Hoare Logic

Zhaoguo Wang

Adapted From:

ETH Zurich program verification, Lecture 6, 7
(https://www.pm.inf.ethz.ch/education/courses/program-verification.html)
<<Forward with Hoare>>
<<Weakest-Precondition of Unstructured Programs>>
Foundations of Programming Languages(https://cs.nju.edu.cn/xyfeng/teaching/FOPL, Hoare Logic)
<<Background reading on Hoare Logic>>, Mike Gordon
Cornell CS412/CS413, Introduction to Compilers, Lecture 24: Control Flow Graphs

Hoare Logic（霍尔逻辑）

1.1 Propositional Logic

Step 0. Proof.

Step 1. Convert program into mathematical formula.

Step 2. Ask the computer to solve the formula.

1.2 First Order Logic

Step 1. Convert it into first logic formula.

Step 2. Ask the computer to solve the formula.

1.3 Interactive Proof

Step 1. Axiom system

Step 2. Ask the computer to check the invariants

A Quick Recap – Axiom System

- Axiom System
- Hoare Triples

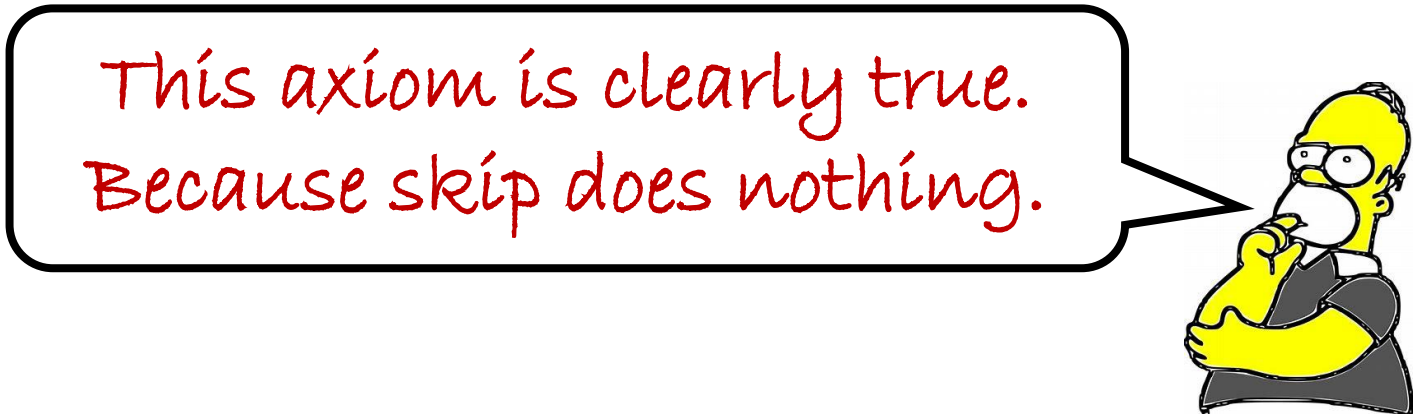
Outline – This Lecture

- Axioms and Inference Rules
- Interactive Theorem Prover
- Predicate Transformer
- Weakest Precondition
- Automated Verifier

Problem:
How to prove program correctness?

Axioms and Inference Rules

$$\{P\}skip\{P\}$$



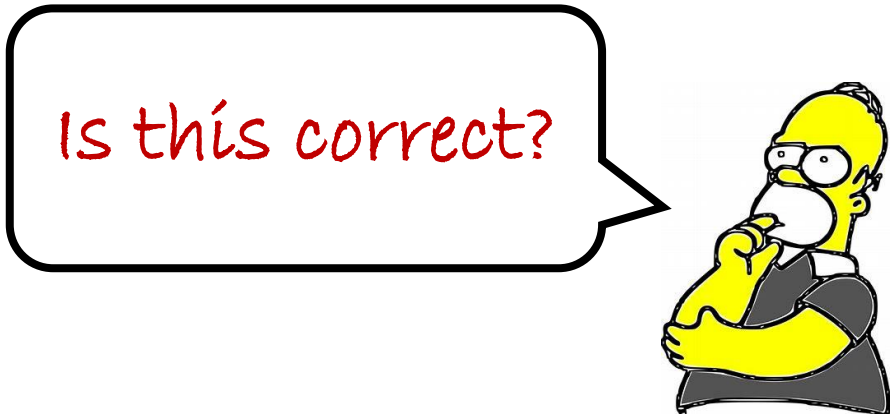
Axioms and Inference Rules

$$\{?\}a := e\{?\}$$

Axioms and Inference Rules

$$\{P\}a := e\{P[a/e]\}$$

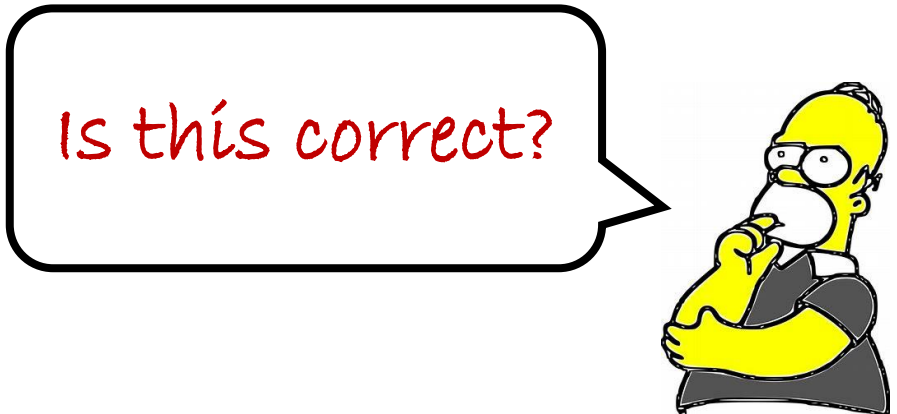
$$\{a = 0\}a := 1\{a = 0\}$$ ☒



Axioms and Inference Rules

$$\{P\}a := e\{P \wedge a = e\}$$

$$\{a = 5\}a := a + 1\{a = 5 \wedge a = a + 1\}$$ ☒



Axioms and Inference Rules

v is a fresh variable.

$$\{P\}a := e\{(\exists v)(a = e[v/a] \wedge P[v/a])\}$$

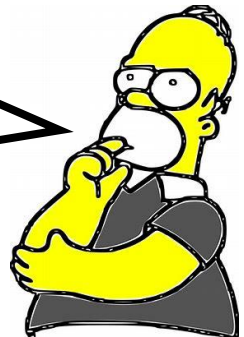
$$\{a = 1\}a := a + 1\{(\exists v)(a = (a + 1)[v/a] \wedge (a = 1)[v/a])\}$$

$$\{a = 1\}a := a + 1\{(\exists v)(a = v + 1 \wedge v = 1)\}$$

simplification

$$\{a = 1\}a := a + 1\{a = 2\}$$

This is correct and a forward assignment axiom.



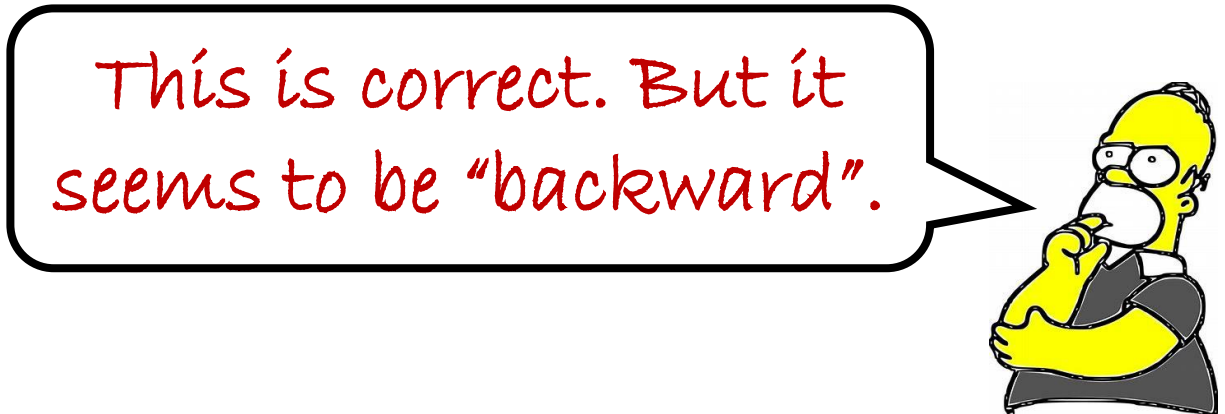
Axioms and Inference Rules

$$\{P[e/a]\}a := e\{P\}$$

$$\{b + 1 = 42\}a := b + 1\{a = 42\}$$

$$\{42 = 42\}a := 42\{a = 42\}$$

$$\{a - b > 3\}a := a - b\{a > 3\}$$



Axioms and Inference Rules

The abbreviation is AS-FW.

$$\{P\}a := e\{(\exists v)(a = e[v/a] \wedge P[v/a])\}$$

- Forward
- Quantifier
- Intuitive

The abbreviation is AS.

$$\{P[e/a]\}a := e\{P\}$$

- Backward
- Quantifier-free
- Not intuitive

It is more widely used because it's quantifier-free.

Axioms and Inference Rules

$$\{a = 1\}a := a + 1\{(\exists v)(a = (a + 1)[v/a] \wedge (a = 1)[v/a])\}$$

$$\{a = 1\}a := a + 1\{(\exists v)(a = v + 1 \wedge v = 1)\}$$

$$\{a = 1\}a := a + 1\{a = 2\}$$

The simplification is correct.

But Hoare logic is an axiom system. We need some rules to do this simplification.

Axioms and Inference Rules

strong $P \to Q$ weak

$$\frac{\{P\}C\{Q\} \quad Q \to R}{\{P\}C\{R\}}$$

The line means inference.

weakening consequent (WC)

$$\frac{\{a = 1\}a := a + 1\{(\exists v)(a = (a + 1)[v/a] \wedge (a = 1)[v/a])\} \quad (\exists v)(a = (a + 1)[v/a] \wedge (a = 1)[v/a]) \to a = 2}{\{a = 1\}a := a + 1\{a = 2\}}$$

Axioms and Inference Rules

strong $P \to Q$ weak

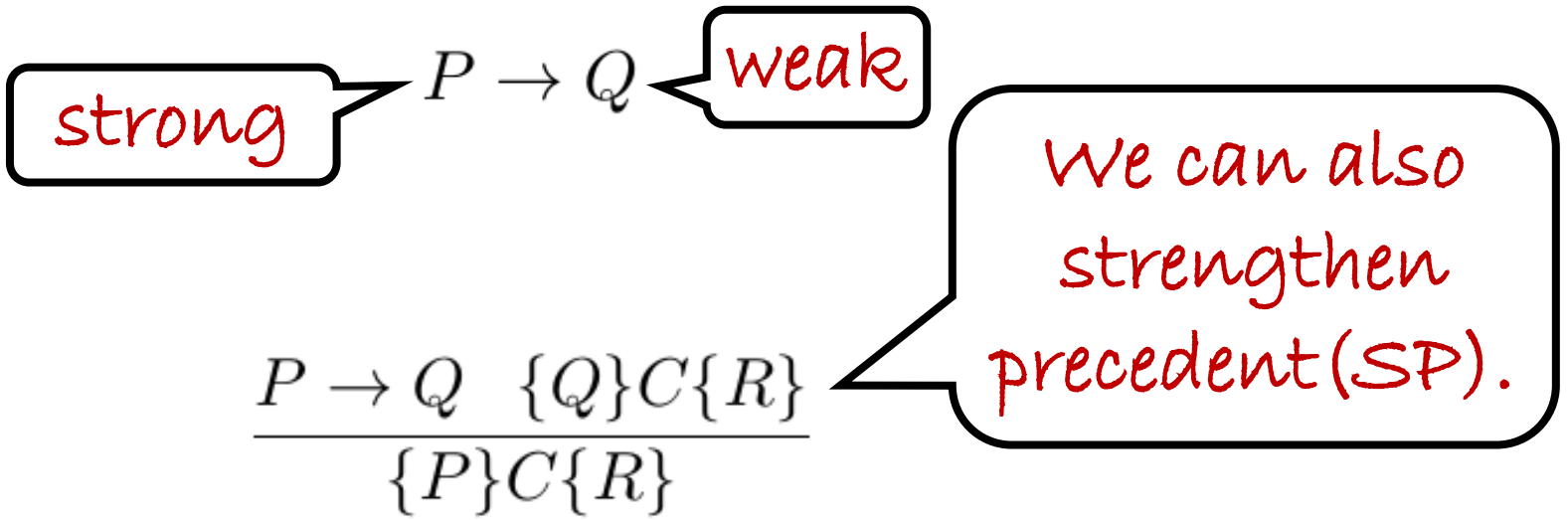
$$\frac{P \to Q \quad \{Q\}C\{R\}}{\{P\}C\{R\}}$$

We can also strengthen precedent (SP).

$\{a = n\}a := a + 1\{a = n + 1\}$

$Proof.$

Axioms and Inference Rules

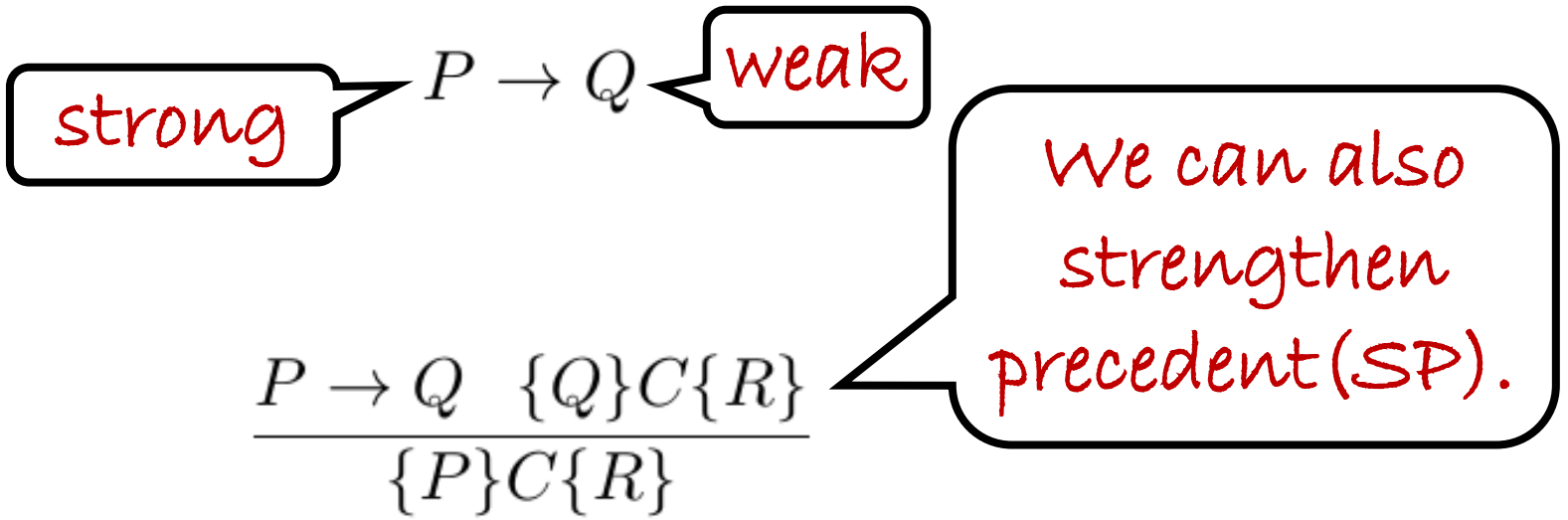


$\{a = n\}a := a + 1\{a = n + 1\}$

Proof.

$(1)a = n \to a + 1 = n + 1$ *predicate logic*

Axioms and Inference Rules



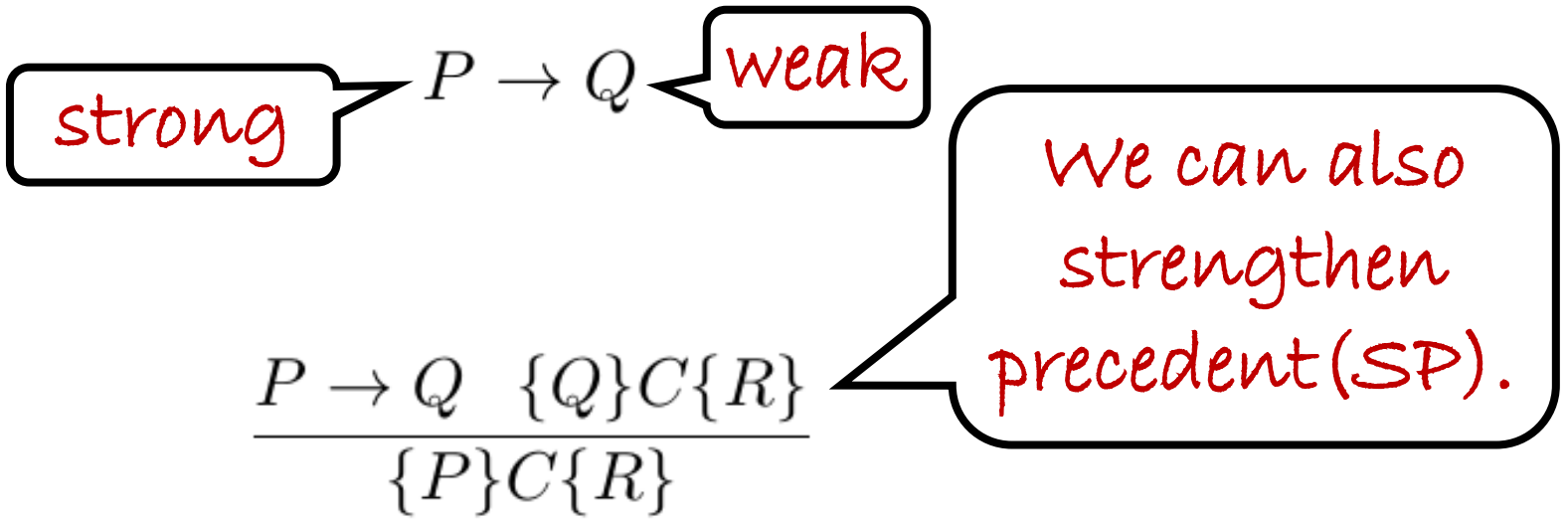
$\{a = n\}a := a + 1\{a = n + 1\}$

Proof.

