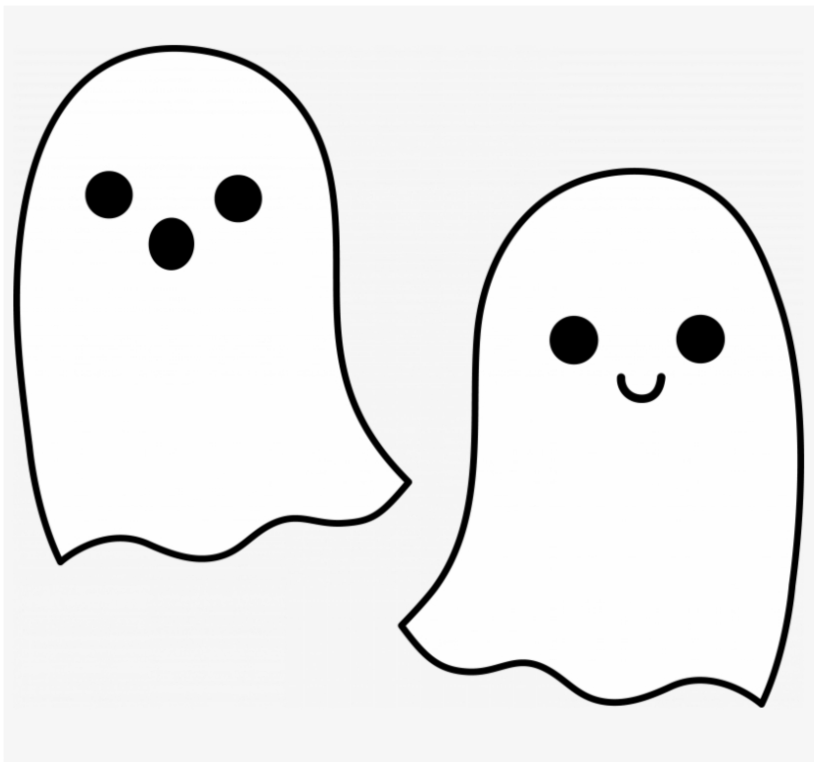
UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE



Project outside course scope

Jacob Herbst(mwr148), Matilde Broløs (jtw868)

IFC

An application for dynamic evaluation and static verification of programs

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 work of Robert W. Floyd, who presented a form of program verification using logical assertions in his paper from 1967[2]. Floyds work led to the axiomatic system presented by C.A.R Hoare in 1969, [3], also known as Floyd-Hoare logic. 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, gives 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!

2 Background

In this section, we present the grammar and semantics of the *While* language used throughout the project. Next, we explain Floyd-Hoare logic and verification condition generation through semantic predicate transformers and how this provides a basis for formal verification. 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 built-in assertion language, enabling the possibility of statically verifying programs by the use of external provers. The assertions also enable us to dynamically check the program during evaluation. This section presents the syntax and semantics of both the imperative language and the assertion language.
The language is simple, as the focus in this project has been to correctly generate verification conditions for said programs. Table 1 shows the grammar of *While*. In essence, a *While* program is a statement with syntax in the C-family. Although not strictly part of the language itself, we require a program to have a *header*, succeeded by one or more statements. We leave out the syntax of the header in the grammar but will describe it in detail in Section 3.5, where we explain why it is necessary as well. The syntax is straightforward and does, in general, not provide anything new. One of the interesting program constructs are while loops. A while loop consists of a loop condition, one or more invariants, and zero or one variant, followed by a loop body. The reason for the inclusion of loop invariants and variants will be apparent in Section 2.2. Another interesting construct is if statements, which can be constructed both with and without an else branch. Finally, assertion statements are of great importance for this project. An assertion is denoted by a hashtag and curly braces. We will explain more about assertions in Section 2.2 and Section 2.3.

Identifiers must start either with a lowercase symbol or an underscore, and may then be proceeded by 0 or more characters, digits or underscores. ghost-variables must start with either a ghost-emoji or for users more comfortable with ascii, a $, directly succeeded by an identifier.

Integers can be specified as decimal, hexadecimal, binary and octal.

While includes many common operators, which follows the usual conventions. For arithmetic expressions (`aexpr`) we have:

- Parentheses
- Negation
- Multiplication, division and modulo (left associative)
- Addition and subtraction (left associative)

where parentheses binds tightest. Operators on boolean expressions (`bexpr`) follow the standard precedence rules:

- (!) (negation)

