Bull. Nov. Comp. Center, Comp. Science, 28 (2008), 1–29
© 2008 NCC Publisher

A three-stage method of C program verification[*]

I. S. Anureev

Abstract. A three-stage method of C program verification is presented. It is a further development of the two-stage method in the framework of the C-light project. An additional stage of normalization of C-light programs is introduced and optimization of the two-stage method caused by this introduction is considered.

1. Introduction

The C-light project is being developed in the laboratory of theoretical programming of the Institute of Informatics Systems. In the framework of the C-light project, a two-stage method [1, 2, 3, 4] of C program verification has been developed. It is applied to a subset C-light of the C language that has a formal operational semantics [1, 2] and covers a major part of C. At the first stage, a C-light program is translated into an intermediate language C-kernel in order to eliminate some C-light constructs, difficult for axiomatic semantics, as well as to design axiomatic semantics in a more compact and transparent form [3]. At the second stage, verification conditions are generated by means of the rules of C-kernel axiomatic semantics [4].

A three-stage method of C program verification presented in this paper is a further development of the two-stage method. The idea of this development consists in partial carry of complexity of a program verification problem from deductive inference to static analysis by introduction of one more stage of normalization of C-light programs. Reduction of a C-light program to its normal form is performed by some kind of a static analysis. Application of the two-stage method to normalized C-light programs allows us to simplify the C-light machine and, consequently, to optimize the rules of operational semantics of C-light and axiomatic semantics of C-kernel.

The paper has the following structure. A survey of the C-light language is given in Section 2. Section 3 presents the abstract machine of C-light. Operational semantics of C-light is considered in Section 4. An overview of the C-kernel language is given in Section 5. An annotation language that is used for description of the properties of states of the C-light machine and for writing annotations in axiomatic semantics of C-kernel is described in Section 6. Axiomatic semantics of C-kernel is presented in Section 7.

[*]This research is partially supported by RFBR grant 06-01-00464-a.

2. Survey of the C-light language

The C-light language is a subset of the standard C99 [5], extended by typed memory management operators `new` and `delete`. Let us consider the restrictions imposed on this subset.

Types. Admissible types of C-light are as follows:

• **Base types**	
– integer:	_Bool,
	i ::= char, int, short [int], long [int], long long int,
	τ ::= signed i, unsigned i,
– real floating:	float, double, long double,
– empty:	void
• **Derived types**	pointers, arrays, structures, enumerations, functions

Thus complex types and unions are forbidden in C-light.

The following restrictions are imposed on admissible types:

1) The type char is always signed;
2) The values of pointers are noninterpreted constants;
3) Incomplete array types are only admitted as the function argument types;
4) Bit fields in structures are forbidden;
5) Functions with a variable number of arguments are forbidden;

Declarations. Let us consider the main distinctions of C-light declarations from C99 declarations. Tentative definitions are forbidden. Abstract declarations of arguments are forbidden. The use of a specifier static in dimension of an array that is a function parameter is forbidden. All specifiers and qualifiers of types, except for storage class specifiers and specifiers of a sign and size for scalar types, are forbidden.

Expressions. The order of expression evaluation is fixed in C-light: arguments of operators and functions are evaluated from right to left, initializer lists are evaluated from left to right. Compound literals are only allowed as initializers in declarations. A type cast for a compound initializer is forbidden. A pointer conversion is restricted by conversion from the type void$*$ to an arbitrary type $\tau*$.

To manage memory, special typed operators new and delete are used:

- new(τ) allocates memory for an object of a type τ and returns an address of the object;
- delete(c, τ) frees memory from an object of a type τ stored at an address c.

The C-light language has one semantic restriction. Access to the values stored at the addresses of program variables beyond their scopes is forbidden. A program for which this restriction is violated is considered incorrect. In most cases this restriction can be checked with the help of a static analysis. Similar context-sensitive conditions are used in the MISRA-C specification.

Statements. There are two restrictions on statements:

1. All case-labels and the label default in the switch statement are at the same level of nesting, i. e. the following variant is forbidden:

```
switch(i){
    case 1: if(a>0) {case 2: b = 3;}
            else {case 3: c = 0;}}
```

2. Jumping into a block by the goto statement is forbidden.

The source program. The source text of a program is a sequence of declarations. Preprocessor instructions are forbidden.

3. Abstract machine of the C-light language

Operational semantics of C-light defines execution of C-light programs in terms of states of an abstract machine called a C-light machine and transitions from one state to another. A description of the C-light machine includes:

- a type system that specifies the values handled by the C-light machine;
- variables of the C-light machine called metavariables that specify a state of the C-light machine;
- abstract functions that specify the base operations of the C-light machine.

3.1. Type system of the C-light machine

The C-light machine has the following system of types:

- base types:	CTypes, ID, Addrs, TypeSpecs, LogTypeSpecs
- functions:	$\tau \to \tau'$
- Cartesian products:	$\tau \times \tau'$

Here CTypes is a union of all admissible types of C-light considered in Section 2. ID is a set of identifiers of the C language. Addrs is a set of addresses of objects. $\mathsf{TypeSpecs}$ is a set of abstract type names [5]. $\mathsf{LogTypeSpecs}$ is a set of logical types that are logical representations of abstract type names. It is defined as follows:

- $\tau \in \mathsf{LogTypeSpecs}$, if τ is a base type;
- $\mathsf{pointer}(\tau) \in \mathsf{LogTypeSpecs}$, if $\tau \in \mathsf{LogTypeSpecs}$;
- $\tau_1 \times \ldots \times \tau_n \to \tau \in \mathsf{LogTypeSpecs}$, if $\tau_1, \ldots, \tau_n, \tau \in \mathsf{LogTypeSpecs}$;
- $\mathsf{struct}(\mathsf{s}, (\tau_1, \mathsf{m}_1), \ldots, (\tau_n, \mathsf{m}_n)) \in \mathsf{LogTypeSpecs}$, if $\tau_1, \ldots, \tau_n \in \mathsf{LogTypeSpecs}$ and $\mathsf{s}, \mathsf{m}_1, \ldots, \mathsf{m}_n \in \mathsf{ID}$;
- $\mathsf{enum}(\mathsf{s}, (\mathsf{m}_1, \mathsf{c}_1) \ldots, (\mathsf{m}_n\, \mathsf{c}_n)) \in \mathsf{LogTypeSpecs}$, if $\mathsf{s}, \mathsf{m}_1, \ldots, \mathsf{m}_n \in \mathsf{ID}$ and $\mathsf{c}_1, \ldots, \mathsf{c}_n \in \mathsf{CTypes}$;
- $\mathsf{array}(\tau, \mathsf{n}) \in \mathsf{LogTypeSpecs}$, if $\tau \in \mathsf{LogTypeSpecs}$, n is a positive integer.

3.2. Metavariables and states of the C-light machine

A notation $\phi(\mathsf{c}) = \perp$ is used to denote that an element c does not belong to a partial function ϕ. A state of the C-light machine is defined by the following variables called metavariables:

1) MD *is a variable of the type* $\mathsf{Addrs} \to \mathsf{CTypes}$ *such that* $\mathsf{MD}(\mathsf{c})$ *returns the value stored at the address* c*;*
2) Val *is a variable of the type* CTypes *that specifies the value returned by a function or an expression.*

The *state of the C-light machine* is a function from metavariables to their values. Let States be the set of all states. Greek letters σ, τ, possibly with indexes, are used to denote states. Let $\mathsf{MD}_\sigma(\mathsf{c})$ be a short for $(\sigma(\mathsf{MD}))(\mathsf{c})$.

3.3. The normal form of C-light programs

The C-light machine processes C-light programs in the normal form. A C-light program is in the normal form if the following conditions hold:

1. The same identifier cannot denote different entities at different points in the program.
2. There are no declarations of structures and enumerations without tags.
3. Variable declarations are in the normal form.
4. Additional information is provided for C-light consructs. This information called a local environment of a construct precedes the construct and has the form $/ * * \, \mathsf{p}_1(\bar{\mathsf{e}}_1) \ldots \mathsf{p}_n(\bar{\mathsf{e}}_n) \, * */$, where p_i are predicates, and $\bar{\mathsf{e}}_i$ are values for which p_i are true.

Normalization of a program is performed as follows. Condition 1 is provided by renaming of occurences of identifiers in the program. Condition 2 is provided by addition of new tags.

A declaration $\mathsf{e} = \mathsf{e}'$; is in the normal form if the initializer e' is in a fully bracketed form and does not contain designators, and the object declared in e has a complete type. For instance, declarations

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

and

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

contain equivalent initializers, but only the first of them is in the normal form. A declaration $\mathsf{d} = \mathsf{d}'$; is the normal form of a declaration $\mathsf{e} = \mathsf{e}'$; if

1) $\mathsf{d} = \mathsf{d}'$; is in the normal form;
2) These declarations initialize the same object by the same value.

In accordance with [5], any declaration can be reduced to the normal form.

Information used in local environments of constructs of the source program is collected by static analysis. Let us list predicates used in local environments. Let e be a construct for which a local environment is defined:

- If e is an enumeration constant, then $\mathsf{enumctype}(\tau)$ and $\mathsf{enumcval}(\mathsf{v})$ mean that e has the type τ and the value v, respectively.
- If e is a cast expression or a new operator, then $\mathsf{logtype}(\tau)$ means that the abstract type name in e is associated with the logical type τ.
- If e is a call of a function, then $\mathsf{autovar}(\mathsf{y}_1, \ldots, \mathsf{y}_\mathsf{m})$ means that $\mathsf{y}_1, \ldots, \mathsf{y}_\mathsf{m}$ is the list of all automatic variables and parameters of the function.
- If e is a declaration of a variable, then $\mathsf{logtype}(\tau)$, $\mathsf{storage}(\mathsf{st})$ and $\mathsf{name}(\mathsf{x})$ mean that the variable has the logical type τ, the storage class st and the name x, respectively.
- If e is a declaration of a function, then $\mathsf{logtype}(\tau_1 \times \ldots \times \tau_\mathsf{n} \to \tau)$ means that e has the type $\tau_1 \times \ldots \times \tau_\mathsf{n} \to \tau$.
- If e is a case label, then $\mathsf{caseval}(\mathsf{v})$ means that v is the value of the label e.
- If e is a return statement occuring in the body of a function, then $\mathsf{name}(\mathsf{f})$, $\mathsf{rettype}(\tau)$, $\mathsf{autovar}(\mathsf{x}_1, \ldots, \mathsf{x}_\mathsf{m})$, $\mathsf{autovarval}(\mathsf{a}_1, \ldots, \mathsf{a}_\mathsf{m})$ mean that the function has the name f and the return type τ, and $\mathsf{x}_1, \ldots, \mathsf{x}_\mathsf{m}$ is the list of all automatic variables and parameters of the function with values designated by the logical variables $\mathsf{a}_1, \ldots, \mathsf{a}_\mathsf{m}$.

- If e is a program, then $\mathsf{prgvar}(\mathsf{y}_1, \ldots, \mathsf{y}_\mathsf{k})$ and $\mathsf{mainpar}(\mathsf{z}_1, \ldots, \mathsf{z}_\mathsf{l})$ mean that $\mathsf{y}_1, \ldots, \mathsf{y}_\mathsf{k}$ is the list of all variables and function parameters of the program e, except the parameters $\mathsf{z}_1, \ldots, \mathsf{z}_\mathsf{l}$ of the main function.

3.4. Abstract functions

The operational definition of the C-light machine uses special abstract functions which are listed below.

To refer to separate elements of aggregate types (arrays and structures), access to addresses of these elements is required. The function mb specifies access to addresses of elements of arrays and fields of structures. If c is the address of an array, the function $\mathsf{mb}(\mathsf{c}, \mathsf{l})$ returns the address of the array element with the index l. If c is the address of a structure, the function $\mathsf{mb}(\mathsf{c}, \mathsf{l})$ returns the address of the field l of the structure. Let us notice that the address of an array coincides with the address of the first element of this array. Thus, $\mathsf{mb}(\mathsf{c}, 0) = \mathsf{c}$ for each array with the address c.

The function $\mathsf{naddr}(\mathsf{MD})$ returns a new address in accordance with the current value of the metavariable MD. The metavariable MD specifies the space of new addresses as a set of all addresses c such that $\mathsf{MD}(\mathsf{c}) =\perp$. Thus, $\mathsf{MD}(\mathsf{naddr}(\mathsf{MD})) =\perp$. The function $\mathsf{std}(\mathsf{x})$ returns the type associated with the identifier x that denotes either a typedef name, or a tag of a structure, or enumeration. The function $\mathsf{retType}(\tau)$ returns the type τ_0, if the logical type τ has the form $\tau_1 \times \ldots \times \tau_\mathsf{n} \rightarrow \tau_0$, and τ otherwise.

C-light inherits the C operators. The functions $\mathsf{UnOpSem}$ and $\mathsf{BinOpSem}$ are used to define computations of side-effect free operators in a symbolic way without considering concrete implementations of C. The function $\mathsf{UnOpSem}(\odot, \mathsf{v}, \tau)$ returns the result of applying the unary operator $\odot$ to the value v of the type τ. The function $\mathsf{BinOpSem}(\odot, \mathsf{v}_1, \tau_1, \mathsf{v}_2, \tau_2)$ returns the result of applying the binary operator $\odot$ to the values v_1 and v_2 of the types τ_1 and τ_2, respectively.

