Project: wlp-based Bounded Symbolic Verification

In this project we will implement a **wlp***-based* bounded symbolic verification *tool for a variation of GCL. Given a program F, in the bounded verification approach, we only verify a finite number of F's program paths. Obviously such an approach is incomplete, but on the other hand it is fully automatic (modulo the decidability of your back-end prover) as there is no need to manually annotate the program with invariants. In the symbolic verification approach we basically convert every program path to a logical formula p, which is then fully verified using e.g. a theorem prover. Since there can be many, or even, infinite number of concrete executions that would traverse the same program path, full verification of program paths offers a much stronger correctness guarantee than traditional testing*[1].
An **overview** of what you are expected to do in this project:

(6.5 pt)	write a basic implementation of a bounded symbolic verification tool (mandatory).
(2 pt)	write a **paper** presenting your tool (mandatory).
(1.5 pt)	there are optional parts that you can do to get more points.

A note on implementation language. You will use Haskell as the implementation language; so you need to be proficient with it, or at least willing to learn it as we go. You will get a Parser for GCL in Haskell; it already defines a data type for representing GCL's abstract syntax trees. In theory you can do this project in C# or Java, if you are willing to work without a parser. And the design pattern to represent abstract syntax tree is also less clean in these OO languages.

Getting your Z3-backend to work may require some effort, so I recommend to address this first. GHC 8.10.7, 9.2.2, 9.2.5 work. With 9.4.7 it works on some setups, but it does **not** work on my setup. Since I have to review your work, I recomment not to use the latter. I also recommend to use ghcup to manage multiple versions of GHC on your machine. Installing haskell binding for Z3 (called `z3`) in a Windows machine may have an issue; there is a doc in the GCL-parser package (see below) about this. The package also contains some examples on how to invoke Z3 from Haskell.

1 GCL

The syntax of the variant of GCL that we will use is defined below. A parser is available (in Haskell) here:

https://github.com/wooshrow/gclparser

Programs have parameters. They do not return a value, but instead they use a single output parameter to pass back the result of their computation. For example, the program below takes `x` and `y` as parameters. The latter is an output parameter. The program affects the caller's value of `y`, but it does not affect the caller's value of `x`, despite the assignment to `x`.

```
P(x:int | y:int){ x:=x+1; y:=y+x }
```

The syntax of programs is shown below.

[1]Do keep in mind that BMC also has its shortcomings; we will discuss them in the lectures.

Program ::= *Name* – the program's name
"("*Parameters* "|" *Parameter*")"
{ *Stmt* } – statement

Parameters ::= zero or more *Parameter* separated by comma
Parameter ::= *Name* : *Type*

Stmt ::= – statement
BasicStmt
| *BasicStmt* ; *Stmt*
| **if** *Expr* **then** { *Stmt* } **else** { *Stmt* }
| **while** *Expr* **do** { *Stmt* }
| **var** *VarDecls* { *Stmt* }

BasicStmt ::= – basic statement
skip
| **assert** *Expr*
| **assume** *Expr*
| *Name* := *Expr* – assignment
| *Name*[*Expr*] := *Expr* – array assignment

VarDecls ::= zero or more *VarDecl* separated by comma
VarDecl ::= – declaration of local variable
Name : *Type*

Expr ::= – expression
Literal
| *Name*
| #*Name* – array size
| *Expr BinaryOp Expr*
| ~ *Expr* – negation
| *Name* "["*Expr*"]"
| "("`forall` *Name* :: *Expr*")"
| "("`exists` *Name* :: *Expr*")"
BinaryOp ::= Let's just have: +, −, ∗, /, &&, ||, ⇒, <, ≤, >, ≥, =

Type ::= *PrimitiveType* | *ArrayType*
PrimitiveType ::= only `int` or `bool`
ArrayType ::= "[]" *PrimitiveType* – only one-dimensional array

Quantified expressions e.g. $(\forall i :: p_i)$ always quantify over the space of `int`. So, $(\forall i :: p_i)$, where p_i is some predicate p with i as a free variable, means: p is true on all **integer** i.

There is no separate syntax for pre- and post-conditions, because we already have **assume** and **assert** in GCL. For example, consider the (faulty) program below with its specification:

```
{* 1<x *}   -- pre-condition
E(x:int | y:int){
  while 0<x do{ x:=x-1 } ;
  y:=x
}
{* y=1 *}   -- post-condtion
```

Such pre- and post-conditions can be embedded inside the program itself, as follows:

```
E(x:int | y:int){
  assume 1<x; --encoding the pre-cond as assume
```

```
  while 0<x do{ x:=x-1 } ;
  y:=x ;
  assert y=1  --encoding the post-cond as assert
}
```

2 PRELIMINARY: SOME BASIC CONCEPTS

Program paths

Assignments, **skip**, **assert**, and **assume** are called the *basic* statements of GCL, whereas **if** and **while** are called *compound statements* because they are made of other statements. The above GCL has two guarded constructs: **if** and **while**. These constructs always have two branches. The branches of **if** are its **then** and **else** statements. The branches of **while** are its body and the next statement after the **while**. Consider a guard g that guards two branches S_1 and S_2; one is taken if g is true, and otherwise the other is taken. In the first case, we say that the *branch-condition* of the branch is g, whereas the other has $\neg g$ as its branch condition.