$$
\begin{array}{lcl}
\langle \mathit{statements} \rangle & ::= & \langle \mathit{statement} \rangle\ \text{';'}\ \langle \mathit{statements} \rangle \mid \epsilon \\
\\
\langle \mathit{statement} \rangle & ::= & \langle \mathit{ghostid} \rangle\ \text{':='}\ \langle \mathit{aexpr} \rangle \\
 & \mid & \langle \mathit{identifier} \rangle\ \text{':='}\ \langle \mathit{aexpr} \rangle \\
 & \mid & \text{'if'}\ \langle \mathit{bexpr} \rangle\ \text{'\{'}\ \langle \mathit{statements} \rangle\ \text{'\}'} \\
 & \mid & \text{'if'}\ \langle \mathit{bexpr} \rangle\ \text{'\{'}\ \langle \mathit{statements} \rangle\ \text{'\}'}\ \text{'else'}\ \text{'\{'}\ \langle \mathit{statements} \rangle\ \text{'\}'} \\
 & \mid & \text{'while'}\ \langle \mathit{bexpr} \rangle\ \langle \mathit{invariant} \rangle\ \langle \mathit{variant} \rangle\ \text{'\{'}\ \langle \mathit{statements} \rangle\ \text{'\}'} \\
 & \mid & \text{'\#\{'}\ \langle \mathit{assertion} \rangle\ \text{'\}'} \\
 & \mid & \text{'skip'} \mid \text{'violate'} \\
\\
\langle \mathit{invariant} \rangle & ::= & \text{'?\{'}\ \langle \mathit{assertion} \rangle\ \text{'\}'} \\
 & \mid & \langle \mathit{invariant} \rangle\ \text{';'}\ \langle \mathit{invariant} \rangle \\
\\
\langle \mathit{variant} \rangle & ::= & \text{'!\{'}\ \langle \mathit{aexpr} \rangle\ \text{'\}'} \mid \epsilon \\
\\
\langle \mathit{assertion} \rangle & ::= & \text{'}\forall\text{'}\ \langle \mathit{identifier} \rangle\ \text{'.'}\ \langle \mathit{assertion} \rangle \\
 & \mid & \text{'}\exists\text{'}\ \langle \mathit{identifier} \rangle\ \text{'.'}\ \langle \mathit{assertion} \rangle \\
 & \mid & \text{'}\sim\text{'}\ \langle \mathit{assertion} \rangle \\
 & \mid & \langle \mathit{assertion} \rangle\ \langle \mathit{assertionop} \rangle\ \langle \mathit{assertion} \rangle \\
 & \mid & \langle \mathit{bexpr} \rangle \\
\\
\langle \mathit{assertionop} \rangle & ::= & \text{'}\wedge\text{'} \mid \text{'}\vee\text{'} \mid \text{'}\Rightarrow\text{'} \\
\\
\langle \mathit{bexpr} \rangle & ::= & \text{'true'} \mid \text{'false'} \mid \\
 & \mid & \text{'!'}\langle \mathit{bexpr} \rangle \\
 & \mid & \langle \mathit{bexpr} \rangle\ \langle \mathit{bop} \rangle\ \langle \mathit{bexpr} \rangle \\
 & \mid & \langle \mathit{bexpr} \rangle\ \langle \mathit{rop} \rangle\ \langle \mathit{bexpr} \rangle \\
 & \mid & \text{'('}\ \langle \mathit{bexpr} \rangle\ \text{')'} \\
\\
\langle \mathit{bop} \rangle & ::= & \text{'\&\&'} \mid \text{'||'} \\
\\
\langle \mathit{rop} \rangle & ::= & \text{'}<\text{'} \mid \text{'}\leq\text{'} \mid \text{'='} \mid \text{'}\neq\text{'} \mid \text{'}>\text{'} \mid \text{'}\geq\text{'} \\
\\
\langle \mathit{aexpr} \rangle & ::= & \langle \mathit{identifier} \rangle \\
 & \mid & \langle \mathit{ghostid} \rangle \\
 & \mid & \langle \mathit{integer} \rangle \\
 & \mid & \text{'-'}\langle \mathit{axpr} \rangle \\
 & \mid & \langle \mathit{aexpr} \rangle\ \langle \mathit{aop} \rangle\ \langle \mathit{aexpr} \rangle \\
 & \mid & \text{'('}\ \langle \mathit{aexpr} \rangle\ \text{')'} \\
\\
\langle \mathit{aop} \rangle & ::= & \text{'}+\text{'} \mid \text{'}-\text{'} \mid \text{'}*\text{'} \mid \text{'/'} \mid \text{'\%'} \\
\\
\langle \mathit{ghostid} \rangle & ::= & \text{ghost}\ \langle \mathit{identifier} \rangle \mid \$\langle \mathit{identifier} \rangle
\end{array}
$$

Table 1: Grammar of IFC

- (&&) (right associative)
- (||) (right associative)

and the *First Order Logic* operators used in assertions follow the following precedence rule:

- $\sim$ (negation)
- $\wedge$ (right associative)
- $\vee$ (right associative)
- $\forall$ and $\exists$
- $\Rightarrow$ (right associative)

where $\sim$ binds tightest. Relational operators works on *aexpr* and is non-associative.

An important point about the syntax is that we want to make a clear distinction between boolean expressions in the *While* language and the *First Order Logic* used in assertions. Thus, only one is allowed based on the context despite multiple ways of expression and conjunction. We find that this both makes it more evident when we are dealing with assertions and when we are dealing with actual program logic. Furthermore, it eliminates any complications of mixing operators.
In Table 2 we show the semantics of the different statements in *While*. The program variables are kept in a store, mapping variables to integers. If undefined behavior occurs, we have an abnormal termination. Thus the evaluation of commands goes from a store to a (potentially new) store or an abnormal termination, denoted by $\lightning$.
We have left out the semantics for arithmetic and boolean expressions, as these follow the standard semantics. However, it should be noted that we do not allow division and *modulo* by zero. Thus, if one of these occurs, the program will terminate with an abnormal store. Furthermore, ghost variables can only occur in assertions and not as part of the program logic. We will come back to this in Section 2.2.
The semantics describe how to evaluate statements in *While*. The most interesting semantic rules in our language are the ones for while loops. Here we have to assert that the loop invariant holds both at the beginning of each loop iteration and after loop termination. If the invariant ever evaluates to *false*, the program terminates abnormally.

```
1 res := 0;
2 while (q > 0) {
3       res := res + r;
4       q := q - 1;
5 };
```

Figure 1: Example program `mult.ifc`

Multiplication as an example. To show the language in action, Figure 1 presents a small example program computing the multiplication of two integers q and r. Line 2 shows the syntax of assignments, lines 3-6 show the syntax of while loops, and the entire program demonstrates a sequence of statements. For now, we leave out the assertions, but we will come back to these in Section 2.2.
Now even though this small *While* language is very simple, it is still very interesting as the base language for our project. By creating an assertion language that can prove the

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

Table 2: Semantics for the *While* language.

correctness of programs written in this simple language, we can investigate the possibility of proving termination and correctness of while loops, which are definitely the tricky construct 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 prove the correctness, and ideally termination, of such program constructs. That is why this small language is well suited for this project.

