15-411 Lab 6 Report: Interligua

Scott Mionis
Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania
smionis@andrew.cmu.edu

Akshath Jain
Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania
arjain@andrew.cmu.edu

***Abstract*—In this report we summarize the term project completed for 15-411 Lab6. Specifically, we implemented a Formal Verification system utilizing Static Analysis (Hoare Logic[1][2] and Symbolic Execution[3]) to gather information about our program at various program points. We then use these 'facts' to construct boolean expressions representing the correctness of the program, and solve by them querying a 3rd-party SMT[5] solver using the Why3[4] OCaml interface. The goal of this project is to prove, at compile-time, the correctness of programs with respect to the requires and ensures conditions set by the programmer, and additionally indicate when the program is either provably incorrect or not enough information is present to make a conclusion. We will outline the design of our formally-verifiable language (greatly inspired by Dafny [6]) and specify the rules governing how facts are inferred from program statements. We then discuss implementation details, and end by analyzing our results and describing future work to be done.**

***Index Terms*—Compilers, 15-411, Dafny, Formal Verification, Hoare Logic, Static Analysis, Why3, Z3, Alt-Ergo, Theorem Proving, SMT**

Introduction. For Lab6, we implemented a formal verification system to analyze, at compile-time, the correctness of compiled programs for a modified version of the L3 language. The goal of the project is, given a source program annotated by the programmer with loop invariants, requires, and ensures statements, to prove that the source program respects the specification. Specifically, we leverage Hoare Logic and Symbolic Execution to prove that for all functions f:

$$\forall \mathbf{f},\ \mathbf{Requires(f)} \implies \mathbf{Ensures(f)}$$
$$\forall \mathbf{loops} \in \mathbf{f},\ \mathbf{Invariant(loop)}_n \implies \mathbf{Invariant(loop)}_{n+1}$$

(We will continue to use this mathematical notation later on)

We start by modifying the L3 language to accept specifications from the programmer in the form of 'Requires' statements, 'Ensures' statements, and loop invariants (see the project specification section for more details). Once these have been parsed, typechecked, and elaborated, we then feed this program into our verification module.

Our verification module (prover.ml/mli) then performs static analysis and tags each statement in the code with a complete "fact" list representing our knowledge at that program point. Each "fact" is a boolean expression that restricts the value of temporary variables, or an 'Implies' construct representing an implication between two facts or two groups of facts (this is most useful for classifying our knowledge about function return values, because we know that we can only rely on the ensures statement if we satisfy the requires of the function).

Once we have completely analyzed the program, we construct a series of proof tasks that completely summarize the correctness of the function. These proof tasks include, but are not limited to, proving that all knowledge at each function return correctly implies the postcondition, and that each loop invariant implies that the invariant for the next iteration holds.

Once we have summarized the program into these boolean proof statements, we convert them into a Why3 term object and feed them into an SMT solver (Alt-Ergo or Z3) to verify the program (more will be written about these solvers and the Why3 interface later on). If all statements are Valid, the program is correct. If any statement is Invalid, the program is incorrect. If no statements are Invalid but some are inconclusive, the program *could* be correct, but the requires statement(s) is not strong enough to imply the postcondition(s).

Our results are extremely promising. While the downloadable version of Alt-Ergo is prone to unexpected timeouts (despite our program analysis and SMT construction being correct), we can easily format the problems and feed them to the online version of Alt-Ergo at https://alt-ergo.ocamlpro.com/try.html, and in all instances the prover agrees with our module. Therefore we conclude that our analysis is correct, and we obtain the results we expect in all cases. Caveats to this limited, and are expressed in the analysis section. Since the native install of Alt-Ergo is unreliable (we still completely integrate the solver/interface into our code and present SMT results for completeness), we also output all of our analysis into a .proof file, where we format our queries into the format that the online solver expects (to enable easy copy-pasting for skeptical TAs).

Despite our success, there are still several aspects of our static analysis that we could improve, and this is presented in the closing section.

Project Specification.