Let F be a GCL program. A *program path* of F is a finite sequence of basic statements and branch-conditions in F representing terminating executions that follow the same control path. As examples, below are all the program paths of the program E (from Section 1, the version with pre- and post-conditions embedded) of length ≤ 8. The branch-conditions are colored red.

- A program path of length 4, let's call it $\rho_{(E,4)}$; it corresponds to executions of E where its loop immediately terminates:

  $\rho_{(E,4)}$: **assume** 1<x ; ¬0<x ; y := x ; **assert** y=1

  This path is actually *unfeasible*: there is no input for E that satisfies its pre-condition and will trigger an execution along this path.
- A program path of length 6, let's call it $\rho_{(E,6)}$; it corresponds to the execution of E where its loop iterates once:

  $$\rho_{(E,6)} : \begin{cases} \textbf{assume } 1<x ; \\ 0<x ;\ x := x-1 ; \\ \neg 0<x ;\ y := x ; \\ \textbf{assert } y=1 \end{cases}$$

  This path is also *unfeasible*.
- A program path of length 8; it corresponds to the execution of E where its loop iterates twice:

  $$\rho_{(E,8)} : \begin{cases} \textbf{assume } 1<x ; \\ 0<x ;\ x := x-1 ; \\ 0<x ;\ x := x-1 ; \\ \neg 0<x ;\ y := x ; \\ \textbf{assert } y=1 \end{cases}$$

  This path is feasible.

A program path is called *full* if it reaches the end of the program. All the program paths in the examples above are full paths of the program E, including the unfeasible path $\rho_{(E,6)}$

There are several more things to note:

(1) A program path has finite length. Furthermore, it does not contain any compound construct such as loop. This implies that the correctness of a single path can be automatically verified, assuming the decidability of GCL's expressions.
(2) There may be multiple, even infinitely many, concrete executions that traverse the same program path. So, full verification a single path gives you a much stronger result than traditional testing of the path.
(3) Some program paths may be unfeasible. Putting effort to verify them is thus wasteful. On the other hand, avoiding them requires you to know which paths are unfeasible.

Given a program path σ, we notice that the correctness (validity) of the program path is actually equivalent with the validity of its **wlp** over **true**. To show this, let us first pretend that we transform the program path into a (linear) statement by converting all branch-conditions c in the path into `assume` c. For example, the previous example paths $\sigma_{(E,4)}$ and $\sigma_{(E,8)}$ can be seen as the following linear statements:

- The statement representation of $\rho_{(E,4)}$:

  assume 1<x ; **assume** ¬0<x ; y := x ; **assert** y=1

  You can now calculate its **wlp** over **true**, resulting in the formula:

  $$1<x \Rightarrow (\neg(0<x) \Rightarrow x=1) \quad (1)$$

  The good news is that the above formula is valid (true for all values of x). So, the path is correct. Unfortunately, in this case this is not a very useful result. The formula ¬0<x corresponds to the first branch condition of the path (well, it only has one branch condition). Its conjunction with the pre-condition: 1<x && ¬0<x is unsatisfiable, implying that the path is unfeasible.
  Any unfeasible path is trivially valid; so verifying it is waste of effort.
- The statement representation of $\rho_{(E,8)}$:

  ```
  assume 1<x ;
  assume 0<x ; x := x−1 ;
  assume 0<x ; x := x−1 ;
  assume ¬0<x ; y := x ;
  assert y=1
  ```

  The **wlp** of the above statement over **true** is:

  $$1<x \Rightarrow (0<x \Rightarrow (0<x-1 \Rightarrow (\neg(0<x-1-1) \Rightarrow x-1-1 = 1))) \quad (2)$$

  Let me simplify this to make discussion easier; it is equivalent to:

  $$1<x \wedge 0<x \wedge 1<x \wedge x\leq 2 \;\Rightarrow\; x=3$$

  This formula is *not valid* (the left hand side of ⇒ implies that x = 2, instead of 3). This means the path is incorrect!
  If the post-condition was y=0, this would result in this formula as the **wlp**, after simplification:

  $$1<x \wedge 0<x \wedge 1<x \wedge x\leq 2 \;\Rightarrow\; x=2$$

  This one is valid. So, the same path, but with the post-condition x = 2 is a correct path.