(1) $a = n \to a + 1 = n + 1$ *predicate logic*

(2) $\{a + 1 = n + 1\}a := a + 1\{a = n + 1\}$ *AS*

Axioms and Inference Rules



$\{a = n\}a := a + 1\{a = n + 1\}$

Proof.

$(1) a = n \to a + 1 = n + 1$	*predicate logic*
$(2)\{a + 1 = n + 1\}a := a + 1\{a = n + 1\}$	AS
$(3)\{a = n\}a := a + 1\{a = n + 1\}$	$SP\ (1)(2)$

Axioms and Inference Rules

Program structure

1. Sequence : s1; s2
2. Branch : if(b1) { s1 } then { s2 }
3. Loop structure : while(b1) { s }

We will introduce inference rules for these structures.

This is what we want the most.

Axioms and Inference Rules

$$\frac{\{P\}C1\{R\} \;\; \{R\}C2\{Q\}}{\{P\}C1;C2\{Q\}}$$

The sequential composition rule(SC)

$\{b > 3\}a := 2 * b; a := a - b\{a \geq 4\}$

Proof.

Axioms and Inference Rules

$$\frac{\{P\}C1\{R\} \;\; \{R\}C2\{Q\}}{\{P\}C1;C2\{Q\}}$$

The sequential composition rule(SC)

$\{b > 3\}a := 2 * b; a := a - b\{a \geq 4\}$

Proof.

$(1)\{a - b \geq 4\}a := a - b\{a \geq 4\}$ AS

Axioms and Inference Rules

$$\frac{\{P\}C1\{R\} \;\; \{R\}C2\{Q\}}{\{P\}C1;C2\{Q\}}$$

The sequential composition rule(SC)

$\{b > 3\}a := 2 * b; a := a - b\{a \geq 4\}$

Proof.

(1) $\{a - b \geq 4\}a := a - b\{a \geq 4\}$ AS

(2) $\{2 * b - b \geq 4\}a := 2 * b\{a - b \geq 4\}$ AS

Axioms and Inference Rules

$$\frac{\{P\}C1\{R\} \;\; \{R\}C2\{Q\}}{\{P\}C1;C2\{Q\}}$$

The sequential composition rule(SC)

$\{b > 3\}a := 2 * b; a := a - b\{a \geq 4\}$

Proof.

$(1)\{a - b \geq 4\}a := a - b\{a \geq 4\}$	AS
$(2)\{2 * b - b \geq 4\}a := 2 * b\{a - b \geq 4\}$	AS
$(3)b > 3 \rightarrow 2 * b - b \geq 4$	$predicate\ logic$

Axioms and Inference Rules

$$\frac{\{P\}C1\{R\} \;\; \{R\}C2\{Q\}}{\{P\}C1;C2\{Q\}}$$

The sequential composition rule(SC)

$\{b > 3\}a := 2 * b; a := a - b\{a \geq 4\}$

Proof.

$(1)\{a - b \geq 4\}a := a - b\{a \geq 4\}$	AS
$(2)\{2 * b - b \geq 4\}a := 2 * b\{a - b \geq 4\}$	AS
$(3)b > 3 \rightarrow 2 * b - b \geq 4$	$predicate\ logic$
$(4)\{b > 3\}a := 2 * b\{a - b \geq 4\}$	$SP(2)(3)$

Axioms and Inference Rules

$$\frac{\{P\}C1\{R\} \;\; \{R\}C2\{Q\}}{\{P\}C1;C2\{Q\}}$$

the sequential composition rule(SC)

$\{b > 3\}a := 2 * b; a := a - b\{a \geq 4\}$

Proof.

$(1)\{a - b \geq 4\}a := a - b\{a \geq 4\}$	AS
$(2)\{2 * b - b \geq 4\}a := 2 * b\{a - b \geq 4\}$	AS
$(3)b > 3 \rightarrow 2 * b - b \geq 4$	*predicate logic*
$(4)\{b > 3\}a := 2 * b\{a - b \geq 4\}$	$SP(2)(3)$
$(5)\{b > 3\}a := 2 * b; a := a - b\{a \geq 4\}$	$SC(1)(4)$

It's clearly proved backwards.

Axioms and Inference Rules

Convert b to a proposition.

Convert b to a proposition.

$$\frac{\{P \wedge istrue(b1)\}C1\{Q\} \quad \{P \wedge isfalse(b1)\}C2\{Q\}}{\{P\}if(b1)\ then\ C1\ else\ C2\ \{Q\}}$$

The conditional rule (CD)

Axioms and Inference Rules

$$\{T\}\ if(a < b)\ then\ c := b\ else\ c := a\{c = max(a, b)\}$$

$Proof.$

Axioms and Inference Rules

$$\{T\}\ if(a < b)\ then\ c := b\ else\ c := a\{c = max(a, b)\}$$

$Proof.$

$$(1)\{b = max(a, b)\}c := b\{c = max(a, b)\} \qquad AS$$

Axioms and Inference Rules

$\{T\}\ if(a < b)\ then\ c := b\ else\ c := a\{c = max(a, b)\}$

$Proof.$

$(1)\{b = max(a, b)\}c := b\{c = max(a, b)\}$ AS

$(2)\{a = max(a, b)\}c := a\{c = max(a, b)\}$ AS

Axioms and Inference Rules

$\{T\}\ if(a < b)\ then\ c := b\ else\ c := a\{c = max(a, b)\}$

$Proof.$

$(1)\{b = max(a, b)\}c := b\{c = max(a, b)\}$ AS

$(2)\{a = max(a, b)\}c := a\{c = max(a, b)\}$ AS

$(3)T \wedge a < b \rightarrow b = max(a, b)$ $predicate\ logic$

Axioms and Inference Rules

$\{T\}\ if(a < b)\ then\ c := b\ else\ c := a\{c = max(a, b)\}$

$Proof.$

$(1)\{b = max(a, b)\}c := b\{c = max(a, b)\}$ AS

$(2)\{a = max(a, b)\}c := a\{c = max(a, b)\}$ AS

$(3)T \wedge a < b \rightarrow b = max(a, b)$ $predicate\ logic$

$(4)T \wedge \neg(a < b) \rightarrow a = max(a, b)$ $predicate\ logic$

Axioms and Inference Rules

$\{T\}\ if(a < b)\ then\ c := b\ else\ c := a\{c = max(a, b)\}$

$Proof.$

$(1)\{b = max(a, b)\}c := b\{c = max(a, b)\}$ AS

$(2)\{a = max(a, b)\}c := a\{c = max(a, b)\}$ AS

$(3)T \wedge a < b \rightarrow b = max(a, b)$ $predicate\ logic$

$(4)T \wedge \neg(a < b) \rightarrow a = max(a, b)$ $predicate\ logic$

$(5)\{T \wedge a < b\}c := b\{c = max(a, b)\}$ $SP(1)(3)$

Axioms and Inference Rules

$\{T\}\ if(a < b)\ then\ c := b\ else\ c := a\{c = max(a, b)\}$

$Proof.$

$(1)\{b = max(a, b)\}c := b\{c = max(a, b)\}$ AS

$(2)\{a = max(a, b)\}c := a\{c = max(a, b)\}$ AS

$(3)T \wedge a < b \rightarrow b = max(a, b)$ $predicate\ logic$

$(4)T \wedge \neg(a < b) \rightarrow a = max(a, b)$ $predicate\ logic$

$(5)\{T \wedge a < b\}c := b\{c = max(a, b)\}$ $SP(1)(3)$

$(6)\{T \wedge \neg(a < b)\}c := a\{c = max(a, b)\}$ $SP(2)(4)$

Axioms and Inference Rules

$\{T\}\ if(a < b)\ then\ c := b\ else\ c := a\{c = max(a, b)\}$

$Proof.$

$(1)\{b = max(a, b)\}c := b\{c = max(a, b)\}$	AS
$(2)\{a = max(a, b)\}c := a\{c = max(a, b)\}$	AS
$(3)T \wedge a < b \rightarrow b = max(a, b)$	$predicate\ logic$
$(4)T \wedge \neg(a < b) \rightarrow a = max(a, b)$	$predicate\ logic$
$(5)\{T \wedge a < b\}c := b\{c = max(a, b)\}$	$SP(1)(3)$
$(6)\{T \wedge \neg(a < b)\}c := a\{c = max(a, b)\}$	$SP(2)(4)$
$(7)\{T\}\ if(a < b)\ then\ c := b\ else\ c := a\{c = max(a, b)\}$	$CD(5)(6)$

Axioms and Inference Rules

loop invariant

$$\frac{\{I \wedge istrue(b1)\}C\{I\}}{\{I\}\ while(b1)\ C\ \{I \wedge isfalse(b1)\}}$$

The while rule (WHP)

It says that

- if executing C once preserves the truth of I, then executing C any number of times also preserves the truth of I
- after a while loop has terminated, the test must be false

Axioms and Inference Rules

loop invariant

$$\frac{\{I \wedge istrue(b1)\}C\{I\}}{\{I\}\ while(b1)\ C\ \{I \wedge isfalse(b1)\}}$$

The while rule(WHP)

It can be proved by induction on number of loop times

- If the loop execute for 0 time, I of course holds.
- If I holds for n times loops, it holds for n+1 times because of the condition above the line.

Axioms and Inference Rules

$$\{a \leq 10\}\ while(a! = 10)\ a := a + 1\ \{a = 10\}$$

$$\frac{\{I \wedge istrue(b1)\}C\{I\}}{\{I\}\ while(b1)\ C\ \{I \wedge isfalse(b1)\}}$$

$Proof.$

Axioms and Inference Rules

$$\frac{\{I \wedge istrue(b1)\}C\{I\}}{\{I\}\ while(b1)\ C\ \{I \wedge isfalse(b1)\}}$$

$\{a \leq 10\}\ while(a! = 10)\ a := a + 1\ \{a = 10\}$

$Proof.$

$(1)\{a + 1 \leq 10\}a := a + 1\{a \leq 10\}$ AS

loop invariant

Axioms and Inference Rules

$$\frac{\{I \wedge istrue(b1)\}C\{I\}}{\{I\}\ while(b1)\ C\ \{I \wedge isfalse(b1)\}}$$

$\{a \leq 10\}\ while(a! = 10)\ a := a + 1\ \{a = 10\}$

$Proof.$

$(1)\{a + 1 \leq 10\}a := a + 1\{a \leq 10\}$ AS

$(2)a \leq 10 \wedge a \neq 10 \rightarrow a + 1 \leq 10$ $predicate\ logic$

Axioms and Inference Rules

$$\frac{\{I \wedge istrue(b1)\}C\{I\}}{\{I\}\ while(b1)\ C\ \{I \wedge isfalse(b1)\}}$$

$\{a \leq 10\}\ while(a! = 10)\ a := a + 1\ \{a = 10\}$

$Proof.$

$(1)\{a + 1 \leq 10\}a := a + 1\{a \leq 10\}$	AS
$(2)a \leq 10 \wedge a \neq 10 \rightarrow a + 1 \leq 10$	$predicate\ logic$
$(3)\{a \leq 10 \wedge a \neq 10\}a := a + 1\{a \leq 10\}$	$SP(1)(2)$

loop invariant always holds.

Axioms and Inference Rules

$$\frac{\{I \wedge istrue(b1)\}C\{I\}}{\{I\}\ while(b1)\ C\ \{I \wedge isfalse(b1)\}}$$

$\{a \leq 10\}\ while(a! = 10)\ a := a + 1\ \{a = 10\}$

$Proof.$

$(1)\{a + 1 \leq 10\}a := a + 1\{a \leq 10\}$	AS
$(2)a \leq 10 \wedge a \neq 10 \rightarrow a + 1 \leq 10$	$predicate\ logic$
$(3)\{a \leq 10 \wedge a \neq 10\}a := a + 1\{a \leq 10\}$	$SP(1)(2)$
$(4)\{a \leq 10\}\ while(a! = 10)\ a := a + 1\ \{a \leq 10 \wedge a = 10\}$	$WHP(3)$

Axioms and Inference Rules

$$\frac{\{I \wedge istrue(b1)\}C\{I\}}{\{I\}\ while(b1)\ C\ \{I \wedge isfalse(b1)\}}$$

$\{a \leq 10\}\ while(a! = 10)\ a := a + 1\ \{a = 10\}$

$Proof.$

$(1)\{a + 1 \leq 10\}a := a + 1\{a \leq 10\}$	AS
$(2)a \leq 10 \wedge a \neq 10 \rightarrow a + 1 \leq 10$	$predicate\ logic$
$(3)\{a \leq 10 \wedge a \neq 10\}a := a + 1\{a \leq 10\}$	$SP(1)(2)$
$(4)\{a \leq 10\}\ while(a! = 10)\ a := a + 1\ \{a \leq 10 \wedge a = 10\}$	$WHP(3)$
$(5)a \leq 10 \wedge a = 10 \rightarrow a = 10$	$predicate\ logic$
$(6)\{a \leq 10\}\ while(a! = 10)\ a := a + 1\ \{a = 10\}$	$WC(4)(5)$

Axioms and Inference Rules

$$\frac{\{I \wedge istrue(b1)\}C\{I\}}{\{I\}\ while(b1)\ C\ \{I \wedge isfalse(b1)\}}$$

$\{a \leq 10\}\ while(a! = 10)\ a := a + 1\ \{a = 10\}$

Proof.

$(1)\{a + 1 \leq 10\}a := a + 1\{a \leq 10\}$	AS
$(2)a \leq 10 \wedge a \neq 10 \rightarrow a + 1 \leq 10$	*predicate logic*
$(3)\{a \leq 10 \wedge a \neq 10\}a := a + 1\{a \leq 10\}$	$SP(1)(2)$
$(4)\{a \leq 10\}\ while(a! = 10)\ a := a + 1\ \{a \leq 10 \wedge a = 10\}$	$WHP(3)$
$(5)a \leq 10 \wedge a = 10 \rightarrow a = 10$	*predicate logic*
$(6)\{a \leq 10\}\ while(a! = 10)\ a := a + 1\ \{a = 10\}$	$WC(4)(5)$

Axioms and Inference Rules

There is no automatic algorithm to generate loop invariant.

You must find it by yourself. The invariant should say that:

- what has been done so far together with what remains to be done
- holds at each iteration of the loop
- and gives the desired result when the loop terminates

Exercise

```
void f(unsigned a)
{
    unsigned i = 0;
    {i = 0 ∧ a ≥ 0}
    while(i < a)
    {
        i = i + 1;
    }
    {i = a}
}
```

pre-condition

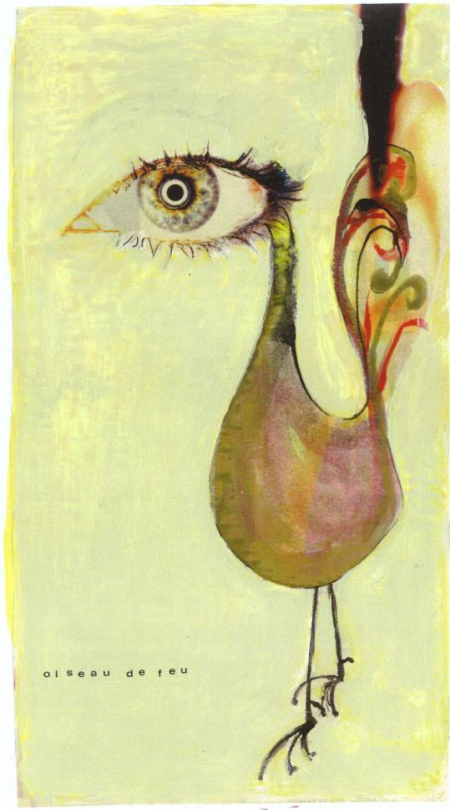
post-condition

$$I : i \geq 0 \wedge i \leq a \wedge a \geq 0$$

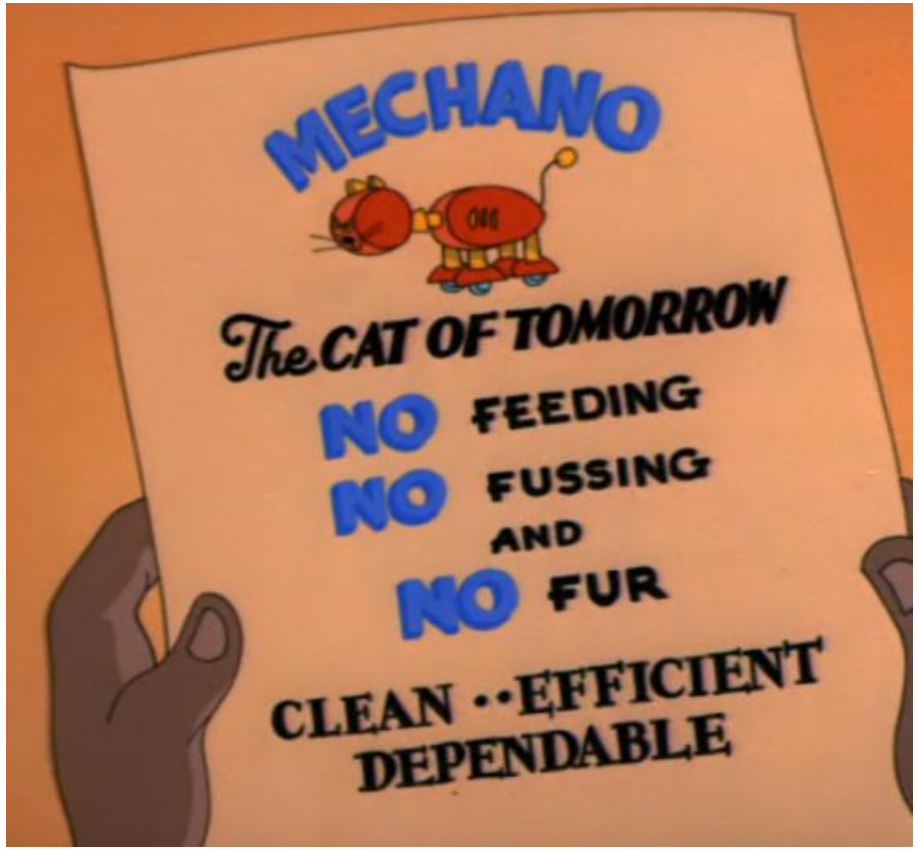
loop invariant?

