Bull. Nov. Comp. Center, Comp. Science, 29 (2009), 89–105

The problems of C program verification*

A. V. Promsky

Abstract. The fully automatic verification of programs is a tempting and hardly accessible goal of modern programming. The low-level nature of the most popular programming languages, such as C and C+, has raised its difficulty to a new level. New formal methods and specification languages are required, because the classical Hoare approach and first-level logics are no more adequate for the task. This paper has two aims. First, it gives an overview of the two-level approach to verification of C-light programs and, second, it describes the successful application of this method to verification of so called "verification challenges". An interface with the theorem provers Simplify and Z3 is presented when discussing the proof of verification conditions.

1. Introduction

The project of C program verification is being developed in the IIS laboratory of theoretical programming. The C programming language remains one of the most widespread languages, so formalization of its semantics and automatic verification are the objects of great interest.

The two-level approach is an attempt to reconcile two controversial goals. First of all, a wide coverage of C is a requirement of practical interest. That is why C-light [4] includes a large part of the C language. The formal definition of C-light has the form of a structured operational semantics. Secondly, regardless of its limitations, the axiomatic approach [2] is still the best choice. As a result, we chose a compact core of C-light — the C-kernel language — and developed its Hoare-like logic. The first level of our approach consists in translation from C-light into C-kernel. The formally defined set of translation rules admits the proof of equivalence. At the second stage, the verification conditions are derived in C-kernel axiomatic semantics. The truth of these conditions guarantees the correctness of the source program w.r.t. its specifications.

In order to test our method, we verified some programs from the well-known collection [3] illustrating the verification challenges. Originally written in Java, they are represented in C with a simple rejection of object-oriented wrapping.

*This paper is supported by RFBR grant 09-01-00361-a "Automated program verification using SAT solvers".

The strategy of our project consists in generation of verification conditions in the representation independent of a prover. In case of simple verification examples, the lightweight provers, such as Simplify and Z3, can be used. However, to verify the "real" C programs, an interface to the industrial strength tools like PVS or Promela must be provided.

The overview of existing approaches to C program verification can be found in [9]. Among those works the CompCert project is worthy of a special attention. It uses a compiler front-end that translates the Clight[1] subset of the C language into the Cminor intermediate language and the formal semantics of Clight is mechanized using the Coq proof assistant.

The rest of the paper is organized as follows: Section 2 briefly describes the two-level verification method. Some strategies of verification condition simplification are proposed in Section 3. Section 4 presents the examples of verification challenges together with their verification. Section 5 is a conclusion of the paper.

2. The two-level verification method

Let us briefly describe the C-light verification project and present information required for reading the examples. The details can be found in [4, 5, 6].

2.1. Operational semantics of C-light

To formalize C-light, we use the method of "Structural Operational Semantics" in the Plotkin style. All language constructs are defined by the axioms and inference rules over the transition relation which binds the abstract machine configurations. Instead of classical variants of the program state, where a state simply maps the program variable names onto their possible values, we define the state as a function over metavariables. In their turn, the metavariables are higher level objects (functions and cartesian products) which model the addresses and types of program objects, the memory dump and the interim storage for the calculated expression values. Formally, a *state* of the C-light abstract machine is a map over the following metavariables:

1. the metavariable MeM (**Me**mory **M**anagement) which maps the names of usual program variables onto the addresses of the corresponding objects in the memory;
2. the metavariable MD (**M**emory **D**ump) which maps the program object addresses onto their values;
3. the metavariable TP which retrieves the types of the program objects from their addresses;

[1] Comparable to our C-light.

4. the metavariable STD which establishes the connection between type synonyms[2] and maps the tags onto the corresponding structure types[3];
5. the metavariable GLF (**G**lobal/**L**ocal **F**lag) which contains the current nesting level;
6. the metavariable Val which contains the value of the last evaluated expression.

The abstract machine also uses the set of abstract functions which perform some kinds of static analysis (such as expression type derivation) and encapsulate the low-level aspects of program behavior (for example, the allocation of new addresses).

