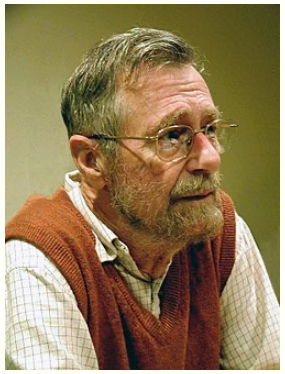
软件分析与架构设计

程序验证

何冬杰

重庆大学

关于程序验证：名人名言



"Program testing can be used to show the presence of bugs, but never to show their absence!"

Edsger Dijkstra,1930-2002, 荷兰计算机科学家,1972年图灵奖

C.A.R. Hoare, 1934--, 英国计算机科学家,1980年图灵奖

"It has been found a serious problem to define these languages[ALGOL, FORTRAN, COBOL] with sufficient rigor to ensure compatibility among all implementations. one way to achieve this would be to insist that all implementations of the language shall satisfy the axioms and rules of inference which underlie proofs of properties of programs expressed in the language. In effect, this is equivalent to accepting the axioms and rules of inference as the ultimately definitive specification of the meaning of the language."

"Thus the practice of proving programs would seem to lead to solution of three of the most pressing problems in software and programming, namely, **reliability**, **documentation**, and **compatibility**. However, program proving, certainly at present, will be difficult even for programmers of high caliber; and may be applicable only to quite simple program designs."

关于程序验证：History

- **Program verification is almost as old as programming (e.g., “Checking a Large Routine”, Turing 1949)**
- **In the late ’60s, Floyd had rules for flow-charts and Hoare had rules for structured languages**
- **Since then, there have been axiomatic semantics for substantial languages, and many applications**
  - Program verifiers (70s and 80s)
  - PREfix: Symbolic execution for bug hunting (WinXP)
  - Software validation tools
  - Malware detection
  - Automatic test generation

Semantics Review

- **Operational semantics**
  - relatively simple, many flavors
  - adequate guide for an implementation of the language
  - not compositional
- **Denotational semantics**
  - Mathematical, Canonical and Compositional
- **Neither is good for showing program correctness**
  - Operational semantics requires running the code
  - Denotational semantics requires complex calculations
- **Need a semantics for arguing program correctness**

公理语义（Axiomatic Semantics）

- **An axiomatic semantics consists of**
  - A language for making assertions about programs
  - Rules for establishing when assertions hold
- **Typical assertions (or properties)**
  - This program terminates
  - If this program terminates, the variables x and y have the same value throughout the execution of the program
  - The array accesses are within the array bounds
- **Some typical languages of assertions**
  - **First-order logic**
  - Other logics (temporal, linear)
  - Special-purpose specification languages (Z, Larch, JML)

公理语义（Axiomatic Semantics）

- **Describes properties of program state**
  - **State:** a function σ from variables to values
    - E.g., program with 3 variables x, y, z: $\sigma(x) = 9, \sigma(y) = 5, \sigma(z) = 2$
    - For simplicity, we only consider integer variables σ: $V \rightarrow \{0, -1, 1, -2, 2, \cdots\}$
  - **Sets of States**:
    - E.g., "$x = 1, y = 2, z = 1$ *or* $x = 1, y = 2, z = 2$ *or* $x = 1, y = 2, z = 3$"
- **Assertions are specified through first-order logic**
  - E.g., $x = 1 \Lambda\ y = 2 \Lambda\ 1 \leq z \leq 3$
  - An assertion **p** represents the set of states that satisfy the assertion
    - *We will write **{p}** to denote this set of states*
- **Concerned with constructing proofs for such properties**
  - Axiomatic semantics is a **set of rules for constructing proofs**
  - Should be able to prove all true statements about the program, and not be able to prove any false statements

First-Order Logic

- **Terms**
  - If x is a variable, x is a term
  - If n is an integer constant, n is a term
  - If t_1 and t_2 are terms, so are $t_1 + t_2$, $t_1 - t_2$, …
- **Formulas**
  - **true** and **false**
  - $t_1 < t_2$ and $t_1 = t_2$ for terms t_1 and t_2
  - $f_1 \wedge f_2$, $f_1 \vee f_2$, $\neg f_1$ for formulas f_1,f_2
  - $\exists x.f$ and $\forall x.f$ for a formula f
- **Use of first-order logic**
  - Variables from the program as terms
  - Operations from the programming language: e.g. +, -, …
  - The usual suspects from first-order logic: true, false, $\vee, \wedge, \neg, \exists, \forall$

When does a State Satisfy an Assertion?

