Independent Study: Tool-Assisted Verification of C Code using Floyd-Hoare Logic

Nikolas Mählmann

April 9, 2020

Contents

1 Introduction

This document explains the usage and inner workings of `verify-c`. `verify-c` is a command line tool which verifies programs written in a subset of C. The tool is based on Floyd-Hoare logic, which was intensively discussed in the course *Korrekte Software: Grundlagen und Methoden* at the University of Bremen held by Serge Autexier and Christoph Lüth in the summer semester 2019. `verify-c` is an educational implementation of the discussed techniques aimed to deepen their understanding and explore which challenges arise when formally verifying software. `verify-c` is written in Haskell and the source code is available online at `https://github.com/nmaehlmann/verify-c`.

2 Installation

`verify-c` can be built using the Haskell build tool `stack` by calling:

```
stack install
```

in the root directory. Additionally `verify-c` relies on the `Z3` theorem prover which has to be installed and added to the `PATH` variable. It can be downloaded at `https://github.com/Z3Prover/z3`.

3 Usage

`verify-c` parses program code written in a subset of C. Each function is annotated with logical pre- and postconditions, which specify the contract of the function. Based on the parsed program a set of verification conditions is generated. The verification conditions are exported to a theorem prover, which checks whether or not they are satisfied. If all verification conditions are proven successfully, the implemented functions satisfy their contract.

Let's start with a simple example. Listing 1 shows a verified implementation of the faculty function.

Listing 1: faculty.c0

```
 1  int faculty(int n){
 2      precondition("n >= 0");
 3      postcondition("\result == fac(n)");
 4  
 5      p = 1;
 6      c = 1;
 7      while(c <= n){
 8          invariant("p == fac(c - 1) && c <= n + 1 && c > 0");
 9          p = p * c;
10          c = c + 1;
11      }
12      return p;
13  }
```

It is written in regular C code but additional function calls have been added to specify the contract of the function in order to verify it. The precondition in l.2 states that the function argument `n` has to be positive. The postcondition in l.3 that the function will return `fac` of `n`. When verified successfully these conditions have the following semantic: If the function `faculty` is called with a positive argument `n` and it terminates, then the return value of this function will equal `fac` of `n`. Furthermore the while loop is annotated with an invariant (l.8). The invariant has to be satisfied before the while loop is entered as well as after each iteration of the loop. The specification of preconditions, postconditions, and invariants is mandatory and missing specifications will result in a parser error.

Additionally to the C source code, `verify-c` requires an environment file if custom functions or predicates are used in specifications. The function `fac` used in the precondition and in the invariant is such a custom function. It is specified in listing 2.

Listing 2: faculty.env

```
(declare-fun fac (Int) Int)
(assert (= (fac 0) 1))
(assert (forall ((nn Int)) (implies (< 0 nn) (= (fac nn) (* nn
    ↪ (fac (- nn 1)))))))
```

The specification of the environment is written in the SMT-LIB format and verbatim fed into the `Z3` prover. More information regarding the SMT-LIB language can be found online at `http://smtlib.cs.uiowa.edu/language.shtml`. In order to be found by `verify-c`, the environment has to have the same name as the source file but with an `.env` extension. With the environment in place the source code of the faculty function can now be verified by calling:

```
verify-c faculty.c0
```

which produces the following output:

```
Generated 3 verification condition(s). Starting proof:
[1/3] : Precondition faculty : OK
[2/3] : While Case True (l:8) : OK
[3/3] : While Case False (l:8) : OK

Summary: VERIFICATION OK
```

Three verification conditions were generated by `verify-c` and successfully proven by `Z3`. One originates from the precondition of the faculty function, two from the invariant of the while loop.
In this case every verification condition could be proven, which is indicated by the status code `OK`. The other possible status codes are:

- `SIMPLIFY FAILED`: The verification conditions could not be simplified enough to be proven. This is most likely caused by ambiguous dereferencing.

- `SMT EXPORT FAILED`: The verification condition could not be translated into SMT-LIB code. This is most likely caused by ambiguous referencing.

- `VIOLATED`: The verification condition was disproven. The specification and program do not match.