Feasibility of program paths

The **wlp** of an unfeasible program path is always valid. So, as we already remarked, putting effort on verifying unfeasible paths is wasteful. However, deciding if a program path is feasible is also not trivial; it will take some effort.

A program path σ is feasible if there exists a combination of input values for the program, satisfying the program pre-condition, that would trigger an execution that satisfies every branch condition along σ. In other words, every branch condition in σ has to be feasible as well.

The feasibility of a branch condition g in σ can be checked by first calculating its **wlp**. That is, we look at the prefix σ_0 of σ that ends in g, and calculate the **wlp** over this prefix. However, the **wlp** over **assume** is calculated *conjunctively*. So:

$$\textbf{wlp}\ (\textbf{assume}\ p)\ q\ =\ p \wedge q$$

In other words, we replace **assume** with **assert**.

Since in feasibility checking we are not really interested in knowing whether the assertions in the program would hold (as said, it is about 'feasibility'), the **assert**s that are originally in the program path can be ignored. In fact, they must be ignored, or else we might end up excluding an invalid path where all instances of concrete executions through the path lead to an assertion violation, but falsely believeing that it is unfeasible.

For example, the three branch conditions in the path $\rho_{(E,8)}$ result in the following three conjunctive **wlp**:

$$\begin{array}{lcl} B_0 & : & 1<x \wedge 0<x \\ B_1 & : & 1<x \wedge 0<x \wedge 1<x \\ B_2 & : & 1<x \wedge 0<x \wedge 1<x \wedge x\leq 2 \end{array}$$

A branch condition is satisfiable if its conjunctive **wlp** is *satisfiable* (there exists instances of its free variables that would make the formula true). All three formulas above are satisfiable, and hence all three branch conditions in $\rho_{(E,8)}$ are feasible. Hence, $\rho_{(E,8)}$ is feasible.

To immediately verify $\rho_{(E,8)}$ we have to verify (2). Your back-end theorem prover will have to expend some CPU cycles to do this, which can potentially be pretty expensive if the path is long and the post-condition is complex.

If we first check its feasibility, we have to check the satisfiability of all the B_i above. If they all turn out to be satisfiable, we still need to verify (2). So we end up using more computation time. If B_0 (the smallest one above) turns out to be unsatisfiable, the whole path is unsatisfiable. There is no need to check other B_i's, nor (2). In this case, you save yourself some computation time.

I will leave it to you to implement your own heuristic.

Symbolic-execution-based Verification

Symbolically verifying a program path using **wlp** as explained above is also called verification by *symbolic execution*. Performing **wlp**-calculation can be thought as executing the path, though backwardly. There is also forward-symbolic execution, e.g. as implemented in the tool Klee. We will stick with **wlp**-based approach, and will briefy discuss the forward approach during the lectures.

If we verify a program by verifying its program paths (up to some depth k) one at a time by symbolically executing them, the verification approach is called 'verification by symbolic execution'. This is what we will do in this project.

In contrast, tools like CBMC and JBMC would unroll the program (we assume it has already annotated with 'asserts') to some depth k, and covert the entire unrolled program to a single formula (e.g. by calculating the **wlp** over the whole unrolled program), which is verified for its validity. This potentially yield a very large formula. Such an approach is called *bounded symbolic model checking* (BSMC). Similarly, after unrolling the program we would obtain a tree-structured program with if-then-else nodes. We can then calculate one single wlp of the tree, and then check if this is implied by the program given pre-condition. This essentially does the same thing as the BSMC approach mentioned above.

There are pros and cons in both approaches; we can discuss them during the lectures. In any case, the cons of the symbolic execution approach is that the number of paths to verify may explode. A possible mitigation ranges from to prioritizing the paths to verify (incomplete, but may improve your probability to find bugs), to allowing a program path to still contain if-then-else statements (this prevents the doubling in the number of paths, at the expense of obtaining a more complex **wlp**). In the literature, this approach is similar to so-called *state merging* (well, in our case it is 'path merging'). Just greedily doing that would turn the approach to BSMC. A heuristic is needed to decide when it is a good idea to merge paths.

Some notes on implementation

(1) A possible, but naive, approach could to first extract the set Π of all program paths up to some maximum depth, and then we filter out those that are infeasible. Then we verify the rest. However, when a program path $\rho_1 + \rho_2$ is infeasible because the prefix ρ_1 in infeasible, note that all extensions of ρ_1 in Π are actually infeasible as well. So, a post-filtering approach, while intuitive, can be very wasteful.
(2) AST is probably not a very convenient structure for generating symbolic executions. You may want to first unroll your AST to something like a tree, where every path in the tree would be a program path. This tree can be built lazily in Haskell. Furthermore, when you are about to unroll a branch from a node in the tree, and discover that the branch is unfeasible, you can then just remove the branch, and effectively prune the whole subtree under that branch.
(3) The most expensive part of your verifier is likely the calls to the back-end theorem prover. This is a well known phenomenon in symbolic execution based approaches. Writing a front-end simplifier pays off, as it can help e.g. in quickly discovering that a certain branch is unfeasible, without invoking the theorem prover.