Explicit use of addresses to access memory objects allows us to work with pointers as well as to handle the problem of aliasing which are always the challenges for verification. A typical rule of C-light semantics can be represented by the example of a simple assignment:

$$\frac{\begin{array}{ccccc}\sigma_0 \models e_1 : \mathrm{lv}[\tau_1] & & \tau_1 \text{ is not an array} & & \\ \langle e_2, \sigma_0\rangle \to_e^* \langle \epsilon, \sigma_1\rangle & \langle \&e_1, \sigma_1\rangle \to_e^* \langle \epsilon, \sigma_2\rangle & \mathrm{Val}_{\sigma_1} = (v, \tau_2) & & IC(\tau_1, \tau_2)\end{array}}{\langle e_1 = e_2, \sigma_0\rangle \to_e \langle \epsilon, \sigma_2''\rangle},$$

where $\sigma_2'' = \sigma_2(\mathrm{MD}(\mathrm{Val}_{\sigma_2} \leftarrow \gamma_{\tau_2,\tau_1}(v)))(\mathrm{Val} := (\gamma_{\tau_2,\tau_1}(v), \tau_1))$. So, the evaluation of the assignment expression is divided into the following steps: 1) by means of the type system, we make certain that e_1 is a modifiable l-value; 2) the right-hand part of the assignment is evaluated and the result, together with its type, is stored in the metavariable Val; 3) the left-hand side is evaluated, simultaneously evaluating the address of the object located by e_1; 4) the relation of implicit type coercion of both parts should be established; 5) the value of e_2 is the object of typecast γ_{τ_2,τ_1}; 6) this final value is the value of the whole assignment expression and is stored in memory location found during the step 3) (modification of the metavariable MD).

2.2. Translation from C-light into C-kernel

The idea of such a rewriting is to represent some undesirable C-light constructs by sequences of constructs from the smaller subset of C-light. The translation rules are divided into groups according to their purpose: the declaration rewriting, statement rewriting and expression rewriting. In some groups, the rules in their turn are divided into normalization rules, decomposition rules and elimination rules. In accordance with the notation, a normalization rule brings a construction to some canonical form, a decomposition rule breaks a complex sequence of constructions, and an elimination rule removes a "bad" construction.

[2]Thus, it handles the `typedef` declarations.

[3]We need it to resolve possible recursion in structure type definitions.

As an illustration, we can consider the elimination rule for the increment operator. According to it, an expression of the form `e++` should be replaced by the expression `(q = &e, y = *q, *q = *q + 1, y)`, where `q` and `y` are new variables declared to have the types of `&e` and `e`, respectively. This guarantees that the possible side-effects in `e` will take place only once. In its turn, the resulting sequence will be the object of an appropriate decomposition rule.

The main advantage is that C-kernel is still a part of C-light, so the same operational semantics can be involved to prove correctness of the translation rules.

2.3. Axiomatic semantics of C-kernel

Reasonable restrictions on C-light and its translation into C-kernel allow us to avoid many of problematic features of C which result in unsoundness and incompleteness of axiomatic semantics. We also use the higher-order assertion language instead of the classical predicate logic to specify the programs. That is why our C-kernel axiomatic semantics is simple enough. For example, the inference rule for the `while` loop looks like

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

The only difference with Hoare's classical rule is the involvement of metavariables and abstract functions for e which reflects the nature of the C language.

The resulting simplicity of C-kernel axiomatization makes the theoretical justification of soundness straightforward.

3. Verification condition simplification

The two-level access model resolves the problems of composite objects and aliasing but leads to explosion in the size of verification conditions (VC). Let us consider a simple example:

```
int x, y, z;
x = 1;
y = 2;
z = 3;
z = 4;
```

Indeed, the original C-light program is so simple that it could be verified in the classical Hoare logic [2]. However, in our approach we use the metavariables instead of program variables. Let us compare the VCs obtained in each approach.

In each case, we can use `true` as a precondition.

