UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Project outside course scope

Jacob Herbst (mwr148), Matilde Broløs (jtw868)

Verified Weakest Precondition

Comparing deductive verification methods of a Weakest precodintion Calculus

Github repository: `https://github.com/Spatenheinz/VCGen_in_Why3`

Advisor: Ken Friis Larsen

4th of November 2022

Abstract

Contents

1 Introduction

- Why do we want verification conditions and very brief background?
- Why would we want to verify VC?
- What tools have we used and what have we looked at?
- How is the report structured?

2 Background

When writing a verified VC generator, we must consider both the language that we want to interpret, the syntax and semantics, both operational and axiomatic, of the object language, and the language that we use for writing the VC generator in.
Firstly, we consider an object language WHILE which will be the input for the compiler. In 2.1 we describe the syntax and semantics of the object language.
Secondly, we consider the axiomatic semantics of the language, which we use for generating verification conditions for a given program. This is presented in 2.2, where we describe the Hoare logic and weakest precondition calculus that we build VC generation on.
Thirdly, we consider the host language used for writing the compiler. We use Why3 as our host language, with some additional proofs conducted in Isabelle. A brief introduction to both is given in 2.3.2 and 2.3.3 respectively.

2.1 WHILE

In this project we work with a simple WHILE language. We design it to be a minimal language in which we can still express meaningful programs. In 2.1.1 we define the syntax of the language, and in 2.1.2 the operational semantics.

2.1.1 Syntax

As object language we consider a simple imperative language often in the literature referred to as either IMP or WHILE and from here on out we will refer to it as WHILE. Notice the language might differ slightly from other definitions but any such deviations should be insignificant.
We consider a language with two basic types, integers and booleans. Figure 1 shows the syntax for arithmetic and boolean expressions. An arithmetic expression can be either a variable, an integer, or a binary operation on two arithmetic expressions in which the operation can be addition, subtraction or multiplication. Notice here than we only define a single rule for binary operation in arithmetic expressions, instead of defining both subtraction and addition. We then have an arithmetic operator type that can be either of the valid operators. Using this syntax it is easy for us to extend the language in case we wanted to include more operators.
Boolean expressions can be either a constant bool, a negation, a conjunction, or a binary relation. As of right now we only support the relational operation $\leq$, as this in combination with $\neg$ allows us to express all other relational operators. Likewise with the functionally complete set $\{\wedge, \neg\}$ we can define all possible boolean expressions.

$$
\begin{array}{lll}
\langle aexpr \rangle & ::= & \langle identifier \rangle \\
& | & \langle integer \rangle \\
& | & \langle aexpr \rangle \ \langle aop \rangle \ \langle aexpr \rangle \\
\\
\langle aop \rangle & ::= & \text{'}-\text{'} \mid \text{'}*\text{'} \mid \text{'}+\text{'} \\
\\
\langle bexpr \rangle & ::= & \langle bool \rangle \\
& | & \text{'}\neg\text{'} \langle bexpr \rangle \\
& | & \langle bexpr \rangle \ \text{'}\wedge\text{'} \ \langle bexpr \rangle \\
& | & \langle aexpr \rangle \ \langle rop \rangle \ \langle aexpr \rangle \\
\\
\langle rop \rangle & ::= & \text{'}\leq\text{'}
\end{array}
$$

Figure 1: Grammar for arithmetic & boolean expressions

Next we want to define assertions in our language, using formulas corresponding to a first order logic. These formulas are used for two things: 1) the Hoare logic and Verification condition generation, and 2) in assertions that the programmer can use throughout the program to strengthen the precondition. These assertions are non-blocking, meaning that they are not evaluated at runtime, but only used for computing the final verification condition. We have a functionally complete set in $\{\forall, \wedge, \neg\}$.
Note here that in other definitions of assertion languages in relation to hoare logic it is common to include ghost variables, which are not allowed in the execution of programs and hence are only allowed in the logic language. In our current state, we do not include such variables. The syntax for formulas is depicted in Figure 2.

$$
\begin{array}{lll}
\langle formula \rangle & ::= & \langle bexpr \rangle \\
& | & \langle formula \rangle \ \text{'}\wedge\text{'} \ \langle formula \rangle \\
& | & \text{'}\neg\text{'} \ \langle formula \rangle \\
& | & \langle formula \rangle \ \text{'}\Rightarrow\text{'} \ \langle formula \rangle \\
& | & \text{'}\forall\text{'} \ \langle identifier \rangle \ \text{'}.\text{'} \ \langle formula \rangle
\end{array}
$$

Figure 2: Syntax of formulas

Lastly we define the syntax of statements. A WHILE program is essentially a statement, where a statement can be either a variable binding, an if-else statement, a while loop, an assertion, a skip command, or a sequence of two statements. Thus an "empty" program consists of only skip commands. Figure 3 shows the syntax of statements.

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

Figure 3: Grammar for a WHILE program

In future work it could be interesting to extend the language with additional expressions in each category, but for now we keep the structure of the object language simple.

2.1.2 Semantics

We describe WHILE by its natural semantics. We use a store to keep the values of any program variables, and this is implemented using a finite map:

$$\sigma \in \Sigma = Var \longrightarrow \mathbb{Z} \cup \{\bot\}$$

where Var denotes all possible variables and $\bot$ denotes the abnormal result in case of unbound variables. Hence we consider a store to be a total function.

Arithmetic expressions. We define the judgement for arithmetic expressions as

$$\langle a, \sigma\rangle \Downarrow_a (n|\bot)$$

where $n \in \mathbb{Z}$, a is an arithmetic expression, and $x|y$ denotes either x or y. All the inference rules for arithmetic expressions are shown in Figure 4. The rules intuitively describes evaluation of expressions. Not that when encountering abnormal behaviour $\bot$ the behaviour is propagated.

$$\text{EACst}\ \frac{}{\langle n, \sigma\rangle \Downarrow_a n}$$

$$\text{EAVar}\ \frac{}{\langle v, \sigma\rangle \Downarrow_a n}\ (\sigma(v) = n) \qquad \text{EAVar\_Err}\ \frac{}{\langle v, \sigma\rangle \Downarrow_a \bot}\ (\sigma(v) = \bot)$$

$$\text{EABin}\ \frac{\langle a_0, \sigma\rangle \Downarrow_a n_0 \qquad \langle a_1, \sigma\rangle \Downarrow_a n_1}{\langle a_0 \oplus a_1, \sigma\rangle \Downarrow_a n_0 \oplus n_1}$$

$$\text{EABin\_Err1}\ \frac{\langle a_0, \sigma\rangle \Downarrow_a \bot}{\langle a_0 \oplus a_1, \sigma\rangle \Downarrow_a \bot} \qquad \text{EABin\_Err2}\ \frac{\langle a_0, \sigma\rangle \Downarrow_a n_0 \qquad \langle a_1, \sigma\rangle \Downarrow_a \bot}{\langle a_0 \oplus a_1, \sigma\rangle \Downarrow_a \bot}$$

Figure 4: Semantics of arithmetic expressions in WHILE.

Boolean expressions. In a similar manner we define a judgement for boolean expressions as

$$\langle b, \sigma\rangle \Downarrow_b (t|\bot)$$

where $t \in \{true, false\}$ and b is a boolean expression. The semantics for boolean expressions are presented in Figure 5.

$$\text{EBCst}\ \frac{}{\langle t, \sigma\rangle \Downarrow_b t}$$

$$\text{EBLeq}\ \frac{\langle a_0, \sigma\rangle \Downarrow_b n_0 \qquad \langle a_1, \sigma\rangle \Downarrow_b n_1}{\langle a_0 \leq a_1, \sigma\rangle \Downarrow_b (n_0 \leq n_1)}$$

$$\text{EBLeq\_Err1}\ \frac{\langle a_0, \sigma\rangle \Downarrow_b \bot}{\langle a_0 \leq a_1, \sigma\rangle \Downarrow_b \bot} \qquad \text{EBLeq\_Err2}\ \frac{\langle a_0, \sigma\rangle \Downarrow_b n_0 \qquad \langle a_1, \sigma\rangle \Downarrow_b \bot}{\langle a_0 \leq a_1, \sigma\rangle \Downarrow_b \bot}$$

$$\text{EBAnd}\ \frac{\langle b_0, \sigma\rangle \Downarrow_b b'_0 \qquad \langle b_1, \sigma\rangle \Downarrow_b b'_1}{\langle b_0 \wedge b_1, \sigma\rangle \Downarrow_b b'_0 \wedge b'_1}$$

$$\text{EBAnd\_Err1}\ \frac{\langle b_0, \sigma\rangle \Downarrow_b \bot}{\langle b_0 \wedge b_1, \sigma\rangle \Downarrow_b \bot} \qquad \text{EBAnd\_Err2}\ \frac{\langle b_0, \sigma\rangle \Downarrow_b b'_0 \qquad \langle b_1, \sigma\rangle \Downarrow_b \bot}{\langle b_0 \wedge b_1, \sigma\rangle \Downarrow_b \bot}$$

$$\text{EBNot}\ \frac{\langle b, \sigma\rangle \Downarrow_b b'}{\langle \neg b, \sigma\rangle \Downarrow_b \neg b'} \qquad \text{EBNot\_Err}\ \frac{\langle b, \sigma\rangle \Downarrow_b \bot}{\langle \neg b, \sigma\rangle \Downarrow_b \bot}$$

Figure 5: Semantics of boolean expressions in WHILE.

Statements. Likewise for statements a judgement will either end in a new state, or an abnormal behaviour, caused by unbound variables. Thus the judgement is defined by

$$\langle s, \sigma\rangle \Downarrow_s (\sigma | \bot)$$

where s is a statement. The semantics of statements are presented in Figure 6.
Note that we have an $invariant$ in the syntax for while statements. This is only used when proving correctness of a while loop, and not in the actual evaluation of a while statement.