The function $\mathsf{cast}(\mathsf{e}, \tau, \tau')$ converts the value e from the type τ to the type τ'. The function $\mathsf{labels}(\mathsf{S})$ returns a set of goto labels occuring in the sequence of statements S at the high level. The function $\mathsf{defaultValue}(\tau, \mathsf{storage})$ returns the default value of the type τ in accordance with the storage class $\mathsf{storage}$:

- $\mathsf{defaultValue}(\tau, \mathsf{static}) =$ the default value of the type τ,
- $\mathsf{defaultValue}(\tau, \mathsf{auto}) = \omega$.

The function $\mathsf{TypeOfNewVal}(\tau)$ returns the type of a storage unit allocated by the operator new for the aggregate type τ:

- $\mathsf{TypeOfNewVal}(\mathsf{array}(\tau, \mathsf{k})) = \tau$,
- $\mathsf{TypeOfNewVal}(\tau) = \tau$, if τ is a structure.

The function $\mathsf{findex}(\tau)$ returns the first index of the aggregate type τ:

- $\mathsf{findex}(\tau) = 0$, if τ is an array,
- $\mathsf{findex}(\mathsf{struct}(\mathsf{s}, (\tau_1, \mathsf{l}_1), \ldots, (\tau_k, \mathsf{l}_k))) = \mathsf{l}_1$.

The function $\mathsf{next}(\tau, \mathsf{l})$ returns the index of the aggregate type τ that directly follows l:

- $\mathsf{next}(\mathsf{array}(\tau, \mathsf{k}), \mathsf{l}) = \mathsf{l} + 1$, if $\mathsf{l} + 1 < \mathsf{k}$,
- $\mathsf{next}(\mathsf{array}(\tau, \mathsf{k}), \mathsf{l}) = \omega$, if $\mathsf{l} + 1 \geq \mathsf{k}$,
- $\mathsf{next}(\mathsf{struct}(\mathsf{s}, (\tau_1, \mathsf{l}_1), \ldots, (\tau_k, \mathsf{l}_k)), \mathsf{l}_i) = \mathsf{l}_{i+1}$, if $\mathsf{i} < \mathsf{k}$,
- $\mathsf{next}(\mathsf{struct}(\mathsf{s}, (\tau_1, \mathsf{l}_1), \ldots, (\tau_k, \mathsf{l}_k)), \mathsf{l}_i) = \omega$, if $\mathsf{i} \geq \mathsf{k}$.

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

- $\mathsf{itype}(\mathsf{array}(\tau, \mathsf{k}), \mathsf{i}) = \tau$,
- $\mathsf{itype}(\mathsf{struct}(\mathsf{s}, (\tau_1, \mathsf{l}_1), \ldots, (\tau_k, \mathsf{l}_k)), \mathsf{i}) = \tau_i$.

The function $\mathsf{ctype}(\mathsf{u})$ returns the type of the constant u. The function $\mathsf{optype}(\odot, \tau_1, \tau_2)$ returns the type of the value returned by the operator $\odot$ in the case that the operator has arguments of types τ_1 and τ_2. The optional argument τ_2 is needed for binary operators and a conditional operator (for which the types of the second and third arguments are only considered). Semantics of this function is defined in accordance with [5]. The function $\mathsf{vtype}(\mathsf{x})$ returns the type of the identifier x. The function $\mathsf{type}(\mathsf{e})$ returns the type of the expression e:

- $\mathsf{type}(\mathsf{e}) = \mathsf{ctype}(\mathsf{e})$, if e is a constant,
- $\mathsf{type}(\mathsf{e}) = \mathsf{vtype}(\mathsf{e})$, if e is a variable,
- $\mathsf{type}(/**\mathsf{enumctype}(\tau)\ **/\quad \mathsf{e}, \mathsf{MD}) = \tau$, if e is an enumeration constant,
- $\mathsf{type}(\odot\, \mathsf{e}) = \mathsf{optype}(\odot, \mathsf{type}(\mathsf{e}))$,
- $\mathsf{type}(\mathsf{e}_1 \odot \mathsf{e}_2) = \mathsf{optype}(\odot, \mathsf{type}(\mathsf{e}_1), \mathsf{type}(\mathsf{e}_2))$,
- $\mathsf{type}(\mathsf{e}_1\, ?\, \mathsf{e}_2 : \mathsf{e}_3) = \mathsf{optype}(?:, \mathsf{type}(\mathsf{e}_2), \mathsf{type}(\mathsf{e}_3))$,
- $\mathsf{type}((\mathsf{e})) = \mathsf{type}(\mathsf{e})$,
- $\mathsf{type}(\mathsf{e}(\mathsf{e}_1, \ldots, \mathsf{e}_n)) = \tau$, if $\mathsf{type}(\mathsf{e}) = \tau_1 \times \ldots \times \tau_n \to \tau$.

The function $\mathsf{val}(\mathsf{e}, \mathsf{MD})$ returns the value of the expression e in accordance with the value of the metavariable MD:

- $\mathsf{val}(\mathsf{e}, \mathsf{MD}) = \mathsf{MD}(\mathsf{e})$, if e is a variable,

- $\mathsf{val}(\mathsf{e},\ \mathsf{MD}) = \mathsf{e}$, if e is a constant,
- $\mathsf{val}(/**\mathsf{enumcval}(\mathsf{v})\ **/\quad \mathsf{e},\ \mathsf{MD}) = \mathsf{v}$,
  if e is an enumeration constant,
- $\mathsf{val}(\mathsf{e}[\mathsf{e}'],\ \mathsf{MD}) = \mathsf{MD}(\mathsf{mb}(\mathsf{val}(\mathsf{e},\ \mathsf{MD}),\ \mathsf{val}(\mathsf{e}',\ \mathsf{MD})))$,
- $\mathsf{val}(\mathsf{e.m},\ \mathsf{MD}) = \mathsf{MD}(\mathsf{mb}(\mathsf{val}(\mathsf{e},\ \mathsf{MD}),\ \mathsf{m}))$,
- $\mathsf{val}(\&\mathsf{e},\ \mathsf{MD}) = \mathsf{addr}(\mathsf{e},\ \mathsf{MD})$,
- $\mathsf{val}(*\mathsf{e},\ \mathsf{MD}) = \mathsf{MD}(\mathsf{val}(\mathsf{e},\ \mathsf{MD}))$,
- $\mathsf{val}(/**\mathsf{logtype}(\tau)\ **/\quad (\tau')\,\mathsf{e},\ \mathsf{MD}) = \mathsf{cast}(\mathsf{val}(\mathsf{e},\ \mathsf{MD}),\ \mathsf{type}(\mathsf{e}),\ \tau)$,
- $\mathsf{val}(\odot\,\mathsf{e},\ \mathsf{MD}) = \mathsf{UnOpSem}(\odot,\ \mathsf{val}(\mathsf{e},\ \mathsf{MD}),\ \mathsf{type}(\mathsf{e}))$,
- $\mathsf{val}(\mathsf{e}\ \odot\ \mathsf{e}',\ \mathsf{MD}) =$
  $\mathsf{BinOpSem}(\odot,\ \mathsf{val}(\mathsf{e},\ \mathsf{MD}),\ \mathsf{type}(\mathsf{e}),\ \mathsf{val}(\mathsf{e}',\ \mathsf{MD}),\ \mathsf{type}(\mathsf{e}'))$.

The function $\mathsf{addr}(\mathsf{e},\ \mathsf{MD})$ returns the address of a storage unit in which the value of the expression e is stored:

- $\mathsf{addr}(\mathsf{e},\ \mathsf{MD}) = \mathsf{e}$, if e is a variable,
- $\mathsf{addr}(\mathsf{e},\ \mathsf{MD}) =\perp$, if e is a constant,
- $\mathsf{addr}(\mathsf{e}[\mathsf{e}'],\ \mathsf{MD}) = \mathsf{mb}(\mathsf{val}(\mathsf{e},\ \mathsf{MD}),\ \mathsf{val}(\mathsf{e}',\ \mathsf{MD}))$,
- $\mathsf{addr}(\mathsf{e.m},\ \mathsf{MD}) = \mathsf{mb}(\mathsf{val}(\mathsf{e},\ \mathsf{MD}),\ \mathsf{m})$,
- $\mathsf{addr}(\&\mathsf{e},\ \mathsf{MD}) =\perp$,
- $\mathsf{addr}(*\mathsf{e},\ \mathsf{MD}) = \mathsf{val}(\mathsf{e},\ \mathsf{MD})$,
- $\mathsf{addr}((\tau)\,\mathsf{e},\ \mathsf{MD}) =\perp$,
- $\mathsf{addr}(\odot\,\mathsf{e},\ \mathsf{MD}) =\perp$,
- $\mathsf{addr}(\mathsf{e}\ \odot\ \mathsf{e}',\ \mathsf{MD}) =\perp$.

Let τ_1 be not an aggregate type and τ_2 be an aggregate type. The function $\mathsf{init}(\tau,\ \mathsf{e},\ \mathsf{MD})$ initializes a storage unit of the type τ in accordance with the initializer specifier e and the value of the metavariable MD, modifying the metavariable MD and returning the initialized value. The initializer specifier e can be either an initializer, or an initializer with evaluated elements that have the form $(\mathsf{v},\ \tau)$, where v is the value of the element, and τ is the type of the element, or a storage class (if there is no initializier in the declaration that defines the initialized object). In the latter case initialization is performed by default. The function init is defined as follows:

- $\mathsf{init}(\tau_1,\ \mathsf{storage},\ \mathsf{MD}) = (\mathsf{MD},\ \mathsf{defaultValue}(\tau_1,\ \mathsf{storage}))$,
- $\mathsf{init}(\tau_1,\ \mathsf{e},\ \mathsf{MD}) = (\mathsf{MD},\ \mathsf{cast}(\mathsf{val}(\mathsf{e},\ \mathsf{MD}),\ \mathsf{type}(\mathsf{e}),\ \tau_1))$,
- $\mathsf{init}(\tau_1,\ (\mathsf{v},\ \tau),\ \mathsf{MD}) = (\mathsf{MD},\ \mathsf{cast}(\mathsf{v},\ \tau,\ \tau_1))$,

- $\mathsf{init}(\tau_2,\ \mathsf{e},\ \mathsf{MD}) = (\mathsf{updv}(\mathsf{MD},\ \mathsf{nc},\ \tau_2,\ \mathsf{e}),\ \mathsf{nc})$, where $\mathsf{nc} = \mathsf{naddr}(\mathsf{MD})$.

The function $\mathsf{updv}(\mathsf{MD},\ \mathsf{nc},\ \tau,\ \mathsf{e})$, modifying the metavariable MD, specifies allocation and initialization of storage units starting with the address *nc* for an object of the aggregate type τ with the initializer specifier e:

- $\mathsf{updv}(\mathsf{MD},\ \mathsf{nc},\ \tau,\ \mathsf{e}) = \mathsf{updv}(\mathsf{MD},\ \mathsf{nc},\ \tau,\ \mathsf{e},\ \mathsf{findex}(\tau))$,
- $\mathsf{updv}(\mathsf{MD},\ \mathsf{nc},\ \tau,\ \mathsf{storage},\ \mathsf{l}) =$
  $\mathsf{updv}(\mathsf{upd}(\mathsf{MD}',\ \mathsf{mb}(\mathsf{nc},\ \mathsf{l}),\ \mathsf{Val}'),\ \mathsf{nc},\ \tau,\ \mathsf{storage},\ \mathsf{next}(\mathsf{l}))$,
  if $\mathsf{l} \neq \omega$ and $(\mathsf{MD}',\ \mathsf{Val}') = \mathsf{init}(\mathsf{itype}(\tau,\ \mathsf{l}),\ \mathsf{storage},\ \mathsf{MD})$,
- $\mathsf{updv}(\mathsf{MD},\ \mathsf{nc},\ \tau,\ \{\mathsf{e}_1,\ \mathsf{e}_2,\ \ldots,\ \mathsf{e}_\mathsf{k}\},\ \mathsf{l}) =$
  $\mathsf{updv}(\mathsf{upd}(\mathsf{MD}',\ \mathsf{mb}(\mathsf{nc},\ \mathsf{l}),\ \mathsf{Val}'),\ \mathsf{nc},\ \tau,\ \{\mathsf{e}_2,\ \ldots,\ \mathsf{e}_\mathsf{k}\},\ \mathsf{next}(\mathsf{l}))$,
  if $\mathsf{l} \neq \omega$ and $(\mathsf{MD}',\ \mathsf{Val}') = \mathsf{init}(\mathsf{itype}(\tau,\ \mathsf{l}),\ \mathsf{e}_1,\ \mathsf{MD})$,
- $\mathsf{updv}(\mathsf{MD},\ \mathsf{nc},\ \tau,\ \{\},\ \mathsf{l}) =$
  $\mathsf{updv}(\mathsf{upd}(\mathsf{MD}',\ \mathsf{mb}(\mathsf{nc},\ \mathsf{l}),\ \mathsf{Val}'),\ \mathsf{nc},\ \tau,\ \{\},\ \mathsf{next}(\mathsf{l}))$,
  if $\mathsf{l} \neq \omega$ and $(\mathsf{MD}',\ \mathsf{Val}') = \mathsf{init}(\mathsf{itype}(\tau,\ \mathsf{l}),\ \mathsf{static},\ \mathsf{MD})$,
- $\mathsf{updv}(\mathsf{MD},\ \mathsf{nc},\ \tau,\ \mathsf{e},\ \omega) = \mathsf{MD}$.

