UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Project outside course scope

Jacob Herbst(mwr148), Matilde Broløs (jtw868)

IFC

An imperative language with static verification

Advisor: Ken Friis Larsen

1. April 2022

Abstract

Contents

1 Introduction

2 Background

2.1 Language

IFC is a small imperative programming language with a build in assertion language enabeling the possibility for statically verifying programs by the use of external provers. The assertions also enables us to dynamically check the program during evaluation. In this section we will present the syntax and semantics of both the imperative language and the assertion language.

$$
\begin{array}{lcl}
\langle \mathit{statement} \rangle & ::= & \langle \mathit{statement} \rangle \text{ ';' } \langle \mathit{statement} \rangle \\
 & | & \langle \mathit{ghostid} \rangle \text{ ':=' } \langle \mathit{aexpr} \rangle \\
 & | & \langle \mathit{id} \rangle \text{ ':=' } \langle \mathit{aexpr} \rangle \\
 & | & \text{'if' } \langle \mathit{bexpr} \rangle \text{ '\{' } \langle \mathit{statement} \rangle \text{ '\}'} \\
 & | & \text{'if' } \langle \mathit{bexpr} \rangle \text{ '\{' } \langle \mathit{statement} \rangle \text{ '\}' 'else' '\{' } \langle \mathit{statement} \rangle \text{ '\}'} \\
 & | & \text{'while' } \langle \mathit{bexpr} \rangle \ \langle \mathit{invariant} \rangle \ \langle \mathit{variant} \rangle \text{ '\{' } \langle \mathit{statement} \rangle \text{ '\}'} \\
 & | & \text{'\#\{' } \langle \mathit{assertion} \rangle \text{ '\}'} \\
 & | & \text{'skip' | 'violate'} \\
 & & \\
\langle \mathit{invariant} \rangle & ::= & \text{'?\{' } \langle \mathit{assertion} \rangle \text{ '\}'} \\
 & | & \langle \mathit{invariant} \rangle \text{ ';' } \langle \mathit{invariant} \rangle \\
 & & \\
\langle \mathit{variant} \rangle & ::= & \text{'!\{' } \langle \mathit{aexpr} \rangle \text{ '\}' | } \epsilon \\
 & & \\
\langle \mathit{assertion} \rangle & ::= & \text{'forall' } \langle \mathit{id} \rangle \text{ '.' } \langle \mathit{assertion} \rangle \\
 & | & \text{'exists' } \langle \mathit{id} \rangle \text{ '.' } \langle \mathit{assertion} \rangle \\
 & | & \text{'}\sim\text{' } \langle \mathit{assertion} \rangle \\
 & | & \langle \mathit{assertion} \rangle \ \langle \mathit{assertionop} \rangle \ \langle \mathit{assertion} \rangle \\
 & | & \langle \mathit{bexpr} \rangle \\
 & & \\
\langle \mathit{assertionop} \rangle & ::= & \text{'/}\backslash\text{' | '}\backslash\text{/' | '=>'} \\
 & & \\
\langle \mathit{bexpr} \rangle & ::= & \text{'true' | 'false' |} \\
 & | & \text{'!'}\langle \mathit{bexpr} \rangle \\
 & | & \langle \mathit{bexpr} \rangle \ \langle \mathit{bop} \rangle \ \langle \mathit{bexpr} \rangle \\
 & | & \langle \mathit{bexpr} \rangle \ \langle \mathit{rop} \rangle \ \langle \mathit{bexpr} \rangle \\
 & | & \text{'(' } \langle \mathit{bexpr} \rangle \text{ ')'} \\
 & & \\
\langle \mathit{bop} \rangle & ::= & \text{'\&\&' | '||' |} \\
 & & \\
\langle \mathit{rop} \rangle & ::= & \text{'<' | '<=' | '=' | '/=' | '>' | '>='} \\
 & & \\
\langle \mathit{aexpr} \rangle & ::= & \langle \mathit{id} \rangle \\
 & | & \langle \mathit{ghostid} \rangle \\
 & | & \langle \mathit{integer} \rangle \\
 & | & \text{'-'}\langle \mathit{axpr} \rangle \\
 & | & \langle \mathit{aexpr} \rangle \ \langle \mathit{aop} \rangle \ \langle \mathit{aexpr} \rangle \\
 & | & \text{'(' } \langle \mathit{aexpr} \rangle \text{ ')'} \\
 & & \\
\langle \mathit{aop} \rangle & ::= & \text{'+' | '}-\text{' | '}*\text{' | '/' | '\%'} \\
 & & \\
\langle \mathit{ghostid} \rangle & ::= & \text{👻 } \langle \mathit{string} \rangle \\
 & & \\
\langle \mathit{id} \rangle & ::= & \langle \mathit{string} \rangle
\end{array}
$$

