*Programming and Computer Software, Vol. 28, No. 6, 2002, pp. 314–323. Translated from Programmirovanie, Vol. 28, No. 6, 2002.*


# Towards Verification of C Programs. C-Light Language and Its Formal Semantics

**V. A. Nepomniaschy, I. S. Anureev, I. N. Mikhailov, and A. V. Promskii**

*Ershov Institute of Information Systems, Siberian Division, Russian Academy of Sciences, pr. Akademika Lavrent'eva 6, Novosibirsk, 630090 Russia*
*e-mail: vnep@iis.nsk.su; anureev@iis.nsk.su; promsky@iis.nsk.su*

Received June 7, 2002

**Abstract**—The C-light language is described, which is a representative subset of C. C-light permits deterministic expressions, limited use of `switch` and `goto` statements, and, instead of library functions for work with dynamic memory, includes C++ statements `new` and `delete`. A survey of structured operational semantics of the C-light language in Plotkin's style is given.

## 1. INTRODUCTION

Formal verification of programs is a topical trend in modern programming. Of basic interest is verification of programs written in widely used system programming languages, such as C [1, 2] and C++. A sufficiently simple formal semantics is a necessary prerequisite for a language to be convenient for the verification [3, 4]. The C++ language is not convenient for the verification since it does not have a formal semantics. Therefore, the separation of a representative subset of C++, which is convenient for the verification, is a topical problem.

It is natural to study this problem for the C language. Although a formal semantics for the complete C language corresponding to the ANSI standard [2] does not exist either, it has been suggested for a rather representative subset of C, in which, in particular, natural restrictions on the calculation of expressions have been introduced [5, 6]. Note that the semantics suggested in these works is not verification-oriented. A representative subset of C, for which a structured operational semantics in Plotkin's style [7] has been developed, has been suggested in [8, 9]. This subset does not include library functions, `switch` and `goto` statements, and some types, such as, e.g., unions, but it permits nondeterministic expressions. The operational semantics [9] is verification-oriented owing to the inclusion of the theorem proving system HOL into the input language. However, proving program properties in an operational semantics is considerably more complicated than in an axiomatic semantics based on the Hoare logic [3, 4]. Moreover, the lack of library functions in this subset of C [9] makes it impossible to verify C programs that work with dynamic memory.

The goal of this work is to describe the verification-oriented language C-light and to survey its structured operational semantics. This language permits a limited use of `switch` and `goto` statements and includes C++ operations `new` and `delete` to work with dynamic memory.

In Section 2, a survey of the C-light language is given. Basic notions of operational semantics, such as semantic domain, states, and abstract machine configurations, are discussed in Section 3. Axiomatics of a type system is given in Section 4. Sections 5 and 6 are devoted to the semantics of expressions and declarations, respectively. The semantics of statements is discussed in Section 7. In conclusion, advantages of the C-light language and perspectives of the verification method development for programs in this language are discussed.

## 2. C-LIGHT SURVEY

The C-light language is a representative subset of ANSI C. Consider the restrictions determining this subset.

**Types**. Admissible types in C-light are as follows.

• **Base types:**

integer: `bool`, `wchar_t`, `enum`

*i* ::= `char`, `int`, `short`, `long`

τ ::= `signed` *i*, `unsigned` *i*;

real: `float`, `double`, `long double`;

empty: `void`.

• **Derived types:**

pointers: $T$*;

arrays: $T$[n];

structures: `struct`$(T_1 v_1; \ldots; T_n v_n)$;

functions: $T_1 \times \ldots \times T_n \longrightarrow T$.

The new base types of C introduced by the latest standard [2] are not supported. Note also that the name of the logical type is denoted by `bool`, as in C++. The following restrictions are imposed on the admissible types:

0361-7688/02/2806-0314$27.00 

1. The type `char` always has a sign.

2. Elements of enumerated types are objects of type `signed int`.

3. Pointers to functions are forbidden.

4. Specification of an array without specifying its size is allowed only for function argument types.

5. In structures, bit fields are forbidden.

6. Standard library functions are supported only if the source files are available.

7. Functions with a variable number of arguments are forbidden.

8. Abstract declarations of arguments are forbidden.

9. A structure containing an array may not be returned as a value of a function.

**Declarations**. Consider basic differences between the declarations in C-light and those in ANSI C. Indeterminate definitions from ANSI C are forbidden. Abstract declarations of arguments and default arguments are forbidden. An empty list of arguments is not allowed; in this case, `void` is required. All type specifiers and modifiers are forbidden, except for storage class specifiers, sign specifiers, and size specifiers for scalar types. Name resolution rules are extended to specifications. Names of *all* static objects in a code are unique. Unlike in C, "on-the-fly" variable declarations and data type definitions accepted in C++ are permitted. In C-light, the initialization of only one-dimensional arrays is permitted.

**Expressions.** The basic differences of C-light from C are determinism of calculations of expressions and the prohibition of bitwise operations. In C-light, all assignment operations from C are permitted, except for compound assignments containing bitwise operations. Only safe type casting is permitted. For pointers, the additional restriction is introduced: only casting from type `void*` to `T*` is allowed, where `T` is any admissible type. To allocate and deallocate dynamic memory, operations `new` and `delete` from C++ are used.

**Statements.** Conditional expressions in loops and conditional statements may have either scalar or `bool` types. The type of an expression in the `return` statement must be cast by default to the type returned by the function in which this statement is contained. All `case` labels in the `switch` statement must be on the same nesting level. It is forbidden to transfer control by means of `goto` into any block from a block enclosing it or from one block to another if they do not overlap. However, control can be transferred from a nested block to the enclosing one. Like in C++, control transfer by `goto` to bypass initialization is forbidden.

**Source code.** A source code is a sequence of external declarations. C-light has no preprocessor, and, unlike in [9], no preliminary compilation is provided. Modularity is not supported in the process of verification. Therefore, from the standpoint of semantics, any source code consists of one file. User's libraries or the standard C library are considered separately from the code that uses them. The function `main` must not have arguments.

## 3. OPERATIONAL SEMANTICS: BASIC NOTIONS

*Semantics* is a mapping that makes each element from the syntax domain correspond to a certain value or interpretation, i.e., to an element from the semantic domain. Operational semantics describes the process of program execution in terms of changes of states of an abstract machine and, possibly, in terms of the interaction of the program with the environment. The degree of detail in the state description may be different. However, in accordance with the C standard, such an abstract machine must not rely on a particular architecture. In our approach, the machine memory is modeled at a high level of abstractness, whereas the evaluation of expressions relies on a particular approach [10, 11]. This allows us to considerably simplify machine states compared to [9]. We do not consider the standard C library and calls of system functions if the source texts are not available. Therefore, the interaction with the environment is not modeled.

**Semantic domain.** For every type $T$, a set of values $D_T$, called a *support* for the type $T$, is fixed. Since we do not rely on a particular architecture, boundaries of the supports for the base types are given by symbolic constants. Some of them are listed below.

- $D_{\mathbf{bool}} = \{\text{FALSE, TRUE}\}$;
- $D_{\mathbf{unsigned\ char}} = \{0 \ldots \text{MAX\_UNSIGNED\_CHAR}\}$;
- $D_{\mathbf{wchar\_t}} = D_{\mathbf{unsigned\ short}}$;
- $D_{\mathbf{enum}} = D_{\mathbf{signed\ int}}$ (for any enumeration);
- $D_{\mathbf{void}} = \emptyset$ (empty set);
- $D_{T*} = D_{\mathbf{unsigned\ int}}$ for any type $T$;
- $D_{T[\mathbf{n}]} = D_T^n$ (for any admissible type $T$);
- $D_{\mathbf{struct}(T_1 v_1; \ldots; T_n v_n)} = D_{T_i} \times \ldots \times D_{T_n}$;
- $D_{T_1 \times \ldots \times T_n \to T} = D_{T_1} \times \ldots \times D_{T_n} \longrightarrow D_T$ (the set of all functions from the Cartesian product of the sets $D_{T_1}$, …, $D_{T_n}$ to the set $D_T$).

