SMT-based Verification of Heap-manipulating Programs

Ruzica Piskac

Eleventh SRI Summer School on Formal Techniques

May 30 - June 5, 2022

A Mathematical Proof of Program Correctness?

Can you verify my program?

```
public void add (Object x)

{
  Node e = new Node();
  e.data = x;
  e.next = root;
  root = e;
  size = size + 1;
}
```

Which property are you interested in?

Example Questions in Verification

- Will the program crash?
- Does it compute the correct result?
- Does it leak private information?
- How long does it take to run?
- How much power does it consume?
- Will it turn off automated cruise control?

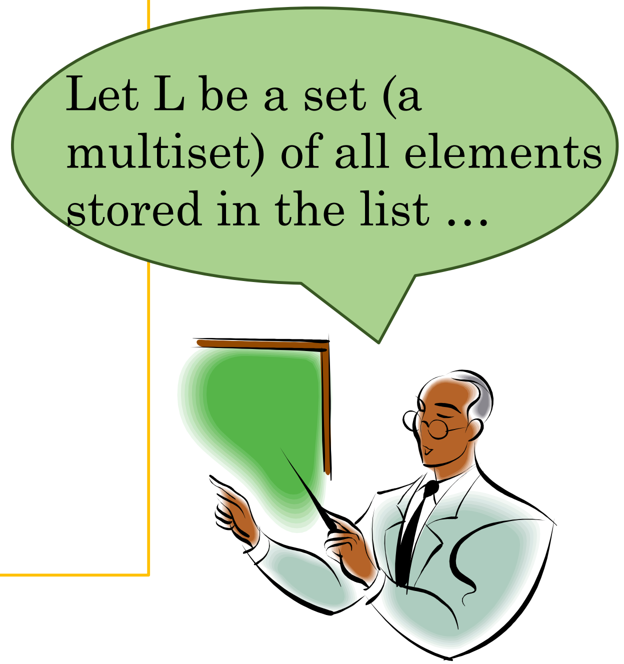
A Mathematical Proof of Program Correctness?

I just want to be sure that no element is lost in the list – if I insert an element, it is really there

```
lic void add (Object x)

ode e = new Node();
  e.data = x;
  e.next = root;
  root = e;
  size = size + 1;
}
```

A Mathematical Proof of Program Correctness?

```
//: L = data[root.next*]

public void add (Object x)

{
   Node e = new Node();
   e.data = x;
   e.next = root;
   root = e;
   size = size + 1;
}
```



A Mathematical Proof of Program Correctness?

Annotations

```
//: L = data[root.next*]
//: invariant: size = card L
public void add (Object x)
//: ensures L = old L + {x}
{
   Node e = new Node();
   e.data = x;
   e.next = root;
   root = e;
   size = size + 1;
}
```

Annotations

- Written by a programmer or a software analyst
- Added to the original program code to express properties that allow reasoning about the programs
- Examples:
  - Preconditions:
    - Describe properties of an input
  - Postconditions:
    - Describe what the program is supposed to do
  - Invariants:
    - Describe properties that have to hold in every program point

Decision Procedures for Collections

```
//: L = data[root.next*]
//: invariant: size = card L
public void add (Object x)
//: ensures L = old L + {x}
{
  Node e = new Node();
  e.data = x;
  e.next = root;
  root = e;
  si
}
```

Prove that the following formula always holds:

$\forall X.\ \forall L.\ |X| = 1 \rightarrow |\ L \uplus X\ | = |L| + 1$

Verification condition

Verification Conditions

- Mathematical formulas derived based on:
  - Code
  - Annotations
- If a verification condition always holds (valid), then to code is correct w.r.t. the given property
- It does not depend on the input variables
- If a verification condition does not hold, we should be able to detect an error in the code

Verification Condition: Example

```
//: assume (x > 0)
def simple (Int x)
//: ensures y > 0
{
  ??
  return y
}
```

Verification Condition: Example

```
//: assume (x > 0)
def simple (Int x)
//: ensures y > 0
{
  val y = x - 2
  return y
}
```

Verification condition:

$$\forall x.\ \forall y.\ x > 0 \ \wedge y = x - 2 \rightarrow y > 0$$

Verification Condition: Example

```
//: assume (x > 0)
def simple (Int x)
//: ensures y > 0
{
  val y = x - 2
  return y
}
```

Verification condition:

$$\forall x.\ \forall y.\ x > 0 \ \wedge y = x - 2 \rightarrow y > 0$$

Preconditions

Verification Condition: Example

```
//: assume (x > 0)
def simple (Int x)
//: ensures y > 0
{
  val y = x - 2
  return y
}
```

Verification condition:

$$\forall\, x.\ \forall\, y.\ x > 0\ \wedge \underbrace{y = x - 2}_{\text{Program}} \rightarrow y > 0$$

Verification Condition: Example

```
//: assume (x > 0)
def simple (Int x)
//: ensures y > 0
{
  val y = 1
  return y
}
```

Verification condition:

$$\forall x.\ \forall y.\ x > 0 \ \wedge y = 1 \rightarrow y > 0$$

Postconditions

Verification Condition: Example

```
//: assume (x > 0)
def simple (Int x)
//: ensures y > 0
{
  val y = x - 2
  return y
}
```

Verification condition:

$$\forall x.\ \forall y.\ x > 0\ \wedge y = x - 2 \rightarrow y > 0$$

Formula does not hold for input x = 1

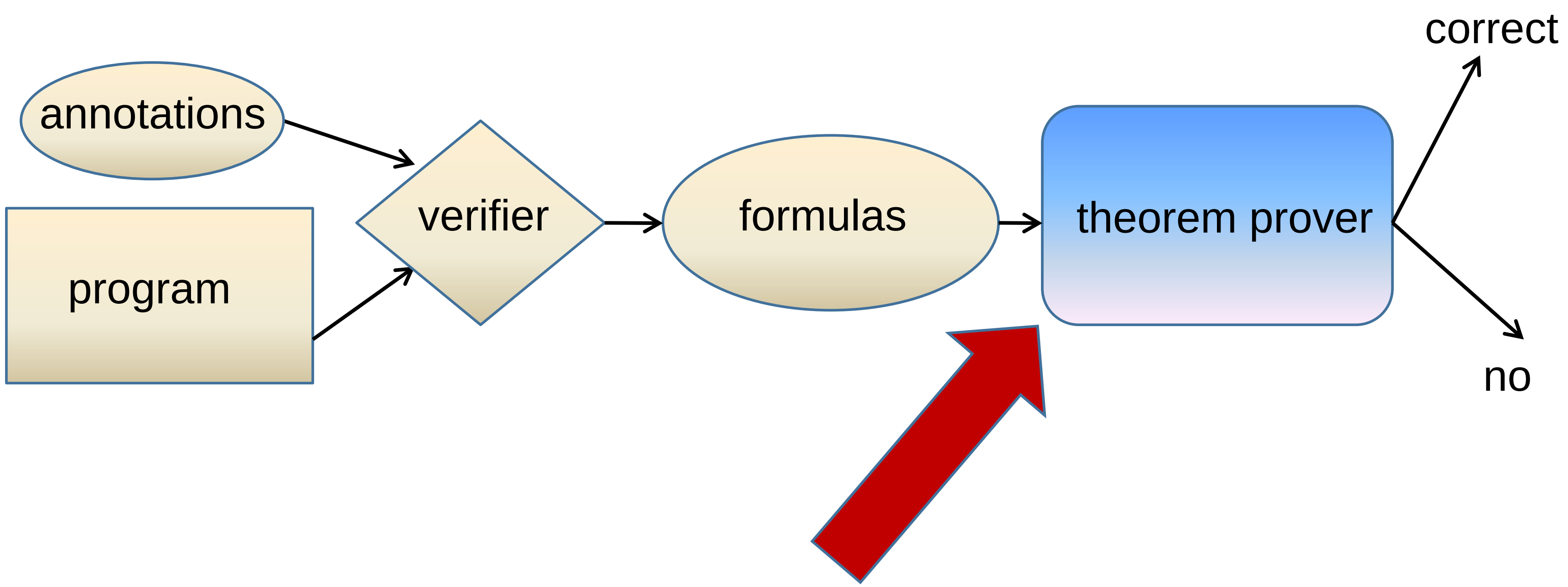
Automation of Verification

- Windows XP has approximately 45 millions lines of source code

$\cong$ 300.000 DIN A4 papers
$\cong$ seven times my size high paper stack

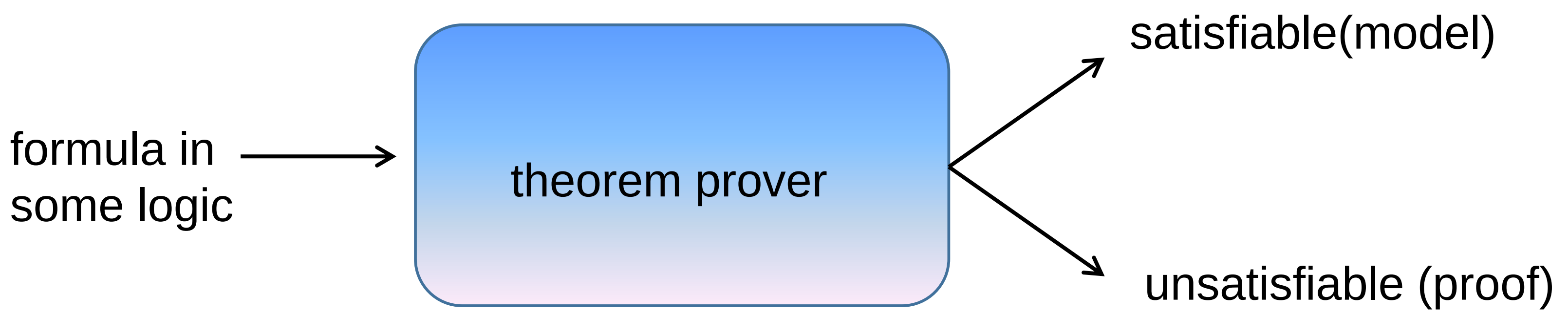
Verification should be automated!!!