3 YOUR ASSIGNMENT, THE MANDATORY PARTS

The mandatory parts of your assignment, explained below, require you to implement your own basic bounded, symbolic-execution-based, verification tool for GCL (6.5 pt) and then to write a paper presenting your tool. Please do not under estimate the paper part. It is worth 2 pt; I expect you to deliver a paper, which is more than just a routine lab report.

You can get more points by doing the optional parts, where you will implement additional features, each with its own challenge.

3.1 Base implementation (6.5 pt, mandatory)

Implement your basic bounded verification tool :) You will need a back-end theorem prover for checking the validity of your **wlp**s and the satisfiability of your branch conditions. You can for example use Z3. There is a Haskell binding that will allow you to call Z3 from Haskell; see the course Website for links.

In this base implementation you can assume that you are given input programs that do not throw any exception (e.g. because you try to access an array outside its valid range).

- For the purpose of your study (see below), you should able to turn your heuristics on and off, so that you can compare the performance of your verifier with and without the heuristics.
- Your verifier should be able to accept a parameter $K \geq 0$. Given an input program F, it should then verify all F's *full* program paths of length op to and including K. The *length* of a program path is defined as the number of basic statements in the path (including the added **assume** statements). We focus on the verification of full paths (paths that reach the end of F). So, partial paths (non-full paths) are not required to be verified (actually, we should leave them out to make your results comparable to each other).
  We will have two verdicts. If the verifier finds an incorrect full path, the verifier **should stop** (so, we stop at the first incorrect path) and report a **reject**. It should also print (to the screen or to a file) the incorrect path (so that you can inspect it) along with a concrete input of F that would trigger the incorrect path.
  If the verifier cannot find an incorrect path, it reports an **accept**.
- Your verifier should also report some **basic statistics**:
  (1) Total number of inspected paths, and of these, the number of paths you manage to identify as unfeasible.
  (2) Consumed computation time.
  (3) Total size of the formulas that you have to verify. We'll define *size* of a formula f to be the number of leaves in the AST of the formula (so, it is the number of variables and literals in the formula). For example the formula $1<x \wedge 0<x$ has four leaves, and its size is 4.
- **Study the performance** of your verification tool by systematically running experiments against **the benchmarks in Appendix A**. Try to script your experiments so that you can easily rerun them with different parameters. Make it so that the script also dumps the results to some csv files so that you can import them into a spreadsheet program, or a graph plotting program, so that you can easily visualize the results.
  Do not under estimate this part, as this forms an important part of your paper.

Array assignment issue. Assignments to array elements will give you some challenge. Consider the program path below:

```
1    assume i=1 ; i:=i-1
2    a[i] := 0
3    a[i+1] := 1
4    assert a[i]=0
```

The corresponding **wlp**:

$$\texttt{i=1} \Rightarrow ((\texttt{i-1=i} \rightarrow \texttt{1} \mid (\texttt{i-1=i-1} \rightarrow \texttt{0} \mid \texttt{a[i-1]})) = \texttt{0}) \quad (3)$$

Notice that each assignment to an array element generates an if-then-else expression in the **wlp**. In the above formula the two array assignments end up generating three cases that ultimately your back-end theorem prover must inspect when it tries to prove the validity of the above **wlp**. This can be a problem if you have a loop that contains such an assignment. As you unfold the loop several times, you may end up with a long program path with many array assignments, thus generating many cases that the theorem prover must inspect.

If you look at the process of how the **wlp** is calculated, we notice that the **wlp** at location 3 above is:

$$(\underbrace{\texttt{i=i+1}}_{g} \rightarrow \texttt{1} \mid \texttt{a[i]}) = \texttt{0} \quad (4)$$

At this point we can choose to consider/investigate if we can already reduce this in-then-else expression. There are three cases:

(1) If the guard g is valid, (4) can be reduced to `1=0`.
(2) If $\neg g$ is valid, we can reduce (4) to `a[i]=0`.
(3) Both of the above cases leads to the reduction of the if-then-else expression. However, if neither g nor $\neg g$ is valid[2], (4) cannot be reduced.

In the example above, $\neg g$ is the formula `i` $\neq$ `i+1`, which can easily be filtered to be valid, without even invoking a theorem prover, and hence reduction of (4) to `a[i]=0` can be made with little overhead.

We can now proceed with the calculation of the **wlp**, which then gives the following intermediate **wlp** at location 2:

$$(\texttt{i=i} \rightarrow \texttt{0} \mid \texttt{a[i]}) = \texttt{0} \quad (5)$$

If we now repeat the above reduction scheme, we easily discover that the condition `i = i` is valid at location 2, and hence (5) can be reduced to `0 = 0`.

