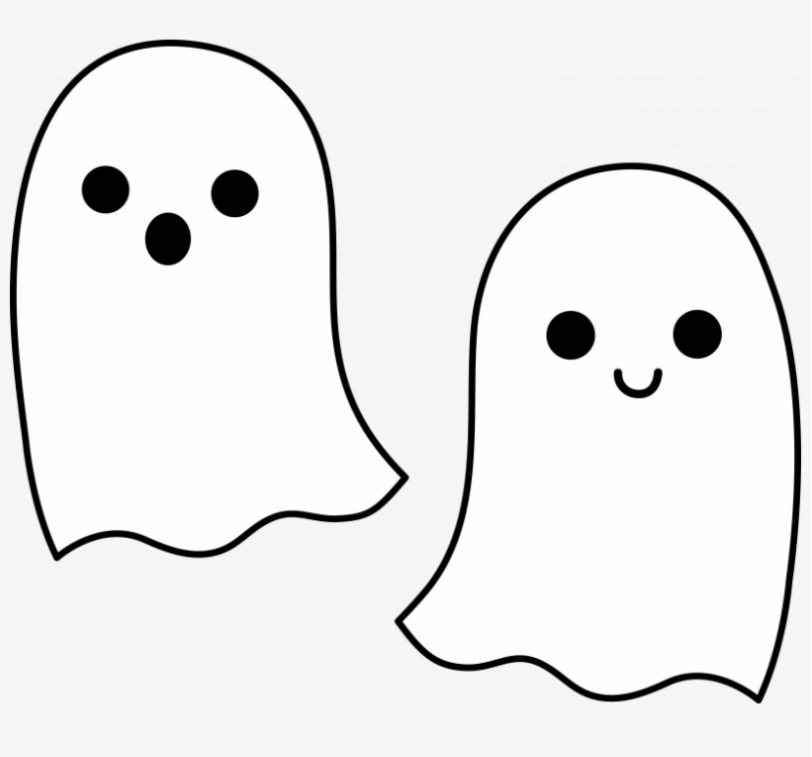
UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE



Project outside course scope

Jacob Herbst(mwr148), Matilde Broløs (jtw868)

IFC

An imperative language with static verification

Advisor: Ken Friis Larsen

1. April 2022

Abstract

Contents

1 Introduction

In this project we present a small imperative *While* language in the C-style family with a built-in assertion language. This language uses Hoare logic and predicate tranformer semantics to allow for verification condition generation similar to *Why3*. Our VC generator discharges verification conditions fit for the SMT solver *Z3*, allowing us to prove the partial or total correctness of a program.
The main motivation for implementing this language is to make a base which we can easily extend to include *Partial Equivalence Relation* (PER) logic, which is a logic proposed to reason about *Information Flow Control*[]. This can be used for making assertions about the security of code.
Predicate Transformer Semantics, which is used for verification condition generation, was introduced by Dijkstra in 1975[1]. These conditions are based on the axiomatic system presented by C.A.R Hoare in 1969, [2]. Using this system, and the rules of inference that it introduces, one can formally prove the correctness of a program. Combined, this provides an approach for automatically generating conditions which can verify the correctness of certain programs.
Why3 is an example of a platform which provides automatic program verification. *Why3* requires the programmer to write assertions inside the program code, and then utilises these assertions to compute verification conditions for the program. Once the verification condition is generated, *Why3* can discharge this condition to a variety of SMT solvers or proof assistants, which can then determine whether this condition can be proved.
This report presents the work of this project as follows:

- In Section 2, we present the syntax and semantics of our language. Next we explain how VC generation works, how it is related to Hoare logic and how it can be used for automatically proving certains properties about programs. Finally we describe the coupling between VC-generators and SMT-solvers.
- In Section 3, we present our implementation, which can both dynamically evaluate a program and statically prove the correctness. In the implementation we transform the formal semantics of the language into equivalent Haskell code and utilizes predicate transformer semantics to compute verification conditions for a program.
- In Section 4, we present our strategy for evaluting the code quality and give an assessment of the implementation. Here, a combination of blackbox tests, automatic tests, and test programs, is used to assess the quality.
- Lastly, Section 5 presents some ideas for future work, presents a discussion of the work, and concludes on the project.

Throughout the project our main focus has been on getting a good understanding of how verification conditions work, and how to convert this into Haskell code. It has also been on building an implementation that is robust and easily extended and maintained. Furthermore it has been interesting for us to investigate what is necessary to prove correctness of a program, using our implementation, and how to thouroughly test the correctness of our work.
Enjoy!

```
res := 0;
while (q > 0) {
      res := res + r;
      q := q - 1;
};
```

Figure 1: Example program `mult.ifc`

2 Background

In this section we present the simple *While* language that we work with throughout the project. Next we explain about verification condition generation and Hoare logic, and how this provides a basis for proving correctness of code. Finally, we describe how this can be coupled with SMT solvers, to allow automatic proving of programs.

2.1 Language

While is a small imperative programming language with a build in assertion language, enabeling the possibility of statically verifying programs by the use of external provers. The assertions also enables us to dynamically check the program during evaluation. In this section we will present the syntax and semantics of both the imperative language and the assertion language.
TODO: update while to also include invariant.
TODO: add rules for while-loops with invariants and variants? TODO: add argumentation for semantic equivalence assertion $\sim$ if assertion then skip else violate.
Table 1 shows the grammar of IFC. In essence an IFC program is a statement with syntax in the C-family. The language is small, as the focus in this project has been to correctly being able to generate verification conditions for said programs. In Table 2 we show the semantics of the different statements.
Arithmetic expressions (`aexpr`) follow the standard rules of precedence and associativity:

- parenthesis
- negation
- Multiplication, division and modulo (left associative)
- Addition and subtraction (left associative)

where parenthesis binds tightest.
Likewise boolean expressions (`bexpr`) follow the standard precedence rules:

- negate (!)
- logical and (&&)
- logical or (||)

⟨*statement*⟩ ::= ⟨*statement*⟩ ';' ⟨*statement*⟩
| ⟨*ghostid*⟩ ':=' ⟨*aexpr*⟩
| ⟨*id*⟩ ':=' ⟨*aexpr*⟩
| 'if' ⟨*bexpr*⟩ '{' ⟨*statement*⟩ '}'
| 'if' ⟨*bexpr*⟩ '{' ⟨*statement*⟩ '}' 'else' '{' ⟨*statement*⟩ '}'
| 'while' ⟨*bexpr*⟩ ⟨*invariant*⟩ ⟨*variant*⟩ '{' ⟨*statement*⟩ '}'
| '#{' ⟨*assertion*⟩ '}'
| 'skip' | 'violate'

⟨*invariant*⟩ ::= '?{' ⟨*assertion*⟩ '}'
| ⟨*invariant*⟩ ';' ⟨*invariant*⟩

⟨*variant*⟩ ::= '!{' ⟨*aexpr*⟩ '}' | ϵ

⟨*assertion*⟩ ::= 'forall' ⟨*id*⟩ '.' ⟨*assertion*⟩
| 'exists' ⟨*id*⟩ '.' ⟨*assertion*⟩
| '$\sim$' ⟨*assertion*⟩
| ⟨*assertion*⟩ ⟨*assertionop*⟩ ⟨*assertion*⟩
| ⟨*bexpr*⟩

⟨*assertionop*⟩ ::= '$\wedge$' | '$\vee$' | '$\Rightarrow$'