- `TIMEOUT`: `Z3` timed out while trying to prove the verification condition. It could neither disprove nor prove it.

- `SMT ERROR`: `Z3` produced an unkown error.

- `SKIPPED`: The verification condition was skipped because of a previous error.

The generated verification conditions and SMT-LIB code as well as logfiles are stored in the `.\target` folder created by `verify-c`. This is the place to look at in case verification fails.

`verify-c` can be further configured by using command line options. A list of all available can be displayed by calling:

```
verify-c -h
```

which outputs:

```
  Help Options:
  -h, --help
    Show option summary.
  --help-all
    Show all help options.

Application Options:
  --color :: bool
    Whether or not to use ANSI colors.
    default: false
  --timeout :: int
    SMT solver timeout in seconds.
    default: 5
  --no-skip :: bool
    Whether or not to continue verification after a condition
       ↪  could not be
    verified.
    default: false
```

4 Implementation

`verify-c` is written in Haskell and the source code is available online at `https://github.com/nmaehlmann/verify-c`.

4.1 Parsing

Parsing of the source code is done using the parser combinators library `parsec`. This is a standard procedure so this document will not go into further details about the parsing process. The result of the parsing process is an Abstract Syntax Tree (AST) which is annotated with first order logic formulas.

4.2 Logical Formulas

Logical formulas are the core data structure on which most of the verification logic operates. They are implemented by the GADT `BExp` shown in listing 3.

Listing 3: BExp

```
data BExp l m where
    BTrue :: BExp l m
    BFalse :: BExp l m
    BNeg :: BExp l m -> BExp l m
    BBinExp :: BBinOp -> BExp l m -> BExp l m -> BExp l m
    BComp :: CompOp -> AExp l m -> AExp l m -> BExp l m
    BForall :: Idt -> BExp FO m -> BExp FO m
    BExists :: Idt -> BExp FO m -> BExp FO m
    BPredicate :: Idt -> [AExp FO m] -> BExp FO m
```

The `BExp` type is parameterized by two arguments `l` and `m`.
The first argument `l` characterizes the type of logic that is used. It can take two values:

1. `CO`: the logical operations that can be used as a part of the C programming language, for example to formulate the condition of a while loop

2. `FO`: first order logic which is used to specify preconditions, postconditions and invariants

While the `true` and `false` constants, negations, boolean operators, and comparisons are available in every supported type of logic, quantifiers and predicates are only available in first order logic. This is guaranteed by the type system through the usage of GADTs. Since `BExp CO m` is a subset of `BExp`

FO m, the first is converted to the latter during the actual generation of verification conditions.
The second argument of BExp is m which characterizes the memory model that is used. It can also take two values:

1. Plain: a user facing symbolic memory model that is used during the development of the program and the specification

2. Refs: an axiomatic memory model which is used internally during verification condition generation

4.3 Arithmetic Expressions

Arithmetic expressions are expressions which evaluate to an integer. They are used on the right hand side of assignments, as array indices or as a part of a comparison operation in a logical formula. Arithmetic expressions are modelled by the AExp GADT which is shown in listing 4.

Listing 4: AExp

```
data AExp l m where
    ALit :: Integer -> AExp l m
    AIdt :: LExp l m -> AExp l m
    ABinExp :: ABinOp -> AExp l m -> AExp l m -> AExp l m
    AFunCall :: Idt -> [AExp FO m] -> AExp FO m
    ALogVar :: Idt -> AExp FO m
    AAddress :: LExp l Plain -> AExp l Plain
```

As parts of BExps they are also parameterized with the type of logic and memory model. Integer literals, variable names, and binary calculations are supported for every type of logic and memory model. Logical variables and function calls as parts of an AExp require first order expressions, so they can only be used in for specification purposes. As part of the C program code function calls do not form an AExp but are treated as a separate statement. This limitation is further explained in section 4.7. The address operator (&) is transformed into an LExpression in the Refs memory model so it is only available in the Plain memory model.

