Finish Verification Condition Generation
Simple Prover for FOL

CS 294-8
Lecture 10

Not Quite Weakest Preconditions

- Recall what we are trying to do:

false ⇒ true
Pre(s, B)
strong — weak
A
weakest precondition: WP(s, B)
verification condition: VC(s, B)

- We shall construct a verification condition: VC(s, B)
  - The loops are annotated with loop invariants !
  - VC is guaranteed stronger than WP
  - But hopefully still weaker than A: $A \Rightarrow VC(s, B) \Rightarrow WP(s, B)$

Verification Condition Generation

- Computed in a manner similar to WP
- Except the rule for while:

$VC(\text{while}_I\ E \text{ do } s, B) =$
$I \wedge (\forall x_1 \ldots x_n.\ I \Rightarrow (E \Rightarrow VC(s, I)) \wedge \neg E \Rightarrow B))$

I holds on entry; I is preserved in an arbitrary iteration; B holds when the loop terminates

- I is the loop invariant (provided externally)
- $x_1, \ldots, x_n$ are all the variables modified in s
- The ∀ is similar to the ∀ in mathematical induction:
  $P(0) \wedge \forall n \in \mathbb{N}.\ P(n) \Rightarrow P(n+1)$

Forward Verification Condition Generation

- Traditionally VC is computed backwards
  - Works well for structured code
- But it can be computed in a forward direction
  - Works even for low-level languages (e.g., assembly language)
  - Uses symbolic evaluation (important technique #2)
  - Has broad applications in program analysis
    - e.g. the PREfix tools works this way

Symbolic Evaluation

- Consider the language:
  x := E | f() | if E goto L | goto L | L: | return | inv E
- The "inv E" instruction is an annotation
  - Says that boolean expression E holds at that point
- Notation: I_k is the instruction at address k

Symbolic Evaluation. The State.

- We set up a symbolic evaluation state:

$\Sigma : Var \cup \{ \mu \} \rightarrow SymbolicExpressions$

$\Sigma(x)$ = the symbolic value of x in state Σ
$\Sigma[x:=E]$ = a new state in which x's value is E

We shall use states also as substitutions:
$\Sigma(E)$ - obtained from E by replacing x with $\Sigma(x)$

Symbolic Evaluation. The Invariants.

- The symbolic evaluator keeps track of the encountered invariants

$Inv \subseteq \{1 \ldots n\}$

- If $k \in Inv$ then
  - I_k is an invariant instruction that we have already executed
- Basic idea: execute an inv instruction only twice:
  - The first time it is encountered
  - And one more time around an arbitrary iteration

Symbolic Evaluation. Rules.

- Define a VC function as an interpreter:

  VC : 1..n × SymbolicState × InvariantState → Predicate

$VC(k, \Sigma, Inv) =$

Value	Condition
$VC(L, \Sigma, Inv)$	if I_k = goto L
$E \Rightarrow VC(L, \Sigma, Inv) \wedge \neg E \Rightarrow VC(k+1, \Sigma, Inv)$	if I_k = if E goto L
$VC(k+1, \Sigma[x:=\Sigma(E)], Inv)$	if I_k = x := E
$\Sigma(Post_{current})$	if I_k = return
$\Sigma(Pre_f) \wedge \forall a_1..a_m.\Sigma'(Post_f) \Rightarrow VC(k+1, \Sigma', Inv)$ (where $y_1, \ldots, y_m$ are modified by f) and $a_1, \ldots, a_m$ are fresh parameters and $\Sigma' = \Sigma[y_1 := a_1, \ldots, y_m := a_m]$	if I_k = f()

Symbolic Evaluation. Invariants.

Two cases when seeing an invariant instruction:

1. We see the invariant for the first time
   - I_k = inv E.
   - $k \notin Inv$
   - Let $\{y_1, \ldots, y_m\}$ = the variables that could be modified on a path from the invariant back to itself
   - Let $a_1, \ldots, a_m$ fresh new symbolic parameters

$VC(k, \Sigma, Inv) =$

$$\Sigma(E) \wedge \forall a_1 \ldots a_m.\ \Sigma'(E) \Rightarrow VC(k+1, \Sigma', Inv \cup \{k\})$$

with $\Sigma' = \Sigma[y_1 := a_1, \ldots, y_m := a_m]$

Symbolic Evaluation. Invariants.