Let τ_1 be not an aggregate type and τ_2 be an aggregate type. The function $\mathsf{new}(\tau,\ \mathsf{MD})$ allocates a storage unit for an object of the type τ, modifying the metavariable MD and returning the address of the storage unit:

- $\mathsf{new}(\tau_1,\ \mathsf{MD}) = (\mathsf{upd}(\mathsf{MD},\ \mathsf{nc},\ \omega),\ \mathsf{nc})$, where $\mathsf{nc} = \mathsf{naddr}(\mathsf{MD})$,
- $\mathsf{new}(\tau_2,\ \mathsf{MD}) = \mathsf{init}(\tau_2,\ \mathsf{auto},\ \mathsf{MD})$.

The function $\mathsf{delete}(\mathsf{MD},\ \mathsf{nc},\ \tau)$ releases memory, starting with the storage unit at the address nc that stores an object of the type τ:

- $\mathsf{delete}(\mathsf{MD},\ \mathsf{nc},\ \tau) = \mathsf{delete}(\mathsf{upd}(\mathsf{MD}\,\mathsf{nc},\ \bot),\ \mathsf{nc},\ \tau,\ \mathsf{findex}(\tau'))$,
  if τ is an aggregate type,
- $\mathsf{delete}(\mathsf{MD},\ \mathsf{nc},\ \tau) = \mathsf{upd}(\mathsf{MD},\ \mathsf{nc},\ \bot)$, if τ is not an aggregate type,
- $\mathsf{delete}(\mathsf{MD},\ \mathsf{nc},\ \tau,\ \mathsf{l}) = \mathsf{delete}(\mathsf{delete}(\mathsf{MD},\ ),\ \mathsf{nc},\ \tau,\ \mathsf{next}(\tau,\ \mathsf{l}))$, if $\mathsf{l} \neq \omega$,
- $\mathsf{delete}(\mathsf{MD},\ \mathsf{nc},\ \tau,\ \omega) = \mathsf{MD}$.

The optional fourth argument of the function delete designates an index of an array or a field of a structure.

3.5. Configurations of C-light machine

Operational semantics of a programming language is defined as a set of pairs of configurations of its abstract machine specified by the transition relation. Definition of a configuration can vary depending on the programming language. In our case the standard notion of a configuration is used, since all features of the programming language and computations are «incapsulated» in states.

Definition. A *configuration of C-light machine is a pair* $\langle \mathsf{P}, \sigma \rangle$*, where* P *is a program and* σ *is a state.*

An axiom of operational semantics has the form $\langle \mathsf{A}, \sigma \rangle \to \langle \mathsf{B}, \sigma' \rangle$. This means that one step of execution of the program fragment A starting in the state σ leads to the state σ' and B is the program fragment that remains to execute. A rule of operational semantics has the form

$$\frac{\mathsf{P}_1 \quad \ldots \quad \mathsf{P}_\mathsf{n}}{\langle \mathsf{A}, \sigma \rangle \to \langle \mathsf{B}, \sigma' \rangle}.$$

This means that if the conditions $\mathsf{P}_1, \ldots, \mathsf{P}_\mathsf{n}$ are fulfilled, there is a transition from the configuration $\langle A, \sigma \rangle$ to the configuration $\langle B, \sigma' \rangle$.

Thus, execution of programs in operational semantics is specified by (possibly infinite) sequences of configurations:

$$\langle \mathsf{S}_1, \sigma_1 \rangle \to \langle \mathsf{S}_2, \sigma_2 \rangle \to \ldots \to \langle \mathsf{S}_\mathsf{i}, \sigma_\mathsf{i} \rangle \to \ldots .$$

The configuration $\langle \mathsf{S}_1, \sigma_1 \rangle$ is called *initial.* If the sequence is finite and $\langle \mathsf{S}_\mathsf{n}, \sigma_\mathsf{n} \rangle$ is the last configuration in this sequence, it is said that execution of the fragment S_1 starting in the state σ_1 leads to the *final* configuration $\langle \mathsf{S}_\mathsf{n}, \sigma_\mathsf{n} \rangle$. Let $\to^*$ denote a transitive reflexive closure of the relation $\to$. Then the above execution can be denoted by $\langle \mathsf{S}_1, \sigma_1 \rangle \to^* \langle \mathsf{S}_\mathsf{n}, \sigma_\mathsf{n} \rangle$.

Let ϵ denote an empty program fragment. An empty fragment can be either an empty program or an empty expression.

4. Operational semantics

Operational semantics models execution of programs in an abstract machine. The rules of operational semantics of C-light are classified according to three main groups of C-light constructs — expressions, declarations and statements. Beyond that point there is a special group which includes constructs that does not belong to the mentioned groups (for example, a sequence of statements).

4.1. Semantics of expressions

A function call. Semantics of a function call is defined by three rules. The first rule specifies evaluation of function arguments. The second rule specifies execuiton of a function returning a value. The third rule specifies execution of a function that returns no value. The intermediate state after evaluation of function arguments is given by an auxiliary construct FCall.

The first rule has the form:

$$\frac{\begin{array}{c}\langle \mathsf{e}_0,\, \sigma\rangle \to^* \langle \mathsf{v}_0,\, \sigma_0\rangle \qquad \mathsf{type}(\mathsf{e}_0) = \tau_1 \times \ldots \times \tau_\mathsf{n} \to \tau \\ \langle \mathsf{e}_\mathsf{n},\, \sigma_0\rangle \to^* \langle \mathsf{v}_\mathsf{n},\, \sigma_1\rangle \ldots \langle \mathsf{e}_1,\, \sigma_{\mathsf{n}-1}\rangle \to^* \langle \mathsf{v}_1,\, \sigma_\mathsf{n}\rangle\end{array}}{\begin{array}{l}\langle /**\mathsf{autovar}(\mathsf{y}_1,\, \ldots,\, \mathsf{y}_\mathsf{m})\ **/\quad \mathsf{e}_0(\mathsf{e}_1,\, \ldots,\, \mathsf{e}_\mathsf{n}),\, \sigma\rangle \to \\ \langle /**\mathsf{autovar}(\mathsf{y}_1,\, \ldots,\, \mathsf{y}_\mathsf{m})\ **/\quad \mathsf{FCall}(\mathsf{fst}(\mathsf{v}_0))(\\ \quad \mathsf{cast}(\mathsf{fst}(\mathsf{v}_1),\, \mathsf{type}(\mathsf{e}_1),\, \tau_1),\, \ldots,\, \mathsf{cast}(\mathsf{fst}(\mathsf{v}_\mathsf{n}),\, \mathsf{type}(\mathsf{e}_\mathsf{n}),\, \tau_\mathsf{n})),\, \sigma_\mathsf{n}\rangle\end{array}}$$

Semantics of the auxiliary construct FCall is defined by two rules depending on whether the function returns a value or not.

Let FCallPremise denote a premise

$$\begin{array}{l}\mathsf{MD}_\sigma(\mathsf{f}) = (\tau,\, \tau_1 \times \ldots \times \tau_\mathsf{n},\, [\mathsf{x}_1,\, \ldots,\, \mathsf{x}_\mathsf{n}],\, \mathsf{S}) \wedge \\ \mathsf{MD}_\sigma(\mathsf{y}_1) = \mathsf{a}_1 \wedge \ldots \wedge \mathsf{MD}_\sigma(\mathsf{y}_\mathsf{m}) = \mathsf{a}_\mathsf{m} \wedge \\ \mathsf{MD}' = \mathsf{upd}(\mathsf{MD}_\sigma,\, (\mathsf{x}_1,\, \ldots,\, \mathsf{x}_\mathsf{n}),\, (\mathsf{v}_1,\, \ldots,\, \mathsf{v}_\mathsf{n}))\end{array}$$

The rule for a function returning a value has the form:

$$\frac{\mathsf{FCallPremise} \qquad \langle \mathsf{S} \quad \mathsf{returnStop},\, \sigma(\mathsf{MD} \leftarrow \mathsf{MD}')\rangle \to^* \langle \mathsf{v},\, \sigma'\rangle}{\begin{array}{l}\langle /**\mathsf{autovar}(\mathsf{y}_1,\, \ldots,\, \mathsf{y}_\mathsf{m})\ **/\quad \mathsf{FCall}(\mathsf{f})(\mathsf{v}_1,\, \ldots,\, \mathsf{v}_\mathsf{n}),\, \sigma\rangle \to \\ \langle \mathsf{v},\, \sigma'(\mathsf{MD} \leftarrow \mathsf{upd}(\mathsf{MD},\, (\mathsf{y}_1,\, \ldots,\, \mathsf{y}_\mathsf{m}),\, (\mathsf{a}_1,\, \ldots,\, \mathsf{a}_\mathsf{m})))\rangle\end{array}}$$

An auxiliary construct returnStop catches exceptions Exc(returnStart(e)) and Exc(returnStart), thrown by a statement return with an argument and without it, respectively:

$$\langle \mathsf{returnStop},\, \sigma\rangle \to \langle \epsilon,\, \sigma\rangle$$

$$\langle \mathsf{Exc}(\mathsf{returnStart}) \quad \mathsf{returnStop},\, \sigma\rangle \to \langle \epsilon,\, \sigma\rangle$$

$$\langle \mathsf{Exc}(\mathsf{returnStart}(\mathsf{v})) \quad \mathsf{returnStop},\, \sigma\rangle \to \langle (\mathsf{v},\, \bot),\, \sigma\rangle$$

The rule for a function that returns no value has the form:

$$\frac{\mathsf{FCallPremise} \qquad \langle \mathsf{S} \quad \mathsf{returnStop},\, \sigma(\mathsf{MD} \leftarrow \mathsf{MD}')\rangle \to^* \langle \epsilon,\, \sigma'\rangle}{\begin{array}{l}\langle /**\mathsf{autovar}(\mathsf{y}_1,\, \ldots,\, \mathsf{y}_\mathsf{m})\ **/\quad \mathsf{FCall}(\mathsf{f})(\mathsf{v}_1,\, \ldots,\, \mathsf{v}_\mathsf{n}),\, \sigma\rangle \to \\ \langle \epsilon,\, \sigma'(\mathsf{MD} \leftarrow \mathsf{upd}(\mathsf{MD},\, (\mathsf{y}_1,\, \ldots,\, \mathsf{y}_\mathsf{m}),\, (\mathsf{a}_1,\, \ldots,\, \mathsf{a}_\mathsf{m})))\rangle\end{array}}$$

Old values of the automatic variables $\mathsf{y}_1,\, \ldots,\, \mathsf{y}_\mathsf{m}$ of a function with the address f, including the parameters of the function, are restored after exiting the function.

Memory allocation operators. The new operator allocates one storage unit for scalar types and a set of storage units for aggregate types:

$$\frac{(\mathsf{MD}',\, \mathsf{V}) = \mathsf{new}(\tau,\, \mathsf{MD}_\sigma)}{\langle /**\mathsf{logtype}(\tau)\ **/\quad \mathsf{new}\ \mathsf{e},\, \sigma\rangle \to \langle \mathsf{V},\, \sigma(\mathsf{MD} \leftarrow \mathsf{MD}')\rangle}$$

A rule for the delete operator has the form:

$$\frac{\langle \mathsf{e},\, \sigma\rangle \to^* \langle \mathsf{v},\, \sigma'\rangle \quad \mathsf{MD}' = \mathsf{delete}(\mathsf{MD}_{\sigma'},\, \mathsf{fst}(\mathsf{v}),\, \tau)}{\langle /*\ \mathsf{logtype}(\tau)\ */\quad \mathsf{delete}\ (\mathsf{e},\, \tau'),\, \sigma\rangle \to \langle \epsilon,\, \sigma'(\mathsf{MD} \leftarrow \mathsf{MD}')\rangle}$$

The delete operator releases the memory allocated for a data structure of the type τ, starting with the address $\mathsf{val}(\mathsf{v})$.