⟨*bexpr*⟩ ::= 'true' | 'false' |
| '!'⟨*bexpr*⟩
| ⟨*bexpr*⟩ ⟨*bop*⟩ ⟨*bexpr*⟩
| ⟨*bexpr*⟩ ⟨*rop*⟩ ⟨*bexpr*⟩
| '(' ⟨*bexpr*⟩ ')'

⟨*bop*⟩ ::= '&&' | '||'

⟨*rop*⟩ ::= '<' | '$\leq$' | '=' | '$\neq$' | '>' | '$\geq$'

⟨*aexpr*⟩ ::= ⟨*id*⟩
| ⟨*ghostid*⟩
| ⟨*integer*⟩
| '-'⟨*axpr*⟩
| ⟨*aexpr*⟩ ⟨*aop*⟩ ⟨*aexpr*⟩
| '(' ⟨*aexpr*⟩ ')'

⟨*aop*⟩ ::= '+' | '$-$' | '$*$' | '/' | '%'

⟨*ghostid*⟩ ::= ghost ⟨*string*⟩ | \$⟨*string*⟩

⟨*id*⟩ ::= ⟨*string*⟩

Table 1: Grammar of IFC

$violate$	$\frac{}{\langle \texttt{violate}, \sigma \rangle \downarrow \lightning}$
$skip$	$\frac{}{\langle \texttt{skip}, \sigma \rangle \downarrow \sigma}$
$assign$	$\frac{\langle a, \sigma \rangle \downarrow n}{\langle X := a, \sigma \rangle \downarrow \sigma[X \mapsto n]}$
$sequence$	$\frac{\langle s_0, \sigma \rangle \downarrow \sigma'' \quad \langle s_1, \sigma'' \rangle \downarrow \sigma'}{\langle s_0 ; s_1, \sigma \rangle \downarrow \sigma'}$
$if\text{-}true$	$\frac{\langle b, \sigma \rangle \downarrow \texttt{true} \quad \langle s_0, \sigma \rangle \downarrow \sigma'}{\langle \texttt{if } b \texttt{ then } s_0 \texttt{ else } s_1 \rangle \downarrow \sigma'}$
$if\text{-}false$	$\frac{\langle b, \sigma \rangle \downarrow \texttt{false} \quad \langle s_1, \sigma \rangle \downarrow \sigma'}{\langle \texttt{if } b \texttt{ then } s_0 \texttt{ else } s_1 \rangle \downarrow \sigma'}$
$while\text{-}false$	$\frac{\langle b, \sigma \rangle \downarrow \texttt{false} \quad \langle i, \sigma \rangle \downarrow \texttt{true}}{\langle \texttt{while } b \texttt{ invariant } i \texttt{ do } s_0 \rangle \downarrow \sigma}$
$while\text{-}true$	$\frac{\langle b, \sigma \rangle \downarrow \texttt{true} \quad \langle i, \sigma \rangle \downarrow \texttt{true} \quad \langle s_0, \sigma \rangle \downarrow \sigma'' \quad \langle \texttt{while } b \texttt{ invariant } i \texttt{ do } s_0, \sigma'' \rangle \downarrow \sigma'}{\langle \texttt{while } b \texttt{ invariant } i \texttt{ do } s_0 \rangle \downarrow \sigma'}$
$while\text{-}i\text{-}false$	$\frac{\langle i, \sigma \rangle \downarrow \texttt{false}}{\langle \texttt{while } b \texttt{ invariant } i \texttt{ do } s_0 \rangle \downarrow \lightning}$

Table 2: Semantics for the *while* language.

Multiplication as an example. To show what the language is capable of, a small example program computing the multiplication of two integers q and r is presented in Figure 1. Line 2 shows the syntax of assignments, lines 3-6 shows the syntax of while loops, and the entire program is also a demonstration of sequencing statements. For now, we leave out the assertions, but we will come back to this in Section 2.2.
Now even though this small while language is very simple, it is still very interesting as base language for our project. By creating an assertion language that can prove the correctness for programs written in this simple language, we can show that it is possible to prove termination and correctness of while loops, which are definitely the tricky part of this language. Loops are specifically challenging because we cannot know whether they ever terminate. Furthermore the way a loop affects program variables is less see-through than for example the effects of an assignment. Therefore it is desirable to be able to prove the correctness of such programs, and that is why this small language is well suited for the purpose.

2.2 Hoare Logic

Now we look at Hoare logic, and how we can use it to verify our program. To assert that a program works as intended, we want to be able to prove it formally. To avoid the proofs being too detailed and comprehensive, one wants to look at the essential properties of the constructs of the program. We look at partial correctness of a program, meaning that we do not ensure that a program terminates, only that *if* it terminates, certain properties will hold.

```
1  vars: [q,r]
2  requirements: {q >= 0 /\ r >= 0}
3  <!=_=!>
4  res := 0;
5    a   := q;
6  while (q > 0) ?{res = (  a   - q) * r /\ q >= 0} !{q} {
7        res := res + r;
8        q := q - 1;
9  };
10 #{res =   a   * r};
```

Figure 2: Example program `mult.ifc`

2.2.1 Assertions

For expressing these properties we use *assertions*, which are a way of claiming something about a program state at a specific point in the program. Assertions consist of a *precondition* P and a *postcondition* Q, and is written as a triple

$$\{P\}S\{Q\}$$

This triple states that *if* the precondition P holds in the initial state, and *if* S terminates when executed from that state, *then* Q will hold in the state in which S halts. Thus the assertion does not say anything about whether S terminates, only that if it terminates we know Q to hold afterwards. These triples are called Hoare triples.

Logical variables vs program variables. When using assertions we differ between *program variables* and *logical variables*. Sometimes we might need to keep the original value of a variable that is changed through the program. Here we can use a *logical variable*, or *ghost variable*, to maintain the value. The ghost variables can only be used in assertions, not in the actual program, and thus cannot be changed. All ghost variables must be fresh variables. For example, the assertion $\{x = n\}$ asserts that x has the same value as n. If n is not a program variable, this is equivalent to declaring a ghost variable n with the same value as x. As n is immutable, we can use n to make assertions depending on the initial value of x later in the program, where the value of x might have changed.

Multiplication example. As an example of this, Figure 2 shows how our example program `mult.ifc` looks when adding some assertions.
First we have the input variables listed, and the requirements for the input given in line 1-2. The requirements states that bith integers must be nonnegative, as we do not wish to multiply by zero. In line 10 we have a assertion claiming that the final result will be the original value of q multiplied by r. Here we use a ghost variable, or a logical variable, for keeping the original value og q. We also have an invariant and a variant in the *while*-loop in line 6, but we will come back to that later.

2.2.2 Axiomatic system for partial correctness

The Hoare logic specifies am inference system for the partial correctness assertions, that show the axiomatic semantics for the different constructs of the language. The axiomatic system is shown in Table 3.

This axiomatic system shows how assertions are evaluated in the Hoare logic. For example, for an assignment we can say that if we bind x to the evaluated value of a in the initial state, and if P holds in this state, then after assigning x to a, P must still hold. For the *skip* command we see that the assertion must hold both before and after, as *skip* does nothing. The axiomatic semantics for an assertion around a sequence of statements $\{P\}S_1; S_2\{R\}$ state that if P hold in the initial state, and executing S_1 in this state produces a new state in which Q holds, and if executing S_2 in this new state produces a state in which R holds, then $\{P\}S_1; S_2\{R\}$ holds. Another interesting construct is the *while*-loop, especially for our small *while*-language. Here P denotes the loop invariant, and b is the loop condition. If b evaluates to $true$ and P holds in the initial environment, and executing S in this environment produces a new state in which P holds, then we know that after the *while*-loop has terminated we must have a state where P holds and where b evaluates to $false$.