Language Specification. We support the full L3 language, with slight modifications to allow for reuires, ensures, and loop invariants. The only caveat to this is that we explicitly disallow function calls to be made in:

- If conditionals
- While loop conditionals
- Return statements, if a corresponding 'Ret' expression is present in the ensures condition (to be explained in the ensures section)

And these restrictions exist primarily because inferring facts in these situations turned out to be rather difficult given our internal representation from Lab5. In order to spend time on more interesting parts of the module, we felt these were rather small restrictions to put on the programmer. We stuck to the L3 language versus the L4 language because while handling arrays was relatively easy, handling struct fields really taxed the typing system present in the 3rd-party SMT solvers we wanted to use. We also felt justified in this decision, because it allowed us to focus more on formal verification and less on modifying our internal representation for structures. It is important to note that we only enable these restrictions if the file in question has the ".l6" file type, and otherwise we are fully backwards-compatible.

The syntax for 'Requires' and 'Ensures' statements is displayed below:

```
<return_type> <function_name> (<arg_list>) {
  Prove{
    Requires [<boolean_expr_list>];
    Ensures [<boolean_expr_list>];
  }
  ... function body ...
}
```

```
int func_call(int i) {
  Prove {
    Requires [(i > 5)];
    Ensures [(Ret > 0)];
  }
  return (i-5);
}
```

Fig. 1. Function Pre and Post Conditions: left - syntax, right - example

and we support any number of requires and ensures conditions. *Requires* conditions must only refer to input variables, while *Ensures* conditions can refer to any variable in the program that is live at all function exit points. We also support the *Ret* keyword, which simply refers to the return value of the function. We substitute this for the various return values at proof time. We only support simple statements containing *Ret* on the left-hand side, since otherwise it is tricky to be able to substitute out this keyword at proof-time in a more complicated expression. Regardless, we still think it was a nice time-saving additional feature for the programmer.

The syntax for loop invariants is shown below. We introduce a new type of "provable" loop, *while_invariant*, in order to give the programmer the option of using the original loop constructs without invariants. However, using a regular while or for loop will mean that we cannot infer a loop postcondition fact, which may hinder proof verification.

We did not introduce a *for_invariant* for brevity; All for loops can be elaborated to a while loop at program time, so this would be a repetitive construct.

The Induction_Var statement lists all induction variables that the programmer intends to change in the loop. The $< Increasing|Decreasing >$ option indicates whether the variable increases or decreases each loop iteration. While these facts are not necessarily useful for our other analyses, we found it a useful exercise for the programmer to express

```
while_invariant(<boolean_expr>)
  Prove{
    Induction_Var[(<var>, <Increasing or Decreasing>)];
    Halts_When[<boolean_expr_list>];
    Invariant [<boolean_expr_list>];
  }
{
... loop body ...
}
```

```
while_invariant(i > 0)
  Prove{
    Induction_Var[(i, Decreasing), (j, Increasing)];
    Halts_When[(i == 0)];
    Invariant [(n == 2 * (20 -i))];
  }
{
  n = n + 2;
  i = i - 1;
  j = j + 1;
}
```

Fig. 2. Loop Invariants: left - syntax, right - example

this at program-time so that we can detect its correctness. We show in a later section how our system proves that the $< Increases|Decreases >$ conditions are met. We originally intended to use these proofs to ensure that, given the starting fact about the induction variables, that the halting condition is met: This has been deferred to future work due to time constraints, but is the first thing we would implement given more time.

The other fields are of slightly more consequence. *Halts_When* specifies a list of conditions that become true when the loop terminates (and in fact, cannot be true while the loop is running). We verify that these conditions do, in fact, minimally signal that the loop condition is false; once we have this condition it is very useful for inferring the loop postcondition. *Invariant* lists a series of conditions that must be true in the following circumstances:

- True upon entry to the loop
- If true on iteration n, true on iteration n+1
- True upon exiting the loop

And we prove these properties with the formal verification described in a later section.

These invariant statements give us useful facts for proving the rest of the program. Namely, we know that upon loop exit, that both the halting condition and the loop invariant(s) are true. If the Invariant contains variable defined by the halting condition (which it typically does), this gives us powerful knowledge to continue to prove the function ensures.

Our syntax diverges slightly from Dafny-style formal verification, and while it is certainly clunky, it is functional and meets the goals of the project.

