6. Hoare Logic and Weakest Preconditions

Program Verification

ETH Zurich, Spring Semester 2017

Alexander J. Summers

Program Correctness

- There are many notions of *correctness properties* for a given program
  - is the program *guaranteed to reach* a certain program point (e.g. terminate)?
  - when the program reaches this point, are certain *values guaranteed*?
  - could the program encounter *runtime errors* / raise certain exceptions?
  - will the program *leak memory* / *secret data*, etc.?
- To build a verifier, we need *clearly defined correctness criteria*
- We will focus on a classic correctness notion: *partial correctness*
  - A program s is *partially correct* with respect to pre-/post-conditions A_1/A_2 iff:

  All executions of s *starting from states satisfying* A_1 are free of runtime errors, *and*, any such executions which terminate *will do so in states satisfying* A_2

  - For non-terminating executions, we still guarantee absence of runtime errors
  - The above notion is succinctly expressed using a *Hoare triple*: $\{\mathrm{A}_1\}\ \mathrm{s}\ \{\mathrm{A}_2\}$

A Small Imperative Language

- Program *variables* x,y,z,… (can be assigned to)
- Expressions e,e_1,e_2,… (we won't fix a specific syntax for now)
  - e.g. includes boolean/arithmetic operators, mathematical functions
  - assume a *standard type system* (no subtyping/casting); we omit the details
  - we'll typically write b,b_1,b_2,… for boolean-typed expressions
  - expression evaluation assumed to be *side-effect-free* for all expressions
- Assertions A used in specifications, Hoare triples, etc.
  - for now, *assertions are just the same as boolean expressions*
  - later in the course, we'll want specifications richer than program expressions
- Statements s (see subsequent slides)
  - We define a small language here; may extend the syntax later

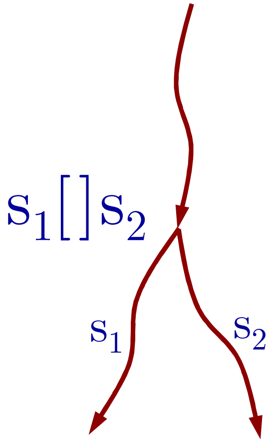
Standard Statements

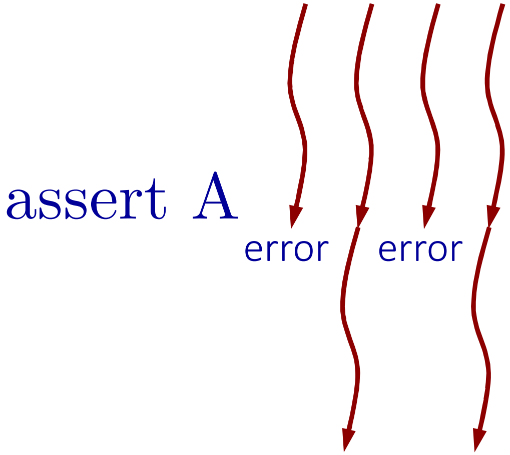
- Our statement language includes the following standard constructs:
  - skip (does nothing when executed)
  - $\mathrm{x} := \mathrm{e}$ (assignment: changes value of x)
  - $\mathrm{s}_1;\mathrm{s}_2$ (sequential composition: execute s_1 followed by s_2)
  - $\mathrm{if(b)\{s_1\}else\{s_2\}}$ (execute s_1 if b evaluates to true; s_2 otherwise)
  - $\mathrm{while(b)\{s\}}$ (repeatedly execute s_1 while b evaluates to true)
- A *runtime state* is a mapping σ from variables to values
- We assume a *small-step operational semantics* (not formalised here)
  - A *runtime configuration* is a pair (s,σ) of a statement s and a program state σ
  - A *trace* is either an *infinite sequence of runtime configurations*, or is a *finite sequence of runtime configurations appended with* one of the following:
    a single runtime state σ (the *final state*), the symbol error, the symbol magic