$$\left[\begin{array}{ll}
c_1 = \mathsf{naddr}(\mathrm{MD}_1) & \wedge \\
\mathrm{MeM}_2 = upd(\mathrm{MeM}_1, \mathtt{x}, c_1) & \wedge \\
\mathrm{TP}_2 = upd(\mathrm{TP}_1, c_1, \mathtt{int}) & \wedge \\
\mathrm{MD}_2 = upd(\mathrm{MD}_1, c_1, \omega) & \wedge \\
c_2 = \mathsf{naddr}(\mathrm{MD}_2) & \wedge \\
\mathrm{MeM}_3 = upd(\mathrm{MeM}_2, \mathtt{y}, c_2) & \wedge \\
\mathrm{TP}_3 = upd(\mathrm{TP}_2, c_2, \mathtt{int}) & \wedge \\
\mathrm{MD}_3 = upd(\mathrm{MD}_2, c_2, \omega) & \wedge \\
c_3 = \mathsf{naddr}(\mathrm{MD}_3) & \wedge \\
\mathrm{MeM} = upd(\mathrm{MeM}_3, \mathtt{z}, c_3) & \wedge \\
\mathrm{TP} = upd(\mathrm{TP}_3, c_3, \mathtt{int}) & \wedge \\
\mathrm{MD}_4 = upd(\mathrm{MD}_3, c_3, \omega) & \wedge \\
\mathrm{MD} = upd(\mathrm{MD}_4, \mathrm{MeM}(\mathtt{x}), 1) & \wedge \\
V_2 = upd(V_1, L(\mathtt{x}), 1) & \wedge \\
V_3 = upd(V_2, L(\mathtt{y}), 2) & \wedge \\
V_4 = upd(V_3, L(\mathtt{z}), 3) & \wedge \\
V = upd(V_4, L(\mathtt{z}), 4) &
\end{array}\right] \Longrightarrow \left[\begin{array}{ll}
V(L(\mathtt{x})) = 1 & \wedge \\
V(L(\mathtt{y})) = 2 & \wedge \\
V(L(\mathtt{z})) = 4 &
\end{array}\right]$$

Figure 1. The verification condition

The postcondition for the classical logic is $\mathtt{x} = 1 \wedge \mathtt{y} = 2 \wedge \mathtt{z} = 4$. Applying the Hoare assignment rule 4 times, we obtain the true assertion:

$$\mathsf{true} \Longrightarrow (1 = 1 \wedge 2 = 2 \wedge 3 = 3). \tag{1}$$

The two-level access in the C-kernel logic immediately complicates the precondition: $\mathrm{MD}(\mathrm{MeM}(\mathtt{x})) = 1 \wedge \mathrm{MD}(\mathrm{MeM}(\mathtt{y})) = 2 \wedge \mathrm{MD}(\mathrm{MeM}(\mathtt{z})) = 4$.

It should be noted that the only difference between the equivalent C-kernel program and the source file concerns the first string. According to the C-kernel restrictions, the joint declaration should be split into three separate declarations.

The verification condition is presented in Fig. 1. Its size surprises. Obviously, verification of real programs can lead to immense assertions which can easily overcome the capacity of theorem provers. That is the price of the detailed memory model of C-light/kernel. Thus, we need some simplification strategies which must precede the proof stage.

Strategy 1. In the consequent of VC, the variables are matched against the constants. So, we can split VC into three formulas in accordance with the number of conjuncts in the consequent. Let us consider one of these new formulas (strategy 1 is supposed to be already applied):

$$\left[\begin{array}{ll} newp(d_3, L_3, V_1, L2) & \wedge \\ newp(d_2, L_2, V_1, L2) & \wedge \\ newp(d_1, L_1, V_1, L2) & \wedge \\ L_2 = upd(L_1, \mathtt{x}, d_1) & \wedge \\ L_3 = upd(L_2, \mathtt{y}, d_2) & \wedge \\ L = upd(L_3, \mathtt{z}, d_3) & \wedge \\ V_2 = upd(V_1, L(\mathtt{x}), 1) & \wedge \\ V_3 = upd(V_2, L(\mathtt{y}), 2) & \wedge \\ V_4 = upd(V_3, L(\mathtt{z}), 3) & \wedge \\ V = upd(V_4, L(\mathtt{z}), 4) & \end{array}\right] \Longrightarrow V(L(\mathtt{z})) = 4. \quad (2)$$

Strategy 2. The type `int` is not a reference type. This guarantees that the object `z` does not have aliases, i.e. the assignments to `x` and `y` affect `z` only if the expressions over `x` and `y` are assigned to `z`. Let us begin to unfold the antecedent from the rightmost conjunct. If a term *upd* binds `z` with a value which is independent of `x` and `y`, then this *upd* can be substituted into the consequent. Otherwise, the *upd* is ignored. This unfolding takes into account the fact that the lower (righter) conjunct corresponds to the later assignment in the source program.

$$newp(d_3, L_3, V_1, L2) \wedge newp(d_2, L_2, V_1, L2) \wedge newp(d_1, L_1, V_1, L2) \Longrightarrow \\ upd(V_4, upd(L_3, \mathtt{z}, d_3)(\mathtt{z}), 4)(upd(L_3, \mathtt{z}, d_3)(\mathtt{z})) = 4. \quad (3)$$

The standard semantics of *upd* provides the truth of this assertion. Though (3) is still bigger than (1), it is significantly simpler than VC from Figure 1. Moreover, the proof does not depend on the fact that d_1, d_2 and d_3 are new addresses, so the whole antecedent can be discarded.

