EECS 700 Assignment 1 Design Note: Hoare Logic Prover

Darshil Patel

October 30, 2025

1 Overview

This document details the design and implementation of extensions to a baseline Hoare Logic automatic prover. The prover, written in Python with a Z3 SMT backend, was extended to support two major language features:

1. **Arrays:** Read and write operations, encoded using Z3's built-in theory of maps.
2. **Procedures (with Recursion):** Function definitions and calls, verified using contracts (requires/ensures), havoc-based Weakest Precondition (WP) generation, and specification-based induction for recursion.

The extensions were integrated into the existing `parser.py` (AST generation) and `prover.py` (Verification Condition Generation).

2 Arrays

2.1 Hoare Rule and Weakest Precondition

The primary challenge for arrays is handling assignment. An array a is not a single value but a map. A write operation $a[i] := e$ does not just change $a[i]$; it creates a *new logical array* where the value at index i is e and all other values are unchanged.

The Weakest Precondition (WP) rule for array assignment is based on McCarthy's 'store' function:

$$\mathrm{WP}(a[i] := e, Q) = Q[\mathrm{Store}(a, i, e)/a]$$

In plain terms, the precondition for this assignment is the postcondition Q where every occurrence of the array a is replaced by the new symbolic array `Store(a, i, e)`.

2.2 Z3 Encoding

We leverage Z3's built-in array theory, which directly models this logic.

- **Declaration:** An array variable 'a' is declared in Z3 as 'Array(IntSort(), IntSort())', a map from integer indices to integer values.
- **Array Read ($a[i]$):** A read expression in our AST, `['select', 'a', 'i']`, is translated to the Z3 expression `Select(a, i)`.
- **Array Write ($a[i] = e$):** An assignment statement `['tastore', 'a', 'i', 'e']` is handled directly by the WP generator. It applies the rule above by substituting the Z3 array a with the Z3 expression `Store(a, i, e)` in the postcondition.

This encoding automatically benefits from Z3's built-in axioms for arrays (the "store-select" laws), which were verified by our test suite:

- $\forall a, i, j, v : (i = j) \implies \text{Select}(\text{Store}(a, i, v), j) = v$
- $\forall a, i, j, v : (i \neq j) \implies \text{Select}(\text{Store}(a, i, v), j) = \text{Select}(a, j)$

This allowed `test_array3.py` (aliasing, $i = j$) and `test_array2.py` (non-aliasing, $i \neq j$) to be verified correctly.

3 Procedures and Recursion

3.1 Procedure Definition and Verification

A procedure is verified by proving that its body satisfies its own contract. The VC for a procedure f with body S, requirement R_f, and postcondition E_f is:

$$(R_f \wedge \text{OldVarsInit}) \implies \text{WP}(S', E'_f)$$

Where:

- **Handling old(v):** For every variable v referenced as $\text{old}(v)$ in E_f, a fresh logical variable v_{old} is introduced.
- **OldVarsInit:** Is the set of assumptions $\bigwedge v_{\text{old}} = v$ for all such variables, capturing the state at the function's entry.
- S': Is the body S with any `return e` statement replaced by `ret := e`.
- E'_f: Is the ensures clause E_f with all $\text{old}(v)$ replaced by v_{old} and return replaced by ret.

3.2 Procedure Call Rule

At a call site $x = f(e)$ (with f modifying a set of variables M), the WP logic is more complex. Verification is a two-step process:

1. Precondition Check (VC) The prover must first generate a VC ensuring the call site satisfies the procedure's requirement.

$$\text{Pre}_{\text{call}} \implies R_f[e/\text{formals}]$$

Where Pre_{call} is the current weakest precondition at the call site.

2. Postcondition Assumption (WP Generation) After proving the precondition, the VCGen *assumes* the procedure's postcondition holds. This involves two concepts:

- **Havoc:** All variables $v \in M$ (the 'modifies' list) are "havoced" by substituting them with fresh, unconstrained symbolic variables (e.g., $v_1, v_2, \ldots$).
- **Frame Condition:** All variables $v \notin M$ are constrained to be unchanged: $v_1 = v_{\text{pre-call}}$.

The prover then continues WP generation from a new state defined by:

$$\text{Ensures}' = E_f[e/\text{formals}, x/\text{ret}, v_1/v] \wedge \text{FrameCondition}$$

The final WP for the call is $\forall v_1 \in M : (\text{Ensures}' \implies Q[v_1/v])$. This was successfully demonstrated in `test_proc3.py` (array swap), where the array a was modified but the scalar z was proven to be unchanged by the frame condition.

3.3 Recursion

Recursion is handled not by inlining, but by **specification-based induction**. When verifying a procedure f, any recursive call to f within its body is treated as a standard procedure call. The prover simply assumes its *own specification* holds for the recursive call.

For functions used *in* expressions (e.g., in the 'ensures' clause of `sum_array`), they are modeled as Z3 **uninterpreted functions**. To allow Z3 to reason about them, the function's own specification is added as a universal **axiom** to the solver (e.g., $\forall n : \text{Requires}(n) \implies \text{Ensures}(n, \texttt{sum\_array}(n))$).

4 Counterexample Analysis

A key demonstration of the prover's correctness is its ability to find valid counterexamples. Consider `test_recursive1.py`:

```
def fact(n):
  requires(n >= 0)
  ensures((n == 0 and ret == 1) or (n > 0 and ret >= 1))
  ...

# Main program:
assume(x == 3)
y = fact(x)
assert(y == 6)
```

The prover correctly returns **INCORRECT** with the counterexample `[x = 3]`.

Analysis: The specification for `fact` is *weak*. It only promises that for $n > 0$, the result is $\text{ret} \geq 1$. It *does not* promise $\text{ret} = n!$. At the call site, the prover knows:

1. $x = 3$
2. From `fact(3)`'s ensures: $(3 > 0 \land y \geq 1)$, which simplifies to $y \geq 1$.

It then tries to prove the VC: $(x = 3 \land y \geq 1) \implies y = 6$. This is logically false (e.g., $y = 5$ satisfies the premise but not the conclusion). The prover is correct: the `assert(y == 6)` is not justified by the program's specification. This confirms the procedure call logic is working as intended.