$[violate_p]$	$\{false\}\texttt{violate}\{Q\}$
$[skip_p]$	$\{P\}\texttt{skip}\{P\}$
$[ass_p]$	$\{P[x \mapsto \mathcal{A}[\![a]\!]]\}\ x := a\ \{P\}$
$[seq_p]$	$\frac{\{P\}S_1\{Q\} \quad \{Q\}S_2\{R\}}{\{P\}S_1;S_2\{R\}}$
$[if_p]$	$\frac{\{\mathcal{B}[\![b]\!]\wedge P\}S_1\{Q\} \quad \{\neg\mathcal{B}[\![b]\!]\wedge P\}S_2\{Q\}}{\{P\}\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\{Q\}}$
$[while_p]$	$\frac{\{\mathcal{B}[\![b]\!]\wedge P\}S\{P\}}{\{P\}\ \texttt{while}\ b\ \texttt{do}\ S\{\neg\mathcal{B}[\![b]\!]\wedge P\}}$
$[cons_p]$	$\frac{\{P'\}S\{Q'\}}{\{P\}S\{Q\}}$ if $P \Rightarrow P'$ and $Q' \Rightarrow Q$

Table 3: Axiomatic system for partial correctness.

TODO: state semantic equivalence between this notation, and the notation that we use in our language.

2.2.3 Total correctness

Right now, the Hoare logic can help us prove partial correctness, but does not guarantee termination of loops. We would like to be able to prove total correctness, meaning we want to prove termination. To verify that a program terminates, we need a stronger assertion, in form of a loop variant. The loop variant is an expression that decreases with each iteration of a loop, for example in the *while*-loop from our example program q is the variant, as can be seen in line 8 of the code (see code listing Figure 2). To express the logic of using a variant for *while*-loops, we use the following modified syntax.

$$\frac{\{I \wedge e \wedge v = \xi\}s\{I \wedge v \prec \xi\} \quad wf(\prec)}{\{I\}\ \texttt{while}\ e\ \texttt{invariant}\ I\ \texttt{variant}\ v, \prec\ \texttt{do}\ s\{I \wedge \neg e\}}$$

where v is the variant of the loop, and ξ is a fresh logical variable. $\prec$ is a well-founded

relation, and because the data type consists of all unbounded integers, we use the well-founded relation:

$$x \prec y \quad = \quad x < y \wedge 0 \leq y$$

reference: (poly1.pdf)
Using these loop invariants and variants it is possible to prove total correctness of programs containing *while*-loops.

2.3 Verification Condition Generation

In the previous section we presented Hoare Triples and how they can be used to "assign meaning to programs". However, it might not always be possible that the Precondition is known. In such a case, we can use Weakest Precondition Calculus (denoted WP), which essentially states:

$$\{WP(S, Q)\}S\{Q\}$$

That is, we can use a well-defined calculus to find the weakest (least restrictive) precondition that will make Q hold after S. By this calculus, which we present in this section, validation of Hoare Triples can be reduced to a logical sentence, since Weakest Precondition calculus is functional compared to the relational nature of Hoare logic. By computing the weakest precondition, we get a formula of first-order logic that can be used to verify the program, and that is called *verification condition generation*.

2.3.1 Weakest Liberal Precondition

Below the structure for computing the weakest liberal precondition for the different constructs is shown.
put in also violate

$$\begin{aligned}
WLP(\texttt{skip}, Q) &= Q \\
WLP(x := e, Q) &= \forall y, y = e \Rightarrow Q[x \leftarrow y] \\
WLP(s_1; s_2, Q) &= WLP(s_1, WLP(s_2, Q)) \\
WLP(\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2, Q) &= (e \Rightarrow WLP(s_1, Q)) \\
&\quad \wedge (\neg e \Rightarrow WLP(s_2, Q)) \\
WLP(\texttt{while } e \texttt{ invariant } I \texttt{ do } s, Q) &= I \wedge \\
&\quad \forall x_1, ..., x_k, \\
&\quad (((e \wedge I) \Rightarrow WLP(s, I)) \\
&\quad\quad \wedge ((\neg e \wedge I) \Rightarrow Q))[w_i \leftarrow x_i] \\
&\quad \text{where } w_1, ..., w_k \text{ is the set of} \\
&\quad \text{assigned variables instatement } s \text{ and} \\
&\quad x_1, ..., x_k \text{ are fresh logical variables.} \\
WLP(\{P\}, Q) &= P \wedge Q \quad \text{where P is an assertion}
\end{aligned}$$

The rules are somewhat self-explanatory, but we would like to go through some rules which have been important for our work.

Assignments. The rule for computing weakest liberal precondition for assignments says that for all variables y where $y = e$, we should exchange x in Q with y. That is, we exchange all occurrences of x with the value that we assign to x.
SAy something about forall.

Sequence. For finding the *wlp* of a sequence of statements $s_1; s_2$, we need to first find the *wlp* Q' of s_2 with Q, and then compute the *wlp* of s_1 with Q'. This shows how we compute the weakest liberal precondition using a bottom-up approach.

While-loops. To compute the *wlp* of a while loop we have to eensure that the loop invariant holds before, inside, and after the loop. That is why the first condition is simply that the invariant I must evaluate to $true$. Next we need to assert that no matter what values the variables inside the loop have, the invariant and loop condition will hold whenever we go into the loop, and the invariant and negated loop condition will hold when the loop terminates. For these variables, we must also exchange all the occurrences in the currently accumulated weakest liberal precondition Q.

2.3.2 Weakest Precondition

Now the weakest liberal precondition does not prove termination. If we want to prove termination in addition to the partial correctness obtained from *wlp*, we need a *weakest precondition* which is much like *wlp*, but require that *while*-loops have a loop variant. Here we have to use the modified Hoare logic for *while*-loops presented in Section 2.2.3.
The structure for computing the weakest preconditions for the constructs for total correctness is much like the one for computing weakest liberal precondition, except for the structure of *while*-loops, which can be seen below.

$$WP\left(\begin{array}{l}\texttt{while } e \texttt{ invariant } I \\ \quad \texttt{variant } v, \prec \texttt{ do } s\end{array}, Q\right) = I \wedge$$
$$\forall x_1, ..., x_k, \xi,$$
$$(((e \wedge I \wedge \xi = v) \Rightarrow WP(s, I \wedge v \prec \xi))$$
$$\wedge ((\neg e \wedge I) \Rightarrow Q))[w_i \leftarrow x_i]$$
where $w_1, ..., w_k$ is the set of assigned variables in statement s and $x_1, ..., x_k, \xi$ are fresh logical variables. (1)

We see that the difference between the computation of *wp* and *wlp* is the presence of the *variant*. When given a variant expression v for the loop, we add the assertion that this expression decreases with each iteration, using the logical variable ξ to keep the old value of v to compare with.
For our example program `mult.ifc`, we can compute the weakest precondition, as we have both a variant and an invariant in the while loop. By applying the *wp* rules on the example, we get the weakest precondition seen in Figure 3.
The interesting thing is how the *while*-loop is resolved. First the *invariant* is checked in line 4, according to the first part of the *wp* rule. Next the rule requires a forall statement over all the variables altered in the loop bode, and that is what happens in line 5 of the formula. Inside this forall we now check that the loop invariant and condition holds when entering each iteration (lines 6-9), and that the invariant and the negated loop condition holds when the loop terminates(line 10). The variant is used in line 6, where the logical variable ξ is set to hold the value of the variant, and in line 9, where it is checked that the variant has decreased by comparing to the logical variable holding the value that the variant had at the beginning of the loop.

