Not Quite Weakest Preconditions

- Recall what we are trying to do:
  - false \( \Rightarrow \) true
  - A 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 \( A: A \rightarrow VC(s, B) \rightarrow WP(s, B) \)

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 tool works this way

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 \( \Sigma \)
  \( \Sigma(x=E) \) = 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

\[ \text{Inv} \subseteq \{1..n\} \]

- If \( k \in \text{Inv} \) 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. Invariants.

Two cases when seeing an invariant instruction:
1. We see the invariant for the first time
   - \( I_k = \text{inv} E \)
   - \( k \in \text{Inv} \)
   - \( y_k \) are the variables that could be modified on a path from the invariant back to itself
   - Let \( a_1, ..., a_n \) be fresh new symbolic parameters

\[
\text{VC}(k, \Sigma, \text{Inv}) = \\
\Sigma(E) \land \forall a_1, ..., a_n. \Sigma(E) = \text{VC}(\text{inv } E, \Sigma, \text{Inv} \cup \{k\})
\]

with \( \Sigma \equiv \Sigma[y_1 \rightarrow a_1, ..., y_n \rightarrow a_n] \)

2. We see the invariant for the second time
   - \( I_k = \text{inv } E \)
   - \( k \in \text{Inv} \)

\[
\text{VC}(k, \Sigma, \text{Inv}) = \Sigma(E)
\]

- Some tools take a more simplistic approach
  - Do not require invariants
  - Iterate through the loop a fixed number of times
  - \text{PRE} \times \text{ITERATE} \times \text{POST} \times \text{PRED} \times \text{POST} \times \text{PRED}
  - Sacrifice completeness for usability

Symbolic Evaluation. Putting it all together

- Let
  - \( y_1, ..., y_n \) be all the variables and \( a_1, ..., a_n \) be fresh parameters
  - \( \Sigma \) be the state \( \{ \Sigma[y_1 = a_1, ..., y_n = a_n] \} \)
  - \( k \) be the empty Inv set

- For all functions \( f \) in your program, compute

\[
\forall a_1, a_2, \Sigma, \text{pre}(E) = \text{VC}(\text{entry}, \Sigma, \text{empty})
\]

- If all of these predicates are valid then:
  - If you start the program by invoking \( f \) in a state that satisfies \( \text{pre} \), the program will execute such that
    - At all \( \text{inv } E \)'s the \( E \) holds, and
    - If the function returns then \( \text{post} \) holds
  - Can be proved w.r.t. a real interpreter (operational semantics)


- Define a VC function as an interpreter:

\[
\begin{align*}
\text{VC} &: \text{Inv } \times \text{SymbolicState } \times \text{InvariantState } \rightarrow \text{Predicate} \\
\text{VC}(\text{inv } E, \Sigma, \text{Inv}) &= \text{if } I_k = \text{gate } L \\
\text{E} &= \text{VC}(\text{inv } E, \Sigma, \text{Inv}) \\
\text{E} &= \text{VC}(\text{inv } E, \Sigma, \text{Inv}) \\
\text{VC}(k+1, \Sigma, \text{Inv}) &= \text{if } I_k = \text{gate } L \\
\text{Post}(\text{pre} \times \text{iter} \times \text{post} \times \text{pre} \times \text{post} \times \text{pre}) &= \text{if } I_k = \text{gate } L
\end{align*}
\]

VC Generation Example

- Consider the program

\[
\begin{align*}
1: & \text{ I = 0 } & \text{ Precondition: } B : \text{ bool } \land A : \text{ array(bool, L)} \\
R : & B \\
3: & \text{ if I > 0 then } R := \text{ bool} \\
\text{ if I = L then } & \text{ go to 9} \\
\text{ assert if } & \text{ } A[\text{ L}] \\
T := & A[\text{ I}] \\
I := & I + 1 \\
R := & T \\
& \text{ go to 3} \\
9: & \text{ return } R & \text{ Postcondition: } R : \text{ bool}
\end{align*}
\]
VC Generation Example (cont.)

\[ \forall A, B, \forall i, i' \in I \]
\[ B : bool \land A : array(bool, L) \Rightarrow \]
\[ 0 \geq 0 \land B : bool \land \]
\[ \forall i, i' \in I \]
\[ i \geq 0 \land i' : bool \Rightarrow \]
\[ i \geq i' \Rightarrow i : bool \land \]
\[ i < i' \Rightarrow safe(A + I) \land \]
\[ i \times 1 \geq 0 \land \]
\[ i \times 1 < I(i+1) \land \]
\[ \text{VC contains both proof obligations and assumptions about the control flow} \]

Invariants

- Consider the Hoare triple:
  \[ \{ x \leq 0 \} \text{while } x < 5 \text{ do } x \leftarrow x + 1 \{ x \leq 6 \} \]

