COMP 150-SEN
Software Engineering Foundations

Program Verification (cont’d)

Spring 2019
Review: Verification

• Given: A program and a specification

• Goal: Prove the program abides by the spec

• Unlike testing, verification can show that a program is correct for all inputs
  ▪ Testing: `assert(abs(-5) >= 0), assert(abs(10) >= 0), assert...`
  ▪ Verification: `assert(∀ int x : abs(x) >= 0)`
Review: Weakest Preconditions

• Given: a program statement $S$ and a postcondition $B$

• Find: weakest precondition $A$ for $S$ such that $B$ will hold
  - $A$ is weaker than $A'$ if $A$ is true in more states
    - E.g., $x > 0$ is weaker than $x > 5$
    - Thus, $A$ is weaker than $A'$ if $A' \implies A$
    - (true is the weakest possible assertion)
  - Thus, the weakest precondition $A$ means
    - $\{A\} S \{B\}$ is valid (A is a precondition)
    - If $\{A'\} S \{B\}$ is valid, then then $A' \implies A$
Review: Weakest Preconditions

- Define $wp(S, B)$ to be the weakest precondition of $B$, as follows
  - $wp(x=e, B) = B[x \mapsto e]$
  - $wp(S1; S2, B) = wp(S1, wp(S2, B))$
  - $wp(if E then S1 else S2, B) = E \Rightarrow wp(S1, B) \land \neg E \Rightarrow wp(S2, B)$
Loops

• There is no mechanical way to automatically check whether loops are correct

• Two key problems
  ▪ Weakest preconditions might not terminate, because it needs to keep going backward through the loop repeatedly
  ▪ We’d like to also know whether loops terminate, which weakest preconditions doesn’t reason about

• Instead, we’re going to develop a pen-and-paper way of reasoning about loops
Summation in a Loop

• Goal: prove the following triple correct

```plaintext
[ true ]
sum = a[0];
k = 0;
while (k ≠ n-1) {
    k = k + 1;
    sum = sum + a[k];
}
[ sum = a[0]+a[1]+...+a[n-1] ]
```

- Notice this is total correctness, i.e., we need to prove the loop terminates
- (We will ignore concerns about the array length)
Loop Invariants

- Key idea: think about one loop iteration, generically
- A loop invariant is an assertion that holds at the beginning and end of each execution of the loop

\[
[Q] \text{S_init; while}(B) \{ S \} [R]
\]
- Let \( I \) be the loop invariant

- To prove this loop correct
  1. Show \([Q] \text{S_init} [I]\) (invariant holds before loop)
  2. Show \([I \land B] S [I]\) (invariant holds across loop)
  3. Show \(I \land \neg B \Rightarrow R\) (post holds after loop)
Loop Invariant Example

[true]
sum = a[0];
k = 0;
while (k ≠ n-1) {
    k = k + 1;
    sum = sum + a[k];
}
[sum = a[0]+a[1]+...+a[n-1]]

1. [true] sum=a[0]; k=0; [sum=a[0]+...+a[k] ∧ 0≤k<n]
   - Holds by sequence and assignment wp rules
Loop Invariant Example (cont’d)

2. \[ \text{sum}=a[0]+\ldots+a[k] \land 0 \leq k < n \land k \neq n-1 \]
   
   \[
   k = k + 1; \quad \text{sum} = \text{sum} + a[k];
   \]

   \[ \text{sum}=a[0]+\ldots+a[k] \land 0 \leq k < n \]

   - Holds by assignment rules and consequence
Loop Invariant Example (cont’d)

I: \((\text{sum} = a[0]+…+a[k] \land 0 \leq k < n)\)

3. \((\text{sum}=a[0]+…+a[k] \land 0\leq k<n \land k = n-1) \Rightarrow \)

\text{sum} = a[0]+a[1]+…+a[n-1]

- Holds by standard logical reasoning
Bound Function for Termination

• A bound function $t$ is
  
  - An integer-valued expression defined in terms of program variables
  - $t$ must strictly decrease with every loop iteration
  - $t \geq 0$ always, so that when it reaches 0 the loop terminates

• I.e., a bound function is an upper bound on the number of remaining loop iterations
Bound Function Example

- Check that \( t \) is a bound function
  - \( t \) strictly decreases with each step because \( k \) increases by 1
  - \( t \) can never go below zero because the loop terminates with \( k=n-1 \)
- Therefore, the loop terminates
Verification Conditions

• A verification condition (VC) is a logical formula that is valid if a program abides by its spec
  ▪ Formula validity means it is true for any assignment to its variables

• We can verify programs if we have ways to:
  1. Generate VCs based on programs and their specs.
  2. Check the validity of those VCs.

• For (1), we can use weakest preconditions!

• Given: Hoare triple \( \{A\} \ S \ \{B\} \)

• We can generate VC: \( A \Rightarrow wp(S, B) \)
Generating VCs

• Given program:
  \[\text{if } x \geq 0 \text{ then } \text{abs}_val = x \text{ else } \text{abs}_val = -x\]

• Given postcondition: \(\text{abs}_val \geq 0\)

• No precondition (i.e., precondition is just \text{true})

• Goal: use weakest preconditions to generate VC
Using Weakest Preconditions

- Recall wp rules for assignment and conditionals
  - $wp(x=e, B) = B[x \mapsto e]$
  - $wp(\text{if } E \text{ then } S1 \text{ else } S2, B) = E \Rightarrow wp(S1, B) \land \neg E \Rightarrow wp(S2, B)$

$wp(\text{if } x \geq 0 \text{ then } \text{abs}_val=x \text{ else } \text{abs}_val=-x, \text{abs}_val \geq 0)$

Apply conditional rule:

$= (x \geq 0) \Rightarrow wp(\text{abs}_val=x, \text{abs}_val \geq 0) \land \neg (x \geq 0) \Rightarrow wp(\text{abs}_val=-x, \text{abs}_val \geq 0)$

Apply assignment rule:

$= (x \geq 0) \Rightarrow (x \geq 0) \land \neg (x \geq 0) \Rightarrow (-x \geq 0)$
Generating, Checking VCs

• The weakest precondition was:

\[(x \geq 0) \Rightarrow (x \geq 0) \land \neg (x \geq 0) \Rightarrow (-x \geq 0)\]

• Because the precondition of the program was \text{true}, the VC for this program is:

\[\text{true} \Rightarrow ((x \geq 0) \Rightarrow (x \geq 0) \land \neg (x \geq 0) \Rightarrow (-x \geq 0))\]

• Now, must check validity of VC
• Could write handwritten proof, but an automated proof is preferable
SMT Solvers

• Satisfiability modulo theories (SMT)

• SMT solvers extend boolean satisfiability solving with theories from other domains
  ▪ Satisfiability: Is there an assignment to the variables of a formula that makes the formula true?
  ▪ Integers, real numbers, arrays, records, …

• Allow us to check the satisfiability of formulas involving more than just boolean logic
  \[ \text{true} \Rightarrow ((x \geq 0) \Rightarrow (x \geq 0) \land \neg (x \geq 0) \Rightarrow (-x \geq 0)) \]

• Recent (~15 years) progress in SMT solving is a major reason for advances in program verification
Using SMT Solvers

• Z3 is a popular, state-of-the-art solver
  - Developed at Microsoft Research by Leonardo de Moura, Nikolaj Bjorner
• We will use it to check if our VC is valid
• A formula is valid if and only if its negation is unsatisfiable
• So, we will check the satisfiability of VC’s negation:

  \neg (true \Rightarrow ((x\geq 0) \Rightarrow (x\geq 0) \land \neg (x\geq 0) \Rightarrow (-x\geq 0)))

• Here it is in Z3: https://rise4fun.com/Z3/4sIBH
Putting it all Together

• We have a way to generate VCs
  ▪ Using weakest preconditions
  ▪ Not mechanical for loops, though: need manually provided loop invariants

• We have a way to check the validity of VCs
  ▪ Using SMT solvers

• Putting these together, we have a way to achieve program verification
Dafny

- Dafny is an object-oriented, compiled language with constructs for verification
- Supports formal specs through preconditions, postconditions, and loop invariants
- All specs are verified
  - VCs generated with weakest preconditions, checked with Z3
  - Many, many more algorithms and heuristics used
  - Dafny also proves program termination (w/ programmer help)
- Also developed at Microsoft Research, under direction of Rustan Leino
Using Dafny

• Biggest challenge is coming up with loop invariants
  ▪ No automatic procedure for doing so
  ▪ But, there are heuristics we can use to make good guesses

• We will use Dafny to verify absolute value method, and insertion sort
Review: Insertion Sort

- Pseudocode:

```plaintext
i ← 1
while i < length(A)
    j ← i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j ← j - 1
    end while
    i ← i + 1
end while
```
Using Dafny

- Demo: https://rise4fun.com/Dafny/op05p

- Fully annotated version: https://rise4fun.com/Dafny/GxFe
Using Dafny

• Verifying specs is reasonably practical!

• Some aspects, especially loops, are tricky

• But, it shows the promise of new verification technologies

• Many more verification “success stories” in recent years…
Ironclad Apps

- Paper + implementation by Hawblitzel et al.
- Built four verified apps using Dafny (+ other tools)
  - Notary app securely assigns timestamps to documents
  - Password hasher
  - Differentially-private database
  - Multi-user trusted counter
- Also implemented code used by apps
  - Libraries: Ints, arrays, strings, …
  - Cryptographic tools
  - Network driver
  - …
Ironclad Apps

• Verified a host of correctness, security properties
  ▪ Remote equivalence: “that to a remote party the app’s implementation is *indistinguishable* from the app’s high-level abstract state machine.”
  ▪ No code injection
  ▪ No buffer overflow
  ▪ Correct crypto implementations
  ▪ No data leaks
  ▪ …

• Nontrivial effort
  ▪ ~6K lines of implementation
  ▪ ~30K lines of proof annotations
  ▪ ~3 person-years
CompCert

- Unfortunately, compilers are full of bugs

- The CompCert project produced a formally verified C compiler

- Proves: the executable code produced by the compiler behaves exactly as specified by the semantics of the source C program
CompCert

• Compiler entirely written and verified using Coq, a programming language and proof assistant
  ▪ Coq helps write, and mechanically checks, formal proofs

• High-level idea: Formally define source and target languages, prove compiler preserves semantics
  ▪ For programs source S, target T, compiler C, prove:
    ▪ ∀ input i: T = C(S) ⇒ S(i) = T(i)

• Do this using many intermediate languages
  ▪ Makes proofs easier
CompCert

source: http://compcert.inria.fr/compcert-C.html
CompCert

- A major undertaking
  - ~6 person-years of effort
  - ~100,000 lines of Coq code

- Uses many optimizations, making its runtime competitive in practice:

source: http://compcert.inria.fr/compcert-C.html
More Successes in Verification

- **seL4**: A fully verified operating system kernel
  - Written in C, proven correct using proof assistant Isabelle

- **Refinement type systems**
  - Extend types with expressive predicates
  - e.g., `Integer x { x > 0 }`
  - Have been applied to Haskell, Racket, JavaScript, Ruby, and more

- **Formally verified radiotherapy machine**
  - In use at University of Washington Medical Center!

- …and many more!
Links

• Dafny tutorial

• Z3 tutorial

• Software Foundations: a textbook, written in Coq, teaching the basics of proofs about programs