Software Verification



Prove formulas automatically!

Decision Procedures



- A decision procedure is an algorithm which answers whether the input formula is satisfiable or not
  - formula $x \leq y$ is satisfiable for x=0, y=1
  - formula $x \leq y \wedge x+1 > y+1$ is unsatisfiable

Language Semantics

Formal Semantics of Java Programs

- The Java Language Specification (JLS) [link] gives semantics to Java programs
  - The document has 780 pages.
  - 148 pages to define semantics of expression.
  - 42 pages to define semantics of method invocation.
- Semantics is only defined in prose.
  - How can we make the semantics formal?
  - We need a mathematical model of computation.

Semantics of Programming Languages

- Denotational Semantics
  - Meaning of a program is defined as the mathematical object it computes (e.g., partial functions).
  - Example: Abstract Interpretation
- Axiomatic Semantics
  - Meaning of a program is defined in terms of its effect on the truth of logical assertions.
  - Example: Hoare Logic
- (Structural) Operational Semantics
  - Meaning of a program is defined by formalizing the individual computation steps of the program.
  - Example: Labeled Transition Systems

IMP: A Simple Imperative Language

Before we move on to Java, we look at a simple imperative programming language IMP.

An IMP program:

$p := 0;$

$x := 1;$

`while` $x \leq n$ `do`

  $x := x + 1;$

  $p := p + m;$

IMP: Syntactic Entities

- $n \in \mathbb{Z}$ – integers
- true, false $\in \mathbb{B}$ – Booleans
- $x,y \in Vars$ – Program variables
- $e \in Aexp$ – arithmetic expressions
- $b \in Bexp$ – Boolean expressions
- $c \in Com$ – commands

Syntax of Arithmetic Expressions

- Arithmetic expressions (*Aexp*)

$$
\begin{aligned}
e ::= \; & n, && \text{for } n \in \mathbb{Z} \\
\mid \; & x, && \text{for } x \in \mathit{Vars} \\
\mid \; & e_1 + e_2 \\
\mid \; & e_1 - e_2 \\
\mid \; & e_1 * e_2
\end{aligned}
$$

- Notes:
  - Variables are not declared before use.
  - All variables have integer type.
  - Expressions have no side-effects.

Syntax of Boolean Expressions

- Boolean expressions (*Bexp*)

  b ::= true

  $\mid$ false

  $\mid$ $e_1 = e_2$ for $e_1, e_2 \in Aexp$

  $\mid$ $e_1 \leq e_2$ for $e_1, e_2 \in Aexp$

  $\mid$ $\neg b$ for $b \in Bexp$

  $\mid$ $b_1 \wedge b_2$ for $b_1, b_2 \in Bexp$

  $\mid$ $b_1 \vee b_2$ for $b_1, b_2 \in Bexp$

Syntax of Commands

- Commands (Com)

$$
\begin{array}{rl}
c ::= & \texttt{skip} \\
& \mid x := e \\
& \mid c_1 \,;\, c_2 \\
& \mid \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \\
& \mid \texttt{while } b \texttt{ do } c
\end{array}
$$

- Notes:
  - The typing rules have been embedded in the syntax definition.
  - Other parts are not context-free and need to be checked separately (e.g., all variables are declared).
  - Commands contain all the side-effects in the language.
  - Missing: references, function calls, …

A simple example

```
//: assume (x > 5)
def simple (Int x)
//: ensures y > 7
{
  val y = x + 2
  return y
}
```

What do we need:

- Language in which we are writing programs
- Program execution and what does it mean "whenever the program is in the state"
- Language for annotation
- Combine all this somehow together

We need to express / derive / prove:
"whenever the program takes as input x, such that $x > 5$, and we execute the program, the resulting output y will satisfy that $y > 7$"

$$\forall x.\ \forall y.\ x > 5 \wedge y = x + 2 \rightarrow y > 7$$

Meaning of IMP Programs

Questions to answer:

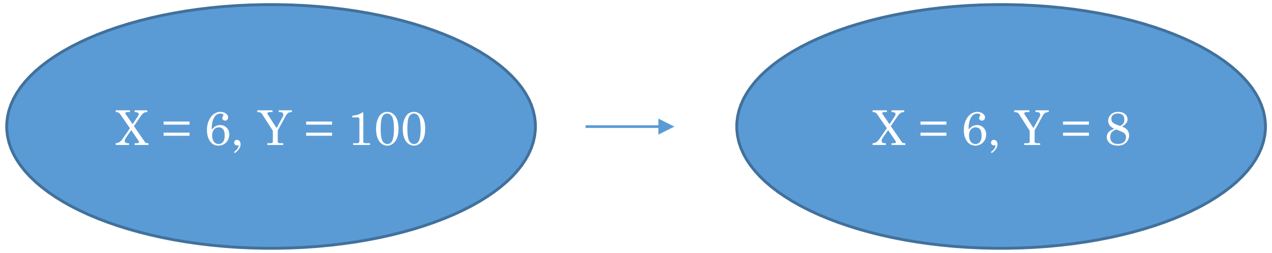
- What is the “meaning” of a given IMP expression/command?
- How would we evaluate IMP expressions and commands?
- How are the evaluator and the meaning related?
- How can we reason about the effect of a command?

Semantics of IMP

- The meaning of IMP expressions depends on the values of variables, i.e. the current state.
- A state at a given moment is represented as a function from $Vars$ to $\mathbb{Z}$
- The set of all states is $Q = Vars \to \mathbb{Z}$
- We use q to range over Q

A simple example

```
//: assume (x > 5)
def simple (Int x)
//: ensures y > 7
{
  val y = x + 2
  return y
}
```



Starting state, preconditions are satisfied

Ending state, postconditions are satisfied

Judgments

- We write $\langle e, q \rangle \Downarrow n$ to mean that e evaluates to n in state q.
  - The formula $\langle e, q \rangle \Downarrow n$ is a judgment
    (a statement about a relation between e, q and n)
  - In this case, we can view $\Downarrow$ as a function of two arguments e and q
- This formulation is called natural operational semantics
  - or big-step operational semantics
  - the judgment relates the expression and its "meaning"
- How can we define $\langle \mathrm{e1} + \mathrm{e2}, \mathrm{q} \rangle \Downarrow \ldots$?

Inference Rules

- We express the evaluation rules as inference rules for our judgments.
- The rules are also called evaluation rules.

An inference rule

$$\frac{F_1 \ldots F_n}{G} \quad \text{where } H$$

defines a relation between judgments $F_1,\ldots,F_n$ and G.

- The judgments $F_1,\ldots,F_n$ are the premises of the rule;
- The judgments G is the conclusion of the rule;
- The formula H is called the side condition of the rule.

If n=0 the rule is called an axiom. In this case, the line separating premises and conclusion may be omitted.

Inference Rules for *Aexp*

- In general, we have one rule per language construct:

$$\langle n, q\rangle \Downarrow n \qquad \text{Axiom} \qquad \langle x, q\rangle \Downarrow q(x)$$

$$\frac{\langle e_1, q\rangle \Downarrow n_1 \qquad \langle e_2, q\rangle \Downarrow n_2}{\langle e_1 + e_2, q\rangle \Downarrow (n_1 + n_2)} \qquad \frac{\langle e_1, q\rangle \Downarrow n_1 \qquad \langle e_2, q\rangle \Downarrow n_2}{\langle e_1 - e_2, q\rangle \Downarrow (n_1 - n_2)}$$

$$\frac{\langle e_1, q\rangle \Downarrow n_1 \qquad \langle e_2, q\rangle \Downarrow n_2}{\langle e_1 * e_2, q\rangle \Downarrow (n_1 * n_2)}$$

- This is called structural operational semantics.
  - rules are defined based on the structure of the expressions.

Inference Rules for *Bexp*

$<true, q> \Downarrow \text{true}$ $\qquad$ $<\text{false}, q> \Downarrow \text{false}$

$$\frac{<e_1, q> \Downarrow n_1 \qquad <e_2, q> \Downarrow n_2}{<e_1 = e_2, q> \Downarrow (n_1 = n_2)} \qquad \frac{<e_1, q> \Downarrow n_1 \qquad <e_2, q> \Downarrow n_2}{<e_1 \le e_2, q> \Downarrow (n_1 \le n_2)}$$

$$\frac{<b_1, q> \Downarrow t_1 \qquad <b_2, q> \Downarrow t_2}{<b_1 \land b_2, q> \Downarrow (t_1 \land t_2)}$$

How to Read Inference Rules?

- Forward, as derivation rules of judgments
  - if we know that the judgments in the premise hold then we can infer that the conclusion judgment also holds.
  - Example:

$$\frac{\langle 2, q\rangle \Downarrow 2 \qquad\qquad \langle 3, q\rangle \Downarrow 3}{\langle 2 * 3, q\rangle \Downarrow 6}$$

How to Read Inference Rules?