Static Analysis. The largest portion of our project was dedicated towards performing static analysis on the input program in order to tag each program point with a boolean expression representing all true knowledge we have at that point. Trivially this can be as simple as determining whether the branching condition is met based off of whether the program point is in the if or the else case, but in reality the 'Inference Rules' are much more complicated.

We will begin by specifying what we mean by "facts". Conversationally, a fact is a any expression written in the L3 language that typechecks to a boolean, with additional *Not* and *Implies* constructs, and disallowing function calls (handled by our inference rules). *Not* can be applied to an expression to negate it (a more conceptual version of the standard 'bang' operator), and *Implies([e1, e2, e3, ...], [r1,*

r2, r3, ...]) represents the the knowledge that if all of the *e* conditions are true, they imply all of the *r* conditions.

Written in formal BNF form we have the following:

Fig. 3. Fact BNF

And so the static analysis problem can be rephrased as the following: Tag each program point with the strongest fact list we can, where each fact in the list is verifiably true coming into that program statement.

We now overview the fact 'Inference Rules' that allow us to derive facts from the program:

(1) We assume all *Requires* facts to be true upon function entry, because we are trying to prove that requires implies ensures.

(2) We can derive facts from simple statements depending on their form. We define simple statements as:

- Move Instructions
- Assert Instructions

For Move instructions we must consider several cases depending on the source expression. We can be certain that the destination is simply a temporary (since we do not have arrays or struct dereference in L3), but the source expression may or may not provide a definitive assignment value for the temporary. Enumerating the possibilities:

- Constant

  given a statement of the form 't = constant', we can derive the fact (t == constant)

- Temporary

  given a statement of the form 't = t2', we can derive the fact (t == t2)

- Function Call

  given a statement of the form 't = call(args)', we can derive the fact (Implies(call_requires(args), call_ensures(t)).
  Specifically, we can take the call_requires conditions and substitute in the argument values for our input parameters, and we know that this ensures the call_ensures conditions if we substitute out all 'Ret' keywords for 't'. Any Ensures statements not containing the Ret keyword are ensuring properties about variables internal to 'call' and so these are of no consequence to the caller (and hence we ignore them).
  The ensures could be an assignment (Ret == 0), a bound (Ret $<$ 0), or multiple bounds (Ret $>$ 5, Ret $<$ 200).

- Binop

  given a statement of the form 't = a op b', we must derive facts relating to a and b, and then merge them with the op operator.
  if $< a|b >$ is a constant or temporary, they generate the intermediate fact(s) $(< a|b >==< constant|temporary >)$
  if $< a|b >$ is a function call, we generate the intermediate implication fact that if the input arguments to the function obey the requires statement, that $< a|b >$ obey the ensures statement. I.e, if a == call(i,j) and i,j satisfy the requires of call, then 'a' satisfies the ensures of call.
  if $< a|b >$ is a trinop $(c?d : e)$, we know that c implies a==d and not c implies a==e.
  Once we have facts for a and b, we must merge with the proper operator, op.
  If we have an 'Implies' facts for one or both of a and b, then any knowledge about the composition of a and b with the op operator requires both implication requirements to evaluate to true. Specifically, given the case where t = a op b, and we can only derive that (cond1 $->$ some knowledge about a) and (cond2 $->$ some knowledge about b), we can only derive an overall fact of the form (cond1 and cond2 $->$ some joint knowledge derived by merging our a and b knowledge with the op operator). We can now focus simply on merging that knowledge of a and b into a useful fact for t.
  This is trivial for if we have definitive assignment information for a and b (i.e. we have intermediate facts (a == xx) and (b == yy). In this case we can simply state that t == xx op yy.
  If we only have bounds on a and b, such as having $(a < 10, a > 0)$ and $(b < 20, b > 4)$, then we can merge the bounds according to op. For example, given the facts above with op = +, we know that $(a + b < 30, a + b > 4)$. Similar rules for other binary operators follow intuitively. If we are missing bounds. such as we know $(a < 10, a > 0)$ but only $(b > 5)$, then we can only derive one of the bounds, i.e., a+b has a lower bound but no verifiable upper bound.

- Trinop

  given an expression 't = (a ? b : c)', we know that a implies t == b and b implies t == c

And it is important to note that we recursively apply these derivation rules to complex left-hand assignments, meaning that the logic for this is fairly complex.