In the next Section, we will demonstrate the VCs after the application of simplification strategies.

4. Verification examples

Our examples are borrowed from [3]. Naturally, the Java-source programs have been translated in their semantic equivalents in the C language. Despite their small size, they represent a problem for verification in the classical Hoare logic.

4.1. Aliasing

This example shows the concept of multiple access to a memory object. In principle, in Java the problem is more acute because the syntax of a reference is not different from a conventional identifier syntax. In C references are implemented through pointers and the dereferencing operation should signal potential problems. In any case, references and pointers make the

classic method of expression interpretation inconsistent. Instead of a classical model $name \rightarrow value$, we have to use $name \rightarrow address \rightarrow value$ model.

```
#include "stdio.h"

struct C {
    struct C *a;
    int i;
};

int m(void) {
    struct C c;
    c.a = &c;
    c.i = 2;
    return c.i + (c.a)->i;
};

int main(void){
    printf("%d", m());
    return 0;
}
```

The return expression of the function `m` references the value of the field `i` of the structure `c` value via an aliased reference to itself in the field `a`.

Note. The standard library `stdio` is used here only for illustrative purpose. In practice, verification of the console output is beyond the scope of our interests. Indeed, how often do the programmers take into account the fact that `printf` is a function returning an integer value which can be matched against the program specification?

Specifications for the function `m`[4] have the form[5]:

$$\begin{array}{lll} \text{Pre}(\texttt{m}) & : & \mathsf{true} \\ \text{Post}(\texttt{m}) & : & \mathsf{Val} = 4 \end{array}$$

The source program is simple enough to satisfy the definition of C-kernel, so, the intermediate program is not different.

The body of `m` forms the single linear area. The axiomatic semantics produces the following verification condition:

[4]Verification of the function `main` is not of much interest.

[5]As a rule, we use the adaptations of corresponding annotations from [3].

$$\left[\begin{array}{l} c_1 = \mathsf{naddr}(\mathsf{MD}_0) \wedge \\ \mathsf{MD}_1 = upd(\mathsf{MD}_0, c1, \omega) \wedge \\ \mathsf{MD}_2 = upd(\mathsf{MD}_1, c1, c1) \wedge \\ \mathsf{MD}_3 = upd(\mathsf{MD}_2, \mathsf{c}, c1) \wedge \\ \mathsf{MD}_4 = upd(\mathsf{MD}_3, \mathsf{mb}(c1, \mathsf{a}), \mathsf{null}) \wedge \\ \mathsf{MD}_5 = upd(\mathsf{MD}_4, \mathsf{mb}(c1, \mathtt{i}), 0) \wedge \\ \mathsf{MD}_6 = upd(\mathsf{MD}_5, \mathsf{mb}(c1, \mathsf{a}), \mathsf{val}(\&\mathsf{c}, \mathsf{MD}_6)) \wedge \\ \mathsf{MD}_7 = upd(\mathsf{MD}_6, \mathsf{mb}(c1, \mathtt{i}), 2) \wedge \\ \mathsf{Val} = \mathsf{BinOpSem}(+, \mathsf{MD}_7(\mathsf{mb}(c1, \mathtt{i})), \mathsf{int}, \mathsf{val}(\texttt{(c.a)->i}), \texttt{int}) \end{array}\right] \Rightarrow \mathsf{Val} = 4.$$

To prove this assertion, the automatic theorem prover Simplify was used. First, it required the axioms for some logical functions such as *upd*, array access *get*, left-values handling, etc.

```
(BG_PUSH (FORALL (a i x)
  (EQ (get (upd a i x) i) x)
))

(BG_PUSH (FORALL (a i x)
  (EQ (upd a i (get a i)) a)
))
(BG_PUSH (FORALL (a i x j)
  (OR
    (EQ i j)
    (EQ
      (get (upd a i x) j)
      (get a j)
    )
  )
))
(BG_PUSH (FORALL (v c)
  (EQ
    (val (lv v c))
    v
  )
))
(BG_PUSH (FORALL (v c)
  (EQ
    (loc (lv v c))
    c
  )
))
...
```