- Backward, as evaluation rules:
  - Suppose we want to evaluate $e_1 + e_2$, i.e., find n s.t. $e_1 + e_2 \Downarrow n$ is derivable using the previous rules.
  - By inspection of the rules we notice that the last step in the derivation of $e_1 + e_2 \Downarrow n$ **must be** the addition rule.
  - The other rules have conclusions that would not match $e_1 + e_2 \Downarrow$ n.
- This is called reasoning by inversion on the derivation rules.
  - Thus we must find n_1 and n_2 such that $e_1 \Downarrow n_1$ and $e_2 \Downarrow n_2$ are derivable.
  - This is done recursively.
- Since there is exactly one rule for each kind of expression, we say that the rules are syntax-directed.
  - At each step at most one rule applies.
  - This allows a simple evaluation procedure as above.

How to Read Inference Rules?

- Example: evaluation of an arithmetic expression via reasoning by inversion:

$$
\dfrac{<\mathrm{x}, \{x \mapsto 3, y \mapsto 2\}> \Downarrow 3 \qquad \dfrac{\begin{array}{c}<y, \{x \mapsto 3, y \mapsto 2\}> \Downarrow 2 \\ <2, \{x \mapsto 3, y \mapsto 2\}> \Downarrow 2\end{array}}{<2 * \mathrm{y}, \{x \mapsto 3, y \mapsto 2\}> \Downarrow 4}}{<\mathrm{x} + (2 * \mathrm{y}), \{x \mapsto 3, y \mapsto 2\}> \Downarrow 7}
$$

Semantics of Commands

- The evaluation of a command in *Com* has side-effects, but no direct result.
- The "result" of a command c in a pre-state q is a transition from q to a post-state q':

$$q \xrightarrow{c} q'$$

- We can formalize this in terms of transition systems.

Labeled Transition Systems

A labeled transition system (LTS) is a structure
$LTS = (Q, Act, \rightarrow)$ where

- Q is a set of states,
- Act is a set of actions,
- $\rightarrow \subseteq Q \times Act \times Q$ is a transition relation.

We write $q \xrightarrow{a} q'$ for $(q, a, q') \in \rightarrow$.

Inference Rules for Transitions

$$q \xrightarrow{\texttt{skip}} q$$

$$\frac{\langle e, q\rangle \Downarrow n}{q \xrightarrow{x := e} q \mathbin{++} \{x \mapsto n\}}$$

$$\frac{q \xrightarrow{c_1} q' \quad q' \xrightarrow{c_2} q''}{q \xrightarrow{c_1\,;\,c_2} q''}$$

$$\frac{\langle b, q\rangle \Downarrow \text{true} \quad q \xrightarrow{c_1} q'}{q \xrightarrow{\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2} q'}$$

$$\frac{\langle b, q\rangle \Downarrow \text{false} \quad q \xrightarrow{c_2} q'}{q \xrightarrow{\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2} q'}$$

$$\frac{\langle b, q\rangle \Downarrow \text{false}}{q \xrightarrow{\texttt{while } b \texttt{ do } c} q}$$

$$\frac{\langle b, q\rangle \Downarrow \text{true} \quad q \xrightarrow{c} q' \quad q' \xrightarrow{\texttt{while } b \texttt{ do } c} q''}{q \xrightarrow{\texttt{while } b \texttt{ do } c} q''}$$

Axiomatic Semantics

Axiomatic Semantics

- An axiomatic semantics consists of:
  - a language for stating assertions about programs;
  - rules for establishing the truth of assertions.
- Some typical kinds of assertions:
  - 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)

Assertions for IMP

- The assertions we make about IMP programs are of the form:

  $$\{A\}\ c\ \{B\}$$

  with the meaning that:
  - If A holds in state q , and $q \xrightarrow{c} q'$
  - then B holds in q'

- A is the precondition and B is the postcondition

- For example:

  $$\{\ y \leq x\ \}\ z := x;\ z := z + 1\ \{\ y < z\ \}$$

  is a valid assertion

- These are called Hoare triples or Hoare assertions

Assertions for IMP

- {A} c {B} is a partial correctness assertion. It does not imply termination of c.
- [A] c [B] is a total correctness assertion meaning that
  - If A holds in state q
  - then there exists q' such that $q \xrightarrow{c} q'$ and B holds in state q'
- Now let's be more formal
  - Formalize the language of assertions, A and B
  - Say when an assertion holds in a state
  - Give rules for deriving valid Hoare triples

The Assertion Language

- We use first-order predicate logic with IMP expressions

$$\begin{aligned} A ::= {} & \text{true} \mid \text{false} \mid e_1 = e_2 \mid e_1 \geq e_2 \\ & \mid A_1 \wedge A_2 \mid A_1 \vee A_2 \mid A_1 \Rightarrow A_2 \mid \forall x.A \mid \exists x.A \end{aligned}$$

- Note that we are somewhat sloppy and mix the logical variables and the program variables.
- Implicitly, all IMP variables range over integers.
- All IMP Boolean expressions are also assertions.

Semantics of Assertions

- We introduced a language of assertions, we need to assign meanings to assertions.
- Notation $q \vDash A$ says that assertion A holds in a given state q.
  - This is well-defined when q is defined on all variables occurring in A.
- The $\vDash$ judgment is defined inductively on the structure of assertions.
- Notation $\vDash A$ says that assertion A holds in any state, ie. it is always true.
- It relies on the semantics of arithmetic expressions from IMP.

Semantics of Assertions

- $q \vDash \text{true}$ always
- $q \vDash e_1 = e_2$ iff $<e_1,q>\Downarrow \,=\, <e_2,q>\Downarrow$
- $q \vDash e_1 \geq e_2$ iff $<e_1,q>\Downarrow \,\geq\, <e_2,q>\Downarrow$
- $q \vDash \mathrm{A}_1 \wedge \mathrm{A}_2$ iff $q \vDash \mathrm{A}_1$ and $\mathrm{q} \vDash \mathrm{A}_2$
- $q \vDash \mathrm{A}_1 \vee \mathrm{A}_2$ iff $q \vDash \mathrm{A}_1$ or $q \vDash \mathrm{A}_2$
- $q \vDash \mathrm{A}_1 \Rightarrow \mathrm{A}_2$ iff $q \vDash \mathrm{A}_1$ implies $q \vDash \mathrm{A}_2$
- $q \vDash \forall x.\mathrm{A}$ iff $\forall n \in \mathbb{Z}.\ q[x{:=}n] \vDash \mathrm{A}$
- $q \vDash \exists x.\mathrm{A}$ iff $\exists n \in \mathbb{Z}.\ q[x{:=}n] \vDash \mathrm{A}$

Semantics of Hoare Triples

- We can define formally the meaning of a partial correctness assertion:

$\vDash \{A\}\ c\ \{B\}$ iff $\forall q \in Q.\ \forall q' \in Q.\ q \vDash A \wedge q \xrightarrow{c} q' \Rightarrow q' \vDash B$

- and the meaning of a total correctness assertion:

$\vDash [A]\ c\ [B]$ iff $\forall q \in Q.\ q \vDash A \Rightarrow \exists q' \in Q.\ q \xrightarrow{c} q' \wedge q' \vDash B$

q – *a state, defines values of variables*

$\{A\}\ c\ \{B\}$ – *a Hoare tripe, it is either true or false*

$q \vDash F$ – *in state q, formula F holds*

Semantics of Hoare Triples

- We can define formally the meaning of a partial correctness assertion:

$\vDash \{A\}\ c\ \{B\}$ iff $\forall q \in Q.\ \forall q' \in Q.\ q \vDash A \wedge q \xrightarrow{c} q' \Rightarrow q' \vDash B$

- and the meaning of a total correctness assertion:

$\vDash [A]\ c\ [B]$ iff $\forall q \in Q.\ q \vDash A \Rightarrow \exists q' \in Q.\ q \xrightarrow{c} q' \wedge q' \vDash B$

Great result: we now formally can describe that a program is correct

Inferring Validity of Assertions

- Now we have the formal mechanism to decide when $\{A\}\ c\ \{B\}$
  - But it is not satisfactory,
  - because $\models \{A\}\ c\ \{B\}$ is defined in terms of the operational semantics.
  - We practically have to run the program to verify an assertion.
  - Also it is impossible to effectively verify the truth of a $\forall x.\ A$ assertion (by using the definition of validity)
- So we define a symbolic technique for deriving valid assertions from others that are known to be valid
  - We start with validity of first-order formulas

Now that we know what correctness means, what's next?