2. We see the invariant for the second time
   - I_k = inv E
   - $k \in Inv$

$VC(k, \Sigma, Inv) = \Sigma(E)$

- Some tools take a more simplistic approach
  - Do not require invariants
  - Iterate through the loop a fixed number of times
  - PREfix (iterates 2 times), versions of ESC (Compaq SRC)
  - Sacrifice completeness for usability

Symbolic Evaluation. Putting it all together

- Let
  - $x_1, \ldots, x_n$ be all the variables and $a_1, \ldots, a_n$ fresh parameters
  - Σ_0 be the state $[x_1 := a_1, \ldots, x_n := a_n]$
  - $\emptyset$ be the empty Inv set
- For all functions f in your program, compute

  $\forall a_1 \ldots a_n.\ \Sigma_0(Pre_f) \Rightarrow$ VC(f_{entry}, \Sigma_0, \empty)
- If all of these predicates are valid then:
  - If you start the program by invoking any f in a state that satisfies Pre_f the program will execute such that
    - At all "inv E" the E holds, and
    - If the function returns then $Post_f$ holds
  - Can be proved w.r.t. a real interpreter (operational semantics)

VC Generation Example

- Consider the program

```
1: I := 0          Precondition: B : bool ∧ A : array(bool, L)
   R := B
3: inv I ≥ 0 ∧ R : bool
   if I ≥ L goto 9
   assert saferd(A + I)
   T := *(A + I)
   I := I + 1
   R := T
   goto 3
9: return R        Postconditon: R : bool
```

VC Generation Example (cont.)

$\forall A.\ \forall B.\ \forall L.\ \forall \mu$
$B : bool \wedge A : array(bool, L) \Rightarrow$
$0 \geq 0 \wedge B : bool \wedge$
$\forall I.\ \forall R.$
$I \geq 0 \wedge R : bool \Rightarrow$
$I \geq L \Rightarrow R : bool$
$\wedge$
$I < L \Rightarrow saferd(A + I) \wedge$
$I + 1 \geq 0 \wedge$
$sel(\mu, A + I) : bool$

- VC contains both proof obligations and assumptions about the control flow

Review

- We have defined weakest preconditions
  - Not always expressible
- Then we defined verification conditions
  - Always expressible
  - Also preconditions but not weakest => loss of completeness
- Next we have to prove the verification conditions
- But first, we'll examine some of their properties

VC and Invariants

- Consider the Hoare triple:
  $\{x \leq 0\}\ \text{while}_I\ x \leq 5\ \text{do}\ x \leftarrow x + 1\ \{x \leq 6\}$
- The VC for this is:
  $x \leq 0 \Rightarrow I(x) \wedge \forall x. (I(x) \Rightarrow (x > 5 \Rightarrow x \leq 6 \wedge x \leq 5 \Rightarrow I(x+1)))$
- Requirements on the invariant:
  - Holds on entry $\forall x.\ x \leq 0 \Rightarrow I(x)$
  - Preserved by the body $\forall x.\ I(x) \wedge x \leq 5 \Rightarrow I(x+1)$
  - Useful $\forall x.\ I(x) \wedge x > 5 \Rightarrow x \leq 6$
- Check that $I(x) = x \leq 6$ works
  - And is the only one that satisfies all constraints

VC Can Be Large

- Consider the sequence of conditionals
  if $x < 0$ then $x \leftarrow -x$; if $x \leq 3$ then x += 3;
  - With the postcondition $P(x)$
- The VC is
  $x < 0 \wedge -x \leq 3 \Rightarrow P(-x + 3) \wedge$
  $x < 0 \wedge -x > 3 \Rightarrow P(-x) \wedge$
  $x \geq 0 \wedge x \leq 3 \Rightarrow P(x + 3) \wedge$
  $x \geq 0 \wedge x > 3 \Rightarrow P(x)$
- There is one conjunct for each path
  => exponential number of paths !
  - Conjuncts for non-feasible paths have un-satisfiable guard !
- Try with $P(x) = x \geq 3$

VC Can Be Large (2)

- VCs are exponential in the size of the source because they attempt relative completeness:
  - It could be that the correctness of the program must be argued independently for each path
- Remark:
  - It is unlikely that the programmer could write a program by considering an exponential number of cases
  - But possible. Any examples?
- Solutions:
  - Allow invariants even in straight-line code
  - Thus do not consider all paths independently !