For Assert instructions we follow a similar protocol to derive what we know about the assert expression. In this case,

we can simply assume that the expression evaluates to true, and work backwards to derive whatever facts we can. For simple statements this is fairly easy, since $(a < b)$ evaluating to true means that we simply add $(a < b)$ as a fact. For expressions containing a function call we unfortunately cannot make conclusions. This is because even though we know that an ensures may hold, ensures statements cannot be used to imply anything about requires statements, and therefore we cannot infer anything about the input arguments, unless we infer the contrapositive (if ensures does NOT hold, then requires has definitely NOT been met).

(3) Facts remain valid as long as the temporaries they rely on have not been overwritten or redefined. In the case they are, we can define a new temporary to represent the old value and keep our old facts around, while still adding knowledge about the new state of the variable. This is an 'on-the-fly' conversion to SSA form, at least from the perspective of our fact lists.

(4) 'If' statements are handled similarly to assert statements. We know that in the 'if' branch the condition is true, and that it is false in the 'else' branch.

To merge the else and if branches, we can simply isolate the "if facts", or new facts generated by the if branch, and "else facts", the new facts generated by the else branch. For the rest of the program, we now know that if condition = true implies "if facts" and if condition = false implies "else facts".

Ideally we have enough information to infer the value of the if condition at the point the decision is made in the code. In that case, we end up AND-ing our merged implication statement with a fact of the form (if condition == false) or (if condition == true), and the SMT solver will be able to isolate which branch was taken.

(5) While loops without invariants are fairly difficult to reason about. All we know in this case is that after the loop, the converse of the looping condition is true. We also must delete facts that are clobbered by the interior of the loop body as per (3).

For isolating what we know to be true in the loop body, we do know that the looping condition is true. This is useful for proving ensures if there is a return in the loop body, but otherwise is not helpful, since we have no invariants to prove in any case.

(6) While_Invariant loops are more exciting. In addition to knowing in the loop body that the loop condition holds, we also know that the loop invariant holds on each iteration, and that the halting condition does not hold.

When we exit the loop, we know that both the halting condition and the invariant holds, giving us a powerful loop postcondition.

Proof Task Formulation. Now that we have fully annotated the program with which facts we know to be true (and when), we can formulate the series of program correctness proofs that we will feed into our SMT solver.

For each function, we have a series of proof goals:

Ensures To prove ensures statements, we can simply prove that for each return point, that the facts at that program exit point are enough to imply all of the function ensures statements.

$$\forall \mathbf{f} \in \mathbf{functions}$$
$$\forall \mathbf{r} \in \mathbf{function\_return\_points}(\mathbf{f}),$$
$$\mathbf{Facts\_True\_In}(\mathbf{r}) \implies \mathbf{Ensures}(\mathbf{f})$$

where Facts_True_In(r) is the set of facts that are true upon entering program point r.

Loop Invariants To prove loop invariants, we must prove that for every loop in each function:

$$\begin{aligned} &\forall \mathbf{i} \in \mathbf{loop\_iterations} \\ &\mathbf{Facts\_True\_Out}(\mathbf{i}) \ \mathbf{and} \\ &\mathbf{Invariant}(\mathbf{i}) \ \mathbf{and} \\ &\mathbf{Looping\_Condition} \\ &\implies \mathbf{Invariant}(\mathbf{i}+\mathbf{1}) \end{aligned}$$

where Looping_Condition is the boolean case that, when true, indicates the loop should run again (i.e., 'expr' in while(expr)), and Facts_True_Out(i) is the set of facts we know to be true after the ith iteration of the loop body.

We prove this by AND-ing together the looping condition, the invariant statement(s), and the 'loop true out' facts we determined during static analysis. The following boolean expression is a function of the loop induction variables. We can then set up, for each one of the loop invariants, the implication statement shown above.

Loop Halting Condition Proving the loop halting condition is rather trivial. We must prove for all loops in each function:

$$\mathbf{Halts\_When} \implies \mathbf{Not}(\mathbf{Looping\_Condition})$$

and this is a fairly easy implication statement to set up.

As a general disclaimer, we never prove that the halting condition is ever reached (which is a rather difficult proof), rather we only prove that when it *is* reached, that the loop terminates. This is left as future work.