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.

2.2.1 Assertions

For expressing properties of a program, we use *assertions*, which are a first order logic formula, which should hold at a specific time in a program. By combining these into a triple with a *precondition* P and a *postcondition* Q and a statement S we get:

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

This triple states that *if* the precondition P holds in the initial state, and *if* S terminates

```
1  vars: [q,r]
2  requirements: {q >= 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`

when executed from that state, *then* Q will hold in the state in which S halts. Thus the triple does not say anything about whether S terminates, only that if it terminates, we know Q to hold afterward. Moreover, if S does not terminate, then any Q will hold. These triples are called Hoare triples.

Logical variables vs program variables. When using assertions we differ between *program variables* and *logical variables*. Program variables are usable in the program logic and are mutable throughout the program. Sometimes we might need to keep the original value of a variable that has changed during 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.

Multiplication example. To show how we can use assertions, we look at the multiplication program presented before. The version with assertions is shown in Figure 2.

In line 1-2, we have the input variables listed, and the requirements for the input given. This is part of the program header, which we will describe in more detail later. The requirements state that q must not be 0 since we would never enter the while loop and get a correct result. Additional logic using if-else statements would alleviate this requirement for a complete function that would also work with negative values for q. We provide this example for simplicity. In line 10, we assert that the final result will be the original value of q multiplied by r. Here we use a ghost variable for keeping the original value of q. We also have an invariant and a variant in the while loop in line 6, but we will come back to the meaning of that in Section 2.2.3.

2.2.2 Axiomatic system for partial correctness

Hoare logic specifies an inference system for partial correctness of a program based on the semantics of 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 holds 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. 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 $false$.

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

Table 3: Axiomatic system for partial correctness of *While*.[5]

The syntax of our version of *While* deviates slightly from the syntax presented in Table 2 and Table 3. The reason why we have not depicted the exact syntax in the tables is that the use of curly braces obscured the meaning, beacuse of the conventional notation for Hoare Triples. However, it is clear that the two syntaxes are semantically equivalent.

2.2.3 Total correctness

The above states how the Hoare logic can prove partial correctness, but does not guarantee termination of loops. We want to be able to prove total correctness, meaning we want to prove termination. However, this indeed cannot be done. To verify that a program terminates, we need a more robust assertion in the form of a loop variant. The loop variant is an expression that needs to be subject to a well-founded relation. In our case, the only possible variant is an arithmetic expression that decreases with each loop iteration. 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 Figure 2). We use the following modified syntax to express the logic of using a variant for a while loop.

$$\frac{\{I \land e \land v = \xi\}s\{I \land v \prec \xi\} \quad wf(\prec)}{\{I\}\ \texttt{while}\ e\ \texttt{invariant}\ I\ \texttt{variant}\ v, \prec\ \texttt{do}\ s\{I \land \neg e\}} \tag{1}$$

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[4]:

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

With both invariants and variants, it is possible to prove total correctness of programs containing while loops, as we can now say something about termination of loops.

2.2.4 Soundness and Completeness

For obvious reasons it is important that the axiomatic system is sound. This means that any property that can be proved by the inference system will be valid according to the semantics. We do not prove the soundness of the axiomatic system but simply refer to [5]. Hoare logic is however only shown to be relatively complete[], which means that the it is only as complete as the logical system of the assertions. Hence we might have a program which is correct according to the semantics, but that we cannot prove to be correct.

2.3 Verification Condition Generation

In the previous section, we presented Hoare Triples and how they can be used to assign meaning to programs. As mentioned for the postcondition to hold true, we want the program to be executed in a state, satisfying the precondition. It should be fair to assume that we know the postcondition of a program, but how can we know that our precondition is saying anything meaningful about our program. We can use Weakest Precondition Calculus (denoted WP) to generate a suitable precondition in such a case. It essentially states:

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

It is 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 single logical sentence. Weakest Precondition calculus is functional compared to the relational nature of Hoare logic, which means that is easier to reason about, as we dont have to consider any consequences. 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 of *While* is shown.
Most of the rules are somewhat self-explanatory, but we would like to go through a selection of the rules that we find to be complex in our language setting.

Assignments. The rule for computing the weakest liberal precondition for assignments says that for all variables y where $y = e$, we should exchange x in Q with y. So we exchange all occurrences of x with the value assigned to x. We use a quantifier for y to avoid exponential growth by letting $y = e$ and substitute x with y in Q. For instance, if $e = 1 + 2 + ... + 100.000$, then by letting y hold the value of e, we avoid substituting in this very long-expression, making the formula significantly more concise.

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.

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

Figure 3: Rules for computing weakest liberal precondition of a statement in *While*.

While-loops. To compute the *wlp* of a while loop, we have to ensure that the loop invariant holds before, inside, and after the loop. The first condition is simply that the invariant *I* must evaluate to $true$ before the loop. 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. Lastly, the invariant and the negated loop condition must hold when the loop terminates. We must also exchange all the occurrences in the currently accumulated weakest liberal precondition Q for these variables.

Additions to the standard rules. Most of the rules are the standard rules for computing the weakest liberal precondition.[1] We have added two rules: 1) the rule for `violate`, which will always give *false*, and 2) the rule for assertions, which adds the assertion P to the accumulated condition Q, s.t. the new wlp is $P \wedge Q$.

Resolving undefined behaviour. As mentioned in the previous subsection, division and modulo by zero is undefined behavior. It is resolved by adding a condition whenever encountering a division or modulo operator, verifying that the operation is legal. For example, if we have an assignment $x = a\%b$, we will need the following precondition

$$
\begin{aligned}
&\forall y.((b = 0 \Rightarrow false) \\
&\qquad \wedge (b \neq 0 \Rightarrow y = a\%b \Rightarrow Q[x \leftarrow y]))
\end{aligned}
$$

which only resolves the assignment if b is not zero, and otherwise will always fail. Resolving division is similar. This modification is not explicitly written in the rules, but follows from the semantics of our *While* language.

[1]Who provides this standard?

2.3.2 Weakest Precondition

The weakest liberal precondition does not prove termination. However, by the variant in the while-loop introduced in Section 2.2.3, we can extend the *weakest liberal precondition* to *weakest precondition*, which will also ensure termination.
The structure for computing the weakest preconditions for the constructs of the language is much like the one for computing the weakest liberal precondition, except for the structure of while loops, which is presented in Figure 4.

$$WP\left(\begin{array}{c}\texttt{while } e \texttt{ invariant } I \\ \texttt{variant } v, \prec \texttt{ do } s\end{array}, Q\right) = I \wedge$$

$$\begin{array}{l}\forall x_1, ..., x_k, \xi, \\ \quad (((e \wedge I \wedge \xi = v) \Rightarrow WP(s, I \wedge v \prec \xi)) \\ \quad\quad \wedge ((\neg e \wedge I) \Rightarrow Q))[w_i \leftarrow x_i]\end{array}$$

where $w_1, ..., w_k$ is the set of assigned variables in statement s and $x_1, ..., x_k, \xi$ are fresh logical variables.

Figure 4: The rule for computing the weakest precondition for while loops.

The difference between the computation of *wp* and *wlp* for while loops is the presence of the *variant*. When given a variant v for the loop, we assert that v 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 5.