2.1.3 Assertion language

Now we examine the semantics of assertions, in the form of formulas. As formulas can contain quantifiers, we cannot define the semantics of formulas by a set of inference rules in terms of operational semantics. Instead we define it as a set of satisfaction relations $\sigma \vDash f$, which can be seen in Figure 7. Note that in the figure $\top$ and $\bot$ denotes that a formula holds or does not hold respectively, thus it is not used as an indication of normal versus abnormal behaviour in this case. It is evident from the rules that true will hold for all $\sigma : \Sigma$, whereas false will never hold.
Another interesting notion about evaluation of formulas is the case of unbound variables. One approach would be to treat abnormal behaviour, such as unbound variables, as false, as a formula like $a \leq 0$ would not be valid if a was unbound. However, in the case of a formula like $\neg(a \leq 0)$ the inner expression would evaluate to false, following the logic we just presented, and then the entire formula would evaluate to true. This would indeed be a mistake, thus we need to ensure that the formula is closed before starting to evaluate it. Therefore evaluation of a formula assumes that the formula is closed. We will see a solution for implementing this in Why3 in 3.5.

2.2 Hoare logic and WP calculus

With the assertion language it is possible to reason about program execution on a logical level. In the following sections we introduce an axiomatic way for proving correct-

$$\text{ESSkip}\ \frac{}{\langle \mathbf{skip}, \sigma\rangle \Downarrow_s \sigma}$$

$$\text{ESAss}\ \frac{\langle a, \sigma\rangle \Downarrow_s n}{\langle x{:=}a, \sigma\rangle \Downarrow_s \sigma[x \mapsto n]} \qquad \text{ESAss\_Err}\ \frac{\langle a, \sigma\rangle \Downarrow_s \bot}{\langle x{:=}a, \sigma\rangle \Downarrow_s \bot}$$

$$\text{ESSeq}\ \frac{\langle s_0, \sigma\rangle \Downarrow_s \sigma'' \quad \langle s_1, \sigma''\rangle \Downarrow_s \sigma' \quad \text{where } \sigma'' \neq \bot}{\langle s_0\ ;\ s_1, \sigma\rangle \Downarrow_s \sigma'}$$

$$\text{ESSeq\_Err}\ \frac{\langle s_0, \sigma\rangle \Downarrow_s \bot}{\langle s_0\ ;\ s_1, \sigma\rangle \Downarrow_s \bot} \qquad \text{ESIfT}\ \frac{\langle b, \sigma\rangle \Downarrow_s true \quad \langle s_0, \sigma\rangle \Downarrow_s \sigma'}{\langle \mathbf{if}\ b\ \{\ s_0\ \}\mathbf{else}\ \{\ s_1\ \}, \sigma\rangle \Downarrow_s \sigma'}$$

$$\text{ESIfF}\ \frac{\langle b, \sigma\rangle \Downarrow_s false \quad \langle s_1, \sigma\rangle \Downarrow_s \sigma'}{\langle \mathbf{if}\ b\ \{\ s_0\ \}\mathbf{else}\ \{\ s_1\ \}, \sigma\rangle \Downarrow_s \sigma'} \qquad \text{ESIf\_Err}\ \frac{\langle b, \sigma\rangle \Downarrow_s \bot}{\langle \mathbf{if}\ b\ \{\ s_0\ \}\ \mathbf{else}\ \{\ s_1\ \}, \sigma\rangle \Downarrow_s \bot}$$

$$\text{ESWhileT}\ \frac{\langle b, \sigma\rangle \Downarrow_s true \quad \langle s, \sigma\rangle \Downarrow_s \sigma'' \quad \langle \mathbf{while}\ b\ \mathbf{invariant}\ f\ \{\ s\ \}, \sigma''\rangle \Downarrow_s \sigma'}{\langle \mathbf{while}\ b\ \mathbf{invariant}\ f\ \{\ s\ \}, \sigma\rangle \Downarrow_s \sigma'}$$

$$\text{ESWhileF}\ \frac{\langle b, \sigma\rangle \Downarrow_s false}{\langle \mathbf{while}\ b\ \mathbf{invariant}\ f\ \{\ s\ \}, \sigma\rangle \Downarrow_s \sigma}$$

$$\text{ESWhile\_ErrB}\ \frac{\langle b, \sigma\rangle \Downarrow_s \bot}{\langle \mathbf{while}\ b\ \mathbf{invariant}\ f\ \{\ s\ \}, \sigma\rangle \Downarrow_s \bot}$$

$$\text{ESWhile\_ErrS}\ \frac{\langle b, \sigma\rangle \Downarrow_s true \quad \langle s, \sigma\rangle \Downarrow_s \bot}{\langle \mathbf{while}\ b\ \mathbf{invariant}\ f\ \{\ s\ \}, \sigma\rangle \Downarrow_s \bot}$$

Figure 6: Semantics of statements in WHILE.

$$\begin{aligned}
\sigma \vDash \mathbf{true} &\iff \top \\
\sigma \vDash \mathbf{false} &\iff \bot \\
\sigma \vDash f_0 \wedge f_1 &\iff (\sigma \vDash f_0) \wedge (\sigma \vDash f_1) \\
\sigma \vDash \neg f &\iff \neg(\sigma \vDash f) \\
\sigma \vDash f_0 \Rightarrow f_1 &\iff (\sigma \vDash f_0) \Rightarrow (\sigma \vDash f_1) \\
\sigma \vDash \forall x.f &\iff \forall n : \mathbb{Z}.(\sigma[x \mapsto n] \vDash f)
\end{aligned}$$

Figure 7: Satisfaction relation for evaluation of formulas.

ness of programs, and a way to automatically generate formulas w.r.t. said system.

2.2.1 Hoare logic

Floyd–Hoare logic (from now simply refered to as Hoare logic) is a formal system proposed by Tony Hoare in 1969, based on earlier work from Robert W, which is used to prove correctness of a program. The core of this formal system is a Hoare triple denoted as $\{P\}s\{Q\}$ where P and Q are assertions. This notion means that if the precondition P holds in the initial state, and s terminates from that state, then Q holds in the halting state of s. More formally we write:

$$\{P\}s\{Q\} = \forall \sigma, \sigma' \in \Sigma.(\sigma \vDash P) \wedge \langle s, \sigma \rangle \Downarrow \sigma' \Rightarrow (\sigma' \vDash Q)$$

Thus if a hoare triple holds, it is correct w.r.t. to the assertions.
Hoare logic provides axioms and inference rules for an imperative language. The statements included in these rules corresponds to statements provided in 2.1.1-2.1.2. Figure 8 shows the inference rules.

$$\text{HSkip}\ \frac{}{\{P\}\mathbf{skip}\{Q\}} \qquad \text{HAssign}\ \frac{}{\{Q[x \mapsto a]\}x := a\{Q\}}$$

$$\text{HIf}\ \frac{\{P \wedge b\}s_0\{Q\} \quad \{P \wedge \neg b\}s_1\{Q\}}{\{P\}\mathbf{if}\ b\ \{\ s_0\ \}\mathbf{else}\ \{\ s_1\ \}\{Q\}} \qquad \text{HSeq}\ \frac{\{P\}s_0\{R\} \quad \{R\}s_1\{Q\}}{\{P\}s_0\ ;\ s_1\{Q\}}$$

$$\text{HWhile}\frac{\{I \wedge b\}s\{I\}}{\{I\}\mathbf{while}\ b\ \mathbf{invariant}\ I\ \{\ s\ \}\{I \wedge \neg b\}}$$

$$\text{HCons}\frac{\vDash P \rightarrow P' \quad \{P\}s_0\{Q\} \quad \vDash Q' \rightarrow Q}{\{P\}s_0\{Q\}}$$

Figure 8: Hoare logic inference rules.

The rules are rather intuitive. For Hskip, the pre and postcondition must be the same. For Hassign all occurences of x in postcondition Q must be substituted with a. For conditionals HIf we have that if the precondition P and the conditional check b holds then Q must hold after the then branch s_0, and if P and not b holds then Q must hold after the else branch s_1. For while the assertion I is called the invariant and must hold before each iteration of the loop and after the loop. In a similar fashion if the conditional b holds before an iteration of s, then the invariant must still hold afterwards. When the loop ends, the condition must not hold any longer. Sequences are rather intuitive. If P holds in the state σ, R holds in the state σ'' that is the result of executing s_0 in σ, and Q holds in state σ' that is the result of executing s_1 in σ'', then P holds before the sequence $s_0; s_1$, and Q holds after. Lastly the conseqeuce rule HCons is used to strengthen a precondition or weaken a postcondition.

Using these inference rules one can show a program to be partially correct. The reason we can only show partial correctness is because the transitional sequence of statements might be infinite, i.e. a while loop might not terminate, and the triple only considers s executing normally, which we can only show for terminating functions. Thus for total correctness we need to be able to prove termination for while-loops. To do this we need additional information. A variant v must be provided and it must be subject to a well-founded relation $\prec$. We can express the Hoare triple as follows:

$$\text{HWhile}\frac{\{I \wedge b \wedge v = \xi\}s\{I \wedge \xi\}}{\{I\}\mathbf{while}\ b\ \mathbf{invariant}\ I\ \mathbf{variant}\ v, \prec\ \{\ s\ \}\{I \wedge \neg b\}}\ \text{where}\ wf(\prec)$$

where, ξ is a fresh variable. By this total correctness can be achieved.

Whether we consider total or partial correctness solely depends on the existence of a variant. In any case Hoare logic can be shown to be sound, that is if a Hoare triple is proveable from the inference rules then the hoare triple is valid, i.e. $\vdash \{P\}s\{Q\}$ implies $\vDash \{P\}s\{Q\}$. This is an important property, since it ensures that it is impossible

to derive partial correctness proofs that does not hold. The proof goes by induction on the derivation of $\vdash \{P\}s\{Q\}$. We omit this proof, because we do not use Hoare logic in our formalization directly, but rather as a consequence for our Verification condition generation.