4.4 LExpressions

LExpressions are expressions which can be used on the left hand side of assignments or as variable identifiers as parts of arithmetic expressions. LExpressions are modelled by the `LExp` GADT which is shown in listing 5.

Listing 5: LExp

```
data LExp l m where
    LIdt :: Idt -> LExp l m
    LArray :: LExp l m -> AExp l m -> LExp l m
    LStructurePart :: LExp l m -> Idt -> LExp l m
    LRead :: State -> LExp l Refs -> LExp l Refs
    LDeref :: LExp l Plain -> LExp l Plain
```

As parts of `AExp`s they are also parameterized with the type of logic and memory model. Identifiers and array and struct accessors are available for every type of logic and memory model. Similar to the address operator, the dereferencing operator (`*`) is only available in the `Plain` memory model. The `LRead` `LExp` is the core of the `Refs` memory model which will be explained in more detail in the next section.

4.5 Memory Models

In the symbolic memory model `C0`, each LExpression is assigned a value. To verify a program using references however, it is necessary to transform the symbolic model into an axiomatic one. In the axiomatic model `Refs`, each LExpression (except `LRead`) is assigned a memory address. The actual value is obtained by looking up the memory address in a program state:

$$read(\sigma, l)$$

Reads are modelled by the `LRead` LExpression. Assigning a value to an address creates an updated state:

$$\sigma_2 = update(\sigma_1, l, v)$$

This is modelled by the `State` type shown in listing 6.

Listing 6: State

```
data State
    = Atomic String
    | Update State (LExp FO Refs) (AExp FO Refs)
```

Using this axiomatic model, LExpressions, Referencing and Dereferencing can be treated uniformly:

$$\texttt{a} \mathrel{\hat{=}} \mathit{read}(\sigma, a)$$

$$\texttt{\&a} \mathrel{\hat{=}} a$$

$$\texttt{*a} \mathrel{\hat{=}} \mathit{read}(\sigma, \mathit{read}(\sigma, a))$$

Often programs contain multiple assignments which leads to deeply nested states, for example:

$$\mathit{update}(\mathit{update}(\mathit{update}(\sigma, l_1, v_1), l_2, v_2), l_3, v_3)$$

The situation gets worse, when references are involved because dereferencing an LExpression doubles the amount of states. To keep the states small the following simplification rules are introduced:

$$l_1 = l_2 \Rightarrow \mathit{update}(\mathit{update}(\sigma, l_2, v_2), l_1, v_1) = \mathit{update}(\sigma, l_1, v_1)$$

$$l_1 = l_2 \Rightarrow \mathit{read}(\mathit{update}(\sigma, l_2, v), l_1) = v$$

$$l_1 \neq l_2 \Rightarrow \mathit{read}(\mathit{update}(\sigma, l_2, v), l_1) = \mathit{read}(\sigma, l_1, v)$$

To apply these simplifications it is crucial to decide whether or not two LExpressions are equal. This however is not always possible. Two references might point to the same address, or two array indices might have the same value:

$$\texttt{*a} \overset{?}{=} \texttt{*b}$$

$$\texttt{a[i]} \overset{?}{=} \texttt{a[j]}$$

The following heuristic comparison algorithm was implemented in the module `Memory.Eq`:

$$\begin{aligned}
cmp(a,a) =& Eq \\
cmp(a,b) =& NotEq && \text{if } a \neq b \text{ is predefined} \\
cmp(a,b) =& NotEq && \text{if } a \text{ was just initialized} \\
cmp(a,b) =& NotEq && \text{if } b \text{ was just initialized} \\
cmp(read(\sigma,a), read(\sigma,a)) =& cmp(a,b) \\
cmp(read(\sigma,a), b) =& Undecidable \\
cmp(a, read(\sigma,b)) =& Undecidable \\
cmp(a.i, b.j) =& cmp(a,b) && \text{if } i = j \\
cmp(a.i, b.j) =& NotEq && \text{if } i \neq j \\
cmp(a[i], b[j]) =& Eq && \text{if } cmp(a,b) = Eq \wedge cmpA(i,j) = Eq \\
cmp(a[i], b[j]) =& NotEq && \text{if } cmp(a,b) = NotEq \vee cmpA(i,j) = NotEq \\
cmp(a[i], b[j]) =& Undecidable && \text{otherwise} \\
cmp(a,b) =& NotEq && \text{otherwise}
\end{aligned}$$