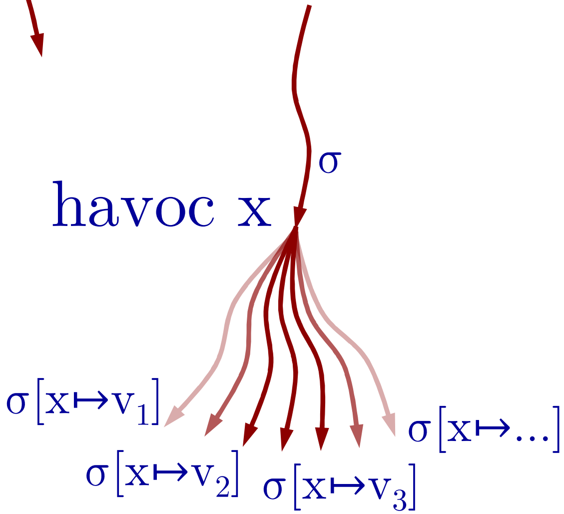
Verification and Non-Deterministic Statements

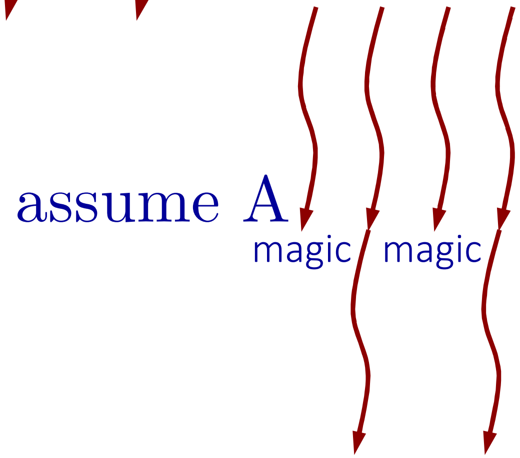
- We add the following statements to those of the previous slide:
  - $\mathrm{assert}\ A$ (traces for which A is false at this point end here with error)
  - $\mathrm{assume}\ A$ (traces for which A is false at this point end here with magic)
  - $s_1[\,]s_2$ (non-deterministic choice: execute either s_1 or s_2 from here)
  - $\mathrm{havoc}\ x$ (non-deterministically assign an arbitrary value to variable x)
- A *failing trace* is one ending in error
- The former two statements *filter traces* (fewer outgoing than ingoing)
- The latter two statements *split traces* (more outgoing than ingoing)
  - this is characteristic of a non-deterministic language
- We write $\sigma \vDash A$ to denote that A is true in σ (σ defines a model of A)

Trace Semantics in Pictures









Hoare Logic I

- *Hoare Logic* is a standard proof style for correctness proofs
  - Proofs are *derivation trees*, built by *instantiating derivation rule schemas*
  - Every derivation tree has a Hoare triple as its *root*
  - Rule schemas must be instantiated for particular statements, assertions etc.
- Rule schemas for standard statements (partial correctness):
  - we write $\mathrm{A}[\mathrm{e}/\mathrm{x}]$ for (capture-avoiding) substitution of e for each free x in A
  - we omit while loops for now (coming soon…)

$$\frac{}{\{\mathrm{A}\}\ \mathrm{skip}\ \{\mathrm{A}\}}\text{(skip)} \qquad \frac{}{\{\mathrm{A}[\mathrm{e}/\mathrm{x}]\}\ \mathrm{x} := \mathrm{e}\ \{\mathrm{A}\}}\text{(ass)} \qquad \frac{\{\mathrm{A}_1\}\ \mathrm{s}_1\ \{\mathrm{A}_2\} \quad \{\mathrm{A}_2\}\ \mathrm{s}_2\ \{\mathrm{A}_3\}}{\{\mathrm{A}_1\}\ \mathrm{s}_1;\mathrm{s}_2\ \{\mathrm{A}_3\}}\text{(seq)}$$

$$\frac{\{\mathrm{A}_1 \wedge \mathrm{b}\}\ \mathrm{s}_1\ \{\mathrm{A}_2\} \quad \{\mathrm{A}_1 \wedge \neg \mathrm{b}\}\ \mathrm{s}_2\ \{\mathrm{A}_2\}}{\{\mathrm{A}_1\}\ \mathrm{if}(\mathrm{b})\{\mathrm{s}_1\}\mathrm{else}\{\mathrm{s}_2\}\ \{\mathrm{A}_2\}}\text{(if)}$$