What we can conclude from the above example is that it is possible to do on-the-fly reduction of if-then-elses expressions generated during the the calculation of the **wlp**. However, such reduction costs computation time, as we need to check the validity of the cases of the guards of those if-then-else's. Depending on the cost of each checking, the effort is wasted if it turns out that most of the guards cannot be reduced. It is hard to say upfront when an attempt for reduction would be well spent.

We leave it to you to decide how you want to address this challenge. As mentioned before, if you implement a front-end simplifier, it can help to perform the above reduction without invoking the theorem prover. There are benchmarks at the end of this document to measure your performance.

Not required. The focus of the project is on verification. Providing a parser is not required (you will get a parser that you can reuse). Implementing a type checker is also not required. You can assume all input programs are type correct.

3.2 Write a paper (2 pt, mandatory)

Write a paper presenting your tool and your evaluation of its performance. The paper **should not** exceed 6 pages excluding references. Write the paper in LaTeX :) using the ACM double column format. You can download the template from https://www.acm.org/publications/proceedings-template.

If you do an optional part, do mention this in the paper, and don't forget to present your solution for the optional part as well.

We expect a neatly written paper. It should include the following parts:

[2] Just a note: since they are negation of each other, if neither are valid, then both of them are satisfiable.

(1) An Abstract.
(2) An "Introduction" section, to introduce the problem(s) that you try to solve, why it matters to solve it, and your paper's contribution (e.g. that the paper presents a prototype solution along with an initial evaluation of its performance). An Introduction section should be accessible by computer scientists in general.
    If you did something smart/special in your bounded verification tool, do mention it in the Introduction, else the reader would not be aware of it. If you do an optional part, you should mention it here too.
(3) A "Related Work" section. I have compiled a list of about 15 papers that you can use as a starting point. You can find this list in /docs of gclparser source code. It should contain links to their pdf.
(4) A presentation of your bounded verification approach. This should target scientists/students in the software engineering field. You do not need to repeat the whole presentation of wlp rules here (unless you deviate from the standard), but it is helpful for the readers if you present some selected rules (after all, not all software engineering scientists are familiar with formal verification).
    If you have implemented something smart in your verification tool, you would want to present it in this section (else the readers, including the reviewers of your work, would not know you did it).
    Do keep in mind that the limit of 6 pages for the paper is strict.
(5) A section presenting your evaluation of the performance of your tool. This should present and discuss the results of your benchmarking against the benchmark in Section A. You are encouraged to present the results visually with graphs (if you are familiar with Python you can use the Pyplot library to produce your graphs).
(6) Conclusion.
(7) List of references.

Additional formatting instruction: use this magic incantation at the start of your LaTeX file to suppress certain things we don't need:

```
\documentclass[sigconf]{acmart}
\settopmatter{printacmref=false}
\renewcommand\footnotetextcopyrightpermission[1]{}
\pagestyle{plain}
\acmConference{Course
    Program Semantic
    \& Verification}{19/20}{}
```

4 OPTIONALS

There are three optional tasks below; two of them of more challenging. They give you more points (but also more risky to take). **Choose one**; you won't have the time to do more.

4.1 Exception handling (challenging, 1.5 pt)

We will now add a bit more realism to your GCL. Expressions can now throw exceptions. There are two kind of exceptions: division by 0, and attempt to read/write to an array outside its range. If during an execution of a program F it encounters an expression and this expression throws an exception, the execution stops. This also means, if either the left side or the right side of an assignment $e_1 := e_2$ throws an exception, the program stops and the assignment itself does not take place.

We also extend GCL with try-catch statements:

$$\textbf{try}\ \{\ S\ \}\ \textbf{catch}(exc)\ \{\ S_h\ \}$$

This structure introduces exc as a local variable of type `int` initialized to 0. The scope of this variable is the entire try-catch statement. The statement will execute the statement S. If S ends normally (without throwing an exception), we are done. If S throws an exception, it breaks off. It sets the value of exc to 1 if the exception was caused by a division by zero, and 2 if the exception was caused by an attempt to access an array outside its valid range. Then, the control jumps to S_h, which acts as the exception handler and executes this handler.

Extend your verification tool so that you can verify GCL programs that may throw exceptions.

Programs with exceptions pose an additional challenge for automated verification. First, it introduces jumps in your execution flows, which makes it more complicated now to drive the generation of your program paths.

Secondly, and this is a more challenging problem, there can be many points in a program where exceptions can be thrown. In fact, in a real programming language like C, Java, etc., every instruction can potentially throw an exception, hence causing a blow up in the number of program paths that have to be inspected,

The problem is more limited in our GCL setup, but you essentially will still have to deal with the same challenges. I leave it to you to come up with your own solution.