- By now we can express formally: *if a program is in the state where preconditions hold, and we execute the program, we will end up in the state where postconditions hold*
- For example, a formula $\{x > 0\}$ $x := x+2$ $\{x > 2\}$ is a correct Hoare triple and we can prove that by hand
- However, it is a manual work, therefore error-prone
- Goal: automatize the process of proving program correctness as much as possible
- End goal: develop a "push-the-button" tool for proving program correctness (something like a your own mini Dafny)

Natural Deduction

- Inference system introduces in 1934 in (Gentzen 1934, Jaśkowski 1934)
- The goal is to have a system that can automatically prove theorems in mathematics
- More reading:
  - https://www.iep.utm.edu/nat-ded/ (at Internet Encyclopedia of Philosophy)

Inference Rules

- We write $\vdash A$ when A can be inferred from basic axioms.
- The inference rules for $\vdash A$ are the usual ones from first-order logic with arithmetic (examples)
- Natural deduction style rules:

$$\frac{\vdash A \qquad \vdash B}{\vdash A \land B} \qquad \frac{\vdash A[a/x]}{\vdash \forall x.\ A} \ \text{where } a \text{ is fresh} \qquad \frac{\vdash \forall x.\ A}{\vdash A[e/x]}$$

$$\frac{\vdash A}{\vdash A \lor B} \qquad \frac{\vdash B}{\vdash A \lor B}$$

$$\frac{\vdash A[e/x]}{\vdash \exists x.\ A} \qquad \frac{\vdash \exists x.\ A \qquad \begin{array}{c}\vdash A[a/x] \\ \cdots \\ \vdash B\end{array}}{\vdash B} \ \text{where } a \text{ is fresh} \qquad \frac{\vdash A \Rightarrow B \qquad \vdash A}{\vdash B} \qquad \frac{\begin{array}{c}[\vdash A] \\ \cdots \\ \vdash B\end{array}}{\vdash A \Rightarrow B}$$

Inference Rules for Hoare Logic

- One rule for each syntactic construct:

$$\vdash \{A\}\ \texttt{skip}\ \{A\} \qquad\qquad \vdash \{A[e/x]\}\ x{:=}e\ \{A\}$$

$$\frac{\vdash \{A\}\ c_1\ \{B\} \qquad \vdash \{B\}\ c_2\ \{C\}}{\vdash \{A\}\ c_1; c_2\ \{C\}}$$

$$\frac{\vdash \{A \wedge b\}\ c_1\ \{B\} \qquad \vdash \{A \wedge \neg b\}\ c_2\ \{B\}}{\vdash \{A\}\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \{B\}}$$

$$\frac{\vdash \{\mathtt{I} \wedge b\}\ c\ \{\mathtt{I}\}}{\vdash \{\mathtt{I}\}\ \texttt{while}\ b\ \texttt{do}\ c\ \{\mathtt{I} \wedge \neg b\}}$$

Inference Rules for Hoare Logic

- One rule for each syntactic construct:

$$\vdash \{A\}\ \mathtt{skip}\ \{A\} \qquad\qquad \vdash \{A[e/x]\}\ x{:=}e\ \{A\}$$

$$\frac{\vdash \{A\}\ c_1\ \{B\} \qquad \vdash \{B\}\ c_2\ \{C\}}{\vdash \{A\}\ c_1;\ c_2\ \{C\}}$$

$$\frac{\vdash \{A \land b\}\ c_1\ \{B\} \qquad \vdash \{A \land \neg b\}\ c_2\ \{B\}}{\vdash \{A\}\ \mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2\ \{B\}}$$

$$\frac{\vdash \{I \land b\}\ c\ \{I\}}{\vdash \{I\}\ \mathtt{while}\ b\ \mathtt{do}\ c\ \{I \land \neg b\}}$$

Loop Invariants

- I is a loop invariant if the following three conditions hold:
  - I holds **initially** in all states satisfying Pre, when execution reaches loop entry, I holds
  - I is **preserved**: if we assume I and loop condition (*b*), we can prove that I will hold again after executing the loop body
  - I is **strong enough**: if we assume I and the negation of loop condition *b*, we can prove that Post holds after the loop execution

Inference Rules for Hoare Triples

- Similarly we write $\vdash \{A\}\ c\ \{B\}$ when we can derive the triple using inference rules
- There is one inference rule for each command in the language.
- Plus, the rule of consequence

$$\frac{\vdash A' \Rightarrow A \qquad \vdash \{A\}\ c\ \{B\} \qquad \vdash B \Rightarrow B'}{\vdash \{A'\}\ c\ \{B'\}}$$

Hoare Logic: Summary

- We have a language for asserting properties of programs.
- We know when such an assertion is true.
- We also have a symbolic method for deriving assertions.

semantics

$\vDash$ {A} P {B} is a valid statement (i.e. always true)

A

{A} P {B}

$\vDash$ A

$\vDash$ {A} P {B}

q $\vDash$ x > 0 "models"

soundness

~~q makes no sense~~

theorem proving

completeness

$\vdash$ A "we can prove A"

$\vdash$ A

$\vdash$ {A} P {B}

Hoare Rules

- For some constructs, multiple rules are possible

  alternative "forward axiom" for assignment:

  $$\vdash \{A\}\ x{:=}e\ \{\exists x_0.\ x_0 = e \land A[x_0/x]\}$$

  alternative rule for `while` loops:

  $$\frac{\vdash \mathtt{I} \land b \Rightarrow C \quad \vdash \{C\}\ \mathtt{c}\ \{\mathtt{I}\} \quad \vdash \mathtt{I} \land \neg b \Rightarrow B}{\vdash \{\mathtt{I}\}\ \mathtt{while}\ b\ \mathtt{do}\ c\ \{B\}}$$

- These alternative rules are derivable from the previous rules, plus the rule of consequence.

Exercise: Hoare Rules

- Is the following alternative rule for assignment still correct?

$$\vdash \{\text{true}\}\ x{:=}e\ \{x = e\}$$

Example: Conditional

$$\frac{\qquad}{\vdash \{\text{true}\}\ \texttt{if}\ y \leq 0\ \texttt{then}\ x := 1\ \texttt{else}\ x := y\ \{x > 0\}}$$

- D1 is obtained by consequence and assignment

$$\frac{\vdash \text{true} \wedge y < 0 \Rightarrow 1 \geq 0 \qquad \vdash \{1 \geq 0\}\ x := 1\ \{x \geq 0\}}{\vdash \{\text{true} \wedge y \leq 0\}\ x := 1\ \{x \geq 0\}}$$

- D2 is also obtained by consequence and assignment

$$\frac{\vdash \text{true} \wedge y > 0 \Rightarrow y > 0 \qquad \vdash \{y > 0\}\ x := y\ \{x > 0\}}{\vdash \{\text{true} \wedge y > 0\}\ x := y\ \{x > 0\}}$$

Example: a simple loop

- We want to infer that
  $\vdash \{x \leq 0\}\ \texttt{while}\ x \leq 5\ \texttt{do}\ x := x + 1\ \{x = 6\}$
- Use the rule for while with invariant $\texttt{I} \equiv x \leq 6$

$$\dfrac{\dfrac{\vdash x \leq 6 \wedge x \leq 5 \Rightarrow x + 1 \leq 6 \qquad \vdash \{x + 1 \leq 6\}\ x := x + 1\ \{x \leq 6\}}{\vdash \{x \leq 6 \wedge x \leq 5\}\ x := x + 1\ \{x \leq 6\}}}{\vdash \{\mathrm{x} \leq 6\}\ \texttt{while}\ \mathrm{x} \leq 5\ \texttt{do}\ \mathrm{x} := \mathrm{x} + 1\ \{\ \mathrm{x} \leq 6 \wedge \mathrm{x} > 5\}}$$

- Then finish-off with the rule of consequence

$$\dfrac{\begin{array}{l}\vdash x \leq 0 \Rightarrow x \leq 6 \\ \vdash x \leq 6 \wedge x > 5 \Rightarrow x = 6 \qquad \vdash \{x \leq 6\}\ \texttt{while} \ldots \{x \leq 6 \wedge x > 5\}\end{array}}{\vdash \{\mathrm{x} \leq 0\}\ \texttt{while} \ldots \{\mathrm{x} = 6\}}$$

Example: a more interesting program

- We want to derive that

$\{n \geq 0\}$

$p := 0;$

$x := 0;$

$\texttt{while}\ x < n\ \texttt{do}$

$\quad x := x + 1;$

$\quad p := p + m$

$\{p = n * m\}$

Example: a more interesting program

Only applicable rule (except for rule of consequence):

$$\frac{\vdash \{A\}\ c_1 \{C\} \qquad \vdash \{C\}\ c_2\ \{B\}}{\vdash \{A\}\ c_1;\ c_2\ \{B\}}$$

$$\frac{\vdash \{n \geq 0\}\ p{:=}0;\ x{:=}0\ \{C\} \qquad \vdash \{C\}\ \text{while } x < n \text{ do } (x{:=}x+1;\ p{:=}p+m)\ \{p = n * m\}}{\vdash \underbrace{\{n \geq 0\}}_{A}\ \underbrace{p{:=}0;\ x{:=}0}_{c_1};\ \underbrace{\text{while } x < n \text{ do } (x{:=}x+1;\ p{:=}p+m)}_{c_2}\ \underbrace{\{p = n * m\}}_{B}}$$

Example: a more interesting program

What is C?Look at the next possible matching rules for c_2!

Only applicable rule (except for rule of consequence):

$$\frac{\vdash \{I \land b\}\ c\ \{I\}}{\vdash \{I\}\ \texttt{while}\ b\ \texttt{do}\ c\ \{I \land \neg b\}}$$

We can match {I} with {C} but we cannot match {I ∧ ¬b} and {p = n * m} directly. Need to apply the rule of consequence first!

$$\frac{\vdash \{n \geq 0\}\ p{:=}0;\ x{:=}0\ \{C\} \qquad \vdash \{C\}\ \text{while } x < n \text{ do } (x{:=}x+1;\ p{:=}p+m)\ \{p = n * m\}}{\vdash \underbrace{\{n \geq 0\}}_{A}\ \underbrace{p{:=}0;\ x{:=}0;}_{c_1}\ \underbrace{\text{while } x < n \text{ do } (x{:=}x+1;\ p{:=}p+m)}_{c_2}\ \underbrace{\{p = n * m\}}_{B}}$$

Example: a more interesting program

What is C?Look at the next possible matching rules for c_2!