- **Notation $\sigma \vDash A$ means A holds in the given state σ**
  - Well-defined when σ is defined on all variables occurring in A
- **Formal definition**

$$
\begin{array}{lcll}
\sigma & \models & \mathsf{true} & \text{always} \\
\sigma & \models & e_1 = e_2 & \text{iff } \langle \sigma, e_1 \rangle \downarrow n_1 \text{ and } \langle \sigma, e_2 \rangle \downarrow n_2 \text{ and } n_1 = n_2 \\
\sigma & \models & e_1 \leq e_2 & \text{iff } \langle \sigma, e_1 \rangle \downarrow n_1 \text{ and } \langle \sigma, e_2 \rangle \downarrow n_2 \text{ and } n_1 \leq n_2 \\
\sigma & \models & A_1 \wedge A_2 & \text{iff } \sigma \models A_1 \text{ and } \sigma \models A_2 \\
\sigma & \models & A_1 \vee A_2 & \text{iff } \sigma \models A_1 \text{ or } \sigma \models A_2 \\
\sigma & \models & A_1 => A_2 & \text{iff } \sigma \models A_1 \text{ implies } \sigma \models A_2 \\
\sigma & \models & \forall x.A & \text{iff } \forall n \in \mathbb{Z} \,.\, \sigma[x := n] \models A \\
\sigma & \models & \exists x.A & \text{iff } \exists n \in \mathbb{Z} \,.\, \sigma[x := n] \models A
\end{array}
$$

Notation: Substitution

- **Free vs. Bound Variable Occurrences**
  - An occurrence of a variable x is **bound** if it is in the scope of $\exists x$ or $\forall x$
  - An occurrence is **free** if it is not bound
  - E.g., $\exists i.\, k = i * j$: k and j are free, i is bound
- $p[e/x]$: **(other notations:** $\mathrm{p}_{\mathrm{e}}^{\mathrm{x}}$, $\mathrm{p}[\mathrm{x} := \mathrm{e}]$**)**
  - the assertion p with all free occurrences of x replaced by e
  - to avoid conflicts, may have to rename some quantified variables

When does a State Satisfy an Assertion?

- **$\{\ p\ \}$ denotes the set of states that satisfy assertion**
- $\{p \vee q\} = \{p\} \cup \{q\}$ **;** $\{p \wedge q\} = \{p\} \cap \{q\}$
- $\{\neg p\} = U\ -\ \{p\}$
  - (U is the universal set)
- **Suppose $p \Rightarrow q$ is true w.r.t. standard mathematics**
  - then $\{p\} \subseteq \{q\}$
  - $x = 2 \wedge y = 3 \Rightarrow x = 2$, so $\{\ x = 2\ \wedge y = 3\ \} \subseteq \{\ x = 2\ \}$

Examples of Assertions

- **Three program variables:**
  - $\{ x = 1 \Lambda y = 2 \}$: infinite set
  - $\{ x = 1 \Lambda 1 \leq y \leq 5 \}$: infinite set
  - $\{ x = y + z \}$: all states s.t. $\sigma(x) = \sigma(y) + \sigma(z)$
  - $\{ x = x \}$: the set of all states
  - $\{ true \}$: the set of all states
  - $\{ x \neq x \}$: the empty set
  - $\{ false \}$: the empty set

Hoare Triples

- **Hoare Triples: By C. A. R. Hoare (Tony Hoare)**
  - Abstract description of constraints that any implementation of the program must satisfy
  - Says nothing about how these relationships will be achieved (in contrast to operational semantics)
- **Hoare Triple $\{p\}\ S\ \{q\}$:**
  - S is a piece of code (program fragment)
  - p and q are assertions
  - p: pre-condition, q: post-condition
  - Meaning:
    - If we start executing S from any state σ that satisfies p, and if S terminates, then the resulting state σ', satisfies q
    - Formally, $\vDash \{p\}\ S\ \{q\}$: $\forall \sigma. \exists \sigma'. (\sigma \vDash p \wedge \langle \sigma, S \rangle \downarrow \sigma') \rightarrow \sigma' \vDash q$
  - the relationship between p and q captures the essence of the semantics of S

