# Static Verification of C0 Programs Using the Z3 Theorem Prover

Matthew McKay, Advised by André Platzer and Frank Pfenning

## Abstract

I have implemented a static verification tool for C0 on top of the current C0 compiler. This is made possible by the contracts that are a part of the C0 language. The tool interfaces with Z3, and uses its SAT-solving capabilities to statically verify that the program adheres to the contracts in every case. It does this by converting the program into SSA form, then traversing the program and making assertions. Using SSA form is crucial, as it allows the verifier to keep around assertions of previous variable generations without having to worry about collision, since each variable is only defined once.
Currently it is successful at finding the following errors:

- Division by zero (and modulo by zero, as well as INT_MIN / or % by -1)
- Integer overflow (not an error, but can detect it)
- Array index out-of-bounds
- Null pointer dereference
- Contract violations

## Verification Conditions Generation

The statement $\Gamma \vdash s \triangleright [\Gamma'; \Delta]$ means that given assertions $\Gamma$, the process of verifying statement s makes new assertions $\Gamma'$ that are known to be true after s is run (e.g. x = 3) with a list of noncritical errors $\Delta$. These errors are satisfiable expressions that could result in an error, given a specific model of the existing variables. Critical errors occur when the code can be proven to have an error on all inputs, which causes verification to halt and return the errors.

## Verification of Conditionals

$$\frac{\Gamma \vdash e \triangleright [\Gamma_1;\Delta_1] \quad \Gamma\wedge\Gamma_1\wedge e \vdash s_1 \triangleright [\Gamma_2;\Delta_2] \quad \Gamma\wedge\Gamma_1\wedge !e \vdash s_2 \triangleright [\Gamma_3;\Delta_3]}{\Gamma \vdash \texttt{if e then s}_1\texttt{ else s}_2 \triangleright [\Gamma_1\wedge((\Gamma_2\wedge e)\vee(\Gamma_3\wedge !e)); \Delta_1,\Delta_2,\Delta_3]}$$

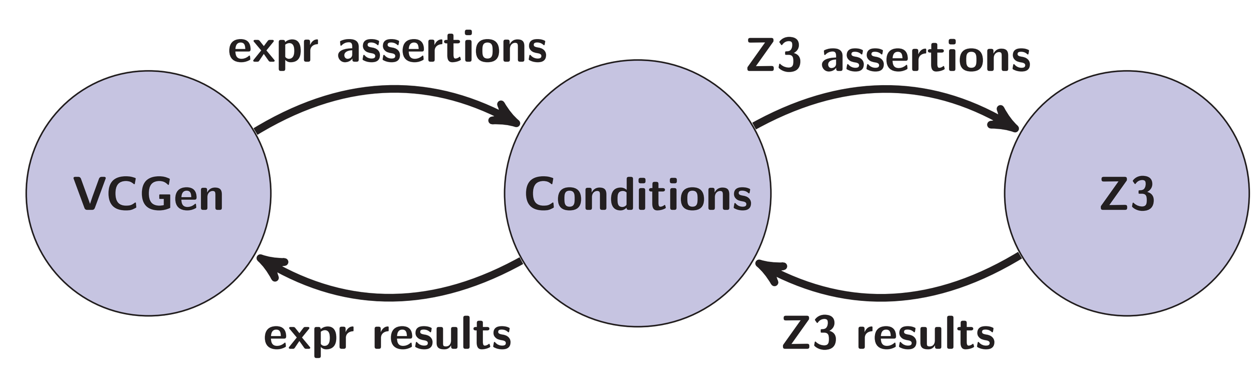
We know that one of the branches must have been taken, so it must be that either the assertions from the then branch or the assertions from the else branch are true.