Loop Induction Variables Proving that the loop induction variables behave properly consist of proving the following statement for all loops in each function:

$$\begin{aligned} &\forall \mathbf{i} \in \mathbf{loop\_iterations} \\ &\forall \mathbf{v} \in \mathbf{induction\_variables} \\ &\mathbf{Invariant}(\mathbf{i}) \ \mathbf{and} \ \mathbf{Looping\_Condition} \\ &\mathbf{and} \ \mathbf{Facts\_True\_Out}(\mathbf{i}) \ \mathit{with} \ \mathbf{v} = \mathbf{v_{old}} \\ &\implies \mathbf{If}(\mathbf{dir}(\mathbf{v}) == \mathbf{Increases}) \\ &\mathit{then}(v_{old} < v_{new}) \\ &\mathit{else}(v_{old} > v_{new}) \end{aligned}$$

where Facts_True_Out(i) is the set of facts generated by the loop body on iteration i (we assume for this proof that the induction variables are uniformly increasing or decreasing, so we just prove for the first iteration), and dir(v) is the direction declared for the induction variable in the syntactic *Halts_When*

statement. v-old is the old value of induction variable v (i.e. before the loop runs) and v-new is the new value of induction variable v.

Theorem Proving. To prove the theorems we declared in the previous section, we first had to format them as an SMT query.

Luckily, the Why3 Ocaml interface provides a reasonably nice way of interfacing with 3rd party SMT solvers like Z3 and Alt-Ergo (we tried CVC4 but to no avail).

In order to plug in our proof tasks we first format each of the theorems derived above into a proof goal:

```
goal ensures_0:
  forall t9, t10, t8, t2, t4: int.
    (((((((t9 == 20) and (t10 == 20)) and (t8 == 0)) and !((t9 > 0)))
    and ((t2 == 0) and (t4 == (2 * (20 - t2))))))
      -> ((t4 == 40)))
```

Fig. 4. Alt-Ergo formatted proof goal

and then each proof can be directly evaluated by the OCaml Alt-Ergo bindings.

It is important to note that we internally convert all booleans into integers, to make the typing easier for our SMT queries.

We output the full analysis and proof results in a .proof file:

```
##################### PROOF FILE ####################
Copyright Scott Mionis


Source Program:
int main () {
  Requires (); Ensures ((ret == 0), );
  int x = 1;
  return x;
}



###### Analyzing Function <main>:
Analysis printed of the form <statement> : <known facts>
main () {
  %t2  <--  1 :
  return %t2 : (%t2 == 1),
}

> Proof Tasks for Ensures Statements
goal ensures_0:
forall t2: int.
(((t2 = 1)) -> ((t2 = 0)))

@[On task, alt-ergo answers Invalid in  5.00 seconds@.

> Proof Tasks for Loops

Apologies for the Timeouts...
Alt_Ergo native install seems to be unreliable
Try online at: https://alt-ergo.ocamlpro.com/try.html
```

Fig. 5. Lab6 .proof output file

In each .proof file we print the original source code. Then we analyze each function one by one, outputting both the annotated code and each of the proof goals. We segment the proof goals into ensures proofs and loop proofs, and the Alt-Ergo results are printed in order.

Since the native install of Alt-Ergo is unfortunately unreliable, and times out, we output the proof goal in Alt-Ergo syntax so you can copy paste easily into the online version linked in the introduction.

To read more about the specifics of these SMT solvers, please reference our implementation section.

Project Implementation. Most of the additional code needed to implement this project was purely in the front end of the compiler. Specifically, we had to modify the Parser/Lexer to accept our new language constructs, the Typechecking module to verify that all requires and ensures cases typecheck to booleans, the Translation/Elaboration module to accept the new language constructs, and then the brand new Prover module to actually perform theorem proving and static analysis. Nothing past the Prover module is affected, since we erase requires and ensures semantics from the program when the prover module returns.

Parser/Lexer. The changes to the parser and lexer were relatively minimal. They primarily consisted of implementing the language specification presented in the earlier section.

Also of note is the Ret keyword. we parse the Ret keyword as an expr (in L3 syntax terms). However, we will throw an error later if we find a Ret keyword anywhere except on the right-hand-side of an ensures expression.