```
1   $\forall q, r.\ (q >= 0 \wedge r >= 0) \Rightarrow$
2     $\forall res_3.\ res_3 = 0 \Rightarrow$
3       $\forall \$a.\ \$a = q \Rightarrow$
4         $(res_3 = (\$a - q) * r \wedge q >= 0)$
5          $\wedge \forall q_2, res_2, \xi_1.$
6            $(q_2 > 0 \wedge res_2 = (\$a - q_2) * r \wedge q_2 >= 0 \wedge \xi_1 = q_2) \Rightarrow$
7              $\forall res_1.\ res_1 = res_2 + r \Rightarrow$
8                $\forall q_1.\ q_1 = q_2 - 1 \Rightarrow$
9                  $(res_1 = (\$a - q_1) * r \wedge q_1 >= 0 \wedge 0 \Leftarrow \xi_1 \wedge q_1 < \xi_1)$
10        $\wedge((q_2 \Leftarrow 0 \wedge res_2 = (\$a - q_2) * r \wedge q_2 >= 0) \Rightarrow res_2 = \$a * r)$
11
```

Figure 5: Weakest precondition of `mult.ifc`

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 universal quantifier over all the variables altered in the loop body, which happens in line 5. Inside the universal quantifier, we check that the loop invariant and condition hold when entering each iteration (lines 6-9) and that the invariant and the negated loop condition hold 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 initial value of v stored in ξ.

2.4 Verifying conditions with automated solvers

SMT-solvers The Weakest precondtions that we can generate, will be a first order logic formula, which we want to validate. For this we use Satisfiability Modulo Theories (SMT). This is a modfication of the boolean satisfiability problem (SAT). Whilst SAT is limited to propositional logic and thus not expressive enough to reason about the verification conditions, SMT can be used to reason about first order logic, through a set of theories. A theory T is a set of sentences. A first order logic formula ϕ is satisfiable modulo this theory if there exists a model M such that $M \vDash_T \phi$. A verification condition for a program should be a valid formula and not just satisfiable, since it must hold for all models. However the problem of validating formula ϕ can be reduced to the problem of satisfying $\neg\phi$, since a formula must be valid if none of its negations are satisfiable.
For this project we need two theories to express the verification condition. Firstly we need the theory of Linear arithmetics, which provided reasoning about *aexpr*. Secondly, we need the theory of quantifiers. The SMT-solver of choice for this project is $Z3$, which uses the *SMT-Lib* standard, allows for both these theories, more specifically the thoery is called LIA.
In practice the way SMT-solvers treat a first order logic formular is by using the theories to reduce the formular to some problem which can then be expressed as a SAT-formula.

Why3. To ensure that our program is working as expected, we want to compare it to $Why3$, a well established tool for program verification. In $Why3$ theories can be built. Most importantly $Why3$ allows us to define functions for which verification conditions can be generated and discharge to a variety of SMT solvers. One of these SMT-solvers are $Z3$, hence if we validate a formular in our language we should also be able to do so for an $Why3$ equivalent program. Hence it provides a well established target for verifying that our program works correctly. We compare our programs to equivalent why3 programs. The semantics will be explained when doing so in

3 Implementation

This section presents our implementation of an application for verification of programs, which we will refer to as *IFC*. The application consists of four parts, and each part carry out a seperate task for the program. This is done to make the code modular, such that each part does not explicitly dependent on any other parts. The four parts of the application performs the following tasks:

1. **Parser:** Transforms an input program, written in *While*, to an Abstract Syntax Tree

