Verification Frameworks and Hoare Logic

Sources

- K. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs (Second Edition)*. Springer-Verlag, Berlin, 1997.
- D. Gries. *The Science of Programming.* Springer-Verlag, Berlin, 1981.

Verification Frameworks

This course is about verifying systems mathematically.

In order to do this, one needs a *verification framework*, which consists of following components.

Class *Sys* of system descriptions. *Sys* must be given mathematically, with (at least) semantics and (usually) syntax.

Class *Spec* of system specifications/requirements. *Spec* must also be given mathematically.

Relation sat $\subseteq$ ***Sys*** $\times$ ***Spec*****.**

Once a verification framework has been defined, verifying a specific systems $S \in$ *Sys* against a specific requirement $R \in$ *Spec* means proving that S sat R.

Verification Frameworks (cont.)

Given a verification framework, how does one establish whether or not S sat R? Two main approaches.

Proof-based: Develop proof rules for proving S sat R and use them to prove correctness

Algorithmic: Give decision procedures for computing if S sat R holds.

In this class we will study several different verification frameworks, including ones that are proof-based and others that are algorithmic.

Hoare Logic and Program Verification

The first verification framework we will study: *Hoare Logic*.

- *Sys* consists of programs written in a simple "guarded commands" programming notation.
- *Spec* consists of pairs of formulas given in first-order logic (= *predicate calculus*); predicates typically refer to program variables.
- S sat R holds if, whenever program is started in state satisfying first predicate and program terminates, the final state satisfies the second predicate.
- Verification conducted using proof rules.

The logic is sometimes called *Floyd-Hoare* logic and dates back to a paper by Hoare in 1969. It was a very active topic of research in 70s and 80s for sequential and parallel programs.

The GC Programming Language

Systems in Hoare Logic are given as programs in a small programming language called *GC* (for *guarded commands*).. We assume existence of following sets.

***Var*:** Program variables (assume integer-valued).

***AE*:** Arithmetic expressions built using constants, variables, operators, etc. (e.g $x + 1$)

***BE*:** Boolean expressions (e.g. $x = 0$, $(y = 0) \wedge (x = 2)$).

Note Together with definitions of $FV_{\mathit{AE},\mathit{BE}}$ and *subst*, we have the syntactic component of a data theory:

$$\langle \mathbb{Z}, \mathit{Var}, \mathit{AE}, \mathit{BE}, FV_{\mathit{AE},\mathit{BE}}, \mathit{subst} \rangle$$

($\mathbb{Z} = \{\ldots, -1, 0, 1, \ldots\}$ is the set of integers.)

GC Statements

Let $v_1, \ldots \subseteq$ *Var*, $e_1, \ldots \subseteq$ *AE*, and $G_1, \ldots \subseteq$ *BE*. Then the set $\mathbb{S}$ of GC statements is defined as follows.

S	$::=$	`skip`	no-op
	$\mid$	`halt`	abort
	$\mid$	$v_1, \ldots, v_n \texttt{:=} e_1, \ldots, e_n$	assignment (*)
	$\mid$	$S; S$	sequential composition
	$\mid$	$\texttt{if}\ G_1 \rightarrow S\ [] \ldots []\ G_n \rightarrow S\ \texttt{fi}$	alternative composition
	$\mid$	$\texttt{do}\ G_1 \rightarrow S\ [] \ldots []\ G_n \rightarrow S\ \texttt{od}$	iteration

(*) In $v_1, \ldots, v_n := e_1, \ldots, e_n$, there is an additional syntactic restriction: if $i \neq j$ then $v_i \neq v_j$ (i.e. v_i and v_j must be different variables).

Semantics of GC

To treat GC programs mathematically, they must be given a mathematical meaning, or *semantics*. There are different ways to do this.

Denotational. Programs are defined as functions mapping states (initial values of variables) to (sets of) states.

Operational. Programs defined in terms of how they execute.