Typechecker. In our Typechecker module, we implemented code to verify that all expressions in the requires and ensures statements are of type boolean. We also typecheck loop invariants, halting conditions, and verify that the induction variables are of type integer. We also had to modify our global declaration typechecker (funtable.ml/mli) to extract requires and ensures information.

Prover Module. The prover module implemented in prover.ml/mli constitutes the bulk of our code. The conceptual layout is as follows (top-level function is called static_analysis):

- Perform Static Analysis to generate facts, and zip program statements together with the appropriate fact lists. (zip_stms function)
- Remove Boolean expressions (rather convert them to purely integer expressions) to make converting to an SMT query easier (fix_bools function)
- Generate a list of ensures proofs (generate_prover_tasklist function)
- run the ensures proofs on Alt-Ergo (run_tasks_alt_ergo function)
- generate a list of loop proofs (generate_loop_prover_tasklist function)
- run the loop proofs on Alt-Ergo (run_tasks_alt_ergo function)
- cleanup any remaining Lab6 objects before passing code to the downstream compiler module. (remove_while_invariants)

And the other functions are simply helper functions, and not worth mentioning besides these significant ones:

- simp_stm_to_facts : converts a simple program statement into the facts it generates (this calls a helper, format_mov_to_facts, which does the implication merging mentioned in the problem specification)

- translate_prob : convert a fact expression into a Why3 task

Of note is the fact that we output all prover results and static analysis traces to a .proof file, which is the go-to resource for seeing why a proof succeeded or failed. As mentioned before, the prover goals are output in such a fashion that they can be copy-pasted into the online version of Alt-Ergo, which is much more reliable than the downloaded version. We will elaborate on this in the testing section, and provide a README in the lab6/tests directory instructing the user how to actually evaluate the results, since the MacOS native install of Alt-Ergo will timeout on statements that are trivial for the browser version to prove.

In terms of algorithms, the only notable algorithm is the Static Analysis algorithm, although this simply is a linear scan from the top of the program to the bottom of the program.

In terms of data structures, the primary data structures we used were syntax trees to represent facts and implications of facts.

Why3. In order to interface with external provers like Alt-Ergo and Z3, we used the Why3 OCaml interface. We chose this package because it is compatible with multiple 3rd-party SMT solver backends, and is installable via the opam package manager. Additionally, the syntactic interface it provides for queries were more similar to our internal syntax trees than the direct Z3 OCaml bindings, and much documentation on Why3 can be found at: http://why3.lri.fr . Installation instructions are provided in the project README file.

File breakdown. While pretty much all new code is isolated to prover.ml, here is a rundown of all the files we modified:

- Top.ml - the top level module that contains the main compile function C0 lexer.mll/C0 parser.mly - Lexer/Parser
- funtable.ml/mli - Global Declaration Typechecker (typecheck global struct definitions, file order, function declarations, etc. . .)
- typechecker.ml/mli - Typechecker
- trans.ml/mli - Translate and Elaborate
- prover.ml/mli - Static Analysis and Theorem proving

Testing Methodology.

Our general approach to testing was to try to get code coverage over all our new features. We also tried to exercise the static analysis module as much as possible, and tried to craft some tricky tests to leverage merging function calls with binary and trinary operators (i.e. t = call() + call() / call() is a tricky statement to infer facts from, but doable with the correct ensures statements). primarily our testing comprised of various internal unit tests, but we also leveraged several real test cases which we describe below.

Additionally, there were only subjective metrics to evaluate whether our prover module was smart enough. Other than testing correctness (i.e., we reject code that is wrong or inconclusive), there is also the question of whether we derive as many facts as we could be (i.e. we accept as many valid programs as possible). We evaluated this metric as well, and were impressed with our results.

Of highest importance is the note that the local opam install of Alt-Ergo is unreliable and seemingly times-out on easily verifiable statements. We have verified this by putting our proof goals into the online version of Alt-Ergo, which agrees with our compiler on all tests.

In each .l6 test case, we specify at the top of the file whether it is a proof fail or proof success. If there are timeouts and you wish to verify this, you can simply copy and paste the proof into the online Alt-Ergo prover. For proof success tests, all proof goals should return Valid. For proof fail tests, at least one proof goal will return Invalid or Unknown.