Interactive Theorem Prover

- Coq is an interactive theorem prover
- We can use Coq to construct an axiom system, such as Hoare logic, and prove theorems
- Coq will help check whether the proofs are correct, avoiding errors in paper proof
- Some learning materials
  - https://6826.csail.mit.edu/2019/schedule.html
  - https://github.com/coq-community/vscoq
  - https://softwarefoundations.cis.upenn.edu/current/index.html
  - http://adam.chlipala.net/cpdt/
  - https://www.labri.fr/perso/casteran/CoqArt/

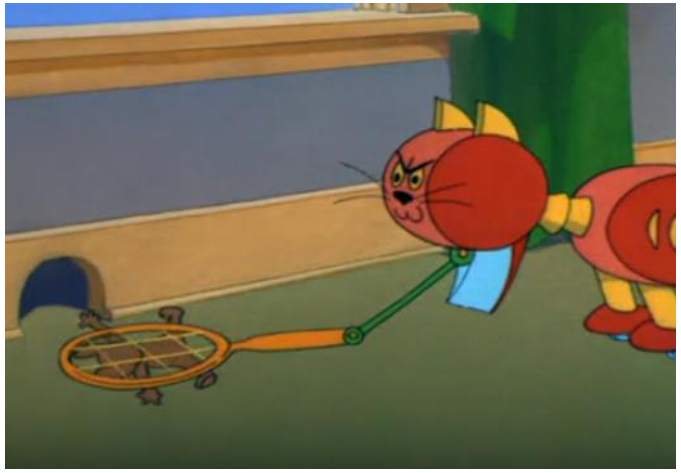
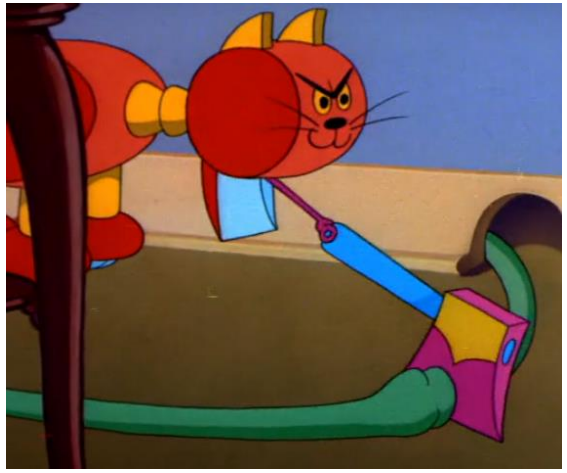
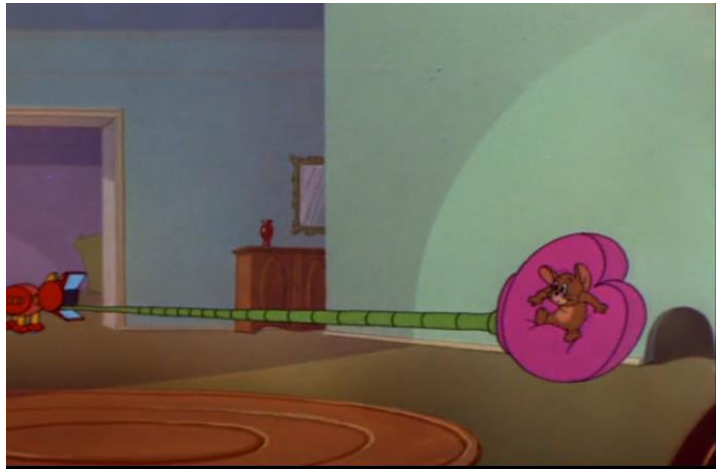


However,



We want an automated program verifier.

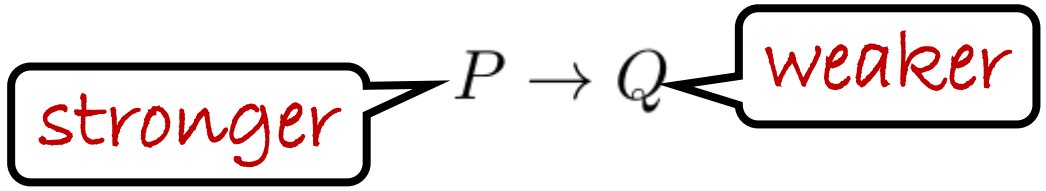
But cannot handle loops

What if we keep all invariants/conditions in the interactive proof?

Automatic Proof, Again???

Predicate Transformer



DEFINITION

Given a program C and postcondition Q, P is the weakest precondition means for all P', {P'} C {Q} iff P' → P. We use wp(C, Q) to denote P.

$$\{a = 2\}a := a + 1\{a = 3\}$$

$$\{a = 2 \wedge b = 4\}a := a + 1\{a = 3\}$$

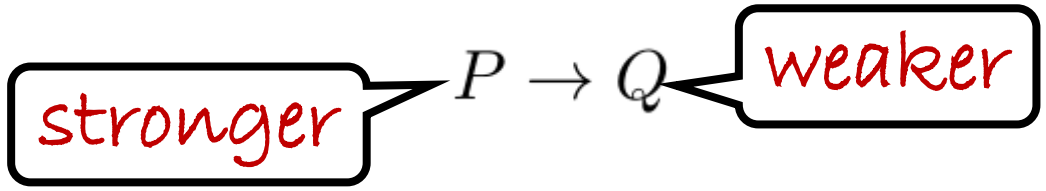
$$\{a = 2 \wedge a > 1\}a := a + 1\{a = 3\}$$

weakest precondition

$$a = 2 \wedge b = 4 \rightarrow a = 2$$

$$a = 2 \wedge a > 1 \rightarrow a = 2$$

Predicate Transformer



DEFINITION

Given a program C and precondition P, Q is the strongest postcondition means for all Q', {P} C {Q'} iff Q → Q'. We use sp(C, P) to denote Q.

strongest postcondition

$$\{a = 1\}a := 3\{a = 3\}$$

$$\{a = 1\}a := 3\{T\} \qquad a = 3 \rightarrow T$$

$$\{a = 1\}a := 3\{(\exists v)(a = v)\} \qquad a = 3 \rightarrow (\exists v)(a = v)$$

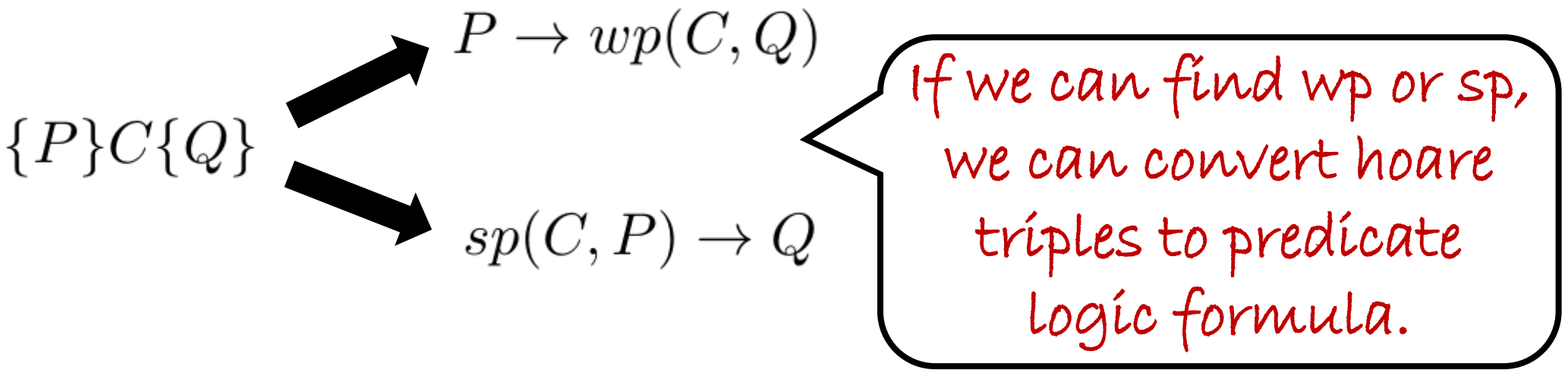
Predicate Transformer

We can consider the problem from the following perspective:

- Given a precondition and program, can we find the strongest postcondition?

- Or given a postcondition and program, can we find the weakest precondition?

- This is called predicate transformer semantics: how programs transform assertions



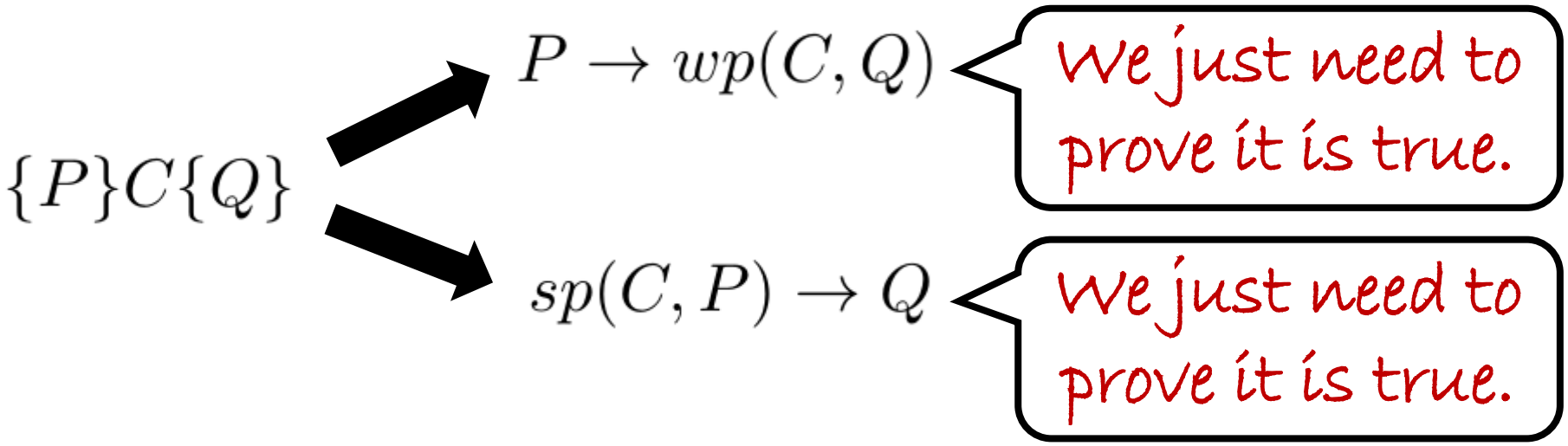
Predicate Transformer

We can consider the problem from the following perspective:

- Given a precondition and program, can we find the strongest postcondition?

- Or given a postcondition and program, can we find the weakest precondition?

- This is called predicate transformer semantics: how programs transform assertions



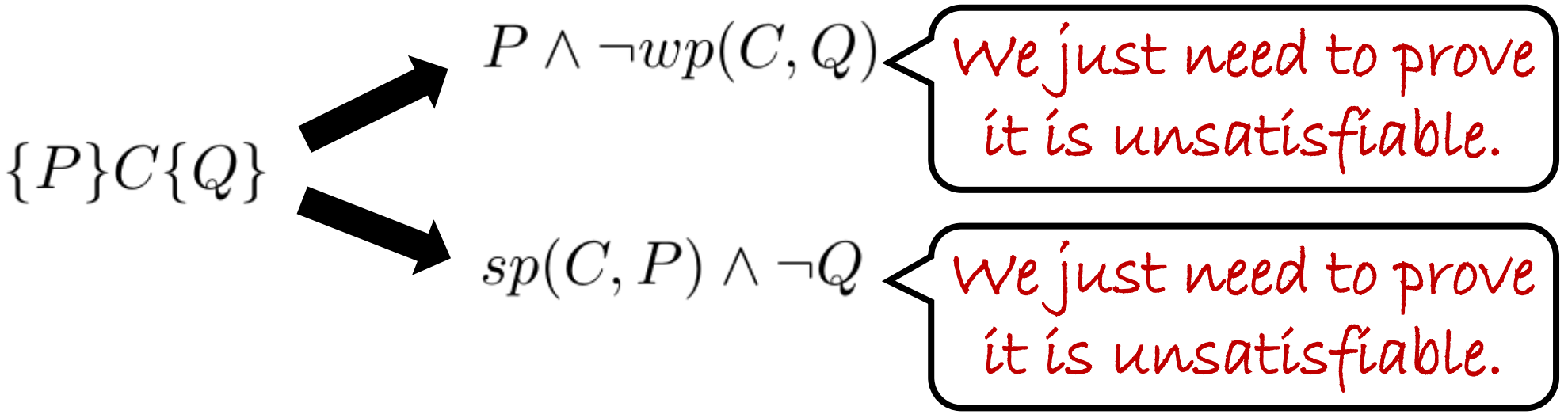
Predicate Transformer

We can consider the problem from the following perspective:

- Given a precondition and program, can we find the strongest postcondition?

- Or given a postcondition and program, can we find the weakest precondition?

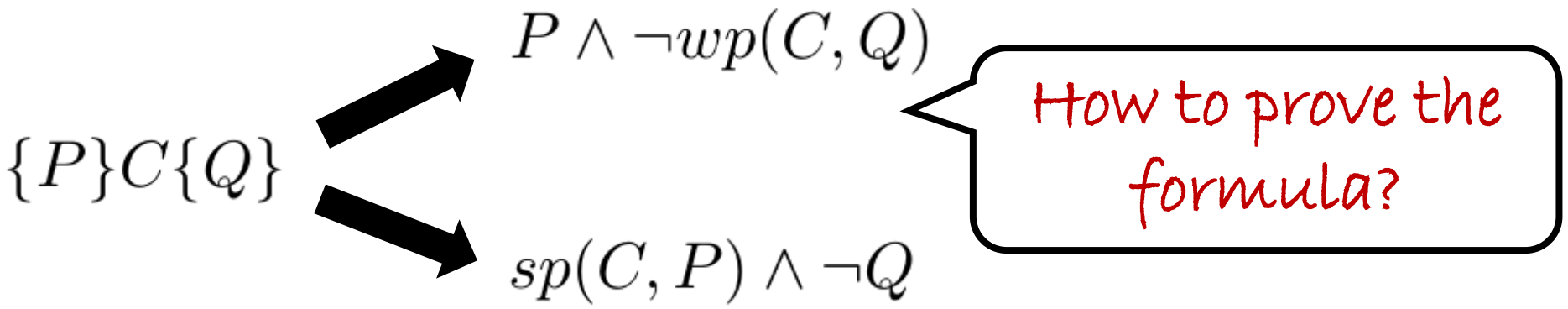
- This is called predicate transformer semantics: how programs transform assertions



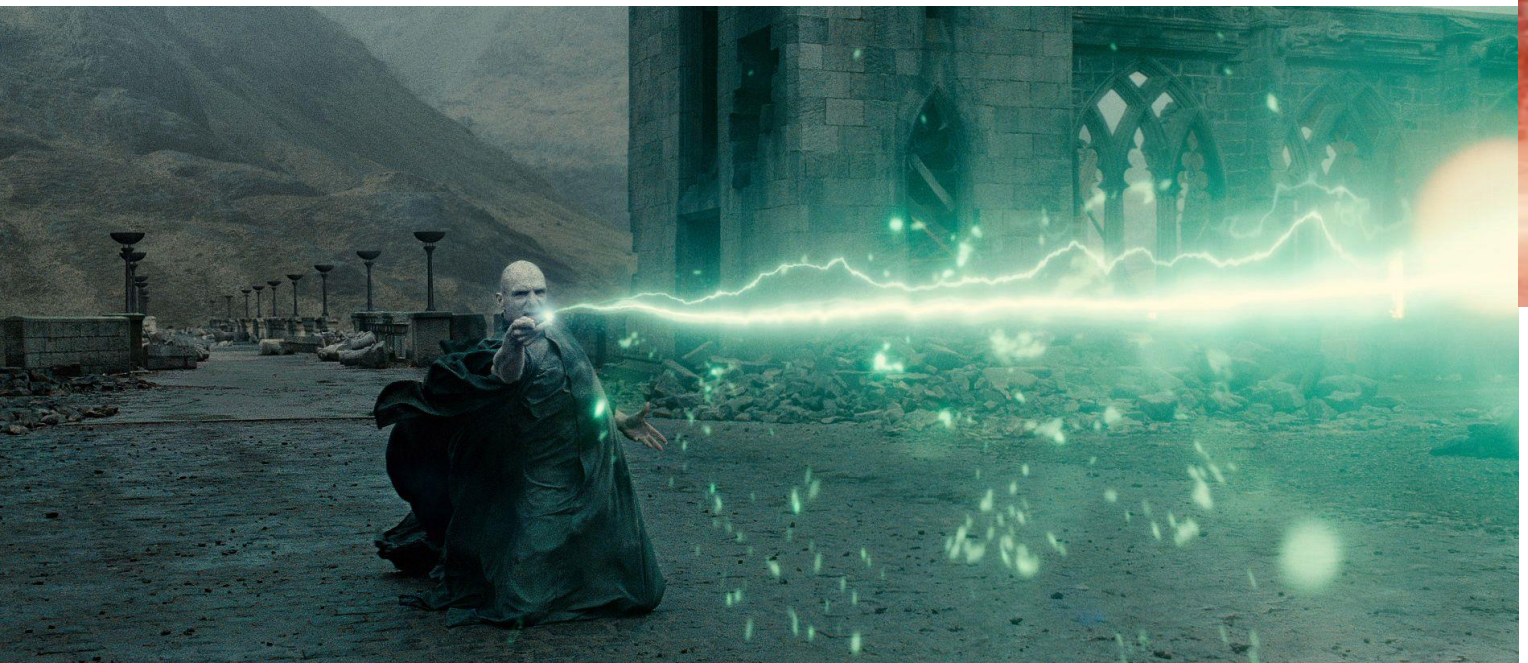
Basic Idea

To prove `{P}C{Q}`

- Find the `wp(C, Q)`, then prove `P` implies `wp(C, Q)`.
- Find the `sp(C, P)`, then prove `sp(C, P)` implies `Q`.



Working with SMT solvers is like magic



Question: How to automatically find the weakest precondition (strongest postcondition)?