$$
\begin{array}{l}
\forall q, r.\ (q >= 0 \wedge r >= 0) \Rightarrow \\
\quad \forall res_3.\ res_3 = 0 \Rightarrow \\
\quad\quad \forall \$a.\ \$a = q \Rightarrow \\
\quad\quad\quad (res_3 = (\$a - q) * r \wedge q >= 0) \\
\quad\quad\quad \wedge \forall q_2, res_2, \xi_1. \\
\quad\quad\quad\quad (q_2 > 0 \wedge res_2 = (\$a - q_2) * r \wedge q_2 >= 0 \wedge \xi_1 = q_2) \Rightarrow \\
\quad\quad\quad\quad\quad \forall res_1.\ res_1 = res_2 + r \Rightarrow \\
\quad\quad\quad\quad\quad\quad \forall q_1.\ q_1 = q_2 - 1 \Rightarrow \\
\quad\quad\quad\quad\quad\quad\quad (res_1 = (\$a - q_1) * r \wedge q_1 >= 0 \wedge 0 \Leftarrow \xi_1 \wedge q_1 < \xi_1) \\
\quad\quad\quad \wedge ((q_2 \Leftarrow 0 \wedge res_2 = (\$a - q_2) * r \wedge q_2 >= 0) \Rightarrow res_2 = \$a * r)
\end{array}
$$

Figure 3: Weakest precondition of `mult.ifc`

2.4 SAT-solvers

A SAT solver is a program that can determine whether a boolean formula is satisfiable. The advantage of this is that is becomes possible to automatically verify whether very large logical formulae are satisfiable. When given a boolean formula, a SAT solver will determine if there are values for the free variables which satisfy the formula. If this is not the case, the SAT solver comes up with a suiting counter example.
Satifiability modulo theories is the problem of determining whether a mathematical formula is satisfiable, and thus generalizes the SAT problem. SMT solvers can take as input some first-order logic formula, and determine if it is satisfiable, similar to SAT solvers. Verification condition generators are often coupled with SMT solvers, so that one can finde the weakest precondition of a program, and then use an SMT solver to determine whether the condition is satisfiable.
An example of an SMT solver is the *Z3 Theorem Prover*, which is such a solver created by Microsoft. The goal of *Z3* is to verify and analyse software. (wiki) In our project we use *Z3* to verify program, by first computing the *wp* of the program, given the assertions provided by the user, and then feeding this to the SMT solver.

3 Implementation

IFC is yet to be more than just a toy-language, however in the design of the language and the actual compiler, we have tried to focus making the code modular and easy to extend. Currently the program consists of four parts. Each part will carry out a seperate task for the program. We have focused on the making the code modular, such that each part does not explicitly dependent on the other part. Though the main interface is setup to the following tasks:

1. **Parser:** Extract the Abstract Syntax Tree generated in the Parser
2. **Evaluator:** Run a program with an initial store
3. **Verification Condition Generator:** Extract the formula by Verification Condition Generator
4. **External SMT-solver API:** Run the formular through an external prover (Z3).

Although each part can work seperately, they are connected as seen in Figure 4 in the application.

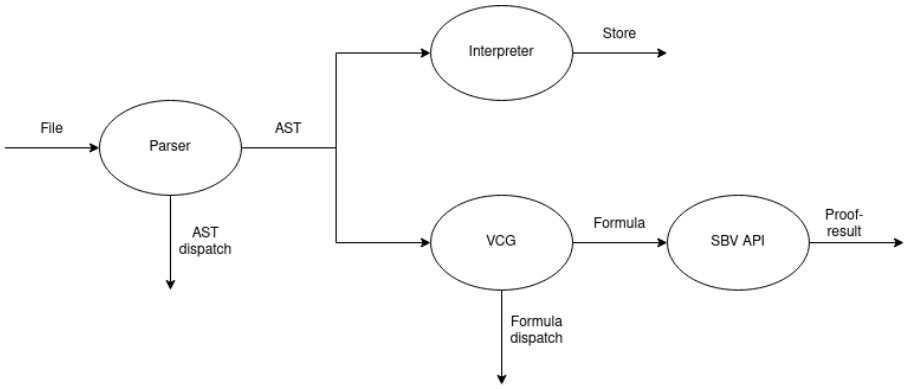


Figure 4: Application layout for IFC

Section 6 explains how to interact with the application. In the following sections we describe how each the four program parts are implemented. Section 3.5, describe an interface which allows for more general proofs about programs.

3.1 Parser

The internal AST is defined pretty close to the grammar in Table 1 and can be seen in Appendix **??**. Two matters are important to state about the parsing stage, ambiguity and ghost-variables.

3.1.1 Grammar ambiguity

For the parsing of a program we use the parser combinator library MegaParsec, which means we have to eliminate ambiguity in the grammar presented. We do so in two different ways. For arithmetic operators and boolean operators, we make use of the expression parser defined in `Control.Monad.Combinators.Expr`. This makes handling operators, precedence and associativity easy. Furthermore, it allows for easy extension with new operators. For parsing the first order logic in assertions, we found the expression-parser unfit. One reason is that we want to allow for syntax "forall x y z." to desugar to "forall x. forall y. forall z". Instead of using the expression-parser we manually introduce precedence, according to conventions, with negation most precedence, then conjuction and disjuction followed by quantifers and lastly implication. Furthermore the grammar has been left-factorized, as such. We further introduces some syntactic sugar in the grammar such as "if c {s};" which will be desugared into "if c {s} else {skip};". However we dont need any special transformation other than using the `option` parser-combinator.

For other parts of the grammar, which could have been syntactic sugar, such as implication and exist, we use the unsugared constructs. Although this introduces more code, the intent is to make the code easier to reason about in terms of the semantics. Furthermore it reads better in the output of the vc generator, and hence translate more directly into the predicate transformer semantics. This at least has made the development process easier. Ideally this could be alleviated by a more comprehensible pretty-printer, but at the moment, we settle for a slightly bigger AST.

$\langle imp \rangle$ ::= $\langle quant \rangle$ $\langle imp' \rangle$

$\langle imp' \rangle$ ::= '$\Rightarrow$' $\langle quant \rangle$ $\langle imp' \rangle$ | ϵ

$\langle quant \rangle$ ::= 'forall' $\langle vname \rangle$ '.' $\langle quant \rangle$
| 'exists' $\langle vname \rangle$ '.' $\langle quant \rangle$
| $\langle cd \rangle$

$\langle cd \rangle$ ::= $\langle neg \rangle$ $\langle cd' \rangle$

$\langle cd' \rangle$ ::= '$\wedge$' $\langle neg \rangle$ $\langle cd' \rangle$ | '$\vee$' $\langle neg \rangle$ $\langle cd' \rangle$ | ϵ

$\langle neg \rangle$ ::= '$\sim$' $\langle factor \rangle$ | $\langle factor \rangle$

$\langle factor \rangle$ ::= $\langle bexp \rangle$ | '(' $\langle imp \rangle$ ')'

3.1.2 Ghost variables

As previously mentioned, it is semantically disallowed to use ghost-variables anywhere in the program-logic other than assertions. Although this is a semantical matter, we handle this in the parser. By simply fail to parse if a ghost variable appears anywhere not allowed, thus treating it as a syntactic issue. We do so by adding a reader monad-transformer, to the internal transformer type `parsecT` of `Megaparsec`.

```
type Parser = ParsecT Void String (ReaderT Bool Identity)
```

The boolean value in this environment will tell if the next parser (by the use of `local`) must allow for parsing ghost variables. Which will certainly only happen in assertions.