Assignment operators. Rules for a simple assignment have the form:

$$\frac{\begin{array}{l}\langle \mathsf{e}_2,\, \sigma\rangle \to^* \langle \mathsf{v}_2,\, \sigma'\rangle \qquad \langle \mathsf{e}_1,\, \sigma'\rangle \to^* \langle \mathsf{v}_1,\, \sigma''\rangle \\ \mathsf{v} = \mathsf{cast}(\mathsf{fst}(\mathsf{v}_2),\, \mathsf{type}(\mathsf{e}_2),\, \mathsf{type}(\mathsf{e}_1)) \qquad \mathsf{type}(\mathsf{e}_1) \text{ is not a structure type}\end{array}}{\langle \mathsf{e}_1 = \mathsf{e}_2,\, \sigma\rangle \to \langle \mathsf{v},\, \sigma''(\mathsf{MD} \leftarrow \mathsf{upd}(\mathsf{MD},\, \mathsf{snd}(\mathsf{v}_1),\, \mathsf{v}))\rangle}$$

$$\frac{\begin{array}{l}\langle \mathsf{e}_2,\, \sigma\rangle \to^* \langle \mathsf{v}_2,\, \sigma'\rangle \qquad \langle \mathsf{e}_1,\, \sigma'\rangle \to^* \langle \mathsf{v}_1,\, \sigma''\rangle \\ \mathsf{type}(\mathsf{e}_1) \text{ is a structure type with fields } \mathsf{l}_1,\, \ldots,\, \mathsf{l}_\mathsf{m}\end{array}}{\begin{array}{l}\langle \mathsf{e}_1 = \mathsf{e}_2,\, \sigma\rangle \to \\ \langle \mathsf{v},\, \sigma''(\mathsf{MD} \leftarrow \mathsf{upd}(\mathsf{MD},\, (\mathsf{mb}(\mathsf{v}_1,\, \mathsf{l}_1),\, \ldots,\, \mathsf{mb}(\mathsf{v}_1,\, \mathsf{l}_\mathsf{m})), \\ \qquad (\mathsf{MB}(\mathsf{mb}(\mathsf{v}_2,\, \mathsf{l}_1)),\, \ldots,\, \mathsf{MB}(\mathsf{mb}(\mathsf{v}_2,\, \mathsf{l}_\mathsf{m})))))\rangle\end{array}}$$

A rule for a compound assignment has the form:

$$\frac{\begin{array}{l}\langle \mathsf{e}_2,\, \sigma\rangle \to^* \langle \mathsf{v}',\, \sigma'\rangle \qquad \langle \mathsf{e}_1,\, \sigma'\rangle \to^* \langle \mathsf{v}'',\, \sigma''\rangle \\ \mathsf{v} = \mathsf{cast}(\mathsf{BinOpSem}(\odot,\, \mathsf{fst}(\mathsf{v}''),\, \mathsf{type}(\mathsf{e}_1),\, \mathsf{fst}(\mathsf{v}'),\, \mathsf{type}(\mathsf{e}_2)), \\ \qquad \mathsf{type}(\odot,\, \mathsf{type}(\mathsf{e}_1),\, \mathsf{type}(\mathsf{e}_2)),\, \tau'')\end{array}}{\langle \mathsf{e}_1\odot = \mathsf{e}_2,\, \sigma\rangle \to \langle \mathsf{v},\, \sigma''(\mathsf{MD} \leftarrow \mathsf{upd}(\mathsf{MD},\, \mathsf{snd}(\mathsf{v}''),\, \mathsf{v}))\rangle}$$

Variables and constants. Semantics of evaluation of variables and constants is defined by the axiom:

$$\langle \mathsf{x},\, \sigma\rangle \to \langle (\mathsf{val}(\mathsf{x},\, \mathsf{MD}_\sigma),\, \mathsf{addr}(\mathsf{x},\, \mathsf{MD}_\sigma)),\, \sigma\rangle$$

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

$$\frac{\langle \mathsf{e},\, \sigma\rangle \to^* \langle \mathsf{v},\, \sigma'\rangle \qquad \langle \mathsf{a},\, \sigma'\rangle \to^* \langle \mathsf{c},\, \sigma''\rangle \qquad \mathsf{b} = \mathsf{mb}(\mathsf{fst}(\mathsf{c}),\, \mathsf{fst}(\mathsf{v}))}{\langle \mathsf{a}[\mathsf{e}],\, \sigma\rangle \to \langle (\mathsf{MD}_{\sigma''}(\mathsf{b}),\, \mathsf{b}),\, \sigma''\rangle}$$

The rule for access to structure elements has the form:

$$\frac{\langle \mathsf{e},\, \sigma\rangle \to^* \langle \mathsf{v},\, \sigma'\rangle \qquad \mathsf{b} = \mathsf{mb}(\mathsf{fst}(\mathsf{v}),\, \mathsf{m})}{\langle \mathsf{e.m},\, \sigma\rangle \to \langle (\mathsf{MD}_{\sigma'}(\mathsf{b}),\, \mathsf{b}),\, \sigma'\rangle}$$

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

$$\frac{\langle \mathsf{e},\, \sigma\rangle \to^* \langle \mathsf{v},\, \sigma'\rangle \qquad \mathsf{b} = \mathsf{fst}(\mathsf{v})}{\langle *\mathsf{e},\, \sigma\rangle \to \langle (\mathsf{MD}_{\sigma'}(\mathsf{b}),\, \mathsf{b}),\, \sigma'\rangle}$$

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

$$\frac{\langle \mathsf{e},\, \sigma\rangle \to^* \langle \mathsf{v},\, \sigma'\rangle}{\langle \&\mathsf{e},\, \sigma\rangle \to \langle \mathsf{snd}(\mathsf{v}),\, \sigma'\rangle}$$

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

$$\frac{\langle \mathsf{e},\, \sigma\rangle \to^* \langle \mathsf{v},\, \sigma'\rangle}{/**\mathsf{logtype}(\tau)\ **/\quad \langle (\tau')\mathsf{e},\, \sigma\rangle \to \langle \mathsf{cast}(\mathsf{fst}(\mathsf{v}),\, \mathsf{type}(\mathsf{e}),\, \tau),\, \sigma'\rangle}$$

The comma operator. Let e not contain comma operators at the high level. The rule for a comma operator has the form:

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

Logical operators && and ||. Let τ be a scalar type. There is a sequence point after evaluation of the first operand of logical operators. Depending on the result of evaluation of the first operand, the second operand of a logical operator can be evaluated or not. An auxiliary construct OrAnd specifies evaluation of the second operand. Rules for the && operator have the form:

$$\frac{\langle \mathsf{e}_1,\, \sigma\rangle \to^* \langle \mathsf{v},\, \sigma'\rangle \qquad \mathsf{cast}(\mathsf{fst}(\mathsf{v}),\, \mathsf{type}(\mathsf{e}_1),\, \mathsf{int}) = 0}{\langle \mathsf{e}_1 \,\&\&\, \mathsf{e}_2,\, \sigma\rangle \to \langle 0,\, \sigma'\rangle}$$

$$\frac{\langle \mathsf{e}_1,\, \sigma\rangle \to^* \langle \mathsf{v},\, \sigma'\rangle \qquad \mathsf{cast}(\mathsf{fst}(\mathsf{v}),\, \mathsf{type}(\mathsf{e}_1),\, \mathsf{int}) \neq 0}{\langle \mathsf{e}_1 \,\&\&\, \mathsf{e}_2,\, \sigma\rangle \to \langle \mathsf{OrAnd}(\mathsf{e}_2),\, \sigma'\rangle}$$

Rules for the || operator have the form:

$$\frac{\langle \mathsf{e}_1,\, \sigma\rangle \to^* \langle \mathsf{v},\, \sigma'\rangle \qquad \mathsf{cast}(\mathsf{fst}(\mathsf{v}),\, \mathsf{type}(\mathsf{e}_1),\, \mathsf{int}) \neq 0}{\langle \mathsf{e}_1 \,||\, \mathsf{e}_2,\, \sigma\rangle \to \langle 1,\, \sigma'\rangle}$$

$$\frac{\langle \mathsf{e}_1,\, \sigma\rangle \to^* \langle \mathsf{v},\, \sigma'\rangle \qquad \mathsf{cast}(\mathsf{fst}(\mathsf{v}),\, \mathsf{type}(\mathsf{e}_1),\, \mathsf{int}) = 0}{\langle \mathsf{e}_1 \,||\, \mathsf{e}_2,\, \sigma\rangle \to \langle \mathsf{OrAnd}(\mathsf{e}_2),\, \sigma'\rangle}$$

Semantics of the construct OrAnd is defined by the following rules:

$$\frac{\langle \mathsf{e},\, \sigma\rangle \to^* \langle \mathsf{v},\, \sigma'\rangle \qquad \mathsf{cast}(\mathsf{fst}(\mathsf{v}),\, \mathsf{type}(\mathsf{e}),\, \mathsf{int}) = 0}{\langle \mathsf{OrAnd}(\mathsf{e}),\, \sigma\rangle \to \langle 0,\, \sigma'\rangle}$$

$$\frac{\langle \mathsf{e},\, \sigma\rangle \to^* \langle \mathsf{v},\, \sigma'\rangle \qquad \mathsf{cast}(\mathsf{fst}(\mathsf{v}),\, \mathsf{type}(\mathsf{e}),\, \mathsf{int}) \neq 0}{\langle \mathsf{OrAnd}(\mathsf{e}),\, \sigma\rangle \to \langle 1,\, \sigma'\rangle}$$

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

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

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

$$\frac{\langle \mathsf{e},\ \sigma\rangle \to^* \langle \mathsf{v},\ \sigma'\rangle}{\begin{array}{l}\langle \mathsf{e} + +,\ \sigma\rangle \to \\ \langle \mathsf{v},\ \sigma'(\mathsf{MD} \leftarrow \mathsf{upd}(\mathsf{MD}_{\sigma'},\ \mathsf{snd}(\mathsf{v}),\ \mathsf{BinOpSem}(+,\ \mathsf{fst}(\mathsf{v}),\ \mathsf{type}(\mathsf{e}),\ 1,\ \mathsf{int})))\rangle\end{array}}$$

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

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

$$\frac{\begin{array}{ll}\langle \mathsf{e}_0,\ \sigma\rangle \to^* \langle \mathsf{v},\ \sigma'\rangle & \mathsf{cast}(\mathsf{fst}(\mathsf{v}),\ \mathsf{type}(\mathsf{e}_0),\ \mathsf{int}) \neq 0 \\ \langle \mathsf{e}_1,\ \sigma'\rangle \to^* \langle \mathsf{v}_1,\ \sigma''\rangle & \tau = \mathsf{type}(?:,\ \mathsf{type}(\mathsf{e}_1),\ \mathsf{type}(\mathsf{e}_2))\end{array}}{\langle \mathsf{e}_0\,?\,\mathsf{e}_1\ :\ \mathsf{e}_2,\ \sigma\rangle \to \langle \mathsf{cast}(\mathsf{v}_1,\ \mathsf{type}(\mathsf{e}_1),\ \tau),\ \sigma''\rangle}$$

$$\frac{\begin{array}{ll}\langle \mathsf{e}_0,\ \sigma\rangle \to^* \langle \mathsf{v},\ \sigma'\rangle & \mathsf{cast}(\mathsf{fst}(\mathsf{v}),\ \mathsf{type}(\mathsf{e}_0),\ \mathsf{int}) = 0 \\ \langle \mathsf{e}_2,\ \sigma'\rangle \to^* \langle \mathsf{v}_2,\ \sigma''\rangle & \tau = \mathsf{type}(?:,\ \mathsf{type}(\mathsf{e}_1),\ \mathsf{type}(\mathsf{e}_2))\end{array}}{\langle \mathsf{e}_0\,?\,\mathsf{e}_1\ :\ \mathsf{e}_2,\ \sigma\rangle \to \langle \mathsf{cast}(\mathsf{v}_2,\ \mathsf{type}(\mathsf{e}_2),\ \tau),\ \sigma''\rangle}$$

The other unary operators. Unary operators that do not have separate rules are defined by the general rule:

$$\frac{\langle \mathsf{e},\ \sigma\rangle \to^* \langle \mathsf{v},\ \sigma'\rangle}{\langle \odot\,\mathsf{e},\ \sigma\rangle \to \langle \mathsf{UnOpSem}(\odot,\ \mathsf{fst}(\mathsf{v}),\ \mathsf{type}(\mathsf{e})),\ \sigma'\rangle}$$

The other binary operators. Binary operators that do not have separate rules are defined by the general rule:

$$\frac{\langle \mathsf{e}_2,\ \sigma_0\rangle \to^* \langle \mathsf{v}_2,\ \sigma_1\rangle \qquad \langle \mathsf{e}_1,\ \sigma_1\rangle \to^* \langle \mathsf{v}_1,\ \sigma_2\rangle}{\langle \mathsf{e}_1\ \odot\ \mathsf{e}_2,\ \sigma\rangle \to \langle \mathsf{BinOpSem}(\odot,\ \mathsf{fst}(\mathsf{v}_1),\ \mathsf{type}(\mathsf{e}_1),\ \mathsf{fst}(\mathsf{v}_2),\ \mathsf{type}(\mathsf{e}_2)),\ \sigma_2\rangle}$$