With regards to completeness Hoare logic was in 1974 shown to be relatively complete with respect to the assertion language used[1]. That is, Hoare logic is no more incomplete than the assertion language. This means that $\vdash \{P\}s\{Q\}$ implies $\vDash \{P\}s\{Q\}$, given that the assertion language is expressive enough. Notice furthermore that without multiplication Hoare logic would not be complete. Since we are mostly interested in verification, we do not really care about completeness.
The proof Cook provided uses the notion of Weakest liberal precondition (wlp).

2.2.2 Weakest Precondition Calculus

Now that we have looked at a general axiomatic semantic of our WHILE language, we will look at weakest precondition calculus, which is another sort of axiomatic semantic for our language.
WP was first introduced by Edgar W. Dijkstra and is defined as follows

$$\forall P.P \Longrightarrow Q \text{ iff } \{P\}s\{Q\} \text{ is valid}$$

That is P is a weakest precodition for s such that Q will hold in the halting state of s.
We use weakest precondition calculus for generating verification conditions, as we can use this calculus to determine the weakest precondition of a program, ie. the weakest precondition that must hold before a statement for the postcondition to hold afte the execution of the statement. This precondition can then be used as a verification condition for the program, meaning that if the weakest precondition holds, then the correctness of the program can be assured. However, the weakest precondition also assures termination, and that is not always possible. When we are interested in asserting correctness, but not termination, we use *weakest liberal precondition*. We denote the weakest liberal precodition as $P = wlp(s, Q)$.
Now we present the rules for determining the weakest liberal precondition of an expression. Next we will present the intuition behind proving the soundness of this system.

Rules for determining weakest liberal precondition The weakest liberal precondition is the minimal condition that must hold to prove correctness of a program, assuming that it terminates. The *wlp* for our WHILE language (see 2.1.1 for syntax) is determined using the rules presented in Figure 9.
The rules define how to compute the wlp when given a statement s and a formula Q, meaning that it is a function $WLP(s, Q)$ which outputs the wlp.
The wlp of `skip` is just the formula Q, as the statement has no effect. The wpl of `x:=e` with Q is simply Q where each occurence of x is exchanged with e, as that is the effect of the assignment. The wlp of `s_0;s_1` is the result of first determining $WLP(s_1, Q) = Q_1$, and then using that result to compute the overall wlp as $WLP(s_0, Q_1)$. This is because we always compute the wlp by looking at the last statement, building up a formula from right to left.
The next two rules are more interesting. For determining the wlp of an `if` statement, the rule states that *if* e is true, then the wlp is $WLP(s_0, Q)$, else it is $WLP(s_1, Q)$, thus choosing a branch according to the condition. Next we have the rule for determining

$$
\begin{aligned}
WLP(\texttt{skip}, Q) &= Q \\
WLP(x := a, Q) &= \forall y, y = a \Rightarrow Q[x \mapsto y] \\
WLP(s_0; s_1, Q) &= WLP(s_0, WLP(s_1, Q)) \\
WLP(\{P\}, Q) &= P \wedge Q \quad \text{where P is an assertion} \\
WLP(\texttt{if } e \texttt{ then } s_0 \texttt{ else } s_1, Q) &= (e \Rightarrow WLP(s_0, Q)) \\
&\quad \wedge (\neg e \Rightarrow WLP(s_1, Q)) \\
WLP(\texttt{while } b \texttt{ invariant } I \texttt{ do } s, Q) &= I \wedge \\
&\quad \forall x_1, ..., x_k, \\
&\quad (((b \wedge I) \Rightarrow WLP(s, I)) \\
&\quad\quad \wedge ((\neg b \wedge I) \Rightarrow Q))[w_i \mapsto x_i] \\
&\quad \text{where } w_1, ..., w_k \text{ is the set of assigned variables in} \\
&\quad \text{statement } s \text{ and } x_1, ..., x_k \text{ are fresh logical variables.}
\end{aligned}
$$

Figure 9: Rules for computing weakest liberal precondition [2].

the wlp of `while` statements. It states that firstly the invariant I must hold. Secondly we want to exchange all assigned variables $w_0, ..., w_k$ with the fresh logical variables $x_0, ..., x_k$. Next we want assert that if both the invariant I and the condition e holds, then $WLP(s, I)$ must hold, as we execute the loop body, and the invariant must be true in the end of each loop iteration. If e doesn not hold, but I does, then we do not loop again, and the statement has the same effect as a `skip` statement, thus we assert that Q holds in this case.
The last rule concerns assertion statements, and this case is trivial, as we simply add the assertion to the formula Q by conjunction.

Soundness of wlp For the weakest liberal precondition calculus to be meaningful to us, we must assert that the calculus is actually sound. This is done by proving that *for all statements s and formula Q, $\{WLP(s, Q)\}s\{Q\}$ is valid for partial correctness*.
We present here the stategy and intuition behind the proof, using the prcedure presented in [2].
The proof is done by considering the rules of finding the wlp for the different statements, thus it uses structural induction on s. As a preliminary remark it should be noted that for any formula ϕ and any state σ, we have that the interpretation of $\forall x_1 ... x_k . \phi[w_i \mapsto x_i]$ in σ does not depend on the values of variables in σ, thus if it holds for one state σ then it should hold for all states σ' that only differs from σ in the value of the variables.

Now lets look at the structure of the proof. We consider each possible case of s, thus considering all the different types of statements. Most of these are straightforward, but the case for $while$-expressions is somewhat tricky. Therefore we will dive a bit into the intuition behind that case.
Lets assume that $s =$ `while` b `invariant` I s'. Now we want to show that $\{WLP(s, Q)\}s\{Q\}$ holds for any Q. Assume that we have a state σ such that $\sigma \vDash WLP(s, Q)$ holds, and that $\langle s, \sigma \rangle \Downarrow \sigma'$
The rest of the proof uses induction on the length of the execution denoted by $\rightsquigarrow^*$.

Case $b = false$**.** This terminates the loop, thus $\sigma = \sigma'$. In this case we have that $\sigma \vDash b = false$, by the assumption. From the wlp rule for while expressions, we get that $\sigma \vDash I$ and $\sigma \vDash ((b = false \wedge I) \Rightarrow Q)[w_i \mapsto x_i]$. By the preliminary notion, we know that we can instatiate each w_i to the variables of σ, and then we get $\sigma \vDash (b = false \wedge I) \Rightarrow Q$. Now, as $\sigma \vDash I$ and $\sigma \vDash b = false$ we get that $\sigma \vDash Q$, which was exactly what we wanted to show.

Case $b = true$**.** In this case the loop does not terminate, but executes another iteration of the body. That means $\sigma \neq \sigma'$, and that $\langle s', \sigma \rangle \Downarrow \sigma''$ and than $\langle s, \sigma'' \rangle \Downarrow \sigma'$.
We can prove that $\{WLP(s, Q)\}s\{Q\}$ by proving that $\sigma'' \vDash I$ and $\sigma'' \vDash WLP(s, Q)$, from where we can use induction on the length of the execution to say that $\sigma' \vDash Q$.
By the assumption we know that $\sigma \vDash b = true$, and by the wlp rule we have that $\sigma \vDash I$ and $\sigma \vDash (((b = true \wedge I) \Rightarrow WLP(s', I)) \wedge ((b = false \wedge I) \Rightarrow Q))[w_i \mapsto x_i]$. The second part of the conjunction is trivially true, hence we dont consider it in the rest of the proof. Again we instatiate all variables to those in σ, and get $\sigma \vDash (b = true \wedge I) \Rightarrow WLP(s', I))$. Because we have that $\sigma \vDash b = true$ and $\sigma \vDash I$, we get that $\sigma \vDash WLP(s', I)$.

By IH on the structure we have $\{WLP(s', I)\}s'\{I\}$, hence $\sigma'' \vDash I$.
As the only difference between σ'' and σ is the values of the variables w_i then by the preliminary remark we get that $\sigma'' \vDash ((b = true \wedge I) \Rightarrow WLP(s', I))[w_i \mapsto x_i]$, which means that we now have all we need to know that $\sigma'' \vDash WLP(s, Q)$. We now use the induction on the length of the execution to get $\sigma' \vDash Q$, which is exactly what we wanted to show.

As a consequence of soundness $\vDash P \Rightarrow WLP(s, Q)$ suffices to show partial correctness. Hence For a VC generator for partial correctness, only an implementation of WLP is needed.

2.3 Tools

In this project we have used two different tool for implementing a verified verification condition generator. In the following section we introduce a brief introduction to both tools and some of their capabilities.

2.3.1 SMT solvers and ATP

In the previous section we described how one can, build a weakest precondition from a statement s and a post-condition Q. From a formula, we want the correctness by checking the validity of said formula. To tell if a formula f is valid, we can reformulate the task to asking if $\neg f$ is satisfiable. That is, if there exists an assignment of variables which makes $\neg f$ satisfiable, then f cannot be true for all states. Thus we can reduce the question of validity to satisfiability and use SAT solvers to automatically prove the validity of a formula. However SAT only works for boolean formulas, and boolean formulas are not expressive enough to reason about the formulas we will generate. Thus we must use more powerful tools. Two well known way to automatically prove a formula is Satisifiability Modulo Theories (SMT) solvers and Automated Theorem Provers (ATPs).

SMT solvers are as the name suggests tackles the question of satisfiability modulo a set of theories. This means that the language of SMT solvers are first order logic, and depending on which theories the SMT solver implements its reasoning follows. For instance, many SMT solvers include the theories of integers, equality, function symbols etc. We can state the validity of a formula f as $M \vDash_T f$ meaning the validity of f from a Model M with respect to the theory T. We will not go into how exactly such formulas are solved.

ATPs deals with a formula by considering it as a conjecture and checking if the conjecture follows from a set of sentences (Axioms and hypotheseses). some ATPs can also reason about higher order logic, but in our case this is uneccessary.

We dont really consider the difference between ATPs and SMT solvers, but only try to exploit that they can reason about first order logic. Not only can the automated tools be used to reason the formulas, we will generate with the WP algorithm, we can also (partially) use it to reason about our implementation. Also we dont interact with any of these directly but use them as external tools from within Why3.

