SMT-based Verification of Heap-manipulating Programs

Ruzica Piskac

VTSA 2016

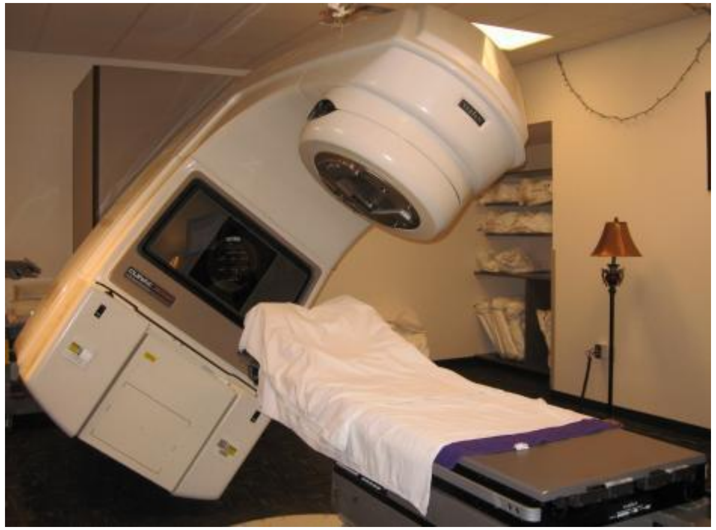
Maiden Flight of Ariane 5 Rocket

- Ariane 5 exploded on its first test flight in 1996
- Cause: failure of flight-control software due to overflow in floating point to integer conversion

- Financial loss: $500,000,000 (including indirect costs: $2,000,000,000)

Therac-25

- Radiation therapy machine
- Two modes:
  - X-ray
  - electron-beam



- Race condition in software caused use of electron-beam instead of X-ray
- six cases of radiation poisoning between 1985 and 1987, three of them fatal

Economics of Software Errors

Estimated annual costs of software errors in the US (2002)

$60 billion (0.6% of GDP)

Estimated size of the US software industry (2002)

$240 billion (50% development)

Estimated

50%

of each software project is spent on testing

Economics of Software Errors

Recent research at Cambridge University (2013, link) showed that the global cost of software bugs is

around 312 billion of dollars annually

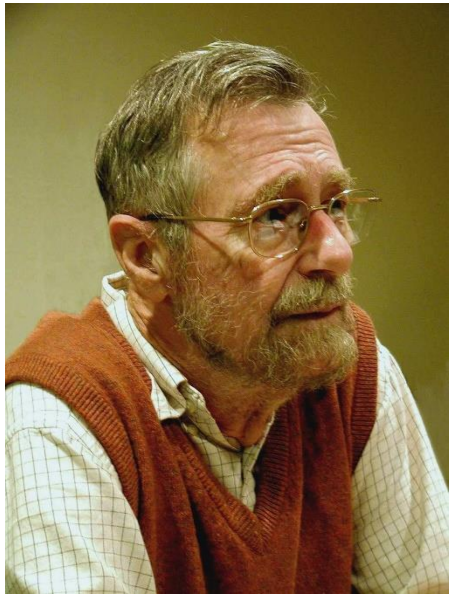
Testing

Software validation the “old-fashioned” way:

- Create a test suite (set of test cases)
- Run the test suite
- Fix the software if test suite fails
- Ship the software if test suite passes

“Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.”

Edsger W. Dijkstra



Very hard to test the portion inside the “if" statement!

```
input x
if (hash(x) == 10) {
    ...
}
```

Verification

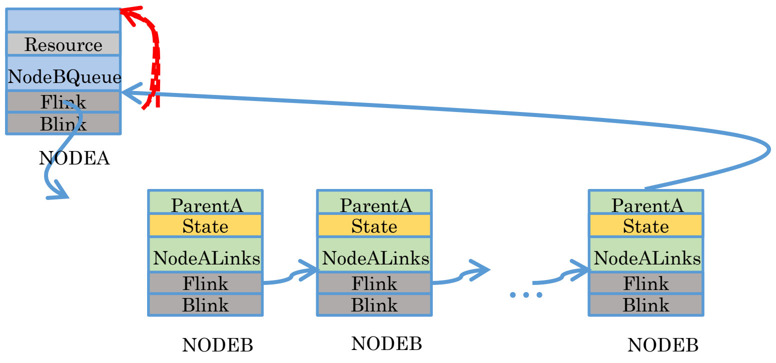
- **Verification:** formally prove that a computing system satisfies its specifications
  - **Rigor**: well established mathematical foundations
  - **Exhaustiveness**: considers **all** possible behaviors of the system, i.e., finds **all** errors
  - **Automation**: uses computers to build reliable computers

Success Stories of Formal Methods

- Astrée Static Analyzer
  - Developed by Patrick Cousot's group and others
  - Verify absence of runtime errors in C code for Embedded Systems
  - Industrial applications include verification of Airbus fly-by-wire software

Success Stories of Formal Methods

- SLAM, Static Driver Verifier, HAVOC, and VCC
  - Developed at Microsoft Research
  - Verification of OS code
  - Applications:
    - Windows Device Drivers
    - Windows File System
    - Windows Hypervisor