## Verification of Loops

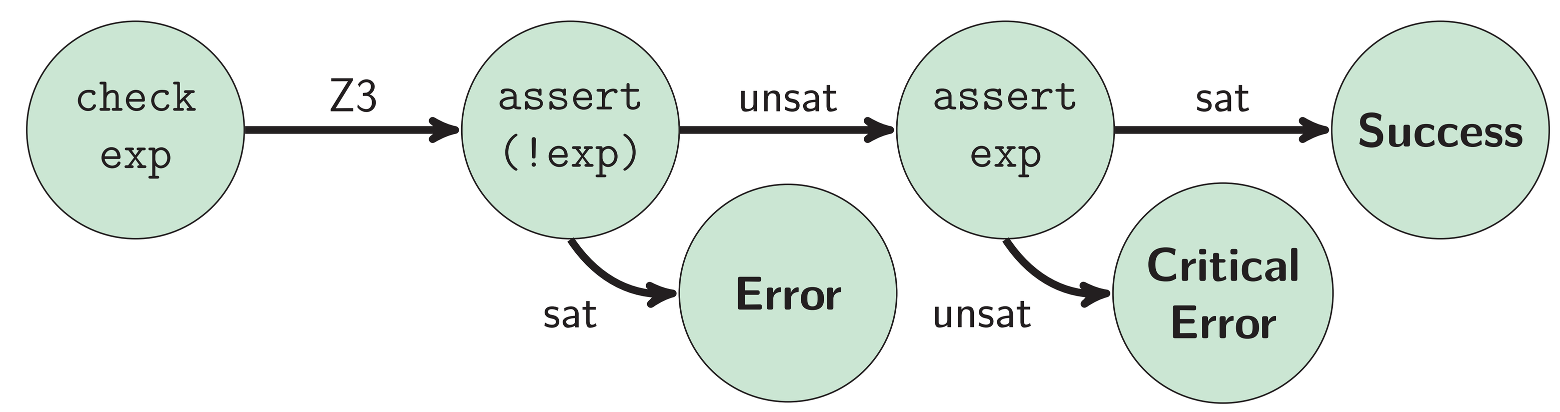
$$\frac{\Gamma \vdash \texttt{invs} \triangleright [\Gamma_1;\Delta_1] \qquad \Gamma\wedge\Gamma_1\wedge\Gamma_2\wedge\texttt{invs}\wedge e \vdash s \triangleright [\Gamma_3;\Delta_3] \qquad \Gamma\wedge\Gamma_1 \vdash e \triangleright [\Gamma_2;\Delta_2] \quad \Gamma\wedge\Gamma_1\wedge\Gamma_2\wedge\Gamma_3\wedge\texttt{invs}\wedge e \vdash \texttt{invs}_{new} \triangleright [\Gamma_4;\Delta_4]}{\Gamma \vdash \texttt{while(e) invs s} \triangleright [\Gamma_1\wedge\texttt{invs}\wedge !e; \Delta_1,\Delta_2,\Delta_3,\Delta_4]}$$

Before verifying the loop we must check that the invariants are true. We must also check that the invariants still hold for the changed variables at the end of the loop (which proves the inductive quality of the invariants). We can keep around all the contexts while in the loop since variable assignments are unique due to SSA. After the loop, only $\Gamma_1$ is related to things outside. Also we only know that the loop invariants are true and loop condition is false if the loop contains no `break` statements.