3.2 Evaluator

The evaluator follows directly from the semantics presented in Section 2.1. That is the intial store provided to the evalutor will be modified over the course of the program according to the semantics. We define the `Eval` type as follows such:

```
type STEnv = M.Map VName Integer
type Eval a = RWST () () STEnv (Either String) a
```

At the moment there is no use for neither `reader` nor `writer`, however when the language in the future is extended to have procedures the reader monad will be a natural choice for the scoping rules of said procedures. Likewise it is highly likely that the language would supports some sort of IO in the future, and atleast being able to print would be nice. The store described in Section 2.1 is kept in the State `STEnv`. The Store is simply a map from VName to Integers. We want the store to be a State as the store after one monadic action should be chained with the next monadic action. This ensures that all variables are in scope for the rest of the program and hereby entails mutability. Duely note that ghost variables will also reside in this environment but will not be mutable or even callable as previously explained. We make use of the error monad to resolve any runtime-errors that would arise, that is if a violation is done, a ghost is assigned, a variable is used before it is defined or if undefined behaviour arises such as division by 0. Semantically, the first error that occurs will be the return value

of the computation.

Each type defined in the AST, `Stmt`, `FOL`, `AExpr`, `BExpr` is evaluated by different functions which all operate under the `Eval` monad, which allows for a clean and modular monadic compiler. They have the following types.[1]

```
eval :: Stmt -> Eval ()
evalFOL :: FOL -> Eval Bool
evalBExpr :: BExpr -> Eval Bool
evalAExpr :: AExpr -> Eval Integer
```

From this we can notice that all of the different constructs except for `Stmt`, will produce a value, whereas `Stmt` can only produce monadic actions, in terms of modifying the store. This monadic context allows us to translate the operational semantics almost directly into haskell code.

```
1  eval :: Stmt -> Eval ()
2  eval (Seq s1 s2) = eval s1 >> eval s2
3  eval (GhostAss vname a) =
4      get >>= maybe (update vname a) (const _e) . M.lookup vname
5  eval (Assign vname a) = update vname a
6  eval (If c s1 s2) =
7      evalBExpr c >>= \c' -> if c' then eval s1 else eval s2
8  eval (Asst f) = evalFOL f >>= \case
9    True  -> return ()
10   False -> _e
11 eval w@(While c invs _var s) =
12   evalFOL invs >>= \case
13     False -> _e
14     True -> evalBExpr c >>= \case
15       True -> eval s >> eval w
16       False -> return ()
17 eval Skip = return ()
18 eval Fail = _e
```

Figure 5: Evaluator for statements[2]

The most complex argument for equivalence between the code and the semantics is the "while" construct. Figure 5, line 11-16, shows how we evaluate it. We first evaluate the invariant, to directly follow the semantical rules. Hence, when the invariant does not hold, we handle this case as abnormal termination, as per rule *while-i-false*, similarly to how we would do for a standard assertion. If the variant holds, but the loop-condition does not, we do nothing, as per rule *while-false*. Lastly if both the invariant and the loop-condition both holds true, we will evaluate the body and then, we evaluate the while loop again. The other statements are simple and we treat them similarly to "while", directly in accordance with the semantics.

One important note about the interpreter is that we have no good way of checking assertions which includes quantifiers. The reasons is that we wanted (possibly erronously) to support arbitrary precision integers. This entails that we currently has no feasible way to check such assertions. A potential solution would be to generate a symbolic reprensentation of the formula and try to satisfy it by using an external prover. Though the interpreter would then also require external dependencies, and not be a standalone

[1] Can we use a typeclass so they all are named eval??? Or is that too Rusty???

program any longer. All in all, this is unideal, and currently the approach is to "ignore" such assertions by considering them true. In **??** we describe a potential extension to IFC, which could help alleviate this problem. Non-qunatified assertions, will still be evauluated as per the operational semantics.

3.3 Verification Condition Generator

The verification condition generator, uses the weakest precondition calculus to construct the condition, or weakest liberal precondition (if while has no specified variant). Like the approach in the evaluator we want to be able to chain the actions and include a state and reader environment in the construction of the verification condition.

```
type Counter = M.Map VName Count
type Env = M.Map VName VName
type WP a = StateT Counter (ReaderT Env (Either String)) a
```

The Counter state is used to give unique names to variables. The purpose of the reader environment is to resolve variable substitution in the formula generated from the weakest preconditions.

As described in Section 2.3, we use the quantified rule for assignment, when encountering assignments, to easily handle multiple occurences of the same variable. When developing the code for the VC generator our initial approach tried to minimize the number of times we had to resolve condition Q, but our approach to this showed failed. The working solution on the other hand is naive.

3.3.1 Failed attempt

[3] As mentioned we tried to minimize the number of times we had to traverse condition Q. The initial approach tried to resolve Q only after the entire formula were build. This would allows for a sentence with all names resolved in only two passes, one over the imperative language AST and one over the structure of the formula. The approach was intended to build up a map as such, where `VName` is an identifier:

```
type Env1 = M.Map VName [VName]
```

Whenever encountering a variable we would add a unique identifier to its value-list along with extending Q by the WP rules, as such:

```
missing
```

The result of running WP would then give a partially resolved formula, meaning that all the bound variables introduced in the quantifiers of the assignment rule will be correctly resolved. All other variables, would have their original name, and hence at this point be free.
The second traversal will the be on this partly resolved formula. Whenever encountering a variable x, we would then look it up in the Map, and then replace x with the head of the list, since this must have been the last introduced bound variable. When encountering a quantifier (except for the first for each free variable), we would then "pop" the head of the appropriate list, as any later occurence of x would be substituted with this newly found variable. The problem with this approach is how the AST for First Order Logic formula is constructed. Consider the following example:

[3] should this even be included?

```
r := 5
r := r + 10
```

which after popping r_1 of the list would then look like

$$\forall r_1 \,.\, r_1 = 5 \Rightarrow \forall r_2 \,.\, r_2 = r + 10 \Rightarrow true$$

This means that we can no longer substitute in r_1 when encountering the next r in $r_2 = r + 10$. On the other hand if we dont "pop" as soon as r_2 is encountered we have no information on when to do so, as it is not necessary the case that there appears an r in the equality before the implication. When this problem arose we turned to the naive solution.

3.3.2 Successful attempt

The second and current approach is to resolve Q whenever we encounters an assignment. We generate the forall as such:

```
wp (Assign x a) q = do
    x' <- genVar x
    q' <- local (M.insert x x') $ resolveQ1 q
    return $ Forall x' (Cond (RBinary Eq (Var x') a) .=>. q')
```

We make a new variable (by generating a unique identifier, based on the State), then we proceed to resolve q with the new environment, such that every occurence of variable x will be substituted by the newly generated variable x'. The `Aexpr` which x evaluates to should not be resolved yet, as this will potentially depend on variables not yet encounted. Ghost variables on the other hand are easy to resolve as they need not be substituted, because of immutability.

The current version does not enforce the formula to be closed, although it will be necessary in the generation of symbolic variables. Although easily fixed by a simple new iteration over the AST of the formula, and checking if any non-ghost variable does not contain a #, we find that since the formula is intended to be fed to the next stage in the compiler it is uneccesary to do so.