Valid Results

- **Hoare triple $\{p\}\ S\ \{q\}$ can be seen as the result of proofs**
- **A result $\{p\}\ S\ \{q\}$ is valid iff for every state σ:**
  - if σ satisfies p (i.e., σ belongs to set $\{p\}$)
  - and the execution of S starting in σ terminates in state σ',
  - then σ', satisfies q (i.e., σ', belongs to set $\{q\}$)
  - Is {false} S {q} valid?
- **Examples**

{ x=1 ^ y=1 } skip { x=1 }	Valid
{ x=1 } skip { x=1 ^ y=1 }	Invalid
{ x=1 } skip { x=1 V y=1 }	Valid
{ x=1 V y=1 } skip { x=1 }	Invalid
{ x=1 } skip { true }	Valid
{ x=1 } skip { false }	Invalid

{ x=1 ^ y=2 } x := x+1 { x ≥ 2 }	Valid
{ x=1 ^ y=2 } x := x+1 { x=y }	Valid
{ x=0 } while x<10 do x:=x+1 { x=10 }	Valid
{ x<0 } while x<10 do x:=x+1 { x=10 }	Valid
{ x≥0 } while x<10 do x:=x+1 { x=10 }	Invalid
{ x≥0 } while x<10 do x:=x+1 { x≥10 }	Valid

Termination

- **A result $\{p\}\ S\ \{q\}$ is valid iff … S starting in σ terminates …**
- **What if S does not terminate?**
  - We are only concerned with initial states for which S terminates
- **All of the following results are valid:**
  - { x=3 } while x≠10 do x:=x+1 { x=10 }
  - { x≥11 } while x≠10 do x:=x+1 { x=10 } // cannot terminate
  - { true } while x≠10 do x:=x+1 { x=10 }

Proof System

- **Valid result:**
  - The result is valid w.r.t an operational model
- **Proof System:**
  - Derive the valid results without using operational reasoning but axioms and rules, *Purely formally*
- **Proof = set of applications of instances of inference rules**
  - Starting from one or more axioms
  - Conclusions are subsequently used as premises
  - The conclusion of the last production is proved (derived) by the proof
- **If a proof exists, the result is provable (derivable)**

Properties of a Proof System

- **Properties of a proof system (axiomatic semantics) A**
  - w.r.t. an operational model M
  - Soundness (consistency): every result we can prove (derive) in A is valid in M
  - Completeness: every result that is valid in M can be derived (proven) in A
- **Gödel's *Incompleteness Theorem***
  - Any consistent formal system that is powerful enough to describe basic arithmetic contains statements that are true, but cannot be proven within that system.
  - In other words, no such system can be both complete (able to prove all truths) and consistent (free of contradictions).
  - Let us define a relatively complete proof system for SIMP

Review the SIMP language

$$
\begin{array}{rcl}
S & ::= & x := a \\
 & | & \text{skip} \\
 & | & S_1;\ S_2 \\
 & | & \text{if } b \text{ then } S_1 \text{ else } S_2 \\
 & | & \text{while } b \text{ do } S
\end{array}
\qquad
\begin{array}{rcl}
b & ::= & \text{true} \\
 & | & \text{false} \\
 & | & \text{not } b \\
 & | & b_1\ op_b\ b_2 \\
 & | & a_1\ op_r\ a_2
\end{array}
\qquad
\begin{array}{rcl}
a & ::= & x \\
 & | & n \\
 & | & a_1\ op_a\ a_2
\end{array}
\qquad
\begin{array}{rcl}
op_b & ::= & \text{and} \mid \text{or} \\
op_r & ::= & < \mid \leqslant \mid = \\
 & | & > \mid \geqslant \\
op_a & ::= & + \mid - \mid * \mid /
\end{array}
$$

- x: integer-valued variables
- expressions are deterministic (for now)
- expression evaluation does not cause error