4.2. Semantics of declarations

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

Let $\mathsf{e};$ be a variable declaration including the only declarator without an initializer. A rule for a declaration of a variable without an initialzer has the form:

$$\frac{\begin{array}{l}(\mathsf{MD}', \mathsf{V}) = \mathsf{init}(\tau, \mathsf{st}, \mathsf{upd}(\mathsf{MD}_\sigma, \mathsf{x}, \omega)) \\ \mathsf{V}' = \mathsf{x}, \text{ if } \tau \text{ is a structure type, and } \mathsf{V}' = \mathsf{V}, \text{ if not}\end{array}}{\begin{array}{l}\langle /**\,\mathsf{logtype}(\tau)\,\mathsf{storage}(\mathsf{st})\,\mathsf{name}(\mathsf{x})\ **/\quad \mathsf{e};, \sigma\rangle \rightarrow \\ \langle \epsilon, \sigma(\mathsf{MD} \leftarrow \mathsf{upd}(\mathsf{MD}', \mathsf{x}, \mathsf{V}'))\rangle\end{array}}$$

Let $\mathsf{e} = \mathsf{e}';$ be a variable declaration including the only declarator with an initializer e'. Rules for a declaration of a variable without an initialzer have the form:

$$\frac{\begin{array}{l}\langle \mathsf{computeInit}(\mathsf{e}'), \sigma\rangle \rightarrow \langle \mathsf{v}, \sigma'\rangle \quad \mathsf{v} \neq \omega \quad (\mathsf{MD}', \mathsf{V}) = \mathsf{init}(\tau, \mathsf{v}, \mathsf{upd}(\mathsf{MD}_{\sigma'}, \mathsf{x}, \omega)) \\ \mathsf{V}' = \mathsf{x}, \text{ if } \tau \text{ is a structure type, and } \mathsf{V}' = \mathsf{V}, \text{ if not}\end{array}}{\begin{array}{l}\langle /**\,\mathsf{logtype}(\tau)\,\mathsf{storage}(\mathsf{st})\,\mathsf{name}(\mathsf{x}), **/\quad \mathsf{e} = \mathsf{e}';, \sigma\rangle \rightarrow \\ \quad \langle \epsilon, \sigma'(\mathsf{MD} \leftarrow \mathsf{upd}(\mathsf{MD}', \mathsf{x}, \mathsf{V}'))\rangle\end{array}}$$

$$\frac{\langle \mathsf{computeInit}(\mathsf{e}'), \sigma\rangle \rightarrow \langle \omega, \sigma'\rangle}{\langle \mathsf{e} = \mathsf{e}';, \sigma\rangle \rightarrow \langle \omega, \sigma'\rangle}$$

The auxiliary construct $\mathsf{computeInit}(\mathsf{e})$ evaluates values of all expressions occuring in the initializer e from left to right:

$$\frac{\begin{array}{l}\langle \mathsf{computeInit}(\mathsf{e}_1), \sigma_0\rangle \rightarrow \langle \mathsf{v}_1, \sigma_1\rangle \qquad \mathsf{v}_1 \neq \omega \\ \ldots \\ \langle \mathsf{computeInit}(\mathsf{e}_k), \sigma_{k-1}\rangle \rightarrow \langle \mathsf{v}_k, \sigma_k\rangle \qquad \mathsf{v}_k \neq \omega\end{array}}{\langle \mathsf{computeInit}(\{\mathsf{e}_1, \ldots, \mathsf{e}_k\}), \sigma_0\rangle \rightarrow \langle \{\mathsf{v}_1, \ldots, \mathsf{v}_k\}, \sigma_k\rangle}$$

$$\frac{\begin{array}{l}\langle \mathsf{computeInit}(\mathsf{e}_1), \sigma_0\rangle \rightarrow \langle \mathsf{v}_1, \sigma_1\rangle \qquad \mathsf{v}_1 \neq \omega \\ \ldots \\ \langle \mathsf{computeInit}(\mathsf{e}_m), \sigma_{m-1}\rangle \rightarrow \langle \mathsf{v}_m, \sigma_m\rangle \qquad \mathsf{v}_m \neq \omega \qquad m < k \\ \langle \mathsf{computeInit}(\mathsf{e}_{m+1}), \sigma_m\rangle \rightarrow \langle \mathsf{v}_{m+1}, \sigma_{m+1}\rangle \qquad \mathsf{v}_{m+1} = \omega\end{array}}{\langle \mathsf{computeInit}(\{\mathsf{e}_1, \ldots, \mathsf{e}_k\}), \sigma_0\rangle \rightarrow \langle \omega, \sigma_k\rangle}$$

Let the initializer e be not enclosed in curly brackets. Then the following rule holds:

$$\frac{\langle \mathsf{e}, \sigma\rangle \rightarrow^* \langle \mathsf{v}, \sigma'\rangle}{\langle \mathsf{computeInit}(\mathsf{e}), \sigma\rangle \rightarrow \langle (\mathsf{fst}(\mathsf{v}), \mathsf{type}(\mathsf{e})), \sigma'\rangle}$$

Type declarations. Axioms for type declarations have the form:

$$\langle \mathsf{typedef}\ \mathsf{e};, \sigma\rangle \rightarrow \langle \epsilon, \sigma\rangle \qquad \langle \mathsf{enum}\ \mathsf{e}\,\{\mathsf{e}'\};, \sigma\rangle \rightarrow \langle \epsilon, \sigma\rangle$$

Function declarations. An axiom for a function declaration has the form:

$$\begin{array}{l}\langle /**\,\mathsf{logtype}(\tau_1 \times \ldots \times \tau_n \rightarrow \tau)\ **/\quad \tau'\ \mathsf{f}(\tau_1'\,\mathsf{x}_1, \ldots, \tau_n'\,\mathsf{x}_n)\{\mathsf{S}\}, \sigma\rangle \rightarrow \\ \langle \epsilon, \sigma(\mathsf{MD} \leftarrow \mathsf{upd}(\mathsf{MD}_\sigma, \mathsf{f}, (\tau, \tau_1 \times \ldots \times \tau_n, [\mathsf{x}_1, \ldots, \mathsf{x}_n], \{\mathsf{S}\})))\rangle\end{array}$$

4.3. Semantics of expressions

Labeled statements. A statement marked by a label L is executed when either it gets control in the usual way or it catches the exception $\mathsf{Exc(gotoStart(L))}$ thrown by a statement $\mathsf{goto\,L}$:

$$\langle \mathsf{L:}\quad \mathsf{T},\ \sigma\rangle \to \langle \mathsf{T},\ \sigma\rangle$$

$$\langle \mathsf{Exc(gotoStart(L))}\quad \mathsf{L:}\quad \mathsf{T},\ \sigma\rangle \to \langle \mathsf{T},\ \sigma\rangle$$

A statement marked by a label case is executed when either it gets control in the usual way or it catches the exception $\mathsf{Exc(switchStart(c))}$ thrown by a statement switch:

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

$$\frac{\mathsf{v = c}}{\langle \mathsf{Exc(switchStart(c))}\quad /**\mathsf{caseval(v)}**/\quad \mathsf{case\ e:}\quad \mathsf{T},\ \sigma\rangle \to \langle \mathsf{T},\ \sigma\rangle}$$

A statement marked by a label $\mathsf{default}$ is executed when either it gets control in the usual way or it catches the exception $\mathsf{Exc(defaultStart)}$ thrown by a statement $\mathsf{switchStop}$:

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

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

The auxiliary construct $\mathsf{switchStop(T)}$ catches the exception $\mathsf{Exc(switchStart(c))}$, thrown by a statement switch with a body T in the case that none of case labels from the body T does not coincide with the value of a controlling expression of the switch statement, and starts to begin the body T again, throwing the exception $\mathsf{Exc(defaultStart)}$. Otherwise, this construct is ignored:

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

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

The auxiliary construct $\mathsf{defaultStop}$ catches the exception $\mathsf{Exc(defaultStart)}$ thrown by the statement $\mathsf{switchStop(T)}$ in the event that T does not contain a label $\mathsf{default}$. Otherwise, this construct is ignored:

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

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

The block statement. An axiom for a block has the form:

$$\langle \{\mathsf{T}\},\, \sigma\rangle \to \langle \mathsf{T} \quad \mathsf{gotoStop(T)},\, \sigma\rangle$$

The auxiliary construct $\mathsf{gotoStop(T)}$ catches the exception $\mathsf{Exc(gotoStart(L))}$ thrown by a statement $\mathsf{goto\,L}$ in the event that the label L occurs in the sequence T of statements and declarations. Otherwise, this construct is ignored:

$$\frac{\mathsf{L} \in \mathsf{labels(T)}}{\begin{array}{l}\langle \mathsf{Exc(gotoStart(L))} \quad \mathsf{gotoStop(T)} \quad \mathsf{T}',\, \sigma\rangle \to \\ \quad \langle \mathsf{Exc(gotoStart(L))} \quad \mathsf{T} \quad \mathsf{gotoStop(T)} \quad \mathsf{T}',\, \sigma\rangle\end{array}}$$

$$\frac{\mathsf{L} \notin \mathsf{labels(T)}}{\begin{array}{l}\langle \mathsf{Exc(gotoStart(L))} \quad \mathsf{gotoStop(T)} \quad \mathsf{T}',\, \sigma\rangle \to \\ \quad \langle \mathsf{Exc(gotoStart(L))} \quad \mathsf{T}',\, \sigma\rangle\end{array}}$$

$$\langle \mathsf{gotoStop(T)},\, \sigma\rangle \to \langle \epsilon,\, \sigma\rangle$$

The expression statement. Execution of an expression statement is reduced to evaluation of the expression associated with it:

$$\langle \mathsf{e;},\, \sigma\rangle \to \langle \mathsf{e},\, \sigma\rangle$$

The null statement. A null statement executes no action:

$$\langle\, ;\, ,\, \sigma\,\rangle \to \langle\, \epsilon,\, \sigma\,\rangle$$

Selection statements. Execution of an `if` statement is defined by the following rules:

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

$$\frac{\langle \mathsf{e},\, \sigma\rangle \to^* \langle \mathsf{v},\, \sigma'\rangle \qquad \mathsf{cast(fst(v),\, type(e),\, int)} \neq 0}{\langle \mathsf{if\,(e)\,S_1\,else\,S_2},\, \sigma\rangle \to \langle \mathsf{S_1},\, \sigma'\rangle}$$

$$\frac{\langle \mathsf{e},\, \sigma\rangle \to^* \langle \mathsf{v},\, \sigma'\rangle \qquad \mathsf{cast(fst(v),\, type(e),\, int)} = 0}{\langle \mathsf{if\,(e)\,S_1\,else\,S_2},\, \sigma\rangle \to \langle \mathsf{S_2},\, \sigma'\rangle}$$

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

$$\frac{\langle \mathsf{e},\, \sigma\rangle \to^* \langle \mathsf{v},\, \sigma'\rangle \qquad \mathsf{cast(fst(v),\, type(e),\, int)} \neq 0}{\langle \mathsf{if\,(e)\,S},\, \sigma\rangle \to \langle \mathsf{S},\, \sigma'\rangle}$$

$$\frac{\langle \mathsf{e},\, \sigma\rangle \to^* \langle \mathsf{v},\, \sigma'\rangle \qquad \mathsf{cast(fst(v),\, type(e),\, int)} = 0}{\langle \mathsf{if\,(e)\,S},\, \sigma\rangle \to \langle \epsilon,\, \sigma'\rangle}$$

The switch statement $\mathsf{switch(e)\,S}$ evaluates the value c of the controlling expression e and throws the exception $\mathsf{Exc(switchStart(c))}$ that can be catched by statements from S marked by labels case and $\mathsf{default}$:

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

$$\frac{\langle \mathsf{e},\, \sigma\rangle \to^* \langle \mathsf{v},\, \sigma'\rangle}{\begin{array}{l}\langle \mathsf{switch(e)}\ \{\mathsf{S}\},\, \sigma\rangle \to \\ \langle \mathsf{Exc(switchStart(fst(v)))}\quad \mathsf{S}\quad \mathsf{switchStop(S)}\quad \mathsf{gotoStop(S)}\quad \mathsf{breakStop},\\ \sigma'\rangle\end{array}}$$

The auxiliary construct breakStop catches the exception Exc(breakStart(c)) thrown by a statement break:

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

$$\langle \mathsf{breakStop},\, \sigma\rangle \to \langle \epsilon,\, \sigma\rangle$$

Iteration statements. The auxiliary construct continueStop(e, S) is used in rules for iteration statements. It catches an exception Exc(continueStart) thrown by a statement continue and in addition checks the controlling expression e of an iteration statement with a body T before exiting the iteration statement:

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

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

$$\frac{\langle \mathsf{e},\, \sigma\rangle \to^* \langle \mathsf{v},\, \sigma'\rangle \qquad \mathsf{cast(fst(v),\ type(e),\ int)} = 0}{\langle \mathsf{continueStop(e,\ T)}\quad \mathsf{T}',\, \sigma\rangle \to \langle \mathsf{T}',\, \sigma'\rangle}$$

$$\frac{\langle \mathsf{e},\, \sigma\rangle \to^* \langle \mathsf{v},\, \sigma'\rangle \qquad \mathsf{cast(fst(v),\ type(e),\ int)} = 0}{\langle \mathsf{Exc(continueStart)}\quad \mathsf{continueStop(e,\ T)}\quad \mathsf{T}',\, \sigma\rangle \to \langle \mathsf{T}',\, \sigma'\rangle}$$

$$\frac{\langle \mathsf{e},\, \sigma\rangle \to^* \langle \mathsf{v},\, \sigma'\rangle \qquad \mathsf{cast(fst(v),\ type(e),\ int)} \neq 0}{\begin{array}{l}\langle \mathsf{continueStop(e,\ T)}\quad \mathsf{T}',\, \sigma\rangle \to \\ \langle \mathsf{T}\quad \mathsf{gotoStop(T)}\quad \mathsf{continueStop(e,\ T)}\quad \mathsf{T}',\, \sigma'\rangle\end{array}}$$

$$\frac{\langle \mathsf{e},\, \sigma\rangle \to^* \langle \mathsf{v},\, \sigma'\rangle \qquad \mathsf{cast(fst(v),\ type(e),\ int)} \neq 0}{\begin{array}{l}\langle \mathsf{Exc(continueStart)}\quad \mathsf{continueStop(e,\ T)}\quad \mathsf{T}',\, \sigma\rangle \to \\ \langle \mathsf{T}\quad \mathsf{gotoStop(T)}\quad \mathsf{continueStop(e,\ T)}\quad \mathsf{T}',\, \sigma'\rangle\end{array}}$$

Execution of iteration statements is reduced to execution of the construct continueStop:

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

$$\begin{array}{l}\langle \mathsf{do\ S\ while(e);},\, \sigma\rangle \to \\ \langle \mathsf{S}\quad \mathsf{gotoStop(S)}\quad \mathsf{continueStop(e,\ S)}\quad \mathsf{breakStop},\, \sigma\rangle\end{array}$$

$$\begin{array}{l}\langle \mathsf{for}(\mathsf{e}_1;\mathsf{e}_2;\mathsf{e}_3)\ \mathsf{S},\ \sigma\rangle \rightarrow\\ \langle \mathsf{e}_1;\quad \mathsf{if}(!\mathsf{e}_2)\ \mathsf{break};\quad \mathsf{S}\quad \mathsf{gotoStop}(\mathsf{S})\\ \quad \mathsf{continueStop}((\mathsf{e}_3,\ \mathsf{e}_2),\ \mathsf{S})\quad \mathsf{breakStop},\ \sigma\rangle\end{array}$$

Jump statements. Jump statements throw exceptions of the form $\mathsf{Exc}(\ldots)$. These exceptions are catched by C-light constructs of the C-light language and auxiliary constructs by analogy with mechanism of exception handling:

$$\langle \mathsf{goto\,L};,\ \sigma\rangle \rightarrow \langle \mathsf{Exc}(\mathsf{gotoStart}(\mathsf{L})),\ \sigma\rangle$$

$$\langle \mathsf{break};,\ \sigma\rangle \rightarrow \langle \mathsf{Exc}(\mathsf{breakStart}),\ \sigma\rangle$$

$$\langle \mathsf{continue};,\ \sigma\rangle \rightarrow \langle \mathsf{Exc}(\mathsf{continueStart}),\ \sigma\rangle$$

$$\langle \mathsf{return};,\ \sigma\rangle \rightarrow \langle \mathsf{Exc}(\mathsf{returnStart}),\ \sigma\rangle$$

$$\frac{\langle \mathsf{e},\ \sigma\rangle \rightarrow^* \langle \mathsf{v},\ \sigma'\rangle}{\begin{array}{l}\langle /**\mathsf{rettype}(\tau)\ **/\quad \mathsf{return\ e};,\ \sigma\rangle \rightarrow\\ \langle \mathsf{Exc}(\mathsf{returnStart}(\mathsf{cast}(\mathsf{fst}(\mathsf{v}),\ \mathsf{type}(\mathsf{e}),\ \tau)),\ \sigma'\rangle\end{array}}$$

$$\frac{\langle \mathsf{e},\ \sigma\rangle \rightarrow^* \langle \omega,\ \sigma'\rangle}{\langle \mathsf{return\ e};,\ \sigma\rangle \rightarrow \langle \omega,\ \sigma'\rangle}$$

4.4. Other rules

The sequence rule. Let S be a statement or an auxiliary construct, T be a nonempty sequence of statements, declarations and auxiliary constructs. A sequence rule has the form:

$$\frac{\langle \mathsf{S},\ \sigma\rangle \rightarrow \langle \mathsf{S}',\ \sigma'\rangle}{\langle \mathsf{S}\quad \mathsf{T},\ \sigma\rangle \rightarrow \langle \mathsf{S}'\quad \mathsf{T},\ \sigma'\rangle}$$

Axioms of exception propagation. These axioms are applied when none of the above axioms and rules are applicable. Let E be a declaration, statement or auxiliary construct. The axioms have the form:

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

$$\langle \omega\quad \mathsf{E}\quad \mathsf{T},\ \sigma\rangle \rightarrow \langle \omega\quad \mathsf{T},\ \sigma'\rangle$$

Program. Rules for a program $\mathsf{Prgm}(\mathsf{T})$ consisting of the sequence T of declarations have the form:

$$\frac{\begin{array}{l}\langle \mathsf{T},\ \sigma(\mathsf{MD} \leftarrow \mathsf{upd}(\mathsf{MD},\ (\mathsf{y}_1,\ \ldots,\ \mathsf{y}_k),\ \bot))\rangle \rightarrow^* \langle \epsilon,\ \sigma'\rangle\\ \mathsf{MD}_{\sigma'}(\mathsf{main}) = (\tau,\ \tau_1 \times \ldots \times \tau_n,\ [\mathsf{z}_1,\ \ldots,\ \mathsf{z}_n],\ \mathsf{S})\\ \langle \mathsf{S}\quad \mathsf{returnStop},\ \sigma'\rangle \rightarrow^* \langle \omega,\ \sigma''\rangle\end{array}}{\langle /*\ \mathsf{prgvar}(\mathsf{y}_1,\ \ldots,\ \mathsf{y}_k)\ */\quad \mathsf{Prgm}(\mathsf{T}),\ \sigma\rangle \rightarrow \langle \omega,\ \sigma''\rangle},$$

$$\frac{\begin{array}{l}\langle \mathsf{T},\ \sigma(\mathsf{MD} \leftarrow \mathsf{upd}(\mathsf{MD},\ (\mathsf{y}_1,\ \ldots,\ \mathsf{y}_k),\ \bot)\rangle \rightarrow^* \langle \epsilon,\ \sigma'\rangle \\ \mathsf{MD}_{\sigma'}(\mathsf{main}) = (\tau,\ \tau_1 \times \ldots \times \tau_n,\ [\mathsf{z}_1,\ \ldots,\ \mathsf{z}_n],\ \mathsf{S}) \\ \langle \mathsf{S}\quad \mathsf{returnStop},\ \sigma'\rangle \rightarrow^* \langle \mathsf{v},\ \sigma''\rangle\end{array}}{\begin{array}{l}\langle / *\ \mathsf{prgvar}(\mathsf{y}_1,\ \ldots,\ \mathsf{y}_k)\ * /\quad \mathsf{Prgm}(\mathsf{T}),\ \sigma\rangle \rightarrow \\ \langle \mathsf{v},\ \sigma''(\mathsf{MD} \leftarrow \mathsf{upd}(\mathsf{MD},\ (\mathsf{y}_1,\ \ldots,\ \mathsf{y}_k,\ \mathsf{z}_1,\ \ldots,\ \mathsf{z}_n),\ \bot))\rangle\end{array}}$$

The parameters $\mathsf{z}_1, \ldots, \mathsf{z}_n$ of the main function specify the program parameters.

5. Overview of the C-kernel language

The C-kernel language is an intermediate language in the two-level program verification schema in which C-light programs are translated. It is a subset of the C-light language.

The set of keywords of the C-kernel language is reduced in comparison with the C-light language and includes the following keywords: auto, _Bool, char, delete, double, else, enum, float, goto, if, int, long, new, return, short, signed, sizeof, static, struct, typedef, unsigned, void, while.

Expressions. The number of side effects in C-kernel is minimized and operators containing sequence points (for example, logical operators && and ||) are forbidden. A normalized expression is an expression that does not contain conditional operations, a comma operator, logical operators && and ||, simple and compound assignments, increment and decrement operators. Let e (may be with indices) denote a normalized expression, τ be a type. An expression statement of C-kernel has a form:

- $\mathsf{e} = \mathsf{e}'(\mathsf{e}_1,\ \ldots,\ \mathsf{e}_n)$;
- $\mathsf{e} = \mathsf{new}(\tau)$;
- $\mathsf{e} = \mathsf{e}'$;
- $\mathsf{e}'(\mathsf{e}_1,\ \ldots,\ \mathsf{e}_n)$;
- delete(e);

Declarations. The following restrictions are put on the set of C-kernel declarations:

1. Lists of declared objects are enabled only in function declarations, any other declaration specifies exactly one object.
2. Storage class specifiers static and auto are obligatory. Other storage class specifiers are forbidden.

Statements. The following operators are valid in the C-kernel language:

1. an expression statement,
2. a null statement,
3. an if statement with the obligatory else branch and a normalized condition,
4. a while statement with a normalized condition,
5. a goto statement,
6. a return statement in which a returning expression is a normalized expression,
7. a block statement.

6. Annotation language

The annotation language is used for description of the properties of states of the C-light machine and for writing annotations in axiomatic semantics of C-kernel.

6.1. Alphabet

The alphabet of the annotation language consists of the following classes of symbols:

- *variables*,
- *constants* (in particular, symbols of operators of the C-kernel language),
- *quantifiers* $\exists$ and $\forall$ and *logical connectives* $\neg$, $\Rightarrow$, $\Leftrightarrow$, $\wedge$, $\vee$,
- *brackets* (,), [,], <, >, {, },
- *punctuation symbols*: period, colon, comma.

6.2. Expressions

Variables and constants can be of any type except void. The set of all variables is denoted by Var. The set of values of a variable of a base type includes the undefined value. The undefined value is denoted by ω.

Expressions of the annotation language are defined by induction:

- a variable v of a type τ is an expression of the type τ;
- a constant c of a type τ is an expression of the type τ;
- if $\mathsf{s}_1, \ldots, \mathsf{s}_\mathsf{n}$ are expressions of types $\tau_1, \ldots, \tau_\mathsf{n}$, repsectively, and s is an expression of the type $\tau_1 \times ... \times \tau_\mathsf{n} \to \tau$, then $\mathsf{s}(\mathsf{s}_1, \ldots, \mathsf{s}_\mathsf{n})$ is an expression of the type τ.

Logical expressions are constructed from expressions of the type $\_\mathsf{Bool}$ with the help of logical connectives and quantifiers. Further the expressions of the type $\_\mathsf{Bool}$ are called assertions.

Expressions of the annotation language have a prefix form. Particularly this means that the standard C-light operators are written in this form. For example, the expressions like $\mathsf{a}[3]$, $\mathsf{r.f}$ and $\mathsf{x}+3$ will have the form $[\,](\mathsf{a},3)$, $\mathsf{elem}(\mathsf{r},\mathsf{f})$ and $+(\mathsf{x},3)$, repsectively. Further in examples we write such expressions in a usual infix form for convenience.

6.3. Substitutions

The concept of substitution in expressions is used in the definition of axiomatic semantics of C-kernel. Substitution is a function which maps expressions to expressions. The substitution of a term for a variable is the replacement of all free occurences of the variable.

A substitution of an expression t for a variable u in an expression s is usually written as $\mathsf{s}(\mathsf{u} \leftarrow \mathsf{t})$ and is defined by induction of the expression structure:

- if s is a variable, then
$$\mathsf{s}(\mathsf{u} \leftarrow \mathsf{t}) \equiv \begin{cases} \mathsf{t}, & \text{if } \mathsf{s} \equiv \mathsf{u}, \\ \mathsf{s}, & \text{otherwise;} \end{cases}$$
- if s is a constant of a base type, then $\mathsf{s}(\mathsf{u} \leftarrow \mathsf{t}) \equiv \mathsf{s}$;
- if $\mathsf{s} \equiv \mathsf{f}(\mathsf{s}_1, \ldots, \mathsf{s}_n)$ for some expression f of the functional type, then $\mathsf{s}(\mathsf{u} \leftarrow \mathsf{t}) \equiv \mathsf{f}(\mathsf{u} \leftarrow \mathsf{t})(\mathsf{s}_1(\mathsf{u} \leftarrow \mathsf{t}), \ldots, \mathsf{s}_n(\mathsf{u} \leftarrow \mathsf{t}))$.

6.4. Interpretation of types

The set of values associated with each type τ is called the carrier of the type τ and denoted by D_τ. Since we do not describe semantics for a specific architecture, carrier borders of the base types are specified by symbolic constants. Note that one of the ways of interpretation of these constants can be initialisation by values from the standard file `limits.h`.