4.2 Pointers (challenging, 1.5 pt)

Let's add pointers/references to GCL. To start, we add `ref` as a new primitive type. We can declare a parameter or a local variable to have this type, e.g. as in `u : ref`. This declares `u` as a parameter/-variable of type `ref`. The value of `u` is either **null** or it is a pointer, pointing to a *store* containing an integer. For simplicity, we will only have integer stores.

We add the following new constructs to GCL. Let `u` be a variable of type `ref`:

(1) `u`.**val** is an expression. If `u` is not null, `u`.**val** returns the integer value contained in the store that `u` points to[3].
(2) An expression r is a `ref`-typed if it is either a literal **null** or a variable of type `ref`.
    If r_1 and r_2 are `ref`-typed expression, $r_1 == r_2$ is an expression that returns a boolean. It returns true if both r_1 and r_2 are **null**, or if both are `ref` variables pointing to the same store. Otherwise $r_1 == r_2$ returns false.
(3) Assignment `u` := **new**(e) where e is an `int`-typed expression. First, the assignment creates a fresh store, let's call it o. It then evaluates e. The resulting integer is copied to the store o. Finally, `u` is made to point to o.
    If e evaluates to a concrete integer i, after this assignment the expression `u`.**val** = `i` would be true. Moreover, for any `ref`-type expression r which is not syntactically the same

[3] `u`.**val** corresponds to the dereference operator ∗`u` in C.

as u, r cannot point to the same store as u (since o is a fresh store). So, u == r would be false.

(4) Assignment u := r, where r is an ref-typed expression. This copies the value of r to u. Note that this assignment does **not** copy r's store content to u.
After this assignment, we would have u == r true.

(5) Assignment u.**val** := e, where e is an int-typed expression. If u is not **null**, this assignment copies the value of e to the store pointed by u.
After the assignment, we would have u.**val** = e true. But do note, if $u == v$ holds, then we would also have v.**val** = e after the assignment.

Extend your verification tool so that you can also verify GCL programs with pointers. Try to first formulate the **wlp** rule for each of the new assignment type (there are three), before you fiddle with your implementation. In the paper, I will require you to also present your new **wlp** rules.

You can expect the treatment of the u.**val** := e type of assignment will generate an if-then-else expression during the **wlp** calculation, similar to the phenomenon you saw in the array assignment, and therefore may also trigger the same blow up. In programming languages where the use of pointers is prevalent (e.g. as in Java and C#), this indeed becomes a major issue. Although in your GCL setup the blow up is much more limited, you essentially still have to deal with the same challenge. I again leave it to you to come up with your own solution.

4.3 Loop with invariant, (0.8 pt, optional)

You can treat an `assert` statement directly before a `while`-loop as a candidate invariant provided by the programmer.

If a loop is annotated with a correct invariant, your verification can skip over the loop by just assuming the invariant. Not only that this saves you computation effort, you verification becomes complete as well ('complete' in the sense that when your verifier approves a program, then *all* its program paths are valid).

Extend your testing tool so that it can benefit from such annotations. Note that just because the programmer annotates the invariant of a loop does not mean that the annotated expression is indeed an invariant. So, make sure that you also check the correctness of the annotations (see again the inference rule for while loop).

Note: we can first covert a statement like S; **do** g *body* ; T to:

$$S;\ \textbf{assert}\ I\ ;\ \textbf{var}\ x_1, x_2, ..\ \{\textbf{assume}\ I \wedge \neg g\ ;\ T\}$$

as well as emitting a verification condition that $\{I \wedge g\}$ *body* $\{I\}$ should be valid. The variables $x_1, x_2, ...$ are variables assigned in *body*. By introducing them as new local variables we assume that their value can be anything, but they do satisfy **assume** I.

A APPENDIX: BENCHMARK

The benchmark programs are listed below. Each program F in the benchmark has an experiment parameter N, which should be replaced by a concrete value e.g. 4, depending on the setup of the experiment —this will be elaborated below. This N determines how many times the loop in F needs to iterate before it can terminate, and therefore N affects how deep K should be to have at least one full and feasible path.

The post-conditions of all benchmark programs are valid[4]. There are also invalid*Name*.gcl variants of the benchmark programs with invalid post-conditions. Just for your own testing, you can check if your verifier gives an 'accept' on all benchmark programs, and 'reject' on their invalid-variants. You can use low N for testing/debugging.

We will run two sets of experiments. One to evaluate the completeness (ability to find bug) of your verifier, and one to evaluate the performance of your heuristics.

(1) *Evaluate your verifier's completeness.* For the **invalid**-variant of each benchmark program F_N, measure the *time* needed to find the violation **against increasing** N, namely $N \in [2..10]$. Measure your other statistics too (# inspected paths, # pruned paths, total wlp size, see Section 3.1). You can use your best heuristic setup. Your K should be fixed, at least for every F.