Proof System for SIMP

- p is an arbitrary assertion
- **Skip axiom:** $\{ p \}\ skip\ \{ p \}$
- **Assignment Axiom:** $\{ p[e/x] \}\ x := e\ \{ p \}$
  - Example:
    - {x+1=y+z} x:= x+1 {x = y + z}
    - {y+z > 0} x:= y+z {x > 0}
- **Inference Rule of Composition:**
  - example:

$$\frac{\{p\}\ S_1\{q\} \qquad \{q\}\ S_2\{r\}}{\{p\}\ S_1; S_2\{r\}}$$

{x+1=y+z} skip {x+1=y+z} {x+1=y+z} x:=x+1 {x=y+z }

{x+1=y+z} skip; x:=x+1 {x=y+z }

Proof System for SIMP

weakening its postcondition $q' \Leftarrow q$
strengthening its precondition $p' \Rightarrow p$

❑Inference Rule of Consequence

$$\frac{p' \rightarrow p \quad \{p\}\, S\, \{q\} \quad q \rightarrow q'}{\{\, p' \,\}\, S\, \{\, q' \,\}}$$

- Recall that x → y means $\{x\} \subseteq \{y\}$
- Example:

$$\frac{x{=}1 \wedge y{=}2 \;\rightarrow\; x{=}1 \qquad \{\, x{=}1 \,\}\ \text{skip}\ \{\, x{=}1 \,\}}{\{\, x{=}1 \wedge y{=}2 \,\}\ \text{skip}\ \{\, x{=}1 \,\}}$$

❑Exercise

- Show that the following rule will make the proof system inconsistent (unsound) i.e. it will be possible to prove something that is not operationally valid

$$\frac{\{\, p \,\}\, S\, \{\, q \,\} \quad q' \rightarrow q}{\{\, p \,\}\, S\, \{\, q' \,\}}$$

Proof System for SIMP

❑If-Then-Else Rule:

$$\frac{\{p \wedge b\}\ S_1\ \{q\} \quad \{p \wedge \neg b\}\ S_2\ \{q\}}{\{p\}\ \textit{if } b \textit{ then } S_1 \textit{ else } S_2\ \{q\}}$$

➢Example:

$$\frac{\dfrac{y = 1 \wedge y = 1 \Rightarrow 1 = 1 \quad \{1 = 1\}\ x := 1\ \{x = 1\}}{\{y = 1 \wedge y = 1\}\ x := 1\ \{x = 1\}} \quad \dfrac{y = 1 \wedge \neg(y = 1) \Rightarrow 2 = 1 \quad \{2 = 1\}\ x := 2\ \{x = 1\}}{\{y = 1 \wedge \neg(y = 1)\}\ x := 2\ \{x = 1\}}}{\{y = 1\}\ \textit{if } y = 1 \textit{ then } x := 1 \textit{ else } x := 2\ \{x = 1\}}$$

❑Why not simply

$$\frac{\{p\}\ S_1\ \{q\} \quad \{p\}\ S_2\ \{q\}}{\{p\}\ \textit{if } b \textit{ then } S_1 \textit{ else } S_2\ \{q\}}$$

➢Work for $\{true\}\ \textit{if } y = 1 \textit{ then } x := 1 \textit{ else } x := 2\ \{x = 1 \vee x = 2\}$

➢not work for $\{y = 1\}\ \textit{if } y = 1 \textit{ then } x := 1 \textit{ else } x := 2\ \{x = 1\}$

- { y=1 } x:=2 { x=1 } cannot be proven using axioms and rules
- the proof system becomes incomplete i.e. it becomes impossible to prove something that is, in fact, operationally valid

Proof System for SIMP

- **Problem:** proving {P} while B do S end {Q} for arbitrary P and Q is undecidable
  - Need to encode the knowledge that went into constructing the loop
- **For each loop, we need an invariant I, an assertion that must be satisfied by**
  - the state at the beginning of the loop
  - the state at the end of each iteration
  - the state immediately after the loop exits
  - **Finding a loop invariant is the hard part**
- **While Loop Rule**

$$\frac{\{I \wedge b\}\ S\ \{I\}}{\{I\}\ \textit{while } b \textit{ do } S \textit{ end}\ \{I \wedge \neg b\}}$$

or

$$\frac{p \rightarrow I \quad \{I \wedge b\}\ S\ \{I\} \quad (I \wedge \neg b) \rightarrow q}{\{p\}\ \textit{while } b \textit{ do } S \textit{ end}\ \{q\}}$$