Hoare Logic II

- Rule schemas for verification and non-deterministic statements:

$$\frac{\{A_1\}\ s_1\ \{A_2\} \quad \{A_1\}\ s_2\ \{A_2\}}{\{A_1\}\ s_1[]s_2\ \{A_2\}}\text{(nondet)} \qquad \frac{}{\{\forall y.A[y/x]\}\ \text{havoc x}\ \{A\}}\text{(havoc)}$$

$$\frac{}{\{A_1 \wedge A\}\ \text{assert}\ A_1\ \{A\}}\text{(assert)} \qquad \frac{}{\{A_1 \Rightarrow A\}\ \text{assume}\ A_1\ \{A\}}\text{(assume)}$$

- The *rule of consequence* allows reasoning in terms of the *semantics of the assertions* (this rule is very important!) :

$$\frac{A_1 \vDash A_2 \quad \{A_2\}\ s\ \{A_3\} \quad A_3 \vDash A_4}{\{A_1\}\ s\ \{A_4\}}\text{(conseq)}$$

Hoare Logic (Alternative Rules)

- Consider the following alternative rules

$$\frac{}{\{A\}\ \text{havoc x}\ \{\exists y.A[y/x]\}}\ \text{(havoc-alt)}$$

$$\frac{A \vDash A_1}{\{A\}\ \text{assert}\ A_1\ \{A_1 \wedge A\}}\ \text{(assert-alt)} \qquad \frac{}{\{A\}\ \text{assume}\ A_1\ \{A \wedge A_1\}}\ \text{(assume-alt)}$$

- Exchanging (any of) these rules for their counterparts on the previous slide *doesn't change the derivable Hoare triples* (see exercises)
  - The alternative rules are "forward oriented": it is easier to understand them from left-to-right (those on previous slide are "backward-oriented")
  - You may find one formulation more intuitive than another

Hoare Triples - Properties

- $\{A_1\}\ s\ \{A_2\}$ is *semantically valid*, written $\vDash\{A_1\}\ s\ \{A_2\}$, iff: For all states σ_1 such that $\sigma_1 \vDash A_1$, there are *no failing traces* starting from (s,σ), and for all such traces *ending in final states* σ_2: $\sigma_2 \vDash A_2$
  - *equivalently*, the program $\text{assume } A_1;\ s;\ \text{assert } A_2$ has *no failing traces*
- A *triple* $\{A_1\}\ s\ \{A_2\}$ *is provable*, written $\vdash\{A_1\}\ s\ \{A_2\}$, iff there exists a derivation tree with $\{A_1\}\ s\ \{A_2\}$ as its root
- The derivation rules presented are *sound* (only prove valid triples):
  - For all s, A_1, A_2, if $\vdash\{A_1\}\ s\ \{A_2\}$ then $\vDash\{A_1\}\ s\ \{A_2\}$
- Under certain conditions these derivation rules are also *complete*:
  - If the assertion language is *able to express all weakest preconditions* (explained soon) then: for all s, A_1, A_2, if $\vDash\{A_1\}\ s\ \{A_2\}$ then $\vdash\{A_1\}\ s\ \{A_2\}$
  - in some sense, this tells us we are not "missing" proof rules from the system

Hoare Logic - Example

- Suppose we want to prove the triple $\{x=2\}\ x := x\text{-}1\ \{x>0\}$
  - A *triple* $\{A_1\}\ s\ \{A_2\}$ *is provable*, written $\vdash \{A_1\}\ s\ \{A_2\}$, iff there exists a derivation tree with $\{A_1\}\ s\ \{A_2\}$ as its root
  - This doesn't tell us *how* to construct such a derivation tree, in general
- We have *different choices* for how to construct derivation trees, e.g.:
  - (we typically don't draw the rule premises which are not Hoare triples)

$$\dfrac{\dfrac{}{\{x\text{-}1>0\}\ x := x\text{-}1\ \{x>0\}}\text{(ass)}}{\{x=2\}\ x := x\text{-}1\ \{x>0\}}\text{(conseq)} \qquad \dfrac{\dfrac{}{\{x\text{-}1=1\}\ x := x\text{-}1\ \{x=1\}}\text{(ass)}}{\{x=2\}\ x := x\text{-}1\ \{x>0\}}\text{(conseq)}$$

- This illustrates *redundancy in the proof system*
  - The redundancy is *useful when constructing proofs by hand* (flexible)
  - When trying to automate proofs, it is less desirable (enumerate derivations?)