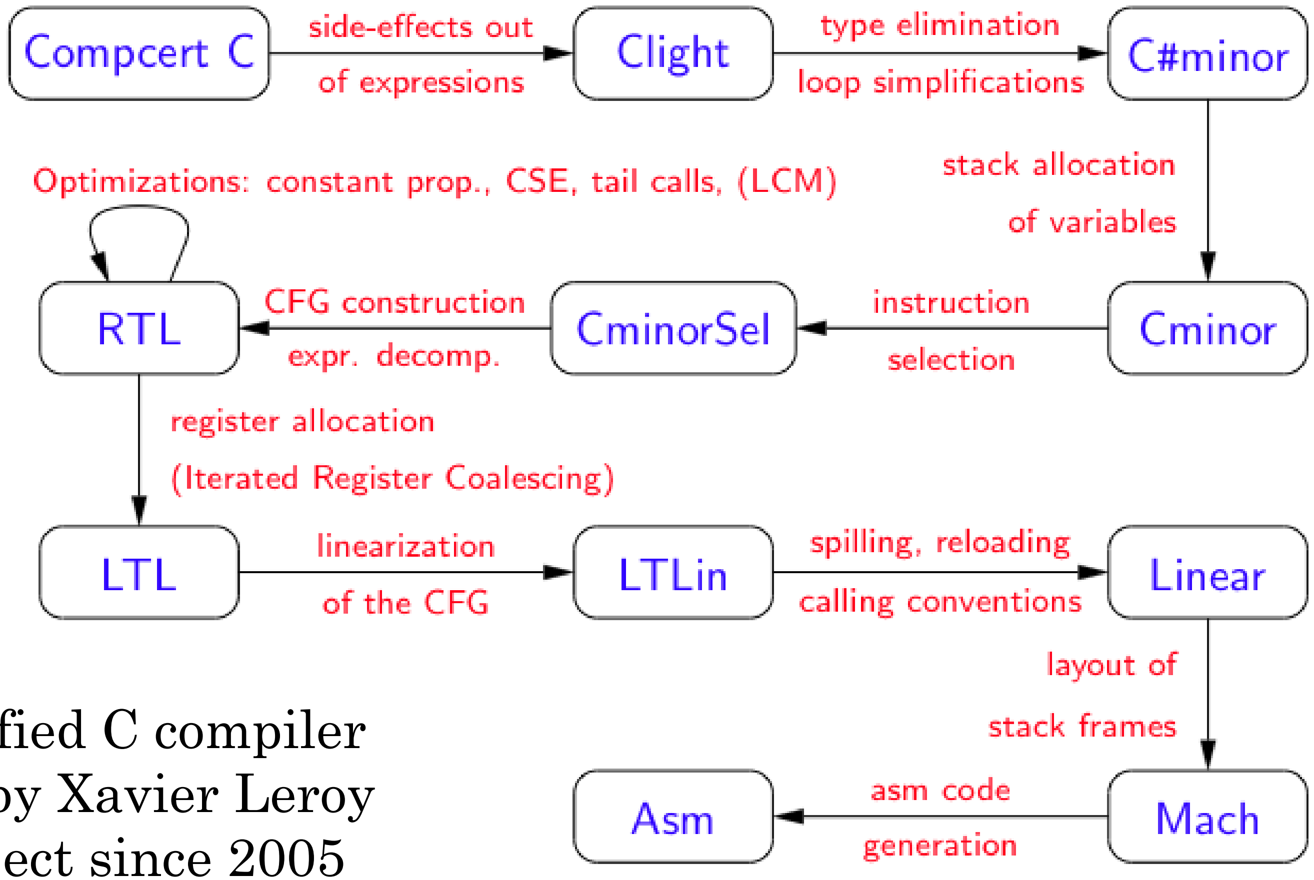



```
#define FIRST_CHILD(x) x->NodeBQueue.Flink
#define NEXT_NODE(x) x->NodeAlinks.Flink

__type_invariant(PNODEA x){
    ENCL_NODEA(FIRST_CHILD(x)) != x ==>
      ENCL_NODEB(FIRST_CHILD(x))->ParentA == x
}

__type_invariant(PNODEB y){
    NEXT_NODE(y) != &(y->ParentA->NodeBQueue) ==>
      y->ParentA == ENCL_NODEB(NEXT_NODE(y))->ParentA
}
```

Compcert Compiler



- Formally verified C compiler
- A project led by Xavier Leroy
- An active project since 2005
- Commercial licenses since 2015
- http://compcert.inria.fr/

Compcert Compiler

- Miscompilation happens

We created a tool that generates random C programs, and then spent two and a half years using it to find compiler bugs. So far, we have reported more than 325 previously unknown bugs to compiler developers. Moreover, every compiler that we tested has been found to crash and also to silently generate wrong code when presented with valid inputs.

X. Yang, Y. Chen, E. Eide, J. Regehr, PLDI 2011.

Compcert Compiler

The striking thing about our CompCert results is that the middleend bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users.