2.3.2 Why3

Why3 is a platform for deductive program verification initially released in 2012. Deductive program verification means that it can be used to prove properties of programs using a set of logical statements. The platform is vast and there are many ways to reason about programs. The tool strives to be a standardized frontend with a high level of automation by using third-party provers, such as the aformentioned SMT-solvers and ATP, but also with the possibility to discharge to interactive theorem provers such as Coq and Isabelle. For instance it is possible to use the tools as an intermediate tool to reason about C, Java programs etc. This can be done through the main language of the platform called WhyML.
In this project we use Why3 as a language for reasoning about the pbject language, meaning we formalize the abstract syntax of our WHILE language directly in the WhuML. As the name suggests, WhyML is a language in the ML family. We will not go into too much detail about the specific syntax, for that we refer to the official documentation. We will however give a brief example of a program/function.

```
1    let rec aeval (a: aexpr) : int
2    variant { a }
3    ensures { eval_aexpr a sigma (Eint result) }
4    raises { E.Var_unbound -> eval_aexpr a sigma Eunbound }
5  = match a with
6    | Acst i -> i
7    | Avar v ->  E.find sigma v
8    | Asub a1 a2 -> aeval a1 - aeval a2
9    | AMul a1 a2 -> aeval a1 - aeval a2
10 end
```

Listing 1: Function for evaluation of arithmetic expressions in Why3

The function `aeval` is a function which evaluates an arithmetic expression as defined in Listing 1. The body of the function is similar to whatever other ML dialect. Where Why3 deviates a lot from other ML dialects is in the header of the function. We define the function using `let rec`, which means that the function is recursive and not necessarily pure. There are many different function declarations but we will only ever be

using function declarations including a let, since any other declaration cannot be exported, and we want to be able to extract the code into OCaml programs. Furthermore the program contains a contract, consisting of 4 optional parts, namely the follwing:

1. `variant`, which as previously described ensures total correctness of a program. The program may be any term with a well formed order. In the example function we use the arithmetic expression a.
2. `requires`, serves as the precondition of the program. For this particular function we have no precondition.
3. `ensures`, this is the postcondition of the program. In this case we state that the evaluation of an arithmetic expression should respect the semantics. The keyword *result* refers to the result of evaluation.
4. `raises`, states that when the function raises an exception the given predicate should hold. In particular we for the example function have that if a variable is unbound, and an exception is raised, then the evaluation results in a $\perp$ result, instead an integer.

From this it should be clear that Why3 tries to make the gap for program verification as small as possible, by allowing program constructs and controlflows familiar to "regular" programming languages, while also having ways of including assertions etc in the code.
One can then use either the Why3 IDE or the terminal interface to discharge the verification condition for the function. We will show how this is done in **??**.
Not only does whyML allow one to reason about a program through its verification condition, but it is also possible to reason about programs through the embedded logic-language (and strictly speaking the contract from before is part of that logic language). For instance it is possible to define predicates and, even more powerful, lemmas about programs. A lemma might look as follows:

```
1  lemma eval_aexpr_fun : forall a s b1 b2.
2     eval_aexpr a s b1 -> eval_aexpr a s b2 -> b1 = b2
3
```

Listing 2: Lemma stating determinism of arithmetic expressions

This lemma states that forall arithmetics expressions `a`, stores `s`, results `b1` and `b2` the semantics of arithmetic expressions ensures that `b1` and `b2` are the same. In other words the function is deterministic. After trying to prove this lemma, whether is proves or not, it will be "saved" as an axiom and can be used in following lemmas. It should furthermore be noted that WhyML is restricted to first order logic to only allow proof-obligations to be of first order logic.

So, by using Why3 we can reason about WHILE and wlp on a high level and by correctness-by-construction we can extract a program to either C or OCaml code[**TODO**]. We will in the later sections see how well the claim of high level of automation holds up, and discuss the possibility of extracting code from Why3 programs.

2.3.3 Isabelle/HOL

Isabelle is a proof assistant, also known as an interactive theorem provers. It allows for generic implementations of logical formalisms. The most used version of Isabelle

called Isabelle/HOL (from now referenced solely as Isabelle). The HOL stands for Higher Order Logic and uses higher order logics and builds on an LCF style. This means that lemmas can only be proven from previously defined functions and lemmas, corresponding to the inference rules of higher order logic.
Despite the fact that Isabelle, like Why3, is developed in ML, the syntax of Isabelle differs a lot and is fairly extensible in terms of user defined operators etc. The syntax might me too "cumbersome" to define at once and we will go through the relevant parts along the way in **??**.
TODO: What actually to write????

3 Implementation

In this section we go through the different components of the implementation and compare the Why3 and Isabelle implementation side by side. For each component we define a set of predicates which must hold for a correct implementation. We furthermore make an overall assessment of the qualities of the two languages gradually by comparing the result.

3.1 Syntax

Firstly our store σ is defined as a mapping from a polymorphic type to an Option Int. We define the syntax by simple ADTs, which closely resembles the syntax described earlier.

```
type store 'a = 'a -> option int

type aop = Mul | Sub

type aexpr 'a =
     | Acst int
     | Avar 'a
     | ABin (aexpr 'a) aop (aexpr 'a)

type bexpr 'a =
     | Btrue
     | Bfalse
     | Bleq (aexpr 'a) (aexpr 'a)
     | Band (bexpr 'a) (bexpr 'a)
     | Bnot (bexpr 'a)
```

```
type_synonym 'a state = "'a \<Rightarrow> int option"

datatype aop = Mul | Sub

datatype 'a aexpr =
    Cst int
  | Var 'a
  | BinOp "'a aexpr" aop "'a aexpr"
```

3.2 Arithmetic Expressions

3.2.1 Semantics

In **??**e defined the big-step semantics of WHILE. In both why3 and Isabelle it is possible to create inductive predicates. An inductive predicate is simply a predicate with a set of proof constructors defined inductively. This is a great tool to make a close to one to one mapping from the pen and paper version and a formalized version. We define the semantics of arithmetic expressions as follows:

```
type e_behaviour 'a = Enormal 'a | Eabnormal

```

```
inductive eval_aexpr (aexpr 'a) (store 'a) (e_behaviour int) =
  | EACst : forall n, s : store 'a.
              eval_aexpr (Acst n) s (Enormal n)
  | EAVar : forall x n, s : store 'a.
              s[x] = Some n ->
              eval_aexpr (Avar x) s (Enormal n)
  | EAVar_err : forall x, s : store 'a.
              s[x] = None ->
              eval_aexpr (Avar x) s Eabnormal
  | EASub : forall a1 a2 n1 n2, s : store 'a, op.
              eval_aexpr a1 s (Enormal n1) ->
              eval_aexpr a2 s (Enormal n2) ->
              op = Sub ->
              eval_aexpr (ABin a1 op a2) s (Enormal (n1 - n2))
  | EAMul : forall a1 a2 n1 n2, s : store 'a, op.
              eval_aexpr a1 s (Enormal n1) ->
              eval_aexpr a2 s (Enormal n2) ->
              op = Mul ->
              eval_aexpr (ABin a1 op a2) s (Enormal (n1 * n2))
  | EAOp_err1 : forall a1 a2, s : store 'a, op.
              eval_aexpr a1 s Eabnormal ->
              eval_aexpr (ABin a1 op a2) s Eabnormal
  | EAOp_err2 : forall a1 a2, s : store 'a, op.
              eval_aexpr a2 s Eabnormal ->
              eval_aexpr (ABin a1 op a2) s Eabnormal
```

Listing 3: semantics of arithmetic expressions

From Listing 3 it should be straight forward to see that each of the predicates, corresponds to each of the inference rules in **??**. Each subterm before `->` is the premises of the specific inference rule and the last term is the conclusion. There is however a slight difference. Initially we split the EAOp rule into an EAMul and EASub, since we though this would be a necessity, and more closely resembles how semantics usually are represented in text books and as such should be considered independently. However we found that this indeed is not necessary because of the rather trivial and mechanized reasoning about such binary operators and in fact reduces complexity of the automated proofs as is apparent in **??**.propsfig:aexpr_props
. The figure shows the 3 goals for arithmetic expressions. For a description of each property see **??**. In both versions of the semantics the goals are Valid, however we see a big difference in the steps it takes to validate it. For the initial version the steps for Goal 1 and Goal 3 takes roughly 2.5 as many steps as the current version. For Goal 2 the initial version requires 4 times as many steps as the current.
The new and improved semantic representation makes uses of a function `eval_op :: axpr -> (int -> int -> int)` and used it in the conclusion as `eval_aexpr (ABin a1 op a2) s (Enormal ((eval_op op) a1 a2)`. Hence we get a direct correspondence to the semantics presented in Figure 4, by removing EAMul and EASub and replacing it with

```
let function eval_op (op : aop) : (int -> int -> int) =
    match op with
    | Mul -> (*) | Sub -> (-)
    end
...

  | EABin : forall a1 a2 n1 n2, s : store 'a, op.
              eval_aexpr a1 s (Enormal n1) ->
              eval_aexpr a2 s (Enormal n2) ->
              eval_aexpr (ABin a1 op a2) s (Enormal ((eval_op op) n1 n2)
    )
```

```
INITIAL VERSION (version where we specifically distingush the two cases
    )
File semantics.mlw:
Goal eval_aexpr_total_fun'vc.
Prover result is: Valid (0.19s, 4170 steps).

File semantics.mlw:
Goal eval_aexpr_fun'vc.
Prover result is: Valid (0.58s, 14543 steps).

File semantics.mlw:
Goal eval_aexpr_total'vc.
Prover result is: Valid (0.03s, 428 steps).

NEW VERSION (Where we convert aop to a why3 binary operator).
File semantics.mlw:
Goal eval_aexpr_total_fun'vc.
Prover result is: Valid (0.09s, 1577 steps).

File semantics.mlw:
Goal eval_aexpr_fun'vc.
Prover result is: Valid (0.20s, 3449 steps).

File semantics.mlw:
Goal eval_aexpr_total'vc.
Prover result is: Valid (0.01s, 105 steps).
```

Figure 10: Steps to prove properties for arithmetic expressions

Notice furthermore we can define the behaviour of an operation as a Discriminated union of either a polymorphic `Enormal` behaviour or an `Eabnormal error`, we do so to reuse the same types for arithmetic and boolean expressions along with statements. In the current implementation the only way to not result in normal behaviour is when we have an unbound variable.
If we in the future were to add additional errors we let Eabnormal hold an ADT containing different errors. We tried looking into this matter, by adding division as an operator and adding additional semantics, with possibility of resulting in an `Ebnormal Ediv0` error. We however did not find a way to prove this, as the SMT solvers would time out on the properties. It is hard to tell exactly why as, adding addition as an operator we get only a fairly small increase in number of steps. TODO what are the numbers. But the additional error cases might make the actual formula discharged be grow exponetially. Considering the structure of the inductive predicate, we cannot simply let division be part of Binop, thus we would need seperate rules for the different operators.

