf{B}, \mathsf{cur}, \mathsf{id(f)}), \mathsf{SP} \vdash \{\mathsf{P}\}\ \mathsf{A}\ \{\mathsf{Q}\}}{(\mathsf{f}, \tau, \mathsf{B}, \mathsf{cur}, \mathsf{id(f)}), \mathsf{SP} \vdash \{\mathsf{P}\}\ \textbf{T;}\ \mathsf{A}\ \{\mathsf{Q}\}} \qquad (3.29)$$

$$\frac{\begin{array}{c}(\mathsf{f}, \tau, \mathsf{B}, \mathsf{cur}, \mathsf{id(f)}), \mathsf{SP} \vdash \{\exists \mathsf{Val'}\ \mathsf{P}(\mathsf{Val} \leftarrow \mathsf{Val'}) \wedge \\ \mathsf{Val} = (\mathsf{cast}(\mathsf{val}(\mathsf{e}, \mathsf{MD}), \mathsf{type}(\mathsf{e}), \tau), \tau)\}\ \mathsf{A}\ \{\mathsf{Q}\}\end{array}}{(\mathsf{f}, \tau, \mathsf{B}, \mathsf{cur}, \omega), \mathsf{SP} \vdash \{\mathsf{P}\}\ \textbf{return e;}\ \mathsf{A}\ \{\mathsf{Q}\}} \qquad (3.30)$$

where T is not a labelled statement and is not blockEnd(...).

Consequence rule.

$$\frac{\mathsf{SP}_{\mathsf{fun}} \models \mathsf{P} \Rightarrow \mathsf{R} \qquad \mathsf{E}, \mathsf{SP} \vdash \{\mathsf{R}\}\ \mathsf{S}\ \{\mathsf{T}\} \qquad \mathsf{SP}_{\mathsf{fun}} \models \mathsf{T} \Rightarrow \mathsf{Q}}{\mathsf{E}, \mathsf{SP} \vdash \{\mathsf{P}\}\ \mathsf{S}\ \{\mathsf{Q}\}} \qquad (3.31)$$

Statement sequence. Let T and T' be sequences of non-empty statements and auxiliary constructions.

$$\frac{\mathsf{E}, \mathsf{SP} \vdash \{\mathsf{P}\}\ \mathsf{T}\ \{\mathsf{R}\} \qquad \mathsf{E}, \mathsf{SP} \vdash \{\mathsf{R}\}\ \mathsf{T'}\ \{\mathsf{Q}\}}{\mathsf{E}, \mathsf{SP} \vdash \{\mathsf{P}\}\ \textbf{T T'}\ \{\mathsf{Q}\}} \qquad (3.32)$$

The program. The rule for the program Prgm(T) consisting of a declaration sequence T has the form:

$$\frac{\begin{array}{c}(f,\ \tau_f,\ S_f,\ bid_f,\ \omega),\ ((SP_{pre},\ SP_{post}),\ SP^f{}_{lab}) \vdash \\ \{SP_{pre}(f)(x_1,\ \dots,\ x_n)(MD,\ Val)\}\ S_f;\ blockEnd(bid_f) \\ \{SP_{post}(f)(x_1,\ \dots,\ x_n)(MD,\ Val)\} \\ \text{through all function names } f \text{ defined in } T \text{ except for } main, \\ (main,\ \tau_{main},\ S_{main},\ bid_{main},\ \omega),\ SP \vdash \\ \{P\}\ T\ S_{main};\ blockEnd(bid_{main})\ \{Q\}\end{array}}{E,\ SP \vdash \{P\}\ \mathbf{Prgm(T)}\ \{Q\}} \qquad (3.33)$$

where x_1, ..., x_n are formal parameters of the function f.

4. CONSISTENCY OF MIXED AXIOMATIC SEMANTICS

One of the main properties of mixed axiomatic semantics is its consistency as an inference system with respect to mixed operational semantics i. e. we can get valid formulae from valid formulae during inference.

Let us define the notion of Hoare triple truth. For the program prg, the truth of the Hoare triple {P} prg {Q} is defined as follows: {P} prg {Q} is true in the sense of partial correctness (designation $\models$ {P} prg {Q}) if $\mathcal{M}[[prg]](||P||) \subseteq ||Q||$ (i. e. the statement Q is true in all final states of all possible program prg executions started in those states, where P is true).