3.3.3 While - invariants and variants

The `while` statement has the most complicated Weakest Precondition. The code for computing wp for a `while`-loop is presented in Figure 6.
We have tried to make the code generic in terms existence of the variant to eliminate code duplication. line 6-9, will generate the conditions needed for the variant. In case no variant is defined we use that predicate-logic and conjuction forms a monoid with $\top$ as identity element, such that, we generate no new $\forall$-quantification, and have subformular $b \Rightarrow WP(s, b)$. Is there a variant, we generate the equality $\xi = v$, along with the well-founded relation for unbounded integers. This approach should translate pretty well, with possible other types that have a well-founded relation. The rest of the code simply checks which variables are assigned in the body of the while-loop, and generate a variable for each. Collectively this code will generate the weakest precondition for a while statement as per described in equation 1.

3.4 proof-assistant API

The proof-assistant API uses the SMT Based Verification library (SBV), which tries to simplify symbolic programming in Haskell. The library is quite generic and extensive

```
wp (While b inv var s) q = do
  st <- get
  (fa, var', veq) <- maybe (return (id,
                            Cond $ BoolConst True,
                            Cond $ BoolConst True)) resolveVar var
  w <- wp s (inv ./\. var')
  fas <- findVars s []
  let inner = fa (((Cond b ./\. inv ./\. veq) .=>. w)
                          ./\. ((anegate (Cond b) ./\. inv) .=>. q))
  env <- ask
  let env' = foldr (\\(x,y) a -> M.insert x y a) env fas
  inner' <- local (const env') $ resolveQ1 inner
  let fas' = foldr (Forall . snd) inner' fas
  return $ inv ./\. fas'
  where
    resolveVar :: Variant -> WP (FOL -> FOL, FOL, FOL)
    resolveVar var = do
      x <- genVar "variant"
      return (Forall x,
              Cond (Negate (RBinary Greater (IntConst 0) (Var x)))
               ./\. Cond (RBinary Less var (Var x))
             , Cond (RBinary Eq (Var x) var)
             )
```

Figure 6: Weakest precondition for `while`

compared to what we use it for. We mostly make use of the higher level functions and dont mess with any of the internals, however the default type of SBV does not quite fit our needs. Instead we use the provided transformer `SymbolicT`, such that we can embed the `Except` monad. We want to do so, as when iterating over the formular, we might encounter a variables not yet defined and here fail gracefully, instead of throwing an error. The code which generates a `Predicate ~ Sym SBool` is simple, since the formula generated in the previous stage, will be already first order logic formula, which straight forwardly can be converted into SBV's types. For the entire highlevel logic we resolve it as simple as this:

```
type Sym a = SymbolicT (ExceptT String IO) a
type SymTable = M.Map VName SInteger

fToS :: FOL -> SymTable -> Sym SBool
fToS (Cond b) st = bToS b st
fToS (Forall x a) st = forAll [x] $ \(x'::SInteger) ->
  fToS a (M.insert x x' st)
fToS (Exists x a) st = forSome [x] $ \(x'::SInteger) ->
  fToS a (M.insert x x' st)
fToS (ANegate a) st = sNot <$> fToS a st
fToS (AConj a b) st = onlM2 (.&&) (`fToS` st) a b
fToS (ADisj a b) st = onlM2 (.||) (`fToS` st) a b
fToS (AImp a b) st = onlM2 (.=>) (`fToS` st) a b
```

And equally easy it is to resolve `bexpr` and `aexpr`. Ideally we would add a ReaderT to the transformer-stack to get rid of the explicit state. We have not been able to resolve the type for this, because of the following type constraint `forAll :: MProvable m a => [String] -> a -> SymbolicT m SBool`, and `m` must be an `ExtractIO`, of which Reader and State cannot be implemented[]. The predicate constructed by traversing the formula from the last stage will then be used as argument for the SBV function `prove`, which will try to prove the predicate using Z3. If the program can

correctly be proved by the external SMT solver, the output will be `Q.E.D.` , whereas if the formula cannot be proved, a falsifiable instance of the variables will be presented. For instance the output of the following program will obviously always be falsified.

```
violate;
```

whereas the multiplication program in Figure 2 is proveable.

3.5 Interface for proofs

As may have been apparant from the example programs presented earlier in this report, we requires each IFC program to have a header that looks as follows:

```
vars: [ <variables> ]
requirements: { <preconditions> }
<!=_=!>
```

where ¡variables¿ describes the variables, which is initially in the "store" and ¡preconditions¿ is an assertion, which specifies a condition that should hold before the program starts. The header does not provide anything new to the evaluator, (although we prepend the requirement to the program as an assertion), but it enables us to make more generic proofs about said programs. In the current state, there is no procedures in IFC, which makes it difficult to reason about the input of variables, when trying to prove the weakest precondition, hence why we define said header. The inspiration comes from how the whyML language defines procedures. Figure 7 is a whyML program equivalent to the mult.ifc program. It is possible for why3 to generate a

```
module Mult

  use int.Int
  use ref.Refint

  let mult (&q : ref int) (r: int) : int
    requires { q >= 0 && r >= 0 }
    =
    let ref res = 0 in
    let ghost a = q in
    while q > 0 do
      invariant { res = (a - q) * r && q >= 0}
      variant { q }
      decr q;
      res += r
    done;
    assert { res = a * r };
    res
end
```

Figure 7: Why3 program equivalent to mult.ifc

vector of input variables and then a precondition for each the `requires`, such that $\forall x_1, ..., x_n.\ requires \Rightarrow WP(body, ensures)$. That is whenever the requires holds, then the weakest precondition of the body should hold, where ensures states the post-condition. Notice that Figure 7 does not use an "ensures". We define it this way as it more closely resemples our language. And since we dont have any return values, we dont really need the ensures, since this might as well be part of the actual program.[4]

[4] is this clear if you dont know whyML?

4 Assessment

In this section, we make an assessment of the correctness of our implemention of the interpreter and verification condition generator for *While*. We also assess the robustness and maintenance of the implementation, especially regarding any possible extension of the implementation. To perform the assessment, we have designed a test suite consisting of both automated tests, blackbox tests, and test programs. In Section 4.1, we present the experiments conducted as part of the testing strategy, and in Section 4.2 we assess the code based on the results of the tests, and perform an evaluation of the work as a whole.

4.1 Experiments

As described above, we have conducted a variety of tests to assess the implementation, consisting of both automatic tests and blackbox tests. To thouroughly test our implementation, the tests follow this testing strategy:

- The *Parser* is tested using a blackbox approach, which should cover all aspects of the grammar presented in Table 1. Furthermore, we have a QuickCheck test suite for property based tests, which can generate ASTs. This is used to test the *Parser* by generating an AST, and then comparing the result of parsing the pretty printed AST with the actual AST.
- We have a test suite using property based testing to test the semantics of the *While* language, which compares the result of running the *Evaluator* on two semantically equivalent generated program constructs.
- We have attempted to use property based testing to compare evaluation of generated programs with the result of using our VC-generator coupled with *Z3* on the same program.
- Finally, we have used QuickCheck to generate random input for our example programs, which we use to run the programs and compare with the result of running the same computation in Haskell. This is to assert that the example programs does indeed do as we expect them to, teling us whether they should be provable or not.
- TODO: some tests about VC and SAT solving the example programs

Besides the tests suites, we have based our assessment on example programs, which can be found in the `examples` folder. Of these some works correctly and some intentionally dont. In Table 4, each program can be seen along with a short description of what they do, and what the results of running them through the *Evaluator* and *VC generator* is. The leftmost column lists the name of the example programs, the midter column describes the functionality of the program, and the roghtmost column states how both the *Evaluator* and *VC generator* behaves with the program as input. We will analyse the table more thoroughly in Section 4.1.3.
In the next subsections we will present in-depth the different parts of our testing strategy. First we present the strategy of our blackbox *Parser* tests. Next we present our approach to generating automatic tests with property based testing. Finally we go into details with the example programs, and how they are designed to test the functionality of the implementation.