Our semantics of GC will be an operational semantics given in the Structural Operational Semantics (SOS) style developed by Scottish computer scientist Gordon Plotkin in the early 80s.

- He only published a peer-reviewed paper on the subject in 2004; results previously available in 1981 Aarhus University (Denmark) technical report (Tech. Rep. DAIMI FN-19).
- SOS relies on specifying operational semantics using inference rules.
- Two types of SOS specifications: *evaluation* ("big-step") defines what programs evaluate to, while *transition* ("small-step") defines a program's atomic execution steps.

A Big-Step SOS for GC

The big-step SOS for GC will involve defining a relation $\Longrightarrow \subseteq (\mathbb{S} \times \Sigma) \times \Sigma$, where Σ is the set of program states defined below. Intuitively,

$$\langle S, \sigma \rangle \Longrightarrow \sigma'$$

holds when statement S, starting in state σ, is able to terminate in state σ'.

Definition Σ, the set of *states*, is defined as the set of mappings *Var* $\longrightarrow \mathbb{Z}$.

Defining $\Longrightarrow$ for GC

The evaluation relation $\Longrightarrow$ for GC is defined inductively using inference rules.

- Premises of rules will typically involve evaluation of subexpressions.
- Conclusion will define evaluation of over-all statement, given premises.

Assumptions:

- A function $[\![-]\!]_{\mathit{AE}} - \in \mathit{AE} \times \Sigma \longrightarrow \mathbb{Z}$; $[\![e]\!]_{\mathit{AE}}(\sigma)$ returns the value of e in state σ.
- A relation $\models_{\mathit{BE}} \subseteq \Sigma \times \mathit{BE}$; $\sigma \models_{\mathit{BE}} G$ holds when G is true in state σ.
- State updating can be generalized as follows: if $\sigma \in \Sigma$ and $v_1, \ldots, v_n$, $k_1, \ldots, k_n$ are sequences of variables/integers with the property that if $i \neq j$, then $v_i \neq v_j$, then

$$(\sigma[v_1 \mapsto k_1, \cdots, v_n \mapsto k_n])(x) = \begin{cases} k_i & \text{if } x = v_i \\ \sigma(x) & \text{otherwise} \end{cases}$$

Note $\langle [\![-]\!]_{\mathit{AE}} -, \models_{\mathit{BE}} \rangle$ is the semantic part of the data theory of program expressions!

Defining $\Longrightarrow$: Rules (1/2)

$$\frac{-}{\langle \mathtt{skip}, \sigma\rangle \Longrightarrow \sigma}\ (\mathtt{skip})$$

$$\frac{k_1 = [\![e_1]\!]_{\mathit{AE}}(\sigma) \quad \ldots \quad k_n = [\![e_n]\!]_{\mathit{AE}}(\sigma)}{\langle v_1, \ldots, v_n \mathtt{:=} e_1, \ldots, e_n, \sigma\rangle \Longrightarrow \sigma[v_1 \mapsto k_1, \ldots, v_n \mapsto k_n]}\ (\mathtt{:=})$$

$$\frac{\langle S_1, \sigma\rangle \Longrightarrow \sigma' \quad \langle S_2, \sigma'\rangle \Longrightarrow \sigma''}{\langle S_1 \mathtt{;} S_2, \sigma\rangle \Longrightarrow \sigma''}\ (\mathtt{;})$$

$$\frac{\sigma \models_{\mathit{BE}} G_i \quad \langle S_i, \sigma\rangle \Longrightarrow \sigma'}{\langle \mathtt{if}\ G_1 \rightarrow S_1 [\!] \ldots [\!]\, G_n \rightarrow S_n\ \mathtt{fi}, \sigma\rangle \Longrightarrow \sigma'}\ (\mathtt{if})$$