X. Yang, Y. Chen, E. Eide, J. Regehr, PLDI 2011.

This course: From Programs to Formulas

- Next few slides describe informally how to derive formulas from program
- In this course we will learn more details about it – you can even implement your own verification condition generator, see:

http://www.cs.yale.edu/homes/piskac/teaching/vtsa2016.html

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;
}
```

Let L be a set (a multiset) of all elements stored in the list …

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:

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

Program

Verification Condition: Example

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

Verification condition:

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

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

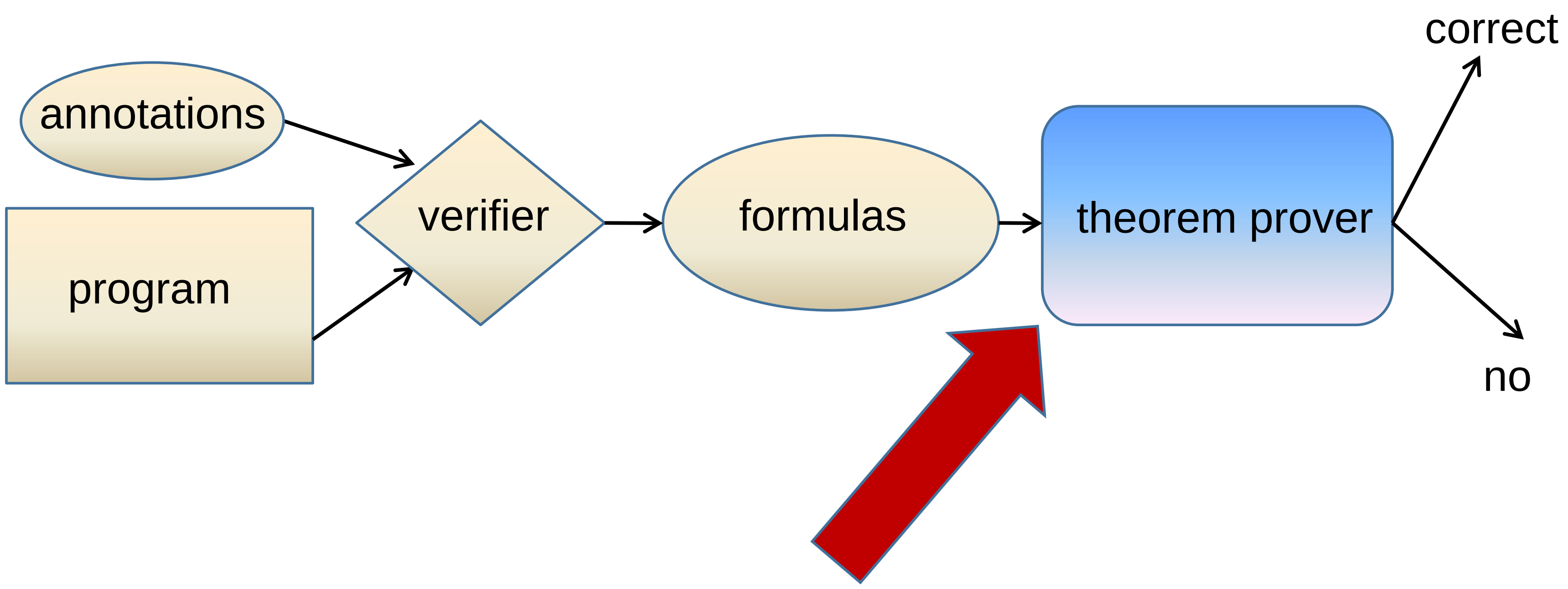
Automation of Verification

- Windows XP has approximately 45 millions lines of source code

≅ 300.000 DIN A4 papers
≅ 12m high paper stack

Verification should be automated!!!

Software Verification



Prove formulas automatically!

How to prove program correctness?

Proving program correctness