program	description	results
always_wrong	This program is simply a `violate` statement and should thus always fail.	The evaluator evaluates to `false`, and the VC generator gives a falsifiable verification condition.
assign	Assigns values to two variables, and asserts that they get assigned correctly.	The evaluator gives the expected result, and the VC genration gives a solvable verification condition.
collatz	This program calculates the length of collatz-conjecture until the repeating pattern $1, 4, 2, 1, ...$	TODO: something cooler
div	Takes as input two variables a, b and computes the euclidean division of a by b.	The evaluator gives the expected result, and the VC generator generates a provable verification condition.
fac	Takes as input an integer r and computes $r!$.	TODO: fix invariant and variant ala sum program maybe
fib	Takes as input an integer a, and computes th a'th Fibonacci number.	The evaulator gives the expected result, but the VC generator falsifies the computed verification condition.
mult	Takes as input two variables q, r and multiplies them.	The evaluator gives the expected result, and the VC generator computes a provable verification condition.
max	Takes as input two variables x, y and find the maximum of the two.	The evaluator gives the expected result, and the VC generator computes a provable verification condition.
skip	Asserts that the store is unchanged when executing a `skip` command.	The evaluator gives the expected result, and the VC generator computes a provable verification condition.
sum	Takes as input a positive integer n and computes the sum from 1 to n.	The evaluator gives the expected result, and the VC generator computes a provable verification condition.
generic_sum	Takes as input three integers $cur, lim, step$ and computes the sum of all numbers in a list starting in cur, stepping $step$, up to lim.	TODO: clean-up code and run tests

Table 4: Overview of example programs

4.1.1 Blackbox testing

To assert that our implementation correctly parses input programs, we have designed a blackbox test suite for systematically testing each construct of the grammar. The strategy is to test parsing of smaller constructs such as variable names and expressions, and then combine them into larger constructs such as statements. Taking a very systematical approach we jope to cover all aspects of the grammar.
Furthermore, we use blackbox tests to assert that the *Parser* handles ambuguities, associativity and precedence correctly.
Finally, we have designed both positive and negative nests for the *Parser*, to assert that it behaves as intended when given both good and bad input.
TODO: credit to Andrzej Filinski for nice tests :)

4.1.2 Quickchecking instances

To further test certain properties of the implementation, we build a test suite consisting of property based tests. With these automatic tests we attempt to verify that

- That the *Parser* correctly parses larger program constructs.
- That the *Evaluator* correctly follows the semantics of the *While* language.
- That the result of running a program through the *VC generator* is equivalent to running it through the *Evaluator*.

To do this, we build generators for generating suitable input for the tests, and next we find appropriate properties to test using input from the generators.

Generating input. TODO: skriv dette færdigt. Using the QuickCheck interface, we define instances of *Arbitrary* for the different AST types. When doing this, there are certain important considerations. Firstly, to ensure that the size of the generated expressions do not explode, we use *sized expressions* to control expansion, and to ensure that we get a good distribution of the various constructs we use *frequency* to choose between them. Secondly, we want the number of possible variables to be limited, such that the *Evaluator* will not fail too often, by using variables that have not yet been defined. This is done by limiting the options for variable names to be only single character string. Thirdly, while-loops might not terminate, hence we want to define a small subset, or a skeleton, for while-loops that we can be sure terminates.
To accomodate this third consideration, we build a skeleton for while-loops that should always terminate. This is done by defining three versions of while-loops as shown in Figure 8. TODO: put in correct code!!!
TODO: koden virker ikke lige nu, vi kommer tilbage til det snart :))

QuickCheck properties. Now that we have investigated how to generate input for the tests, we move on to finding meaningful properties to test. In this test suite we use property based testing for the following:

- **Parser tests.** To complement the blackbox tests for the parser, we use automatic testing to generate ASTs and assert that these programs are parsed correctly. This is done by generating an AST, and then using a pretty printer to convert it into a program that can be given as input to the parser. It is then checked whether the result of parsing this program is equal to the original generated AST. This is

```
whileConds = elements $ zip3 (replicate 4 ass) [gt, lt, eq] [dec, inc,
    change]
  where v = whilenames
        v' = Var <$> v
        ass = liftM2 Assign v arbitrary
        gt = liftM2 (RBinary Greater) v' arbitrary
        dec = liftM2 Assign v (liftM2 (ABinary Sub) v' (return $
    IntConst 1))
        lt = liftM2 (RBinary Less) v' arbitrary
        inc = liftM2 Assign v (liftM2 (ABinary Add) v' (return $
    IntConst 1))
        eq = liftM2 (RBinary Eq) v' arbitrary
        change = liftM2 Assign v arbitrary
```

Figure 8: Generation of while-loops using a skeleton.

supposed to assert that the parser can handle a lot of different combinations of constructs, potentially finding bugs that would not have been discovered through our systematic blackbox testing of simple constructs.

- **Semantic equivalence.** To test whether the evaluator corretly implements the semantics of the *While* language, we have tested certain equivalence properties. From studying the semantic system for *While*, we have designed equivalence properties according to the semantics. Examples of such equivalence properties are `if true then s1 else s2` $\sim$ `s1` and `while false do s` $\sim$ `skip`. These are designed to cover all the cases of the small-step semantics, and the properties uses generators for generating suitable input variables and statements, i.e. the body of an if-statement is generated automatically, but the equivalence relation is defined manually.

- **Evaluating a program vs solving with VC generator and Z3.** When given an input program, it should always be true that if the *VC generator* computes a provable verification condition, then the *Evaluator* will evaluate all assertions in the program to *true*. To test this, we generate random programs and assert that if the *VC generator* can verify the program, then the *Evaluator* will evaluate to *true*. Here we use generators to generate input for the test. However, testing this automatically is quite a complex situation, since we need to be able to have a strong enough loop-invariant to prove the correctness of the program. It has come to our attention that the generated programs are not very useful, and currently we have not been succesful in implementing such a property-test. We will come back to this in the assessment in Section 4.2. It should also be noted that this is only true in one direction. We discuss why in Section 4.1.3.

The above bulletpoints presents the intuition behind the QuickCheck testing of the implementation. A presentation of the test results and assessment of the code will be given in Section 4.2.

4.1.3 Dynamic execution compared to static proofs

Besides the QuickCheck and blackbox testing, we use example programs to check the quality of our implementation. In this subsection we present some of the example programs written for testing, and describe how and why they are interesting. Next we set

forth the experiments that we conduct using the programs as input to both the *Evaluator* and the *VC generator*. Finally we explain how we compare dynamic execution to static verification of the programs.

Example programs. In Table 4 we present an overview of the example programs written to test the implementation. The program `always_wrong` tests the `violate` construct, and simply checks whether the program correctly fails. The `skip` program tests the `skip` command, by asserting that the store is unchanged after executing the command. The programs `assign` and `max` are very simple too, and computes the result without use of while-loops, thus testing simple statement constructs.
The rest of the example programs are more interesting, as the use while-loops, and thus requires stronger assertions to prove correctness and termination of the programs. The following present some of them, what they test, and why they are interesting.

- `mult.ifc`. This is the example that we use throughout the report, because it is a showcase of a sequence of statements including assignments, assertions, use of ghost variables, and while-loops with both an invariant and a variant. Thus the program tests a simple combination of statements, and includes a while-loops that always terminates, and therefore the program should be provable. An interesting thing about the program is to investigate whether the program is provable with the given invariant, and whether a variant is needed to prove correctness. The code for `mult.ifc` is shown in Figure 2.