The semantic domain $\boldsymbol{D}$ is defined as the union over all types $\boldsymbol{D} = \bigcup_T D_T$.

For any constant of type $T$, we fix its value in the support $D_T$, and the constant is said to *denote* this value. Each constant of the base type (as well of the type "array" and "structure") is assumed to denote itself. In turn, constants of type "function" denote corresponding functions.

Unlike constants, values of variables are determined in terms of states of the abstract machine.

**Abstract machine state.** By definition, a state of the C-light abstract machine consists of the following components:

(1) ID, information about names of program objects;

(2) ADR, information about addresses of program objects;

(3) MeM, information about the "allocation" of program objects in the memory;

(4) MD, information about values of program objects;

(5) Γ, information about types of program objects;

(6) Σ, information about structures;

(7) TD, information about aliases of the types;

(8) Val, information about the result of the last computed expression;

(9) GLF, information about the nesting level;

(10) BC, program operation history.

Consider these components in more detail.

(1) ID is a family of sets an element of which corresponds to a separate scope, ID = $\{ID^i | ID^i$ is a set of identifiers, $i \geq 0\}$. Index 0 corresponds to the scope, and index $i$ corresponds to the $i$th nesting level in the body of the current function. This approach makes it possible to resolve name conflicts in the program and to model the process of object creation and deletion. When entering a certain block, a new (empty) part $ID^i$ with the value of the index corresponding to the nesting level of the given block is added. When the block is left, this part is removed.

(2) ADR is a family of sets of program object addresses, ADR = $\{ADR^i | ADR^i$ is the set of addresses, $i \geq 0\}$. Addresses of dynamically created objects are stored in the global component $ADR^0$, since these objects are not deleted when they leave the block where they have been created. In all other respects, we have the following correspondence: $ADR^i$ are addresses of the objects from $ID^i$.

(3) MeM is a family of functions, MeM = $\{MeM^i | MeM^i: ID^i \longrightarrow ADR^i, i \geq 0\}$. Since addresses of dynamic objects are stored in $ADR^0$, $MeM^i$ is a bijection only for $i > 0$. For $i = 0$, this a one-to-one mapping.

(4) If the address of an object is known, we can learn the value stored at this address. The component MD makes it possible to get values of modified objects and program constants; i.e., MD: ADR $\longrightarrow$ ***D***. Clearly, this mapping is not completely defined, since the function is also an object.

(5) Information about object types is determined by a mapping from names onto types; i.e., Γ: ID $\longrightarrow$ Types.

(6) The mapping Σ puts a structure tag in correspondence with a sequence of pairs of names and types of the structure members. Like in [9], we use $\Sigma_1$ to denote the sequence of names. The sequence of types is denoted by $\Sigma_2$.

(7) To determine the correspondence between the types and their possible aliases, the component TD is used. This mapping sets up the correspondence between an identifier and the type associated with this identifier by the `typedef` declaration.

(8) When evaluating expressions, it is required to store values of subexpressions. After a function call, the value returned is stored by means of the component Val. In addition to expression values, it can store several special values of the abstract machine, which are considered below when the machine configuration is described. Since the type of a value is uniquely determined by the type system (see Section 4), we may assume that this component specifies the pair: a value and its type.

(9) The first six components are families, with the indices being associated with the nesting levels in the program. The nesting level of the current program construct being executed is determined by the special component, the flag GLF. At the beginning of the program operation, this flag is set equal zero. Upon entering any block, it is increased by one; when leaving a block, it is decreased by one.

(10) Semantics of the jump statements (`goto`, `break`, `continue`) is difficult to determine without information about the context where they are located. For any other statement, it is sufficient to know the current state of the abstract machine. In the case of a jump statement, it is required to know additionally where the control is transferred to. The dependence on the context results in a cumbersome semantics. For `break` and `continue` statements, it is possible to restrict the context to the enclosing loop or `switch` (for `break`) statements. However, for `goto` statement, the context may be, in the general case, arbitrary. For this reason, `goto` statement is not considered at all in many works on formal semantics. In our approach, jump statements generate exceptions, which are detected on the block level. Then, to transfer control to the required point of a program, it is sufficient to know information about the state of the abstract machine at the moment of entry to the block. The tuple BC just stores elements of the first six components of this state that have maximal indices.

**Auxiliary functions.** To define semantics of C-light, the following special functions, which are applied to the states of the abstract machine, are used.

1. Labels determines the set of labels of a block.

2. UnOpSem and BinOpSem specify semantics of unary and binary operations, respectively.

3. InsPars determines mechanism of substitution of actual arguments for formal parameters upon a function call.

4. –– and ++ are the decrement and increment of the flag GLF.

All these functions have an additional implicit argument, namely, the current state.

The argument of the function Labels is the body (a sequence of statements) of some block. The result is a set of names of labels belonging to this block, except

for those beginning from the key words `case` and `default`.

To uniquely specify the calculation of operations in expressions, the functions UnOpSem and BinOpSem are used [9]. The arguments of the former are an operation symbol and the value to which this operation is applied. The arguments of the latter function are an operation symbol, two values, and their types. The result of the function is that of the application of the corresponding operation to the specified values.

The list of parameter declarations in the definition of a function is a sequence of declarations of objects with local definiteness domain. Upon the function call, these objects are initialized by the values of the actual arguments. The processing of this sequence with the simultaneous initialization is implemented in the function InsPars, the arguments of which are a list of function parameter declarations and list of values.

The functions ++ and –– change the value of the flag GLF by one. In addition, they modify the first six components of the state with regard to the change of the nesting level. The function –– deletes elements of the first six components with the indices equal to the value of the flag at the given state and, then, reduces the flag by one. Since jump statements may change the sequence of the execution of program statements, the function ++ must take the history BC into account. This function, first, increases the flag by one and, then, copies elements from the tuple BC to the corresponding components of the new state on the places with the index equal to the new value of the flag. Then, the tuple BC is cleared; i.e., each element becomes an empty set. Thus, the functions –– and ++ are not symmetric.

The set of all states is denoted by the word *States*. To denote the states, we use indexed Greek letters $\sigma$, $\tau$, and others. Simultaneous change of several components can be written, e.g., as follows:

$$\sigma(\mathrm{ID}^i := \mathrm{ID}^i \cup \{\mathrm{id}\}, \Gamma^i(\mathrm{id} \leftarrow \texttt{float}), \mathrm{Val} := 3).$$

An assertion $p$ is said to be true at the state $\sigma$ ($\sigma \models p$) if

(i) $p$ is a variable of type `bool` or `int` and $\mathrm{MD}_\sigma(\mathrm{MeM}_\sigma(p))$ is `TRUE` or nonzero, respectively;

(ii) $p$ depends on expressions $e_1, \ldots, e_n$ and $p(v_1, \ldots, v_n)$ = `TRUE`, where $v_i$ is the result of calculation of the expression $e_i$ at the state $\sigma$ for $i = 1, \ldots, n$ (the expression semantics is described in Section 5).