Second, the verification condition was presented in the form acceptable for the provers Simplify and Z3. Because of lack of a space, let us consider only its part:

```
(IMPLIES
  (AND
    (DISTINCT i a)
    (DISTINCT c_1 c_2)
    (EQ
      MD
      (upd MD1_5 (get MeM i)
        (+ (get MD1_5 (get MeM i)) 2)
      )
    )
    (EQ
      MD1_5
      (upd
        MD1_4
        (get MD1_4 (get MeM a))
        (+ (get MD1_4 (get MD1_4 (get MeM a))) 2)
      )
    )
    (EQ MD1_4 (upd MD1_3 (get MeM a) (get MeM i)))
    (EQ (get MD1_2 c_2) |@undef|)
    (EQ MD2_2 MD1_3)
    (EQ MeM (upd MeM1_3 a c_2))
    (EQ MeM1_3 (upd MeM1_2 i c_1))
    (EQ MD1_3 (upd MD1_2 c_2 0))
    (EQ (get MD1_1 c_1) |@undef|)
    (EQ MD2_1 MD1_2)
    (EQ MD1_2 (upd MD2_1 c_1 0))
    (EQ
      MeM1_2
      (upd (upd MeM1_1 i |@undef|) a |@undef|)
    )
  )
  (EQ val 4)
)
```

Here the member access function *mb* is already rewritten using the address function MeM. The condition was automatically proven.

In the following sections, we omit the detailed presentation of verification conditions focusing mainly on the challenges themselves.

4.2. Breaking out of a loop

The exit from a loop without a loop condition check is usually performed via a `break` statement (rarely, via `goto`). This complicates the underlying control flow semantics.

As you know, the programming theorists consider the `goto` statement harmful. However, for the Hoare logic, this statement is far better than the statements `break`, `continue` or `return`. The treatment of `goto` in the Hoare logic is based on the notion of the label invariant. The classical axiom for `goto` is

$$\{Inv(\texttt{l})\}\,\texttt{goto l}\,\{\mathsf{false}\}\ ,$$

where $Inv(\texttt{l})$ is an assertion which is supposed to be true every time the control reaches label `l`. The false postcondition means that the statement directly after `goto l` is unreachable as long as the control is transferred to another point.

However, there is no label for the `break` statement, so we cannot propose an assertion as a precondition. Obviously, the semantics of `break` cannot be defined without semantics of the enveloping loop statement (or `switch`). This will complicate the semantics of both statements[6].

Instead, we replace `break` by a jump to a new label just after the loop body, so the reliable `goto` semantics can be applied.

The source C-light program looks like

```
int ia[] = {2, 3, 4, 5, -2, 4, 7};

void NegateFirst(int ia[], int Length) {
    int i;
    for (i = 0; i < Length; i++) {
        if (ia[i] < 0) {
            ia[i] = -ia[i];
            break;
        }
    }
}

int main(void){
    int Length = sizeof(ia)/sizeof(ia[0]), i;
    NegateFirst(ia, Length);
    return 0;
}
```

When the first negative element is reached, its sign changes and the loop aborts.

[6]The situation with `continue` and `return` is the same.

In [3] the authors vaguely described the reason why the corresponding Java program was not verified in ESC/Java. In fact, the unconditional exit from the loop together with a possible array modification leads to complex specifications:

$$\mathrm{Pre}(\texttt{NF()}) \;:\; \exists old : \texttt{int[]}.\ \mathsf{MD}(\texttt{ia}) \neq \mathsf{null} \wedge \mathsf{MD}(\texttt{ia}) = \mathsf{MD}(old)$$

$$\begin{aligned}\mathrm{Post}(\texttt{NF()}) \;:\; \forall i.\ (0 \leq i \leq \mathsf{MD}(\texttt{Length}) \Longrightarrow \\ ((\mathsf{MD}(\mathsf{mb}(old, i)) < 0 \wedge (\forall j.\ 0 \leq j < i \Rightarrow \\ \mathsf{MD}(\mathsf{mb}(old, j)) \geq 0)) \Rightarrow \\ \mathsf{MD}(\mathsf{mb}(\texttt{ia}, i)) = -\mathsf{MD}(\mathsf{mb}(old, i)) \wedge \\ old[i] \geq 0 \Rightarrow \mathsf{MD}(\mathsf{mb}(\texttt{ia}, i)) = \mathsf{MD}(\mathsf{mb}(old, i)))\end{aligned}$$

$$\begin{aligned}\mathrm{Inv}(\texttt{for}) \;:\; 0 \leq \mathsf{MD}(\texttt{i}) \leq \mathsf{MD}(\texttt{Length}) \wedge \\ (\forall j.\ 0 \leq j < \mathsf{MD}(\texttt{i}) \Rightarrow \\ (\mathsf{MD}(\mathsf{mb}(\texttt{ia}, j)) \geq 0 \wedge \\ \mathsf{MD}(\mathsf{mb}(\texttt{ia}, j)) = \mathsf{MD}(\mathsf{mb}(old, j)))\ .\end{aligned}$$