2. **Interpreter:** Interprets a program, expressed by an AST, given an initial store

3. **Verification Condition Generator:** Computes the Verification Condition formula of a program, expressed by an AST, using predicate transformer semantics

4. **External SMT-solver API:** Run a verification condition formular through an external prover (Z3), to assert program correctness

Although each part can work seperately, they are connected in the application, as seen in Figure 6.

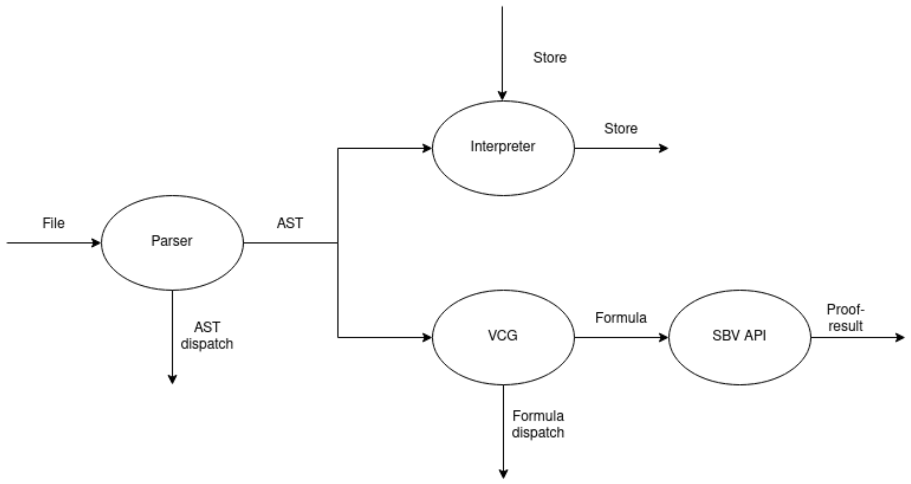


Figure 6: Application layout for IFC

Appendix A explains how to interact with the application. In the Section 3.1 to Section 3.4 we describe key points of the implementation, and Section 3.5 describes an interface which allows for more general proofs about programs.

3.1 Parser

The parser parses an input program, written in *While*, into an abstract syntax tree. The abstract syntax is very similar to the grammar in Table 1 and can be seen in **??**. Most of the implementation follows directly from the grammar, however, in the matter of resolving ambiguity, and introducing of ghost variables, we have made non-trivial design choices. These are presented in the following subsections.

3.1.1 Grammar ambiguity

The parser is built using 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 first order logic in assertions, we found the expression-parser unfit. One reason for this is that we want to allow for syntactic sugar, such as "∀ x y z.", which should desugar to "∀ x. ∀ y. ∀ z". Therefore, instead of using the expression-parser, we manually introduce precedence, according to conventions. That is, negation binds tightest, then conjuction, disjuction, quantifers, and lastly implication. Furthermore the grammar has been left-factorized. The resulting grammar can be seen in Figure 7.
We also introduce some syntactic sugar in the grammar, such as "if c {s};" which will be desugared into "if c {s} else {skip};". This can easily be resolved in the parser by using the `option` parser-combinator.

For other parts of the grammar, which could have been syntactic sugar, such as im-

$$
\begin{array}{lll}
\langle \textit{imp} \rangle & ::= & \langle \textit{quant} \rangle\ \langle \textit{imp'} \rangle \\[1em]
\langle \textit{imp'} \rangle & ::= & \text{'}\Rightarrow\text{'}\ \langle \textit{quant} \rangle\ \langle \textit{imp'} \rangle \mid \epsilon \\[1em]
\langle \textit{quant} \rangle & ::= & \text{'}\forall\text{'}\ \langle \textit{vname} \rangle\ \text{'.'}\ \langle \textit{quant} \rangle \\
 & \mid & \text{'}\exists\text{'}\ \langle \textit{vname} \rangle\ \text{'.'}\ \langle \textit{quant} \rangle \\
 & \mid & \langle \textit{or} \rangle \\[1em]
\langle \textit{or} \rangle & ::= & \langle \textit{and} \rangle\ \text{'}\vee\text{'}\ \langle \textit{or} \rangle \mid \langle \textit{and} \rangle \\[1em]
\langle \textit{and} \rangle & ::= & \langle \textit{neg} \rangle\ \text{'}\wedge\text{'}\ \langle \textit{and} \rangle \mid \langle \textit{neg} \rangle \\[1em]
\langle \textit{neg} \rangle & ::= & \text{'}\sim\text{'}\ \langle \textit{factor} \rangle \mid \langle \textit{factor} \rangle \\[1em]
\langle \textit{factor} \rangle & ::= & \langle \textit{bexp} \rangle \mid \text{'('}\ \langle \textit{imp} \rangle\ \text{')'}
\end{array}
$$

Figure 7: Modified grammar of *While*.

plication 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.

3.1.2 Ghost variables

As previously mentioned in Section 2.2, it is semantically disallowed to use ghost variables anywhere in the program logic, except in assertions. Although this is a semantical matter, we handle this in the parser. If a ghost variable occurs in an invalid context, parsing simply fails, 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)
```