```
def f(x : Int, y : Int) : Int {
   if (y == 0)
     return 0
   } else {
   if  (y % 2 == 0) {
     val  z = f(x, y / 2);
     return 2*z
   } else {
     return x + f(x, y - 1)
   }
}
```

- Does f terminate?
- What does f compute?

Proving program correctness

Using mathematical notation:

$$f(x,y) = \begin{cases} 0, & \text{if } y = 0 \\ 2f(x, \lfloor \frac{y}{2} \rfloor), & \text{if } y > 0, \text{ and } y = 2k \text{ for some } k \\ x + f(x, y-1), & \text{if } y > 0, \text{ and } y = 2k+1 \text{ for some } k \end{cases}$$

- Does f terminate?
- What does f compute?

Annotations

- To prove program correctness we need annotations
  - Otherwise we do not know what we are supposed to prove
- 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

How can we automate verification?

Important algorithmic questions:

- verification condition generation: compute formulas expressing program correctness
  - Hoare logic, weakest precondition, strongest postcondition
- theorem proving: prove verification conditions
  - proof search, counterexample search
  - decision procedures
- loop invariant inference
  - predicate abstraction
  - abstract interpretation and data-flow analysis
  - pointer analysis
- reasoning about numerical computation
- pre-condition and post-condition inference
- ranking error reports and warnings
- finding error causes from counterexample traces

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.

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

- $n \in \mathsf{Z}$ – integers
- true,false $\in \mathsf{B}$ – Booleans
- $x,y \in L$ – locations (program variables)
- $e \in Aexp$ – arithmetic expressions
- $b \in Bexp$ – Boolean expressions
- $c \in Com$ – commands

Syntax of Arithmetic Expressions

- Arithmetic expressions (*Aexp*)
  $e ::= n$ for $n \in \mathbb{Z}$
  $\quad \mid e_1 + e_2$
  $\quad \mid e_1 - e_2$
  $\quad \mid e_1 * e_2$
- 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

  | false

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

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

  | $\neg\, b$ for $b \in Bexp$

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

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

Syntax of Commands

- Commands (Com)

  c ::= `skip`
  | $x := e$
  | $c_1 ; c_2$
  | `if` b `then` c_1 `else` c_2
  | `while` b `do` c

- 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, …

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 L to Z^{m}
- The set of all states is $Q = L \rightarrow \mathsf{Z}^{\mathsf{m}}$
- We use q to range over Q

Judgments

- We write $<e, q> \Downarrow n$ to mean that e evaluates to n in state q.
  - The formula $<e, q> \Downarrow n$ 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 $<\mathrm{e1} + \mathrm{e2}, \mathrm{q}> \Downarrow \ldots$?

Inference Rules for *Aexp*

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

$$<n, q> \Downarrow n \qquad \text{Axiom} \qquad <x, q> \Downarrow q(x)$$

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

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

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

Inference Rules for *Bexp*

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

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

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

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

$$\frac{<b_1, q> \Downarrow t_1 \qquad <e_2, q> \Downarrow t_2}{<b_1 \wedge b_2, q> \Downarrow (t_1 \wedge t_2)}$$

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

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

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

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

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

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

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

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

Axiomatic Semantics

- 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

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

- 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.
- It relies on the semantics of arithmetic expressions from IMP.

Semantics of Assertions

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

Inferring Validity of Assertions

- Now we have the formal mechanism to decide when $\{A\}\ c\ \{B\}$
  - But it is not satisfactory,
  - because $\vDash \{A\}\ c\ \{B\}$ 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

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.
- Natural deduction style rules:

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

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

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

Tony Hoare

Inference Rules for Hoare Logic

- One rule for each syntactic construct:

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

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

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