The original array content is stored in an auxiliary variable *old*.

The intermediate C-kernel program is as follows[7]:

```
static int ia[] = {2, 3, 4, 5, -2, 4, -7};

void NegateFirst(int ia[], int Length) {

    auto int i;
    i=0;
    while(i < Length)
    {
        if (ia[i]<0)
        {
            ia[i] = -ia[i];
            goto L;
        }
        auto int* q1;
        q1 = &i;
        *q1 = *q1 + 1;
    }
    L:;
}

int main(void){
```

[7] From here we omit the console output.

```
    auto int Length = sizeof(ia)/sizeof(ia[0]);
    auto int i;
    NegateFirst(ia, Length);
    return 0;
}
```

Pay your attention to the replacement of `i++` by manipulations over the new pointer `q1`. Though such a translation seems redundant, when the increment operation forms the complete expression statement, it is absolutely required, when the value of `i++` is used[8]. Moreover, this program is slightly optimized. In a general case, the value of `i++` should be kept in an additional variable.

Let us consider the verification conditions. The first condition corresponds to the linear area from the beginning of `NegateFirst` to the entry point of the loop, i.e. it establishes connection between the precondition and the loop invariant:

$$Pre(MD)(MD \leftarrow MD1) \wedge MD = upd(MD1, i, 0) \Rightarrow INV(MD)$$

As long as the Simplify prover natively supports arithmetics and arrays, the proof of the verification conditions is straightforward.

4.3. The side-effects and logical operators

The main feature of the logical AND/OR operators in C is that evaluation of the rightmost argument can be omitted depending on the value of the first subexpression. This phenomenon must be accurately modeled in the semantics. Such a modeling is simple in operational semantics but it is a real challenge for the Hoare logic. The problem is intensified by possible side-effects in the arguments. Thus, a complete rewriting of those operators is inevitable. Consider the following program:

```
#include "stdbool.h"

_Bool b = true;
_Bool result1, result2;

_Bool f() {
    b = !b;
    return b;
}

int main(void){
    result1 = f() || !f();
```

[8]For example, `i++` as the controlling expression of a loop.

```
    result2 = !f() && f();
    return 0;
}
```

The result is surprising for the logician: $f() \vee \neg f() = \mathsf{false}$ and $\neg f() \wedge f() = \mathsf{true}$.

To get rid of logical operators, we use an ordinary `if` statement. The C-kernel program looks like

```
static _Bool b = 1;
static _Bool result1;
static _Bool result2;

_Bool f() {
    auto _Bool x1;
    x1 = b;
    if(x1) { b = 0; } else { b = 1; }
    return b;
}

int main(void){
    auto _Bool x1;
    x1 = f();
    if(x1) { result1 = 1; }
    else
    {
        auto _Bool x2;
        x2 = f();
        if(x2) { result1 = 0; } else { result1 = 1; }
    }
    auto _Bool x3;
    auto _Bool x4;
    x4 = f();
    if(x4) { x3 = 0; } else { x3 = 1; }
    if(x3) { result2 = f(); } else { result2 = 0; }
    return 0;
}
```

Note that we actively use auxiliary variables x_i to store the intermediate values. Our method includes an implicit preprocessing stage, so the integer value 1 is used instead of the logical constant `true`.

Every occurrence of the conditional statement doubles the number of verification conditions. The declarations of variables increase their length. Let us omit them. It is sufficient to note that all of them proved true. The details can be found in [8, 9].

4.4. The function pointers

The problem with function pointers in the C language does not differ from the challenge of virtual functions in Java [3] or delegates in C#. And again, the reason is the limited power of Hoare logic. Since the axiomatic semantics in fact represents the symbolic execution, it works well with the static information but cannot handle all aspects of dynamic behavior. A function pointer (virtual function/delegate) represents an interface to the set of functions. In a general case, the Hoare logic cannot recognize which function is actually invoked. So the quantifier over all appropriate functions should be used to specify the generic functions.