$$\begin{aligned}
cmpA(a,a) =& Eq \\
cmpA(a,b) =& NotEq && \text{if } a, b \text{ are both literals} \\
cmpA(a,b) =& Eq && \text{if } a, b \text{ are both identifiers} \wedge cmp(a,b) = Eq \\
cmpA(a,b) =& Undecidable && \text{otherwise}
\end{aligned}$$

cmp compares two LExpressions and should return *Eq* if two LExpressions evaluate to the same memory address. *cmpA* is used to compare array indices and should return *Eq* if two arithmetic expressions evaluate to the same integer. $a \neq b$ is a predefined inequality in the right hand side of an implication if $a \neq b$ is true in the left hand side of that implication. a was just initialized if the previous statement is the declaration of a. In this case it is assumed, that the operating system assigned a fresh memory address to a. Both, the predefined inequalities and the set of just initialized identifiers have to be passed to the functions as a context.

4.6 Simplification

With the simplification rules and comparison functions in place the simplification algorithm can be implemented. Applying one simplification rule to a logical formula can lead to the opportunity to apply another one, leading to an expression collapsing step by step. Therefore the simplification algorithm has to run repeatedly until no further simplifications are possible. To conveniently keep track whether a simplification has happened, or the result remained unchanged a custom `Updated` monad shown in 7 is introduced:

Listing 7: the Updated monad

```
data Updated a = Updated a | Unchanged a

instance Monad Updated where
   return a = Unchanged a
   (Updated a) >>= f = Updated $ unwrap $ f a
   (Unchanged a) >>= f = f a

unwrap :: Updated a -> a
unwrap (Updated a) = a
unwrap (Unchanged a) = a
```

If an `Updated` value is composed, the result is also `Updated`. As previously presented, the comparison algorithm for LExpressions requires a context in which predefined inequalities and local variables can be looked up. This context is made accessible by wrapping the `Updated` monad into a Reader monad, which carries the required information. The obtained nested monad is aliased as `Simplified` and presented in listing 8.

Listing 8: the Simplified monad

```
type Simplified = ReaderT SimplificationCtx Updated

data SimplificationCtx = SimplificationCtx
   { inequalities :: Set Inequality
   , localVars :: Set (LExp FO Refs)
   }

type Inequality = Set (LExp FO Refs)

update :: a -> Simplified a
update a = lift $ Updated a
```

The simplification algorithm can now be implemented as a set of functions which recursively traverse and simplify a BExp. It is located in the Logic.Simplification module. The implementation of each of the three simplification rules is shown in listing 9

Listing 9: implementation of the simplification rules

```
simplifyState :: State -> Simplified State
simplifyState original@(Update (Update s l1 _) l2 w) = do
    memComparison <- compareLExp l1 l2
    case memComparison of
        MemEq -> update $ Update s l2 w
        _ -> simplifyState' original
simplifyState s = simplifyState' s

simplifyAExp :: AExpFO -> Simplified AExpFO
simplifyAExp original@(AIdt (LRead (Update state toUpdate aExp)
    ↪  toRead)) = do
    memComparison <- compareLExp toRead toUpdate
    case memComparison of
        MemEq -> update aExp
        _ -> simplifyAExp' original
simplifyAExp a = simplifyAExp' a

simplifyLExp :: LExpFO -> Simplified LExpFO
simplifyLExp original@(LRead (Update state toUpdate _) toRead)
    ↪ = do
    memComparison <- compareLExp toRead toUpdate
    case memComparison of
        MemNotEq -> update $ LRead state toRead
        _ -> simplifyLExp' original
simplifyLExp l = simplifyLExp' l
```

simplifyState', simplifyAExp', and simplifyLExp' are not shown in the listing, as they just recursively descend the expression. The function simplifyBExp, which starts simplification on the formula level has a special handling for implications. Inequalities specified on the left hand side of an implication can be used to simplify the right hand side of the same implication as described in section 4.5. For this purpose the context used to simplify the right hand side is enriched with the inequalities that were found in the left hand side, which is depicted in listing 10.