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

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 (e), 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 e, 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 Rules

- For some constructs, multiple rules are possible

  alternative "forward axiom" for assignment:

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

  alternative rule for `while` loops:

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

- 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

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

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 $\texttt{I} \equiv x \leq 6$

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

Example: a more interesting program

- We want to derive that

$\{n \geq 0\}$

$p := 0;$

$x := 0;$

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

$\quad x := x + 1;$

$\quad p := p + m$

$\{p = n * m\}$

Inference Rules for Hoare Logic

- One rule for each syntactic construct:

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

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

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

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

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

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

Example: a more interesting program

- We want to derive that

$\{\mathrm{n} \geq 0\}$

$p := 0;$

$x := 0;$

`while` $x < n$ `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):

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

We can match $\{\mathtt{I}\}$ with $\{C\}$ but we cannot match $\{\mathtt{I} \wedge \neg b\}$ 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):

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

Rule of consequence:

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

$\mathtt{I} = A = A' = C$

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

Example: a more interesting program

What is I? Let's keep it as a placeholder for now!

Next applicable rule:

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

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

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):

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

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

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):

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

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

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

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

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

Example: a more interesting program

Let's just remember the open proof obligations!

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

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

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

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

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

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

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

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

Example: a more interesting program

Let's just remember the open proof obligations!

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

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

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

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

Now we only need to solve the remaining constraints!

Example: a more interesting program

Find $\mathtt{I}$ such that all constraints are simultaneously valid:

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

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

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

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

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

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

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

All constraints are valid!

Exercise:

{true}

$x := n$; $y := m$;

(`if` $0 \leq n$ `then` $z := -1$ `else` $z := 1$);

$\{ I \}$

`while` $x \neq 0$ `do`

$y := y + z$;

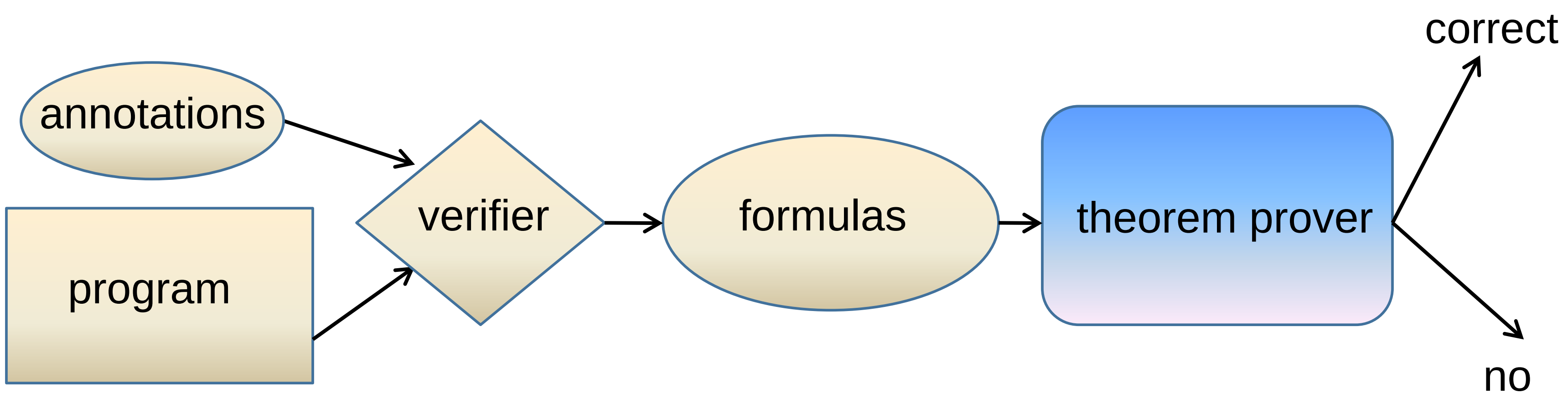
$x := x + z$;

$\{y = m - n\}$

Using Hoare Rules

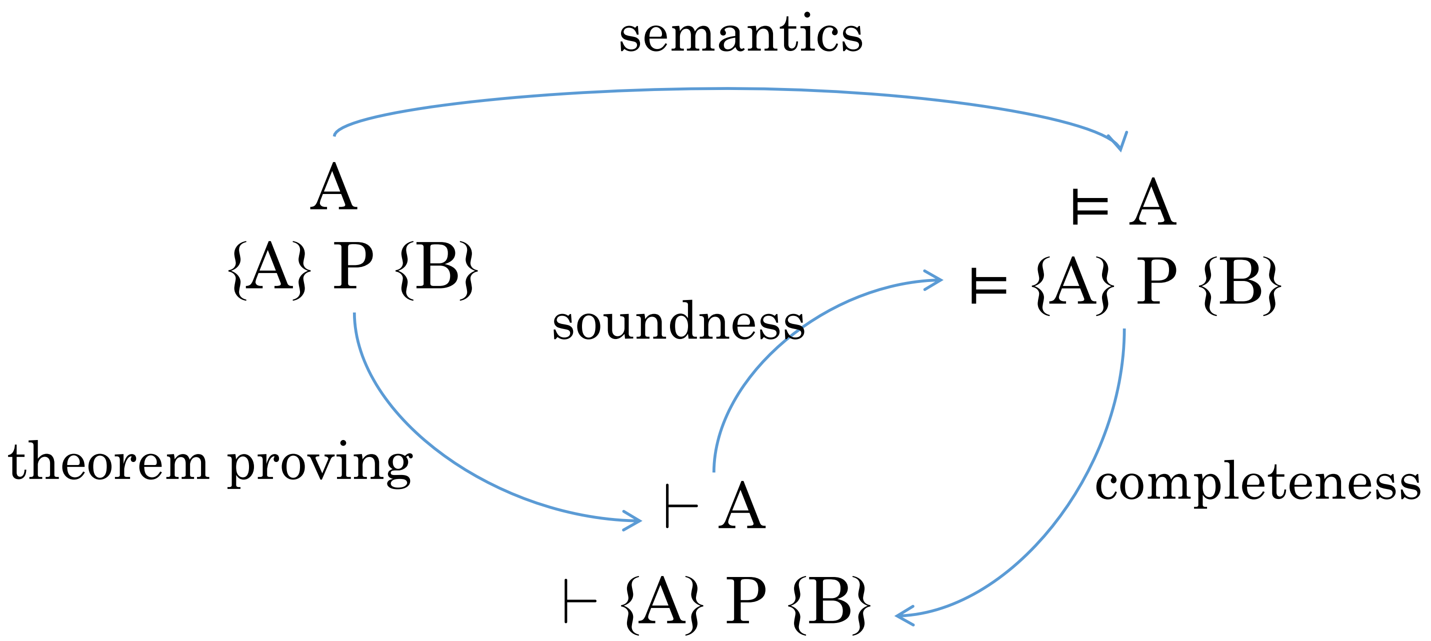
- Hoare rules are mostly syntax directed
- There are three obstacles to automation of Hoare logic proofs:
  - When to apply the rule of consequence?
  - What invariant to use for `while`?
  - How do you prove the implications involved in the rule of consequence?
- The last one is how theorem proving 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?

Software Verification



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.



Computing VC

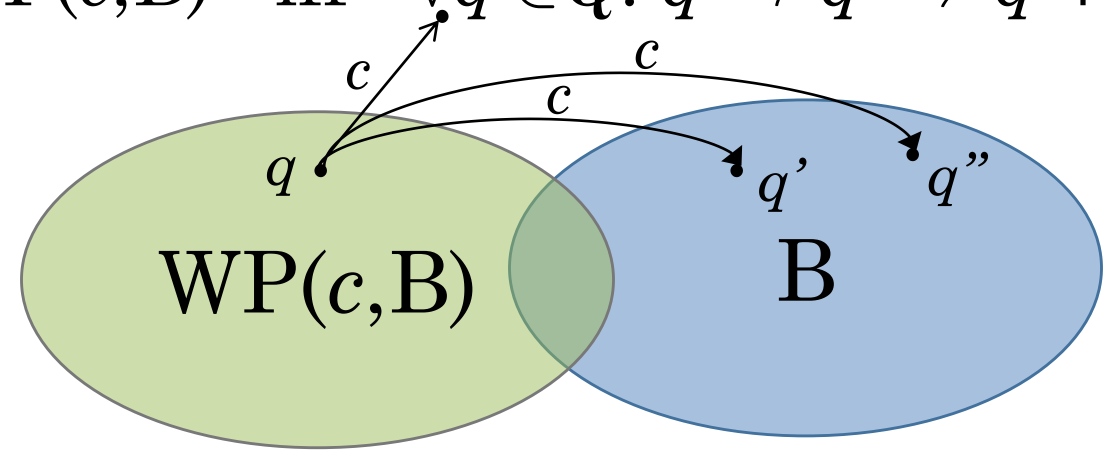
Verification Condition Generation

- Idea for VC generation: propagate the post-condition backwards through the program:
  - From {A} P {B}
  - generate $A \Rightarrow F(P, B)$
- 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
  | $c_1 \, ; \, c_2$
  | $c_1 \square c_2$

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) $\square$
  assume $\neg b$

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

Example: VC Generation

$\{n \geq 0\}$

$p := 0;$

$x := 0;$

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

while $x < n$ 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 $H \equiv \{A\}\ P\ \{B\}$
- Compute c_H = assume A; GC(P); assert B
- Compute $VC_H = WP(c_H, \text{true})$
- Infer $\vdash VC_H$ 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 $p = x * m \land x \leq n$;
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)
$\square$ 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 $p = x * m \wedge x \leq n$;

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

$\land$ 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 $\wedge$ 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; assume false, $p = n * m$)

$\wedge$ 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 $p = x * m \wedge x \leq n$, WP (assume false, $p = n * m$)

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

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

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

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

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

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

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

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 $p = p_1 + m$,

$p = x * m \land x \leq n$)

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

Example: VC Generation

- Computing the weakest precondition

WP (assume n ≥ 0;

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

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

assert p = x * m ∧ x ≤ n;

havoc x; havoc p; assume p = x * m ∧ x ≤ n,

WP (assume x < n;

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

assume $p_1 = p$; havoc p,

$p = p_1 + m$ ⇒ p = x * m ∧ x ≤ n)

∧ x ≥ 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 $x = x_1 + 1,$

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

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

Example: VC Generation

- Computing the weakest precondition

$$
\begin{aligned}
&\mathrm{WP}\ (\text{assume } n \geq 0; \\
&\quad \text{assume } p_0 = p;\ \text{havoc } p;\ \text{assume } p = 0; \\
&\quad \text{assume } x_0 = x;\ \text{havoc } x;\ \text{assume } x = 0; \\
&\quad \text{assert } p = x * m \wedge x \leq n; \\
&\quad \text{havoc } x;\ \text{havoc } p;\ \text{assume } p = x * m \wedge x \leq n, \\
&\quad \mathrm{WP}\ (\text{assume } x < n;\ \text{assume } x_1 = x;\ \text{havoc } x, \\
&\qquad x = x_1 + 1 \wedge\ p_1 = p \wedge\ pa_1 = p_1 + m \\
&\qquad \Rightarrow pa_1 = x * m \wedge x \leq n) \\
&\quad \wedge x\ \geq n \Rightarrow p = n * m)
\end{aligned}
$$

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,

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

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

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

Example: VC Generation

- Computing the weakest precondition

WP (assume n ≥ 0;

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

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

assert p = x * m ∧ x ≤ n;

havoc x; havoc p; assume p = x * m ∧ x ≤ n,

WP (assume x < n,

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

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

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

Example: VC Generation

- Computing the weakest precondition

$$
\begin{aligned}
&\mathrm{WP}\ (\text{assume } n \geq 0;\\
&\qquad \text{assume } p_0 = p;\ \text{havoc } p;\ \text{assume } p = 0;\\
&\qquad \text{assume } x_0 = x;\ \text{havoc } x;\ \text{assume } x = 0;\\
&\qquad \text{assert } p = x * m \wedge x \leq n;\\
&\qquad \text{havoc } x;\ \text{havoc } p;\ \text{assume } p = x * m \wedge x \leq n,\\
&\qquad (x < n \wedge\ x_1 = x \wedge xa_1 = x_1 + 1 \wedge\ p_1 = p \wedge\ pa_1 = p_1 + m\\
&\qquad\qquad \Rightarrow pa_1 = xa_1 * m \wedge xa_1 \leq n)\\
&\qquad \wedge x\ \geq n \Rightarrow p = n * m)
\end{aligned}
$$

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

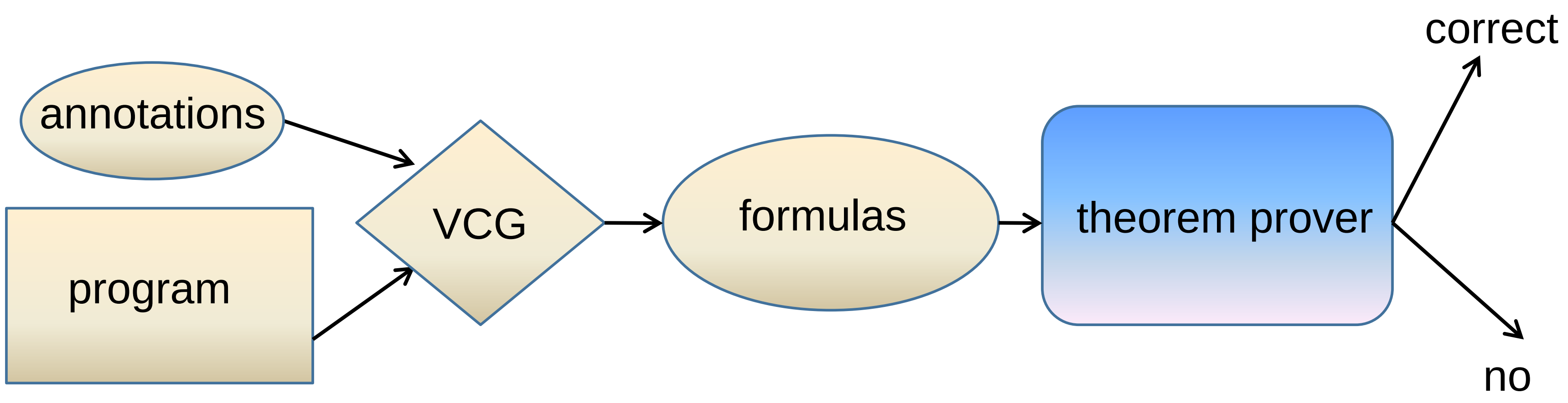
$$\mathrm{n} \geq 0 \Rightarrow 0 = 0 * m \wedge 0 \leq n$$

$$xa_2 \leq n \wedge \mathrm{xa}_2 \geq \mathrm{n} \Rightarrow xa_2 * m = n * m$$

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

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

Software Verification



SMT Solvers

- Used as a core engine in many tools in
  - Program analysis
  - Software engineering
  - Program model checking
  - Hardware verification, …
- Combine propositional satisfiability search techniques with specialized theory solvers
  - Linear arithmetic
  - Bit vectors
  - Uninterpreted functions with equality

