

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

Nikolas Mählmann

April 6, 2020

## 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 it generates a set of verification conditions. 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. Lets 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 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 after calling 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 loop is completed. 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

```
1 (declare-fun fac (Int) Int)
2 (assert (= (fac 0) 1))
3 (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 (1:8) : OK
[3/3] : While Case False (1:8) : OK

Summary: VERIFICATION OK
```

Hooray! 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. Other 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 unknown 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 I will not go into further details about parsing. 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`

```
1 data BExp l m where
2   BTrue  :: BExp l m
3   BFalse :: BExp l m
4   BNeg   :: BExp l m -> BExp l m
5   BBinExp :: BBinOp -> BExp l m -> BExp l m -> BExp l m
6   BComp   :: CompOp -> AExp l m -> AExp l m -> BExp l m
7   BForall :: Idt -> BExp FO m -> BExp FO m
8   BExists :: Idt -> BExp FO m -> BExp FO m
9   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. `C0`: 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. `F0`: 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 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 C0 m` is a subset of `BExp F0 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 in order to support references.

### 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`

```

1 data AExp l m where
2   ALit :: Integer -> AExp l m
3   AIdt :: LExp l m -> AExp l m
4   ABinExp :: ABinOp -> AExp l m -> AExp l m -> AExp l m
5   AFuncall :: Idt -> [AExp FO m] -> AExp FO m
6   ALogVar :: Idt -> AExp FO m
7   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 combination of parameters. 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 5.2. 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

```

1 data LExp l m where
2   LIdt :: Idt -> LExp l m
3   LArray :: LExp l m -> AExp l m -> LExp l m
4   LStructurePart :: LExp l m -> Idt -> LExp l m
5   LRead :: State -> LExp l Refs -> LExp l Refs
6   LDeref :: LExp l Plain -> LExp l Plain

```

As parts of AExps they are also parameterized with the type of logic and memory model. Identifiers and array and struct accessors are available for every combination of parameters. 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 on which we will go into 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, 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 is creates an updated a state:

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

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

Listing 6: State

```

1 data State
2   = Atomic String
3   | Update State (LExp F0 Refs) (AExp F0 Refs)

```

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

$$\mathbf{a} \hat{=} read(\sigma, a)$$

$$\&\mathbf{a} \hat{=} a$$

$$*\mathbf{a} \hat{=} read(\sigma, read(\sigma, a))$$

Often programs assign a lot of values which leads to deeply nested states, for example:

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

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

$$S1 : l_1 = l_2 \Rightarrow read(update(\sigma, l_2, v), l_1) = v$$

$$S2 : l_1 \neq l_2 \Rightarrow read(update(\sigma, l_2, v), l_1) = read(\sigma, l_1, v)$$

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:

$$*\mathbf{a} \stackrel{?}{=} *\mathbf{b}$$

$$\mathbf{a}[i] \stackrel{?}{=} \mathbf{a}[j]$$

## 5 Limitations

### 5.1 Primitives

### 5.2 Function Calls