**Abstract machine configurations.** An operational semantics of a programming language is represented as a set of pairs of configurations of the abstract machine related by transition relations. Like the states, the configurations may be different depending on the programming language and goals. In our case, a classical notion of the configuration is used, since all language and computation specific features are hidden in the states.

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

Program semantics is defined by induction on the program structure in terms of the transition $\longrightarrow$ relation. Let us describe axioms and rules of the operational semantics of the C-light language. Axioms are represented as pairs of configurations related by a transition relation. The notation $\langle A, \sigma\rangle \longrightarrow \langle B, \tau\rangle$ means that one step of the execution of the fragment $A$ of the source program starting from the state $\sigma$ results in the state $\tau$, and $B$ is the fragment of the source program that remains for the execution. Any semantic rule has the form

$$\frac{P_1 \ldots P_n}{\langle A, \sigma\rangle \longrightarrow \langle B, \tau\rangle},$$

which implies that, if conditions $P_1, \ldots, P_n$ are fulfilled, there exists a transition from the configuration $\langle A, \sigma\rangle$ to the configuration $\langle B, \tau\rangle$.

An empty fragment is denoted by $\epsilon$. An empty fragment may be both an empty program and an empty expression. An undefined value is denoted by $\omega$.

Below is the list of special values of the abstract machine that can be stored in the component Val in addition to the values of program types (unlike ordinary values, these values do not have types and inform about an event in the course of program operation):

(1) OkVal, the operator has normally been terminated;

(2) GoVal(L), the statement `goto L` has been met;

(3) BreakVal, the statement `break` has been met;

(4) ContVal, the statement `continue` has been met;

(5) RetVal(e), the statement `return e` has been met;

(6) CaseVal(e), the statement `switch (e){…}` has been met;

(7) $\omega$, the expression has been calculated with an indeterminate result;

(8) Fail, the statement has been terminated abnormally.

These values are explained in a more detail in Section 7.

## 4. TYPE SYSTEM

Let us define typing rules for C-light expressions. This implies that we will define the static semantics, or *type system*, of C-light. As in C, the value of an expression may be a link to an object in the memory, the so-called L-value. The set of L-values of type $T$ is denoted as lv[$T$].

Numbers are represented by the letter $n$; characters, by $c$; and variables, functions, and structure members, by `id`. An expression is separated from its type by the colon. The type of expressions in C-light is defined by induction. The induction base is axioms for constants, variables, and statement `sizeof`.

The first group consisting of seven axioms defines types of numerical constants,

$n$ : `signed int`; $n_1.n_2$[E$n_3$] : `double`;

$n$`L` : `long`; $n_1.n_2$[E$n_3$]`F` : `float`;

$n$`U` : `unsigner int`; $n_1.n_2$[E$n_3$]`L` : `long double`;

$n$`S` : `short int`.

The second group of four axioms specifies types of symbolic and string constants,

$$'c' : \texttt{char};\ \text{“}c_1 \ldots c_n\text{”} : \texttt{char}[n+1];$$

$$\texttt{L}'c_1c_2' : \texttt{wchar\_t};\ \texttt{L}\text{“}c_1 \ldots c_n\text{”} : \texttt{wchar\_t}\left[\frac{n}{2}+1\right].$$

Two axioms are used for zero pointer, `NULL`: `void*` and 0: $T$*.

The type of an object named a simple identifier `id` is defined by the axiom `id`: $\Gamma$(`id`).

The next two axioms are due to the new C standard [2] (in view of the incorporation of arrays of variable length, the operation `sizeof` is no longer a constant calculated on the compilation stage):

`sizeof`(*expn*) : `unsigned int`;

`sizeof`(*type_spec*) : `unsigned int`.

Types of all other expressions are determined by inference rules. The first two are related to referencing and dereferencing operations.

$$\frac{e : \mathrm{lv}[T]}{\&e : T*};\quad \frac{e : T* \quad T \neq \texttt{void}}{*e : \mathrm{lv}[T]}.$$

Note that there is no separate rule for indexed expressions, since the expression `a[i]` is a convenient notation for `*(a + i)`.

The relation between L-values and ordinary values is specified by the following two rules:

$$\frac{e : \mathrm{lv}[T] \quad T \text{ is not an array}}{e : T};\quad \frac{e : \mathrm{lv}[T[n]]}{e : T*}.$$

This relation is violated only for arrays that are nonmodifiable L-values. In all expressions where the value is required, the array identifier is meant as a pointer to the first element, except for the case of the operations `sizeof` and `&`.

Two rules for structures are considered separately,

$$\frac{e : \mathrm{lv}[\texttt{struct } s] \quad (\texttt{id}, T) \in \Sigma(s)}{e.\texttt{id} : \mathrm{lv}[T]};$$

$$\frac{e : \texttt{struct } s \quad (\texttt{id}, T) \in \Sigma(s) \quad T \text{ is not an array}}{e : \texttt{id} : T}.$$

A structure may be either an L-value or an ordinary value (e.g., upon return from a function). The notation of the form `ptr->f` is written as `(*ptr).f`.

The rules for the type casting with regard to the constraints on the C-light language have the form

$$\frac{e : T_0 \quad (T_0 \text{ and } T \text{ are scalar types}) \vee (T = \texttt{void})}{(T)e : T}.$$

To simplify rules for function calls and assignments, a default casting relation, denoted as *IC*, is introduced. This relation allows us to avoid cumbersome representation of the exact matching argument types. The relation *IC* is an equivalence relation [9]. Let *IC*(`void*`, $T$*) for any $T$ and, if $T_1$ and $T_2$ are arithmetic types, $IC(T_1, T_2)$.

The rule for a function call has the form

$$\frac{e : T_1 \times \ldots \times T_n \longrightarrow T \qquad \forall i.1 \le i \le n \Rightarrow \exists T'.(e_i : T' \wedge IC(T_i, T'))}{e(e_1, \ldots, e_n) : T}.$$

For a compact representation of typing rules for operations, special predicates $P_{op}$, where *op* is the operation symbol, are defined. We use symbols $\square$ and $\odot$ for unary and binary operations, respectively. Postfix increment/decrement operations are considered separately:

$$\frac{e : T}{\square e : T};\quad \frac{e_1 : T_1 \quad e_2 : T_2 \quad P_{\odot}(T_1, T_2, T)}{e_1 \odot e_2 : T};$$

$$\frac{e_0 : T_0 \text{ is a scalar} \quad e_1 : T_1 \quad e_2 : T_2 \quad P_{?}(T_1, T_2, T)}{e_0\ ?\ e_1 : e_2 : T}.$$

In the assignment operations, the expression on the right-hand side must be a modifiable L-value. Rules for a simple and compound assignments have the following form:

$$\frac{e_1 : \mathrm{lv}[T] \quad e_2 : T_0 \quad IC(T_0, T) \quad T \text{ is not an array}}{e_1 = e_2 : T};$$

$$\frac{e_1 : \mathrm{lv}[T] \text{ is not an array} \quad e_1 \odot e_2 : T_0 \quad IC(T_0, T)}{e_1 \odot= e_2 : T}.$$

Postfix increment and decrement operations are defined in the same way. Note that the prefix increment and decrement can be processed by means of the rule for the compound assignment, since the expressions `++i` and `--i` are equivalent to `i+=1` and `i-=1`,

$$\frac{e : \mathrm{lv}[T] \quad T \text{ is an a scalar type}}{e{+}{+} : T}.$$

## 5. SEMANTICS OF EXPRESSIONS

To specify expression semantics, two groups of rules are used. In the first group, normal calculation of expressions is considered. This group is presented below. In the second group, calculations resulting in an error are considered. The difference between the corresponding rules from these groups is as follows. If there appears an error when calculating an expression, the

value ω is recorded in the component Val, and all subsequent calculations are ignored in accordance with the indeterminate value extension rule

$$\frac{\mathrm{Val}_\sigma = \omega}{\langle e, \sigma\rangle \longrightarrow \langle \epsilon, \sigma\rangle}.$$

The rules from the second group are not presented here.