```
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))
)
```

Theory of Arrays T_A

- $\Sigma_A = \{read, write, =\}$

- $read\ (a, i)$ is a binary function:
  - reads an array a at the index i

- $write\ (a, i, v)$ is a ternary function:
  - writes a value v to the index i of array a

Axioms of T_A

1. $\forall a\,,\, i, j\,.\; i = j \rightarrow read\,(a, i) = read\,(a, j)$
   (array congruence)
2. $\forall a\,,\, v, i, j.\; i = j \rightarrow read\,(write\,(a, i, v), j) = v$
   (read – write 1)
3. $\forall a\,,\, v, i, j.\; i \neq j \rightarrow read\,(write\,(a, i, v), j) = read\,(a, j)$
   (read – write 2)

How to deal with arrays?

- Very easily: use the following observation:

  a[i] := v is actually a := write(a, i, v)

- Everything else is the same
- SMT solvers supports arrays

Dealing with Arrays - An Example

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

$$\begin{aligned}
\text{WP}(\text{GC}, F) &= \text{WP}(\text{assume } tmp = a;\ \text{havoc } a;\ \text{assume } (a = \text{write}(\text{tmp}, i, v)), F) \\
&= \text{WP}(\text{assume } tmp = a;\ \text{havoc } a;\ a = \text{write}(\text{tmp}, i, v) \Rightarrow F) \\
&= \text{WP}(\text{assume } tmp = a;\ af = \text{write}(\text{tmp}, i, v) \Rightarrow F[af/a]) \\
&= tmp = a \ \Rightarrow af = \text{write}(\text{tmp}, i, v) \Rightarrow F[af/a] \\
&= tmp = a \ \wedge af = \text{write}(\text{tmp}, i, v) \Rightarrow F[af/a] \\
&= af = \text{write}(a, i, v) \Rightarrow F[af/a]
\end{aligned}$$

Separation Logic

Reasoning about Pointers

What is Separation Logic?

- Extension of Hoare logic
  - low-level imperative programs
  - shared mutable data structures

Problems with Aliasing