Invariants in Straight-Line Code

- Purpose: modularize the verification task
- Add the command "after s establish I"
  - Same semantics as c (I is only for verification purposes)
  $VC(\text{after } s \text{ establish } I, P) =_{def} VC(s,I) \wedge \forall x_i.\ I \Rightarrow P$
    - where x_i are the ModifiedVars(s)
- Use when s contains many paths
  after if $x < 0$ then $x \leftarrow -x$ establish $x \geq 0$;
  if $x \leq 3$ then x += 3 { P(x) }
- VC now is (for $P(x) = x \geq 3$)
  $(x < 0 \Rightarrow -x \geq 0) \wedge (x \geq 0 \Rightarrow x \geq 0) \wedge$
  $\forall x.\ x \geq 0 \Rightarrow (x \leq 3 \Rightarrow P(x+3) \wedge x > 3 \Rightarrow P(x))$

Dropping Paths

- In absence of annotations drop some paths
- VC(if E then c_1 else c_2, P) = choose one of
  - $E \Rightarrow VC(c_1, P) \wedge \neg E \Rightarrow VC(c_2, P)$
  - $E \Rightarrow VC(c_1, P)$
  - $\neg E \Rightarrow VC(c_2, P)$
- We sacrifice soundness !
  - No more guarantees but possibly still a good debugging aid
- Remarks:
  - A recent trend is to sacrifice soundness to increase usability
  - The PREfix tool considers only 50 non-cyclic paths through a function (almost at random)

VCGen for Exceptions

- We extend the source language with exceptions without arguments:
  - throw — throws an exception
  - try s_1 handle s_2 — executes s_2 if s_1 throws
- Problem:
  - We have non-local transfer of control
  - What is VC(throw, P) ?
- Solution: use 2 postconditions
  - One for normal termination
  - One for exceptional termination

VCGen for Exceptions (2)

- Define: VC(c, P, Q) is a precondition that makes c either not terminate, or terminate normally with P or throw an exception with Q
- Rules
  - VC(skip, P, Q) = P
  - VC(c_1; c_2, P, Q) = VC(c_1, VC(c_2, P, Q), Q)
  - VC(throw, P, Q) = Q
  - VC(try c_1 handle c_2, P, Q) = VC(c_1, P, VC(c_2, Q, Q))
  - VC(try c_1 finally c_2, P, Q) = ?

Mutable Records - Two Models

- Let r : RECORD f1 : T1; f2 : T2 END
- Records are reference types
- Method 1
  - One "memory" for each record
  - One index constant for each field. We postulate $f1 \neq f2$
  - r.f1 is sel(r,f1) and r.f1 $\leftarrow$ E is r $\leftarrow$ upd(r,f1,E)
- Method 2
  - One "memory" for each field
  - The record address is the index
  - r.f1 is sel(f1,r) and r.f1 $\leftarrow$ E is f1 $\leftarrow$ upd(f1,r,E)

VC as a "Semantic Checksum"

- Weakest preconditions are an expression of the program's semantics:
  - Two equivalent programs have logically equivalent WP
  - No matter how similar their syntax is !
- VC are almost as powerful

VC as a "Semantic Checksum" (2)

- Consider the program below
  - In the context of type checking:

```
x ← 4
x ← x == 5
  assert x : bool
x ← not x
  assert x
```

- High-level type checking is not appropriate here
- The VC is: 4 == 5 : bool $\wedge$ not (4 == 5)
- No confusion because reuse of x with different types

Invariance of VC Across Optimizations

- VC is so good at abstracting syntactic details that it is syntactically preserved by many common optimizations
  - Register allocation, instruction scheduling
  - Common-subexpression elimination, constant and copy prop.
  - Dead code elimination
- We have identical VC whether or not an optimization has been performed
  - Preserves syntactic form, not just semantic meaning !
- This can be used to verify correctness of compiler optimizations (Translation Validation)

VC Characterize a Safe Interpreter

- Consider a fictitious "safe" interpreter
  - As it goes along it performs checks (e.g. saferd, validString)
  - Some of these would actually be hard to implement
- The VC describes <u>all</u> of the checks to be performed
  - Along with their context (assumptions from conditionals)
  - Invariants and pre/postconditions are used to obtain a finite expression (through induction)
- VC is valid $\Rightarrow$ interpreter never fails
  - We enforce same level of "correctness"
  - But better (static + more powerful checks)