Only applicable rule (except for rule of consequence):

$$\frac{\vdash \{I \wedge b\}\ c\ \{I\}}{\vdash \underbrace{\{I\}}_{A}\ \underbrace{\texttt{while}\ b\ \texttt{do}\ c}_{c'}\ \underbrace{\{I \wedge \neg b\}}_{B}}$$

$I = A = A' = C$

Rule of consequence:

$$\frac{\vdash A' \Rightarrow A \qquad \vdash \{A\}\ c'\ \{B\} \qquad \vdash B \Rightarrow B'}{\vdash \{A'\}\ c'\ \{B'\}}$$

$$\frac{\vdash \{n \geq 0\}\ p{:=}0;\ x{:=}0\ \{C\} \qquad \vdash \overbrace{\{C\}}^{A'}\ \overbrace{\text{while } x < n \text{ do } (x{:=}x+1;\ p{:=}p+m)}^{c'}\ \overbrace{\{p = n * m\}}^{B'}}{\vdash \{n \geq 0\}\ p{:=}0;\ x{:=}0;\ \text{while } x < n \text{ do } (x{:=}x+1;\ p{:=}p+m)\ \{p = n * m\}}$$

Example: a more interesting program

What is $\mathtt{I}$? Let's keep it as a placeholder for now!

Next applicable rule:

$$\frac{\vdash \{A\}\ c_1\ \{C\} \qquad \vdash \{C\}\ c_2\ \{B\}}{\vdash \{A\}\ c_1;\ c_2\ \{B\}}$$

$$\dfrac{\vdash \{n \geq 0\}\ p{:=}0;\ x{:=}0\ \{\mathtt{I}\} \qquad \dfrac{\dfrac{\vdash \overbrace{\{\mathtt{I} \land x<n\}}^{A}\ \overbrace{x := x+1;}^{c_1}\ \overbrace{p{:=}p+m}^{c_2}\ \overbrace{\{\mathtt{I}\}}^{B}}{\vdash \{\mathtt{I}\}\ \text{while } x < n \text{ do } (x{:=}x+1;\ p{:=}p+m)\ \{\mathtt{I} \land x \geq n\}} \qquad \vdash \mathtt{I} \land x \geq n \Rightarrow p = n * m}{\vdash \{\mathtt{I}\}\ \text{while } x < n \text{ do } (x{:=}x+1;\ p{:=}p+m)\ \{p = n * m\}}}{\vdash \{n \geq 0\}\ p{:=}0;\ x{:=}0;\ \text{while } x < n \text{ do } (x{:=}x+1;\ p{:=}p+m)\ \{p = n * m\}}$$

Example: a more interesting program

What is C?Look at the next possible matching rules for c_2!

Only applicable rule (except for rule of consequence):

$\vdash \{A[e/x]\}\ x{:=}e\ \{A\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash \overbrace{\{I \land x<n\}}^{A}\ \overbrace{x := x+1}^{c_1}\ \{C\} \qquad \vdash \{C\}\ \overbrace{p{:=}p+m}^{c_2}\ \overbrace{\{I\}}^{B}}{\vdash \{I \land x<n\}\ x := x+1;\ p{:=}p+m\ \{I\}}}{\vdash \{I\}\ \text{while } x < n \text{ do } (x{:=}x+1;\ p{:=}p+m)\ \{I \land x \geq n\}} \quad \vdash I \land x \geq n \Rightarrow p = n * m}{\vdash \{n \geq 0\}\ p{:=}0;\ x{:=}0\ \{I\} \quad \vdash \{I\}\ \text{while } x < n \text{ do } (x{:=}x+1;\ p{:=}p+m)\ \{p = n * m\}}}{\vdash \{n \geq 0\}\ p{:=}0;\ x{:=}0;\ \text{while } x < n \text{ do } (x{:=}x+1;\ p{:=}p+m)\ \{p = n * m\}}$$

Example: a more interesting program

What is C?Look at the next possible matching rules for c_2!

Only applicable rule (except for rule of consequence):

$\vdash \{A[e/x]\}\ x{:=}e\ \{A\}$

$$\dfrac{\vdash \{I \land x<n\}\ x{:=}x+1\ \{I[p+m/p]\} \qquad \vdash \{I[p+m/p]\}\ p{:=}p+m\ \{I\}}{\vdash \{I \land x<n\}\ x{:=}x+1;\ p{:=}p+m\ \{I\}}$$

$$\dfrac{\vdash \{I \land x<n\}\ x{:=}x+1;\ p{:=}p+m\ \{I\}}{\vdash \{I\}\ \text{while } x < n \text{ do } (x{:=}x+1;\ p{:=}p+m)\ \{I \land x \geq n\}}$$

$$\dfrac{\vdash \{I\}\ \text{while } x < n \text{ do } (x{:=}x+1;\ p{:=}p+m)\ \{I \land x \geq n\} \qquad \vdash I \land x \geq n \Rightarrow p = n * m}{\vdash \{I\}\ \text{while } x < n \text{ do } (x{:=}x+1;\ p{:=}p+m)\ \{p = n * m\}}$$

$$\dfrac{\vdash \{n \geq 0\}\ p{:=}0;\ x{:=}0\ \{I\} \qquad \vdash \{I\}\ \text{while } x < n \text{ do } (x{:=}x+1;\ p{:=}p+m)\ \{p = n * m\}}{\vdash \{n \geq 0\}\ p{:=}0;\ x{:=}0;\ \text{while } x < n \text{ do } (x{:=}x+1;\ p{:=}p+m)\ \{p = n * m\}}$$

Example: a more interesting program

Only applicable rule (except for rule of consequence):

$\vdash \{A[e/x]\}\ x{:=}e\ \{A\}$

Need rule of consequence to match $\{I \wedge x<n\}$ and $\{I[x+1/x,\ p+m/p]\}$

$$\dfrac{\vdash \{n \geq 0\}\ p{:=}0;\ x{:=}0\ \{I\} \qquad \dfrac{\begin{array}{c}\dfrac{\dfrac{\vdash \{I \wedge x<n\}\ x{:=}x+1\ \{I[p+m/p]\} \qquad \vdash \{I[p+m/p]\}\ p{:=}p+m\ \{I\}}{\vdash \{I \wedge x<n\}\ x{:=}x+1;\ p{:=}p+m\ \{I\}}}{\vdash \{I\}\ \text{while } x<n \text{ do } (x{:=}x+1;\ p{:=}p+m)\ \{I \wedge x \geq n\}} \\ \vdash I \wedge x \geq n \Rightarrow p = n * m\end{array}}{\vdash \{I\}\ \text{while } x<n \text{ do } (x{:=}x+1;\ p{:=}p+m)\ \{p = n * m\}}}{\vdash \{n \geq 0\}\ p{:=}0;\ x{:=}0;\ \text{while } x<n \text{ do } (x{:=}x+1;\ p{:=}p+m)\ \{p = n * m\}}$$

Example: a more interesting program

Let's just remember the open proof obligations!

$$\vdash \{I[x+1/x,\ p+m/p]\}\ x{:=}x+1\ \{I[p+m/p]\}$$

$$\vdash I \land x < n \Rightarrow I[x+1/x,\ p+m/p]$$

$$\vdash \{I \land x<n\}\ x{:=}x+1\ \{I[p+m/p]\} \qquad \vdash \{I[p+m/p]\}\ p{:=}p+m\ \{I\}$$

$$\vdash \{I \land x<n\}\ x{:=}x+1;\ p{:=}p+m\ \{I\}$$

$$\vdash \{I\}\ \text{while } x < n \text{ do } (x{:=}x+1;\ p{:=}p+m)\ \{I \land x \geq n\}$$

$$\vdash I \land x \geq n \Rightarrow p = n * m$$

$$\vdash \{n \geq 0\}\ p{:=}0;\ x{:=}0\ \{I\} \qquad \vdash \{I\}\ \text{while } x < n \text{ do } (x{:=}x+1;\ p{:=}p+m)\ \{p = n * m\}$$

$$\vdash \{n \geq 0\}\ p{:=}0;\ x{:=}0;\ \text{while } x < n \text{ do } (x{:=}x+1;\ p{:=}p+m)\ \{p = n * m\}$$

Example: a more interesting program

Let's just remember the open proof obligations!

$\vdash \mathtt{I} \wedge x < n \Rightarrow \mathtt{I}[x+1/x,\ p+m/p]$

$\vdash \mathtt{I} \wedge x \geq n \Rightarrow p = n * m$

Continue with the remaining part of the proof tree, as before.

$$\dfrac{\dfrac{\dfrac{\vdash n \geq 0 \Rightarrow \mathtt{I}[0/p,\ 0/x] \qquad \vdash \{\mathtt{I}[0/p,\ 0/x]\}\ p{:=}0\ \{\mathtt{I}[0/x]\}}{\vdash \{n \geq 0\}\ p{:=}0\ \{\mathtt{I}[0/x]\}} \qquad \vdash \{\mathtt{I}[0/x]\}\ x{:=}0\ \{\mathtt{I}\}}{\vdash \{n \geq 0\}\ p{:=}0;\ x{:=}0\ \{\mathtt{I}\}} \qquad \dfrac{\vdots}{\vdash \{\mathtt{I}\}\ \text{while } x < n \text{ do } (x{:=}x+1;\ p{:=}p+m)\ \{p = n * m\}}}{\vdash \{n \geq 0\}\ p{:=}0;\ x{:=}0;\ \text{while } x < n \text{ do } (x{:=}x+1;\ p{:=}p+m)\ \{p = n * m\}}$$

Now we only need to solve the remaining constraints!