To prove the consistency of the inference system MHSC, we should generalize the notion of partial correctness semantics for program fragments which differ from the whole program. In this case it is necessary to take into consideration the global context of program fragment execution that connects this fragment with the whole program. Information about the program context is defined by the environment E and program specification SP.

Semantics of partial correctness with respect to the specification SP is defined for an arbitrary C-light construction S as a map $\mathcal{M}_{SP}[[S]]$ from States to 2^{States}:

$$\mathcal{M}_{SP}[[S]](\sigma) = \{\sigma' \mid \langle S\ gotoStop(S),\ \sigma\rangle \to^* \langle \varepsilon,\ \sigma'\rangle\} \cup$$
$$\{\sigma'(Val \leftarrow (v,\ \tau)) \mid \langle S\ gotoStop(S),\ \sigma\rangle \to^* \langle Val(v,\ \tau),\ \sigma'\rangle\} \cup$$
$$\{\sigma' \mid \langle gotoStart(L)\ S\ gotoStop(S),\ \sigma''\rangle \to^* \langle \varepsilon,\ \sigma'\rangle$$
$$\text{for a certain label } L \text{ contained in } S \text{ and a state } \sigma'' \text{ such that } \sigma'' \models SP_{lab}\}$$

This means that the semantics considers all possible executions of the program construction S such that this construction finishes in a normal way under the condition that the construction execution starts either in the state σ or in the state when a jump to this construction by the statement goto L occurs. In the last case the state must satisfy the invariant $SP_{lab}(L)$ of the label L. Addition of the construction gotoStop(S) allows us to take into account the loops made with the help of goto statement.

To prove that the specification $SP_{\mathbf{lab}}(L)$ is really a label invariant, we need one more sort of semantics. The semantics of exit in the label L with respect to the specification SP is defined for an arbitrary C-light construction S as a map $\mathcal{M}_{SP}{}^{L}[[S]]$ from States to 2^{States}:

$$\mathcal{M}_{SP}{}^{L}[[S]](\sigma) = \{\sigma' \mid \langle S\ gotoStop(S), \sigma\rangle \to^{*} \langle gotoStart(L), \sigma'\rangle$$
$$\text{for a certain label } L\} \cup$$
$$\{\sigma' \mid \langle gotoStart(L')\ S\ gotoStop(S), \sigma''\rangle \to^{*} \langle gotoStart(L), \sigma'\rangle$$
$$\text{for a certain label } L' \text{ and the state } \sigma'' \text{ such that } L' \text{ is contained in } S$$
$$\text{and } \sigma'' \models SP_{lab}\}$$

This means that the semantics considers all possible executions of the program construction S in which this construction finishes with a jump to the label L under the condition that construction execution starts either in the state σ or in the state when a jump to this construction by the statement goto L occurs. In the last case the state must satisfy the invariant $SP_{lab}(L)$ of the label L. Addition of the construction gotoStop(S) allows us to take into account the loops made with the help of goto statements.

Also we define

$$\mathcal{M}_{SP}[\![S]\!](||P||) = \bigcup_{\sigma \in ||P||} \mathcal{M}_{SP}[\![S]\!](\sigma)$$

and

$$\mathcal{M}_{SP}^{L}[\![S]\!](||P||) = \bigcup_{\sigma \in ||P||} \mathcal{M}_{SP}^{L}[\![S]\!](\sigma)$$

where P is a formula.

Let us consider an environment E of the form $(f, \tau_f, \{S\}, bid, lab)$. Let Dom(E) stand for the formula

$$\begin{cases} \mathsf{retType(type(\&f))} = \tau_f \wedge \mathsf{snd(MD(\&f))} = \mathsf{S} \text{ if } \mathsf{f} \neq \mathsf{main} \\ \mathsf{true} \text{ otherwise} \end{cases}$$

The Hoare triple {P} S {Q} is true in the sense of partial correctness with respect to the environment E and the specification SP (the designation is E, SP $\models$ {P} S {Q}) if

- $\models$ P $\Rightarrow$ Dom(E);
- $\mathcal{M}$[[S]](||P||) $\subseteq$ ||Q||.