程序验证举例 I： Division

$(x \geq 0) \wedge (y > 0) \rightarrow (x = 0 * y + x) \wedge (0 \leq x)$

Obviously true (2)

❑Proof： q: quotient； r: remainder

$\{ (x \geq 0) \wedge (y > 0) \}$
q := 0;
r := x;
while (r - y) ≥ 0 do
q := q + 1;
r := r - y
end
$\{ (x = q * y + r) \wedge (0 \leq r < y) \}$

(1) 确定不变式

$\{ (x \geq 0) \wedge (y > 0) \}$
q := 0;
r := x;
$\{ (x = q * y + r) \wedge (0 \leq r) \}$
while (r - y) ≥ 0 do
q := q + 1;
r := r - y
end
$\{ (x = q * y + r) \wedge (0 \leq r) \wedge (r - y < 0) \}$

$(I \wedge \neg b) \rightarrow q$ (5)

(3)

$\{ I \wedge b \}\ S\ \{ I \}$

$(x = q * y + r) \wedge (0 \leq r) \wedge (r - y < 0) \rightarrow (x = q * y + r) \wedge (0 \leq r < y)$

Trivially true

$\{ (x = q * y + r) \wedge (0 \leq r) \wedge (r - y \geq 0) \}$
q := q + 1;
r := r - y
$\{ (x = q * y + r) \wedge (0 \leq r) \}$

(4)

$(x = q * y + r) \wedge (0 \leq r) \wedge (r - y \geq 0)$
$\rightarrow (x = (q + 1) * y + r - y) \wedge (r - y \geq 0)$

Can be proved with **simple arithmetic**

程序验证举例 II：Fibonacci Numbers

❑**Proof:** fib(1) = 1, fib(2) = 1, fib(i+1) = fib(i) + fib(i-1) (i >2)

$\{ n > 0 \}$
i := n;
f := 1;
h := 1;
while i > 1 do
h := h + f;
f := h - f;
i := i - 1
end
$\{ f = fib(n) \}$

- **Step 1:** Invariant **I** = f=fib(n-i+1) ∧ h=fib(n-i+2) ∧ i>0
- **Step 2:** { **n > 0** } i := n; f := 1; h := 1; {**I**}
  - $n > 0 \rightarrow 1 = fib(n - n + 1) \wedge 1 = fib(n - n + 2) \wedge n > 0$
- **Step 3:** { I ∧ b } S { I }
  - { I ∧ i > 1 } h := h + f; f := h - f; i := i – 1 {I}
  - { f=fib(n-i+1) ∧ h=fib(n-i+2) ∧ i>0 ∧ i>1 } →{ h+f-f=fib(n-(i-1)+1) ∧ h+f=fib(n-(i-1)+2) ∧ (i-1)>0 }
- **Step 4:** $(I \wedge \neg b) \rightarrow q$
  - $(I \wedge i \leq 1) \rightarrow f = fib(n)$
  - { f=fib(n-i+1) ∧ h=fib(n-i+2) ∧ i > 0 ∧ i≤1 } → f=fib(n)

Proof System for SIMP

- **The proof system for SIMP is relatively complete**
  - Relatively: the logic of pre/post-conditions must be complete
  - Anything that is operationally valid can be proven
- **Why is the proof system complete?**
  - Gödel's *Incompleteness Theorem*
    - A sound and complete system does not exist
  - This system:
    - Do not attempt to prove all truths of arithmetic
    - Rely on arithmetic externally as an oracle

weakest liberal preconditions (wlp)

- **Principle: predicate calculus**
  - calculus to derive preconditions from postconditions
  - order and mechanize the search for intermediate assertions
    - easier to go backwards, mainly due to assignments
  - "liberal" means that we do not care about termination and errors
- wlp: $(\text{prog} \times \text{Prop}) \rightarrow \text{Prop}$
  - wlp(c, P) is the weakest, i.e. most general, precondition ensuring that {wlp(c, P)} c {P} is a Hoare triple
    - greatest state set that ensures that the computation ends up in P
  - Formally: $\{P\}\ c\ \{Q\} \iff (P \Rightarrow wlp(c, Q))$
- **Example:**

wlp(X := X + 1; X = 1) = (X = 0)
wlp(**while** X < 0 X := X + 1; X ≥ 0) = true
wlp(**while** X ≠ 0 X := X + 1; X ≥ 0) = true

