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

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.

⟨*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*⟩ '}'

⟨*assertion*⟩ ::= 'forall' ⟨*id*⟩ ⟨*assertion*⟩
| 'exists' ⟨*id*⟩ ⟨*assertion*⟩
| '∼' ⟨*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*⟩

Vi vil gerne give en kort præsentation. og hvorfor det er interessant. eksempel? mult? skriv noget om det.

2.2 Hoare Logic

- Beskriv hvad hoare logik er, og hvorfor det er brugbart?

2.3 Verification Condition Generation

- snak om Weakest precondition, i en figur? - Beskrive de vigtige af dem? - gennemgå det i eksemplet - Sound og Complete?

2.4 SAT-solvers

- IDK noget klogt.

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, an example of this is the `exists` in the assertion language. 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, ??? THIS IS GARBAGE.
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.

```
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 () String STEnv (Either String) a
```

$$\text{Assign}\frac{x, \sigma}{conclusion}$$

3.3 Verification Condition Generator

3.4 proof-assistant API

- 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