Lemma 1 (about nesting blocks). For any state σ and any statement sequence T, we have

1. $\mathcal{M}_{SP}$[[{{T}}]](σ) = $\mathcal{M}_{SP}$[[{T}]](σ).
2. $\mathcal{M}_{SP}$[[{T}]](σ) = $\mathcal{M}_{SP}$[[T]](σ) if T does not contain declarations at the external level.

The proof is given in [10].

Lemma 2 (about a normalized expression). For any normalized expression e and for any state σ, we have the following:

$$\langle e, \sigma\rangle \rightarrow^{*} \langle Val(val(e, MD_{\sigma}), type(e)), \sigma \rangle$$

The proof is given in [10].

Theorem 1. The inference system MHSC is consistent for the partial correctness property, i. e. E $\vdash_{\mathbf{MHSC}}$ {P} prg {Q} implies E $\models$ {P} prg {Q}.

The proof.

Let us introduce several auxiliary notions. The size size(S) of the program fragment S is defined as follows

- size(Prgm(S)) = 2 * size(S) + 2;
- size(D) = size(S) + 1 if D is a function declaration with the body S;
- size(L: S) = size(S) + 1;

- size({S}) = size(S) + 2;
- size(if (e) S else S') = max(size(S), size(S')) + 1;
- size(while (e) S) = size(S) + 1;
- size(S S') = size(S) + size(S');
- size(S) = 2 otherwise.

Let us expand the function size on Hoare triples and assertion language statements:

- size(φ) = 0 if φ is an assertion language formula;
- size({P} S {Q}) = size(S) otherwise.

Let us define the partial order relation $\prec$ on Hoare triples and assertion language statements as follows: $k \prec m$ if

$$\text{size}(k) < \text{size}(m).$$

Let m and k stand for a Hoare triple and an assertion language formula, A({P} S {Q}) stand for the statement that if the Hoare triple or the formula {P} S {Q} is deducible in MSHC, then the following properties hold:

1. E, SP ⊨ {P} S {Q}.
2. $\mathcal{M}_{\mathbf{SP}}^{\mathbf{L}}[[S]](\|P\|) \subseteq \|SP_{\mathbf{lab}}(L)\|$ for each label contained in the body S_f of the function named f.

The truth of the formula ∀m A(m) implies the truth of the theorem. Let us notice that the premises of all MHSC rules are less with respect to the relation $\prec$ than its conclusions except for the consequence rule. That's why the proof of this formula is divided into three stages using induction on $\prec$. At the first stage we prove that the consequence rule holds the truth, i. e. the truth of premises implies the truth of the corollary. At the second stage the induction base is proved. At the third stage the inductive transition is proved.

Consequence rule.

Let us prove that if $\sigma \models P$ and $\langle S, \sigma\rangle \rightarrow^{*} \langle \varepsilon, \sigma'\rangle$ then $\sigma' \models Q$.

According to the premise $SP_{fun} \models P \Rightarrow R$ of the rule (3.31) $\sigma \models R$.

According to the premise $E, SP \vdash \{R\}\ S\ \{T\}$ of the rules (3.31) $\sigma' \models T$.

According to the premise $SP_{fun} \models T \Rightarrow Q$ of the rule (3.31) $\sigma' \models Q$.

Induction base.

The induction base has the form $\forall\varphi\ A(\varphi)$, where φ is an assertion language formula. Its truth follows from the fact that the set of the derivable MHSC formulae is equal to the set of the true assertion language formulae by the definition of MHSC.

Inductive transition.

The inductive transition has the form $\forall m\ (\forall k\ k \prec m \Rightarrow A(k)) \Rightarrow A(m)$. The proof of the inductive transition is reduced to investigation of cases of different program fragments.

The assignment statement.

Let e be a shared variable, S have the form $e = e';\ A$ and the Hoare triple $m = \{P\}\ S\ \{Q\}$ be derivable.

Let $P' = \exists MD'\ P(MD \leftarrow MD') \wedge MD = upd(MD', addr(val(e, MD'), MD'), cast(val(e_0, MD'), type(e_0), type(e)))$. According to the rule (3.3) the triple $k = \{P'\}\ A\ \{Q\}$ is also derivable.

Let $\sigma \models P$ and $\langle e_0, \sigma\rangle \rightarrow^* \langle Val(v', \tau'), \sigma'\rangle$, $\langle e, \sigma'\rangle \rightarrow^* \langle Val(v'', \tau''), \sigma''\rangle$, $v = cast(v', \tau', \tau'')$, $c = addr(v'')$.