Example: a more interesting program

Find I such that all constraints are simultaneously valid:

$\vdash n \geq 0 \Rightarrow I[0/p, 0/x]$

$\vdash I \wedge x < n \Rightarrow I[x+1/x, p+m/p]$

$\vdash I \wedge x \geq n \Rightarrow p = n * m$

$I \equiv p = x * m \wedge x \leq n$

$\vdash n \geq 0 \Rightarrow 0 = 0 * m \wedge 0 \leq n$

$\vdash p = x * m \wedge x \leq n \wedge x < n \Rightarrow p+m = (x+1) * m \wedge x+1 \leq n$

$\vdash p = x * m \wedge x \leq n \wedge x \geq n \Rightarrow p = n * m$

All constraints are valid!

Another example: check if a number is prime

$\{n \geq 2\}$

$p := 1;$

$i := 2;$

while $i < n$ do

  if $(n$ **mod** $i = 0)$ then $p := 0;$

  $i := i + 1;$

$\{p = 1 \Rightarrow prime(n)\}$

Another example: check if a number is prime

$\{n \geq 2\}$

$p := 1;$

$i := 2;$

while $i < n$ do

  if $(n$ **mod** $i = 0)$ then $p := 0;$

  $i := i + 1;$

$\{p = 1 \Rightarrow \forall k.\ (2 \leq k \wedge k < n \Rightarrow n\ \mathbf{mod}\ k \neq 0)\}$

Another example: check if a number is prime

$\{n \geq 2\}$

$p := 1;$

$i := 2;$

while $i < n$ do

  if (n **mod** $i = 0$) then $p := 0;$

  $i := i + 1;$

$\{p = 1 \Rightarrow prime(n)\}$

Invariant:
$I \equiv (p = 1 \Rightarrow prime(n)) \wedge i \leq n$

Using Hoare Rules

- Hoare rules are mostly syntax directed
- There are three obstacles to automation of Hoare logic proofs:
  - When to apply the rule of consequence?
  - What invariant to use for `while`?
  - How do you prove the implications involved in the rule of consequence?
- The last one is how theorem proving gets in the picture
  - This turns out to be doable!
  - The loop invariants turn out to be the hardest problem!
  - Should the programmer give them?

Computing VC

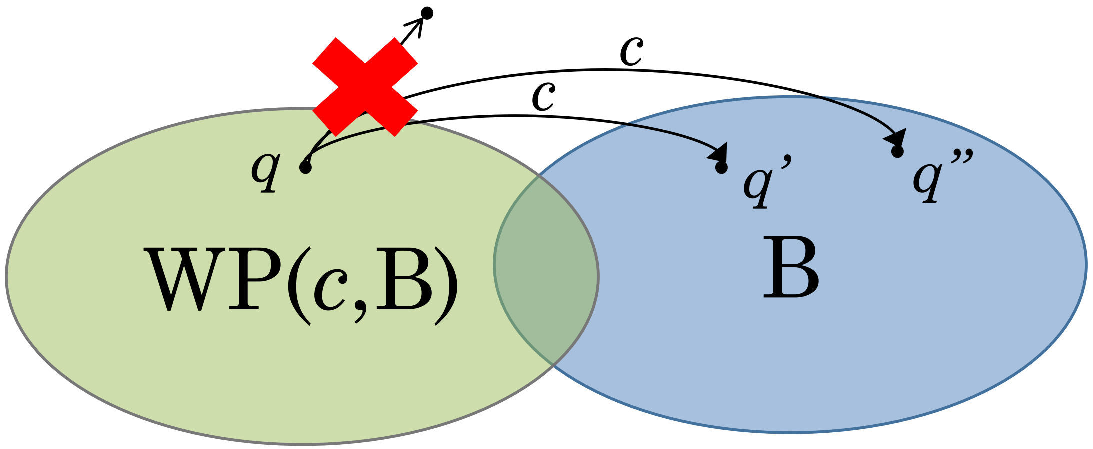
Verification Condition Generation

- Idea for VC generation: propagate the post-condition backwards through the program:
  - From {A} P {B}
  - Generate formula $A \Rightarrow F(P, B)$, where $F(P, B)$ is a formula describing the starting states for program to end in B
- This backwards propagation $F(P, B)$ can be formalized in terms of weakest preconditions.

Weakest Preconditions

- The weakest precondition WP(c,B) holds for any state q whose c-successor states all satisfy B:

  $q \vDash \mathrm{WP}(c,\mathrm{B}) \quad \text{iff} \quad \forall q' \in Q.\ q \xrightarrow{c} q' \Rightarrow q' \vDash \mathrm{B}$

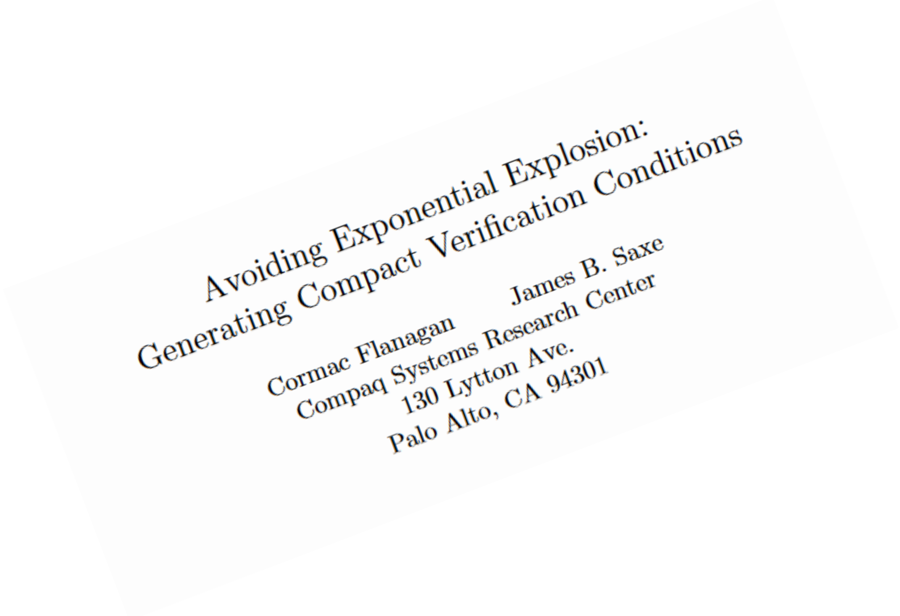


- Compute WP(P,B) recursively according to the structure of the program P.

Loop-Free Guarded Commands

- Introduce loop-free guarded commands as an intermediate representation of the verification condition

- c ::= assume b
  | assert b
  | havoc x
  | c_1 ; c_2
  | $c_1 \square c_2$


Avoiding Exponential Explosion:
Generating Compact Verification Conditions

Cormac Flanagan James B. Saxe
Compaq Systems Research Center
130 Lytton Ave.
Palo Alto, CA 94301

From Programs to Guarded Commands

- GC(`skip`) =

  assume true

- GC(x := e) =

  assume $tmp = x$; havoc x; assume ($x = e[tmp/x]$)

  where tmp is fresh

- GC(c_1 ; c_2) =
  GC(c_1) ; GC(c_2)

- GC(`if` b `then` c_1 `else` c_2) = ?

- GC({`I`} `while` b `do` c) = ?

From Programs to Guarded Commands

- GC(`skip`) =

  assume true

- GC($x := e$) =

  assume $tmp = x$; havoc x; assume ($x = e[tmp/x]$)

  where tmp is fresh

- GC(c_1 ; c_2) =
  GC(c_1) ; GC(c_2)

- GC(`if` b `then` c_1 `else` c_2) =
  (assume b; GC(c_1)) $\square$ (assume $\neg b$; GC(c_2))

- GC({`I`} `while` b `do` c) = ?

Guarded Commands for Loops

- GC({`I`} `while` b `do` c) =

  assert `I`;
  havoc x_1; ...; havoc x_n;
  assume `I`;
  (assume b; GC(c); assert `I`; assume false)
  assume $\neg b$

where x_1, ..., x_n are the variables modified in c

Example: VC Generation

$\{n \geq 0\}$

$p := 0;$

$x := 0;$

$\{p = x * m \land x \leq n\}$

$\texttt{while}\ x < n\ \texttt{do}$

$\quad x := x + 1;$

$\quad p := p + m$

$\{p = n * m\}$

Example: VC Generation

- Computing the guarded command

$\{ n \geq 0 \}$

assume $p_0 = p$; havoc p; assume $p = 0$;

assume $x_0 = x$; havoc x; assume $x = 0$;

assert $p = x * m \land x \leq n$;

havoc x; havoc p; assume $p = x * m \land x \leq n$;

(assume $x < n$;

assume $x_1 = x$; havoc x; assume $x = x_1 + 1$;
assume $p_1 = p$; havoc p; assume $p = p_1 + m$;
assert $p = x * m \land x \leq n$; assume false)

 assume $x \geq n$;

$\{ p = n * m \}$

Computing Weakest Preconditions

- WP(assume b, B) = $b \Rightarrow$ B
- WP(assert b, B) = $b \wedge$ B
- WP(havoc x, B) = B[a/x]   (${a}$ fresh in B)
- WP($c_1;c_2$, B) = WP(c_1, WP(c_2, B))
- WP($c_1 \square c_2$,B) = WP(c_1, B) $\wedge$ WP(c_2, B)

Putting Everything Together

- Given a Hoare triple $H \vdash \{A\}\ P\ \{B\}$
- Compute c_H = assume A; GC(P); assert B
- Compute $VC_H = WP(c_H, \text{true})$
- Check $\vdash VC_H$ using a theorem prover.