Listing 10: searching for predefined inequalities

```
simplifyBExp :: BExpFO -> Simplified BExpFO
simplifyBExp (BBinExp op l r) = do
   updatedL <- simplifyBExp l
   let lhsInequalities = if op == Implies
         then findInequalities updatedL
         else Set.empty
   updatedR <- local (addInequalities lhsInequalities) $
      ↪ simplifyBExp r
   return $ BBinExp op updatedL updatedR
... other cases of simplifyBExp: recursively simplify BExp ...

findInequalities :: BExpFO -> Set Inequality
findInequalities (BComp NotEqual (AIdt l1) (AIdt l2)) =
   Set.singleton $ notEqual l1 l2
findInequalities (BBinExp And fo1 fo2) = Set.union (
   ↪ findInequalities fo1) (findInequalities fo2)
findInequalities _ = Set.empty
```

The search for inequalities is again implemented as a simple heuristic. Inequalities are found if they are either specified on the top level of the formula, or are part of a (possibly nested) conjunction.

4.7 Statements

The program that should be verified is a sequence of statements. The type `Stmt` models a statement and is depicted in listing 11.

Listing 11: Stmt

```
type BExp’ l = BExp l Plain
type AExp’ l = AExp l Plain
type LExp’ l = LExp l Plain

data Stmt
    = Empty
    | Declaration Idt
    | Assignment (LExp’ C0) (AExp’ C0)
    | While (BExp’ C0) (BExp’ FO) Stmt LineNo
    | ITE (BExp’ C0) Stmt Stmt
    | FunCall (Maybe (LExp’ C0)) Idt [AExp’ C0] LineNo
    | Assertion (BExp’ FO) LineNo
    | Return (Maybe (AExp’ C0))
    | Seq Stmt Stmt
```

Since the statements are obtained directly from the parser, all statement use the `Plain` memory model. `verify-c` supports the following statements:

Empty

Empty statements have no effect.

Declarations

Declarations declare the memory address of an identifiers as fresh.

Assignments

Assignments which also act as declarations like `int i = 0;` are split into a declaration and an assignment by the parser.

While Loops

While loops consist of a `C0` logic condition, an `FO` logic invariant and a body. The current line number is included for logging purposes.

If-Then-Else Branches

If-Then-Else branches consist of a `C0` logic condition, a true branch, and a false branch. If conditions without an else branch are parsed as If-Then-Else branches with an empty false branch.

Function Calls

Function Calls consisting of an assignment target, the function name, and a list of function arguments. The current line number is included for logging purposes. Since some functions do not return a result, the assigned variable is optional. Function calls are modelled as separate statements instead of an arithmetic expressions in order to avoid dealing with ambiguous orders of side effects in expressions like `n = inc(&n) + inc(&n);`. Function calls which also act as declarations are split into a declaration and a function call by the parser.

Assertions

Assertions replace the current precondition.The current line number is included for logging purposes.

Return Statements

Return statements may or may not return a value.

Sequences

Statements can be combined into sequences.

4.8 Applying the Floyd-Hoare Logic

The Floyd-Hoare logic was intensively discussed in the *Korrekte Software: Grundlagen und Methoden* course. Given a postcondition, for every statement an approximate weakest precondition (*awp*) is calculated. Traversing the statements in reverse order starting with the programs last statement, the precondition of a statement is the postcondition of the next statement. Some statements like the while loop have a predetermined postcondition. In this case a verification condition is generated which states that the predetermined postcondition implies the precondition of the following statement. To prove the correctness of a program every generated verification condition has to be proven. The following functions are used to calculate the verification conditions of statements:

$$
\begin{aligned}
awp(\texttt{Empty}, Q, Q_R) =& Q \\
awp(\texttt{Assertion a}, Q, Q_R) =& \texttt{a} \\
awp(\texttt{ITE c t f}, Q, Q_R) =& (awp(\texttt{t}, Q, Q_R) \wedge \texttt{c}) \vee (awp(\texttt{f}, Q, Q_R) \wedge \neg\texttt{c}) \\
awp(\texttt{While c i b}, Q, Q_R) =& \texttt{i} \\
awp(\texttt{Seq s1 s2}, Q, Q_R) =& awp(\texttt{s1}, awp(\texttt{s2}, Q, Q_R), Q_R) \\
awp(\texttt{Assignment l e}, Q, Q_R) =& Q[update(\sigma, l, e)/\sigma] \\
awp(\texttt{Declaration i}, Q, Q_R) =& \text{simplify } Q \text{ with } \texttt{i} \neq \text{any other memory address} \\
awp(\texttt{FunCall r fun args}, Q, Q_R) =& pre(\texttt{fun})[\texttt{args}/params(\texttt{fun})] \\
awp(\texttt{Return r}, Q, Q_R) =& Q_R[\texttt{r}/\texttt{\textbackslash result}]
\end{aligned}
$$

$$
\begin{aligned}
wvc(\texttt{Empty}, Q, Q_R) =& \emptyset \\
wvc(\texttt{Assertion a}, Q, Q_R) =& \{\texttt{a} \Rightarrow Q\} \\
wvc(\texttt{ITE c t f}, Q, Q_R) =& wvc(\texttt{t}, Q, Q_R) \cup wvc(\texttt{f}, Q, Q_R) \\
wvc(\texttt{While c i b}, Q, Q_R) =& \{\texttt{i} \wedge \texttt{b} \Rightarrow awp(\texttt{b}, \texttt{i}, Q_R), \texttt{i} \wedge \neg\texttt{b} \Rightarrow Q\} \cup wvc(\texttt{b}, \texttt{i}, Q_R) \\
wvc(\texttt{Seq s1 s2}, Q, Q_R) =& wvc(\texttt{s1}, awp(\texttt{s2}, Q, Q_R), Q_R) \cup wvc(\texttt{s2}, Q, Q_R) \\
wvc(\texttt{Assignment l e}, Q, Q_R) =& \emptyset \\
wvc(\texttt{Declaration i}, Q, Q_R) =& \emptyset \\
wvc(\texttt{FunCall r fun args}, Q, Q_R) =& post(\texttt{fun})[\texttt{args}/params(\texttt{fun})][\texttt{r}/\texttt{\textbackslash result}] \Rightarrow Q \\
wvc(\texttt{Return r}, Q, Q_R) =& \emptyset
\end{aligned}
$$

Q and Q_R are the post- and result conditions for which pre- and verification conditions should be generated. $P[a/b]$ is the formula P in which every occurrence of b has been replaced by a. If a and b are lists, a replacement is made for each element. $pre(f)$, $post(f)$, and $params(f)$ are the precondition, postcondition, and parameter list of the function f. The implementation of *awp* and *wvc* can be found in the module `VC` with some additions.
Additionally the implementation applies the following transformations, which were omitted from the definitions above to keep them small:

- Each generated condition is simplified using the algorithm presented in section 4.6.

- Every time a formula, arithmetic expression, or LExpression, which is part of a statement, is used to calculate the *awp* or *wvc* of that statement, its logic type is converted to `FO` and its memory model is converted to `Refs`. The GADTs that are used to model the different types of expressions ensure at compile time that the correct format is used.

- The generated verification conditions are enriched with debug information like line numbers.

Based on the *wvc* and *awp* functions, verification conditions can be calculated separately for each function of the program as follows:

$$\begin{aligned} wvcF(f) = \{&pre(f) \Rightarrow awp(body(f), false, post(f))\} \\ &\cup wvc(body(f), false, post(f)) \end{aligned}$$

The Floyd-Hoare logic rules implemented in `verify-c` follow the rules presented in the *Korrekte Software: Grundlagen und Methoden* course closely but the following changes were made:

- Declarations were added as separate statements to enable better simplification.

- Function calls produce a verification condition.

- The `\old(x)` syntax was dropped. Users are expected to bind values to logical variables in the precondition instead.