Then, according to the rule for an assignment statement, $\langle e = e_0, \sigma\rangle \rightarrow \langle Val(v, \tau''), \sigma''(MD \leftarrow upd(MD, c, v))\rangle$. Then $\sigma'' \models P'$.

As $k \prec m$, the triple k is true by the inductive hypothesis. It means that if $\langle A, \sigma''\rangle \rightarrow^* \langle \varepsilon, \sigma'''\rangle$ then $\sigma''' \models Q$.

The case when e is a non-shared variable is considered similarly.

Empty statement.

Let A be a non-empty statement sequence and the Hoare triple $m = \{P\}\ ;\ A\ \{Q\}$ be derivable.

According to the rule (3.22), the triple $k = \{P\}\ A\ \{Q\}$ is also derivable.

As $k \prec m$, the triple k is true by the inductive hypothesis. It means that if $\sigma \models P$ and $\langle A, \sigma\rangle \to^* \langle \varepsilon, \sigma'\rangle$, then $\sigma' \models Q$.

According to the rule for an empty statement, $\langle ;\ , \sigma\rangle \to \langle \varepsilon, \sigma\rangle$, then $\langle ;\ A, \sigma\rangle \to^* \langle A, \sigma\rangle$. Therefore, $\langle ;\ A, \sigma\rangle \to^* \langle \varepsilon, \sigma'\rangle$ and m is true.

Let the triple $m = \{P\}\ ;\ \{Q\}$ be derivable.

According to the rule (3.23), the formula $P \Rightarrow Q$ is also derivable.

As $(P \Rightarrow Q) \prec m$ the formula $P \Rightarrow Q$ is true by the inductive hypothesis. Therefore, if $\sigma \models P$, then $\sigma \models Q$ and, according to the rule for an empty statement, $\langle ;\ , \sigma\rangle \to \langle \varepsilon, \sigma\rangle$.

The if statement.

Let the Hoare triple $m = \{P\}$ if (e) S_1 else S_2; $A\ \{Q\}$ be derivable. According to the rule (3.24), the Hoare triples $k_1 = \{P \wedge \text{logval}(e)\}\ S_1;\ A\ \{Q\}$ and $k_2 = \{P \wedge \neg\text{logval}(e)\}\ S_2;\ A\ \{Q\}$ are also derivable.

As $k_1 \prec m$ and $k_2 \prec m$, the triples k_1 and k_2 are true by the inductive hypothesis.

Let $\sigma \models P \wedge \text{logval}(e)$, then $\sigma \models P$.

Let $\langle e, \sigma\rangle \to^* \langle \text{Val}(v, \tau), \sigma'\rangle$, logval(e) be true and $\langle S_1;\ A, \sigma'\rangle \to^* \langle \varepsilon, \sigma''\rangle$. Then $\sigma'' \models Q$.

According to the rule for if statement, $\langle \text{if } (e)\ S_1 \text{ else } S_2, \sigma\rangle \to \langle S_1, \sigma'\rangle$. Then $\langle \text{if } (e)\ S_1 \text{ else } S_2;\ A, \sigma\rangle \to \langle S_1;\ A, \sigma'\rangle$, therefore $\langle \text{if } (e)\ S_1 \text{ else } S_2;\ A, \sigma\rangle \to^* \langle \varepsilon, \sigma''\rangle$.

So m is true. The case of else-branch is considered similarly.

Statement sequence.

Let the triple $m = \{P\}\ T\ T'\ \{Q\}$ be derivable. According to the rule (3.32), the triples $k_1 = \{P\}\ T\ \{R\}$ and $k_2 = \{R\}\ T'\ \{Q\}$ are also derivable. As $k_1 \prec m$ and $k_2 \prec m$, therefore k_1 and k_2 are true by the inductive hypothesis.

Let us prove that if $\sigma \models P$ and $\langle S, \sigma\rangle \to^* \langle \varepsilon, \sigma'\rangle$, then $\sigma' \models Q$.

