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

1 Introduction

This is clearly, in cursive

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. In this section we will present the syntax and semantics of the imperative language and the assertion language.
Vi vil gerne give en kort præsentation. og hvorfor det er interessant. eksempel? mult? skriv noget om det.

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.
For expressing this 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.
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.
As an example of this we will show how our example project mult.ifc looks when adding some assertions.

```
1 q := 10;
2 r := 55;
3 #{q >= 0 /\ r >= 0};
4 res := 0;
5 ghosta := q;
6 while (q > 0) {
7       res := res + r;
8       q := q - 1;
9 };
```

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

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

⟨variant⟩ ::= '![' ⟨aexpr⟩ '}' | ϵ

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

⟨assertionop⟩ ::= '/\' | '\/' | '=>'

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

⟨bop⟩ ::= '&&' | '||'

⟨rop⟩ ::= '<' | '<=' | '=' | '/=' | '>' | '>='

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

⟨aop⟩ ::= '+' | '−' | '∗' | '/' | '%'

⟨ghostid⟩ ::= ghost⟨string⟩

⟨id⟩ ::= ⟨string⟩

Table 1: Grammar of IFC

```
10 #{res = ghosta * r };
```

Listing 1: Example program mult.ifc

In line 3 we have an assertion saying that both q and r must be non-negative,

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.
Most programs written in our small while-language will trivially terminate, only when using the while construct are we concerned with termination. To prove termination of such a loop, one needs one or more loop invariants and possibly a variant.
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 2.
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 $false$.

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

Table 2: Axiomatic system for partial correctness.

2.3 Verification Condition Generation

In the previous section we presented Hoare Triples and how they can be used to "assign meaning to programas". 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.
Below the structure for computing the weakest liberal precondition for the different constructs is shown.

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

Modified syntax for while-loop, in order to prove termination using weakest precondition.

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

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. This makes the difference between partial and total correctness.
The loop variant is an expression that decreases, for example in the while-loop form our example program decreases q in each iteration, as can be seen in line 8 of the code (see code listing 2.2).
Now 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.

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

2.4 SAT-solvers

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 (TODO!!!! REALLY NEED THIS)
3. Extract the formula by Verification Condition Generator
4. Run the formular through an external prover (Z3).

Section ?? explains 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.

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

The boolean value in this environment will tell if the next parser 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.

3.2 Evaluator

The evaluator follows directly from the semantics presented in section ??. 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 section ?? 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 qunatifiers. The reasons is that we wanted to support arbitrary precision integers, however this means that we at the moment has no feasible way to handle these. 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. Hereby assertions with quantifiers do nothing at the moment, while non-qunatified assertions, will be evauluated as per the operational semantics (At the moment they do nothing).

3.3 Verification Condition Generator

The verification condition generator, uses the weakest liberal precondition, or weakest precondition (if while contains a variant). Again we want to be able to chain the actions and include a state and reader environment.

```
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 section ??, we use the forall rule, 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 a variable. 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 (simply generating a unique identifier), then we proceed to resolve q with the new environment, such that every occurence of x will be substituted by x'. The Aexpr which x evaluates to should not be resolved in this case, as this will potentially depend on variables not yet encounted.

This approach will go over the entire formula every time an assignment is made, which is quite inefficient, hence why the initial approach seemed better (and potentially still is, if we can make it work).

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. This would easily be fixed by a simple new iteration over the AST of the formula, and checking if any non-ghost variable does not contain a #.

3.3.3 While - invariants and variants

NOTICE WE HAVE A BUG: If two assignments are done inside the while loop we cannot prove it. otherwise it seem to work???
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. Thus we enfore the user to provide as a minimum the invariant. If no invariant is provided an error is reported. If an invariant is provided, we start by folding the (potentially many) invariants together by conjunction. We then handle the variant if any. We do so by either generate a new invariant variable x, and return a Forall x function, if the variant is not present, we return id, for monadic composition. Then compute wlp for the body, and resolve the entire formula as explained in ??.

3.4 proof-assistant API

The proof-assistant API uses the SMT Based Verification library (SBV), which tries to simply 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. The code which generates a Predicate is simple, since the formula generated in the previous stage, will be a 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 SymTable = M.Map VName SInteger

fToS :: FOL -> SymTable -> Predicate
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 = on (liftM2 (.&&)) (`fToS` st) a b
fToS (ADisj a b) st = on (liftM2 (.||)) (`fToS` st) a b
fToS (AImp a b) st = on (liftM2 (.=>)) (`fToS` st) a b
```

And equally easy it is to resolve bexpr and aexpr. 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;
```

on the otherhand the previously shown multiplication example, will be proved. - Why have we taken the approach we have. Multistage generation? - Following is interesting: - quantifers - ghosts - while (invarianter og varianten) - Ideally static and dynamic execution should give equivalent results. - Testing