By changing the boolean value in this environment when entering an assertion context, we denote whether we are allowed to parse ghost variables at a certain point in the program. By only setting the boolean to *true* when in an assertion environment, we ensure that parsing of ghost variables will never happen in the program logic.

3.2 Interpreter

The interpreter 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 need to support some sort of output in the future. The store described in Section 2.1 is kept in the State `STEnv`. The store is simply a map from VNames 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. Duly 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 occurs, 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:

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

From this we 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. Figure 8 presents the code for evaluating statements.

```
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 8: Evaluator for statements.[2]

The most complex argument for equivalence between the code and the semantics is the “while” construct. Figure 8, 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* from **??**, 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 evaluates to *true*, we 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 to support arbitrary precision integers. This entails that we currently have 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. A downside to this approach is that the interpreter would then also require external dependencies, and not be a standalone program anymore. 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 *While*, which could help alleviate this problem. Non-quantified assertions are still evauluated as per the operational semantics.

3.3 Verification Condition Generator

The verification condition generator uses weakest precondition calculus, or weakest liberal precondition if a while loop has no specified variant, to construct the verification condition of a program. Like the approach in the interpreter we want to chain the actions and include a state and reader environment in the construction of the verification condition. This is done by using the following code:

```
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 precondition calculus.

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. In our initial approach for implementing the VC generator, we tried to minimize the number of times we had to resolve condition Q, but our approach to this did not work. The working solution is a more naive approach. The next subsections presents both the initial attempt and the current approach.

3.3.1 Failed attempt

As mentioned, the initial approach tried to minimize the number of times we traversed condition Q. The initial approach tried to resolve Q only once, after the entire formula was built. This would allow resolving a sentence in just 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:

```
TODO: fix det her :)))
```

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.
To update the list to substitute x with the appropriate bound variable, we would update the list each time a quantifier is encountered. The first time a quantifier is encountered we do nothing, as this simply means the first bound variable in the list is the one that should currently be used. Next time we encounter a quantification of the according free variable, we "pop" the first element, thus moving on to the next bound variable, and all occurrences of x from that point on will be substituted with the new bound variable, until encountering yet another quantification.
The problem with this approach is how the AST for First Order Logic formula is constructed. Consider the following example:

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

The first traversal would give the paritally resolved formula

$$\forall r\ .\ r = 5 \Rightarrow \forall r\ .\ r = r + 10 \Rightarrow true$$

and a tuple $(r, [r_1, r_2])$ for holding the list of bound variables for r. Now the result of the second traversal would then look like

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

However, we see that the last r in the formula should actually be substituted with r_1, to make the formula correct according to the rules, but as r_1 has already been popped off the stack we substitute with $r_{"}$ instead, which is wrong. On the other hand, if we dont "pop" as soon as the second quantification is encountered we have no information on when to do so, as it is not necessarily the case that there appears an r in the equality before the implication. When this problem was dicovered we turned to the naive solution presented in the next subsection.

3.3.2 Successful attempt

The second and current approach is to resolve Q whenever we encounter 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')
```

First 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 stored in x'. The `Aexpr` which x evaluates to should not be resolved yet, as it could depend on variables not yet encounted. Ghost variables are easy to resolve, as they are immutable, and thus can be resolved right away.

The current version does not require the formula to be closed, but this is necessary for generating symbolic variables. To resolve this, we could simply do another iteration over the formula, checking that all program variables contain a #. However, 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 loops with invariants and variants

The `while` statement has the most complicated rule in the weakest precondition calculus. The code for computing wp for a while loop is presented in Figure 9.

```
 1 wp (While b inv var s) q = do
 2   st <- get
 3   (fa, var’, veq) <- maybe (return (id,
 4                             Cond $ BoolConst True,
 5                             Cond $ BoolConst True)) resolveVar var
 6   w <- wp s (inv ./\. var’)
 7   fas <- findVars s []
 8   let inner = fa (((Cond b ./\. inv ./\. veq) .=>. w)
 9                           ./\. ((anegate (Cond b) ./\. inv) .=>. q))
10   env <- ask
11   let env’ = foldr (\\(x,y) a -> M.insert x y a) env fas
12   inner’ <- local (const env’) $ resolveQ1 inner
13   let fas’ = foldr (Forall . snd) inner’ fas
14   return $ inv ./\. fas’
15   where
16     resolveVar :: Variant -> WP (FOL -> FOL, FOL, FOL)
17     resolveVar var = do
18       x <- genVar "variant"
19       return (Forall x,
20               Cond (Negate (RBinary Greater (IntConst 0) (Var x)))
21              ./\. Cond (RBinary Less var (Var x))
22             , Cond (RBinary Eq (Var x) var)
23             )
```

Figure 9: Weakest precondition for `while`

We attempt to make the code generic in terms existence of the variant, to eliminate code duplication. Lines 6-9 generate the conditions needed for the variant. If no variant is defined we use that predicate logic and conjuction forms a monoid with $\top$ as identity element. This way 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 to 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 generates a variable for each. Collectively this code will generate the weakest precondition for a while statement as per described in 4.

3.4 Proof-assistant API

The proof-assistant API uses the SMT Based Verification library (SBV), which simplifies symbolic programming in Haskell. The library is quite generic and extensive compared to what we need. We mostly make use of the higher level functions, not utilising any internal functions.
Because the default type of SBV does not quite fit our needs, we instead use the provided transformer `SymbolicT`, to embed the `Except` monad. We want to do so as, when iterating over the formular, we might encounter a variable not yet defined and here fail gracefully, instead of throwing an error. Generating a `Predicate ~ Sym SBool` is relatively simple, since the formula generated in the previous stage is already a first order logic formula, making it straight forward to convert it into SBV’s types. For the

entire highlevel logic we resolve it as as presented in Figure 10.

```
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
```

Figure 10: Code for converting a first order logic formular into a symbolic bool.

It is equally straight forward 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 because `m` must be an `ExtractIO`

$$, which Reader and State does not implement [].$$

The predicate constructed by traversing the formula from the last stage is then be used as argument for the SBV function `prove`, which try to prove the predicate using Z3. If the program can be proved by the external SMT solver, the output will be `Q.E.D.`. If the formula is falsifiable, a caounter example is presented. For instance the output of the following program will obviously always be falsified.

```
violate;
```

whereas the multiplication program in Figure 2 is provable.

3.5 Interface for proofs

As described briefly in Section 2.1 we require that programs have a program header. This can also be seen from the example program presented earlier in this report. The header must look as follows:

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

where
In the current implementation, there are no procedures in *While*, which makes it difficult to reason about the input of variables. Therefore we need the header to be able to generate and prove the weakest precondition for a program. The inspiration comes from how the *whyML* language defines procedures. Figure 11 is a whyML program equivalent to the `mult.ifc` program.