3.2.2 Properties

The following paragraphs describes the properties we want to hold for arithmetic expressions. They are determinism and totality.

Determinism. We want our program to be deterministic which means each part of the semantics must be deterministic, i.e. the semantic relation must be functional.
To show that a judgement or inductive relation is functional, we have

$$\langle a, \sigma \rangle \Downarrow n \text{ and also } \langle a, \sigma \rangle \Downarrow n' \text{ then } n = n'$$

We can formalize this very easily in Why3 using a lemma, as such.

```
lemma eval_aexpr_fun_cannot_show : forall a, s : store 'a, b1 b2.
   eval_aexpr a s b1 -> eval_aexpr a s b2 -> b1 = b2
```

Listing 4: lemma for functional arithmetic expressions

The problem arises when we try to discharge this proof obligation to an SMT solver. It seems like this cannot directly proven. And the reason can be found in the usual way such formalism is proved for ASTs. More specifically, we do so by computational induction, and lemmas in why3 has no notion of induction. It is possible to transformations on a lemma, and a strategy such as `induction_pr` would correspond to this computational induction. Doing such a transformation we split the lemma into 6 subgoals. One for each inductive predicate rule. These can then be discharged. The first three goals corresponds to EACst EAVar and EAVar_err can be proven by Alt-Ergo, the remaining cases are for binary operators and these cannot be shown by an SMT solver. So in this we would want to use either Coq or Isabelle. We tried opening the proofs in both Isabelle and Coq. The Isabelle driver, simply does not work. gicing the following error:
There is no verification condition "eval_aexpr_fun_cannot_show".
And this suggest that we cannot reason about lemmas using Isabelle. Opening the proof in Coq allows us to perform usual reasoning in Coq. With our limited knowledge of Coq, we quickly abandoned this approach.
Instead we tried a different approach. Although why3 does not allow for induction, it is possible to define recursive lemmas. A recursive lemma differs from a regular lemma, in that it looks more like a "function". We can define `eval_aexpr_fun_cannot_show` as a recursive lemma as seen in Listing 5. The lemma is defined by two parameters the arithmetic expression and a state. Recursive lemmas is proved through a Verification condition, hence we define the variant, to ensure total correctness, and then we specify the post-condition in the `ensures`. The post-condition will serve as the conclusion of the lemma. This corresponds to the lemma in Listing 4. The body of the function will then dictate the unfolding of the recursive structure. Notice here, how the result of the body is (), and we only use ensures for the actual lemma.

```
let rec lemma eval_aexpr_fun (a: aexpr 'a) (s: store 'a)
    variant { a }
    ensures { forall b1 b2. eval_aexpr a s b1 ->
                            eval_aexpr a s b2 ->
                            b1 = b2
    }
  = match a with
    | Acst _ | Avar _ -> ()
    | ABin a1 _ a2 -> eval_aexpr_fun a1 s; eval_aexpr_fun a2 s
      end
```

Listing 5: Recursive lemma for functional arithmetic expressions

When discharged, a goal is defined through the verification condition of this lemma, in a similar manner to functions. After the goal (proven or not) the lemmas is axiomatised. The specific axiom for `eval_aexpr_fun` can be seen in Listing 6.

```
axiom eval_aexpr_fun :
  forall a:aexpr 'a, s:'a -> option int.
   forall b1:e_behaviour int, b2:e_behaviour int.
    eval_aexpr a s b1 -> eval_aexpr a s b2 -> b1 = b2
```

Listing 6: Axiom of functional lemma

The axiom clearly is equivalent to the lemma. Notice here, that the arguments for the lemma gets universally quantified.

Totality. Furthermore we want the semantics to be total, meaning all input has an output, such that we never encounter undefined behaviour. Formally we have

$$\forall a \in A, \sigma \in \Sigma.(\exists n.\langle a, \sigma\rangle \Downarrow n) \vee \langle a, \sigma\rangle \Downarrow \bot$$

where A is the set of all arithmetic expressions. Again we first formalised it as a plain lemma, but as the proof follows the structure of a, we had to refomulate this as a recursive lemma. The resulting lemma is shown in Listing 7.

```
let rec lemma eval_aexpr_total (a: aexpr 'a) (s : store 'a)
    ensures {
        eval_aexpr a s (Eabnormal Eunbound) \/ exists n. eval_aexpr a
    s (Enormal n)
    }
  = match a with
    | Acst _ | Avar _ -> ()
    | ABin a1 _ a2 -> eval_aexpr_total a1 s; eval_aexpr_total a2 s
    end
```

Listing 7: Axiom of functional lemma

Combination of determinism and totality. Another way to state that arithmetic expression are both total and deterministic is to define the actual total function. We can do so, again by a recursive lemma, as presented in Listing 8. Notice here that the lemma in this case will simulate the actual computation of the total function, thus we can use the `result` keyword in the post-condition. The post condition will ensure that the total function adheres to the semantics, since the inductive predicate `eval_aexpr` should hold for all `a` and `s`.

```
let rec lemma eval_aexpr_total_fun (a: aexpr 'a) (s: store 'a)
    variant { a }
    ensures { eval_aexpr a s result }
  = match a with
    | Acst n -> Enormal n
    | Avar v -> match s[v] with
                  | Some n -> Enormal n
                  | None -> Eabnormal Eunbound
                end
    | ABin a1 op a2 ->
      match eval_aexpr_total_fun a1 s, eval_aexpr_total_fun a2 s with
        | Enormal n1, Enormal n2 -> Enormal ((eval_op op) n1 n2)
        | Eabnormal e, _   -> Eabnormal e
        | _ , Eabnormal e  -> Eabnormal e
      end
    end
```

Listing 8: Lemma combining totality and determinism for arithmetic expressions

3.3 Boolean Expressions

3.3.1 Semantics and properties

The formulation of boolean expressions follows the same pattern as arithmetic expressions. We define an inductive predicate stating the inference rules in **??**. For reference see **??**.

predicate \lemma	&&	andb
&&	12008	56390
	1058	1058
	51046	51046
andb	*timeout*	14508
	1144	1144
	12146	12146

Table 1: Table of the number of steps taken to prove the three properties.

Likewise the properties we use for reasoning about correctness of our boolean semantics resemble the properties for arithmetic expressions. Thus we have the following lemmas:

1. `eval_bexpr_fun`
2. `eval_bexpr_total`
3. `eval_bexpr_total_fun`

In the formulation of the boolean semantics, we initially encounted some problems in proving our properties. In the first iteration of the inductive predicate we used the built0in operator &&, which works for boolean values. We then became aware of the function `andb`, which is defined in the standard library in module bool.Bool. `andb` is short-circuitting and presumably the && operator is as well. From the properties these also seem the two functions have equivalent semantics. Interestingly they do not require the same amount of steps in the proofs. Table 1 shows the result of proving the properties with different combinations of use of the operators. The columns defines the operator used in the lemma `eval_bexpr_total_fun`, and the rows defines which operator is used in the inductive predicate. Each line in a cell corresponds to the number of steps require for Alt-Ergo to prove the lemma. The lemmas are stated in the order from above.
From the figure, we can see that using && makes the two sub-lemmas a little simpler to prove, however for the total function lemma, it requires substantially more steps. On the other hand `andb` requires a bit more steps for the two sub-lemmas but is more than 4 times more efficient for the total function lemma. Most bizzarely is it that using `andb` in `eval_bexpr` and using && in `eval_bexpr_total_fun` cannot even be shown, despite the fact that their semantics evidently should be the same.

3.4 Statements

3.4.1 Semantics and properties

Semantics. Once again, we define the semantics by an inductive predicate. The semantics are generally not that complex, however there are many more cases to account for when dealing with statements. An example of this, is the inference rules for while. We must consider 4 different cases:

1. The boolean condition is true, and thus the body "evaluates" to some new state, which is used for the next iteration of the while loop.

2. The boolean condition is false, and thus the loop ends in the same state.

3. The boolean condition results in abnormal behaviour and thus the enitre statement should result in abnormal behaviour.

4. The body results in abnormal behaviour and likewise the entire loop results in abnormal behaviour.

However for assertions, we only consider a single case, since as mentioned in the inference rules **??**, we dont consider assertions in the operational semantics, but rather include them for strengthening the verification condition of a program.