(2) *Evaluate the performance of your heuristics.* Measure the time and other statistics for verifying each benchmark program F_N (the valid variant) for a fixed N, namely N=10, **against increasing depth** K.
Have several setups, e.g. with no heuristic, with all heuristics turned on, and with a selected heuristic on. Run the measurement for each setup so that you can compare how your heuristic performs against the setup with no heuristic enabled.

Gather the measurement data from the experiments, which you then presents in your paper.

A.1 The benchmark programs

(1) The following program checks if x occurs in the array a. This program should be valid. The N in the pre-condition is an experiment parameter. It is a concrete integer constant, ranging over $[2..10]$.
This problem is intended to see how you deal with path explosion. It can force you to generate up to 2^N different paths before you find one that is actually feasible (the program always iterates over the whole a, so paths that try to leave early are unfeasible).

```
memberOf(x:int, a:[]int | found:bool) {
   // N is an experiment parameter
   assume #a>=N  // note the ">="
          && #a>=0
          && (exists k:: 0<=k && k<#a && a[k]=x) ;
   var k:int {
     k := 0 ;
     found := false ;
     while k<#a do {
       if a[k]=x then found := true else skip ;
       k := k+1
     }
   } ;
   assert found && k>=0 && k==#a && k>=N
}
```

(2) The contrived program below checks if x is divisible by N. N is an experiment parameter, to range over $[2..10]$. The problem is intended to see if your verifier can deal with nested loops.

```
divByN(x:int | divisible:bool) {
  // N is an experiment parameter
```

[4] Otherwise we have a bug; do let me know about it!

```
  assume x>1 ;
  var k:int {
    k := 1 ;
    divisible := false ;
    while k<=x && ~divisible do {
      var i:int {
        i := 0 ;
        while i<N do {
          i := i+1
        } ;
      divisible := i*k = x
      } ;
      k := k+1
    }
  } ;
  assert divisible = (exists m:: 0<m && m<=x && m*N = x)
}
```

(3) The following contrived program modifies an array a such that each a[k+1] is at least equal to $a[k]$+1. N is an experiment parameter, to range over [2..10].
This problem is to see how well your verifier handles array assignment. The program will cause the verifier to generate an exponential number of refby expressions, but perhaps your heuristic can reduce them without using the back-end theorem prover.

```
pullUp(step:int , a:[]int | b:[]int|) {
   // N is an experiment parameter
   assume #a>=2 && #a=N && step>0 ;

   if a[0] >= a[1]
      then { a[1] := a[0] + step }
      else { skip } ;

   var k:int {
     k := 1 ;
     while k < #a - 1 do {
       if a[k] >= a[k+1]
         then { a[k+1] := a[k] + step }
         else { skip } ;
       k := k+1
     }
   } ;
   b := a ;
   assert b[N-1] >= b[0] + #b - 1
}
```

(4) For the Exception-Optional (Sec. 4.1). The following program checks if there are two consecutive elements a[i] and a[i+1] such that

$$\frac{2}{\mathsf{a[i]} + \mathsf{a[i+1]}} < 1$$

If so, it returns with z = 2, and else with z = 1. If the division throws an exception, the program returns with z = 0.
The pre-condition that a contains only positive integers is intentionally added. Under this pre-condition the above division will not throw an exception; we'll see if your verifier can exploit this.
N is an experiment parameter, to range over [2..10]. The pre-condition (∀i : 0≤i<N ⇒ a[i]=1) is intentionally added to force your verifier to explore deeper to find feasible paths.

```
find12(a:[]int | z:int) {
   // N is an experiment parameter
   assume #a>0 && #a=N
          && (forall i:: 0<=i && i<#a ==> a[i]>0)
          && (forall i:: 0<=i && i<N  ==> a[i]=1) ;
   var k:int, r:int {
     k := 0 ;
     z := -1 ;
     while k < #a && (z<0 || (z==1)) do {
       z := -1 ;
       try {
          r := a[k] + a[k+1]
       }
       catch(e) {
          if e=2 // array-range-exception
             then { z := 1 }
             else { skip   }
       } ;
       try {
          r := 2/r   // idea: can we first check if r=0 is feasib
       }
       catch(e) {
          if e=1 // division by zero
            then { z := 0 }
            else { skip   }
       } ;
       if z<0 && r < 1
            then { z := 2 }
            else { skip } ;
      k := k+1
     }
    } ;
    assert z>0
}
```