**L-expressions.** This group of rules describes the semantics of L-expressions. As in C, an L-expression is an expression the value of which is a link to an object in the memory.

Variables are identifiers whose type is not array or function,

$$\frac{\Gamma_\sigma(\mathtt{id}) = \mathrm{lv}[\tau] \quad \tau \text{ is not an array or function}}{(\mathtt{id}, \sigma) \longrightarrow \langle \epsilon, \sigma(\mathrm{Val} := (\mathrm{MD}_\sigma(\mathrm{MeM}_\sigma(\mathtt{id})), \tau))\rangle}.$$

Arrays,

$$\frac{\Gamma_\sigma(\mathtt{id}) = \mathrm{lv}[\tau[n]]}{(\mathtt{id}, \sigma) \longrightarrow \langle \epsilon, \sigma(\mathrm{Val} := (\mathrm{MD}_\sigma(\mathrm{MeM}_\sigma(\mathtt{id})), \tau^*))\rangle}.$$

Indexed expression,

$$\sigma \vDash a : \mathrm{lv}[\tau[n]] \qquad \tau \text{ is an array}$$

$$\langle e, \sigma\rangle \longrightarrow^* \langle \epsilon, \sigma_1\rangle \quad \mathrm{Val}_{\sigma_1} = (v, \tau') \quad \tau' \text{ is a scalar}$$

$$\frac{\langle a, \sigma_1\rangle \longrightarrow^* \langle \epsilon, \sigma_2\rangle \quad \mathrm{Val}_{\sigma_2} = (c, \tau^*)}{\langle a[e], \sigma\rangle \longrightarrow \langle \epsilon, \sigma_2(\mathrm{Val} := (\mathrm{MD}_{\sigma_2}(c + v), \tau))\rangle}.$$

Element selection expression,

$$\frac{\Gamma_\sigma(s) = \mathrm{lv}[\mathtt{struct}(\tau_1, \ldots, \tau_n)] \quad (\mathtt{id}, \tau) \in \Sigma_\sigma(s)}{\langle s.\mathtt{id}, \sigma\rangle \longrightarrow \langle \epsilon, \sigma(\mathrm{Val} := (\mathrm{MD}_\sigma(\mathrm{MeM}_\sigma(s)).\mathtt{id}, \tau))\rangle}.$$

Dereferencing expression,

$$\sigma \vDash e : \mathrm{lv}[\tau^*] \quad \tau \text{ is not an array or function}$$

$$\frac{\langle e, \sigma\rangle \longrightarrow^* \langle \epsilon, \sigma_1\rangle \quad \mathrm{Val}_{\sigma_1} = (v, \tau^*)}{\langle {}^*e, \sigma\rangle \longrightarrow \langle \epsilon, \sigma_1(\mathrm{Val} := (\mathrm{MD}_{\sigma_1}(v), \tau))\rangle}.$$

Direct addressing expression,

$$\frac{\sigma \vDash e : \mathrm{lv}[\tau^*] \quad \tau \text{ is not a function} \quad \langle e, \sigma\rangle \longrightarrow^* \langle \epsilon, \sigma'\rangle}{\langle \&e, \sigma\rangle \longrightarrow \langle \epsilon, \sigma'(\mathrm{Val} := (\mathrm{MeM}_\sigma(e), \tau^*))\rangle}.$$

**Operations resulting in values.** Let ⊙ be a binary C-light operation without side effects that is different from the logical operations AND, OR and from the sequential calculation operation. Then, the rule for this operation is given by

$$\langle e_2, \sigma_0\rangle \longrightarrow^* \langle \epsilon, \sigma_1\rangle \quad \langle e_1, \sigma_1\rangle \longrightarrow^* \langle \epsilon, \sigma_2\rangle$$

$$\frac{\mathrm{Val}_{\sigma_1} = (v_2, \tau_2) \quad \mathrm{Val}_{\sigma_2} = (v_1, \tau_1) \quad IC(\tau_1, \tau_2)}{\langle (e_1 \odot e_2), \sigma\rangle \longrightarrow \langle \epsilon, \sigma_2'\rangle},$$

where $\sigma_2' = \sigma_2(\mathrm{Val} := \mathrm{BinOpSem}(\odot, v_1, \tau_1, v_2, \tau_2))$.

For a unary operation $\square \in \{+, -, !, (.), \mathtt{sizeof}\}$, the rule has the form

$$\frac{\langle e_2, \sigma\rangle \longrightarrow^* \langle \epsilon, \sigma_1'\rangle}{\langle \square e, \sigma\rangle \longrightarrow \langle e, \sigma'(\mathrm{Val} := \mathrm{UnOpSem}(\square, \mathrm{Val}_{\sigma'}))\rangle}.$$

The rule for the type casting operation uses the family of functions such that the function $\gamma_{\tau_1, \tau_2}$ transforms a value of type $\tau_1$ to a value of type $\tau_2$:

$$\frac{\langle e, \sigma\rangle \longrightarrow^* \langle \epsilon, \sigma'\rangle \quad \mathrm{Val}_{\sigma'} = (v_0, \tau_0) \quad \gamma_{\tau_0, \tau}(v_0) = v}{\langle (\tau)e, \sigma\rangle \longrightarrow \langle \epsilon, \sigma'(\mathrm{Val} := (v, \tau))\rangle}.$$

The sequential calculation operation is classified as an operation producing a value. Let the expression *head* do not contain this operation on the upper level. Then,

$$\frac{\langle head, \sigma\rangle \longrightarrow^* \langle \epsilon, \sigma'\rangle}{\langle head, tail, \sigma\rangle \longrightarrow \langle tail, \sigma'(\mathrm{Val} := \mathrm{OkVal})\rangle}.$$

The logical operations AND and OR, as well as the conditional operation, are separated into a separate group. Expressions containing these operations may be calculated not completely. Here is an example of one of eight rules from this group:

$$\frac{\langle e_1, \sigma\rangle \longrightarrow^* \langle \epsilon, \sigma'\rangle \quad \mathrm{Val}_{\sigma'} = (0, \tau) \quad \tau \text{ is a scalar}}{\langle e_1 \&\& e_2, \sigma\rangle \longrightarrow \langle \epsilon, \sigma'(\mathrm{Val} = (\mathtt{FALSE}, \mathtt{bool}))\rangle}.$$

**Operations with side effects.** Any change of the memory content in a C-light program occurs through side effects. All side effects generated upon calculation of expressions are calculated immediately, which is different from [9], where a multiset of delayed side effects is used. The rule for a simple assignment has the form

$$\sigma \vDash e_1 : \mathrm{lv}[\tau_1]\langle e_2, \sigma_0\rangle$$
$$\longrightarrow^* \langle \epsilon, \sigma_1\rangle\langle e_1, \sigma_1\rangle \longrightarrow^* \langle \epsilon, \sigma_2\rangle$$

$$\frac{\mathrm{Val}_{\sigma_1} = (v_2, \tau_2) \quad \mathrm{MeM}_{\sigma_2}(e_1) = c \quad IC(\tau_1, \tau_2)}{\langle e_1 = e_2, \sigma_0\rangle \longrightarrow \langle \epsilon, \sigma_2''\rangle},$$

where $\sigma_2'' = \sigma_2(\mathrm{MD}(c \leftarrow v_2))(\mathrm{Val} := (\gamma_{\tau_2, \tau_1}(v_2), \tau_1))$.