Weakest Preconditions - Idea

- Suppose we want to prove a particular triple $\{A_1\}\ s\ \{A_2\}$
- This would require us to find suitable *intermediate assertions*
  - e.g. to prove $\{A_1\}\ s_1;s_2\ \{A_2\}$ we will need assertion(s) used as postcondition of s_1 / precondition of s_2, which also satisfy the appropriate proof rules
- We can *orient the search* for these assertions, and for a derivation
  - For example, starting from our postcondition, work backwards through s
  - At each sub-statement, find the precondition for this statement which works
  - We should always choose this precondition to be as *logically weak as possible*
- This idea is formalised by a *weakest precondition function* $\mathrm{wlp}(s,A)$
  - A is the intended postcondition (e.g. A_2 above); s the statement in question
  - name wlp stands for *weakest "liberal" precondition*: it ignores termination
  - dual *strongest postcondition* notion exists; we focus on weakest preconditions

Weakest Preconditions – Desired Properties

- We'd like a function $\mathrm{wlp(s,A)}$ returning a predicate on states s.t. :
  - *Expressibility*: For all s and A, $\mathrm{wlp(s,A)}$ is expressible as an assertion
  - *Soundness*: For all s and A, it is guaranteed that $\vDash\{\mathrm{wlp(s,A)}\}\ \mathrm{s}\ \{\mathrm{A}\}$
  - *Minimality*: For all s, $\mathrm{A_1}$, $\mathrm{A_2}$, if $\vDash\{\mathrm{A_1}\}\ \mathrm{s}\ \{\mathrm{A_2}\}$ then $\mathrm{A_1}\vDash\mathrm{wlp(s,A)}$
  - *Computability*: For all s and A, $\mathrm{wlp(s,A)}$ is computable (ideally, efficiently)
- We can now explain the condition for completeness (3 slides ago):
  - Soundness + Minimality semantically define a *unique predicate on states*
  - Our assertion language is *able to express all weakest preconditions* iff, for any s and A, there exists some $\mathrm{A_1}$ whose meaning is equivalent to this predicate
- Suppose we had a suitably-defined $\mathrm{wlp(s,A)}$ notion
  - We could build a *program verifier to check validity of triples* $\vDash\{\mathrm{A_1}\}\ \mathrm{s}\ \{\mathrm{A_2}\}$ by computing $\mathrm{wlp(s,A_2)}$ and then *checking the entailment* $\mathrm{A_1}\vDash\mathrm{wlp(s,A_2)}$

Weakest Preconditions – Definition I

- Attempt to define $\mathrm{wlp}(\mathrm{s},\mathrm{A})$ by "reading" derivation rules backwards
  - We want to find the weakest way to fill the $???$ in: $\vDash \{???\}\ \mathrm{s}\ \{\mathrm{A}\}$
- For example, consider the following three derivation rules:

$$\frac{}{\{\mathrm{A}\}\ \mathrm{skip}\ \{\mathrm{A}\}}\text{(skip)} \qquad \frac{}{\{\mathrm{A}[\mathrm{e}/\mathrm{x}]\}\ \mathrm{x} := \mathrm{e}\ \{\mathrm{A}\}}\text{(ass)} \qquad \frac{\{\mathrm{A}_1\}\ \mathrm{s}_1\ \{\mathrm{A}_2\} \quad \{\mathrm{A}_2\}\ \mathrm{s}_2\ \{\mathrm{A}_3\}}{\{\mathrm{A}_1\}\ \mathrm{s}_1;\mathrm{s}_2\ \{\mathrm{A}_3\}}\text{(seq)}$$