- $\mathsf{D}_{\_\mathsf{Bool}} = \{\mathsf{FALSE}, \mathsf{TRUE}\}$;
- $\mathsf{D}_{\mathsf{unsigned\ char}} = \{0 \ldots \mathsf{UCHAR\_MAX}\}$;

  $\ldots$
- $\mathsf{D}_{\mathsf{wchar\_t}} = \mathsf{D}_{\mathsf{unsigned\,short}}$;
- $\mathsf{D}_{\mathsf{enum}} = \mathsf{D}_{\mathsf{signed\,int}}$ for any enumeration;
- $\mathsf{D}_{\mathsf{void}} = \emptyset$;
- $\mathsf{D}_{\tau *} = \mathsf{D}_{\mathsf{Locations}}$ for each type τ;

- $\mathsf{D}_{\tau[\mathsf{n}]} = \mathsf{D}_{\tau}^{\mathsf{n}}$ (Cartesian power of n) for each nonempty and nonfunctional type τ;
- $\mathsf{D}_{\mathsf{struct}\{\tau_1\,\mathsf{v}_1;\,\dots;\,\tau_\mathsf{n}\,\mathsf{v}_\mathsf{n};\}} = \mathsf{D}_{\tau_1} \times \dots \times \mathsf{D}_{\tau_\mathsf{n}}$;
- $\mathsf{D}_{\tau_1\times\dots\times\tau_\mathsf{n}\to\tau} = \mathsf{D}_{\tau_1} \times \dots \times \mathsf{D}_{\tau_\mathsf{n}} \to \mathsf{D}_{\tau}$, i. e. the set of all functions from the Cartesian product of sets $\mathsf{D}_{\tau_1}, \dots, \mathsf{D}_{\tau_\mathsf{n}}$ to the set D_{τ};
- D_{ID} = the set of identifiers of the C language;
- $\mathsf{D}_{\mathsf{Addrs}}$ = the set of non-interpreted constants;
- $\mathsf{D}_{\mathsf{TypeSpecs}}$ = the set of abstract type names of the C language.

The semantic domain D is defined as a union on all types: $\mathsf{D} = \bigcup\limits_{\tau} \mathsf{D}_{\tau}$.

6.5. Interpretation of expressions

We assume that each constant of a base type denotes itself.

The values of variables are defined by the states of the C-light machine. *A state* is a map which assigns a value of the domain D_τ for each variable of the type τ.

Semantics (interpretation) $\mathcal{I}\|\mathsf{s}\|$ of the expression s of the type τ is the map

$$\mathcal{I}\|\mathsf{s}\| : \mathsf{States} \to \mathsf{D}_\mathsf{T},$$

which is defined by induction on the structure of s:

- if s is a variable, then $\mathcal{I}\|\mathsf{s}\|(\sigma) = \sigma(\mathsf{s})$;
- if s is a constant designating a value d, then $\mathcal{I}\|\mathsf{s}\|(\sigma) = \mathsf{d}$;
- if $\mathsf{s} \equiv \mathsf{op}(\mathsf{s}_1, \dots, \mathsf{s}_\mathsf{n})$ for some expression op, then

$$\mathcal{I}\|\mathsf{s}\|(\sigma) = \mathcal{I}\|\mathsf{op}\|(\sigma)(\mathcal{I}\|\mathsf{s}_1\|(\sigma), \dots, \mathcal{I}\|\mathsf{s}_\mathsf{n}\|(\sigma)).$$

Since $\mathcal{I}$ holds everywhere, further we will write $\sigma(\mathsf{s})$ instead of $\mathcal{I}\|\mathsf{s}\|(\sigma)$.

6.6. Interpretation of assertions

We define an update of a state σ denoted by $\sigma(\mathsf{u} \leftarrow \mathsf{d})$, where u is a variable of a type τ, and d is an element of the type τ. It is a state which coincides with σ everywhere except for, may be, the variable u, and $\sigma(\mathsf{u}) = \mathsf{d}$. Updates are used to model assignments of values to variables.

Finally we define the concept of truth of an assertion p in a state σ denoted by $\sigma \models \mathsf{p}$. Truth is defined by induction on the structure of the assertion p.

- $\sigma \models \mathsf{B}$ if and only if $\sigma(\mathsf{B}) = \mathsf{true}$, if B is an elementary logical formula (i. e. an elementary expression of the type $\mathsf{\_Bool}$);

- $\sigma \models \neg \mathsf{p}$ if and only if it is wrong that $\sigma \models \mathsf{p}$ (denoted by $\sigma \not\models \mathsf{p}$);
- $\sigma \models \mathsf{p} \vee \mathsf{q}$ if and only if $\sigma \models \mathsf{p}$ or $\sigma \models \mathsf{q}$;
- $\sigma \models \exists \mathsf{x}.\, \mathsf{p}$ if and only if $\sigma(\mathsf{x} \leftarrow \mathsf{d}) \models \mathsf{p}$ for some element d.

For the rest of logical connectives and the quantifier $\forall$, truth is defined by common logical relations ($\mathsf{p} \wedge \mathsf{q} \equiv \neg(\neg \mathsf{p} \vee \neg \mathsf{q})$ and others).

If $\sigma \models \mathsf{p}$, then we say that p is satisfied in σ. Also we use the concept of the truth domain of an assertion defined as

$$\|\mathsf{p}\| = \{\sigma \mid \sigma \text{ is a state and } \sigma \models \mathsf{p}\}.$$

We say that an assertion p *is true* or is satisfied if $\|\mathsf{p}\| = \mathsf{States}$.

7. Axiomatic semantics of C-kernel

7.1. Main notions

Axiomatic semantics of a programming language is defined by a system of axioms and inference rules over formulas of the form $\{\mathsf{P}\}\,\mathsf{S}\,\{\mathsf{Q}\}$ called Hoare triples, where P and Q are formulas of the annotation language and S is a program or a program fragment. The Hoare triple $\{\mathsf{P}\}\,\mathsf{S}\,\{\mathsf{Q}\}$ is true, if the following holds: if the precondition P is true before execution of S and execution of S is terminated, then the postcondition Q is true after termination.

We assume that each function in a program is provided by pre- and postconditions, and each label marking a labelled statement is provided by a formula of the annotation language called an invariant of the label. An invariant of a label specifies a condition that becomes true when execution of a program reaches the label. Information about pre- and postconditions of functions and invariants of labeled statements is specified by functions Pre, Post and Inv, respectively. The functions $\mathsf{Pre}(\mathsf{c})$ and $\mathsf{Post}(\mathsf{c})$ return formulas called a precondition and a postcondition of a function with an entry point that is allocated at the address c. The precondition $\mathsf{Pre}(\mathsf{c})$ depends on the metavariable MD, logical variables argv_1, ..., argv_n that specify the values of arguments of the function. The postcondition $\mathsf{Post}(\mathsf{c})$ depends on the metavariables MD and Val, and logical variables argv_1, ..., argv_n. The metavariable Val specifies the value returned by the function. Pre- and postconditions can additionally depend on the logical variable fpar that is a parameter of the Hoare triple for calls of the function. The function $\mathsf{Inv}(\mathsf{L})$ returns an invariant of the label L.

7.2. The expression statement

Function call. Semantics of a function call is defined by two sorts of rules: rules for a function returning a value and a rule for a function that returns no value.

Let P' and Q' denote formulas

$$\mathsf{Pre}(\mathsf{val}(\mathsf{e}_0, \mathsf{MD}))(\mathsf{argv}_1 \leftarrow \mathsf{val}(\mathsf{e}_1,\ \mathsf{MD}),\ \ldots,\ \mathsf{argv}_\mathsf{n} \leftarrow \mathsf{val}(\mathsf{e}_\mathsf{n},\ \mathsf{MD}))$$

and

$$\mathsf{Post}(\mathsf{val}(\mathsf{e}_0, \mathsf{MD}_1))(\mathsf{argv}_1 \leftarrow \mathsf{val}(\mathsf{e}_1,\ \mathsf{MD}_1),\ \ldots,\ \mathsf{argv}_\mathsf{n} \leftarrow \mathsf{val}(\mathsf{e}_\mathsf{n},\ \mathsf{MD}_1))$$

, respectively.

Axioms for a call of a function designated by the expression e_0 in the case that it returns a value have the form:

$$\begin{array}{l}
\{\exists\, \mathsf{fpar}\, (\mathsf{P}'(\mathsf{MD}) \wedge \\
\forall\, \mathsf{MD}'.\, \forall\, \mathsf{Val}'.\, \mathsf{Q}'(\mathsf{MD}',\ \mathsf{Val}') \Rightarrow \\
\quad \mathsf{Q}(\mathsf{MD} \leftarrow \mathsf{upd}(\mathsf{MD}',\ \mathsf{val}(\mathsf{e},\ \mathsf{MD}'),\ \mathsf{Val}'))(\mathsf{Val} \leftarrow \mathsf{Val}')(\mathsf{MD}_1 \leftarrow \mathsf{MD}))\} \\
\quad \mathsf{e} = \mathsf{e}_0(\mathsf{e}_1,\ \ldots,\ \mathsf{e}_\mathsf{n}); \\
\{\mathsf{Q}\}
\end{array}$$

if $\mathsf{type}(\mathsf{e})$ is not a structure type.

$$\begin{array}{l}
\{\exists\, \mathsf{fpar}\, (\mathsf{P}'(\mathsf{MD}) \wedge \\
\forall\, \mathsf{MD}'.\, \forall\, \mathsf{Val}'.\, \mathsf{Q}'(\mathsf{MD}',\ \mathsf{Val}') \Rightarrow \\
\quad \mathsf{Q}(\mathsf{MD} \leftarrow \mathsf{upd}(\mathsf{MD}',\ (\mathsf{mb}(\mathsf{v},\ \mathsf{l}_1),\ \ldots,\ \mathsf{mb}(\mathsf{v},\ \mathsf{l}_\mathsf{m})), \\
\qquad (\mathsf{MD}'(\mathsf{mb}(\mathsf{Val}',\ \mathsf{l}_1)),\ \ldots,\ \mathsf{MD}'(\mathsf{mb}(\mathsf{Val}',\ \mathsf{l}_\mathsf{m})))))(\mathsf{Val} \leftarrow \mathsf{Val}')(\mathsf{MD}_1 \leftarrow \mathsf{MD}))\} \\
\quad \mathsf{e} = \mathsf{e}_0(\mathsf{e}_1,\ \ldots,\ \mathsf{e}_\mathsf{n}); \\
\{\mathsf{Q}\}
\end{array}$$

where $\mathsf{v} = \mathsf{val}(\mathsf{e},\ \mathsf{MD}')$, if $\mathsf{type}(\mathsf{e})$ is a structure type with fields $\mathsf{l}_1,\ \ldots,\ \mathsf{l}_\mathsf{m}$. The metavariable Val occuring in the postcondition Q' of the function stores the value returned by the function.

The axiom for a call of a function designated by the expression e_0 in the case that it returns no value has the form:

$$\begin{array}{l}
\{\exists\, \mathsf{fpar}.\, (\mathsf{P}'(\mathsf{MD}) \wedge \forall\, \mathsf{MD}'.\, \mathsf{Q}'(\mathsf{MD}',\ \mathsf{Val}') \Rightarrow \mathsf{Q}(\mathsf{MD} \leftarrow \mathsf{MD}')(\mathsf{MD}_1 \leftarrow \mathsf{MD}))\} \\
\mathsf{e}_0(\mathsf{e}_1,\ \ldots,\ \mathsf{e}_\mathsf{n}); \\
\{\mathsf{Q}\}
\end{array}$$

The new operator. Axioms for a new operator have the form:

$$\begin{array}{l}
\{\mathsf{Q}(\mathsf{MD} \leftarrow \mathsf{upd}(\mathsf{fst}(\mathsf{U}),\ \mathsf{addr}(\mathsf{e},\ \mathsf{fst}(\mathsf{U})),\ \mathsf{snd}(\mathsf{U})))\} \\
\ \mathsf{e} = /**\,\mathsf{logtype}(\tau)\ **/\ \mathsf{new}\,\mathsf{e}';\ \{\mathsf{Q}\},
\end{array}$$

where $\mathsf{U} = \mathsf{new}(\tau,\ \mathsf{MD})$, if τ is not a structure type.

$$\begin{array}{l}
\{\mathsf{Q}(\mathsf{MD} \leftarrow \mathsf{upd}(\mathsf{MD}',\ (\mathsf{mb}(\mathsf{val}(\mathsf{e},\ \mathsf{MD}'),\ \mathsf{l}_1),\ \ldots,\ \mathsf{mb}(\mathsf{val}(\mathsf{e},\ \mathsf{MD}'),\ \mathsf{l}_\mathsf{m})), \\
\qquad\qquad (\mathsf{MD}'(\mathsf{mb}(\mathsf{snd}(\mathsf{U}),\ \mathsf{l}_1)),\ \ldots,\ \mathsf{MD}'(\mathsf{mb}(\mathsf{snd}(\mathsf{U}),\ \mathsf{l}_\mathsf{m})))))\} \\
\ \mathsf{e} = /**\,\mathsf{logtype}(\tau)\ **/\ \mathsf{new}\,\mathsf{e}';\ \{\mathsf{Q}\},
\end{array}$$

where $\mathsf{U} = \mathsf{new}(\tau,\ \mathsf{MD})$ and $\mathsf{MD}' = \mathsf{fst}(\mathsf{U})$, if τ is a structure type with fields $\mathsf{l}_1,\ \ldots,\ \mathsf{l}_\mathsf{m}$.