Properties. The properties for asserting the correctness of the implementation of the semantics is build upon the lemmas regarding totality and determinism of boolean and arithmetic expressions. Hence we consider the lemma `eval_stmt_deterministic`. Unfortunately, our trick from the previous grammar constructs does not render useful for statements. Running the recursive lemma for 300 seconds in Alt-Ergo, does not provide a proof. There might be many reasons for this.
First and foremost we cannot include while in the recursive lemma. The reason for this, is that we have to include the variant for a recursive lemma to even discharge the proof. I.e. if we cannot prove termination of a lemma it cannot hold true in the system if why3. Since this would only entail partial correctness. It should be clear from the ESWhileT case of the inference rules, that we do not reduce the structure of s and hence s does not respect the well founded order. Had we included a variant in WHILE we might have been able to express the termination.
We then considered the proof without including the While in the language. Also in this case we could not find a proof. We then tried to remove If. Yet again, the SMT solvers timed out. Finally we found that removing Sequences, would allow us to prove the determinism.
specifically, when excluding Seq, If, While, it took Alt-Ergo 5956 steps to prove the lemma. While only excluding Seq and While the prover to 13965 steps. Again this shows how 3 additional inference rules can increase the number of step by a significant amount.
One perculiarity we further found was that one can compare mappings by = without importing the module map.MapExt (in the standard library), which specifically defines extensionality of mappings. But again this might be overlapping instances of the build in functionality and the standard library. However this, along with the conumdrum of && and `andb`, begs the question as to how well the standard library is structured.

3.5 Assertion Language (Formulas)

Formalization. As mentioned in **??**, we cannot formalize the assertion language in a similar manner to the onther language constructs because of quantifers. We instead decided to use a predicate to describe the semantics. As Formulas are a logic construct the choice naturally was to use a predicate over a function, since predicates are part of why3's logical languages and therefore is useful in reasoning about logics. The predicate `eval_closed_formula` is shown in Listing 9. The semantics for conjuction implication and negation are rather simple and directly follows the corresponding semantics of why3's logic. Universal quantification uses the `forall` defined in why3, and updates the state accordingly. For the term expression we use the semantics defined earlier for binary expressions. We state that for a term to be true, the term should

evaluate to true under the judgement of binary expressions. If it evaluates to true, the term is trivially true. If it does not evaluate to true, there are two potential evaluations. Either it can evaluate to false or it results in $\bot$. This is problem in formulas such as $\neg(x \leq 10)$ and x is unbound. The inner expression $x \leq 10$ will be a false term and by negation the entire formula is true. This should not be possible as formulas with unbound variables should not make sense. Hence the semantics only works for closed formulas. We therefore need a wrapper function, so we only consider closed form formulas. The wrapper `eval_formula` will first check if any free variables exists if this is the case, then the formula is false. In case we have a closed formula, we evalaute the formula using the defined predicate. Determining if any free variables exists in the formula, we recursively traverse the formula to ensure all variables are bound in the given store, and in case of universal quantification, we bind the variable to 0, so it is present in the store and the value is irrelevant.

```
predicate eval_closed_formula (f: formula 'a) (st: store 'a)  =
  match f with
  | Fterm b -> eval_bexpr b st (Enormal true)
  | Fand f1 f2 -> eval_closed_formula f1 st /\ eval_closed_formula f2
    st
  | Fnot f -> not (eval_closed_formula f st)
  | Fimp f1 f2 -> eval_closed_formula f1 st -> eval_closed_formula f2
    st
  | Fall y f -> forall n. eval_closed_formula f st[y <- n]
  end

predicate eval_formula (f : formula 'a) (st : store 'a) =
  if is_closed_formula f st then eval_closed_formula f st
  else false
```

Listing 9: Predicate defining the semantics of formulas

Why the semantics are non-blocking. We mentioned briefly in **??** for semantics of statements that our semantics are non-blocking. The reason we opted for this, was that we preemptively knew that dealing with quantifiers would be problematic. We want to make the code extractable to ocaml, and this means we cannot rely on the runtime of why3 and hence we would need to implement some algorithm to deal with quantified proofs. This was not an objective of this project.
Possible solutions for implementing a blocking semantics are to either have two different assertion languages, one for executable assertions (I.e. user-specified assertions for runtime) and another language for assertions used in the verification condition generation to strengthening the precondition. In the executable assertions we not allow quantifiers, but all other logical constructors would be included. In the assertions for VC we include all the constructs.
Another solutions would be to simulate blocking using the predefined constructors. The expressiveness of binary expressions are equivalent to the assertion language with quantification, and any assertion which should fail can be simulated using an unbound variable. For instance, we could have

```
if assertion then skip else s = x
```

where `assertion` defines the assertion that must hold, and `x` is an unbound variable.

3.6 Evaluator

With our formalization of the semantics, we can define an evaluator and must do so if we want an extractable evaluator for WHILE. We showed earlier how, the total function lemmas got folded out and expressed the total function satisfying the semantics. The evaluator essentially implements this approach, although modelling the store as a mapping is not satisfactory. Firstly, the semantics for statements results in a store under normal evaluation and for an evaluator which returns a mapping to be useful we want the be able to reason about the final store. This can be done by mappings, but requires any post processing to have knowledge of the variables assigned in the program either by the user or computationally. An easier approach would be to have a data-structure which stores key value pairs. We ended up using a mutable list of key value pairs, which has some promises about its state.

3.6.1 Modelling a store

To do so, we had to implement a seperate module `ImpMap`, presented in Listing 10, since we ran into a number of roadblocks using any of the predefined data-structures in the standard library. These will be explained further in **??**. We define the store, through a record called state. This record contains two different fields. Firstly we have lst which is a linked list of a key value pair, whilst the second state is a mapping of the same type as the state used for defining the semantics. Notice here that the model is marked with ghost. This means that the model can only be used in a logical context and therefore cannot manipulate any computations. We use the model for reasoning about the store through the verification conditions. For instance, for instantiation of the state by `empty` should ensure that the image of the store is $\{\text{None}\}$. Likewise when adding a key value pair in to the list the model should also contain this mapping. Lastly, for `find`, we try to find the key in the list and if it does not exist we throw an error. This enables us to easily propagate the error throughout the evaluation.

```
type model_t = M.map int (option int)

predicate match_model (k: int) (v: int) (m : model_t) =
match M.get m k with
| None -> false
| Some v' -> v = v'
end

function helper (m : model_t) (pair: (int,int)) : bool =
let (k,v) = pair in match_model k v m

type state = { mutable lst : list (int, int);
               ghost mutable model : model_t
             }
      (* invariant { for_all (helper model) lst } *)
      (* by { lst = Nil ; model = (fun (_ : int) -> None) } *)

exception Unbound

function domain (s : state) : M.map int (option int) = s.model

let function empty () : state
ensures {forall k. M.get result.model k = None }
= { lst = Nil ; model = (fun (_ : int) -> None) }

let add (k: int) (v: int) (s : state) : ()
writes { s.lst }
```

```
writes { s.model }
ensures { s.model = M.((old s.model)[k <- Some v]) }
ensures { hd s.lst = Some (k,v) /\ match_model k v s.model }
= s.lst <- (Cons (k,v) s.lst);
  s.model <- M.(s.model[k <- Some v])

let rec find (k: int) (s : state) : int
  variant { s.lst }
  ensures { match_model k result s.model}
  raises { Unbound -> M.get s.model k = None}
= match s.lst with
  | Cons (k', n) s' -> if andb (k <= k') (k' <= k) then n
     else find k { lst = s'; model = s.model}
  | Nil -> raise Unbound
  end
```

Listing 10: Model for a store

3.6.2 Evaluation

For the evaluation we define the imperative store, and four functions, one for each construct, arithmetic epressions, boolean expressions and statements, and a "top-level" function which evaluates evaluates a statement and extracts the result of evaluation, and also clearing the store.
In the contract of each function, we describe both what happens under normal execution and what happens under abnormal execution, using the `ensures` and `raises`, for instance we have:

```
let rec aeval (a: aexpr id) : int
  variant { a }
  ensures { eval_aexpr a (IM.domain sigma) (Enormal result) }
  raises { IM.Unbound -> eval_aexpr a (IM.domain sigma) (Eabnormal
    Eunbound) }
= match a with
  | Acst i -> i
  | Avar v ->  IM.find v sigma
  | ABin a1 op a2 -> (eval_op op) (aeval a1) (aeval a2)
end
```

Listing 11: Evaluation of arithmetic expression

Listing 11 present the evaluation of arithmetic expressions. Like for the lemma about the semantics being a total function, we want the evaluator to adhere to the semantics from the inductive predicate. In the case of an unbound variable the `Imperative Map find` will raise an exception, in this case we will ensure that the behaviour of the evaluation is Eabnormal Eunbound. It should be clear, that this can easily be extended to other errors, by doing a conjunction of all possible (exception $\rightarrow$ semantical behaviour) pairs. We follow the same structure for evaluation of boolean expressions and statements. The vc is easily proved for arithmetic and boolean expressions. Again we cannot show it for statements, neither for total nor partial correctness.

3.7 Verification Condition Generation

The other major part of the projet is the formalisation of an extractable verification condition generator. in **??** we will introduce the formalisation of the weakest liberal precondition, but before we can do so, we need to address the matter of variables in formulas.