Defining $\Longrightarrow$: Rules (2/2)

$$\frac{\sigma \not\models_{BE} G_1 \quad \ldots \quad \sigma \not\models_{BE} G_n}{\langle \texttt{do}\ G_1 \rightarrow S_1 \,[]\cdots[]\, G_n \rightarrow S_n\ \texttt{od}, \sigma\rangle \Longrightarrow \sigma} \quad (\texttt{do}_1)$$

$$\frac{\sigma \models_{BE} G_i \quad \langle S_i, \sigma\rangle \Longrightarrow \sigma' \quad \langle \texttt{do}\ G_1 \rightarrow S_1 \,[]\cdots[]\, G_n \rightarrow S_n\ \texttt{od}, \sigma'\rangle \Longrightarrow \sigma''}{\langle \texttt{do}\ G_1 \rightarrow S_1 \,[]\cdots[]\, G_n \rightarrow S_n\ \texttt{od}, \sigma\rangle \Longrightarrow \sigma''} \quad (\texttt{do}_2)$$

Notes

- There are no rules for `halt`! This means there can be no states σ, σ' such that $\langle \texttt{halt}, \sigma\rangle \Longrightarrow \sigma'$. Implication: `halt` does not terminate.
- `if` $\cdots$ `fi` does not terminate if all the guards are false in a given state. (Why?)
- `if` $\cdots$ `fi` and `do` $\cdots$ `od` can give rise to nondeterminism. (How?)

Notation If $S \in \mathbb{S}$ and $\sigma \in \Sigma$ then $[\![S]\!](\sigma) = \{\, \sigma' \in \Sigma \mid \langle S, \sigma\rangle \Longrightarrow \sigma' \,\}$.

State Predicates

In Hoare Logic we now define $Sys = \mathbb{S}$. What about $Spec$?

- Program specifications will involve first-order formulas (sometimes called *state predicates* in the literature).
- What are state predicates? First-order logical formulas over the data theory of program expressions (i.e. *AE*, *BE*, etc.).

Examples

1. $\mathtt{x} < 3$
2. $\forall j \in \mathbb{Z}.\ 0 \leq j < \mathtt{i} \Rightarrow (\mathtt{min} \leq \mathtt{A}[j])$

Let Φ be the set of state predicates. The semantics of state predicates is the usual one for first-order logic — a relation $\models \subseteq \Sigma \times \Phi$, where $\sigma \models \phi$ means "ϕ is true in state σ."

Specifications in Hoare Logic

We can now define the set of specifications in Hoare Logic.

$$Spec = \Phi \times \Phi$$

In a specification $\langle P, Q \rangle$:

- P is called the *precondition*.
- Q is called the *postcondition*.

The Satisfaction Relation in Hoare Logic

The third component in the Hoare Logic verification framework is "sat": when does a program satisfy a specification?

Definition Let S be a program (statement), $\langle P, Q \rangle$ be a precondition/postcondition pair. Then S sat $\langle P, Q \rangle$ if and only if, for every σ, σ' such that that $\sigma \models P$ and $\langle S, \sigma \rangle \Longrightarrow \sigma'$, $\sigma' \models Q$.

Terminology Judgments in Hoare Logic are called *Hoare triples* and are written

$$\{P\}\ S\ \{Q\}.$$

A Hoare triple $\{P\}\ S\ \{Q\}$ is *valid* if it is the case that S sat $\langle P, Q \rangle$.

Note The notation of satisfaction here is called *partial correctness*, because programs are not required to terminate. *Total correctness* imposes an additional termination requirement.

Proving the Validity of Hoare Triples

We have now defined the Hoare Logic verification framework. How do we now verify programs?

Traditional approach relies on proofs using a collection of inference rules.

- Rules have form $\boxed{\dfrac{\textit{premises}}{\textit{conclusion}}}$ $(\textit{name})$,
- *premises* is a list of judgments and first-order statements (i.e. elements of Φ!), and *conclusion* is a judgment.
- If a rule has no hypotheses, it is sometimes called an *axiom*.
- A rule encodes a single step of reasoning and can be applied once its premises have been proved.