A calculus for wlp

- **wlp is defined by induction on the syntax of programs**
  - $\mathrm{wlp}(\boldsymbol{skip}, \mathrm{P}) \stackrel{\mathrm{def}}{=} \mathrm{P}$
  - $\mathrm{wlp}(\boldsymbol{fail}, \mathrm{P}) \stackrel{\mathrm{def}}{=} \mathrm{true}$
  - $\mathrm{wlp}(\mathrm{x} := \mathrm{e}, \mathrm{P}) \stackrel{\mathrm{def}}{=} \mathrm{P}[\mathrm{e}/\mathrm{X}]$
  - $\mathrm{wlp}(\mathrm{s}; \mathrm{t}, \mathrm{P}) \stackrel{\mathrm{def}}{=} \mathrm{wlp}(\mathrm{s}, \mathrm{wlp}(\mathrm{t}, \mathrm{P}))$ (backwards)
  - $\mathrm{wlp}(\boldsymbol{if}\ \mathrm{e}\ \boldsymbol{then}\ \mathrm{s}\ \boldsymbol{else}\ \mathrm{t}, \mathrm{P}) \stackrel{\mathrm{def}}{=} (\mathrm{e} \rightarrow \mathrm{wlp}(\mathrm{s}, \mathrm{P})) \wedge (\neg \mathrm{e} \rightarrow \mathrm{wlp}(\mathrm{t}, \mathrm{P}))$
  - $\mathrm{wlp}(\boldsymbol{while}\ \mathrm{e}\ \boldsymbol{do}\ \mathrm{s}, \mathrm{P}) \stackrel{\mathrm{def}}{=} \mathrm{I} \wedge ((\mathrm{e} \wedge \mathrm{I}) \rightarrow \mathrm{wlp}(\mathrm{s}, \mathrm{I})) \wedge ((\neg \mathrm{e} \wedge \mathrm{I}) \rightarrow \mathrm{P})$
    - While loops require providing an invariant predicate I
- Example:

$$\mathrm{wlp}(\mathrm{if}\ \mathrm{X} < 0\ \mathrm{then}\ \mathrm{Y} := -\mathrm{X}\ \mathrm{else}\ \mathrm{Y} := \mathrm{X}, \mathrm{Y} \geq 10)$$
$$= (\mathrm{X} < 0 \rightarrow \mathrm{wlp}(\mathrm{Y} := -\mathrm{X}, \mathrm{Y} \geq 10)) \wedge (\mathrm{X} \geq 0 \rightarrow \mathrm{wlp}(\mathrm{Y} := \mathrm{X}, \mathrm{Y} \geq 10))$$
$$= (\mathrm{X} < 0 \rightarrow -\mathrm{X} \geq 10) \wedge (\mathrm{X} \geq 0 \rightarrow \mathrm{X} \geq 10) = \mathrm{X} \geq 10 \vee \mathrm{X} \leq -10$$

注：$\mathrm{e} \rightarrow \mathrm{Q}$ is equivalent to $\mathrm{Q} \vee \neg \mathrm{e}$

wlp generates complex formulas, it is important to simplify them from time to time

Properties of wlp

- **Excluded miracle:** $\mathrm{wlp}(c, \mathrm{false}) \equiv \mathrm{false}$
- **Monotonicity:** $\text{if } P \rightarrow Q, \text{then } \mathrm{wlp}(c, P) \rightarrow \mathrm{wlp}(c, Q)$
- **Distributivity:**
  - $\mathrm{wlp}(c, P) \wedge \mathrm{wlp}(c, Q) \equiv \mathrm{wlp}(c, P \wedge Q)$
  - $\mathrm{wlp}(c, P) \vee \mathrm{wlp}(c, Q) \equiv \mathrm{wlp}(c, P \vee Q)$
- **Duality:** $(P \rightarrow \mathrm{wlp}(c, Q)) \Leftrightarrow (\mathrm{slp}(P, c) \rightarrow Q)$
- **Strongest liberal postconditions:** $\mathrm{slp}: (\mathrm{Prop} \times \mathrm{prog}) \rightarrow \mathrm{Prop}$
  - Allows **forward** reasoning
  - Strongest postcondition: $\{P\}c\{Q\} \Leftrightarrow \mathrm{slp}(P, c) \rightarrow Q$
  - Liberal: Does not care about non-termination