- We can "read off" three suitable cases of the wlp definition:
  - $\mathrm{wlp}(\mathrm{skip},\mathrm{A}) = \mathrm{A}$
  - $\mathrm{wlp}(\mathrm{x} := \mathrm{e},\mathrm{A}) = \mathrm{A}[\mathrm{e}/\mathrm{x}]$
  - $\mathrm{wlp}(\mathrm{s}_1;\mathrm{s}_2\ ,\mathrm{A}) = \mathrm{wlp}(\mathrm{s}_1\ ,\mathrm{wlp}(\mathrm{s}_2\ ,\mathrm{A}))$
- What about the other statements of our language?

Weakest Preconditions – Definition II

- Consider the other loop-free statements of our language:

$$\frac{\{A_1 \wedge b\}\ s_1\ \{A_2\} \quad \{A_1 \wedge \neg b\}\ s_2\ \{A_2\}}{\{A_1\}\ \mathrm{if}(b)\{s_1\}\mathrm{else}\{s_2\}\ \{A_2\}}\text{(if)} \qquad \frac{\{A_1\}\ s_1\ \{A_2\} \quad \{A_1\}\ s_2\ \{A_2\}}{\{A_1\}\ s_1[]s_2\ \{A_2\}}\text{(nondet)}$$

$$\frac{}{\{\forall y.A[y/x]\}\ \mathrm{havoc}\ x\ \{A\}}\text{(havoc)}$$

$$\frac{}{\{A_1 \wedge A\}\ \mathrm{assert}\ A_1\ \{A\}}\text{(assert)} \qquad \frac{}{\{A_1 \Rightarrow A\}\ \mathrm{assume}\ A_1\ \{A\}}\text{(assume)}$$

  - $\mathrm{wlp}(\mathrm{if}(b)\{s_1\}\mathrm{else}\{s_2\}\ ,A) = (b \Rightarrow \mathrm{wlp}(s_1\ ,A)) \wedge (\neg b \Rightarrow \mathrm{wlp}(s_2\ ,A))$
  - $\mathrm{wlp}(s_1[]s_2\ ,A) = \mathrm{wlp}(s_1\ ,A) \wedge \mathrm{wlp}(s_2\ ,A)$
  - $\mathrm{wlp}(\mathrm{havoc}\ x\ ,A) = \forall y.A[y/x]$
  - $\mathrm{wlp}(\mathrm{assert}\ A_1\ ,A) = A_1 \wedge A$
  - $\mathrm{wlp}(\mathrm{assume}\ A_1\ ,A) = A_1 \Rightarrow A$

Weakest Preconditions – Loops

- The standard Hoare Logic rule for loop constructs is the following:

$$\frac{\{A_I \wedge b\}\ s\ \{A_I\}}{\{A_I\}\ \mathrm{while}(b)\{s\}\ \{A_I \wedge \neg b\}}\ \text{(while-Hoare)}$$

- The assertion A_I is called a *loop invariant*
  - In general, our *current postcondition* won't be a suitable loop invariant
- The rule doesn't give us a direct definition of $\mathrm{wlp}(\mathrm{while}(b)\{s\}\ ,A)$
  - The obstacle is in finding an appropriate loop invariant for the above rule
  - We might imagine unrolling the loop to "define" $\mathrm{wlp}(\mathrm{while}(b)\{s\}\ ,A)$ as: $A \wedge \neg b \vee b \wedge \mathrm{wlp}(s, A \wedge \neg b) \vee b \wedge \mathrm{wlp}(s,\ b \wedge \mathrm{wlp}(s,\ A \wedge \neg b)) \vee \ldots$
  - This definition is *not effectively computable* for general loops: it tries to compute an "infinite" assertion (may not even be an expressible assertion)