By rules (Theorems)…

A Quick Recap

AS-FW gives the strongest postcondition.

$$\{P\}a := e\{(\exists v)(a = e[v/a] \wedge P[v/a])\}$$

- Forward
- Quantifier
- Intuitive

AS gives the weakest postcondition.

$$\{P[e/a]\}a := e\{P\}$$

- Backward
- Quantifier-free
- Not intuitive

It is more widely used because it's simpler.

A Quick Recap

AS-FW gives the strongest postcondition.

$$\{P\}a := e\{(\exists v)(a = e[v/a] \wedge P[v/a])\}$$

AS gives the weakest postcondition.

$$\{P[e/a]\}a := e\{P\}$$

- Forward
- Quantifier
- Intuitive

It is more widely used because it's simpler.

- Backward

Quantifier-free

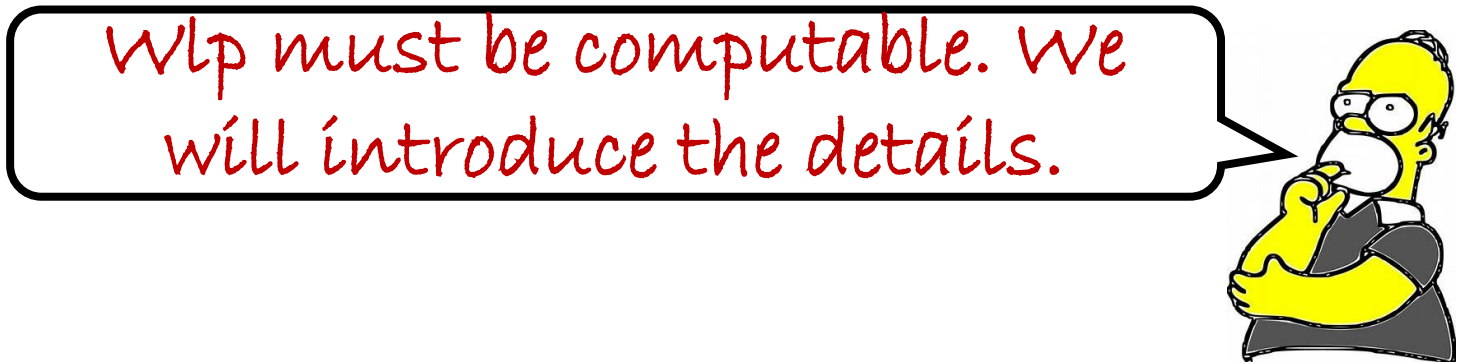
We will focus on wp.

tive

Weakest Precondition

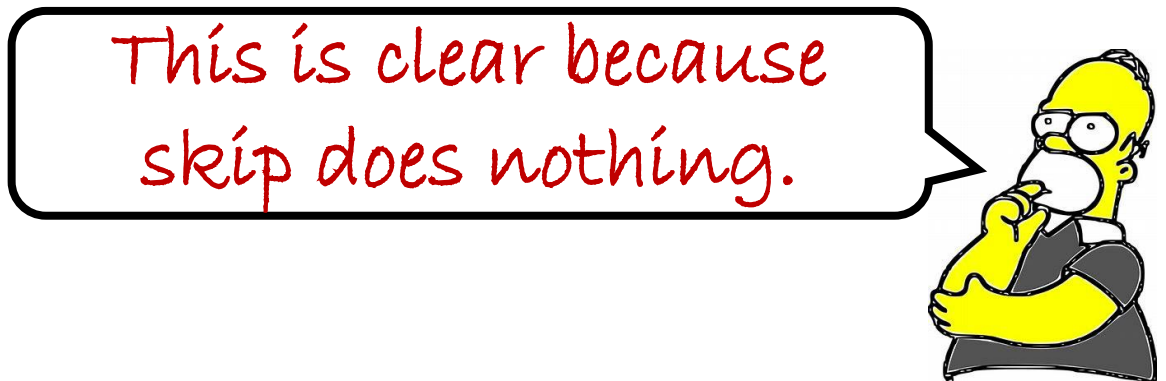
We use wlp(C, Q) instead of wp(C, Q) in following slides:

- C is the program, Q is the intended postcondition
- Name stands for weakest “liberal” precondition: it ignores termination
- For all C and Q, it is guaranteed that { wlp(C, Q) } C {Q} is true
- For all P, C and Q, {P} C {Q} iff $P \rightarrow wlp(C, Q)$



Weakest Precondition

$$wlp(skip, Q) = Q$$



Weakest Precondition

This is similar to AS in Hoare logic.

$$\{P[e/a]\}a := e\{P\}$$

$$wlp(a := e, P) = P[e/a]$$

$$wlp(a := b + 1, a = 42) = b + 1 = 42$$

$$wlp(a := a - b, a > 3) = \quad ?$$

Weakest Precondition

This is similar to AS in Hoare logic.

$$\{P[e/a]\}a := e\{P\}$$

$$wlp(a := e, P) = P[e/a]$$

$$wlp(a := b + 1, a = 42) = b + 1 = 42$$

$$wlp(a := a - b, a > 3) = a - b > 3$$

Weakest Precondition

$$wlp(C1; C2, Q) = wlp(C1, wlp(C2, Q))$$

$$wlp(a := a + 1; b := a, a = 3 \wedge b > 0)$$

Weakest Precondition

$$wlp(C1; C2, Q) = wlp(C1, wlp(C2, Q))$$

$$wlp(a := a + 1; b := a, a = 3 \wedge b > 0)$$

$$= wlp(a := a + 1, wlp(b := a, a = 3 \wedge b > 0))$$

$$= wlp(a := a + 1, a = 3 \wedge a > 0)$$

$$= a + 1 = 3 \wedge a + 1 > 0$$

Weakest Precondition

$$wlp(if(b1)\ then\ \{C1\}\ else\ \{C2\}, Q)$$

$$=(istrue(b1) \rightarrow wlp(C1, Q)) \wedge (isfalse(b1) \rightarrow wlp(C2, Q))$$

$$wlp(if(a == 0)\ then\ \{b := 1\}\ else\ \{b := 0\}, b = 1 \vee b = 0)$$

Weakest Precondition

$$
\begin{aligned}
&wlp(if(b1)\ then\ \{C1\}\ else\ \{C2\}, Q) \\
=&(istrue(b1) \rightarrow wlp(C1, Q)) \wedge (isfalse(b1) \rightarrow wlp(C2, Q))
\end{aligned}
$$

$$
\begin{aligned}
&wlp(if(a == 0)\ then\ \{b := 1\}\ else\ \{b := 0\}, b = 1 \vee b = 0) \\
=&(a = 0 \rightarrow wlp(b := 1, b = 1 \vee b = 0)) \wedge (a \neq 0 \rightarrow wlp(b := 0, b = 1 \vee b = 0)) \\
=&(a = 0 \rightarrow 1 = 1 \vee 1 = 0) \wedge (a \neq 0 \rightarrow 0 = 1 \vee 0 = 0)
\end{aligned}
$$

Weakest Precondition

$$wlp(while(b1)\{C\}, Q) =?$$

:-) 呵呵

Question: How does an automated verifier implement these?

Basic Design of An Automated Verifier

Input

- Invariants (e.g., Pre-condition) provided by user
- A program to be proved

Workflow

- Convert the program into Control Flow Graph (CFG).
- Insert the invariants into the CFG.
- Translate CFG into Hoare Triples
  - Find the weakest precondition
  - Prove the precondition imply the weakest precondition (Predicate Logic)

Basic Design of An Automated Verifier

Input

- Invariants (e.g., Pre-condition) provided by user
- A program to be proved

Workflow

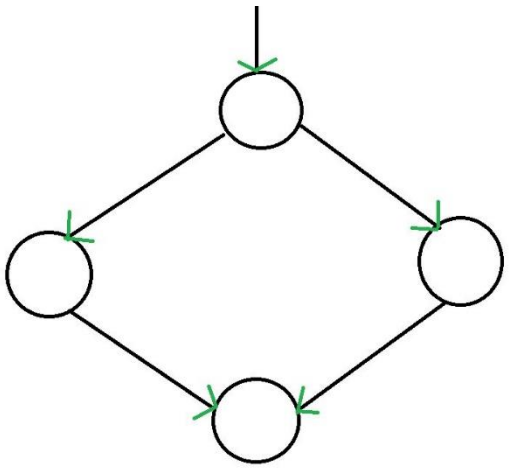
- Convert the program into Control Flow Graph (CFG).
- Insert the invariants into the CFG.
- Translate CFG into Hoare Triples
  - Find the weakest precondition
  - Prove the precondition imply the weakest precondition (Predicate Logic)

Automated Verifier

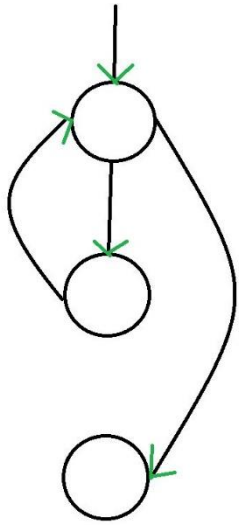
In automated verifiers, programs are represented by the control-flow graph.

DEFINITION

A control-flow graph (CFG) is a representation, using graph notation, of all paths that might be traversed through a program during its execution.



If-then-else

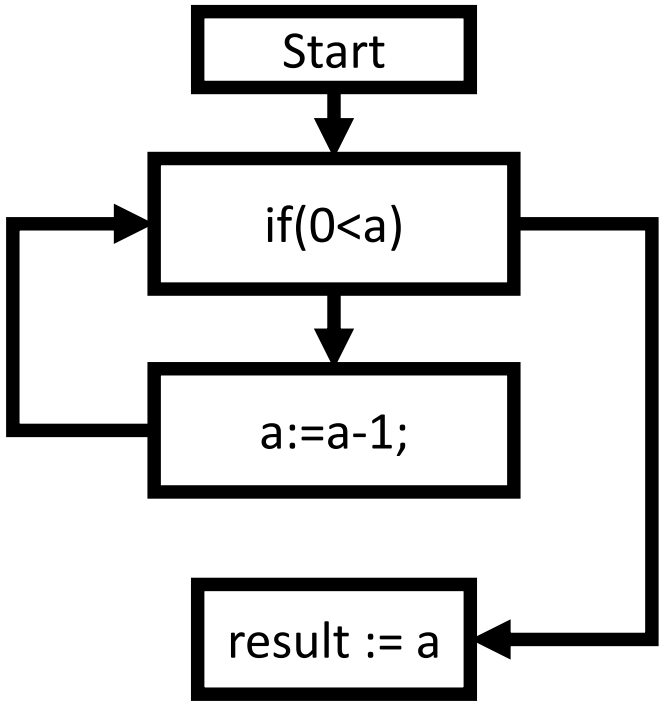


while

CFG

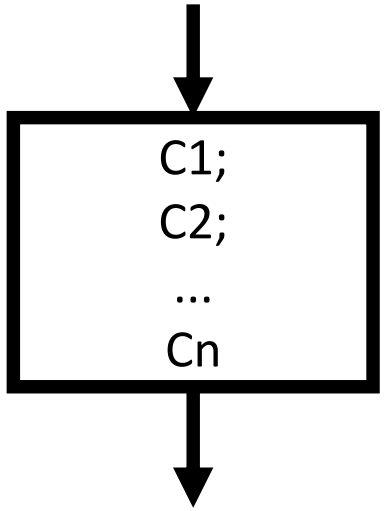
- Each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets
- Jump targets start a block, and jumps end a block
- Directed edges are used to represent jumps in the control flow

```
int example(int a)
{
	while(0 < a)
	{
		a := a - 1;
	}
	return a;
}
```

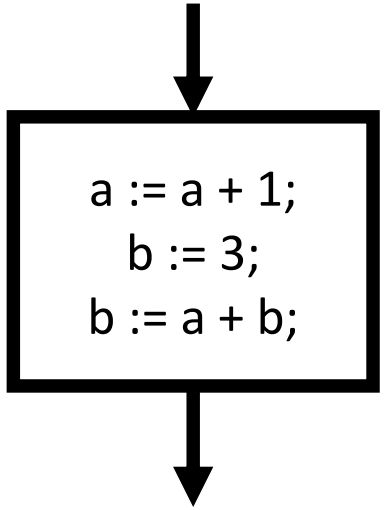


CFG for sequential programs

CFG(C1; C2; …; Cn)

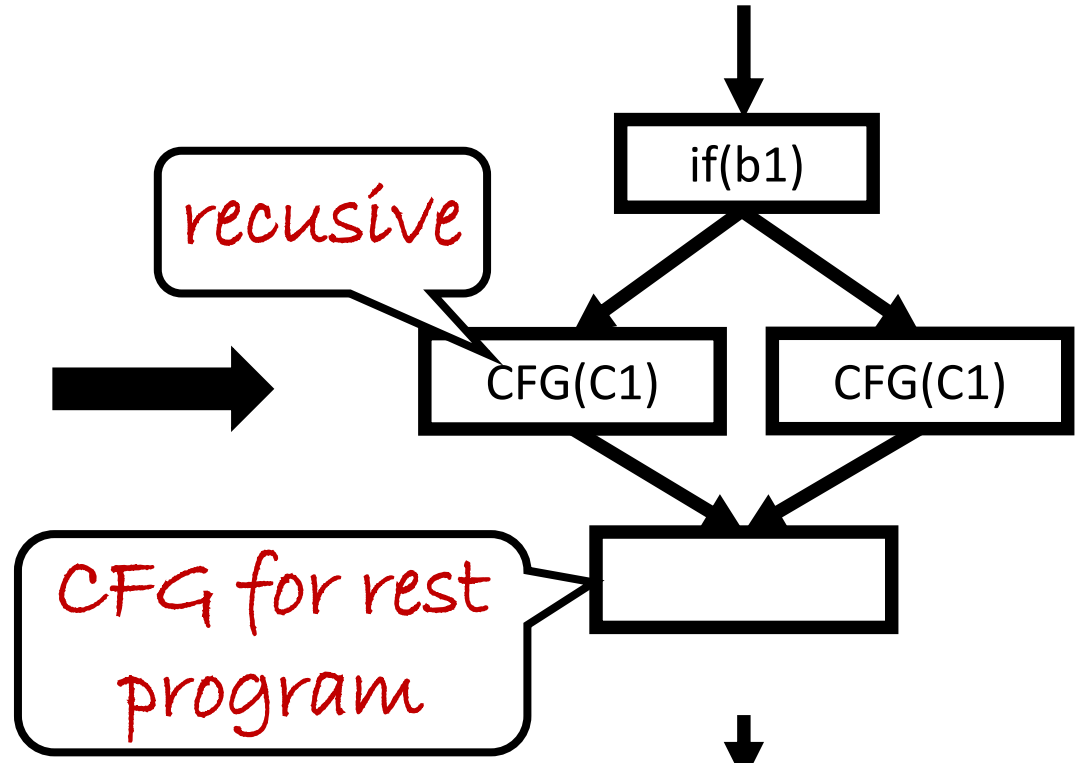



```
……
a := a + 1;
b := 3;
b := a + b;
……
```

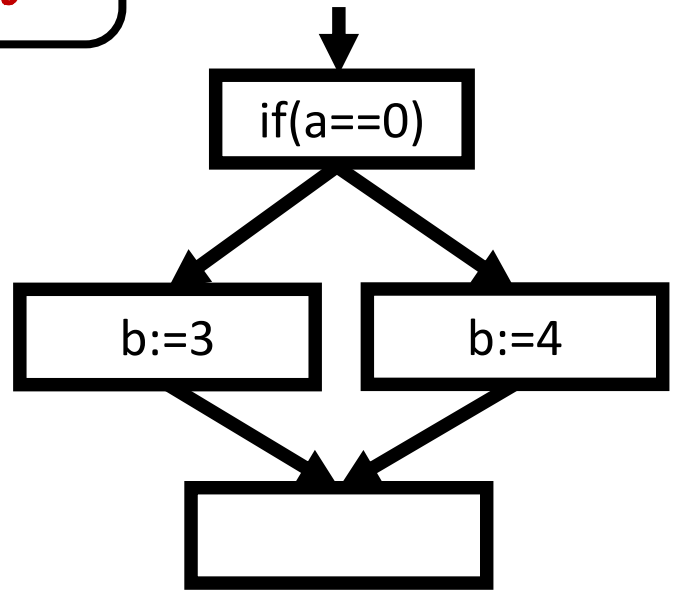


CFG for if-then-else

CFG(if(b1) C1; else C2)

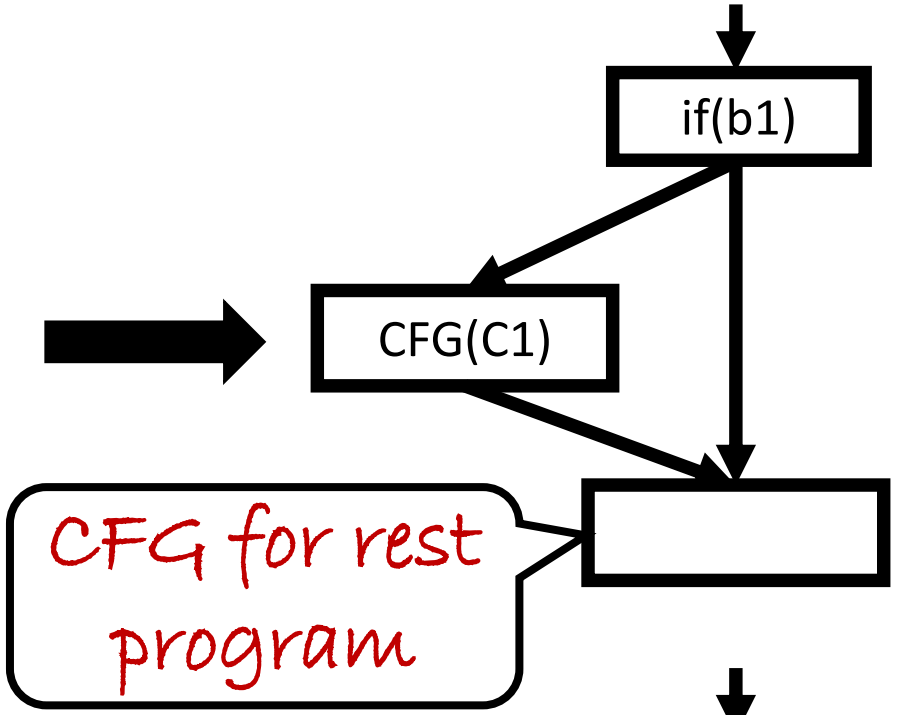



```
……
if(a == 0)
        b := 3;
else
        b := 4;
……
```