- The VC for this is:
  \[ x < 0 \Rightarrow \neg i(x) \land \forall x, i(x) \Rightarrow (x < 5 \Rightarrow x < 6 \land x < 5 \Rightarrow i(x+1)) \]

- Requirements on the invariant:
  - Holds on entry \[ \forall x, i(x) \Rightarrow (x < 5 \Rightarrow x < 6 \land x < 5 \Rightarrow i(x+1)) \]
  - Preserved by the body \[ \forall x, i(x) \Rightarrow (x < 5 \Rightarrow x < 6 \land x < 5 \Rightarrow i(x+1)) \]
  - Check that \[ i(x) = x \leq 6 \text{ works} \]
  - And is the only one that satisfies all constraints

VC Can Be Large

- Consider the sequence of conditionals
  \[ \text{if } x < 0 \text{ then } x \leftarrow x; \text{ if } x < 5 \text{ then } x \leftarrow 3. \]
  - With the postcondition \( P(x) \)
    - The VC is
      \[ x < 0 \land x < 5 \Rightarrow P(x+2) \land x < 0 \land x < 3 \Rightarrow P(x) \land x < 0 \land x > 3 \Rightarrow P(x) \]
    - There is one conjunct for each path
      \[ = \text{exponential number of paths!} \]
    - Conjuncts for non-feasible paths have un-satisfiable guard!
    - Try with \( P(x) = x \geq 3 \)

Invariants in Straight-Line Code

- Purpose: modularize the verification task
- Add the command "after s establish I" (where \( s \) is the modified state)
- Use \( s \) when \( s \) contains many paths
  \[ \text{if } x < 0 \text{ then } x \leftarrow x \text{ establish } x \geq 0; \]
  \[ \text{if } x < 3 \text{ then } x \leftarrow 3 \{ P(x) \} \]
  \[ \text{VC now is (for } P(x) = x \geq 3 \}
  \[ (x < 0 \land x < 5) 
  \[ (x < 0 \land x < 3 \Rightarrow P(x+2) \land x < 0 \land x > 3 \Rightarrow P(x) \land x < 0 \land x > 3 \Rightarrow P(x) \]
  \[ \text{Thus do not consider all paths independently!} \]
Dropping Paths

- In absence of annotations drop some paths
- VC\(E\) if \(E = C_i\) else \(C_j\) = choose one of
  - \(E \Rightarrow V(C_i, P) \land -E \Rightarrow V(C_j, P)\)
  - \(E \Rightarrow V(C_i, P)\)
  - \(-E \Rightarrow V(C_j, P)\)
- 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 PEPA tool considers only 50 non-cyclic paths through a function (almost at random)

Prof. Nicola CS 294-8 Lecture 10

---

VCGen for Exceptions

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

Prof. Nicola CS 294-8 Lecture 10

---

VCGen for Exceptions (2)

- Define: \(V(c, P, Q)\) is a precondition that makes \(c\) either not terminate, or terminate normally with \(P\) or throw an exception with \(Q\)
- Rules
  \[ \begin{align*}
  V(k,p, P, Q) &= P \\
  V(c_2, c_1, P, Q) &= V(c_1, V(c_2, P, Q), Q) \\
  V(h, m, P, Q) &= Q \\
  V(t y, c_1, handle, c_2, P, Q) &= V(c_2, V(P, V(c_1, Q, Q))) \\
  V(t y, c_1, finally, c_2, P, Q) &= ?
  \end{align*} \]

Prof. Nicola CS 294-8 Lecture 10

---

Mutable Records - Two Models

- Let \(r: RECORD f_1: T_1; f_2: T_2 END\)
- Records are reference types
- Method 1
  - One “memory” for each record
    - One index constant for each field. We postulate \(f_1 \neq f_2\)
      - \(r.f_1.r.f_2 = r.e = r \leftarrow \text{updf}(f_1, E)\)
  - Method 2
    - One “memory” for each field
      - The record address is the index
        - \(r.f_1.r.f_2 = r.e = r \leftarrow \text{updf}(f_1, E)\)

Prof. Nicola CS 294-8 Lecture 10

---

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

Prof. Nicola CS 294-8 Lecture 10

---

VC as a “Semantic Checksum” (2)

- Consider the program below
  - In the context of type checking
    \[
    \begin{align*}
    x & \leftarrow 4 \\
    x & \leftarrow x \gg 5 \\
    \text{assert } & x \\text{ bool} \\
    x & \leftarrow \text{ not } x \\
    \text{assert } & x
    \end{align*}
    \]
  - High-level type checking is not appropriate here
  - The VC is: \(4 \gg 5 : \text{ bool} \land \text{ not } (4 = 5)\)
  - No confusion because reuse of \(x\) with different types

Prof. Nicola CS 294-8 Lecture 10

---
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 propagation
  - 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. assert, validString)
  - Some of these would actually be hard to implement
- The VC describes all 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 ⇔ interpreter never fails
  - We enforce same level of correctness
  - But better (static vs. more powerful checks)

VC and Safe Interpreters

- Essential components of VC:
  - Conjunction + sequencing of checks
  - Implications - capture flow information (context)
  - Universal quantification
    - To extend checking for all input values
    - To hide doing the 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 compiled backward/forward on structured/unstructured code
  - Can be compiled on high-level/low-level code

Next: We start proving VC predicates

Where Are We?