3.7.1 Variable substitution

From **??** it should be clear that there are multiple situations where we want to substitute all free occurences of a variable by an new variable. For instance when s is $x := a$ in $wlp(s, Q)$ where Q is a formula, and s is the assignment of a to x, we want to update each occurence of x in Q, but do not substitute bound instances of x. The rules are as follows, we only consider cases where variables might occur. f_i is formulas, x, y, z are variables and b_i, a_i describes boolean expression and arithmetic expressions respectively. Conjuction for booleans and formulas are equivalent.

$$
\begin{aligned}
x[x \mapsto y] &= y \\
z[x \mapsto y] &= z
\end{aligned}
$$

$$
\begin{aligned}
(a_1 \leq a_2)[x \mapsto y] &= a_1[x \mapsto y] \leq a_2[x \mapsto y] \\
(b_1 \wedge b_2)[x \mapsto y] &= b_1[x \mapsto y] \wedge b_2[x \mapsto y] \\
(\neg b)[x \mapsto y] &= \neg b[x \mapsto y]
\end{aligned}
$$

$$
\begin{aligned}
(f_1 \Rightarrow f_2)[x \mapsto y] &= f_1[x \mapsto y] \Rightarrow f_2[x \mapsto y] \\
(\forall x.f)[x \mapsto y] &= \forall x.f \\
(\forall z.f)[x \mapsto y] &= \forall z.f[x \mapsto y]
\end{aligned}
$$

The $subst$ must also adhere to the rule that y must be free for x in whatever the expression/formula, meaning a free occurence of x must not be bound when substituted by y.
From the formulation of the substitution function it should be easy to see, that we recurse down the syntax tree. The most interesting case is when we meet a quantifier. If the variable x we want to substitute is bound then we stop recursion, as all occurences of x will then be bound.
It should be noted that we do not ensure that x must be free for y, thus the function is rather unsafe. We only ever use it in a context where this cannot happen, since we instantiate a new variable that does not occur in the formula, bound or free. The variable is so-called fresh in the formula.
The way we generate a fresh variable, is by traversing the syntax tree of the formula and recursively taking the maximal variable value and adding 1. This works since we use integers as identifiers. A similar result can be obtained by all infinite enumerable sets. For instance if identifiers were strings, one could add a new character to the longest variable name.
This seems rather trivial, however we formulated predicates that state whether a variable is fresh in an expression or formula. We then afterward tried making a lemmas which stated:

```
lemma fresh_var_is_fresh : forall f v.
    v = fresh_from f -> fresh_in_formula v f
```

The lemma simply asserts that generating a fresh variable v from f implies that v is fresh in f. We got the inspiration for this approach from [**TODO: WP revisited**]. In their work they formulate the fresh_from function using axioms. That is they provide a function declaration and then the function "computes" the value based on a set of

axioms. For us this is not a viable approach, as such functions cannot be extracted. Therefore we defined the function ourself and stated axiom as a lemma to ensure that `fresh_from` always ensures a fresh variable. Although this seems like a trivial result, we were not able to prove this using the SMT solvers. We tried to define this system in isabelle to see how difficult it was to prove using a proof assistant, where one has more control. We succeeded in proving this. But since, this does not entail that there might be a bug in the why3 code. The isabelle proof might not be the cleverest way to prove it, but we proved it the following way. let V be the set of all vars in an arithmetic expression a. Then assuming v is the maximum identifier in a, we show that $\forall x \in V.v \geq x$. We do so by induction. By this we can then show $v + 1 \notin V$. We then made a lemma stating that given $v \notin V$ then v is fresh in a. From this we can show that $v + 1$ is fresh in V. We then do the same for boolean expression and formulas, and the proof is done. The full proof can be seen in **??**.

3.7.2 Weakest Liberal Precondtion

Implementation of Weakest Liberal Precondition calculus. We implemented the rules for weakest liberal precondition by a recursive function that directly follows the inference rules. We focus on two of the rules, namely assignments and while statements, which both does substitution in the formula. For assignments we define the rule as follows:

```
| Sass x e -> let y = fresh_from q in
              Fall y (Fimp (Fand (Fterm (Bleq (Avar y) e))
                                 (Fterm (Bleq e (Avar y)))) (subst_fmla
    q x y))
```

We first generate a fresh variable, which is then substituted into the postcondition Q. We further follow the semantics, however since we do not have equality we use $y \leq e \wedge e \leq y$.
The other interesting rule, for while, is implemented as follows:

```
| Swhile cond inv body ->
  Fand inv
       (abstract_effects body
       (Fand (Fimp (Fand (Fterm cond) inv) (wp body inv))
             (Fimp (Fand (Fnot (Fterm cond)) inv) q))
       )
```

Here the interesting thing is the function `abstract_effects`. This function quantifies over all assigned variables in the body of the loop and substitute the quantifiers with the free variables in the formula $((cond \wedge inv) \Rightarrow wp(body, inv)) \wedge ((\neg cond \wedge inv) \Rightarrow Q)$. like `fresh_from` this function is inspired by [**TODO**]. The story is the same. They used axioms to define the function, whereas we have to implement it. `abstract_effects` takes a statement s and formula f. If the statement is an assignment a fresh variable is made and substituted into f, and we then quantify the freshly made variable. This is not the most efficient solution, as multiple assignment to the same variable will create unused quantifiers. All the cases will traverse the abstract syntaxt tree or is a leaf and does nothing.

```
let rec function abstract_effects (s : stmt int) (f : formula int) :
    formula int
  variant { s }
= match s with
  | Sskip | Sassert _ -> f
```

```
  | Sseq s1 s2 | Sif _ s1 s2 -> abstract_effects s2 (abstract_effects
    s1 f)
  | Sass x _ -> let v = fresh_from f in
                let f' = subst_fmla f x v in
                Fall v f'
  | Swhile _ _ s -> abstract_effects s f
  end
```

Whilst we implemented this function directly, we still want the properties for the function to hold. specialization. Firstly, we have

```
  lemma abstract_effects_specialize : forall st : store int, s f.
    eval_formula (abstract_effects s f) st -> eval_formula f st
```

which states that evaluation of applying abstract effects off s on f in state st implies that f evaluates to true in st. Essentially it states that if we quantify the variables in f and the formula is true under the quantification, then f also hold if the variables are not quantified. THIS DOES NOT MAKE ANY SENSE??? PLEASE LOOK AT IT TOMORROW.

Quatification over conjunction. Secondly, we have

```
lemma abstract_effects_distrib_conj : forall s p q st.
   eval_formula (abstract_effects s p) st /\ eval_formula (
    abstract_effects s q) st ->
   eval_formula (abstract_effects s (Fand p q)) st

```

Which states that if we apply abstract effects of s on two formulas p and q and they both evaluates to true, then evaluating the result of applying the abstract effects of s over the conjunction of p and q, must also be true.

Monotonicity. Thirdly, we have the property of monotonicity.

```
lemma abstract_effects_monotonic : forall s p q.
   valid_formula (Fimp p q) ->
   forall st. eval_formula (abstract_effects s p) st ->
   eval_formula (abstract_effects s q) st

```

Essentially what monotonicity states, is that applying additional assumptions to a formula will not change the meaning. It should be noted that we want to quantify all states in the entailment, as the property does not hold for a fixed state.

Invariance. Lastly, we consider the notion of invariance. TODO: I DONT THINK THIS IS NECESSARY.
In [**TODO**] the axiomatized version of the following lemma, is used to define which variables should be quantified. We explicitly state this in the body of `abstract_effects`, but for good measure we include it as a lemma. The property states that if a the formula q abstracted by s is true in some state, then the weakest precondition on the same abstraction on q by s is true should also be true. When [] uses this as an axiom, it ensures, that `abstract_effects`

Proofs of properties for abstract_effects We have not been able to automatically prove these lemmas in why3. At the moment we have a partial proof for the function in Isabelle. ???? WRITE SOME MORE ????

Properties and Soundness of WLP One of the main goals in making a formally verified Verification condition generator is to ensure the correctness of the implementation. We consider the correctness through its soundness. To do so we must consider two of the same properties we just stated for a`abstract_effects`, namely monotonicity and conjunction distribution.
Monotonicity of WLP is that if for two formulas p and q the formula $\vDash p \Rightarrow q$ then $\vDash wp(s,p) \Rightarrow wp(s,q)$. Notice again that this must hold for all statements and states. Distribution of weakest precondition over conjuction, is similar to the lemma of abstract effects only the transformation on p and q are now considered for wp.

```
lemma monotonicity: forall s p q.
      valid_formula (Fimp p q) -> valid_formula (Fimp (wp s p) (wp s q)
    )

lemma distrib_conj: forall s sigma p q.
      eval_formula (wp s p) sigma /\ eval_formula (wp s q) sigma ->
      eval_formula (wp s (Fand p q)) sigma
```

Once again we, we not able to directly show this. We tried to unfold the recursion on s, in a similar manner to how we proved determinism. We did not achieve anything by this. We have a formalized proof of both properties in Isabelle. The proof for monotonicity can be seen in Listing 12. As mentioned we prove the lemma by induction on s. And mark p and q as arbitrary, since the lemma should hold for any non-fixed p and q. The proof only shows the cases for sequences, assignments and while. The case for sequences are actually directly proved from the assumptions. We only distinguish this case because we need to simplify the other trivial cases. For both assignment and while we used the sledgehammer to find the proofs by metis, which is a complete automatic theorem prover for first order logic with equality[**TODO sledgehammer paper**].

```
lemma monotonicity : "valid_formula (FImp p q) \<Longrightarrow>
  valid_formula (FImp (wp s p) (wp s q))"
proof(induction s arbitrary: p q)
  case (SSeq s1 s2)
  then show ?case by auto
next
  case (Sassign x1 x2)
  then show ?case
    by (metis abstract_effect_writes abstract_effects.simps(2)
         eval_formula.simps(2) valid_formula_def wp.simps(3))
next
  case (Swhile x1 x2 s)
  then show ?case
    by (metis abstract_effect_writes abstract_effects.simps(2)
        eval_formula.simps(2) valid_formula_def wp.simps(3))
qed (simp_all add: valid_formula_def)
```

Listing 12: Proof of monotonicity in Isabelle

For distribution over a conjuction, the proof is straight forward. We do induction with the same setup. Again the sequences, assignments and while cannot trivially be proved by simplification, but the same metis proof used for monotonicity can be used for all three cases.
With these two properties, we should be able to show soundness for the wp function. We decided to not prove the completeness of the function because this says something about the expressiveness of the function, whereas it is more important to ensure that the function is correct. Using soundness to show correctness is two-fold. On the one hand showing the soundness of wp ensures that $\{wp(s,Q)\}s\{Q\}$ is valid for partial correctness for all s and q, ensures that we cannot generate invalid verification conditions. On

the other hand, we already know weakest precondition to be sound and proving it for the function wp ensures that our implementation adheres to the semantics or atleast an equivalent semantic.
We have not gotten to show this. TODO why have we not done this?????

3.8 Extraction of code

Why3 makes it possible to extract whyML code to either Ocaml or C code. One of the main goals of this project was to make the verified code extractable. To achieve this we had to adhere to some limitations. Mostly these are related the expressiveness and hierachy of whyML.