```
#include <stdio.h>

int main()
{
    int arr[2] = { 1, 2 };
    int i=10;

    /* alias i to arr[2]. */
    arr[2] = 20;

    printf("element 0: %d \t", arr[0]); // outputs 1
    printf("element 1: %d \t", arr[1]); // outputs 2
    printf("element 2: %d \t", arr[2]); // outputs 20
    printf("i: %d \t\t", i); // will also output 20, not 10, because of aliasing
    /* arr size is still 2. */
    printf("arr size: %d \n", (sizeof(arr) / sizeof(int)));
}
```

Motivating Example

```
assume( *x == 3 ∧ x != y ∧ x != z )

assume( y != z )

*y = 4;
*z = 5;
assert( *y != *z )

assert( *x == 3 )
```

Framing Problem

$$\frac{\{\texttt{y != z}\}\ C\ \{\texttt{*y != *z}\}}{\{\texttt{*x == 3} \land \texttt{y != z}\}\ C\ \{\texttt{*y != *z} \land \texttt{*x == 3}\}}$$

- What are the conditions on aliasing btwn x, y, z ?

Framing Problem

$$\frac{\{\,P\,\}\quad C\quad \{Q\}}{\{\ R \wedge P\ \}\ C\ \ \{\,Q \wedge R\,\}}$$

- What are the conditions on C and R?
  - in presence of aliasing and heap
    - Implicit Dynamic Frames

- Separation logic introduces new connective $*$

$$\frac{\{\,P\,\}\quad C\quad \{Q\}}{\{\ R * P\ \}\ \ C\ \ \{\,Q * R\,\}}$$

Permission-based Logics

- Separation Logic
  - O'Hearn, Reynolds, Yang 2001
  - Reynolds 2002
  - …
- Implicit Dynamic Frames
  - Smans, Jacobs, Piessens 2008
  - Parkinson, Summers 2011
- Linear maps
  - Lahiri, Qadeer, Walker 2011
- …

Tools using Permission-based Logics

- CompCert (Inria)
- L4.Verified (NICTA)
- Bedrock (MIT)
- …

interactive deductive verifiers

- Smallfoot (UCL, Imperial)
- Chalice (Microsoft Research)
- VeriFast (KU Leuven)
- HIP (Singapore)
- Viper (ETH)
- GRASShopper (NYU, Yale, MIT)
- …

automated deductive verifiers

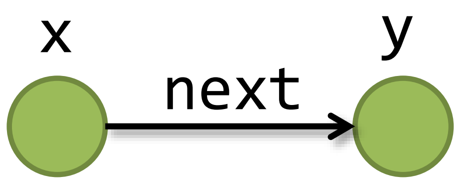
- Space Invader (UCL, Imperial)
- SLAyer (Microsoft Research)
- Infer (Facebook)
- Xisa (Boulder, Paris, Berkeley)
- …

static program analysis tools

A Simple Permission-based Logic

- Pure assertions

```
x.next == y
```



Stack

x	10
y	42
…	

Heap

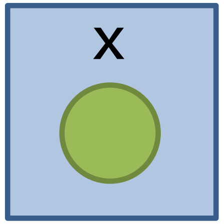
10	42
…	
42	?

```
struct Node {
  var next: Node;
}
```

A Simple Permission-based Logic

- Permission predicates

$$\mathbf{acc}(x)$$

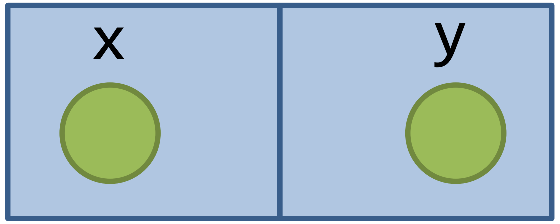


Expresses permission to access (i.e. read/write/deallocate) heap location x.
Assertions describe the program state **and** a set of locations that are allowed to be accessed.

A Simple Permission-based Logic

- Separating conjunction

acc(x) * **acc**(y)

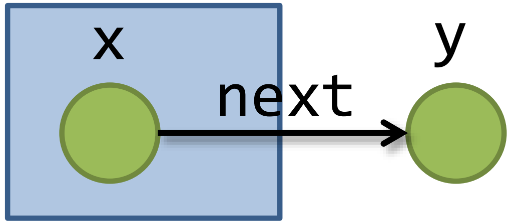


Yields union of permission sets of subformulas.
Permission sets of subformulas must be disjoint.

A Simple Permission-based Logic

- Separating conjunction

```
acc(x) * x.next == y
```

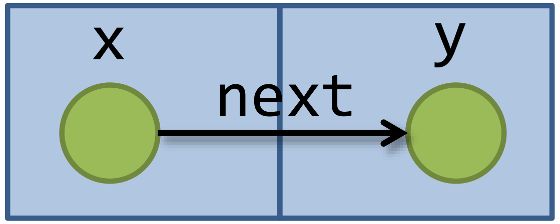


Pure assertions yield no permissions.

A Simple Permission-based Logic

- Separating conjunction

```
acc(x) * acc(y) * x.next == y
```



A Simple Permission-based Logic

- Separating conjunction

```
acc(x) * acc(x) * x.next == y
```

?

unsatisfiable

A Simple Permission-based Logic

- Classical conjunction

$$\texttt{\textbf{acc}(x)} \wedge \texttt{x.next == y}$$

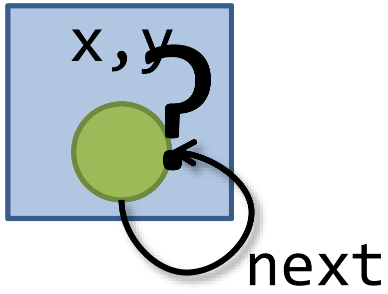
?

unsatisfiable

A Simple Permission-based Logic

- Classical conjunction

$$\texttt{\textbf{acc}(x)} \wedge \texttt{\textbf{acc}(y)} * \texttt{x.next == y}$$



Convention: ∧ has higher precedence than *

Syntactic Short-hands

- Empty heap:

  $\texttt{emp} \equiv (\texttt{x == x})$

- Points-to predicates:

  $\texttt{x.next} \mapsto \texttt{y} \equiv \textbf{acc}(\texttt{x}) * \texttt{x.next == y}$