Table 1: Grammar of IFC

Put in also assertions and violate

skip	:	$\dfrac{}{\langle \texttt{skip},\sigma\rangle\downarrow\sigma}$
assign	:	$\dfrac{\langle a,\sigma\rangle\downarrow n}{\langle X{:=}a,\sigma\rangle\downarrow\sigma[X\mapsto n]}$
seq	:	$\dfrac{\langle s_0,\sigma\rangle\downarrow\sigma'' \quad \langle s_1,\sigma''\rangle\downarrow\sigma'}{\langle s_0;s_1,\sigma\rangle\downarrow\sigma'}$
if-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-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-false	:	$\dfrac{\langle b,\sigma\rangle\downarrow\texttt{false}}{\langle \texttt{while}\ b\ \texttt{do}\ s_0\rangle\downarrow\sigma}$
while-true	:	$\dfrac{\langle b,\sigma\rangle\downarrow\texttt{true} \quad \langle s_0,\sigma\rangle\downarrow\sigma'' \quad \langle \texttt{while}\ b\ \texttt{do}\ s_0,\sigma''\rangle\downarrow\sigma'}{\langle \texttt{while}\ b\ \texttt{do}\ s_0\rangle\downarrow\sigma'}$

Table 2: Semantics for the *while* language.

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

Figure 1: Example program `mult.ifc`

Table 1 shows the grammar of IFC. In essence an IFC program is a statement with syntax in the C-family. The language is small, as the focus in this project has been to correctly being able to generate verification conditions for said programs. In Table 2 we show the semantics of the different statements.
Arithmetic expressions (`aexpr`) follow the standard rules of precedence and associativity:

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

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

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

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

2.2 Hoare Logic

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

2.2.1 Assertions

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

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

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

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

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

Figure 2: Example program `mult.ifc`

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

2.2.2 Axiomatic system for partial correctness

The Hoare logic specifies am inference system for the partial correctness assertions, that show the axiomatic semantics for the different constructs of the language. The axiomatic system is shown in Table 3.
This axiomatic system shows how assertions are evaluated in the Hoare logic. For example, for an assignment we can say that if we bind x to the evaluated value of a in the initial state, and if P holds in this state, then after assigning x to a, P must still hold. For the *skip* command we see that the assertion must hold both before and after, as *skip* does nothing. The axiomatic semantics for an assertion around a sequence of statements $\{P\}S_1; S_2\{R\}$ state that if P hold in the initial state, and executing S_1 in this state produces a new state in which Q holds, and if executing S_2 in this new state produces a state in which R holds, then $\{P\}S_1; S_2\{R\}$ holds. Another interesting construct is the *while*-loop, especially for our small *while*-language. Here P denotes the loop invariant, and b is the loop condition. If b evaluates to *true* and P holds in the initial environment, and executing S in this environment produces a new state in which P holds, then we know that after the *while*-loop has terminated we must have a state where P holds and where b evaluates to $false$.

2.2.3 Total correctness

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

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

Table 3: Axiomatic system for partial correctness.

line 8 of the code (see code listing Figure 2). To express the logic of using a variant for *while*-loops, we use the following modified syntax.