Notation If $P \in \Phi, v_1, \ldots \in$ *Var*, $e_1, \ldots \in$ *AE* then $P^{v_1,\ldots,v_n}_{e_1,\ldots,e_n}$ is “Hoare-speak” for (simultaneous) substitution of each v_i by e_i in P.

Axioms of Hoare Logic

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}}\ (\texttt{skip})$$

$$\frac{}{\{P\}\ \texttt{halt}\ \{Q\}}\ (\texttt{halt})$$

$$\frac{}{\{P^{v_1,\ldots,v_n}_{e_1,\ldots,e_n}\}\ v_1,\ldots,v_n \texttt{:=} e_1,\ldots,e_n\ \{P\}}\ (\texttt{:=})$$

Inference Rules of Hoare Logic: Program Constructs

$$\frac{\{P\}\ S_1\ \{Q\} \qquad \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1\,\texttt{;}\,S_2\ \{R\}}\ (\texttt{;})$$

$$\frac{\{P \wedge G_1\}\ S_1\ \{Q\} \quad \cdots \quad \{P \wedge G_n\}\ S_n\ \{Q\}}{\{P\}\ \texttt{if}\ G_1 \rightarrow S_1\ [\!]\cdots[\!]\ G_n \rightarrow S_n\ \texttt{fi}\ \{Q\}}\ (\texttt{if})$$

$$\frac{\{I \wedge G_1\}\ S_1\ \{I\} \quad \cdots \quad \{I \wedge G_n\}\ S_n\ \{I\}}{\{I\}\ \texttt{do}\ G_1 \rightarrow S_1\ [\!]\cdots[\!]\ G_n \rightarrow S_n\ \texttt{od}\ \{I \wedge \neg G_1 \wedge \cdots \wedge \neg G_n\}}\ (\texttt{do})$$

Note In Rule $(\texttt{do})$, state predicate I is often called a *loop invariant*.

Inference Rules of Hoare Logic: State Predicate Reasoning

$$\frac{P' \Rightarrow P \qquad \{P\}\, S\, \{Q\} \qquad Q \Rightarrow Q'}{\{P'\}\, S\, \{Q'\}} \; (\Rightarrow)$$

Note In this rule, $P' \Rightarrow P$ and $Q \Rightarrow Q'$ are intended *tautologies*. They must be proven using a proof system for first-order logic (i.e. not using Hoare Logic).

Example

Consider the following program `Pr`, which should calculate the quotient and remainder of dividing `x` by `y`.

```
r, q := x, 0;
do
   y ≤ r → r, q := r-y, q+1
od
```

We would like to prove

$$\{\mathtt{x} \geq 0\}\ \mathtt{Pr}\ \{\mathtt{x} = \mathtt{q} \cdot \mathtt{y} + \mathtt{r}\ \wedge\ 0 \leq \mathtt{r} < \mathtt{y}\}$$

In what follows, define $I \stackrel{\Delta}{=} (\mathtt{x} = \mathtt{q} \cdot \mathtt{y} + \mathtt{r}\ \wedge\ 0 \leq \mathtt{r})$.

Example (cont.)

$\boxed{\text{Recall}}$ $I \stackrel{\Delta}{=} (\mathtt{x} = \mathtt{q} \cdot \mathtt{y} + \mathtt{r} \ \wedge \ 0 \leq \mathtt{r})$

$$\{\mathtt{x} = (\mathtt{q}+\mathtt{1}) \cdot \mathtt{y} + \mathtt{r} - \mathtt{y} \wedge \mathtt{0} \leq (\mathtt{r}-\mathtt{y})\} \ \mathtt{r},\mathtt{q} := \mathtt{r}-\mathtt{y},\mathtt{q}+\mathtt{1} \ \{I\} \qquad (\mathtt{:=}) \qquad (1)$$