```
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 11: Why3 program equivalent to mult.ifc

It is possible for why3 to generate a vector of input variables and then a precondition for each the conditions in `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 postcondition. Notice that Figure 11 does not state any *ensures*, this more closely resembles our language. And since we dont have any return values, we dont really need *ensures*, since this might as well be an assertion in the actual program.

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 support 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 a collective evaluation of our work.

4.1 Experiments

As metioned, 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 *Interpreter* 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.
- We have tested that a correct dynamic evaluation also results in a provable verification condition.
- To support the automated tests, we have done some analysis of some programs and compared them to equivalent *Why3* programs.

A substantial part of tests suites and the experimentation is based on example programs, which can be found in the `examples` folder. Of these some works correctly and some intentionally dont. Table 4, presents the program (leftmost column), a short description of what they do (middle column), and the results of running them through the *Interpreter* and *VC generator* (rightmost column). We will analyse the table more thoroughly in Section 4.1.3.
In Sections 4.1.1 to 4.1.3 we present testing strategy more thoughly. 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.

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 hope to cover all aspects of the grammar.
Furthermore, we use blackbox tests to assert that the *Parser* handles ambiguities, 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.[3]

4.1.2 Quickchecking instances

Other than the blackbox tests for the parser, our tests are property based. With these automatic tests we attempt to verify that

[3]The test-setup is heavily inspired by OnlineTA test by Andrzej Filinski

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 $1, 4, 2, 1, \ldots$.	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

- That the *Parser* correctly parses larger program constructs.

```
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 12: Generation of while-loops using a skeleton.

- That the *Interpreter* correctly follows the semantics of the *While* language.
- That the there is consistency between the *VC-generator* and the *Interpreter*.

We have build generators for the AST, to generate arbitrary input for the tests, and then test properties specified above.

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 *Interpreter* 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 12. 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 want the following property to hold: *parse (prettyprint a) = a*, where a is an arbitrary AST. This is 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 Interpreter 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 relations are directly related to the small-step semantics. Note that we have not presented these. 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.** To ensure consistency between the static and dynamic evaluation we wanted to generate some arbitrary programs and check for consistency. 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.

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 experimented with the 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 *Interpreter* 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 some 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 more interesting examples uses while-loops, in which the invariant and variant is important. 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 showcases 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 under total correctness. 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`.[4] 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

[4]check dette eksempel igennem igen.

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`. This program calculates the collatz sequence of the input variable n. The while-loop will run until $n = 1$. The idea behind this program is to have a while-loop which we cannot provide a variant. Alas this program terminates for all $n > 0$. We can see this by the following:

  Interestingly

```
while (n /= 1) ?{ n > 0}{
      if (n % 2 = 0) {
         n := n / 2;
      } else {
         n := 3 * n + 1;
      };
};
if n = 1 { k := 42; };
#{ k = 42};
```

```
while (n /= 1) ?{ n > 0}{
      if (n % 2 = 0) {
         n := n / 2;
      } else {
         n := 3 * n + 1;
      };
};
#{false};
```

  enough, the program on the left can be proved, whilst the program on the right cannot. This tells us that the loop must terminate, even without a variant. As the program on the right would be partially correct given that the loop does not terminate, since any postcondition Q is valid if the command does not terminate.[5]

Experiments with example programs. We have automated tests for testing that the *Interpreter* can correctly evaluate a program, and by property, we test that the programs that are provable using the VC generator will also evaluate to *true* in the *Interpreter*.
The first type of test, testing the evaluation of programs, is done 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.
The second type of test asserts that provably correct programs will evaluate correctly as well. This is tested by first feeding the programs to the VC generator coupled with Z3, and then to the *Interpreter* with random input. Given that the program is provable the *Interpreter* evaluates all assertions to *true*. Note that if we only show partially correctness, the *Interpreter* might run forever, so we only test for terminating instances. If the program is falsified, then the test will run the program with the falsifiable instance and assert that the *Interpreter* will terminate abnormally.

Provable by VC generation ensures successful evaluation. As described, we value consistency highly. And it should always hold true that if we can prove total correctness of a program the dynamically evaluation should give the expected result. Once

[5]or have i misunderstood something