Calculus for slp

- $\mathrm{slp}(P, \mathrm{skip}) \stackrel{\mathrm{def}}{=} P$
- $\mathrm{slp}(P, \mathrm{fail}) \stackrel{\mathrm{def}}{=} \mathrm{false}$
- $\mathrm{slp}(P, X := e) \stackrel{\mathrm{def}}{=} \exists v\colon\ P[v/X] \wedge X = e[v/X]$
- $\mathrm{slp}(P, s; t) \stackrel{\mathrm{def}}{=} \mathrm{slp}(\mathrm{slp}(P, s), t)$
- $\mathrm{slp}(P, \mathrm{if}\ e\ \mathrm{then}\ s\ \mathrm{else}\ t) \stackrel{\mathrm{def}}{=} \mathrm{slp}(P \wedge e, s) \vee \mathrm{slp}(P \wedge \neg e, t)$
- $\mathrm{slp}(P, \mathrm{while}\ e\ \mathrm{do}\ s) \stackrel{\mathrm{def}}{=} (P \rightarrow I) \wedge (\mathrm{slp}(I \wedge e, s) \rightarrow I) \wedge (\neg e \wedge I)$
  - I is the loop invariant

Program Verification

- **Given an axiomatic semantics, prove {p} S {q} and/or [p] S [q]**
  - S is a program fragment
  - p is something we can guarantee
  - q is something we want S to achieve
  - Need to find loop invariants
  - Need to find termination function for proving total correctness
- **Generate Verification Conditions:**
  - generate mechanically logic formulas ensuring the correctness
    - wlp, slp calculus
  - reduction to a mathematical problem, no longer any reference to a program
  - use an automatic SAT/SMT solver to prove (discharge) the formulas or an interactive theorem prover
- **If we find a proof, S is correct**
- **A counter-example uncovers a bug**

Verification Condition Generation (vcg)

- **Idea: propagates postconditions backwards and accumulates verification conditions from loops**
- $\mathrm{vcg_p}: \mathrm{prog} \rightarrow \boldsymbol{\mathcal{P}}(\mathrm{Prop})$
  - $\mathrm{vcg_p}(\{P\}c\{Q\}) \stackrel{\mathrm{def}}{=} \mathrm{let}\ (R, C) = \mathrm{vcg_s}(c, Q)\ \mathrm{in}\ C \cup \{P \rightarrow R\}$
  - C is the accumulated verification conditions from loops
- $\mathrm{vcg_s}: (\mathrm{stmt} \times \mathrm{Prop}) \rightarrow (\mathrm{Prop} \times \boldsymbol{\mathcal{P}}(\mathrm{Prop}))$
  - $\mathrm{vcg_s}(\mathrm{skip}, Q) \stackrel{\mathrm{def}}{=} (Q, \emptyset)$
  - $\mathrm{vcg_s}(X := e, Q) \stackrel{\mathrm{def}}{=} (Q[e/X], \emptyset)$
  - $\mathrm{vcg_s}(s; t, Q) \stackrel{\mathrm{def}}{=} \mathrm{let}\ (R, C) = \mathrm{vcg_s}(t, Q)\ \mathrm{in}\ \mathrm{let}\ (P, D) = \mathrm{vcg_s}(s, R)\ \mathrm{in}\ (P, C \cup D)$
  - $\mathrm{vcg_s}(\mathrm{if}\ e\ \mathrm{then}\ s\ \mathrm{else}\ t, Q) \stackrel{\mathrm{def}}{=} \mathrm{let}\ (S, C) = \mathrm{vcg_s}(s, Q)\ \mathrm{in}\ \mathrm{let}\ (T, D) = \mathrm{vcg_s}(t, Q)\ \mathrm{in}\ ((e \rightarrow S) \wedge (\neg e \rightarrow T), C \cup D)$
  - $\mathrm{vcg_s}(\mathrm{while}\ \{I\}\ e\ \mathrm{do}\ s, Q) \stackrel{\mathrm{def}}{=} \mathrm{let}\ (R, C) = \mathrm{vcg_s}(s, I)\ \mathrm{in}\ (I, C \cup \{(I \wedge e) \rightarrow R, (I \wedge \neg e) \rightarrow Q\})$