(5) For the Pointer-Optional (Sec. 4.2). The program below returns the smallest element of an array a. Ref-typed variables and creation of new stores are deliberately added to stress your verifier. The assignments to u.val and x.val are also deliberately added for the same reason; they should not affect the post-condition as their pointers cannot point to the same store pointed to by m (but your verifier has to prove this first of course).
N is an experiment parameter, to range over [2..10].

```
min(a:[]int, x:ref, u:ref | m:ref) {
  assume #a>0 && #a=N
         && ~(x == null)
         && (forall i:: 0<=i && i<#a ==> a[i]>0) ;
  var k:int {
     k := 0 ;
     while k<#a do {
        u := new(a[k]) ;
        if k=0 then { m := u } else { skip } ;
        if u.val < m.val then { m:=u }
                         else { u.val := u.val + 1 } ;
        if ~(m == null) then { x.val := m.val + 1 }
                        else { skip } ;
        k := k+1
     }
  } ;
  assert (forall i:: 0<=i && i<#a ==> m.val <= a[i])
}
```

(6) For the Loop-Optional (Sec. 4.3). The following program has does bubble sort. It should be valid. It has a nested loop. Candidate invariants for both its loops are annotated.
N is an experiment parameter, to range over [2..10].

```
bsort( a : []int | b : []int) {
  assume #a>=0 && #a>=N ;
  var k:int {
    k := 0 ;
    // inv of outer loop
```

```
    assert 0<=k && k<=#a
           && (forall i:: 0<=i && i<k ==>
                 (forall j:: i<=j && j<#a
                     ==> a[i]<=a[j])) ;
    while k<#a do {
      var m:int, tmp:int {
         m := #a-1 ;
         // inv of inner loop
         assert 0<=k && k<=#a
                && (forall i:: 0<=i && i<k ==>
                      (forall j:: i<=j && j<#a
                          ==> a[i]<=a[j]))
                && k<=m && m<#a
                && (forall j:: m<=j && j<#a ==> a[m]<=a[j]) ;
         while k<m do {
            if a[m]<a[m-1]
               then {
                  tmp := a[m] ;
                  a[m] := a[m-1] ;
                  a[m-1] := tmp
               }
               else { skip } ;
            m := m-1 }
      } ;
      k := k+1 }
  } ;
  b := a ;
  assert (forall i:: 0<=i && i<#b ==>
            (forall j:: i<=j && j<#b
                ==> b[i]<=b[j]))
}
```

B APPENDIX: ABSTRACTLY REPRESENTING A LANGUAGE IN AN OO LANGUAGE

If you read this section, it means you decided in implement this project in an OO language like C# or Java. Since I don't have a C# nor Java parser for GCL for you, you will have to work without one (there is not enough time for you to implement one yourself). However, we can still represent a program structurally as objects, and I will show you how. In any case, you do not want to work on a plain string representation of programs because it will become very messy for you to identify the structures of the programs.

A structured representation of a program (or more generally, of a sentence satisfying some formal grammar) is also call an *abstract syntax tree*. This section provides some basic on how to represent such a tree in an OO language; I will assume Java as the implementation language.

Let us consider, as an example, a simple language of *Expr* expression, as an example. The syntax is shown below:

$$\begin{array}{rcl} Expr & ::= & IntegerLiteral \\ & | & Name \\ & | & Expr \text{ ”+” } Expr \end{array}$$

So, for example $x+1+y$ is a sentence of *Expr*. I will show you how to implement a simple expression transformer. As an example let's take the substitution operation $q[e/x]$ which is supposed to replaces all occurences of x in the expression q with e. Note our *Expr* has no construct to represent quantified expressions, so substitution will be very simple compared to subsitution in your GCL. But this will do as an example.

In Java, we can use an (abstract) class, let's call it `Expr`, to represent *all sorts* of expression, and a subclass of `Expr` to represent *each sort* of expressions. This is shown below.

```
abstract class Expr { }

class Lit_ extends Expr { // representing int literal
   int val ;
   Lit_(int val) { this.val = val ; }
}

class Name_ extends Expr {
   String name ;
   Name_(String name) { this.name = name ; }
}

class Plus_ extends Expr {
   Expr a ;
   Expr b ;
   Plus_ (Expr a, Expr b) { this.a = a ; this.b = b ; }
}
```

For convenience, we can extend the definition a little bit, as shown below:

```
abstract class Expr {
   Plus_ plus(Expr b) { return new Plus_(this,b) ; }
   static Lit_ lit(int val) { return new Lit_(val) ; }
   static Name_ name(String name) {
          return new Name_(name) ;
   }
}
```

With the new methods, the expression $x+1+y$ can be represented by (with the help of static imports);

```
name("x").plus( lit(1).plus(name("y")))
```

Substitution can be implemented as shown below. There is a nicer way to implement such a function, namely using the so-called "Visitor" design pattern, but you can look this up yourself in the Internet.

```
static Expr subst(Expr q, Expr e, String x) {
  if (q instanceof Lit_) return q ;
  if (q instanceof Name_) {
      if (((Name_) q).name.equals(x)) return e ;
      else return q ;
  }
  Plus_ p = (Plus_) q ;
  return new Plus_(subst(p.a,e,x), subst(p.b,e,x)) ;
}
```