The premise of a rule for a compound assignment differs from the premise of the previous rule by only the addition of $\mathrm{Val}_{\sigma_2} = (v_1, \tau_1)$, and the inference has the form $\langle e_1 \odot = e_2, \sigma_0\rangle \longrightarrow \langle \epsilon, \sigma_2''\rangle$, where

$$\sigma_2'' = \sigma_2(\mathrm{MD}(c \leftarrow v_{res}))(\mathrm{Val} := (\gamma_{\tau, \tau_1}(v_{res}), \tau_1)),$$

$$(v_{res}, \tau) = \mathrm{BinOpSem}(\odot, v_1, \tau_1, v_2, \tau_2).$$

Thus, the side effects possible in $e_1$ are not duplicated.

For the increment, the rules have the form

$$\frac{\sigma \models e : \mathrm{lv}[\tau] \quad \tau \text{ is not an array}}{\langle ++e, \sigma\rangle \longrightarrow \langle e+=1, \sigma\rangle};$$

$$\frac{\sigma \models e : \mathrm{lv}[\tau] \quad \tau \text{ is not an array} \quad \langle e, \sigma\rangle \longrightarrow^* \langle \epsilon, \sigma_1\rangle \quad \mathrm{Val}_{\sigma_1} = (v, \tau) \quad \mathrm{MeM}_{\sigma_1}(e) = c}{\langle e{+}{+}, \sigma\rangle \longrightarrow \langle \epsilon, \sigma_1(\mathrm{MD}(c \leftarrow (v+1)))\rangle}.$$

The rules for the decrement have a similar form.

**Operations on dynamic memory.** To control dynamic memory, C-light uses C++ operations `new` and `delete`. Let $\tau$ be a type different from "function," and let *nc* be a new address. The axiom and rule for the operation `new` have the form

$$\langle \mathtt{new}\ \tau, \sigma\rangle \longrightarrow \langle \epsilon, \sigma'\rangle,$$

where

$$\sigma' = \sigma(\mathrm{ADR}^0 \leftarrow \mathrm{ADR}^0 \cup \{nc\})$$
$$(\mathrm{MD}^0(nc \leftarrow 0))(\mathrm{Val} := (nc, \tau^*));$$
$$\langle \mathtt{size}, \sigma\rangle \longrightarrow^* \langle \epsilon, \sigma'\rangle$$
$$\frac{\mathrm{Val}_{\sigma'} = (v, \mathtt{unsigned\ int}) \quad v \geq 0}{\langle \mathtt{new}\ \tau[size], \sigma\rangle \longrightarrow \langle \epsilon, \sigma''\rangle},$$

and $\sigma'' = \sigma'(\mathrm{ADR}^0 \leftarrow \mathrm{ADR}^0 \cup \{nc+0\})$

$$(\mathrm{MD}^0(nc+0 \leftarrow 0))$$
$$(\mathrm{ADR}^0 \leftarrow \mathrm{ADR}^0 \cup \{nc+1\})$$
$$(\mathrm{MD}^0(nc+1 \leftarrow 0))$$
$$\ldots$$
$$(\mathrm{ADR}^0 \leftarrow \mathrm{ADR}^0 \cup \{nc+v-1\})$$
$$(\mathrm{MD}^0(nc+v-1 \leftarrow 0))$$
$$(\mathrm{Val} := (nc+0, \tau^*)).$$

Thus, the allocation of memory implies that a new address for a simple object and a set of new addresses for an array appear.

The rule for the operation `delete` has the form

$$\frac{\sigma \models e : \mathrm{lv}[\tau^*] \quad \tau \text{ is not a function}}{\langle \mathtt{delete}\ e, \sigma\rangle \longrightarrow \langle \epsilon, \sigma'\rangle},$$

where $\sigma' = \sigma(\mathrm{ADR}^0 \leftarrow \mathrm{ADR}^0 \backslash \{\mathrm{MD}^i(\mathrm{MEM}^i(e))\})$, $i = \mathrm{GLF}_\sigma$.

The rule for the operation `delete` [] has a similar form, since the deallocation of memory in both cases is modeled by simply removing the first address.

**Function calls.** When a function is called, the control is transferred to its body with the argument passing by value. Let $\sigma \models f : \tau_1 \times \ldots \times \tau_n \longrightarrow \tau$. The semantics of a function call is represented by two rules: in the first rule, the values of arguments are calculated; in the second rule, the control is transferred with passing the values obtained,

$$\langle e_n, \sigma_0\rangle \longrightarrow^* \langle \epsilon, \sigma_1\rangle \quad \mathrm{Val}_{\sigma_1} = (v_n, \tau_n')IC(\tau_n, \tau_n')$$
$$\frac{\langle e_1, \sigma_n\rangle \longrightarrow^* \langle \epsilon, \sigma_{n+1}\rangle \quad \mathrm{Val}_{\sigma_{n+1}} = (v_1, \tau_1')IC(\tau_1, \tau_1')}{\langle f(e_1, \ldots, e_n), \sigma_0\rangle \longrightarrow \langle FCall(f)(v_1, \ldots, v_n), \sigma_{n+1}\rangle};$$

$$\tau f(pars)\{S\} \text{ definition of function } f$$
$$\langle S, \mathrm{InsPars}(\sigma(\mathrm{GLF}{+}{+}),$$
$$\mathrm{pars}, [v_1, \ldots, v_n]\rangle \longrightarrow^* \langle \epsilon, \sigma'\rangle$$
$$\frac{\mathrm{Val}_{\sigma'} = \mathrm{RetVal}(v, \tau') \vee \mathrm{OkVal} \quad IC(\tau, \tau')}{\langle \mathbf{FCall}(f)(v_1, \ldots, v_n), \sigma\rangle \longrightarrow \langle \epsilon, \sigma''\rangle},$$

where $\sigma'' = \sigma'(\mathrm{Val} := \mathrm{Val}_{\sigma'}(\mathrm{GLF}{-}{-}))$.

The normal termination of a function call that does not return a value corresponds to the value OkVal.

## 6. SEMANTICS OF DECLARATIONS

Let us consider axioms and rules for the declaration of types and objects. The value of the index *i* in the rules for declarations is equal to 0 if *storage* has the form `static` and $\mathrm{GLF}_\sigma$ in the case of `auto`. Below, only rules for the declarations of variables with the initialization are considered. For the initialization, a universal function *Init* is used. It has a variable number of arguments, with the first two arguments being required (the object name and its storage class). The third argument may be an initializing value. If it is lacking, the variable is initialized by default. The rules for the declaration of variables without initialization differ from those with the initialization only by the absence of the rule premise and the third argument of the function *Init*. Let *nc* denote further a new address.

**Types.** The semantics of the declaration `typedef` is defined by the following two axioms:

$$\langle \mathtt{typedef}\ type\_spec\mathtt{Id};, \sigma\rangle$$
$$\longrightarrow \langle \epsilon, \sigma(\mathrm{TD}(\mathtt{Id} \leftarrow \mathrm{TD}(type\_spec)))\rangle;$$
$$\langle \mathtt{typedefstruct}\{\ fields\}\mathtt{Id};, \sigma\rangle \longrightarrow \langle S, \sigma\rangle,$$

where *S* has the form

$$\mathtt{struct\ new\_tag}\{\ feilds\};$$
$$\mathtt{typedef\ structnew\_tag\ Id};$$

and `new_tag` is a new unique identifier. New tags for nameless structure types are introduced to make the mapping $\Sigma$ defined.