Example: VC Generation

- Computing the weakest precondition

  WP (assume $n \geq 0$;

  assume $p_0 = p$; havoc p; assume $p = 0$;

  assume $x_0 = x$; havoc x; assume $x = 0$;

  assert $p = x * m \wedge x \leq n$;

  havoc x; havoc p; assume $p = x * m \wedge x \leq n$;

  (assume $x < n$;

  assume $x_1 = x$; havoc x; assume $x = x_1 + 1$;

  assume $p_1 = p$; havoc p; assume $p = p_1 + m$;

  assert $p = x * m \wedge x \leq n$; assume false)

  □ assume $x \geq n$;

  assert $p = n * m$, true)

Example: VC Generation

- Computing the weakest precondition

WP (assume n $\geq$ 0;

assume $p_0 = p$; havoc p; assume $p = 0$;

assume $x_0 = x$; havoc x; assume $x = 0$;

assert p = x * m $\land$ x $\leq$ n;

havoc x; havoc p; assume p = x * m $\land$ x $\leq$ n;

(assume x < n;

assume x_1 = x; havoc x; assume x = x_1 + 1;

assume $p_1 = p$; havoc p; assume $p = p_1 + m$;

assert p = x * m $\land$ x $\leq$ n; assume false)

 assume x $\geq$ n, $p = n * m$)

Example: VC Generation

- Computing the weakest precondition

WP (assume $n \geq 0$;

assume $p_0 = p$; havoc p; assume $p = 0$;

assume $x_0 = x$; havoc x; assume $x = 0$;

assert $p = x * m \land x \leq n$;

havoc x; havoc p; assume $p = x * m \land x \leq n$,

WP ((assume $x < n$;

assume $x_1 = x$; havoc x; assume $x = x_1 + 1$;

assume $p_1 = p$; havoc p; assume $p = p_1 + m$;

assert $p = x * m \land x \leq n$; assume false)

□ assume $x \geq n$, $p = n * m$))

Example: VC Generation

- Computing the weakest precondition

$\mathrm{WP}\ ($assume $n \geq 0;$

assume $p_0 = p;$ havoc $p;$ assume $p = 0;$

assume $x_0 = x;$ havoc $x;$ assume $x = 0;$

assert $p = x * m \wedge x \leq n;$

havoc $x;$ havoc $p;$ assume $p = x * m \wedge x \leq n,$

$\mathrm{WP}\ ($assume $x < n;$

assume $x_1 = x;$ havoc $x;$ assume $x = x_1 + 1;$
assume $p_1 = p;$ havoc $p;$ assume $p = p_1 + m;$
assert $p = x * m \wedge x \leq n;$ assume false, $p = n * m)$

$\wedge\ \mathrm{WP}\ ($assume $x \geq n,\ p = n * m))$

Example: VC Generation

- Computing the weakest precondition

WP (assume $n \geq 0$;

assume $p_0 = p$; havoc p; assume $p = 0$;

assume $x_0 = x$; havoc x; assume $x = 0$;

assert $p = x * m \land x \leq n$;

havoc x; havoc p; assume $p = x * m \land x \leq n$,

WP (assume $x < n$;

assume $x_1 = x$; havoc x; assume $x = x_1 + 1$;

assume $p_1 = p$; havoc p; assume $p = p_1 + m$;

assert $p = x * m \land x \leq n$; assume false, $p = n * m$)

$\land\, x \geq n \Rightarrow p = n * m$)

Example: VC Generation

- Computing the weakest precondition

WP (assume n $\geq$ 0;

assume $p_0 = p$; havoc p; assume $p = 0$;

assume $x_0 = x$; havoc x; assume $x = 0$;

assert $p = x * m \wedge x \leq n$;

havoc x; havoc p; assume $p = x * m \wedge x \leq n$,

WP (assume $x < n$;

assume $x_1 = x$; havoc x; assume $x = x_1 + 1$;
assume $p_1 = p$; havoc p; assume $p = p_1 + m$;
assert $p = x * m \wedge x \leq n$, WP (assume false, $p = n * m$)

$\wedge\ x \geq n \Rightarrow p = n * m$)

Example: VC Generation

- Computing the weakest precondition

$\mathrm{WP}\ (\mathrm{assume}\ n \geq 0;$

$\quad \mathrm{assume}\ p_0 = p;\ \mathrm{havoc}\ p;\ \mathrm{assume}\ p = 0;$

$\quad \mathrm{assume}\ x_0 = x;\ \mathrm{havoc}\ x;\ \mathrm{assume}\ x = 0;$

$\quad \mathrm{assert}\ p = x * m \wedge x \leq n;$

$\quad \mathrm{havoc}\ x;\ \mathrm{havoc}\ p;\ \mathrm{assume}\ p = x * m \wedge x \leq n,$

$\quad \mathrm{WP}\ (\mathrm{assume}\ x < n;$

$\qquad \mathrm{assume}\ x_1 = x;\ \mathrm{havoc}\ x;\ \mathrm{assume}\ x = x_1 + 1;$

$\qquad \mathrm{assume}\ p_1 = p;\ \mathrm{havoc}\ p;\ \mathrm{assume}\ p = p_1 + m;$

$\qquad \mathrm{assert}\ p = x * m \wedge x \leq n,\ \mathrm{false} \Rightarrow p = n * m)$

$\quad \wedge\ x \geq n \Rightarrow p = n * m)$

Example: VC Generation

- Computing the weakest precondition

WP (assume $n \geq 0$;

assume $p_0 = p$; havoc p; assume $p = 0$;

assume $x_0 = x$; havoc x; assume $x = 0$;

assert $p = x * m \wedge x \leq n$;

havoc x; havoc p; assume $p = x * m \wedge x \leq n$,

WP (assume $x < n$;

assume $x_1 = x$; havoc x; assume $x = x_1 + 1$;

assume $p_1 = p$; havoc p; assume $p = p_1 + m$;

assert $p = x * m \wedge x \leq n$, true)

$\wedge\ x \geq n \Rightarrow p = n * m$)

Example: VC Generation

- Computing the weakest precondition

$\mathrm{WP}\ (\text{assume } n \geq 0;$

$\quad \text{assume } p_0 = p;\ \text{havoc } p;\ \text{assume } p = 0;$

$\quad \text{assume } x_0 = x;\ \text{havoc } x;\ \text{assume } x = 0;$

$\quad \text{assert } p = x * m \wedge x \leq n;$

$\quad \text{havoc } x;\ \text{havoc } p;\ \text{assume } p = x * m \wedge x \leq n,$

$\quad \mathrm{WP}\ (\text{assume } x < n;$

$\quad\quad \text{assume } x_1 = x;\ \text{havoc } x;\ \text{assume } x = x_1 + 1;$

$\quad\quad \text{assume } p_1 = p;\ \text{havoc } p;\ \text{assume } p = p_1 + m,$

$\quad\quad p = x * m \wedge x \leq n)$

$\quad \wedge\ x \geq n \Rightarrow p = n * m)$

Example: VC Generation

- Computing the weakest precondition

WP (assume $n \geq 0$;

assume $p_0 = p$; havoc p; assume $p = 0$;

assume $x_0 = x$; havoc x; assume $x = 0$;

assert $p = x * m \wedge x \leq n$;

havoc x; havoc p; assume $p = x * m \wedge x \leq n$,

WP (assume $x < n$;

assume $x_1 = x$; havoc x; assume $x = x_1 + 1$;

assume $p_1 = p$; havoc p,

$p = p_1 + m \Rightarrow p = x * m \wedge x \leq n$)

$\wedge\ x \geq n \Rightarrow p = n * m$)

Example: VC Generation

- Computing the weakest precondition

$\text{WP}\,(\text{assume } n \geq 0;$

$\quad \text{assume } p_0 = p;\ \text{havoc } p;\ \text{assume } p = 0;$

$\quad \text{assume } x_0 = x;\ \text{havoc } x;\ \text{assume } x = 0;$

$\quad \text{assert } p = x * m \wedge x \leq n;$

$\quad \text{havoc } x;\ \text{havoc } p;\ \text{assume } p = x * m \wedge x \leq n,$

$\quad \text{WP}\,(\text{assume } x < n;$

$\qquad \text{assume } x_1 = x;\ \text{havoc } x;\ \text{assume } x = x_1 + 1,$

$\qquad p_1 = p \wedge\ pa_1 = p_1 + m \ \Rightarrow pa_1 = x * m \wedge x \leq n)$

$\quad \wedge\, x \ \geq n \Rightarrow p = n * m)$

Example: VC Generation

- Computing the weakest precondition

$\mathrm{WP}\ (\text{assume } n \geq 0;$

$\quad \text{assume } p_0 = p; \text{ havoc } p; \text{ assume } p = 0;$

$\quad \text{assume } x_0 = x; \text{ havoc } x; \text{ assume } x = 0;$

$\quad \text{assert } p = x * m \wedge x \leq n;$

$\quad \text{havoc } x; \text{ havoc } p; \text{ assume } p = x * m \wedge x \leq n,$

$\quad \mathrm{WP}\ (\text{assume } x < n; \text{ assume } x_1 = x; \text{ havoc } x,$

$\qquad x = x_1 + 1 \wedge p_1 = p \wedge pa_1 = p_1 + m$

$\qquad \Rightarrow pa_1 = x * m \wedge x \leq n)$

$\quad \wedge x \geq n \Rightarrow p = n * m)$

Example: VC Generation

- Computing the weakest precondition

$WP\ (\text{assume } n \geq 0;$

$\quad \text{assume } p_0 = p; \text{ havoc } p; \text{ assume } p = 0;$

$\quad \text{assume } x_0 = x; \text{ havoc } x; \text{ assume } x = 0;$

$\quad \text{assert } p = x * m \wedge x \leq n;$

$\quad \text{havoc } x; \text{ havoc } p; \text{ assume } p = x * m \wedge x \leq n,$