Extraction is correct by construction. The extraction of whyML programs is correct by construction[**TODO : manual for why3**]. This means that each syntactical object is directly translated into an equivalent object in the target language. For instance a program function written in whyML denoted by either `let`, `let function`, `let rec function` or `let rec` directly translate into its equivalent Ocaml function. This ensures that extracted code does the same when exported. On the downside this means that we can only extract our code to Ocaml, but not to C, since there is no language defined notion of Abstract Data Types. Thus our choice to represent our object language as ADT's have limited our ability to extract code. However structuring the code in C friendly manner would likely become quite tedious.

Program functions and Logical functions. Just as the object level limits what we can do in terms of extracted code, so is the logical level. There is a clear distinction between logical functions and program functions in why3. All program functions is specified with a `let` and can be extracted, while the logical functions can be used to reason about the program functions. Actually extracting the code has been a bit of a challenge. Firstly, as mentioned earlier we compromised in regards to assertions in the operational semantics of the program. For the reason that it is hard to argue about logical quantifiers on a program level, while it is much easier to do on a logical level. Hence why evaluation of Formulas is done with a predicate and not a program function. Furthermore, making the actual evaluator was quite troublesome. For defining the actual semantics we used inductive predicates, which is a logical construct, and this is extremely useful because we can argue the correctness of evaluation of a statement on a program level. In this reasoning we used maps, which is simply functions, with some syntactical sugar for updates and computations. The problem arises when we need to define the environment for the actual interpreter.
We were not able to find any module which was extractable and would adhere to the same logical meaning as that of maps. Hence we had to implement the module explained in **??**. And while there might be better ways to handle the store, Our current approach seems to suffice.
In regards to the verification condition generation, we did not run into too much trouble. As mentioned previously our approach to the weakest precondition generation follows the same structure as WP revisited in why3[], but deviates in how functions are defined. Axiomatized functions can clearly not be exported, since they dont have a function body but simply must adhere to a set of axioms. And therfore we need to prove explicitly that our function is correct, whereas axiomatization will be correct by definition. But actually extracting the wp function is rather simple, as it essentially just transforms an ADT. And clearly can be defined in on the program level.

Using the extracted code. The code is essentially split into two different functionalities. The evaluator does imperative evaluation on some statement in the mutable environment. We can extract the code for evaluation by the following command:

```
why3 extract --recursive -D ocaml64 -L . evaluator.mlw -o eval.ml
```

This will tell to recursively add dependencies into the module defined in the evaluator.mlw file, using the ocaml64 driver and create a module in the file eval.ml. We can then use the module in an ocaml project (where we might define a parser etc. for the language). Listing 13 shows an example of a very simple program. The program, defines two WHILE programs, which are very simple. `prog` is a simple assignment. One thing to note is that because we used the int type in why3, we have to use the $Z.of\_int$ because int's in why3 are unbounded integers from the zarith library. `prog'` defines a slightly more complicated program, which does 2 assignments, and the second assignment tries to assign an expression with an unbound variable to a variable.

```
open Eval
let prog =
              Sass  (Z.of_int 1, ABin (
                         Acst (Z.of_int 5)
                         , Mul
                         , Acst (Z.of_int 10)))
let prog' = Sseq (
              Sass  (Z.of_int 1, ABin (
                         Acst (Z.of_int 5)
                         , Mul
                         , Acst (Z.of_int 10)))
              , Sass (Z.of_int 2 , ABin (
                          Acst (Z.of_int 5)
                        , Add
                        , Avar (Z.of_int 3))))
let res = eval_prog prog
let () = List.iter (fun (k,v) -> Format.printf "%d, %d\n" (Z.to_int k)
    (Z.to_int v)) res
let () = List.iter (fun (k,v) -> Format.printf "%d, %d\n" (Z.to_int k)
    (Z.to_int v)) sigma.lst
let res' = eval_prog prog'
```

Listing 13: ocaml program using the evaluator

Compiling the program with

```
\textdollar ocamlbuild -pkg zarith main.native
```

and running it by:

```
\textdollar ./main.native
```

The following output is produced:

```
1, 50
Fatal error: exception Eval.Unbound
```

The evaluation of the first program is that variable 1 is 50. The second List.iter prints nothing, this shows that we correctly resets the environment. Lastly `prog'` correctly produces an Unbound exception.

The procedure for the verification condition generator is the same. It is worth noting that we have not implemented any way to use the generated formula for anything. And thus it may render useless for now to extract it. In case one want to actually verify the correctness of a WHILE program, we can use the povers of Why3 and use the meta level of logic using a goal. Take the following program

```
q = 10;
r = 5;
res = 0;
ghost = q;
while (q > 0)
invariant {(res $\leq$ (ghost - q) * r && (ghost - q) * r $\leq$ res)
    /\ 0 \leq q} {
      res = res + r;
      q = q - 1;
};
assert {res $\leq$ ghost * r /\ ghost * r $\leq$ res};
```

Listing 14: WHILE program which multiples q and r by repeated addition

The syntax should be straight forward. The program will define 4 variables, q, r, res and $ghost$. The program will execute the while loop q times. In the body of the loop we add r to res and decrement q by 1. We keep as invariant that $res = (ghost - q) * r$. Lastly we assert that res is the orignal value of q, described by $ghost$, multiplied with r. Writing this as an AST, we have:

```
let function prog () =
  (* var 2 is q *)
  (Sseq (Sass 2 (Acst 10))
  (* var 1 is res *)
  (Sseq (Sass 1 (Acst 0))
  (* var 0 is r *)
  (Sseq (Sass 0 (Acst 5))
  (* Ghost var = q *)
  (Sseq (Sass (-1) (Avar 2))

  (Sseq (Swhile (Bleq (Acst 0) (ABin (Avar 2) Add (Acst 1)))
        (* res <= (ghost - q) * r /\ res >= (ghost - q) * r *)
        (Fand (Fterm (Band (Bleq (Avar 1)  (ABin (ABin (Avar (-1)) Sub
    (Avar 2)) Mul (Avar 0)))
              (Bleq (ABin (ABin (Avar (-1)) Sub (Avar 2)) Mul (Avar 0))
    (Avar 1))))
        (* 0 <= q *)
             (Fterm (Bleq (Acst 0) (Avar 2))))
        (* NOW BODY OF THE LOOP *)
        (Sseq (Sass 1 (ABin (Avar 1) Add (Avar 0)))
              (Sass 2 (ABin (Avar 2) Sub (Acst 1)))))

        (* Assert {res = ghost * r} *)
        (Sassert (Fterm (Band (Bleq (Avar 1) (ABin (Avar (-1)) Mul (
    Avar 0)))
        (Bleq (ABin (Avar (-1)) Mul (Avar 0)) (Avar 1)))))))))))

```

Listing 15: WHILE AST in why3

We can then construct a goal checking if the weakest precondition of this program is valid.

```
goal mult_is_correct :
   valid_formula (wp (prog ()) (Fterm (BCst true)))
```

Feeding this to an SMT solver such as Eprover, we can show the program to be correct, in around half a second.

4 Results

In this section we go over the results from the project. We will cover the findings, what limitations we encountered, if we succeeded in making a verified Verification Condition Generator in Why3 and if it is even feasible to do so.

4.1 Overview of Findings from Implementation

One of the goals we set out to do, was to investigate the expressiveness and effectiveness of different SMT solvers and ATP's in the why3 backend. The idea was that we should do this through the implementation of the interpreter and the VC generator. From **??** it should be apparent that most of the properties we want to hold cannot be proved, (atleast from our experiments), hence we have abandoned the idea of making an analysis of which provers are capable of show specific lemmas. What we can say from the implementation is that whilst we are able to model some of the lemmas other than the straight forward way, it is not easy to verify programs in Why3. Although it might be possible to model the lemmas in a way such that the goals we set out to do can be met, especially with better knowledge of transformations in the Why3 IDE. We still dont have enough experience with Why3 to use these effectively and from the naming of the transformations, and their fairly limited descriptions in the official manual, one might as well just use Coq when transformations are needed.

4.2 limitiations for automated verification

In *WP revisited in Why3* they make heavy use of both transformations and Coq to prove their lemmas, and whilst it is almost 10 years ago, it seems Why3 is still not at a place where this can be achieved with 100 percent of automated proofs. In fact we would argue, that Why3 were more automated in 2012. The reason for this, can be seen in how most of the Coq proofs are done. They start by adding a tactic to their Coq proofs, as follows:

```
1 Require Import Why3.
2 Ltac ae := why3 ``alt-ergo'' timelimit 5
```

By this they are able to interface with Why3 from Coq. So most of the proofs in the article, massages the proofs and then discharges the rest of the program to a Why3 backend SMT solver, in this case Alt-ergo. This feature, is alledgedly still available[**TODO:LIERS**]. However the files one has to use is nowhere to be found in the folders they are supposed to be in. If this is a relic of the past or badly documented is hard to tell. In any case, we were not able to get this feature to work. When also considering that we are not able to discharge the proof obligation to Isabelle, we are in a place where we have less tools available than they had in the 2012 article.

4.3 Has this project been useful?

Although we have accomlplished the goals we set out to do, we still have gotten a good reason about or formalization of the weakest precondition calculus and how we can formalize the semantics in a programming language such as Why3. But to say that our product is useful would be a stretch. Without verifying our formalization we could just aswell have written the VC generator in another language. For what is it worth we have a partially proved implementation in Isabelle. We began exploring using Isabelle too

late in the project to finish an implementation we could extract. Had we had more time, we would have finished the formalization and done a side by side comparison of the Why3 and Isabelle. In any case, it seems like it is "easier" to formalize a Verification Condition Generator in Isabelle.

5 Conclusion

Was we successful at all and is in feasible to make VCGen verified.
Thanks for reading!

References

[1] Stephen A. Cook. "Soundness and Completeness of an Axiom System for Program Verification". In: *SIAM Journal on Computing* 7.1 (1978), pp. 70–90. DOI: 10.1137/0207005. eprint: https://doi.org/10.1137/0207005. URL: https://doi.org/10.1137/0207005.

[2] 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.