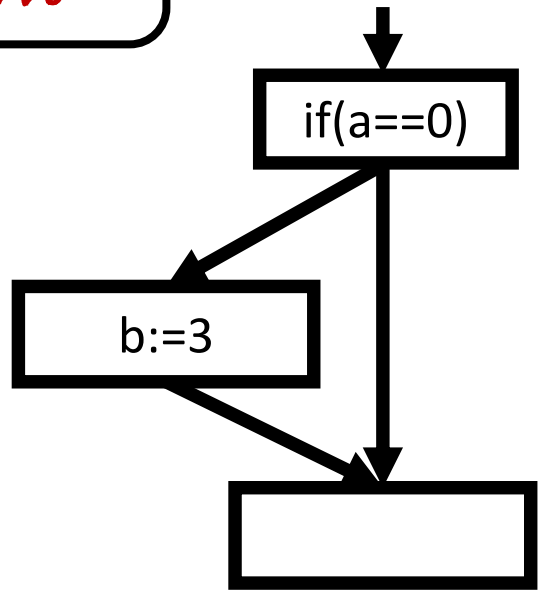


CFG for if-then

CFG(if(b1) C1)

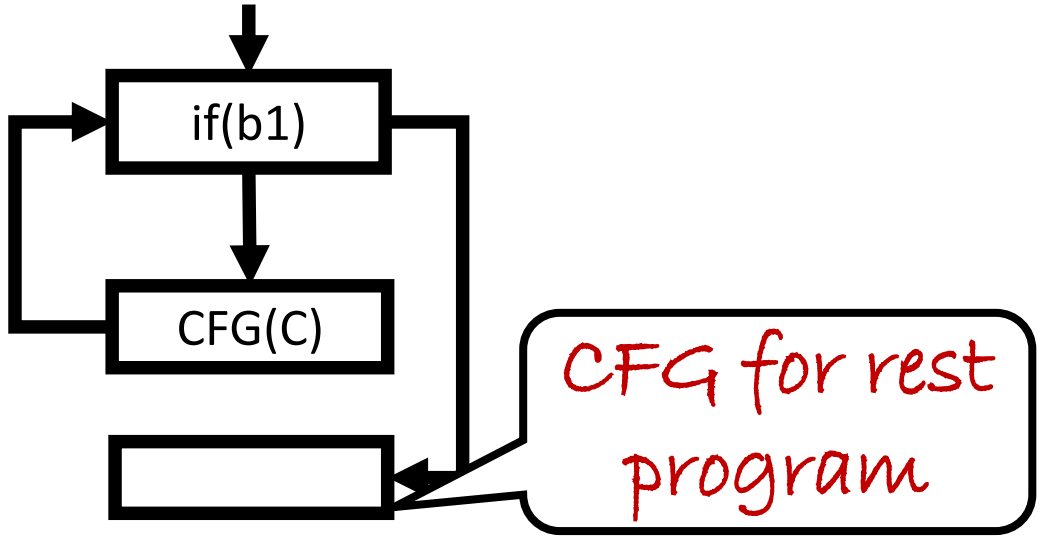



```
……
if(a == 0)
        b := 3;
……
```

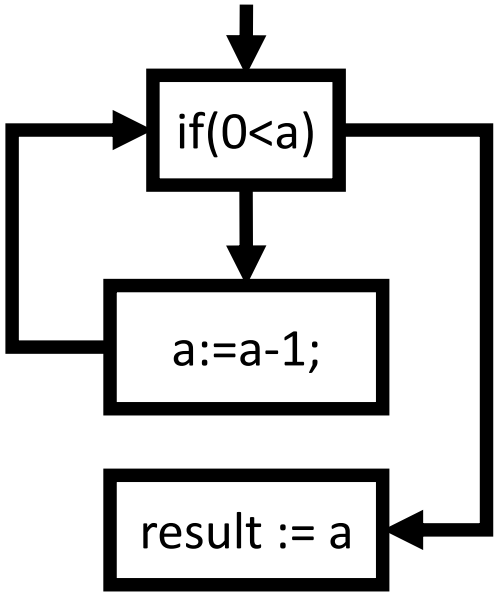


CFG for while

CFG(while(b1) C)

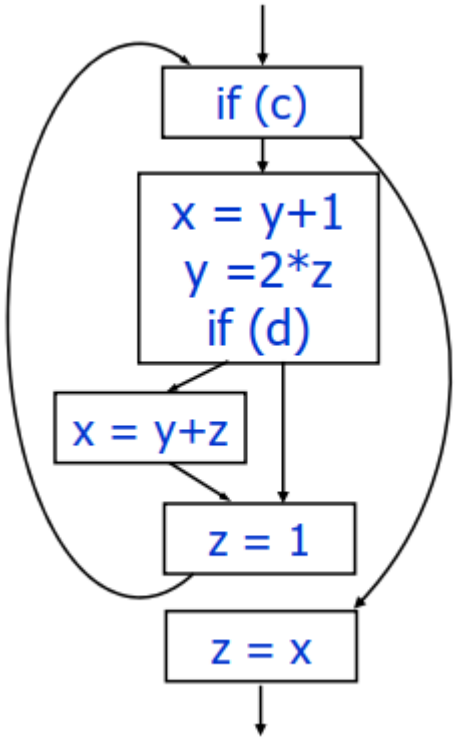



```
……
while(0 < a)
{
        a := a - 1;
}
return a;
```



Exercise

```
while (c) {
        x := y + 1;
        y := 2 * z;
        if (d) x := y+z;
        z := 1;
}
z := x;
```



Construct the CFG recursively.

Basic Design of An Automated Verifier

Input

- Invariants (e.g., Pre-condition) provided by user
- A program to be proved

Workflow

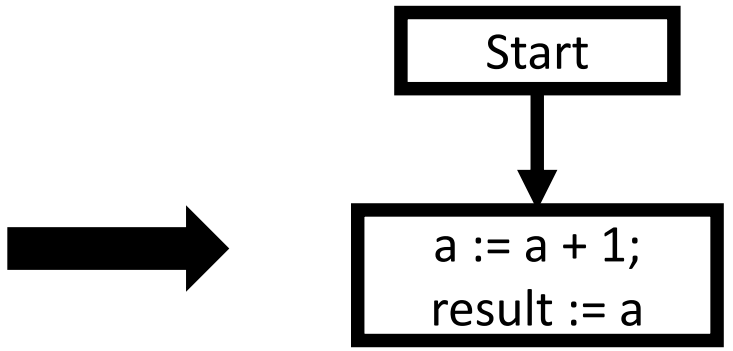
- Convert the program into Control Flow Graph (CFG).
- Insert the invariants into the CFG.
- Translate CFG into Hoare Triples
  - Find the weakest precondition
  - Prove the precondition imply the weakest precondition

Automated Verifier

BASIC IDEA

Automated Verifier transforms control flow graph to logical formulas and uses SMT solvers to solve it.

```
{a=1}
int f(int a)
{
        a := a + 1;
        return a;
}
{result=2}
```



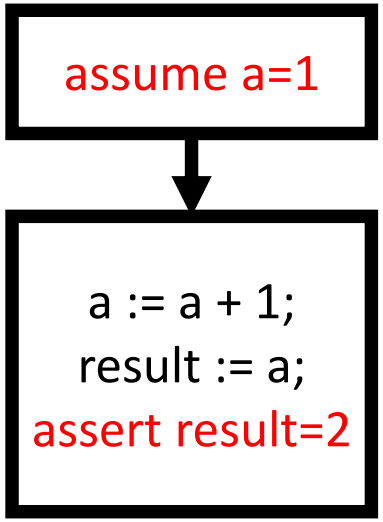
How to denote pre-/post-condition in CFG?

Automated Verifier

To denote pre-/post-conditions, we need add new statements.

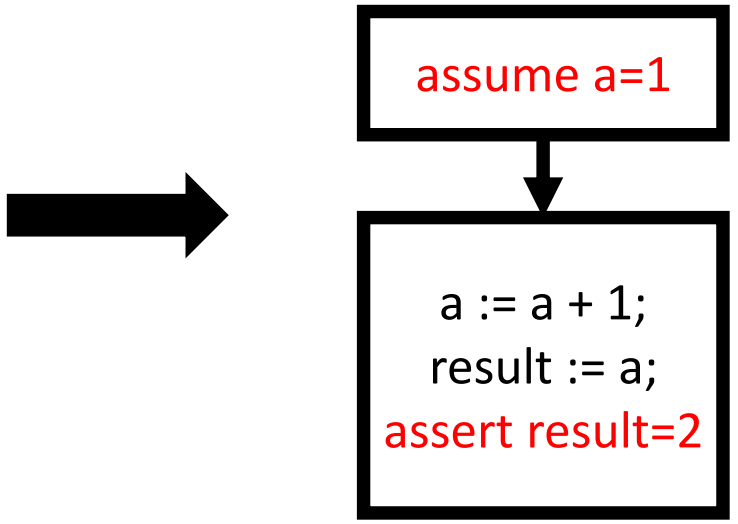
- assume A: assume A is true
- assert A: if A is true, then execute the rest program, else go to error state

```
{a=1}
int f(int a)
{
        a := a + 1;
        return a;
}
{result=2}
```



Automated Verifier

```
{a=1}
int f(int a)
{
        a := a + 1;
        return a;
}
{result=2}
```



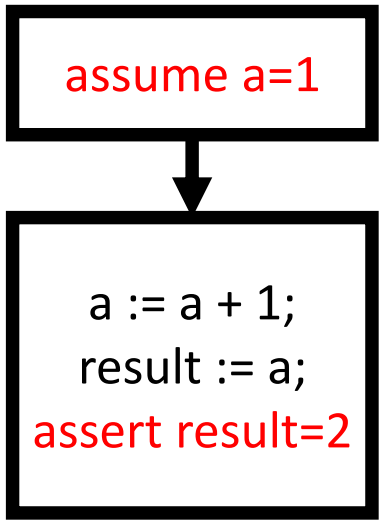
Because there are no loops, we can still use weakest preconditions to reduce the CFG to formulas.

$$wlp(assert\ P, Q) = P \wedge Q$$

$$wlp(assume\ P, Q) = P \rightarrow Q$$

Automated Verifier

```
{a=1}
int f(int a)
{
        a := a + 1;
        return a;
}
{result=2}
```



$$a = 1 \rightarrow$$

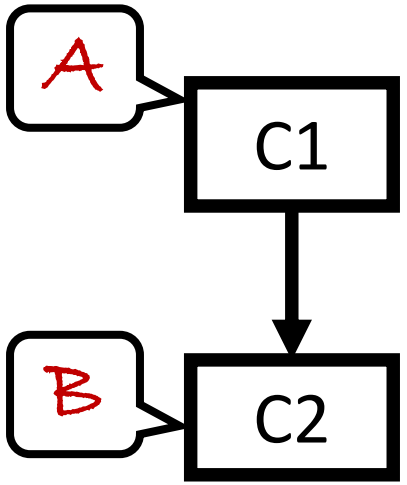
$$wlp(a := a + 1; result := a, result = 2)$$

$$a = 1 \rightarrow$$

$$a + 1 = 2$$

How to compute weakest precondition from CFG?

Automated Verifier



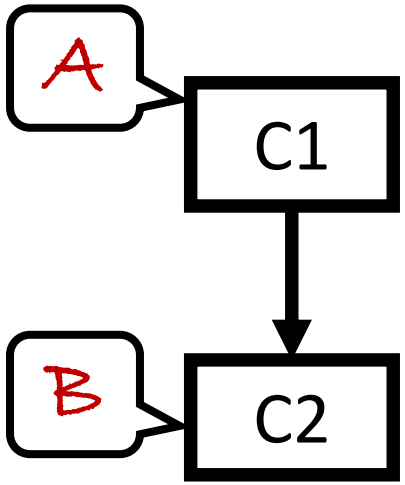
A_{ok} is true if the program is in a state from which all executions beginning from block A are correct.

We define the correctness of CFG:

$$A_{ok} \equiv wlp(C1, B_{ok})$$

Prove a program is correct becomes prove A_{ok} is True.

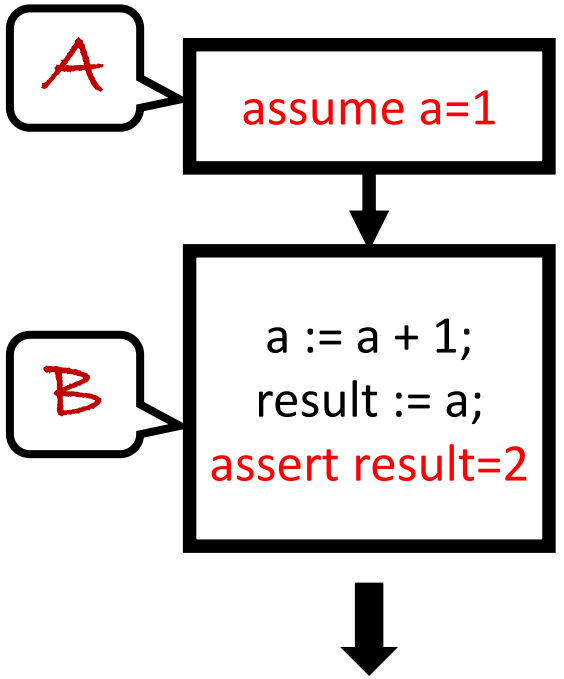
Automated Verifier



We define the correctness

of CFG like this:

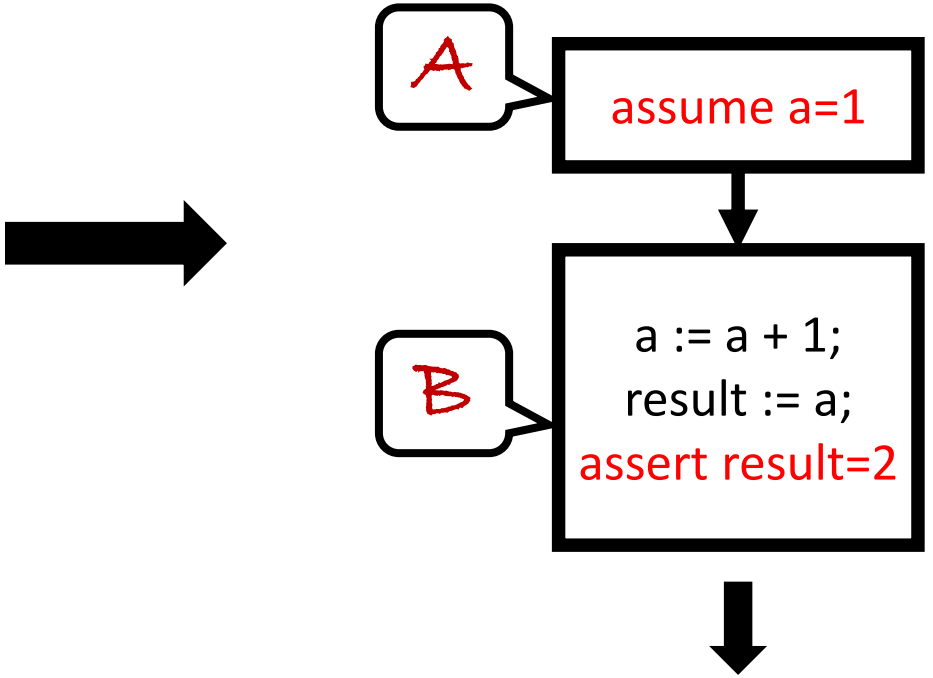
$$A_{ok} \equiv wlp(C1, B_{ok})$$



$$B_{ok} \equiv wlp(a := a + 1; result := a; assert\ result = 2, T)$$
$$\equiv wlp(a := a + 1; result := a, result = 2)$$
$$\equiv wlp(a := a + 1, a = 2)$$
$$\equiv a + 1 = 2$$
$$A_{ok} \equiv wlp(assume\ a = 1, B_{ok})$$
$$\equiv a = 1 \rightarrow a + 1 = 2$$