VC and Safe Interpreters

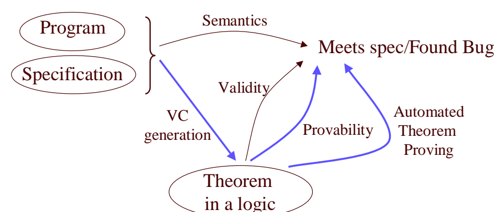
- Essential components of VC's:
  - Conjunction - sequencing of checks
  - Implications - capture flow information (context)
  - Universal quantification
    - To express checking for all input values
    - To aid in formulating induction
  - Literals - express the checks themselves
- So far it looks that only a very small subset of first-order logic suffices

Review

- Verification conditions
  - Capture the semantics of code + specifications
  - Language independent
  - Can be computed backward/forward on structured/unstructured code
  - Can be computed on high-level/low-level code

Next: We start proving VC predicates

Where Are We?



- To discuss:
  - Validity of VCs
  - Provability of VCs
  - Automation of provability (automated theorem proving)

Revisit the Logic

- Recall the we use the following logic:

Goals: $G ::= L \mid true \mid G_1 \wedge G_2 \mid H \Rightarrow G \mid \forall x.\ G$
Hypotheses: $H ::= L \mid true \mid H_1 \wedge H_2$
Literals: $L ::= p(E_1, \ldots, E_k)$
Expressions: $E ::= n \mid f(E_1, \ldots, E_m)$

- This is a subset of FOL
  - Formulas such as "$P_1 \vee P_2$", "$(\forall x.\ P) \Rightarrow Q$" are not (yet) allowed
- This is sufficient for VCGen if:
  - The invariants, preconditions and postcond. are all from H

A Semantic for Our Logic

- Define validity (truth of VC)
  - Each predicate symbol has a meaning: $[\![p]\!] : \mathbb{Z}^k \to \mathbb{B}$
  - Each expression symbol has a meaning: $[\![f]\!] : \mathbb{Z}^n \to \mathbb{Z}$
- We give meaning to each formula:
  - $\models G$ means that the (closed) formula G holds

$\models$ true

$\models G_1 \wedge G_2$ when $\models G_1$ and $\models G_2$

$\models \forall x.G$ when for all $n \in \mathbb{Z}$ we have $\models G[n/x]$

$\models H \Rightarrow G$ when $\models G$ whenever $\models H$

$\models p(E_1, \ldots, E_k)$ when $[\![p]\!]([\![E_1]\!], \ldots, [\![E_n]\!])$ = true

The Theorem Proving Problem

- Write an algorithm "prove" such that:
- If prove(G) = true then $\models G$
  - Soundness, most important
- If $\models G$ then prove(G) = true
  - Completeness, first to sacrifice

A Theorem Prover for our Logic

- We must work symbolically
  - Or otherwise how can we hope to check $\forall n \in \mathbb{Z} \models G[n/x]$?
  - Same trick as in symbolic model checking
- Define the following symbolic "prove" algorithm
  - Prove(H, G) - prove the goal "$H \Rightarrow G$"

Prove(H, true) = true

Prove(H, $G_1 \wedge G_2$) = prove(H, G_1) && prove(H, G_2)

Prove(H, $H_1 \Rightarrow G_2$) = prove($H \wedge H_1$, G_2)

Prove(H, $\forall x. G$) = prove(H, $G[a/x]$) (a is "fresh")

Prove(H, L) = ?

A Theorem Prover for Literals

- So we have reduced the problem to:
  Prove(H, L)
- But H is a conjunction of literals
- Thus we have to prove that $L_1 \wedge \ldots \wedge L_k \Rightarrow L$
- Or equivalently, that $L_1 \wedge \ldots \wedge L_k \wedge \neg L$ is false
- Or equivalently, that $L_1 \wedge \ldots \wedge L_k \wedge \neg L$ is unsatisfiable
  - For any assignment of values to parameters a_i the truth value of the conjunction of literals is false
- Now we can say that
  prove(H, L) = Unsat($H \wedge \neg L$)

How Complete is Our Prover?

- Assume for now that Unsat is sound and complete
- Prove(H, G) is both sound and complete !
  - No search really
  - Goal-directed procedure
  - Very efficient
- Essentially because we use FOL only superficially
- Can we increase the subset of FOL and still maintain these properties ?