**Memory class refinement and decomposition.** This group of axioms places default memory class specifiers, and divides sequences of declarators into separate declarations,

$$(type\_spec\ drator;, \sigma)$$
$$\longrightarrow \langle storage\ type\_spec\ drator;, \sigma\rangle,$$

where *storage* is `static` if $\mathrm{GLF}_\sigma = 0$ and `auto` other-

wise,

$$\langle \texttt{register}\ \textit{type\_spec drator};, \sigma\rangle$$
$$\longrightarrow \langle \texttt{auto}\ \textit{type\_spec drator};, \sigma\rangle;$$

$$\langle \textit{storage}_{opt}\textit{type\_spec drator}, \textit{drator\_list};, \sigma\rangle$$
$$\longrightarrow \langle \textit{storage}_{opt}\textit{type\_spec drator};$$
$$\textit{storage}_{opt}\textit{type\_spec drator\_list};, \sigma\rangle.$$

**Simple variables.** Let TD($\tau$) be not a structure or enumeration. The declaration rule for a simple variable has the form

$$\frac{\langle e, \sigma\rangle \longrightarrow^* \langle \epsilon, \sigma'\rangle \quad \mathrm{Val}_{\sigma'} = (v, \texttt{unsigned int}) \quad v \geq 0}{\langle \textit{storage}\ \tau\ \texttt{id[}e\texttt{]} = \{\textit{init\_list}\};, \sigma\rangle \longrightarrow \langle \epsilon, \sigma''\rangle},$$

where

$$\sigma'' = \sigma'(\mathrm{ID}^i \leftarrow \mathrm{ID}^i \cup \{\texttt{id}\})$$
$$(\mathrm{ADR}^i \leftarrow \mathrm{ADR}^i \cup \{nc + 0, \ldots, nc + v\})$$
$$(\mathrm{MeM}(\texttt{id} \leftarrow nc))\ (\Gamma^i\ (\texttt{id} \leftarrow \tau[v]))$$
$$(\mathcal{J}\textit{nit}(\texttt{id}, \textit{storage}, \textit{init\_list}))$$
$$(\mathrm{Val} := \mathrm{OkVal}).$$

A declaration for a multidimensional array is represented as a sequence of declarations for one-dimensional arrays by means of the axiom

$$\langle \textit{storage}\ \tau\ \texttt{id[}e\texttt{]}\ \textit{dimensions};, \sigma\rangle \longrightarrow \langle S, \sigma\rangle,$$

where $S$ has the form

$$\texttt{typedef}\ \tau\ \texttt{new\_id}\ \textit{dimensions};$$
$$\textit{storage}\ \texttt{new\_id*}\ \texttt{id[}e\texttt{]};$$

and new_id is a new unique identifier.

The declaration rule for a pointer is given by

$$\frac{\langle e, \sigma\rangle \longrightarrow^* \langle \epsilon, \sigma'\rangle \quad \mathrm{Val}_{\sigma'} = (v, \tau^*)}{\langle \textit{storage}\tau\texttt{*}\ \texttt{id} = e;, \sigma\rangle \longrightarrow \langle \epsilon, \sigma'\rangle},$$

where

$$\sigma' = \sigma(\mathrm{ID}^i \leftarrow \mathrm{ID}^i \cup \{\texttt{id}\})(\mathrm{ADR}^i \leftarrow \mathrm{ADR}^i \cup \{nc\})$$
$$(\mathrm{MeM}(\texttt{id} \leftarrow nc))(\Gamma^i(\texttt{id} \leftarrow \tau^*))$$
$$(\mathcal{J}\textit{nit}(\texttt{id}, \textit{storage}, v))\ (\mathrm{Val} := \mathrm{OkVal}).$$

**Structures and enumerations.** The axiom for a declaration of a variable of the structure type has the form

$$\langle \textit{storage}\ \texttt{struct tag id} = \{\textit{init\_list}\};, \sigma\rangle \longrightarrow \langle \epsilon, \sigma'\rangle,$$

where

$$\sigma' = \sigma(\mathrm{ID}^i \leftarrow \mathrm{ID}^i \cup \{\texttt{id}\})(\mathrm{ADR}^i \leftarrow \mathrm{ADR}^i \cup \{nc\})$$
$$(\mathrm{MeM}(\texttt{id} \leftarrow nc))(\Gamma^i(\texttt{id}^{\,i} \leftarrow \texttt{struct}\{\Sigma_2(\texttt{id})\}))$$
$$(\mathcal{J}\textit{nit}(\texttt{id}, \textit{storage}, \textit{init\_list}))$$
$$(\mathrm{Val} := \mathrm{OkVal}).$$

In addition to this axiom, there are several auxiliary axioms for the fragment where a structure type and objects of this type are simultaneously declared. Such a fragment is divided into separate declarations of the type and objects.

A declaration of an enumeration is treated as simultaneous declaration of several variables of type signed int,

$$\langle \texttt{enum tag}\{\textit{drator\_list}\};, \sigma\rangle$$
$$\longrightarrow \langle \texttt{signed int}\ \textit{drator\_list};, \sigma\rangle;$$
$$\langle \texttt{enum tag}\{\textit{drator\_list}_1\}\textit{drator\_list}_2;, \sigma\rangle \longrightarrow \langle S, \sigma\rangle,$$

where $S \equiv \texttt{signed int}\ \textit{drator\_list}_1, \textit{drator\_ list}_2;$.

**Functions.** A prototype and a function definition are processed by one rule (the rule for the function main is given in the next section),

$$\frac{\texttt{id} \neq \texttt{main} \quad B \text{ is a blok or an empty statement}}{\langle \tau\ \texttt{id}\ (\tau_1 v_1, \ldots, \tau_n v_n)B, \sigma\rangle \longrightarrow \langle \epsilon, \sigma'\rangle},$$

where

$$\sigma' = \sigma(\mathrm{ID}^0 \leftarrow \mathrm{ID}^0 \cup \{\texttt{id}\})$$
$$(\Gamma^0(\texttt{id} \leftarrow (\mathrm{TD}(\tau_1) \times \ldots \times \mathrm{TD}(\tau_n) \longrightarrow \mathrm{TD}(\tau))))$$
$$(\mathrm{Va} := \mathrm{OkVal}).$$

## 7. SEMANTICS OF STATEMENTS

We assume that, if no assumptions about the component $\mathrm{Val}_\sigma$ (where $\sigma$ is the state from which the execution of the considered statement starts) are made in the rule premise, then the implicit premise $\mathrm{Val}_\sigma = \mathrm{OkVal}$ is suggested. If a rule requires a different premise, the latter will be written explicitly.