Automated Verifier

```
{a=1}
int f(int a)
{
        a := a + 1;
        return a;
}
{result=2}
```



$$a = 1 \rightarrow$$

$$wlp(a := a + 1; result := a, result = 2)$$

$$a = 1 \rightarrow$$

$$a + 1 = 2$$

$=$

$$B_{ok} \equiv wlp(a := a + 1; result := a; assert\ result = 2, T)$$

$$\equiv wlp(a := a + 1; result := a, result = 2)$$

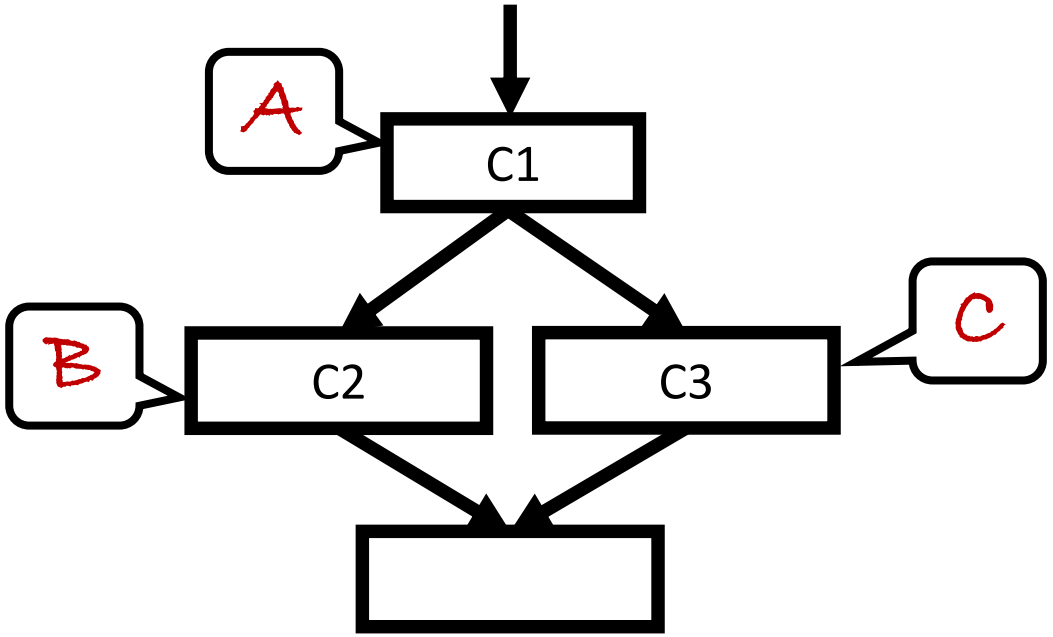
$$\equiv wlp(a := a + 1, a = 2)$$

$$\equiv a + 1 = 2$$

$$A_{ok} \equiv wlp(assume\ a = 1, B_{ok})$$

$$\equiv a = 1 \rightarrow a + 1 = 2$$

Automated Verifier

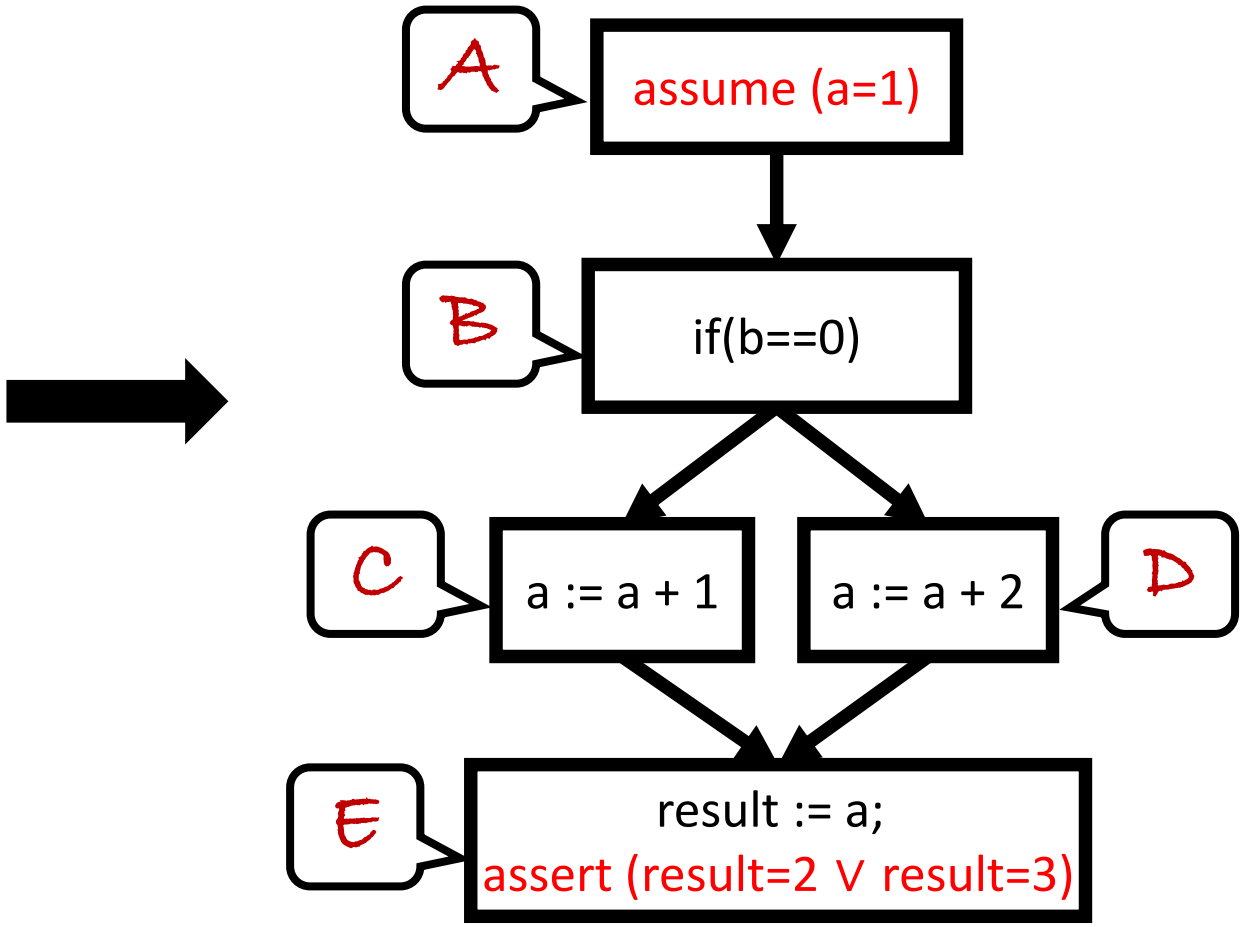


$$A_{ok} \equiv wlp(C1, \bigwedge_{S\ is\ Succ(A)} S_{ok})$$

Succ(A) denotes the set of successors of A in CFG

Automated Verifier

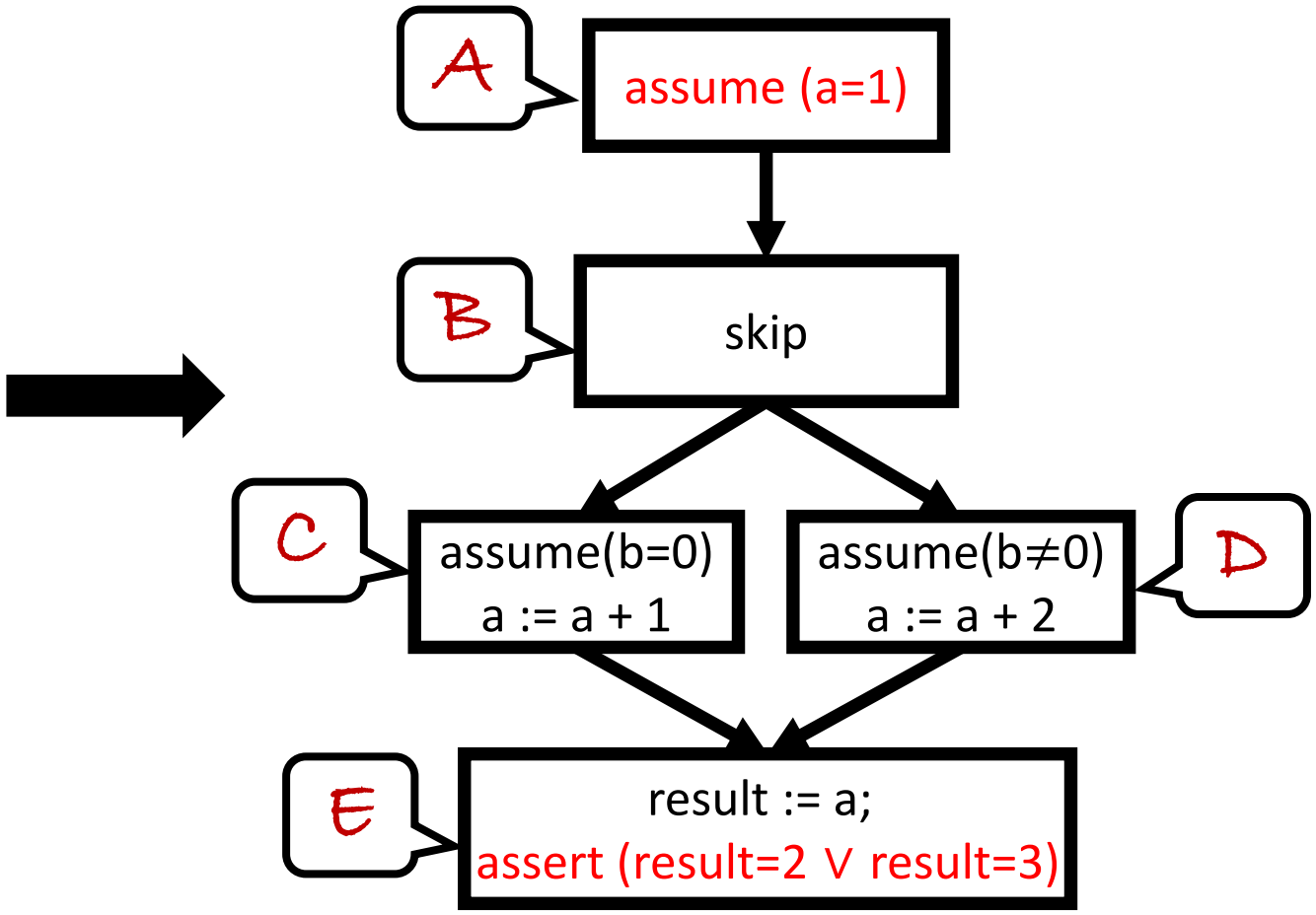
```
{a=1}
int f(int a, int b)
{
        if(b == 0)
                a := a + 1;
        else
                a := a + 2;
        return a;
}
{result=2 ∨ result=3}
```



When addressing branches, we need to add branch condition information to CFG.

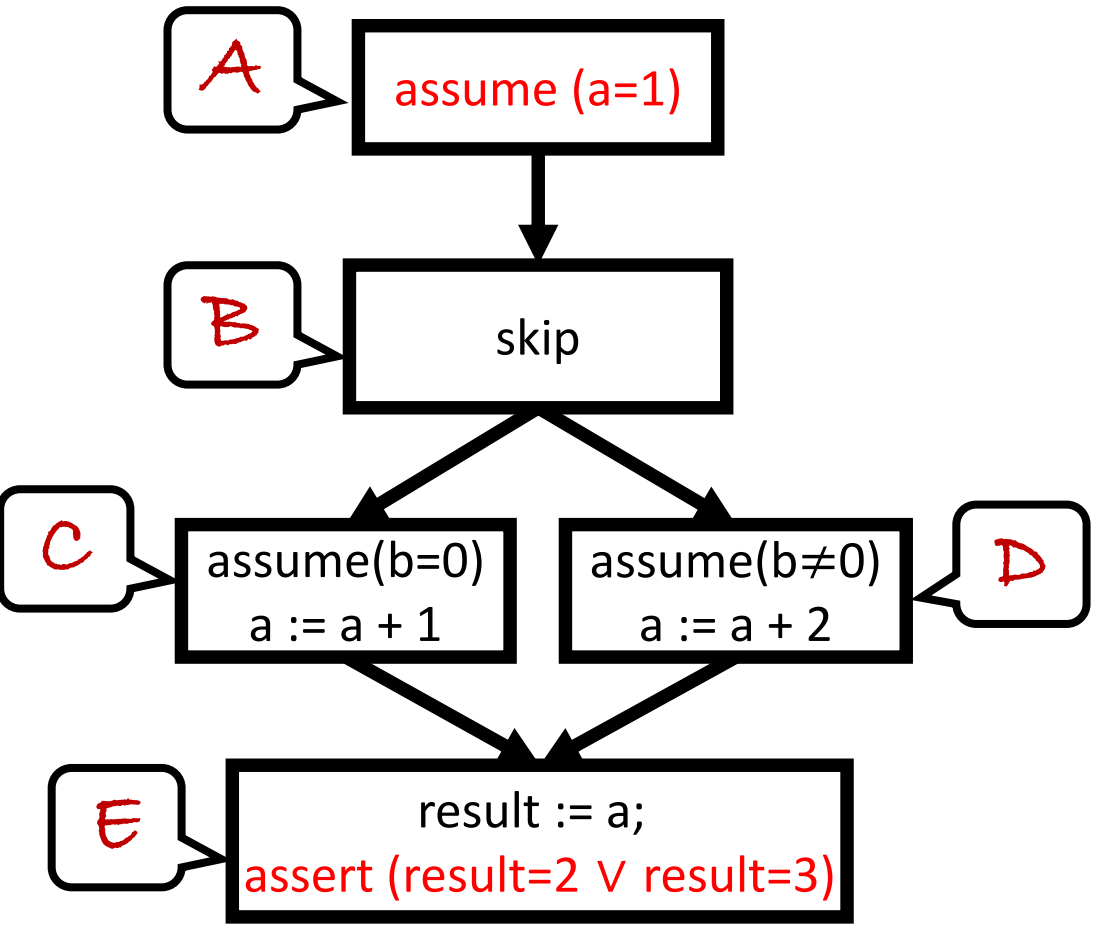
Automated Verifier

```
{a=1}
int f(int a, int b)
{
        if(b == 0)
                a := a + 1;
        else
                a := a + 2;
        return a;
}
{result=2 ∨ result=3}
```



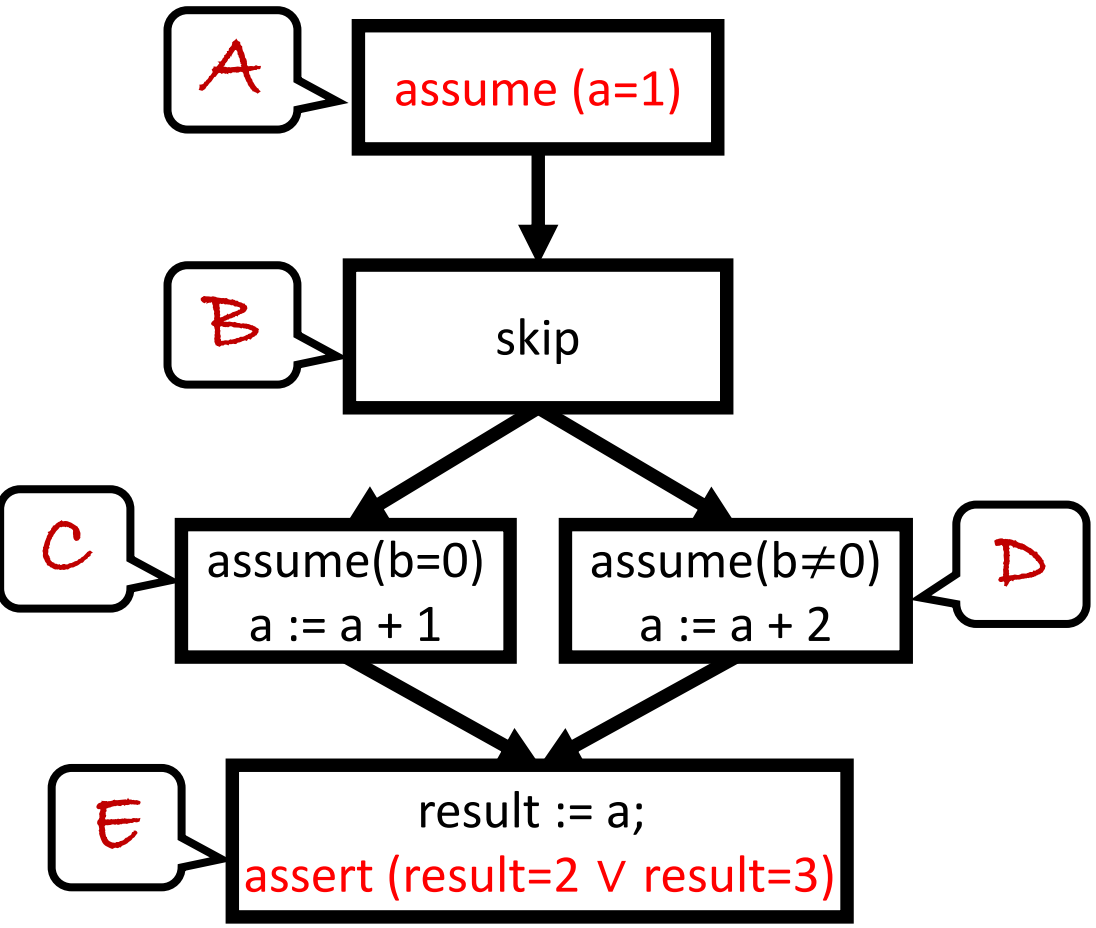
When addressing branches, we need to add branch condition information to CFG.