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

where v is the variant of the loop, and ξ is a fresh logical variable. $\prec$ is a well-founded relation, and because the data type consists of all unbounded integers, we use the well-founded relation:

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

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

2.3 Verification Condition Generation

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

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

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

2.3.1 Weakest Liberal Precondition

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

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

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

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

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

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

2.3.2 Weakest Precondition

Now the weakest liberal precondition does not prove termination. If we want to prove termination in addition to the partial correctness obtained from *wlp*, we need a *weakest precondition* which is much like *wlp*, but require that *while*-loops have a loop variant. Here we have to use the modified Hoare logic for *while*-loops presented in 2.2.3.

The structure for computing the weakest preconditions for the constructs for total correctness is much like the one for computing weakest liberal precondition, except for the structure of *while*-loops, which can be seen below.

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

We see that the difference between the computation of *wp* and *wlp* is the presence of the *variant*. When given a variant expression v for the loop, we add the assertion that this expression decreases with each iteration, using the logical variable ξ to keep the old value of v to compare with.
For our example program `mult.ifc`, we can compute the weakest precondition, as we have both a variant and an invariant in the while loop. By applying the *wp* rules on the example, we get the weakest precondition seen in Figure 3.

```
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 3: 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 forall statement over all the variables altered in the loop bode, and that is what happens in line 5 of the formula. Inside this forall we now check that the loop invariant and condition holds when entering each iteration (lines 6-9), and that the invariant and the negated loop condition holds when the loop terminates(line 10). The variant is used in line 6, where the logical variable ξ is set to hold the value of the variant, and in line 9, where it is checked that the variant has decreased by comparing to the logical variable holding the value that the variant had at the beginning of the loop.

2.4 SAT-solvers

A SAT solver is a program that can determine whether a boolean formula is satisfiable. The advantage of this is that is becomes possible to automatically verify whether very large logical formulae are satisfiable. When given a boolean

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

3 Implementation

IFC is yet to be more than just a toy-language, however in the design of the language and the actual compiler, we have tried to focus making the code modular and easy to extend. Currently the program consists of 4 parts.

1. Parser
2. Evaluator
3. Verification Condition Generator
4. External SMT-solver API

Each part will carry out a task for the program. There is no explicit dependency between any of these parts. Though the main interface is setup to the following tasks:

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

section 6 explains how each of these are used. In the following section we describe how each the 4 program parts are implemented.

3.1 Parser

For the parsing stage of the compiler we use the parser combinator library MegaParsec. Most of the parsing is fairly standard, however the following is noticable.

In the previous sections we described the syntax and the semantics of IFC, in that section we describe only a few of the semantics and state some operators which is simply syntactic sugar for the ones presented in the semantics. However, for certain parts of the actual AST, we use the unsugared constructs, this is mainly seen in the first order logic, in terms of the `exists` and implication. This is purely, because it reads better in the output of the vc generator. This at least has made the development process easier. Ideally this could be alleviated by a more comprehensible pretty-printer, but at the current moment, we settle for a slightly bigger AST, and as of now this does not add much overhead to the other parts of the program, although this might be rethought in the future.

Another point that is worth mentioning is how we handle illegal uses of ghost variables. An illegal use of a ghost variable is a semantical, that is a ghost variable can be declared and occur in the assertion language, but never appear elsewhere in the program logic. However, we have gone with an approach which will simply fail to parse if a ghost variable appears anywhere not allowed. We do so by adding a reader monadtransformer, to the parser type.

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

The boolean value in this environment will tell if the next parser (by the use of `local`) must allow for parsing ghost variables. Which will certainly only happen in assertions. We find that eliminating illegal usecases for ghosts in the parser is far preferable than doing so in both the VC-generator and the evaluator, however this restricts us from generating ghost variables in our Quickcheck generation of statements.

3.2 Evaluator

The evaluator follows directly from the semantics presented in subsection 2.1. That is the intial store provided to the evalutor will be modified over the course of the program according to the semantics. As of now the type of the evaluator is quite bloated compared to what is actually used. 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 reader, 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 there is no real use of the writer monad yet. Again this is for proofing for a future where the language supports some sort of IO, and atleast being able to print would be nice. The store described in subsection 2.1 is kept in the State. The Store is simply a map from VName to Integers. We want the store to be a State as the store after one monadic action should be chained with the next monadic action. This ensures that all variables are in scope for the rest of the program and easily allows for mutable variables. Duely note that ghost variables will also reside in this environment but will not be mutable or even callable as previously explained. We make use of the error monad to resolve any runtime-errors that would arise, that is if a violation is done, a ghost is assigned, a variable is used before it is defined or if undefined behaviour arises such as division by 0.

Each non-terminal in the grammar are resolved by different functions which all operate under the *Eval* monad, which allows for a clean and modular monadic compiler.

One important note about the interpreter is that we have no good way of checking assertions which includes quantifiers. The reasons is that we wanted (possibly erronously) to support arbitrary precision integers. This entails that we currently has no feasible way to check such assertions. A potential solution would be to generate a symbolic reprensentation of the formula and try to satisfy it by using an external prover. Though the interpreter would then also require external dependencies, and not be a standalone program any longer. All in all, this is unideal, and currently the approach is to “ignore” such assertions by considering them true. In **??** we describe a potential extension to IFC, which could help alleviate this problem. Non-qunatified assertions, will still be evauluated as per the operational semantics.

3.3 Verification Condition Generator

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

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

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

As described in subsection 2.3, we use the quantified rule for assignment, when encountering assignments. In the development process we considered two different approaches to resolve this.

3.3.1 First approach

The initial approach tried to resolve Q only after the entire formula were build. This would allows for WLP to be resolve in only two passes, one over the imperative language AST and one over Assertion Language AST in the formula. The approach was intended to build up a map as such, where `VName` is an identifier (String):

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

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