**Empty statement:**

$$\langle ;, \sigma\rangle \longrightarrow \langle \epsilon, \sigma\rangle.$$

**Statement–expression:**

$$\frac{\langle e, \sigma_0\rangle \longrightarrow^* \langle \epsilon, \sigma\rangle \quad \mathrm{Val}_\sigma \neq \omega}{\langle e;, \sigma_0\rangle \longrightarrow^* \langle \epsilon, \sigma(\mathrm{Val} := \mathrm{OkVal})\rangle};$$

$$\frac{\langle e, \sigma_0\rangle \longrightarrow^* \langle \epsilon, \sigma\rangle \quad \mathrm{Val}_\sigma = \omega}{\langle e;, \sigma_0\rangle \longrightarrow \langle \epsilon, \sigma(\mathrm{Val} := \mathrm{Fail})\rangle}.$$

**Labeled statement.** Rules for labeled statements are divided into three groups depending on the label type.

Rules for statements marked by `case`-labels have the form

$$\frac{\mathrm{Val}_\sigma = \mathrm{OkVal}}{\langle \texttt{case}\ c\texttt{:}\ S, \sigma\rangle \longrightarrow \langle S, \sigma\rangle};$$

$$\frac{\mathrm{Val}_\sigma = \mathrm{CaseVal}(c)}{\langle \texttt{case}\ c\texttt{:}\ S, \sigma\rangle \longrightarrow \langle S, \sigma(\mathrm{Val} := \mathrm{OkVal})\rangle};$$

$$\frac{\mathrm{Val}_\sigma = \mathrm{BreakVal} \vee (\mathrm{CaseVal}(c') \wedge c \neq c')}{\langle \texttt{case}\ c\texttt{:}\ S, \sigma\rangle \longrightarrow \langle \epsilon, \sigma\rangle}.$$

Rules for statements marked by `default`-labels have the form

$$\frac{\mathrm{Val}_\sigma = \mathrm{OkVal} \vee \mathrm{CaseVal}(..)}{\langle \texttt{default:}\ S, \sigma\rangle \longrightarrow \langle S, \sigma(\mathrm{Val} := \mathrm{OkVal})\rangle};$$

$$\frac{\mathrm{Val}_\sigma \neq \mathrm{OkVal} \vee \mathrm{CaseVal}(..)}{\langle \texttt{default:}\ S, \sigma\rangle \longrightarrow \langle \epsilon, \sigma\rangle}.$$

Rules for statements marked by ordinary labels have the form

$$\frac{\mathrm{Val}_\sigma = \mathrm{OkVal}}{\langle \texttt{L}:S, \sigma\rangle \longrightarrow \langle S, \sigma\rangle};$$

$$\frac{\mathrm{Val}_\sigma = \mathrm{GoVal}(\texttt{L})}{\langle \texttt{L}:S, \sigma\rangle \longrightarrow \langle S, \sigma(\mathrm{Val} := \mathrm{OkVal})\rangle};$$

$$\frac{\mathrm{Val}_\sigma = \mathrm{GoVal}(\texttt{L'}) \quad \texttt{L} \neq \texttt{L'}}{\langle \texttt{L}:S, \sigma\rangle \longrightarrow \langle \epsilon, \sigma\rangle}.$$

**Control transfer and exclusions.** The method suggested in [9] has been extended to the statement `goto`. As a result, we got a unique semantics for all control transfer statements. Any statement of this kind results in a certain exclusive value. In other constructs, these exclusions are intercepted by analogy with the exclusion processing mechanism.

$$\langle \texttt{goto L;}, \sigma\rangle \longrightarrow \langle \epsilon, \sigma(\mathrm{Val} := \mathrm{GoVal}(\mathrm{L}))\rangle;$$

$$\langle \texttt{break;}, \sigma\rangle \longrightarrow \langle \epsilon, \sigma(\mathrm{BreakVal})\rangle;$$

$$\langle \texttt{continue;}, \sigma\rangle \longrightarrow \langle \epsilon, \sigma(\mathrm{Val} := \mathrm{ContVal})\rangle;$$

$$\langle \texttt{return;}, \sigma\rangle \longrightarrow \langle \epsilon, \sigma(\mathrm{Val} := \mathrm{RetVal}(\emptyset, \texttt{void}))\rangle;$$

$$\frac{\langle e, \sigma\rangle \longrightarrow^* \langle \epsilon, \sigma'\rangle \quad \mathrm{Val}_{\sigma'} = (v, \rho)}{\langle \texttt{return}\ e;, \sigma\rangle \longrightarrow \langle \epsilon, \sigma'(\mathrm{Val} := \mathrm{RetVal}(v, \tau))\rangle}.$$

The advantage of this approach is that there is no need to look for the program point where the control is transferred to. It is much simpler to ignore all subsequent statements until the control occurs at the desired point or at the place where the desired point is easily found.

An unmarked statement is to be ignored when processing special values GoVal(..), BreakVal, ContVal, RetVal(..), and Fail. Therefore, for an unmarked statement $S$, when $\mathrm{Val}_\sigma$ coincides with one of these values, we arrive at the axiom $\langle S, \sigma\rangle \longrightarrow \langle \epsilon, \sigma\rangle$.

**Compound statements.** The basic rule of statement composition has a standard form,

$$\frac{S_1 \text{ is a statement } \langle S_1, \sigma\rangle \longrightarrow \langle S_2, \sigma'\rangle}{\langle S_1 T, \sigma\rangle \longrightarrow \langle S_2 T, \sigma'\rangle}.$$

Semantics of the block operator depends on whether `goto L` operation has been executed in this block and on the location of the label $L$. Depending on whether the condition $\mathrm{Val}_{\sigma'} = \mathrm{GoVal}(\mathrm{L}) \wedge \mathrm{L} \in \mathrm{Labels}(S)$ is fulfilled, we obtain two following rules for the block operator:

$$\langle S, \sigma(\mathrm{GLF{+}{+}})\rangle \longrightarrow^* \langle \epsilon, \sigma'\rangle$$

$$\frac{\sigma'' = \sigma'(\mathrm{BC} := [\mathrm{ID}^i, \mathrm{ADR}^i, \mathrm{MeM}^i, \mathrm{MD}^i, \Gamma^i, \Sigma^i])}{\langle \{S, \sigma\} \longrightarrow \langle \{S\}, \sigma''(\mathrm{GLF{-}{-}})\rangle\rangle};$$

$$\frac{\langle S, \sigma(\mathrm{GLF{+}{+}})\rangle \longrightarrow^* \langle \epsilon, \sigma'\rangle}{\langle \{S, \sigma\} \longrightarrow \langle \epsilon, \sigma'(\mathrm{GLF{-}{-}})\rangle\rangle},$$

where $i = \mathrm{GLF}_{\sigma'}$.

**Conditional statements.** Semantics of `if` statement is determined by four rules. The first rule adds missing `else` branches. Three other rules correspond to different values of the conditional expression: indefinite, true, and false values. For example, the rule for the false value of a conditional expression is given by

$$\frac{\langle e, \sigma\rangle \longrightarrow^* \langle \epsilon, \sigma'\rangle \quad \mathrm{Val}_{\sigma'} = (0, \tau) \quad IC(\tau, \texttt{int})}{\langle \texttt{if(}e\texttt{)}S_1\texttt{else}S_2, \sigma\rangle \longrightarrow \langle S_2, \sigma'(\mathrm{Val} := \mathrm{OkVal})\rangle}.$$