Automated Verifier



$$E_{ok} \equiv wlp(result := a; assert(result = 2 \vee result = 3), T)$$
$$\equiv wlp(result := a, result = 2 \vee result = 3)$$
$$\equiv (a = 2 \vee a = 3)$$

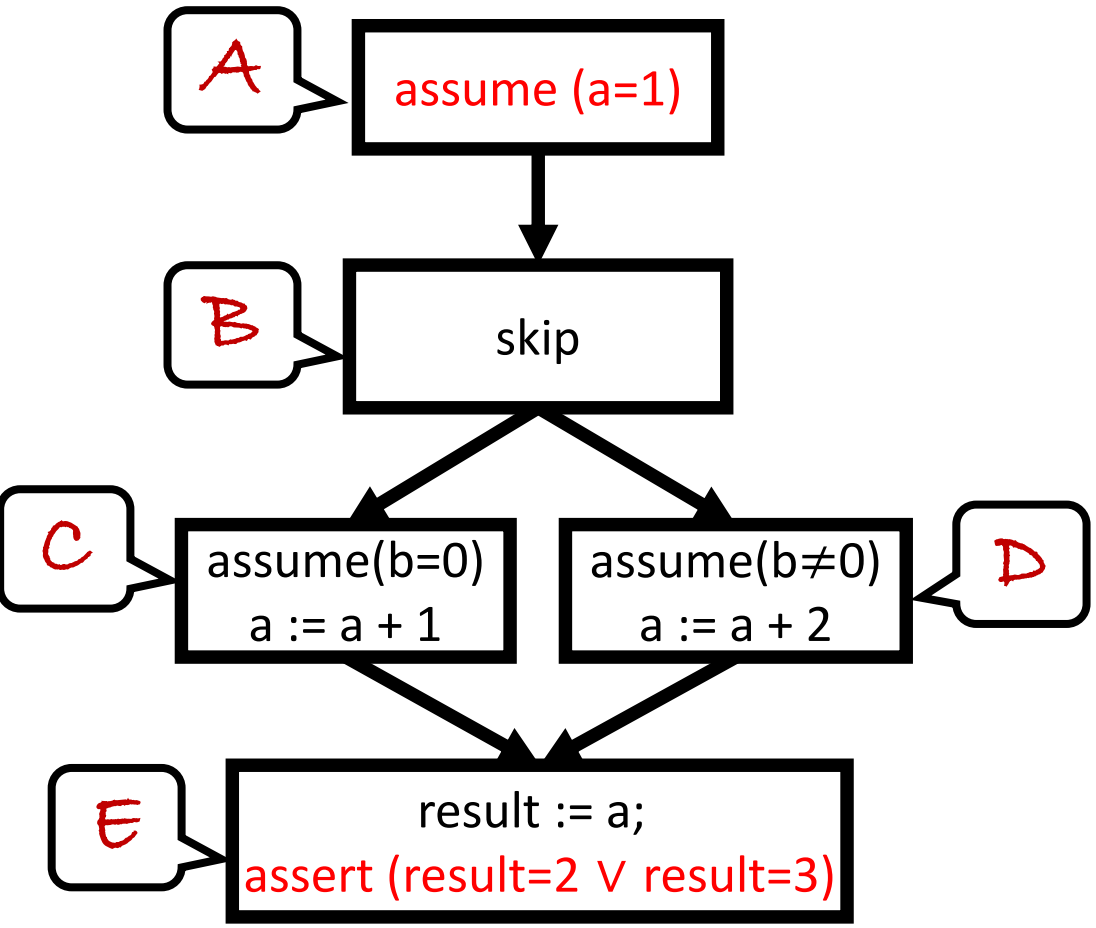
Automated Verifier



$$E_{ok} \equiv (a = 2 \vee a = 3)$$

$$\begin{aligned} D_{ok} &\equiv wlp(assume(b \neq 0); a := a + 2, E_{ok}) \\ &\equiv wlp(assume(b \neq 0), (a + 2 = 2 \vee a + 2 = 3)) \\ &\equiv (b \neq 0 \rightarrow (a + 2 = 2 \vee a + 2 = 3)) \end{aligned}$$

Automated Verifier



$$E_{ok} \equiv (a = 2 \vee a = 3)$$

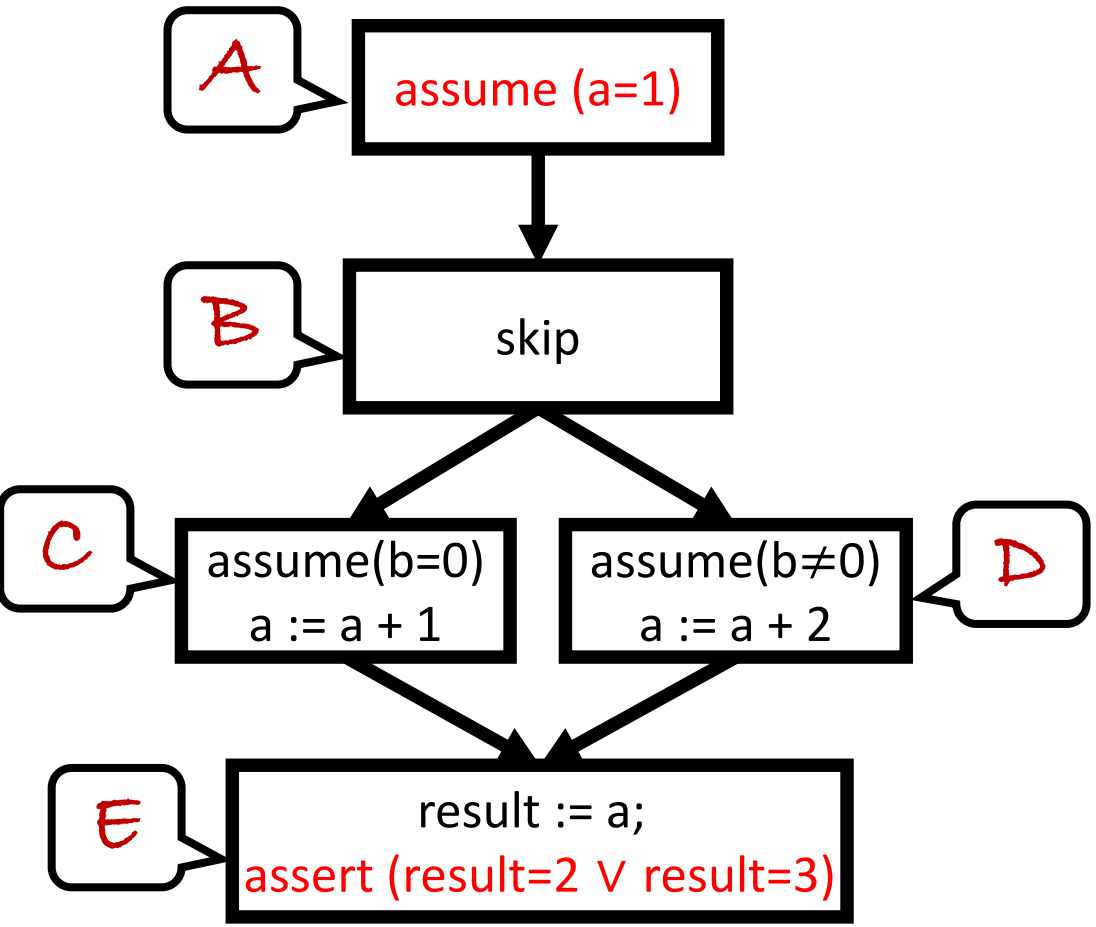
$$D_{ok} \equiv (b \neq 0 \rightarrow (a + 2 = 2 \vee a + 2 = 3))$$

$$C_{ok} \equiv wlp(assume(b = 0); a := a + 1, E_{ok})$$

$$\equiv wlp(assume(b = 0), (a + 1 = 2 \vee a + 1 = 3))$$

$$\equiv (b = 0 \rightarrow (a + 1 = 2 \vee a + 1 = 3))$$

Automated Verifier



$$E_{ok} \equiv (a = 2 \vee a = 3)$$

$$D_{ok} \equiv (b \neq 0 \rightarrow (a + 2 = 2 \vee a + 2 = 3))$$

$$C_{ok} \equiv (b = 0 \rightarrow (a + 1 = 2 \vee a + 1 = 3))$$

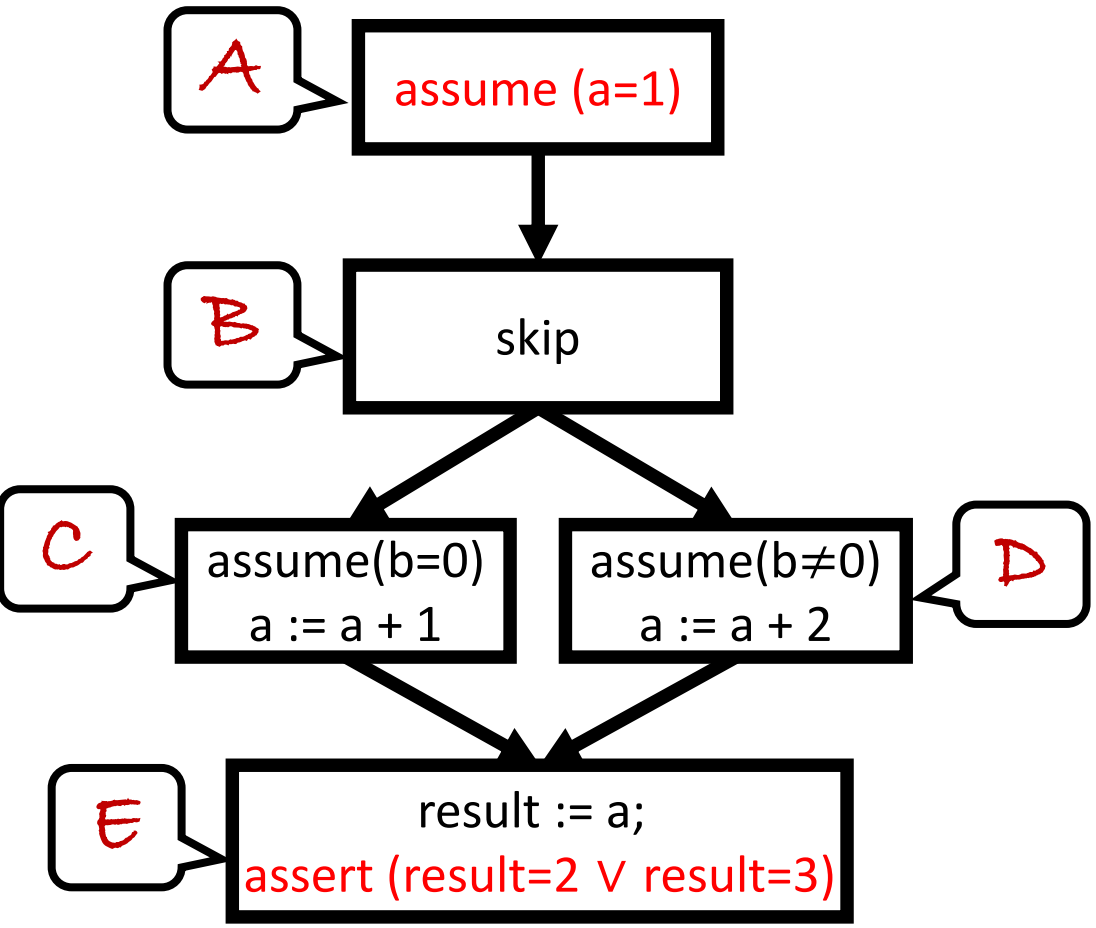
$$B_{ok} \equiv wlp(skip, C_{ok} \wedge D_{ok})$$

$$\equiv C_{ok} \wedge D_{ok}$$

$$\equiv (b = 0 \rightarrow (a + 1 = 2 \vee a + 1 = 3)) \wedge$$

$$(b \neq 0 \rightarrow (a + 2 = 2 \vee a + 2 = 3))$$

Automated Verifier



$$E_{ok} \equiv (a = 2 \vee a = 3)$$

$$D_{ok} \equiv (b \neq 0 \rightarrow (a + 2 = 2 \vee a + 2 = 3))$$

$$C_{ok} \equiv (b = 0 \rightarrow (a + 1 = 2 \vee a + 1 = 3))$$

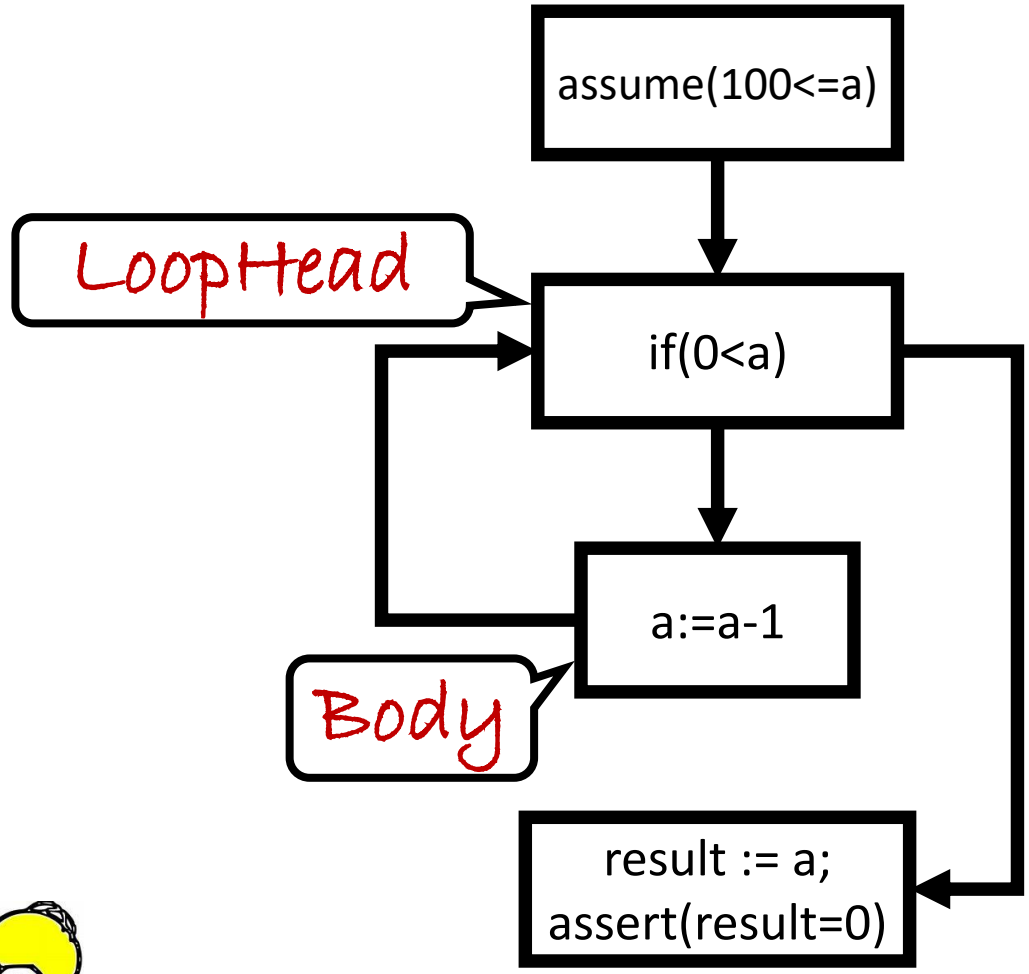
$$B_{ok} \equiv (b = 0 \rightarrow (a + 1 = 2 \vee a + 1 = 3)) \wedge$$
$$(b \neq 0 \rightarrow (a + 2 = 2 \vee a + 2 = 3))$$

$$A_{ok} \equiv wlp(assume(a = 1), B_{ok})$$
$$\equiv (a = 1) \rightarrow$$
$$(b = 0 \rightarrow (a + 1 = 2 \vee a + 1 = 3)) \wedge$$
$$(b \neq 0 \rightarrow (a + 2 = 2 \vee a + 2 = 3))$$

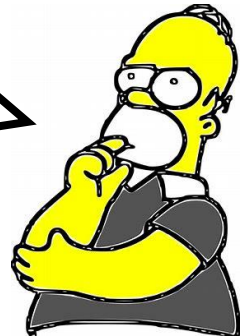
backward

Automated Verifier

```
{100<=a}
int example(int a)
{
        while(0 < a)
        {
                a := a - 1;
        }
        return a;
}
{result=0}
```

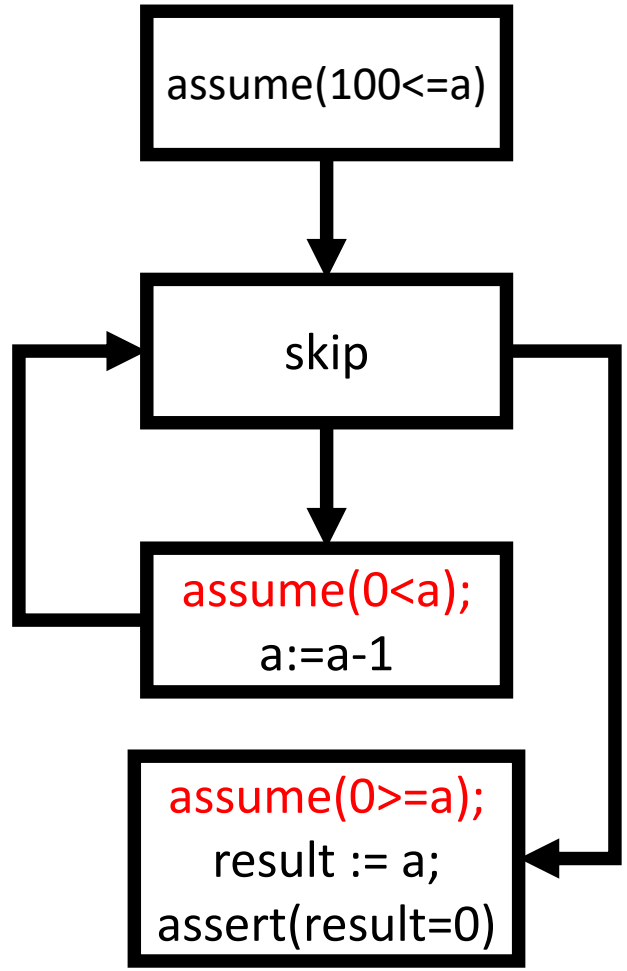


How to address loops?

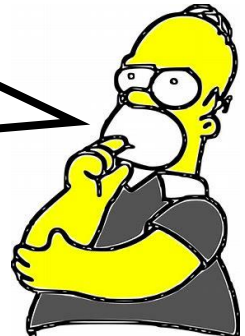


Automated Verifier

```
{100<=a}
int example(int a)
{
        while(0 < a)
        {
                a := a - 1;
        }
        return a;
}
{result=0}
```

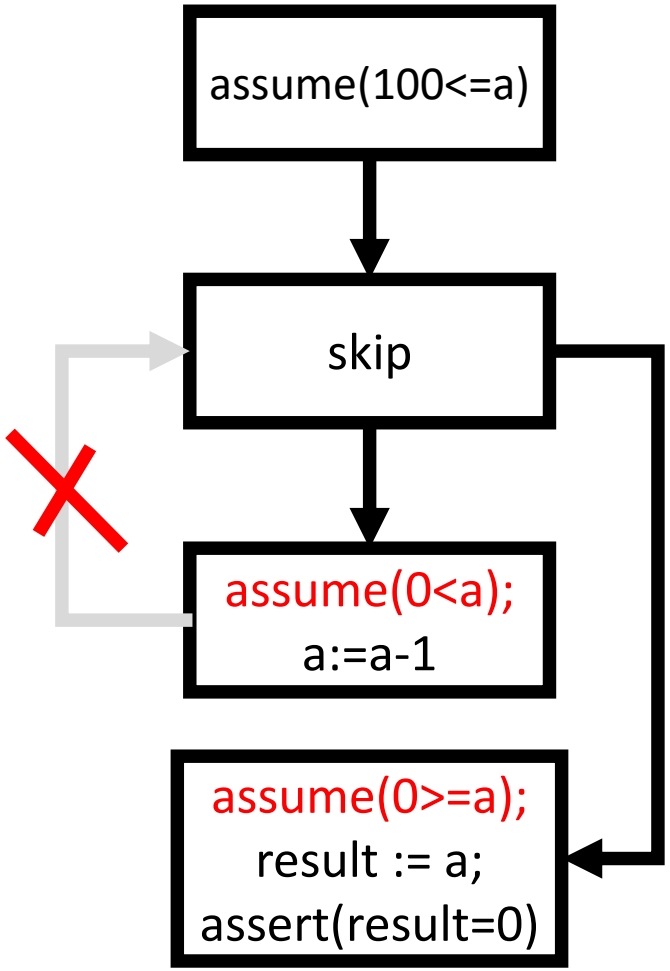


Like that in branch, we need to add loop guard.