The delete operator. An axiom for a delete operator has the form:

$$\{\mathsf{Q}(\mathsf{MD} \leftarrow \mathsf{delete}(\mathsf{MD},\ \mathsf{val}(\mathsf{e},\ \mathsf{MeM}),\ \tau))\}\ /*\ \mathsf{logtype}(\tau)\ */\\ \mathsf{delete}(\mathsf{e},\ \tau');\ \{\mathsf{Q}\}$$

Assignment statements. Let an expression e' not include a function call, operator new and cast operators. Axioms for an assignment statement have the form:

$$\{\mathsf{Q}(\mathsf{MD} \leftarrow \mathsf{upd}(\mathsf{MD},\ \mathsf{addr}(\mathsf{e},\ \mathsf{MD}),\ \mathsf{cast}(\mathsf{val}(\mathsf{e}',\ \mathsf{MD}),\ \mathsf{type}(\mathsf{e}'),\ \mathsf{type}(\mathsf{e}))))\}\\ \mathsf{e} = \mathsf{e}';\ \{\mathsf{Q}\},$$

if type(e) is not a structure type.

$$\{\mathsf{Q}(\mathsf{MD} \leftarrow \mathsf{upd}(\mathsf{MD},\ (\mathsf{mb}(\mathsf{val}(\mathsf{e},\ \mathsf{MD}),\ \mathsf{l}_1),\ \ldots,\ \mathsf{mb}(\mathsf{val}(\mathsf{e},\ \mathsf{MD}),\ \mathsf{l}_\mathsf{m})),\\ (\mathsf{MD}(\mathsf{mb}(\mathsf{val}(\mathsf{e}',\ \mathsf{MD}),\ \mathsf{l}_1)),\ \ldots,\ \mathsf{MD}(\mathsf{mb}(\mathsf{val}(\mathsf{e}',\ \mathsf{MD}),\ \mathsf{l}_\mathsf{m})))))\}\\ \mathsf{e} = \mathsf{e}';\ \{\mathsf{Q}\},$$

if type(e) is a structure type with fields $\mathsf{l}_1, \ldots, \mathsf{l}_\mathsf{m}$.

7.3. Declarations

Variable declarations. Semantics of variable declarations is defined by two axioms. The first axiom specifies declarations without an initializer. The second axiom specifies declarations with an initializer.

The axiom for a variable declaration without an initializer has the form:

$$\{\mathsf{Q}(\mathsf{MD} \leftarrow \mathsf{upd}(\mathsf{fst}(\mathsf{U}),\ \mathsf{x},\ \mathsf{V}))\}\\ /**\ \mathsf{logtype}(\tau)\ \mathsf{storage}(\mathsf{st})\ \mathsf{name}(\mathsf{x})\ **/\quad \mathsf{e};\ \{\mathsf{Q}\},$$

where $\mathsf{U} = \mathsf{init}(\tau,\ \mathsf{st},\ \mathsf{upd}(\mathsf{MD},\ \mathsf{x},\ \omega))$, $\mathsf{V} = \mathsf{x}$, if τ is a structure type, and $\mathsf{V} = \mathsf{snd}(\mathsf{U})$, otherwise.

The axiom for a variable declaration with an initializer has the form:

$$\{\mathsf{Q}(\mathsf{MD} \leftarrow \mathsf{upd}(\mathsf{fst}(\mathsf{U}),\ \mathsf{x},\ \mathsf{V}))\}\ /**\ \mathsf{logtype}(\tau)\ \mathsf{name}(\mathsf{x})\ **/\quad \mathsf{e} = \mathsf{e}';\ \{\mathsf{Q}\},$$

where $\mathsf{U} = \mathsf{init}(\tau,\ \mathsf{e}',\ \mathsf{upd}(\mathsf{MD},\ \mathsf{x},\ \omega))$, $\mathsf{V} = \mathsf{x}$, if τ is a structure type, and $\mathsf{V} = \mathsf{snd}(\mathsf{U})$, otherwise.

Type declaration. Axioms for a type declaration have the form:

$$\{\mathsf{Q}\}\ \mathsf{typedef}\ \mathsf{e};\ \{\mathsf{Q}\} \qquad \{\mathsf{Q}\}\ \mathsf{enum}\ \mathsf{e}\ \{\mathsf{e}'\};\ \{\mathsf{Q}\}$$

Function declarations. The axiom for a function declaration has the form:

$$\{\mathsf{Q}(\mathsf{MD} \leftarrow \mathsf{upd}(\mathsf{MD},\ \mathsf{f},\ (\tau,\ \tau_1 \times \ldots \times \tau_\mathsf{n},\ [\mathsf{x}_1,\ \ldots,\ \mathsf{x}_\mathsf{n}],\ \{\mathsf{S}\}))))\}\\ /**\mathsf{logtype}(\tau_1 \times \ldots \times \tau_\mathsf{n} \to \tau)\ **/\quad \tau'\ \mathsf{f}(\tau_1'\ \mathsf{x}_1,\ \ldots,\ \tau_\mathsf{n}'\ \mathsf{x}_\mathsf{n})\ \{\mathsf{S}\}\ \{\mathsf{Q}\}.$$

7.4. Statements

Labeled statements. The rule for a labeled statement has the form:

$$\frac{\{\mathsf{Inv(L)}\}\,\mathsf{S}\,\{\mathsf{Q}\}}{\{\mathsf{Inv(L)}\}\,\mathsf{L:S}\,\{\mathsf{Q}\}}$$

The block statement. The rule for a block statement has the form:

$$\frac{\{\mathsf{P}\}\,\mathsf{S}\{\mathsf{Q}\}}{\{\mathsf{P}\}\,\{\mathsf{S}\}\,\{\mathsf{Q}\}}$$

The null statement. The axiom for a null statement has the form:

$$\{\mathsf{Q}\}\,;\,\{\mathsf{Q}\}$$

Selection statements. Selection statements of the C-kernel language include only if statements:

$$\frac{\begin{array}{c}\{\mathsf{P} \wedge \mathsf{cast(val(e,\ MD),\ type(e),\ int)} \neq 0\}\,\mathsf{S}_1\,\{\mathsf{Q}\}\\ \{\mathsf{P} \wedge \mathsf{cast(val(e,\ MD),\ type(e),\ int)} = 0\}\,\mathsf{S}_2\,\{\mathsf{Q}\}\end{array}}{\{\mathsf{P}\}\,\mathsf{if(e)}\,\mathsf{S}_1\,\mathsf{else}\,\mathsf{S}_2\,\{\mathsf{Q}\}}$$

Iteration statements. Iteration statements of the C-kernel language include only while statements. The induction principle is used for definition of iteration statements, i. e. it is supposed that the invariant property INV is true after each iteration of the iteration statement:

$$\frac{\{\mathsf{INV} \wedge \mathsf{cast(val(e,\ MD), type(e),\ int)} \neq 0\}\,\mathsf{S}\,\{\mathsf{INV}\}}{\{\mathsf{INV}\}\,\mathsf{while(e)}\,\mathsf{S}\,\{\mathsf{INV} \wedge \mathsf{cast(val(e,\ MD), type(e),\ int)} = 0\}}$$

Jump statements. Jump statements of the C-kernel language are restricted only by statements goto and return. The rule for a goto statement has the form:

$$\{\mathsf{Inv(L)}\}\,\mathsf{goto\,L;}\ \{\mathsf{false}\}$$

Semantics of a return statement is defined by two axioms for the statement with an argument and without it, respectively. The axiom for a return statement with an argument exits the current call of a function and transfers control to the point in which the postcondition of the function is true, returning the value in the metavariable Val:

$$\begin{array}{l}\{\mathsf{Post(f)(MD} \leftarrow \mathsf{upd(MD,\ (x_1,\ \ldots,\ x_m),\ (a_1,\ \ldots,\ a_m)))}\\ \quad \mathsf{(Val} \leftarrow \mathsf{cast(val(e,\ MD),\ type(e),\ }\tau))\}\\ /**\,\mathsf{rettype}(\tau)\,\mathsf{name(f)}\,\mathsf{autovar(x_1,\ \ldots,\ x_m)}\,\mathsf{autovarval(a_1,\ \ldots,\ a_m)}\ **/\\ \quad \mathsf{return\ e;}\ \{\mathsf{false}\}.\end{array}$$

The axiom for a return statement without an argument exits the current call of a function and transfers control to the point in which the postcondition of the function is true

$\{\mathsf{Post(f)(MD \leftarrow upd(MD, (x_1, \ldots, x_m), (a_1, \ldots, a_m)))}\}$
$/ * * \mathsf{autovar(x_1, \ldots, x_m)\, autovarval(a_1, \ldots, a_m)} * */ \quad \mathsf{return;} \{\mathsf{false}\}.$

7.5. Other rules

The consequence rule. The consequence rule has the form:

$$\frac{\mathsf{P \Rightarrow R} \qquad \{\mathsf{R}\}\,\mathsf{S}\,\{\mathsf{T}\} \qquad \mathsf{T \Rightarrow Q}}{\{\mathsf{P}\}\,\mathsf{S}\,\{\mathsf{Q}\}}$$

The sequence rule. Let T and $\mathsf{T'}$ be nonempty sequences of statements, declarations and auxiliary constructs. The sequence rule has the following form:

$$\frac{\{\mathsf{P}\}\,\mathsf{T}\,\{\mathsf{R}\} \qquad \{\mathsf{R}\}\,\mathsf{T'}\,\{\mathsf{Q}\}}{\{\mathsf{P}\}\,\mathsf{T} \quad \mathsf{T'}\,\{\mathsf{Q}\}}$$

Program. The rule for a program $\mathsf{Prgm(T)}$ consisting of the sequence T of declarations has the form:

$$\frac{\begin{array}{l} \{\mathsf{Pre(f)(MD \leftarrow upd(MD, (x_1, \ldots, x_m), (argv_1, \ldots, argv_n, a_{n+1}, \ldots, a_m)))}\} \\ \quad \mathsf{S} \quad \{\mathsf{Post(f)(MD \leftarrow upd(MD, (x_1, \ldots, x_m), (a_1, \ldots, a_m)))}\} \\ \text{for all declarations} \\ / * * \mathsf{name(f)\, autovar(x_1, \ldots, x_m)} \\ \qquad \mathsf{logtype(\tau_1 \times \ldots \times \tau_n \to \tau)\, autovarval(a_1, \ldots, a_m)} * */ \quad \mathsf{e}\,\{\mathsf{S}\} \\ \text{of functions defined in } \mathsf{T}, \text{ except the } \mathsf{main} \text{ function} \\ \{\mathsf{P}\}\,\mathsf{delPrgVar(y_1, \ldots, y_k)} \quad \mathsf{T} \quad \mathsf{S_{main}} \\ \quad \mathsf{delPrgVar(y_1, \ldots, y_k, z_1, \ldots, z_n)}\,\{\mathsf{Q}\} \end{array}}{\{\mathsf{P}\}\, / * \ \mathsf{prgvar(y_1, \ldots, y_k)\, mainpar(z_1, \ldots, z_l)} * / \quad \mathsf{Prgm(T)}\,\{\mathsf{Q}\}}$$

where $\{\mathsf{S_{main}}\}$ is a body of the main function. The parameters $\mathsf{z_1, \ldots, z_l}$ of the main function specify the program parameters. The auxiliary construct $\mathsf{delPrgVar}$ releases memory used by program variables:

$$\{\mathsf{Q(MD \leftarrow upd(MD, (y_1, \ldots, y_k), \bot))}\}\,\mathsf{delPrgVar(y_1, \ldots, y_k)}\,\{\mathsf{Q}\}$$

8. Conclusion

A three-stage method of C program verification has been presented. It is applied to C-light subset of the C language, that has formal operational semantics and covers a major part of C. At the first stage, a C-light program is brought to its normal form. This form is the result of some kind of static analysis that allows us to simplify the follow-up stages of verification. At the second stage, the normal form is translated into an intermediate language C-kernel in order to eliminate some C-light constructs difficult for axiomatic semantics, as well as to design axiomatic semantics in a more compact and transparent form. At the third stage, verification conditions are generated by means of the rules of C-kernel axiomatic semantics.

References

[1] Nepomniashy V.A., Anureev I.S., Promsky A.V. Verification-Oriented Language C-light and Its Structural Operational Semantics // Proc. of Conf. "Perspectives of System Informatics". – Lect. Notes Comput. Sci. – 2003 – Vol. 2890. – P. 1–5.

[2] Nepomniaschy V.A., Anureev I.S., Michailov I.N., Promsky A.V. Towards Verification of C Programs. C-Light Language and Its Formal Semantics // Programming and Computer Software. – 2002. – N 28(6). – P. 314–323.

[3] Nepomniaschy V.A., Anureev I.S., Promsky A.V. Towards Verification of C Programs. C-light and Its Transformation Semantics // Problems in Programming. – 2006. – N 2-3. – P. 359–368. (In Russian).

[4] Nepomniaschy V.A., Anureev I.S., Promsky A.V. Towards Verification of C Programs. Axiomatic Semantics of the C-kernel Language // Programming and Computer Software. – 2003. – N 29(6). – P. 338–350.

[5] Programming languages – C. – ISO/IEC 9899: 1999. – 1999. – 566 p.