$$(I \wedge (\mathtt{y} \leq \mathtt{r})) \Rightarrow (\mathtt{x} = (\mathtt{q}+\mathtt{1}) \cdot \mathtt{y} + (\mathtt{r}-\mathtt{y}) \wedge 0 \leq (\mathtt{r}-\mathtt{y})) \qquad (2)$$

$$\{I \wedge (\mathtt{y} \leq \mathtt{r})\} \ \mathtt{r},\mathtt{q} := \mathtt{r}-\mathtt{y},\mathtt{q}+\mathtt{1} \ \{I\} \qquad (\Rightarrow)2,1 \qquad (3)$$

$$\{I\} \ \textsf{do} \ldots \textsf{od} \ \{I \wedge (\mathtt{y} > \mathtt{r})\} \qquad (\textsf{do})3 \qquad (4)$$

$$\{\mathtt{x} = \mathtt{0} \cdot \mathtt{y} + \mathtt{x} \wedge (\mathtt{0} \leq \mathtt{x})\} \ \mathtt{r},\mathtt{q} := \mathtt{x},\mathtt{0} \ \{I\} \qquad (\mathtt{:=}) \qquad (5)$$

$$\mathtt{x} \geq 0 \Rightarrow ((\mathtt{x} = \mathtt{0} \cdot \mathtt{y} + \mathtt{x}) \wedge (\mathtt{0} \leq \mathtt{x})) \qquad (6)$$

$$\{\mathtt{x} \geq 0\} \ \mathtt{r},\mathtt{q} := \mathtt{x},\mathtt{0} \ \{I\} \qquad (\Rightarrow)5,6 \qquad (7)$$

$$\{\mathtt{x} \geq 0\} \ \mathtt{r},\mathtt{q} := \mathtt{x},\mathtt{0}; \textsf{do} \ldots \textsf{od} \ \{I \wedge (\mathtt{y} > \mathtt{r})\} \qquad (\mathtt{;})7,4 \qquad (8)$$

$$(I \wedge (\mathtt{y} > \mathtt{r})) \Rightarrow (\mathtt{x} = \mathtt{q} \cdot \mathtt{y} + \mathtt{r} \wedge 0 \leq \mathtt{r} < \mathtt{y}) \qquad (9)$$

$$\{\mathtt{x} \geq 0\} \ \textsf{Pr} \ \{(\mathtt{x} = \mathtt{q} \cdot \mathtt{y} + \mathtt{r} \wedge 0 \leq \mathtt{r} < \mathtt{y}\} \qquad (\Rightarrow)8,9 \qquad (10)$$

Reasoning in Practice: Proof Outlines

Proofs are usually given as *proof outlines*.

1. State predicates are inserted into program text so that every statement (simple and compound) has a pre- and postcondition.
2. A proof outline is *valid* if every embedded triple is valid and adjacent predicates related by implication.

$\{\mathtt{x} \geq 0\}$

$\{\mathtt{x} = 0 \cdot \mathtt{y} + \mathtt{x} \ \wedge \ 0 \leq \mathtt{x}\}$

```
r, q := x, 0;
```

$\{I\}$

```
do
```

$\quad \mathtt{y} \leq \mathtt{r} \ \rightarrow \qquad \{I \wedge \mathtt{y} \leq \mathtt{r}\}$

```
                  r, q := r-y, q+1
```

$\qquad\qquad\qquad \{I\}$

```
od
```

$\{I \wedge \mathtt{y} > \mathtt{r}\}$

$\{\mathtt{x} = \mathtt{q} \cdot \mathtt{y} + \mathtt{r} \ \wedge \ 0 \leq \mathtt{r} < \mathtt{y}\}$

Formal Definition of Set $\mathcal{P}$ of Proof Outlines

Given inductively!