The incorporation of the value CaseVal(..) makes it possible not to look for the required branch of the statement `switch` but to go to its body,

$$\frac{\langle e, \sigma\rangle \longrightarrow^* \langle \epsilon, \sigma'\rangle \quad \mathrm{Val}_{\sigma'} = \omega}{\langle \texttt{switch(}e\texttt{)\{}B\texttt{\}}, \sigma\rangle \longrightarrow \langle \epsilon, \sigma'(\mathrm{Val} := \mathrm{Fail})\rangle};$$

$$\langle e, \sigma\rangle \longrightarrow^* \langle \epsilon, \sigma_1\rangle \quad \mathrm{Val}_{\sigma_1} = (v, \tau) \quad IC(\tau, \texttt{int});$$

$$\langle B, \sigma_1(\mathrm{GLF{+}{+}})(\mathrm{Val} := \mathrm{CaseVal}(v))\rangle \longrightarrow^* \langle \epsilon, \sigma_2\rangle$$

$$\frac{\mathrm{Val}_{\sigma_2} \neq \mathrm{BreakVal}}{\langle \texttt{switch(}e\texttt{)\{}B\texttt{\}}, \sigma\rangle \longrightarrow \langle \epsilon, \sigma_2(\mathrm{GLF{-}{-}})\rangle};$$

$$\langle e, \sigma\rangle \longrightarrow^* \langle \epsilon, \sigma_1\rangle \quad \mathrm{Val}_{\sigma_1} = (v, \tau) \quad IC(\tau, \texttt{int})$$

$$\langle B, \sigma_1(\mathrm{GLF{+}{+}})(\mathrm{Val} := \mathrm{CaseVal}(v))\rangle \longrightarrow^* \langle \epsilon, \sigma_2\rangle$$

$$\frac{\mathrm{Val}_{\sigma_2} = \mathrm{BreakVal}}{\langle \texttt{switch(}e\texttt{)\{}B\texttt{\}}, \sigma\rangle \longrightarrow \langle \epsilon, \sigma_2\rangle},$$

where $\sigma'_2 = \sigma(\mathrm{GLF{-}{-}})(\mathrm{Val} := \mathrm{OkVal})$.

The last rule restricts the range of the statement `break` if the latter has operated in the body of `switch` statement.

**Loops.** For modeling loops, the construct **loop**(*e*, *S*) is used, which works like the loop `while` but can additionally find exclusions generated by jump statements. The expression *e* is the condition of the loop, and *S* is its body. For this construct, we have six inference rules corresponding to different cases of the loop execution. One of them—the rule for the loop termination by means of a `break` statement—is given below:

$$\frac{\mathrm{Val}_\sigma = \mathrm{BreakVal}}{\langle \mathbf{loop}(e, S), \sigma\rangle \longrightarrow \langle \epsilon, \sigma(\mathrm{Val} := \mathrm{OkVal})\rangle}.$$

The transformation from ordinary loops to the **loop** construct is determined by the following axioms:

$$\langle \texttt{while(}e\texttt{)\{}S\texttt{\};}, \sigma\rangle \longrightarrow \langle \{\langle \mathbf{loop}\rangle(e, S)\}, \sigma\rangle;$$

$$\langle \texttt{do\{}S\texttt{\}while(}e\texttt{);}, \sigma\rangle \longrightarrow \langle \{S\mathbf{loop}(e, S)\}, \sigma\rangle;$$

$$\langle \texttt{for(}e_1\texttt{;}e_2\texttt{;}e_3\texttt{)\{}S\texttt{\};}, \sigma\rangle \longrightarrow \langle S', \sigma\rangle,$$

where

$$S' \equiv \{e_1\texttt{;if(!}e_2\texttt{) break;}S\mathbf{loop}((e_3,\ e_2), S)\}.$$

**Function** `main`. The execution of a program in C-light starts from control transfer to the function `main`, the semantics of which is determined by the axiom

$$\langle \texttt{int main(void)\{}S\texttt{\}}, \sigma\rangle \longrightarrow \langle \{S\}, \sigma\rangle.$$

## 8. CONCLUSIONS

The C-light language discussed in this paper forms the basis for the project aimed at the development of a method and verification system for C codes. C-light is a representative subset of C that is verification-oriented. Its advantages are the possibility to work with dynamic memory and sufficiently simple structural operational semantics. In addition, C-light is an extension of Pascal, for which the program verification system SPEKTR [12] was developed in 1991–1996.

For the verification of programs in C-light, a two-level scheme is suggested. On the first level, C-light is translated into its subset C-light-kernel. On the second level, correctness conditions are generated by means of the axiomatic semantics of C-light-kernel. The purpose of this translation is to simplify the axiomatic semantics. This method allows for also a formal justification of translation correctness and consistency of the axiomatic semantics of C-light-kernel with respect to its operational semantics.

The above two-level scheme has been already discussed in [10, 11, 13, 14]. The first version of the operational semantics of C-light is presented in [10], and the version described in this paper is discussed in detail in [11]. Although both versions are complete and correct, the latter is more advantageous in that it allows for formal justification of the translation correctness [11].

## ACKNOWLEDGMENTS

This work was supported in part by the Russian Foundation for Basic Research, project no. 00-01-00909.

## REFERENCES

1. Kernighan, B.W. and Ritchie, D., *The C Programming Language*, Englewood Cliffs, New Jersey: Prentice–Hall, 1978. Translated under the title *Yazyk programmirovaniya Ci*, Moscow: Finansy i statistika, 1985.
2. *ISO/IEC 9899:1999, Programming languages—C*, 1999.
3. Nepomniaschy, V.A. and Ryakin, O.M., *Prikladnye metody verifikatsii programm* (Applied Methods of Program Verification), Moscow: Radio i Svyaz', 1988.
4. Apt, K.R. and Olderog, E.R., *Verification of Sequential and Concurrent Programs*, Springer, 1991.
5. Gurevich, Y. and Huggings, J.K., The Semantics of the C Programming Language, *Proc. of the Int. Conf. on Comput. Sci. Logic*; *Lecture Notes Comput. Sci.*, 1993, vol. 702, pp. 274–309.
6. Huggins, J.K. and Shen, W., The Static and Dynamic Semantics of C (extended abstract), *Local Proc. of Int. Workshop on Abstract State Machines, ETH TIK-Rep.*, 2000, no. 87, pp. 272–284.
7. Plotkin, G.D., A Structure Approach to Operational Semantics, *Techn. Rep. FN-19*, Aarhus Univ., DAIMI, 1981.
8. Norrish, M., Deterministic Expressions in C, *Proc. of Europ. Symp. on Programming (ESOP99)*; *Lecture Notes Comput. Sci.*, 1999, vol. 1576, pp. 147–161.
9. Norrish, M., C Formalized in HOL, *PhD Dissertation*, Computer Lab., Univ. of Cambridge, 1998.
10. Nepomniaschy, V.A., Anureev, I.S., Mikhailov, I.N., and Promskii, A.V., Toward Verification of C Programs. Part 1: C-light Language, *Preprint of Inst. of Information Systems, Sib. Div., Russ. Acad. Sci.*, Novosibirsk, 2001, no. 84.
11. Nepomniaschy, V.A., Anureev, I.S., Mikhailov, I.N., and Promskii, A.V., Toward Verification of C Programs. Part 3: Translation from C-light to C-light-kernel and Its Formal Justification, *Preprint of Inst. of Information Systems, Sib. Div., Russ. Acad. Sci.*, Novosibirsk, 2002, no. 97.
12. Nepomniaschy, V.A. and Sulimov, A.A., Problem-Oriented Knowledge Bases and Their Use in the Program Verification System SPEKTR, *Izv. Ross. Akad. Nauk, Teor. Sist. Upr.*, 1997, no. 2, pp. 169–175.
13. Nepomniaschy, V.A., Anureev, I.S., Mikhailov, I.N., and Promskii, A.V., Toward Verification of C Programs: C-light Language, *Konferentsiya, posvyashchennaya 90-letiyu so dnya rozhdeniya A.A. Lyapunova* (Proc. of Conf. Devoted to the 90th Anniversary of the Birth of A.A. Lyapunov), Novosibirsk, 2001, pp. 423–432 (keynote paper).
14. Nepomniaschy, V.A., Anureev, I.S., Mikhailov, I.N., and Promskii, A.V., Toward Verification of C Programs. Part 2: C-light-kernel Language and Its Axiomatic Semantics, *Preprint of Inst. of Information Systems, Sib. Div., Russ. Acad. Sci.*, Novosibirsk, 2001, no. 87.