With respect to the rules of operational semantics, the proof of this statement is equivalent to the proof of the following statement: $\sigma_1 \models P$ and

$\langle E_i\ S, \sigma_i\rangle \to^* \langle E_i', \sigma_i'\rangle$
$\langle E_i'\ S', \sigma_i'\rangle \to^* \langle E_i'', \sigma_i''\rangle$

$$\langle E_i''\ \mathsf{gotoStop}(S\ S'), \sigma_i''\rangle \to \begin{cases} \text{either } \langle E_{i+1}\ S\ S'\ \mathsf{gotoStop}(S\ S'), \sigma_{i+1}\rangle, \\ \text{either } \langle E_{i+1}, \sigma_{i+1}\rangle \end{cases}$$

where

- $1 \le i \le k$;
- the constructions E_i, E_i' and E_i'' have the form ε, $\mathsf{gotoStart}(L)$, $\mathsf{returnStart}$ or $\mathsf{returnStart}(e)$;
- if $2 \le i \le k$, then $E_i \equiv \mathsf{gotoStart}(L)$ for a certain $L \in \mathsf{labels}(S\ S')$;
- if $E_{k+1} \ne \mathsf{gotoStart}(L)$ for any $L \in \mathsf{labels}(S\ S')$, then $\sigma_{k+1} \vDash Q$.

From the truth of the triple k_1 we have that if $E_i' \equiv \varepsilon$, then $\sigma_i \vDash R$.

According to the rule for $\mathsf{gotoStop}$, if $E_{i+1}' \equiv \varepsilon$, then $E_k'' \equiv \varepsilon$ and $\sigma_k'' \equiv \sigma_{k+1}$.

If $E_k' \ne \varepsilon$, then the truth of $\sigma_k'' \vDash Q$ follows from the truth of the triple k_2.

If $E_k' = \varepsilon$, then $\sigma_i'' \vDash R$ and from the truth of the triple k_2 we have $\sigma_k'' \vDash Q$ again.

Then $\sigma_{k+1} \vDash Q$.

The property $\mathcal{M}_{SP}^L[[S]](||P||) \subseteq ||SP_{lab}(L)||$ is proved similarly.

Labeled statement.

Let S have the form $L\colon T;\ A$ and the Hoare triple $m = \{SP_{lab}(L)\}\ S\ \{Q\}$ be derivable. According to the rule (3.18) the triple $k = \{SP_{\mathbf{lab}}(L)\}\ T;\ A\ \{Q\}$ and the formula $P \Rightarrow SP_{lab}(L)$ are also derivable. As $k \prec m$ and $(P \Rightarrow SP_{lab}(L)) \prec m$, the triple k and the formula $P \Rightarrow SP_{lab}(L)$ are true by the inductive hypothesis.

Let $\sigma \vDash P$, then $\sigma \vDash SP_{lab}(L)$. Let $\langle T;\ A, \sigma\rangle \to^* \langle\varepsilon, \sigma'\rangle$. As k is true, then $\sigma' \vDash Q$.

According to the rule for the labeled statement: $\langle L\colon T, \sigma\rangle \to \langle T, \sigma\rangle$. According to the rule for statement sequence: $\langle L\colon T;\ A, \sigma\rangle \to \langle T;\ A, \sigma\rangle$.

So, $\langle S, \sigma\rangle \to^* \langle\varepsilon, \sigma'\rangle$. It means that m is true.

Jump statements.

Let S have the form goto L; A and the Hoare triple m = {P} S {Q} be derivable. According to the rule for goto statement, the triple k = {P} A {Q} is also derivable. As $k \prec m$, the triple k is true by the inductive hypothesis.

Let $\sigma \models P$ and $\langle A, \sigma\rangle \to^* \langle \varepsilon, \sigma'\rangle$. As k is true, then $\sigma' \models Q$.

According to the rule for goto statement: $\langle$goto L;, $\sigma\rangle \to \langle$Exc(gotoStart(L)), $\sigma\rangle$. According to the rule for statement sequence: $\langle$goto L; A, $\sigma\rangle \to \langle$Exc(gotoStart(L)); A, $\sigma\rangle$.

If there is a labeled statement L: T inside the statement sequence A, then $\langle$Exc(gotoStart(L)); A, $\sigma\rangle \to^* \langle$Exc(gotoStart(L)); L: T, $\sigma\rangle \to \langle$T, $\sigma\rangle \to^* \langle\varepsilon, \sigma'\rangle$. So $\langle$goto L; A, $\sigma\rangle \to^* \langle\varepsilon, \sigma'\rangle$.