Verification Condition Generation

- **Example:**
  - consider the following Hoare triple

$$\{N \geq 0\} \quad X \leftarrow 1; I \leftarrow 0;$$
$$\textbf{while } \{X = 2^I \wedge 0 \leq I \leq N\}\ I < N \textbf{ do}$$
$$(X \leftarrow 2X;\ I \leftarrow I + 1)$$
$$\{X = 2^N\}$$

  - We get three verification conditions

$$C_1 \stackrel{\text{def}}{=} (X = 2^I \wedge 0 \leq I \leq N) \wedge I \geq N \Rightarrow X = 2^N$$

$$C_2 \stackrel{\text{def}}{=} (X = 2^I \wedge 0 \leq I \leq N) \wedge I < N \Rightarrow 2X = 2^{I+1} \wedge 0 \leq I + 1 \leq N$$
$$(\text{from } (X = 2^I \wedge 0 \leq I \leq N)[I + 1/I, 2X/X])$$

$$C_3 \stackrel{\text{def}}{=} N \geq 0 \Rightarrow 1 = 2^0 \wedge 0 \leq 0 \leq N$$
$$(\text{from } (X = 2^I \wedge 0 \leq I \leq N)[0/I, 1/X])$$

  - Now, we can check/prove these conditions using SMT/SAT or an interactive theorem prover (e.g., rocq)

Summary

- **Axiom Semantics: describes properties of program state**
- **Hoare Logic:**
  - allows us to reason about program correctness
  - Annotations are required
    - Loop invariants
    - Contracts: preconditions and postconditions
  - Verification can be reduced to proofs of simple logic statements
    - Verification conditions must be proven
    - SMT/SAT solvers or interactive theorem prover
- **Proving has not replaced testing and debugging (and praying)**
- **Applications of axiomatic semantics**
  - Proving the correctness of algorithms (or finding bugs)
  - Proving the correctness of hardware descriptions (or finding bugs)
  - "extended static checking" (e.g., checking array bounds)
  - Documentation of programs and interfaces

(Optional)作业：

❑Prove the following I/O

$\{IN =< 1,2,..,100 > \Lambda\ OUT =<>\}$

read x;
while x≠100 do
write x;
read x;
end

$\{\ OUT\ =< 1,2,\ldots,99 >\}$

❑Prove GCD

$\{x = m \wedge y = n\}$
while (x ≠ y) do
  if (x ≤ y) then y := y - x
  else x := x – y
$\{x = \gcd(m;\ n)\}$

(Optional) 作业：Arrays

❑Hoare logic rule for arrays

- ➢Use a specific theory of arrays (McCarthy 1962)
- ➢Enrich the assertion language with expressions $A\{e \longmapsto f\}$
  - Meaning: the array equal to A except that index e maps to value f
- ➢Axioms:
  - Write: $\{P[A\{e \longmapsto f\}/A]\}\ A(\,e\,) := f\ \{P\}$
  - Read: $A\{e \longmapsto f\}(\,e\,) = f\,;\ (e \neq e') \rightarrow (A\{e \mapsto f\}(e') = A(e'))$

❑Array swap:

- ➢Prove $\{A(I) = x \wedge A(J) = y\}\ p\ \{A(I) = y \wedge A(J) = x\}$
- ➢Where the program $p \stackrel{\text{def}}{=} T := A(I);\ A(I) := A(J);\ A(J) := T$

(Optional) 作业:

- **Hoare rules for**
  - function calls: {P} x:=f(x1, x2,…,xn) {Q}
  - Assignments with side-effects (e.g., memory aliasing)
    - E.g. { *y = 5 or x = y } *x = 5 { *x + *y = 10 }
  - Handling mutable records: : RECORD f1 : T1; f2 : T2 END
    - i.e., Structs in C or class in Java/C++
  - Handling a parallel composition statement *stat* || *stat*
    - $s_1 \,||\, s_2$:
      - execute s_1 and s_2 in parallel,
      - terminates when both *s*1 and *s*2 terminate