Loops – Adding Invariants

- With hindsight, it's not surprising that we don't get an effective wlp
  - checking validity of $\vDash\{A_1\}\ s\ \{A_2\}$ for such a language is typically undecidable (even for *assertion languages* with decidable entailment)
  - if wlp *were* computable for general loops, we'd have an effective algorithm (!)
- We will require loops to be *annotated with loop invariants*
  - these annotations could be manual, generated by a static analysis tool, etc.
- We change the syntax for loops, and write $\mathrm{while(b)invariant}\ A\{s\}$
  - Here, A is the *declared loop invariant* for the while loop
- Replacing the usual rule (previous slide), we use the following one:

$$\frac{\{A_I \wedge b\}\ s\ \{A_I\}}{\{A_I\}\ \mathrm{while(b)invariant}\ A_I\{s\}\ \{A_I \wedge \neg b\}}\ \text{(while-inv)}$$

Weakest Preconditions for Annotated Loops

- For this annotated language, we can give a definition for loops:

$$\frac{\{\mathrm{A_I} \wedge \mathrm{b}\}\ \mathrm{s}\ \{\mathrm{A_I}\}}{\{\mathrm{A_I}\}\ \mathrm{while(b)invariant}\ \mathrm{A_I}\{\mathrm{s}\}\ \{\mathrm{A_I} \wedge \neg \mathrm{b}\}}\text{(while-inv)}$$

- $\mathrm{wlp(while(b)invariant}\ \mathrm{A_I}\{\mathrm{s}\}\ ,\mathrm{A}) =$

$$\mathrm{A_I} \wedge \forall \vec{\mathrm{y}}.((\mathrm{A_I} \wedge \mathrm{b} \Rightarrow \mathrm{wlp(s,\ A_I)}) \wedge (\mathrm{A_I} \wedge \neg \mathrm{b} \Rightarrow \mathrm{A}))[\vec{\mathrm{y}}/\vec{\mathrm{x}}]$$

  where $\vec{\mathrm{x}}$ is the set of variables modified in s, and $\vec{\mathrm{y}}$ are fresh variable names

- This may seem hard to understand directly, but informally:
  - The first conjunct insists that the *invariant* $\mathrm{A_I}$ *holds before the loop*
  - The next conjunct (inside the $\forall \vec{\mathrm{y}}$) corresponds to *checking the loop body*
  - The last conjunct (inside the $\forall \vec{\mathrm{y}}$) corresponds to guaranteeing that the desired *postcondition will hold after the loop* (if the loop terminates)

Eliminating Annotated Loops

- A while loop with loop invariant can be ***desugared*** as follows:
  - Let x_1, x_2 ,…, x_n be the (perhaps zero) variables modified by s
  - Then $\mathrm{while(b)invariant}\ A_I\{s\}$ is rewritten into the program:
  - $\mathrm{assert}\ A_I;\ \ \mathrm{havoc}\ x_1;\ \mathrm{havoc}\ x_2;\ \ldots\ ;\ \mathrm{havoc}\ x_n;$
    $((\mathrm{assume}\ A_I \wedge b;\ s;\ \mathrm{assert}\ A_I;\ \mathrm{assume\ false})$
    $[]$
    $(\mathrm{assume}\ A_I \wedge \neg b))$
  - The first $\mathrm{assert}\ A_I$ statement checks the loop invariant holds initially
  - The havocs model side-effects of an unbounded number of loop iterations
  - The next line checks that the loop invariant is ***preserved by each loop iteration***
  - The final line models termination, if we ever leave the loop, $A_I \wedge \neg b$ will hold
- Try applying wlp to this program and compare with the previous slide
  - you should get an equivalent formula to wlp definition for annotated loops

Approximation via Loop Invariants

- The desugaring on the previous slide requires checking that:
  - the declared loop invariant is preserved starting from states satisfying $A_I \wedge b$
  - the remainder of the program is correct starting from states satisfying $A_I \wedge \neg b$
  - in both cases, this might include states which are *never reached* by the loop
- e.g. $x{:=}5;\ \mathrm{while}(x\neq 2 \wedge x\neq 4)\mathrm{invariant}\ x>0\{x{:=}x\text{-}1\};\ \mathrm{assert}\ x\neq 2$
  - invariant is not re-established from *all* states in which $x>0 \wedge x\neq 2 \wedge x\neq 4$
  - the assert statement may fail for *some* states in which $x>0 \wedge \neg(x\neq 2 \wedge x\neq 4)$
  - But on execution, the invariant will always hold and the assert will succeed
- The loop invariant is used to *over-approximate the loop behaviour*
  - Our wlp is complete with respect to *provable triples* ($\vdash\{A_1\}\ s\ \{A_2\}$)
  - It is *not complete* with respect to *valid triples* ($\vDash\{A_1\}\ s\ \{A_2\}$)
  - Both the proof system and wlp are only complete with *precise loop invariants*