Testing Framework.

To run our tests, navigate to the Lab6 directory and run make test. This will run all tests and generate the .proof files.

In order to run our tests, we utilized the `gradecompiler` program with a compile timeout set to 100 seconds. While our compiler generally didn't take this much time, our third party theorem provers usually did if they got stuck in a timeout.

What this means is that the make test command does not necessarily test the provability of the code, but rather just the correctness of the assembly output. To test proof fail/success, you should inspect the .proof file manually, and if there are timeouts, you should consult the online version of Alt-Ergo. This testing method was chosen because since Timeouts were seemingly so common in the Why3 software, running our automated proof verifier ended up marking too many test cases as failures. We found it much more effective to simply dump our analysis to a .proof file and make the theorems transparent for a programmer to verify with their own tooling if necessary.

make cleantest will delete all .proof files.

Test Cases. We ran a core suite of 19 test cases in order to verify the correctness of our program: these test cases varied from simple l1 programs augmented with requires and ensures to more complex l3 programs with multiple functions. A subset of the testbench we provide is described below, although it is important to not that much of our testing during development involved unit tests versus full testcases.

1) big01.l6 - proof-success: a large test that uses asserts to inserts facts, and has mutliple control flow statements
2) error01.l6 - proof-fail: a simple test that proof-fails because it has a Ret ensures condition in a void function
3) loop01.l6 - proof-fail: multiple functions and loops. The test fails because a called function does not have any contracts, and hence we have no visibility into it.
4) loopinv01.l6 - proof-success: tests a single function with a loop, augmented with loop invariants
5) loopinv02.l6 - proof-success: tests a single function with a loop, augmented with loop invariants
6) loopinv03.l6 - proof-success: tests a single function with nested while loops
7) simple01.l6 - proof-success: tests calling a function augmented with contracts
8) simple02.l6 - proof-fail: failure case where there's not enough information to determine if a function precondition holds, and therefore we cannot infer the ensures

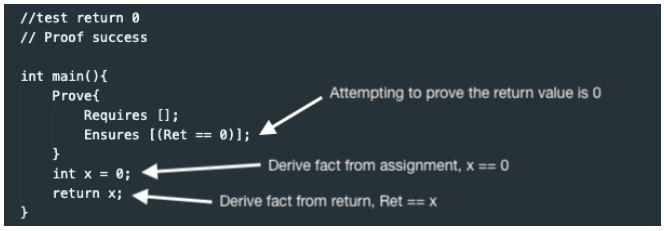
9) simple03.l6 - proof-success: corrected version of simple02.l6
10) simple04.l6 - proof-success: tests a trivially-true requires condition for a function call
11) simple06.l6 - proof-fail: failure case because a helper function doesn't have the proper ensures statements
12) simple07.l6 - proof-fail: test case where a helper doesn't have visibility into another function, and thus cannot prove ensures
13) simple08.l6 - proof-fail: testing a trinop, which is tough to prove because you have to prove both cases
14) simple09.l6 - proof-success: tests control flow
15) simple10.l6 - proof-fail: same as simple09.l6, but a failure case
16) simple11.l6 - proof-success: tests a series of chained if/else statements
17) simple12.l6 - proof-fail: tests an invalid proof where the ensures doesn't match the return value
18) simple13.l6 - proof-fail: unable to prove a helper function because of a lack of ensures
19) trivial.l6 - proof-success: same as simple12.l6, but a success case

Also feel free to try anything that is valid syntax. Although limited, we are confident that the tests we have provide enough code coverage to prove the correctness of our module.

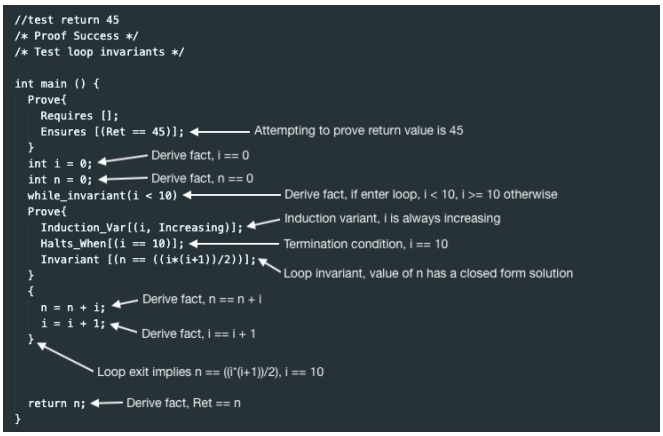
Validation. As stated before, many of our third party theorem provers, such as alt-ergo, often timed out when run locally. As a result, we had to manually verify the output of our program using the web based versions. This was greatly aided by the fact that we generate comprehensive proof files.