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

Revisit the Logic

- Recall the we use the following logic:
  - Goals: \( G ::= L \mid \text{true} \mid G_1 \land G_2 \mid H \Rightarrow G \mid \forall x. G \)
  - Hypotheses: \( H ::= L \mid \text{true} \mid H_1 \land H_2 \)
  - Literals: \( L ::= p(E_1, \ldots, E_n) \)
  - Expressions: \( E ::= n \mid f(E_1, \ldots, E_n) \)
- This is a subset of FOL
  - Formulas such as \( \phi \land \psi \), \( (x \times P) \Rightarrow Q \) are not (yet) allowed
- This is sufficient for VGen 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 \([ p ] : \mathcal{P} \rightarrow \mathbb{B}\)
  - Each expression symbol has a meaning \([ f ] : \mathcal{P}^n \rightarrow \mathbb{B}\)
- We give meaning to each formula:
  \(\models G\) means that the (closed) formula \(G\) holds
  \(\models G\ if \ G_1 \land G_2\) 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 \(\models H\)
  \(\models p(E_1, ..., E_n)\) when \(\models [p(E_1), ..., E_n] = true\)

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
  \(\text{Prove}(H, \text{false}) = \text{false}\)
  \(\text{Prove}(H, G_1 \land G_2) = \text{Prove}(H, G_1) \land \text{Prove}(H, G_2)\)
  \(\text{Prove}(H, H_1 \Rightarrow G_1) = \text{Prove}(H, G_1) \land \text{Prove}(H, H_2)\)
  \(\text{Prove}(H, \forall x. G) = \text{Prove}(H, G(a/x)) (a \text{ is "fresh"})\)
  \(\text{Prove}(H, \text{true}) = \text{true}\)

Theorem Proving Problem

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

A Theorem Prover for Literals

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

How Complete is Our Prover?

- Assume for now that Unsat is sound and complete
- \(\text{Prove}(H, G)\) 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:
  \(G := \text{true} \lor L \lor \forall x. G \lor G_1 \lor G_2\)
- Extend prove as follows:
  \(\text{prove}(H, G_1 \lor G_2) \Rightarrow \text{prove}(H, G_1) \lor \text{prove}(H, G_2)\)
- 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.
  \[ H ::= L \mid \text{true} \mid H_1 \land H_2 \mid \exists x. H \mid G \Rightarrow H \]

- We extend the proved as follows:
  \[ \text{prove}(H, \exists x. H \Rightarrow G) = \text{prove}(H \land H([a/x], G)) \]
  (a fresh)
  \[ \text{prove}(H, (G_1 \Rightarrow H_1) \Rightarrow G) = \]
  \[ \text{prove}(H, G) \lor (\text{prove}(H_1 \land H_2, G) \Rightarrow \text{prove}(H, G)) \]
  - This adds another choice (clause choice in Prolog) expressed
    here also as a disjunctive choice
  - Still complete with backtracking

Theory

- Now we turn to proving Unsat(L_1, ..., L_n)

- 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, ... x, +, = (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 \( \forall x. x > 0 \Rightarrow (3y. x < y + 1) \) in \( \{+, \times, <\} \)

- 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:
  \[
  \begin{align*}
  E &= E \\
  E_1 = E_2 \\
  E_1 + E_2 &= E_1 + E_2 \\
  E_1 = E_2 = E_1 \\
  f(E_1) + f(E_2) &= f(E_2) + f(E_1)
  \end{align*}
  \]

- Example of a satisfiability problem:
  \[ g(g(g(x))) = x \land g(g(g(g(g(x))))) = x \land g(x) = x \]

- Satisfiability problem decidable in \( O(n \log n) \)
Examples of Theories. Presburger Arithmetic.

- The theory of integers with $\times$, $-\times$, $\leq$

- Example of a satisfiability problem:
  \[ y > 2x + 1 \land y + x > 1 \land y < 0 \]

- 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, $\approx$

- Example of a satisfiability problem:
  \[ \text{car}(x) = \text{car}(y) \land \text{cdr}(x) = \text{cdr}(y) \Rightarrow x = y \]

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