- `generic_sum.ifc`. The idea behind this program is to make a generic while-loop summing up a list of values. The program takes as input an offset, a step and a limit, and then sums up all the values of a list starting from the offset, stepping with the given value up to the limit. This is part of an experiment to test more types of while-loops. Provided a loop skeleton, we can generate random statements for the loop body, and in that way test various kinds of while-loops that are ensured to terminate. However, this program has such a strong loop condition compared the the postcondition, that it will always be provable. The goal is to find a loop skeleton which provides just enough information so that the invariant is crucial for proving correctness. If the variant is needed for proving correctness as well, the skeleton is ideal for generating while-loops.

- `collatz.ifc`. The idea behind this program is to have a while-loop that does not necessarily terminate. This can tell valuable things about attempting to prove non-terminating programs. The program utilises assertions in a way that ensures that the postconditon only holds if the loop has terminated. Therefore, a loop variant is necessary to determine whether the while-loop terminates or not.

Experiments with example programs. We have mainly used the example programs for conducting two types of tests: 1) testing that the evaluator can correctly evaluate a program, and 2) testing that the programs that are provable using the VC generator will also evaluate to *true* in the evaluator.
The first type of test, testing the evaluation of programs, were conducted by generating random input for the example programs, and then asserting that the result was in fact what we expect. For examle, when evaluating the `mult.ifc` program with two random values, the result should be equal to the result of multiplying the two values in Haskell. It should be noted that we use generators for generating meaningful input to the programs, to be able to test with all kinds of valid input.

$$
\begin{array}{l}
1\ \forall q, r.\ (q \geq 0 \wedge r \geq 0) \Rightarrow \\
2\quad \forall res_3.\ res_3 = 0 \Rightarrow \\
3\quad\quad \forall \$a.\ \$a = q \Rightarrow \\
4\quad\quad\quad (res_3 = (\$a - q) * r \wedge q \geq 0) \\
5\quad\quad\quad \wedge \forall q_2, res_2, \xi_1. \\
6\quad\quad\quad\quad (q_2 > 0 \wedge res_2 = (\$a - q_2) * r \wedge \xi_1 = q_2) \Rightarrow \\
7\quad\quad\quad\quad\quad \forall res_1.\ res_1 = res_2 + r \Rightarrow \\
8\quad\quad\quad\quad\quad\quad \forall q_1.\ q_1 = q_2 - 1 \Rightarrow \\
9\quad\quad\quad\quad\quad\quad\quad (res_1 = (\$a - q_1) * r \wedge 0 \Leftarrow \xi_1 \wedge q_1 < \xi_1) \\
10\quad\quad\quad \wedge ((q_2 \leq 0 \wedge res_2 = (\$a - q_2) * r) \Rightarrow res_2 = \$a * r)
\end{array}
$$

Figure 9: Generated verification condition for the modified `mult` program.

The second type of test asserts that if programs can be proved correct, then they will evaluate correctly as well. This is tested by first feeding the programs to the VC generator coupled with Z3, and then to the evaluator with random input. It is then asserted that if the program is provable, then the evaluator evaluates all assertions to *true*. If the program is falsified, then the test returns *true* no matter what the result of the evaluator is. Ideally we would extract the counter example from the prover, and assert that the evaluator fails with the given input values. If this does not hold, the assertions are not sufficient to prove correctness of the given program. However, right now the test implements the simple approach, and does not extract the counter example. This property is also addressed in the next paragraph, where we describe why this is only true in one direction.

Provable with VC generation ensures successful evaluation. As described above, an important property on the relation between the dynamic execution of a program through the evaluator, and the static proof of said program, is that if we can correctly prove the correctness of a program, then the dynamic evaluation should also hold. However, the implication does not hold in the other direction. The reason for this is that we might not have provided strong enough assertions to satisfy the generated verification condition, whilst the dynamic execution just needs all assertions to evaluate to *true*. If for example we have a program that uses *true* as a loop invariant, this will hold in each iteration during the dynamic execution, but will probably not be enough to prove any postcondition statically.

Lets take a closer look at the multiplication example program from Figure 2. We have previously argued that the code is correct and can correctly be proved by Z3, but if we relax some of the assertions in the program, this will no longer be the case. Consider exchanging the loop-invariant `?{res = ($a - q) * r /\ q >= 0}` with the looser invariant `?{res = ($a - q) * r}`. Now Z3 will no longer be able to prove the correctness of the program. The generated formula looks as shown in Figure 9.

It becomes quite apparant that the invariant is no longer strong enough to prove the condition, since the restriction on q_2 is too weak. Z3 gives us a counter example where $q_2 = -3$, $res_2 = 6$, $\$a = 0$ and $r = 2$. The two first conjuctions in line 4 and line 5-9 will evaluate to *true*. In the last term in line 10, the RHS of the implication will evaluate to *true* as

$$(-3 \leq 0 \wedge 6 = 3 * 2)$$

but the LHS will not evaluate to *false*, as

$$6 \neq 0 * 2$$

But we know that the program does in fact behave as intended, so is it correct that the prover falsifies the formula? Once again we can verify that our program acts correctly, by doing the same modification to the whyML program in Figure 7. Trying to prove the program correctness gives a falsifiable counter example, as expected. By this, we have confidence that our implementation works correctly, and requires the necessary loop invariants.

4.2 Evaluation

In this section we give a collective assessment

4.2.1 Correctness

Through our testing strategy, we have found that our code is correct.

4.2.2 Completeness

4.2.3 Robustness

We have found bugs, such as the weird thing where modulo 0 is not an error in Z3.

4.2.4 Maintainability

As mentioned this project sould serve as a base, thus we have valued maintainability and extensibility highly. We have done so through the following abstractions. Firstly we are using monad-transformers to enforce seperation of concern. For the evalutor, we have even defined the *Eval* type in terms of *RWST* even though we only uses the *State*-transformer and underlying Either monad. We do so since we highly anticipate that the other monadic effects will be useful in the future, such as being able to print (using writer). Furthermore, the multistage approach, for generating verification conditions should also allow for easier extending possible SMT-backends. In the current solution we only support *Z3*, but allowing for the other SBV supported solvers, should be trivial. But because we generate the verification condition (which barebones are a simple ADT) in a seperate state it should not be too cumbersome to allow for other backends.

In terms of extending the language with new constructs we consider the following cases.

- If the construct can syntacticaly be reduce to one of the statements already defined, then only the parser would need to change.
- New statements, will require implementation in the parser, evaluator and the verification condition generator, but will most likely, not have to modify anything in the SBV API. Since the (toplevel) first order logic is already fully defined. Although this should never have to modify the already defined constructs.
- New types such as collections would require additions in all module but should not add too much extra complexity as to what new statements would.

- Adding a type-system would be a very useful addition, but will most likely be one of the biggest changes. As it would probably require additions to the constructors of the AST, or for an additional typed-AST and in this case the code for the already working parts would have to be modified.

All in all, we find that the code is maintainable and extendable, but some extensions will require some overhaul of the code.

5 Discussion and conclusion

5.1 Future work

5.1.1 Procedures and arrays

5.1.2 Type system

5.1.3 PER-logic for Information Flow Control

5.2 Conclusion

6 How To Use

References

[1] Edsger W. Dijkstra. “Guarded Commands, Nondeterminacy and Formal Derivation of Programs”. In: *Commun. ACM* 18.8 (August 1975), pp. 453–457. ISSN: 0001-0782. DOI: 10.1145/360933.360975. URL: https://doi.org/10.1145/360933.360975.

[2] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Commun. ACM* 12.10 (October 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259.