This proved successful for us, and we are convinced of the correctness of our module by comparing our proof output to the SMT solver results online. Specifically we leveraged the online Alt-Ergo platform (link provided at the beginning of the report).

Example The following shows how facts are derived from a very basic test case.



And the following shows how facts are derived from a more complex test case.



Analysis.

This project was admittably more difficult than the initial specification led us to believe. Interfacing with 3rd-party SMT solvers was tricky, and we tried a few different OCaml bindings and software packages before we found the Why3 interface, which seemingly worked the best. Additionally, while the browser versions of Alt-Ergo and other theorem provers worked fine, the opam versions of these modules seemed very fragile and were prone to timeout failures. We are at ease about this issue because our code is complete, and our results agree with the browser versions of all the theorem provers we tested.

Moving on to a more critical assessment of our project goals, we believe that we met the original project goals. We are able to effectively prove, at compile-time, the correctness of various, nontrivial programs (see the big01.l6 test case). We are confident in our approach to static analysis, and while we did draw several ideas from Hoare Logic and Symbolic Execution techniques, the approach we took was our own.

We also are able to effectively prove loop invariants, although there are a few caveats to our loop proof structure:

- We only loosely prove that induction variable directions (Increasing, Decreasing) hold. Theoretically we should be evaluating this for every iteration. However, we make the assumption (and requirement) that the induction variable be updated in the same fashion each loop iteration. This is not as useful in practice, and so we would adopt a more generalizable approach
- We do not prove that the halting is ever reached. Rather, we onll prove that if the halting condition is reached, that it would actually terminate the loop. Theoretically we should be able to prove that the halting condition is reached by incorporating information about the induction variables, but this was left unimplemented due to size constraints.

Other, more general, limitations of our project are as follows:

- We do not support L4 specifications. These primarily include structures and arrays. While we *did* implement purity analysis in order to tell whether to delete facts about memory locations over function calls or not, we did not decide to focus on getting structures and arrays

working. This is because we felt that the interesting part of the project was static analysis and theorem proving, and we did not want to spend our time on trying to uniquely identify structure fields.
- As mentioned before, it is unfortunate that the opam install of Alt-Ergo is unreliable, However our code is still complete and our examples can be verified online.

Future work we have planned for this project is as follows:

- Make Loop invariants more robust, as per our critical evaluation of it below
- Support the full L4 language specification
- Further debug the opam version of Alt-Ergo to get it to stop timing out
- Implement higher-level constructs for invariants, such as forall statements and exists statements. (i.e., be able to write a requires of the formz"forall i in 0 to 10, $array[i] < 0$" or "exists i in 0 to 10 s.t. $array[i] < 0$")..
- Implement reference counting to keep track of memory pointers, so we can prove that all memory accesses are to valid pointers initialized by a malloc call.
- In the case that a proof is inconclusive, we should be able to tell what information is missing and give informed warnings to the programmer so they can fix the issue

And we hope to be able to continue to work on this project in order to implement these changes.

Thanks.

Thank you to the entire course staff for a great semester!

GitHub Link.

https://github.com/15-411-f20/Interligua/tree/lab6cyoa

REFERENCES

[1] http://www.cs.cmu.edu/ãldrich/courses/17-355-19sp/notes/notes11-hoare-logic.pdf
[2] Hoare C.A.R. An Axiomatic Basis for Computer Programming: October 1969 https://doi.org/10.1145/363235.363259
[3] http://www.cs.cmu.edu/ãldrich/courses/17-355-19sp/notes/notes14-symbolic-execution.pdf
[4] http://why3.lri.fr
[5] http://www.cs.cmu.edu/ãldrich/courses/17-355-19sp/notes/notes12-smt.pdf
[6] https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/