galois

Functional Correctness Proofs in SAW

What is Functional Correctness?

- Functional correctness is concerned with whether the program adheres to its specification
  - In other words, does it compute what it is supposed to
- Specifications can be vague: “Returns a nonzero number”
- Or they can be specific: “Returns the product of the inputs”
- Difference from memory safety proofs: You specify *exactly* what you want SAW to prove
  - Memory safety proofs automatically generate verification conditions for you
  - Caveat: SAW always checks memory safety

Learning Functional Correctness

- Will not learn nearly as many new SAW commands
  - You already know most of what you need to know for functional correctness proofs
- Exercises will largely be adding functional correctness conditions to earlier memory safety proofs
  - This is how real-world proofs go: memory safety first, then functional correctness
- Warning: Functional correctness proofs are harder
  - Expect some exercises to take longer
  - Expect weird error messages
  - We will start easy and work our way up in difficulty
  - Don’t hesitate to ask questions!

Example: Add

```c
uint32_t add(uint32_t x, uint32_t y) { return x + y; }
```

```
let add_spec = do {
    // Create fresh variables for `x` and `y`
    x <- llvm_fresh_var "x" (llvm_int 32);
    y <- llvm_fresh_var "y" (llvm_int 32);

    // Invoke the function with the fresh variables
    llvm_execute_func [llvm_term x, llvm_term y];

    // The function returns a value containing the sum of x and y
    llvm_return (llvm_term {{ x + y }});
};
```

Memory Safety to Functional Correctness

- Many cases look like `add` example
- Preconditions often stay the same
  - Initializing fresh variables
  - Potential difference: Specifying values in global variables
- `llvm_execute_func` arguments often stay the same
- Postconditions are almost always stricter
  - Instead of using llvm_return or llvm_points_to with a fresh var, often use with inline Cryptol
- SAW still checks memory safety conditions in functional correctness proofs
  - One reason why we start with memory safety proofs: to limit the number of things SAW is checking so we have an easier time debugging proofs.

Accessing Cryptol Definitions from SAW

```
import "some_cryptol_file.cry";
```

- Use `import` to load cryptol definitions into SAW
  - Not to be confused with `include` for loading other SAW files
- Interact with loaded definitions via `{{ … }}`
- Ex: "Spec.cry" defines a function `foo`:

```
import "Spec.cry";
let foo_spec = do {
   …
   llvm_return (llvm_term {{ foo x }} );
};
```

Exercise: Popcount

- Complete both parts of the exercise in functional-correctness/popcount/exercise.saw

Demo: Looking at SAW Goals

- Looking at SAW goals can help debug proofs

```
llvm_verify … z3;
```

```
llvm_verify … (do {
    print_goal;
    z3;
});
```

Exercise: u128

- Complete both parts of the exercise in functional-correctness/u128/exercise.saw

Specifying Struct Values

```
llvm_struct_value [<field_0>, …, <field_n>];
```

- Specify struct values with `llvm_struct_values`
- Takes a list of values corresponding to fields in the struct.

Example: person struct

```
struct person { char* name, unsigned int age };
```

Initializing a pointer to the struct in SAW:

```
llvm_points_to
    person_ptr
    (llvm_struct_value [ name_ptr, llvm_term age ]);
```

Exercise: Point

- Look at the C and Cryptol files in functional-correctness/point
- Complete all 3 parts of the exercise in functional-correctness/point/exercise.saw

Exercise: Swap

- Refamiliarize yourself with functional-correctness/swap/swap.c
- Complete parts 1-4 of the exercise in functional-correctness/swap/exercise.saw

Keeping Proof Goal Sizes in Check

- Structure your implementations and proofs to keep goal sizes small
  - Large goals are hard to read, making proofs hard to debug
  - Really large goals are hard on the SMT solver. Proofs may not terminate in a reasonable amount of time, or may run out of memory.
- Keep goal sizes down by making use of composition
  - Prove individual functions, and use generated overrides in proofs of calling functions.
  - Break large functions into smaller functions.
  - Pull loop bodies into separate functions and prove those individually. This can make a huge difference.
- Demo: `selection_sort` proof

Further Restricting Inputs/Outputs

```
llvm_precond {{ <precondition> }};
llvm_postcond {{ <postcondition> }};
```

- Use llvm_precond before llvm_execute_func to add a precondition to the specification.
  - SAW will add this as an assumption to the proof
- Use `llvm_postcond` after `llvm_execute_func` to add a postcondition to the specification
  - SAW will turn this into an additional goal to prove.
- `llvm_precond` often used to restrict inputs to be valid
  - Ex. Index is in bounds: `llvm_precond {{ idx < `len }}`

Exercise: selection_sort Decomposition

- Complete parts 5-6 of the exercise in functional-correctness/swap/exercise.saw

Problem: `argmin` Input Sizes

- We proved `argmin` for len=8, but our composed `selection_sort` calls `argmin` over successively smaller arrays.
    - Therefore, our `argmin` override will fail to match on the second loop iteration when `len=7`.
- Could resolve by creating many overrides manually:

```
argmin_8 <- llvm_verify m “argmin” … (argmin_spec 8);
argmin_7 <- llvm_verify m “argmin” … (argmin_spec 7);
…
argmin_1 <- llvm_verify m “argmin” … (argmin_spec 1);
```

SAW `for` loops

```
argmin_ovs <- for (eval_list {{ [1..a_len] : [a_len][64]}}) (\len ->
    llvm_verify swapmod "argmin" [] true (argmin_spec (eval_int len)) z3
);
```

- `for` <list> <function over list elements>
  - Returns a list containing the result of applying the function to each list element
  - Like `map` in Cryptol
- `eval_list` - Converts a Cryptol list to a SAW list
- `eval_int` - Converts a Cryptol int to a SAW int
- No need to fully understand what's going on here
  - Pattern comes up when dealing with algorithms that repeatedly process an ever-shrinking list
  - Not common in cryptography proofs

Concatenating SAW Lists

```
concat <list1> <list2>;
```

- Use `concat` to concatenate SAW lists
- Useful for combining lists of overrides

Example:

```
concat [1, 2, 3] [4, 5]; ----> [1, 2, 3, 4, 5]
```

Exercise: `selection_sort_composed` proof

- Complete parts 7-8 of the exercise in functional-correctness/swap/exercise.saw

Demo: A nicer goal

- Look at the new goal for `selection_sort_composed`

Uninterpreted Functions

```
w4_unint_z3 [“fn_0”, “fn_1”, …, “fn_n”];
w4_unint_yices [“fn_0”, “fn_1”, …, “fn_n”];
```

- Sometimes the SMT solver can get lost in the weeds.
- Can tell SMT solver to leave certain functions uninterpreted
  - Instructs the solver to only consider argument equality of cryptol functions
- `(fn x) == (fn y)` if and only if `x == y`
  - Ex. if `add` is uninterpreted, then `(add x y) != (add y x)`
- Use when you know arguments to a complex cryptol function are equivalent in proof goal
- Command goes where `z3` or `yices` normally goes

Exercise: wacky_sort

- Complete part 9 of the exercise in functional_correctness/swap/exercise.saw

Review: Best Practices

- Start with a memory safety proof
- Add postconditions one-by-one, ensuring you have a working before proof moving forward
- Prove small functions and use overrides in calling functions
- Beware of endianness
    - Structure Cryptol specs and proofs to avoid manual endianness conversions
- Use `print_goal` and related tactics to debug proofs
- If a proof doesn't terminate, look for loops to refactor and functions to leave uninterpreted.

Questions?