Wlp Summary

- For *annotated programs*, our wlp definition is summarised by

$$\mathrm{wlp}(\mathrm{skip},A) = A \qquad \mathrm{wlp}(x := e,A) = A[e/x]$$
$$\mathrm{wlp}(\mathrm{havoc}\ x\ ,A) = \forall y.A[y/x] \qquad \mathrm{wlp}(s_1;s_2\ ,A) = \mathrm{wlp}(s_1\ ,\mathrm{wlp}(s_2\ ,A))$$
$$\mathrm{wlp}(\mathrm{assert}\ A_1\ ,A) = A_1 \wedge A \qquad \mathrm{wlp}(\mathrm{assume}\ A_1\ ,A) = A_1 \Rightarrow A$$
$$\mathrm{wlp}(s_1[]s_2\ ,A) = \mathrm{wlp}(s_1\ ,A) \wedge \mathrm{wlp}(s_2\ ,A)$$
$$\mathrm{wlp}(\mathrm{if}(b)\{s_1\}\mathrm{else}\{s_2\}\ ,A) = (b \Rightarrow \mathrm{wlp}(s_1\ ,A)) \wedge (\neg b \Rightarrow \mathrm{wlp}(s_2\ ,A))$$
$$\mathrm{wlp}(\mathrm{while}(b)\mathrm{invariant}\ A_I\{s\}\ ,A) =$$
$$A_I \wedge \forall \vec{y}.((A_I \wedge b \Rightarrow \mathrm{wlp}(s,\ A_I)) \wedge (A_I \wedge \neg b \Rightarrow A))[\vec{y}/\vec{x}]$$

- With respect to our *desired properties*, this definition is *sound*, and complete with respect to *provable triples* ($\vdash \{A_1\}\ s\ \{A_2\}$)
  - but not necessarily with respect to *valid triples*: depends on loop invariants
- The definition returns an assertion; it is *expressible* and *computable*

Hoare Logic and Weakest Preconditions - Summary

- We have seen *Hoare Logic*: a means of proving program properties
  - proofs consist of derivation trees – flexible for manual proof efforts
- Automating the checking of Hoare triples via *weakest preconditions*
  - these provide a means of directing the proof search (working backwards)
- For recursion (loops, here), we require *annotated loop invariants*
  - approximate the recursive behaviour of a loop, according to the invariant
- Resulting *computable* notion of weakest precondition w.r.t. invariants
  - we reduce program correctness to checking entailment between assertions
- This idea allows the implementation of a *program verifier*
  - User provides specification and e.g. loop invariants in some assertion syntax
  - For suitable assertion syntaxes, we can check entailments with an SMT solver

Hoare Logic and Weakest Preconditions – References

- Hoare Logic:
    - *Assigning meanings to programs.* R. W. Floyd (1967)
    - *An axiomatic basis for computer programming.* Hoare, C. A. R. (1969)
    - *Soundness and Completeness of an Axiom System for Program Verification.* Stephen A. Cook (1978)

- Weakest Preconditions:
    - *Guarded commands, nondeterminacy and formal derivation of programs.* Edsger W. Dijkstra (1975)
    - *Avoiding exponential explosion: generating compact verification conditions.* Cormac Flanagan, James B. Saxe (2001)
    - *Weakest-precondition of unstructured programs.* Mike Barnett, K. Rustan M. Leino (2005)

- Other teaching material:
    - *Formal Methods and Functional Programming.* Peter Müller (ETH Zurich)
    - *Synthesis, Analysis, and Verification.* Viktor Kuncak (EPFL)