Goal Directed Theorem Proving

- We can add disjunction:

$G ::= \text{true} \mid L \mid G_1 \wedge G_2 \mid H \Rightarrow G \mid \forall x.\, G \mid \underline{G_1 \vee G_2}$

Extend prove as follows:

prove(H, $G_1 \vee G_2$) = prove(H, G_1) || prove(H, G_2)

- This introduces a choice point in proof search
  - Called a "disjunctive choice"
  - Backtracking is complete for this choice selection

Goal Directed Theorem Proving (2)

- Now we extend a bit the language of hypotheses
  - Important since this adds flexibility for invariants and specs.

$H ::= L \mid true \mid H_1 \wedge H_2 \mid \exists x.\ H \mid G \Rightarrow H$

- We extend the proved as follows:

prove(H, $\exists x.H_1 \Rightarrow G$) = prove($H \wedge H_1[a/x]$, G) (a fresh)
prove(H, $(G_1 \Rightarrow H_1) \Rightarrow G$) =
    prove(H, G) || (prove($H \wedge H_1$, G) && prove(H, G_1))

  - This adds another choice (clause choice in Prolog) expressed here also as a disjunctive choice
  - Still complete with backtracking

Goal Directed Theorem Proving (3)

- Finally we extend the use of quantifiers:

$G ::= L \mid true \mid G_1 \wedge G_2 \mid H \Rightarrow G \mid \forall x.\ G \mid G_1 \vee G_2 \mid \exists x.\ G$
$H ::= L \mid true \mid H_1 \wedge H_2 \mid G \Rightarrow H \mid \forall x.\ H \mid \exists x.\ H$

- We have now introduced an existential choice
  - Both in "$H \Rightarrow \exists x.\ G$" and "$\forall x.H \Rightarrow G$"
- Existential choices are postponed
  - Introduce unification variables + unification
- Still sound and complete !
- Hereditary Harrop Logic (extension of Horn Logic)

Theories

- Now we turn to proving Unsat($L_1, \ldots, L_k$)
- A theory consists of a:
  - A set of function and predicate symbols (syntax)
  - Definitions for the meaning of these symbols (semantics)
- Example:
  - Symbols: 0, 1, -1, 2, -2, …, +, -, =, < (with the usual meaning)
    - Theory of integers with arithmetic (Presburger arithmetic)

Decision Procedures for Theories

- The Decision Problem:
  - Decide whether a formula in a theory + FOL is true
- Example:
  - Decide whether $\forall x.\ x > 0 \Rightarrow (\exists y.\ x = y + 1)$ in $\{\mathbb{N}, +, =, >\}$
- A theory is decidable when there is an algorithm that solves the decision problem for the theory
  - This algorithm is the decision procedure for the theory

Satisfiability Procedures for Theories

- The Satisfiability Problem
  - Decide whether a conjunction of literals in the theory is satisfiable
  - Factors out the FOL part of the decision problem
- This is what we need to solve in our simple prover

Examples of Theories. Equality.

- The theory of equality with uninterpreted functions
- Symbols: =, ≠, f, g, …
- Axiomatically defined:

$$\frac{}{E = E} \qquad \frac{E_2 = E_1}{E_1 = E_2} \qquad \frac{E_1 = E_2 \quad E_2 = E_3}{E_1 = E_3} \qquad \frac{E_1 = E_2}{f(E_1) = f(E_2)}$$

- Example of a satisfiability problem:

$$g(g(g(x)) = x \wedge g(g(g(g(g(x))))) = x \wedge g(x) \neq x$$

- Satisfiability problem decidable in O(n log n)

Examples of Theories. Presburger Arithmetic.

- The theory of integers with +, -, =, >
- Example of a satisfiability problem:

  $y > 2x + 1 \wedge y + x > 1 \wedge y < 0$

- Satisfiability problem solvable in polynomial time
  - Some of the algorithms are quite simple

Example of Theories. Data Structures.

- Theory of list structures
- Symbols: nil, cons, car, cdr, atom, =
- Example of a satisfiability problem:

  $car(x) = car(y) \wedge cdr(x) = cdr(y) \Rightarrow x = y$

- Based on equality
- Also solvable in $O(n \log n)$
- Very similar to equality constraint solving with destructors