In the following rules, $R, R_1, \ldots$ are *partial proof outlines*:

- $R \in \mathbb{S}$ may be a program; or
- $R = \{P\}\,S$ may be a program with a precondition ($S \in \mathbb{S}$); or
- $R = S\,\{Q\}$ may be a program with a postcondition ($S \in \mathbb{S}$); or
- $R \in \mathcal{P}$ may be a full proof outline.

Inductive Definition of $\mathcal{P}$

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\} \in \mathcal{P}}$$

$$\frac{}{\{P^{v_1,\ldots,v_n}_{e_1,\ldots,e_n}\}\ v_1,\ldots,v_n := e_1,\ldots,e_n\ \{P\} \in \mathcal{P}}$$

$$\frac{\{P\}\ R_1\ \{P'\} \in \mathcal{P} \quad \{P'\}\ R_2\ \{Q\} \in \mathcal{P}}{\{P\}\ R_1\texttt{;}\ \{P'\}\ R_2\ \{Q\} \in \mathcal{P}}$$

$$\frac{\{P \wedge G_1\}\ R_1\ \{Q\} \in \mathcal{P} \quad \cdots \quad \{P \wedge G_n\}\ R_n\ \{Q\} \in \mathcal{P}}{\{P\}\ \texttt{if}\ G_1 \rightarrow \{P \wedge G_1\}\ R_1\ \{Q\}\ [\!]\cdots[\!]\ G_n \rightarrow \{P \wedge G_n\}\ R_n\ \{Q\}\ \texttt{fi}\ \{Q\} \in \mathcal{P}}$$

$$\frac{\{I \wedge G_1\}\ R_1\ \{I\} \in \mathcal{P} \quad \cdots \quad \{I \wedge G_n\}\ R_n\ \{I\} \in \mathcal{P}}{\{I\}\ \texttt{do}\ G_1 \rightarrow \{I \wedge G_1\}\ R_1\ \{I\}\ [\!]\cdots[\!]\ G_n \rightarrow \{I \wedge G_n\}\ R_n\ \{I\}\ \texttt{od}\ \{I \wedge \bigwedge_i \neg G_i\} \in \mathcal{P}}$$

$$\frac{P' \Rightarrow P \quad \{P\}\ R\ \{Q\} \in \mathcal{P}}{\{P'\}\ \{P\}\ R\ \{Q\} \in \mathcal{P}}$$

$$\frac{\{P\}\ R\ \{Q\} \in \mathcal{P} \quad Q \Rightarrow Q'}{\{P\}\ R\ \{Q\}\ \{Q'\} \in \mathcal{P}}$$

Reasoning in Practice: Where do Preconditions Come from?

- Begin by capturing as a postcondition Q what the result of the program S should be.
- Use axioms and inference rules to reason backwards to obtain a precondition P for S.
- Result is a of the triple $\{P\}\ S\ \{Q\}$.
- This idea can be formalized via *weakest liberal preconditions*.

First, we can associate a set of states with any state predicate as follows.

$\boxed{\text{Definition}}$ Let $\phi \in \Phi$ be a state predicate. Then $[\![\phi]\!] = \{\, \sigma \in \Sigma \mid \sigma \models \phi \,\}$

$\boxed{\text{Definition}}$ Let S be a statement and $\phi \in \Phi$ be a state predicate. Then the *weakest liberal precondition*, *wlp*(S, ϕ) of S with respect to ϕ is given by:

$$\textit{wlp}(S, \phi) = \{\, \sigma \in \Sigma \mid [\![S]\!](\sigma) \subseteq [\![\phi]\!] \,\}$$

Weakest Liberal Preconditions Can Be Computed Syntactically!

Edsger Dijkstra initiated study on this problem in 1975.

Theorem Let S be a program and ϕ a state predicate. Then the following hold.