If there is no labeled statement L: T inside the statement sequence A, then $\langle$Exc(gotoStart(L)); A, $\sigma\rangle \to^* \langle$Exc(gotoStart(L)) gotoStop(T) T', $\sigma\rangle \to \langle$ Exc(gotoStart(L)) T gotoStop(T) T', $\sigma\rangle \to^* \langle\varepsilon, \sigma'\rangle$.

Iteration statements.

Let S have the form while(e) T; A and the Hoare triple m = {P} S {Q} be derivable. Let α = logval(e).

As we can apply only the rule (3.25) to this triple, the triple k_1 = {INV $\wedge$ α} T; {INV} and the triple k_2 = {INV $\wedge$ $\neg\alpha$} A; {Q} are also derivable.

As $k_1 \prec m$ and $k_2 \prec m$, the triples k_1 and k_2 are true by the inductive hypothesis.

It is easy to prove that

$$\mathcal{M}[[\text{while } (\alpha)\ A]] = \cup_{k=0}^{\infty} \mathcal{M}[[\text{while } (\alpha)\ A)^k]],$$

where

- while (α) A$)^0$ = while(1), i. e. we have infinite iteration;
- while (α) A$)^{k+1}$ = if (α) {A; while(α) A^k)}.

Suppose that for a certain proposition P the following property holds:

$$\mathcal{M}_{SP}[[T]](\|P \wedge \alpha\|) \subseteq \|P\|.$$

We will prove by induction on $i \geq 0$ that

$$\mathcal{M}_{SP}[[(\text{while } (a)\ T)^i]](\|P\|) \subseteq \|P \wedge \neg a\|.$$

For $i = 0$ it is obvious. Suppose that this relation holds for a certain $i > 0$. Then

$\mathcal{M}_{SP}[[(\text{while } (a)\ T)^{i+1}]](\|P\|) = \mathcal{M}_{SP}[[\text{if } (a)\ \{T;\ (\text{while } (a)\ T)^i\}]](\|P\|) =$
$\mathcal{M}_{SP}[[\{T;\ (\text{while } (a)\ T)^i\}]](\|P \wedge a\|) \cup \mathcal{M}_{SP}[[;]](\|P \wedge \neg a\|) =$
$\mathcal{M}_{SP}[[(\text{while } (a)\ T)^i]](\mathcal{M}[[T]](\|P \wedge a\|)) \cup \|P \wedge \neg a\| \subseteq$
$\mathcal{M}_{SP}[[(\text{while } (a)\ T)^i]](\|P\|) \cup \|P \wedge \neg a\| \subseteq \|P \wedge \neg a\|.$

Thus

$$\cup_{k=0}^{\infty} \mathcal{M}_{SP}[[(\text{while } (a)\ T)^i]](\|P\|) \subseteq \|P \wedge \neg a\|.$$

Then

$$\mathcal{M}_{SP}[[\text{while } (a)\ T]](\|P\|) \subseteq \|P \wedge \neg a\|.$$

Making a substitution of P by INV, we will get the truth of the triple $\{INV\}$ $\text{while } (a)\ T\ \{INV \wedge \neg a\}$ from the truth of the triple k_1.

But the triple k_2 is also true. Applying the rule for a statement sequence, we get the truth of the triple m.

The satisfiability of the property $\mathcal{M}_{SP}[[S]](\|P\|) \subseteq \|SP_{lab}(L)\|$ for the triple m follows from its satisfiability for the triple k, as S does not contain goto statement at the top level.

Program.

Let S have the form $Prgm(T)$ and the Hoare triple $m = \{P\}\ S\ \{Q\}$ be derivable. As only the rule for a program is applicable to this triple, then the triples k_f of the form

$$\{SP_{pre}(f)\}\ S_f\ \{SP_{post}(f)\}$$

where f runs through all function names defined in T except for the main with bodies S_f and returning values types τ_f, respectively, and the triple k of the form

$$\{P\}\ T\ \{S_{main}\}\ \{Q\}$$

are also derivable. As $k_f \prec m$ and $k \prec m$, the triples k_f and k are true by the inductive hypothesis.

Let us prove that if $\sigma \models P$ and $\langle S, \sigma\rangle \to^* \langle \varepsilon, \sigma'\rangle$, then $\sigma' \models Q$.