```
missing
```

The result of running WP will then give a partially resolved formula and the final state. When resolving ... Jeg tænker lige.

3.3.2 Second approach

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

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

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

This approach is quite inefficient since it will go over the entire formula every time an assignment is made. Hence why we considered the other approach initially.

Because of their uniqueness ghost variables on the other hand is straight forward to resolve as we need no substitution.

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

3.3.3 While - invariants and variants

The `while` construct is the most complicated of the constructs. As previously mentioned, for partial correctness, we need atleast an invariant, and for full correctness also a variant. Hereby we enfore the user to provide as a minimum the invariant. The code for while is thus also quite complex compared to the rest:

```
wp (While _b [] _var _s) _q = -- ERROR
wp (While b inv var s) q = do
  let invs = foldr1 (./\.) inv   -- Foldr all invariants
  st <- get
  -- Give back condition for unbounded integer variant
  (fa, var', veq) <- maybe (return
                             (id,
                             Cond $ BoolConst True,
                             Cond $ BoolConst True)) resolveVar var
  -- wp(s, I /\ invariant condition)
  w <- wp s (invs ./\. var')
  -- check which variables we wanna forall over
  fas <- findVars s []
  let inner = fa (((Cond b ./\. invs ./\. veq) .=>. w)
                           ./\. ((anegate (Cond b) ./\. invs) .=>. q))
  --- Fix the bound variables and resolve them
  env <- ask
  let env' = foldr (\\(x,y) a -> M.insert x y a) env fas
  inner' <- local (const env') $ resolveQ1 inner
  let fas' = foldr (Forall . snd) inner' fas
  return $ invs ./\. fas'
  where
    resolveVar :: Variant -> WP (FOL -> FOL, FOL, FOL)
    resolveVar var = do
      x <- genVar "variant"
      return (Forall x,
              Cond (Negate (RBinary Greater (IntConst 0) (Var x)))
               ./\. Cond (RBinary Less var (Var x))
             , Cond (RBinary Eq (Var x) var)
             )
```

If no invariant is provided an error is reported. As mentioned in subsection 2.1 we allow for syntactically providing multiple invariants, which should then all hold under conjuction. We have tried to make the code generic in terms existence of the variant to eliminate code duplication. line 6-9, will generate the conditions needed for the variant. In case no variant is defined we use that predicate-logic and conjuction forms a monoid with $\top$ as identity element, such that, we generate no new $\forall$-quantification, and have subformular $b \Rightarrow WP(s, b)$. Is there a variant, we generate the equality $\xi = v$, along with the well-founded relation for unbounded integers. This approach should translate pretty well, with possible other types that have a well-founded relation. The rest of the code simply checks which variables are assigned in the body of the while-loop, and generate a variable for each. Collectively this code will generate the weakest precondition for a while statement as per described in equation 1.

3.4 proof-assistant API

The proof-assistant API uses the SMT Based Verification library (SBV), which tries to simplify symbolic programming in Haskell. The library is quite generic and extensive compared to what we use it for. Again this can be good in the future, but is also a big dependency. We mostly make use of the higher level functions and dont mess with any of the internals, however the default type of SBV does not quite fit our needs. Instead we use the provided transformer, such that we can embed the Except monad. We want to do so, as when iterating over the formular, we might encounter a variables not yet defined, fail gracefully. The code which generates a `Predicate ~ Sym SBool` is simple, since the formula generated in the previous stage, will be already first order logic formula, which straight forwardly can be converted into SBV's types. For the entire highlevel logic we resolve it as simple as this:

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

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

And equally easy it is to resolve `bexpr` and `aexpr`. Ideally we would add a ReaderT to the transformerstack to get rid of the explicit state, however we have not been able to resolve the type for this. Notice how this simply will generate a single collective predicate, instead of the often more used DSL like use of monadically generating quantified variables and constraints. The generated predicate will then try to be proved by the SBV function `prove`. If the program can correctly be proved by the external SMT solver, the output will be `Q.E.D.` , whereas if the formula cannot be proved, a falsifiable instance of the variables will be presented. For instance the output of the following program will obviously always be falsified.

```
violate;
```

whereas the multiplication program in Figure 2 is proveable.

In the current state, there is only possibility of using Z3 as prover, however allowing for the use of any of the other SBV-supported SMT-solvers should quick to implement, whereas non-supported solvers will require quite a lot more work.

3.5 Interface for proofs

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

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

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

```
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
```

It is possible for why3 to generate a vector of input variables and then a precondition for each the `requires`, such that $\forall x_1 ..., x_n.\ requires \Rightarrow WP(body, ensures)$. That is whenever the requires holds, then the weakest precondition of the body should hold, where ensures states the postcondition. Notice that our setup is very similar to this, but since we dont have any return values, we dont really need the ensures, since this might as well be part of the actual program.[1]

4 Results

In this section, we will present our assessment of how succesful we have been in implementing an interpreter and a verification condition generator for IFC. We argue so based on our different tests. Furthermore we present a list of most pressing extensions.

4.1 testing

We have conducted a smaller variety of tests. The tests includes both some testing of properties thorugh QuickCheck along with some examples program. The examples can be found in the `examples` folder. Of these some works correctly and some intentionally dont. In the table below, each program can be seen along with a short description on how and why they are included.

program[2]	description	meaning
always_wrong	This program is simply a `violate` statement and should thus always fail.	This is simply to check that violate always makes the store invalid. This Likewise can never be verified.

4.1.1 Quickchecking instances

To complement the example programs our test suite include a couple of property -based tests. Mostly these tests ensure that the evaluator follows the semantics. To enable this, we define an Arbitrary instance on our AST types. Most of these are quite generic, however there are certain considerations that is quite important. Firstly, want the number of possible variables to be limited, such that the evaluator will not fail too often, by using variables that have not yet been defined. Secondly, while-loops might not terminate, hence we want to define a small subset (skeletons) for while-loops which indeed will terminate. We define 3, versions as follows:

```
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
```

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

orker ikke lige.

We generate an arbitrary variable, which we ensure will never clash with any of the variables that are not possible to generate elsewhere. The constructs will then be one of the following:

- Bolean condition is $v > a$, where v is the variable and a is an arbitrary arithmetic expression. Inside the while-loop, we will ensure to decrement v, eventually terminating the loop.
- Bolean condition is $v < a$. Inside the while-loop, we will ensure to increment v, eventually terminating the loop.
- Bolean condition is $v = a$. Inside the while-loop, we will ensure to change v, eventually terminating the loop.

By this arbitrary value we can quickcheck the following properties, following directly from the big-step semantics:

4.1.2 Dynamic execution compared to static proofs.

Another important property on the relation between the dynamic execution of a program via the evaluator and the static proof of said program, is that if we can correctly prove the correctness of a program, then the dynamic evaluation should also hold. It is important to note that the implication does not hold in the other direction. The reason for this is that we might not have provided strong enough assertions to satisfy the generated formula, whilst the dynamic execution, might not need it to be correct.
If we take a closer look at the multiplication example previously.

```
vars: [q,r]
requirements: {q >= 0 /\ r >= 0}
<!=_=!>
res := 0;
$a := q;
while (q > 0) ?{res = ($a - q) * r /\ q >= 0} !{q} {
      res := res + r;
      q := q - 1;
};
#{res = $a * r};
```

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. Instead of the loop-invariant `?{res = ($a - q) * r /\ q >= 0}` consider the invariant `?{res = ($a - q) * r}` in this case Z3 will no longer be able to prove the correctness of said program. The generated formula looks as follows:

```
$\forall q, r.\ (q \geq 0 \land r \geq 0) \Rightarrow$
  $\forall res_3.\ res_3 = 0 \Rightarrow$
    $\forall \$a.\ \$a = q \Rightarrow$
      $(res_3 = (\$a - q) * r \land q \geq 0)$
      $\land \forall q_2, res_2, \xi_1.$
        $(q_2 > 0 \land res_2 = (\$a - q_2) * r \land \xi_1 = q_2) \Rightarrow$
          $\forall res_1.\ res_1 = res_2 + r \Rightarrow$
            $\forall q_1.\ q_1 = q_2 - 1 \Rightarrow$
              $(res_1 = (\$a - q_1) * r \land 0 \Leftarrow \xi_1 \land q_1 < \xi_1)$
      $\land((q_2 \leq 0 \land res_2 = (\$a - q_2) * r) \Rightarrow res_2 = \$a * r)$
```

It becomes quite apparant that the invariant is no longer strong enough to be proved, since the restriction on q_2 is too weak. The two first conjuctions line 4 and line 5-9, will be true, given that $q_2 = -3, res_2 = 6, \$a = 0, r = 2$. In the last term $(q_2 \leq 0 \wedge res_2 = (\$a - q_2) * r) \Rightarrow res_2 = \$a * r$, the RHS of the implication, will be true as $(-3 \leq 0 \wedge 6 = 3 * 2)$, giving a false term. Once again we can verify that our program acts correctly, by doing the same modification to the whyML program in **??**. In this case the proof will as expected give a falsifiable counter-example. By this, we have a good certainty that our implementation is correct.

We test that whenever we can prove the correctness of an IFC program, the result will be correct. But ideally we would also want to be able to generate programs and test this using QuickCheck. However this 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. As of yet, we have not been succesful in implementing such a property-test.

Wrap up

From the result presented in this Section, we have a fairly good solution, and our program actually works as intended on the example programs. In fact most of the problems that still persists in the code is related to the tests, which could be of better quality. Despite this the Quickcheck tests have been useful in finding subtle bugs.

5 Discussion

5.1 Future work

5.1.1 Procedures and arrays

5.1.2 Type system

5.1.3 PER-logic for Information Flow Control

6 How To Use