4.9 SMT Export

The generated verification conditions have to be proven in order to show the correctness of a program. This task is tedious. While the satisfiability of first order logic is not decidable in general, often formulas can be proven by a theorem prover. The `verify-c` exports the generated verification conditions into the SMT-LIB standard and runs the `Z3` solver in order to automate the profs. While this feature is only rudimentarily implemented it is already usable. In order to show how the export works, listing 12 shows one of the verification condition of the faculty program which was presented in section 3 in listing 1. Exports of all verification conditions are located in the `.\target` folder created by `verify-c`.

Listing 12: an exported verification condition

```
 1  (declare-fun read_array (Int Int) Int)
 2  (declare-fun deref (Int) Int)
 3  (declare-fun c () Int)
 4  (declare-fun n () Int)
 5  (declare-fun p () Int)
 6  (declare-fun fac (Int) Int)
 7  (assert (= (fac 0) 1))
 8  (assert (forall ((nn Int)) (implies (< 0 nn) (= (fac nn) (* nn
       ↪ (fac (- nn 1)))))))
 9  (assert (not (implies (and (and (and (= p (fac (- c 1))) (<= c
       ↪ (+ n 1))) (> c 0)) (<= c n)) (and (and (= (* p c) (fac
       ↪ (- (+ c 1) 1))) (<= (+ c 1) (+ n 1))) (> (+ c 1) 0)))))
10  (check-sat)
```

The export can be read as follows:
l.1 - l.2 of listing 12 are predefined functions. They are contained in every export. `read_array` is a function which simulates array access. It takes two integer arguments. The first is interpreted as the memory address of an array, the second is an index. The return value is interpreted as the value of specified array at the specified index. `deref` is a function which simulates dereferencing. It takes one integer argument, which is interpreted as a memory address. The return value is interpreted as the value which is obtained by reading the memory at that address.
l.3 - l.5 are the unbound local variables which appear in the verification condition that should be exported. They are modelled as constant functions which take no input and return the variables value.
l.7 - l.8 contains the environment which is read from the optional `.env` file accompanying the program which should be proven. The environment should contain custom functions and predicates as well as structure accessors.
l.9 contains the negated exported verification condition. In order to export the verification condition its memory model is converted back from `Refs` to `Plain`. For every boolean function which is supported by `verify-c` an

equivalent SMT-LIB function exists, so they can be translated directly. The same goes for arithmetic functions except for the address operator. Verification conditions containing the address operator cannot be exported. The most difficult part is exporting the LExpressions. Identifiers are exported to variable names. Array access and dereferencing is modelled by the predefined functions mentioned above. Structure parts accessors are expected to be modelled by the user as part of the environment, where the name of the accessor should match the name of an unary function.
l.10 tells the SMT solver to start the prof. If the solver can prove that there is no model for the negated verification condition (`unsat`), the verification condition is valid and proven.

5 Limitations

`verify-c` is an educational project and should not be used to verify any critical software. It is only tested on small example programs. No guarantees are made by the author. This section should list known limitations without being exhaustive.

5.1 Correctness

While the purpose of the Floyd-Hoare logic is to prove correctness of programs, there are some classes of errors which cannot be detected by this implementation, including:

- non-terminating loops
- array index out of bounds errors
- overflows

5.2 Scalability

One important feature which is not implemented is the ability to specify which parts of the memory are changed by a function call. To remain sound without this specification only the postcondition of the called function can be assumed. This makes it hard (maybe even impossible) to prove programs like then recursive fibonacci program shown in listing 13.

Listing 13: a recursive fibonacci program

```
int fib(int n) {
    precondition("n >= 0");
    postcondition("\result == fibonacci(n)");

    if(n == 0 || n == 1){
        return 1;
    } else {
        int prev = fib(n - 1);
        int prev2 = fib(n - 2);
        return prev + prev2;
    }
}
```

This severely limits the scale in which `verify-c` can be applied.

5.3 Compatibility with C

`verify-c` only supports a subset of the C programming language. Missing features include:

- primitive types other than integers
- includes
- union types
- syntactic sugar like for loops or increment operators (`i++;`)