```
1  $\forall q, r.\ (q \geq 0 \land r \geq 0) \Rightarrow$
2    $\forall res_3.\ res_3 = 0 \Rightarrow$
3      $\forall \$a.\ \$a = q \Rightarrow$
4        $(res_3 = (\$a - q) * r \land q \geq 0)$
5        $\land \forall q_2, res_2, \xi_1.$
6          $(q_2 > 0 \land res_2 = (\$a - q_2) * r \land \xi_1 = q_2) \Rightarrow$
7            $\forall res_1.\ res_1 = res_2 + r \Rightarrow$
8              $\forall q_1.\ q_1 = q_2 - 1 \Rightarrow$
9                $(res_1 = (\$a - q_1) * r \land 0 \leq \xi_1 \land q_1 < \xi_1)$
10       $\land ((q_2 \leq 0 \land res_2 = (\$a - q_2) * r) \Rightarrow res_2 = \$a * r)$
```

Figure 13: Generated verification condition for the modified `mult` program.

again it is important to note this will only hold for total correctness and not necessarily for partial correctness. Oppositely, correct dynamic evaluation does not imply provable correctness. 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*. This might be even more apparent, because of quantifiers not working correctly in the *interpreter*. 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 13.
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 falsifiable 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 LHS of the implication will evaluate to *true* as $(-3 \leq 0 \land 6 = 3 * 2)$ but the RHS will evaluate to *false*, since $6 \neq 0 * 2$
But we know that the program does in fact behave as intended, so is it correct that the prover falsifies the formula? We can verify that our program acts correctly, by doing the same modification to the whyML program in Figure 11. 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 of the project. We find that the two most important aspects of our application are correctness and maintainability. A non-correct implementation is useless and high maintainability allows for easy extension/modification to build a more robust and complete solution. Overall we find that we satisfy these criteria to an acceptable degree.

4.2.1 Correctness

Overall we find that our implementation coincide with the specifications described and the goals we wanted to achieve with this project. Although the implementation deviate from the formal semantics in a few places, such as for abnormal termination. We do so purely for convenience. And we find that based on our tests, experiments and so forth, presented in the previous section, that a well working solution. Especially that, we can distinguish programs under partial correctness with programs under total correctness. And that the evaluator and static verification gives equivalent results, is a good indication that we have been successful in our implementation. With this said, we are not fully satisfied with the test-suite, and with more time the property based tests would have been stronger (and never time out), and we would have more unit-tests for the basic concepts of the different modules. Furthermore, we need to address the quantifiers in the evaluator. It is inferior that these are not handled appropriately. As stated, this stems from a combination of two events. Firstly, that we went with unbounded integers for the standard arithmetic type, which makes it impossible to go through all possible values, and that we have treated the internal workings of SMT-solvers mostly as a blackbox and thus have not investigated deeply possible techniques for handling qunatifiers. Although it would be very nice to use one of the strategies from an SMT-solver, this is quite a substantial topic, not focused on in this project.

4.2.2 Completeness

We find that we have accomplished our goals for this project except for the parts mentioned in the correctness section. Furthermore it would have been good both for the implementation as well as our general understanding of provability of correctness to have delved more into the internals of SMT-solvers. As of now, our understanding is still basic.

4.2.3 Robustness

Overall we find that our testing has been valuable in insuring robustness of the code. As mentioned we have found perculiarities, in the abnormalilites, which has now been removed. By this we assures that we can never prove anything which would result in abnormal termination in the evaluator. In this philospohy we gracefully handle run-time errors and such gracefully. On the other hand we use *error* when we find that something is an implementation error. Thus we make a clear destinction to the user, if they have made a mistake or there is something wrong with our implementation.

4.2.4 Maintainability

As mentioned this project should serve as a base. Hence 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 as 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 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. As a supporting argument, the Modulo-incident, we were able to fix rather quickly, when we found an approach and it required fairly little additional code.

5 Discussion and conclusion

We have in this report presented the background for our project, which includes both the formal specification of the language, the axiomatic systems for Floyd-Hoare logic and the predicate transformer semantics which we use to generate verification conditions for a program. We have presented our implementation and essential design choices of the program. Lastly we have argued why we find that our specific implementation meet all the goals presented in the introduction.

Nævn disse?

We have argued that or program is correctly able to statically prove correctness of certain programs and dynamically evaluate same program with a correct result. We have done so informally by tests and by a set of example programs, which shows examples of both partial correctness along with total correctness of certain programs. We have further looked at how existing solutions such as *Why3* does verification conditions, by comparing *While*-programs, to *WhyML*. In the current state the language is still sparse and quite restrictive. For the language to be used for more than just toying around, we propose three extensions which we find essential to the language in Section 5.1.

5.1 Future work

We describe three extensions that would make the programming language more useful. The first extension will be additional constructs for the language which would give more expressive power to programs. The next would allow for more usability in terms of different program types. Lastly we describe the extension which we find the most interesting, since the main idea of this project was to allow for reason about Information Flow of programs.

5.1.1 Procedures and arrays

We propose that the language is extended to included procedures and arrays. This would make it possible to make more interesting examples programs, firstly it would allow for better assertions. Imagine a procedure which has the result of the maximum value of two integers. Then we could use this in assertions about variables in another procedure. Most definitely this would provide a better experience using the language. In a similar manner we can make a lot more interesting programs if we have arrays (or atleast some sort of generic-collection), which we can operate on.

5.1.2 Type system

A typesystem would equally be a good extension for better usability. It would allow us for having different types store in variables (and if coupled with the other extension, arrays), which could reduce a lot of repeated code. Furthermore additional types could eliminate a lot of "boilerplate" assertions. An instance of this, would be a type *Nat* over the natural numbers, which would eliminate the need for checks such as $x \geq 0$. Furthermore such types would allow us to fix our faulty implementation of quantifiers in the evaluator. In such a case, we could allow only certain types to be quantified. And although bruteforcing all values of a bound integer would still be extremely poor performace wise it would be a solution to our problem. Although bruteforcing should probably not be used.

5.1.3 PER-logic for Information Flow Control

As mentioned time and time again, this project initially arose to make an implementation of the *Partial Equivalence Relation* logic for information flow control[]. The idea behind this is to prove properties of information flow, similarly to how we have been able to prove properties about simple *While*-constructs. From our assessment we find that the project is ready, or atleast close to ready to include this program logic. Furthermore, its remains unbeknownst to use whether we can generate verification conditions for the logic.

5.2 Conclusion

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] Robert W. Floyd. “Assigning Meanings to Programs”. In: *Proceedings of Symposium on Applied Mathematics* 19 (1967), pp. 19–32. URL: http://laser.cs.umass.edu/courses/cs521-621.Spr06/papers/Floyd.pdf.

[3] 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.

[4] Claude Marché. *Lecture notes in MPRI Course 2-36-1: Proof of Program*. 2013. URL: https://www.lri.fr/~marche/MPRI-2-36-1/2013/poly1.pdf.

[5] Hanne R. Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer*. Undergraduate Topics in Computer Science. Springer, March 2007. ISBN: 1846286913. DOI: http://dx.doi.org/10.1007/978-1-84628-692-6. URL: http://dx.doi.org/10.1007/978-1-84628-692-6.

A How To Use