$\quad WP\ (\text{assume } x < n; \text{ assume } x_1 = x,$

$\qquad xa_1 = x_1 + 1 \wedge p_1 = p \wedge pa_1 = p_1 + m$

$\qquad \Rightarrow pa_1 = xa_1 * m \wedge xa_1 \leq n)$

$\quad \wedge\ x \geq n \Rightarrow p = n * m)$

Example: VC Generation

- Computing the weakest precondition

WP (assume $n \geq 0$;

$\quad$ assume $p_0 = p$; havoc p; assume $p = 0$;

$\quad$ assume $x_0 = x$; havoc x; assume $x = 0$;

$\quad$ assert $p = x * m \wedge x \leq n$;

$\quad$ havoc x; havoc p; assume $p = x * m \wedge x \leq n$,

$\quad$ WP (assume $x < n$,

$$x_1 = x \wedge xa_1 = x_1 + 1 \wedge p_1 = p \wedge pa_1 = p_1 + m$$

$$\Rightarrow pa_1 = xa_1 * m \wedge xa_1 \leq n)$$

$$\wedge\, x \geq n \Rightarrow p = n * m)$$

Example: VC Generation

- Computing the weakest precondition

$\mathrm{WP}\ (\mathrm{assume}\ n \geq 0;$

$\mathrm{assume}\ p_0 = p;\ \mathrm{havoc}\ p;\ \mathrm{assume}\ p = 0;$

$\mathrm{assume}\ x_0 = x;\ \mathrm{havoc}\ x;\ \mathrm{assume}\ x = 0;$

$\mathrm{assert}\ p = x * m \wedge x \leq n;$

$\mathrm{havoc}\ x;\ \mathrm{havoc}\ p;\ \mathrm{assume}\ p = x * m \wedge x \leq n,$

$(x < n \wedge x_1 = x \wedge xa_1 = x_1 + 1 \wedge p_1 = p \wedge pa_1 = p_1 + m$

$\Rightarrow pa_1 = xa_1 * m \wedge xa_1 \leq n)$

$\wedge\ x \geq n \Rightarrow p = n * m)$

Example: VC Generation

- Computing the weakest precondition

$$n \geq 0 \wedge p_0 = p \wedge pa_3 = 0 \wedge x_0 = x \wedge xa_3 = 0 \Rightarrow pa_3 = xa_3 * m \wedge xa_3 \leq n \wedge$$

$$(pa_2 = xa_2 * m \wedge xa_2 \leq n \Rightarrow$$

$$((xa_2 < n \wedge x_1 = xa_2 \wedge xa_1 = x_1 + 1 \wedge$$

$$p_1 = pa_2 \wedge pa_1 = p_1 + m) \Rightarrow pa_1 = xa_1 * m \wedge xa_1 \leq n)$$

$$\wedge (xa_2 \geq n \Rightarrow pa_2 = n * m))$$

Example: VC Generation

- The resulting VC is equivalent to the conjunction of the following implications

$$n \geq 0 \wedge p_0 = p \wedge pa_3 = 0 \wedge x_0 = x \wedge xa_3 = 0 \Rightarrow$$
$$pa_3 = xa_3 * m \wedge xa_3 \leq n$$

$$n \geq 0 \wedge p_0 = p \wedge pa_3 = 0 \wedge x_0 = x \wedge xa_3 = 0 \wedge pa_2 = xa_2 * m \wedge$$
$$xa_2 \leq n \Rightarrow$$
$$xa_2 \geq n \Rightarrow pa_2 = n * m$$

$$n \geq 0 \wedge p_0 = p \wedge pa_3 = 0 \wedge x_0 = x \wedge xa_3 = 0 \wedge pa_2 = xa_2 * m \wedge xa_2 < n$$
$$\wedge\ x_1 = xa_2 \wedge xa_1 = x_1 + 1 \wedge p_1 = pa_2 \wedge pa_1 = p_1 + m \Rightarrow$$
$$pa_1 = xa_1 * m \wedge xa_1 \leq n$$

Example: VC Generation

- simplifying the constraints yields

$$n \geq 0 \Rightarrow 0 = 0 * m \wedge 0 \leq n$$

$$xa_2 \leq n \wedge xa_2 \geq n \Rightarrow xa_2 * m = n * m$$

$$xa_2 < n \Rightarrow xa_2 * m + m = (xa_2 + 1) * m \wedge xa_2 + 1 \leq n$$

- all of these implications are valid, which proves that the original Hoare triple was valid, too.

Translating Method Calls to GCs

```
method m (p₁: T_1, ..., pₖ: Tₖ) returns (r: T)
    requires P
    modifies x₁, ..., xₙ
    ensures Q
```

A method call

```
y := m(y₁, ..., yₖ);
```

is desugared into the guarded command

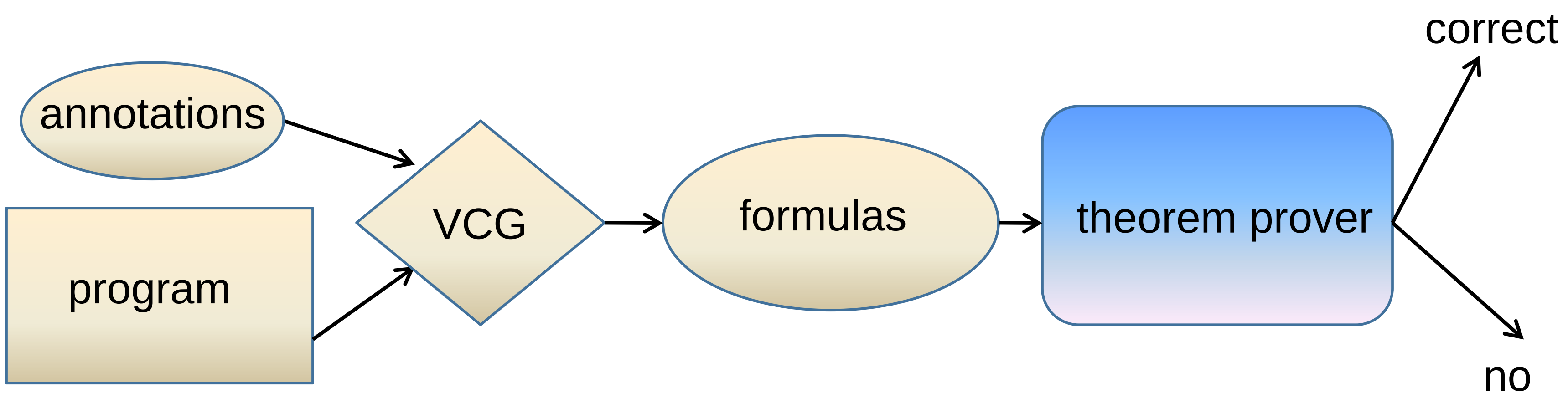
assert $P[y_1/p_1, \ldots, y_k/p_k]$;

havoc x_1; ..., havoc x_n; havoc y;

assume $Q[y_1/p_1, \ldots, y_k/p_k, y/r]$

```
ddickstein:proj1$ ./exec.sh
precondition: { n ≥ 0 }
p := 0;
x := 0;
invariant: { (p = x * m) ∧ (x ≤ n) }
while x < n do
  x := x + 1;
  p := p + m;
postcondition: { p = n * m }

assume n ≥ 0;
assume p0 = p;
havoc p;
assume p = 0;
assume x0 = x;
havoc x;
assume x = 0;
assert (p = x * m) ∧ (x ≤ n);
havoc x;
havoc p;
assume (p = x * m) ∧ (x ≤ n);
(
  assume x < n;
  assume x1 = x;
  havoc x;
  assume x = x1 + 1;
  assume p1 = p;
  havoc p;
  assume p = p1 + m;
  assert (p = x * m) ∧ (x ≤ n);
  assume false;
) □ (
  assume ¬(x < n);
)
assert p = n * m;

(n ≥ 0) ∧ (p0 = p) ∧ (pa3 = 0) ∧ (x0 = x) ∧ (xa3 = 0) → (pa3 = xa3 * m) ∧ (xa3 ≤ n) ∧ (
  (pa2 = xa2 * m) ∧ (xa2 ≤ n) → (
    (xa2 < n) ∧ (x1 = xa2) ∧ (xa1 = x1 + 1) ∧ (p1 = pa2) ∧ (pa1 = p1 + m) → (pa1 = xa1 * m) ∧ (xa1 ≤ n)
  ) ∧ ((xa2 ≥ n) → (pa2 = n * m))
)
```

Software Verification



Adding arrays to language

- Given command: a[i] := v
- In array theory a := write(a, i, v)
- GC: assume $tmp = a$; havoc a; assume (a = write(tmp, i, v))

WP(GC, F) = WP(assume $tmp = a$; havoc a; assume (a = write(tmp, i, v)), F)

= WP(assume $tmp = a$; havoc a; a = write(tmp, i, v) ⇒ F)

= WP(assume $tmp = a$; af = write(tmp, i, v) ⇒ F[af/a])

= $tmp = a$ ⇒ af = write(tmp, i, v) ⇒ F[af/a]

= $tmp = a$ ∧ af = write(tmp, i, v) ⇒ F[af/a]

= af = write(a, i, v) ⇒ F[af/a]

Disclaimer: you need to learn a theory of arrays first

What we have learned so far

- Hoare logic reduces program verification to proving the validity of verification conditions expressed as statements in some assertion logic
- The actual verification process can be completely mechanized modulo
    1. inference of loop invariants / procedure contracts
    2. the actual validity checking of the generated VCs