1. *wlp*$(\texttt{skip}, \phi) = [\![\phi]\!]$
2. *wlp*$(\texttt{halt}, \phi) = \Sigma = [\![\text{tt}]\!]$
3. *wlp*$(v_1, \ldots, v_n \texttt{:=} e_1, \ldots, e_n, \phi) = [\![\phi^{v_1, \ldots, v_n}_{e_1, \ldots, e_n}]\!]$
4. *wlp*$(S_1 \texttt{;}\, S_2, \phi) =$ *wlp*$(S_1,$ *wlp*$(S_2, \phi))$
5. *wlp*$(\texttt{if}\ G_1 \to S_1 [\!]\cdots[\!]\ G_n \to S_n\ \texttt{fi}, \phi) =$
   $([\![\neg G_1]\!] \cup$ *wlp*$(S_1, \phi)) \cap \cdots \cap ([\![\neg G_n]\!] \cup$ *wlp*$(S_n, \phi))$

Note In (5) $[\![\neg G_i]\!] \cup$ *wlp*(S_i, ϕ) may be seen as “equivalent to” $G_i \Rightarrow$ *wlp*(S_i, ϕ).

What about $\texttt{do} \cdots \texttt{od}$?

wlp and `do`

wlp$(\texttt{do} \;\cdots\; \texttt{od}, \phi)$ may be represented syntactically in theory, but it is impractical to compute (relies on encoding computations of while loops as integers). We can come close, however.

$\boxed{\text{Fact}}$ Let $D = \texttt{do}\ G_1 \to S_1 \,[]\cdots[]\, G_n \to S_n\ \texttt{od}$. Then

$$[\![D]\!](\sigma) = [\![\texttt{if}\ G_1 \to (S_1; D) \,[]\cdots[]\, G_n \to (S_n; D) \,[]\, \neg(G_1 \vee \cdots \vee G_n) \to \texttt{skip}\ \texttt{fi}]\!](\sigma)$$

In other words, $[\![D]\!]$ is a solution to a recursive equation involving $\texttt{if} \cdots \texttt{fi}$.

Given D, ϕ, consider the following.

$$\begin{aligned} H_0 &\stackrel{\Delta}{=} [\![\text{tt}]\!] = \Sigma \\ H_j &\stackrel{\Delta}{=} ([\![\neg G_1]\!] \cup \textit{wlp}(S_1, H_{j-1})) \cap \cdots \cap ([\![\neg G_n]\!] \cup \textit{wlp}(S_n, H_{j-1})) \\ &\quad \cap ([\![(\neg(G_1 \vee \cdots \vee G_n)) \Rightarrow \phi]\!]) \end{aligned}$$

$\boxed{\text{Fact}}$ *wlp*$(D, \phi) = \bigcap_{j \geq 0} H_j$

Why is this "close", but not necessarily the answer? Because obvious approach to representing the H_j falls outside the syntax of first-order logic.

Reasoning in Practice: Loop Invariants

- The inference rule for do-loops requires the creation of a *loop invariant* which must hold each time through the loop.
- Coming up with the right loop invariants is often the trickiest aspect of sequential program verification.
  - General strategy: *weaken postcondition* of loop, by e.g. deleting conjuncts, replacing constant with variable, etc.
  - Gries book, Chapter 16, has good tips and examples.
- Invariants generally capture "design information" and are very useful as documentation, even if you don't prove your programs correct.

Soundness and Relative Completeness

Recall soundess and completeness.

Soundness: Can only valid things be proved?

Completeness: Can all valid things be proved?

We can study these issues for Hoare Logic!

Theorem (Soundness) Suppose $\{P\}\ S\ \{Q\}$ is provable. Then it is valid.

Theorem (Relative Completeness) Suppose that there is a complete proof system for establishing $P \Rightarrow Q$. Then the validity of $\{P\}\ S\ \{Q\}$ implies $\{P\}\ S\ \{Q\}$ is provable.

Question Why is only "relative completeness" possible, in general?