It should be noted that the program below was not a part of the Java program collection.

```
#include "stdio.h"

typedef _Bool (* Order)(int e1, int e2);

int Find(int arr[], int Length, Order ord){
        int min = arr[0];
        int xx = 0;
        while(xx < Length){
            if(ord(arr[xx], min)) min = arr[xx];
            xx++;
        }
        return min;
}

_Bool LessThan(int e1, int e2) { return e1 < e2; }

int main(void){
    int arr[] = {3, 5, 1, 7, 4}, found;
    found = Find(arr, sizeof(arr)/sizeof(arr[0]), LessThan);
    printf("%d\n", found);
    return 0;
}
```

After the translation, we have the following:

```
typedef _Bool (* Order)(int e1, int e2);

int Find(int arr[], int Length, Order ord){
        auto int min = arr[0];
        auto int xx = 0;
```

```
        while(1){
            auto int x1;
            x1 = xx < Length;
            if(x1){} else { break; }
            auto int x2;
            auto int x3;
            x3 = arr[xx];
            x2 = ord(x3, min);
            if(x2) { min = arr[xx]; } else {};
            auto int* q1;
            auto int y1;
            q1 = &xx;
            y1 = *q1;
            *q1 = *q1 + 1;
            y1;
        }
        return min;
}

_Bool LessThan(int e1, int e2) {
    auto int x1;
    x1 = e1 < e2;
    return x1;
}

int main(void){
    auto int arr[] = {3, 5, 1, 7, 4};
    auto int found;
    auto int x1;
    x1 = sizeof(arr)/sizeof(arr[0]);
    found = Find(arr, x1, LessThan);
    return 0;
}
```

The verification process for this example involves quantification over the functions possibly pointed at by `Order`. Let us omit those cumbersome verification conditions which were inferred by the Hoare logic. The details of their proof can be found in [8, 9].

5. Conclusion

In this paper, we have described the two-level method of C-light program verification and how it can handle some problematic features of procedural

programming languages. The advantages of C-light and of the approach are as follows:

- The C-light language covers the major part of C99.
- It has a complete formal semantics in the Plotkin style.
- The verification process is based on a simple Hoare-like logic. The simplicity results from translation of semantically difficult C-light constructs into C-kernel.
- The formal operational semantics of C-light was used to guarantee the soundness of axiomatic semantics and the correctness of translation rules.

The two-level method seems to be promising for applications. The development of an automatic verification tool will form the framework of future activity. Possible applicability of our approach to verification of C descendants (such as C++, Java, C#) also is worth mentioning. Another interesting research is modification of the Hoare logic to simplify verification conditions at the generation stage [7].

References

[1] Blazy S., Dargaye Z., Leroy X. Formal verification of a C compiler front-end // Proc. FM 2006: 14th Int. Symp. on FormalMethods. — Lect. Notes Comput. Sci. — 2006. — Vol. 4085. — P. 460-475.

[2] Hoare C.A.R. An axiomatic basis for computer programming // Communs. ACM. — 1969. — Vol. 12, N 1. — P. 576–580.

[3] Jacobs B., Kiniry J.L., Warnier M. Java program verification challenges // Proc. FMCO 2002. — Lect. Notes Comput. Sci. — 2003. — Vol. 2852. — P. 202–219.

[4] Nepomniaschy V.A., Anureev I.S., Mihailov I.N., Promsky A.V. Towards verification of C programs. C-Light language and its formal semantics // Programming and Computer Software. — 2002. — Vol. 28(6). — P. 314–323.

[5] 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. — Vol. 29(6). — P. 338–350.

[6] Nepomniaschy 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.

[7] Maryasov I.V. towards automatic verification of C-light programs. Mixed axiomatic semantics of C-kernel language // Proc. Internat. Workshop on Program Understanding, 19–23 June, Altai Mountains, Russia, 2009. — P. 44–52.

[8] Promsky A.V. The C#-light project: solution of some verification challenges // Bull. Novosibirsk Comp. Center. Ser.: Comput. Sci. — 2007. — Iss. 26. — P. 111–132.

[9] Promsky A.V. Towards C-light program verification: Overcoming the obstacles // Proc. Internat. Workshop on Program Understanding, 19–23 June, Altai Mountains, Russia, 2009. — P. 53–63.