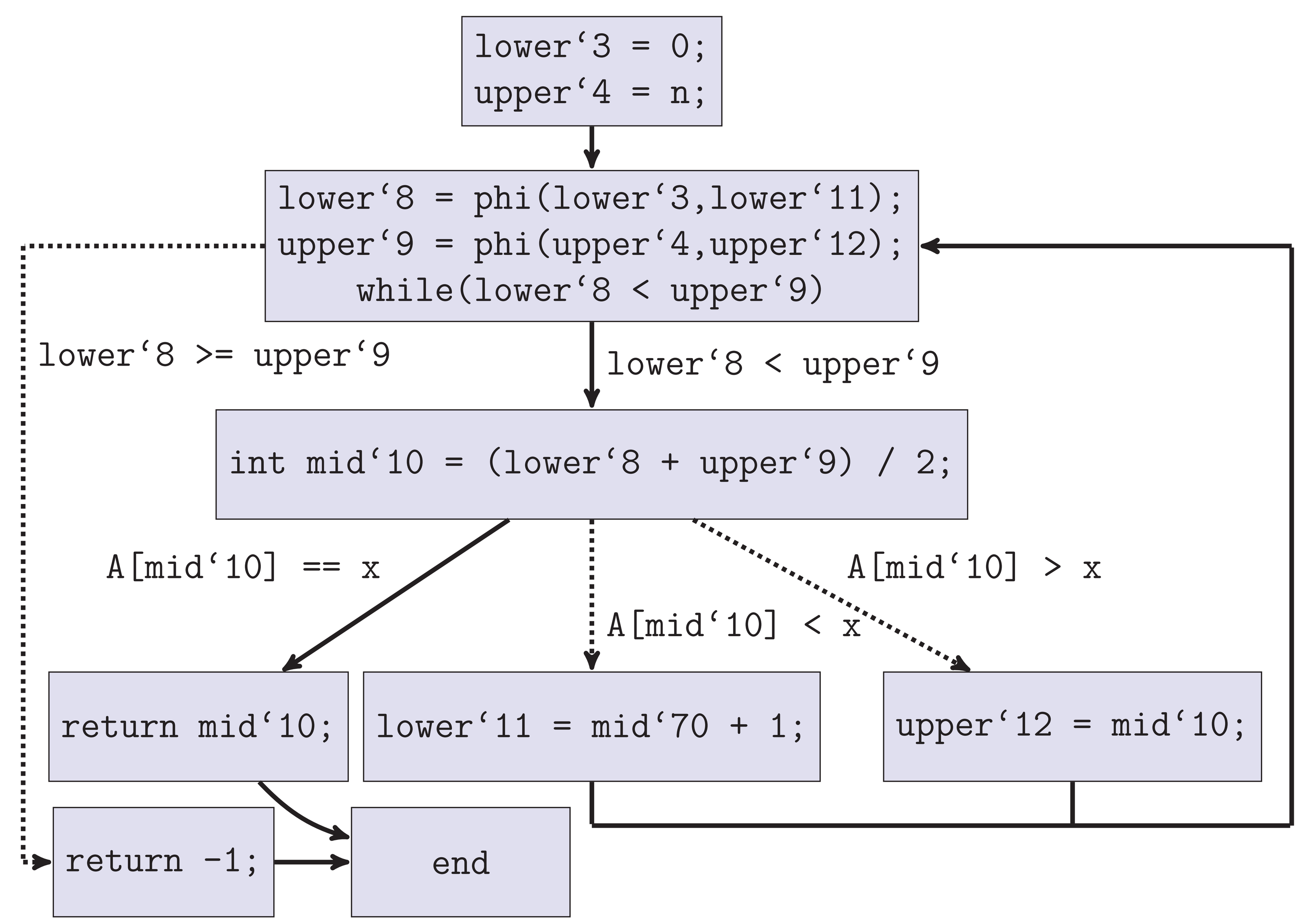
## Assertion Flowchart




## Checking Expressions




## Binary Search Control Flow Graph




## Binary Search (Bad)

```
int binsearch(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@ensures (-1 == \result) || (0 <= \result && \result < n);
{
  int lower = 0; // lower‘3
  int upper = n; // upper‘4
  while (lower < upper) // lower‘8, upper‘9
  //@loop_invariant 0 <= lower && lower <= upper && upper <= n;
  {
    int mid = (lower + upper) / 2; // mid‘10
    //@assert lower <= mid && mid < upper;
    if (A[mid] == x) return mid;
    else if (A[mid] < x) lower = mid+1;
    else upper = mid;
  }
  return -1;
}
```

## Binary Search (Bad) - Output

```
:note:Errors for function binsearch
:error: Error case (mid‘10 < 0x0) is satisfiable with model:
(lower‘8 == 1075021826)
(mid‘10 == ~1071921151)
(n‘2 == 1107296256)
(upper‘4 == 1107296256)
(lower‘3 == 0)
(upper‘9 == 1076103168)
(\length(A‘1) == 1610612736)
```

## Binary Search (Good)

The problem with the above code is that the sum `(lower + upper) / 2` could overflow into the negatives, resulting in an array index out-of-bounds error. To fix this we can replace it with the following:

```
lower + (upper-lower) / 2;
```

Then we get the following output as desired:

```
No verification condition errors could be found.
```

## Code Flowchart

**Source Code**

```
int divide(int x, int y)
//@requires x >= 0 && y > 0;
//@ensures x - \result * y < y;
{
  int r = x;
  int q = 1;
  while(r >= y)
  //@loop_invariant q * y + r == x;
  {
    r -= y;
    q++;
  }
  return q;
}
```

→ **SSA and Isolation** →

**SSA Form**

```
int divide(int x‘0, int y‘1)
//@requires x‘0 >= 0 && y‘1 > 0;
//@ensures x‘0 - \result * y‘1 < y‘1;
{
  r‘2 = x‘0; q‘3 = 1;
  while(r‘4 >= y‘1) {
   r‘4 = phi(r‘2,r‘6); q‘5 = phi(q‘3,q‘7);
   //@loop_invariant q‘5 * y‘1 + r‘4 == x‘0;
  {
    r‘6 = r‘4 - y‘1;
    q‘7 = q‘5 + 1;
  }}
  return q‘5;
}
```

→ **VCGen** →

**VC Assertions**

Initial check of loop invariant

```
assert (x‘0 >= 0 && y‘1 > 0)
assert (r‘2 == x‘0)
assert (q‘3 == 1)
push
check (q‘3 * y‘1 + r‘2 == x‘0)
assert (r‘4 >= y‘1)
assert (q‘5 * y‘1 + r‘4 == x‘0)
assert (r‘6 = r‘4 - y‘1)
assert (q‘7 = q‘5 + 1)
check (q‘7 * y‘1 + r‘6 == x‘0)
pop
assert (!(r‘4 >= y‘1))
assert (q‘5 * y‘1 + r‘4 == x‘0)
check (x‘0 - q‘5 * y‘1 < y‘1)
```

→ **Conditions** →

**Z3 Assertions**

```
...
assert (= r_2 x_0)
assert (= q_3 (_ bv1 32))
push
check (q‘3 * y‘1 + r‘2 == x‘0)
push
assert (not (= (bvadd (bmul q_3 y_1) r_2) x_0))
check-sat
pop
assert (= (bvadd (bmul q_3 y_1) r_2) x_0)
check-sat
assert (bvsge r_4 y_1)
...
pop
...
```