According to the rule for the program we have:

$$\langle T, \sigma\rangle \to^* \langle \varepsilon, \sigma_1\rangle$$
$$\langle \{S_{main}\}, \sigma_1\rangle \to^* \langle \varepsilon, \sigma'\rangle.$$

From the truth of the triple k and the consequence rule, we get $\sigma' \models Q$.

SP_{pre} and SP_{post} are pre- and postconditions of f because the triples k_f are true. The way to prove is induction on the depth of mutual-recursive function calls defined in the program $Prgm(T)$.

5. CONCLUSION

This work represents the mixed axiomatic semantics of C-kernel. This semantics is an essential part of the two-level C-light program verification method. It has the unambiguity of inference and special variants of inference rules for the same program statement which are applied depending on the variable type and static information. This approach allows us to simplify verification conditions significantly. The theorem which provides consistency of the mixed axiomatic semantics has been proved.

REFERENCES

1. Anureev I., Maryasov I., Nepomniaschy V. The Mixed Axiomatic Semantics Method for C-program Verification. // Ershov Informatics Conference: PSI Series, 8th Edition (Preliminary Proceedings). — Novosibirsk: A. P. Ershov Institute of Informatics Systems, 2011. — P. 261–266.
2. Hoare C. A. R. An axiomatic basis for computer programming // Communs ACM. — 1969. — Vol. 12, N 1. — P. 576–580.
3. Maryasov I. V. Towards Automatic Verification of C-light Programs. Mixed Axiomatic Semantics of C-kernel Language. // Perspectives of Systems Informatics (PSI): A. Ershov 7th Int. Conf.: Int. workshop on Program Understanding. — Novosibirsk, 2009. — P. 44–52.
4. Programming languages — C: ISO / IEC 9899:1999. — 1999. — 566 p.

5. Ануреев И. С., Марьясов И. В., Непомнящий В. А. Верификация C-программ на основе смешанной аксиоматической семантики. // Модел. и анализ информ. систем. — 2010. — Т. 17. — № 3. — С. 5–28.
6. Марьясов И. В. На пути к автоматической верификации C-light программ. Смешанная аксиоматическая семантика языка C-kernel. — Новосибирск, 2008. — 32 с. — (Препр. / РАН. Сиб. Отд-ние. ИСИ; № 150).
7. Марьясов И. В. Применение смешанной аксиоматической семантики языка C-kernel к верификации программы топологической сортировки. — Новосибирск, 2010. — 36 с. — (Препр. / РАН. Сиб. Отд-ние. ИСИ; № 155).
8. Непомнящий В. А., Ануреев И. С., Атучин М. М., Марьясов И. В., Петров А. А., Промский А. В. Верификация C-программ в мультиязыковой системе СПЕКТР. // Модел. и анализ информ. систем. — 2010. — Т. 17. — № 4. — С. 88–100.
9. Непомнящий В. А., Ануреев И. С., Михайлов И. Н., Промский А. В. На пути к верификации C программ. Аксиоматическая семантика языка C-kernel. // Программирование. — 2003. — № 6. — С. 1–16.
10. Непомнящий В. А., Ануреев И. С., Михайлов И. Н., Промский А. В. Ориентированный на верификацию язык C-light. // Системная информатика: Сб. научн. труд. — Новосибирск: Изд-во СО РАН, 2004. — Вып. 9: Формальные методы и модели информатики. — С. 51–134.
11. Непомнящий В. А., Атучин М. М., Марьясов И. В., Петров А. А., Промский А. В. Система анализа и верификации C-программ СПЕКТР-2. // Труды семинара «Program Semantics, Specification and Verification». — 5th International Computer Science Symposium in Russia. — Казань, 2010. — С. 76–81.

И. В. Марьясов

МЕТОД СМЕШАННОЙ АКСИОМАТИЧЕСКОЙ СЕМАНТИКИ

Препринт
160

Рукопись поступила в редакцию 01.12. 2011
Редактор А. А. Шелухина
Рецензент И. С. Ануреев

Подписано в печать 20.12.2011
Формат бумаги 60 × 84 1/16 Объём 2,46 уч.-изд. л., 2,69 п. л.
Тираж 60 экз.

Центр оперативной печати «Оригинал 2»
г. Бердск, ул. О. Кошевого, 6, оф. 2, тел. (383-41) 2-12-42