Automated Verifier

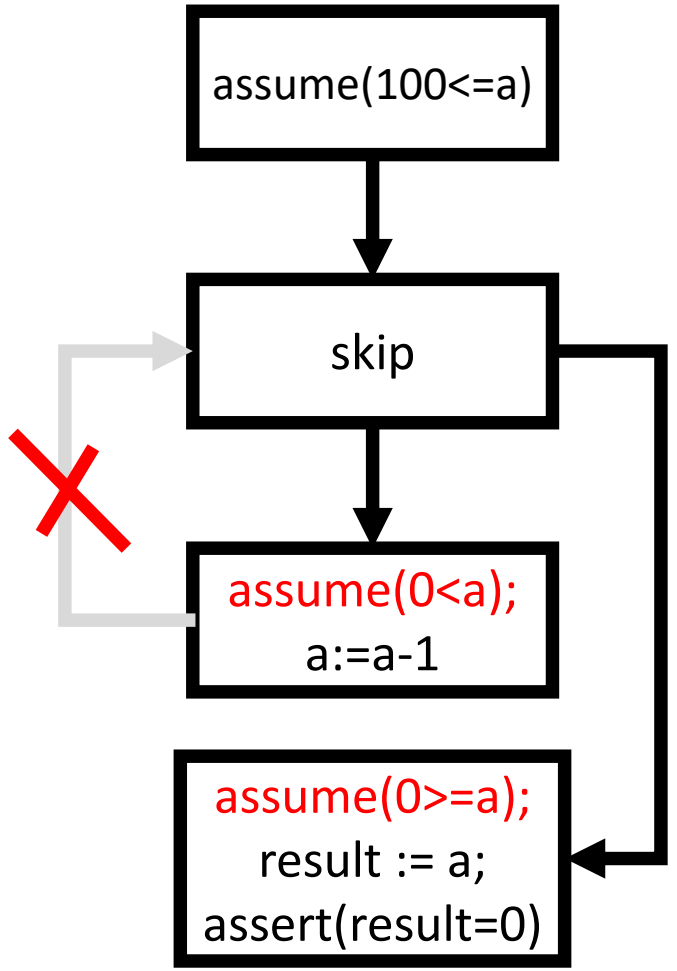
```
{100<=a}
int example(int a)
{
        while(0 < a)
        {
                a := a - 1;
        }
        return a;
}
{result=0}
```



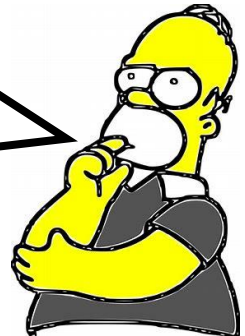
We must eliminate loops.
Otherwise, we cannot use backward methods like before.

Automated Verifier

```
{100<=a}
int example(int a)
{
        while(0 < a)
        {
                a := a - 1;
        }
        return a;
}
{result=0}
```



But it's clearly not correct. We change the meaning of programs.



A Quick Recap

loop invariant

$$\frac{\{I \wedge istrue(b1)\}C\{I\}}{\{I\}\ while(b1)\ C\ \{I \wedge isfalse(b1)\}}$$

The while rule (WHP)

It says that

- if executing C once preserves the truth of I, then executing C any number of times also preserves the truth of I
- after a while loop has terminated, the test must be false

Automated Verifier

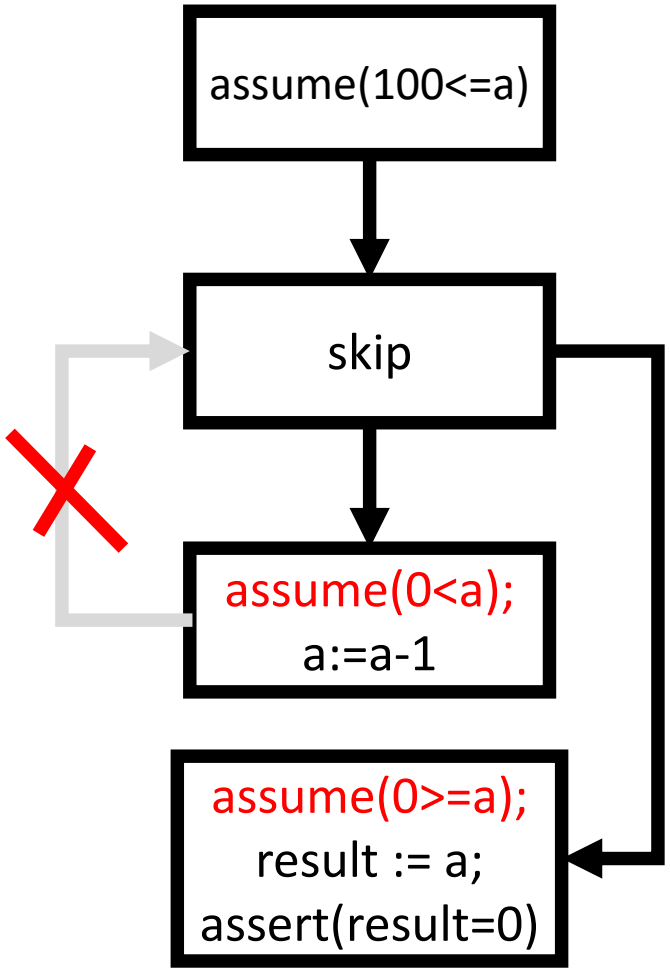
```
{100<=a}
int example(int a)
{
        while(0 < a)
                invariant a>=0
        {
                a := a - 1;
        }
        return a;
}
{result=0}
```

$$\frac{\{I \wedge istrue(b1)\}C\{I\}}{\{I\}\ while(b1)\ C\ \{I \wedge isfalse(b1)\}}$$

1) Prove invariant
2) Use invariant to generate wlp

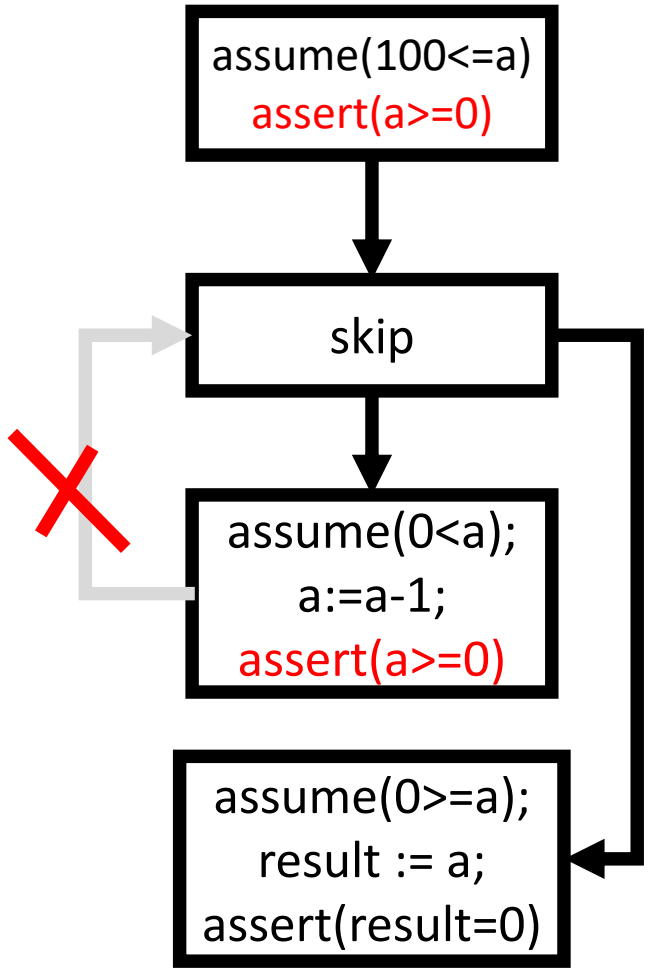
Automated Verifier

```
{100<=a}
int example(int a)
{
        while(0 < a)
                invariant a>=0
        {
                a := a - 1;
        }
        return a;
}
{result=0}
```



Automated Verifier

```
{100<=a}
int example(int a)
{
        while(0 < a)
                invariant a>=0
        {
                a := a - 1;
        }
        return a;
}
{result=0}
```

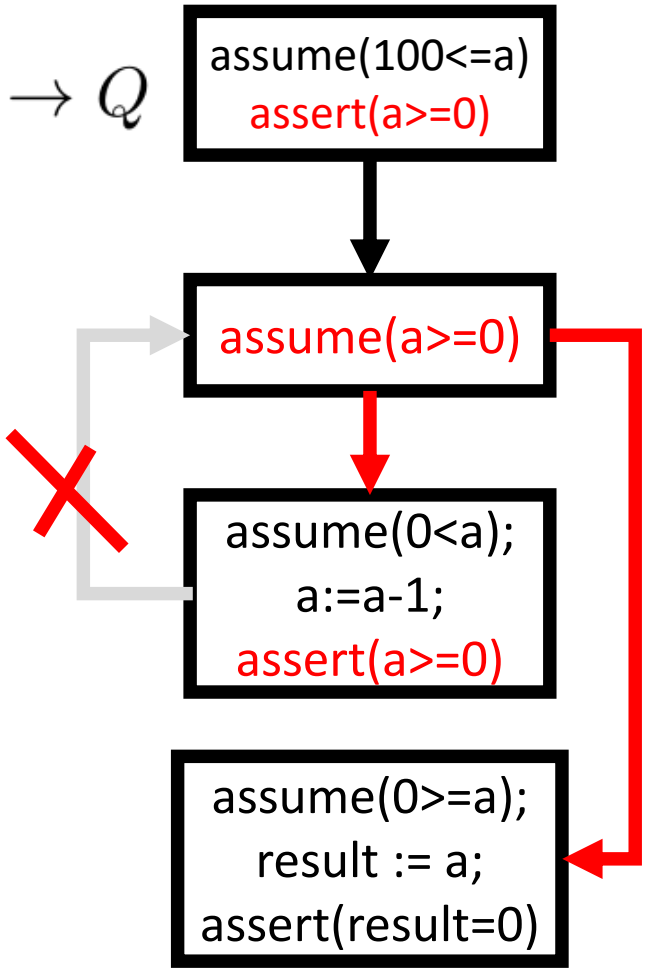


We can use assert to check invariant before and after the loop body.

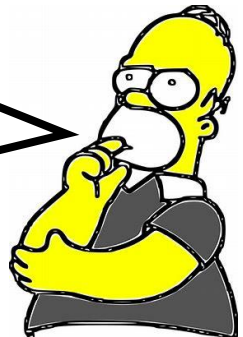
Automated Verifier

```
{100<=a}
int example(int a)
{
        while(0 < a)
                invariant a>=0
        {
                a := a - 1;
        }
        return a;
}
{result=0}
```

$$wlp(assume\ P, Q) = P \rightarrow Q$$



We can use assume to add invariant as premise.



Automated Verifier

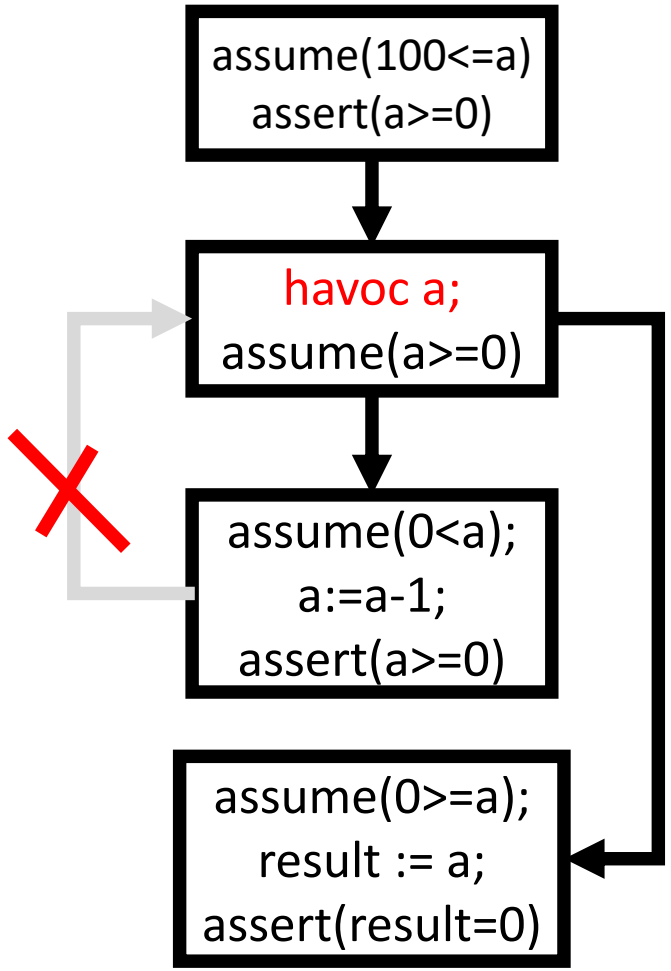
To address loops, we need another new statement.

- havoc a: set a to an arbitrary value
- Variables updated in loop body are called loop targets
- We need to havoc loop targets before loop. Otherwise, it is not a strict proof of invariant

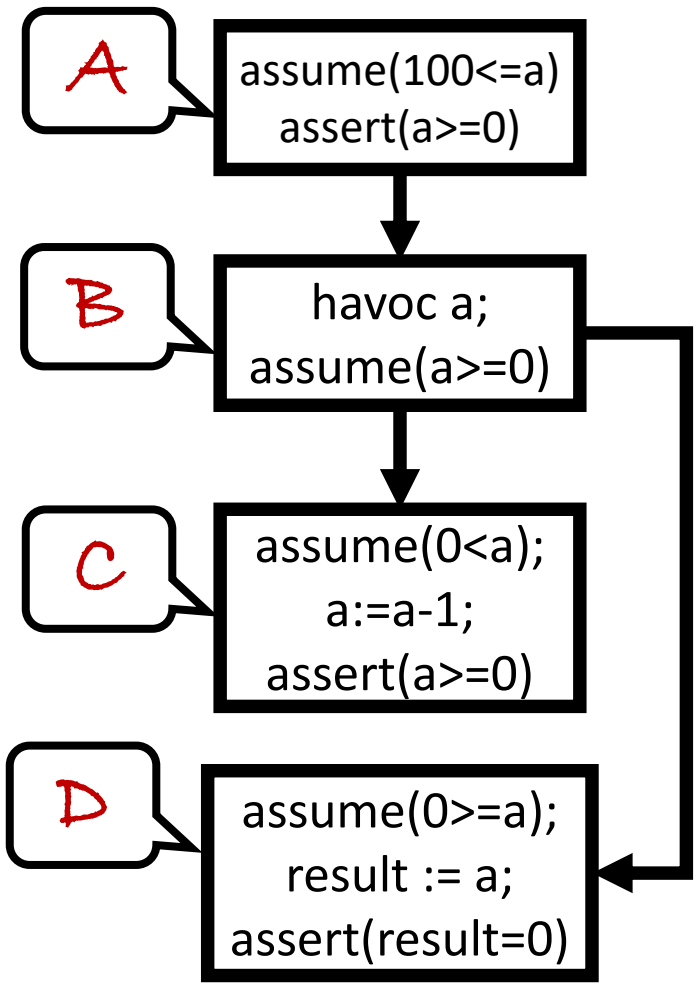
$$wlp(havoc\ a, Q) = (\forall v) Q[v/a]$$

Automated Verifier

```
{100<=a}
int example(int a)
{
        while(0 < a)
                invariant a>=0
        {
                a := a - 1;
        }
        return a;
}
{result=0}
```

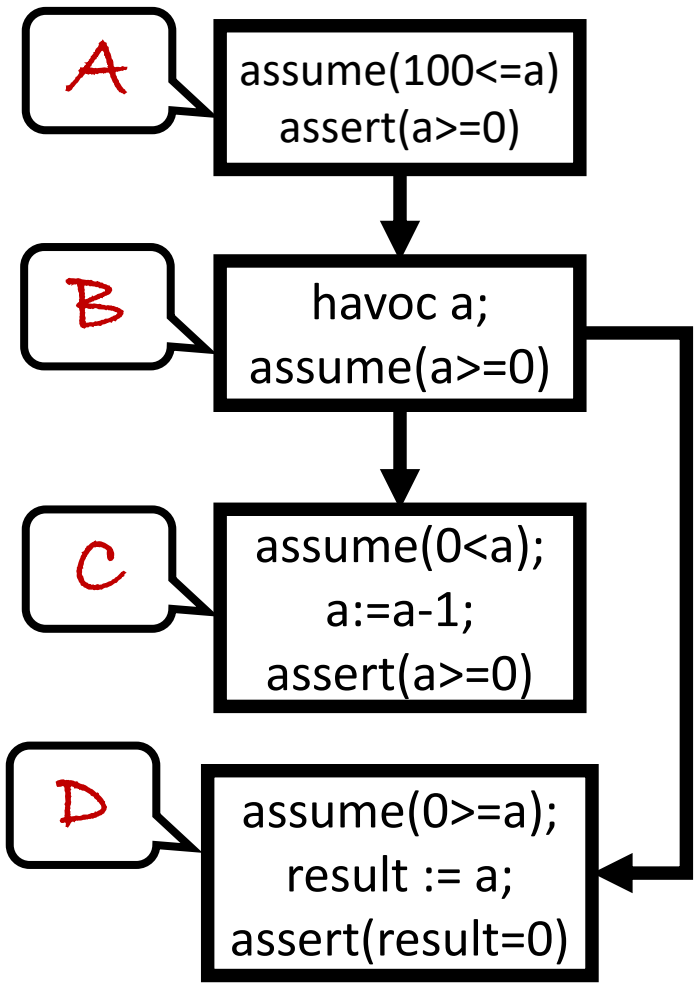


Automated Verifier



$$
\begin{aligned}
D_{ok} &\equiv wlp(assume(0 \geq a); result := a; assert(result = 0), T) \\
&\equiv wlp(assume(0 \geq a); result := a, result = 0) \\
&\equiv wlp(assume(0 \geq a), a = 0) \\
&\equiv (0 \geq a) \rightarrow a = 0
\end{aligned}
$$

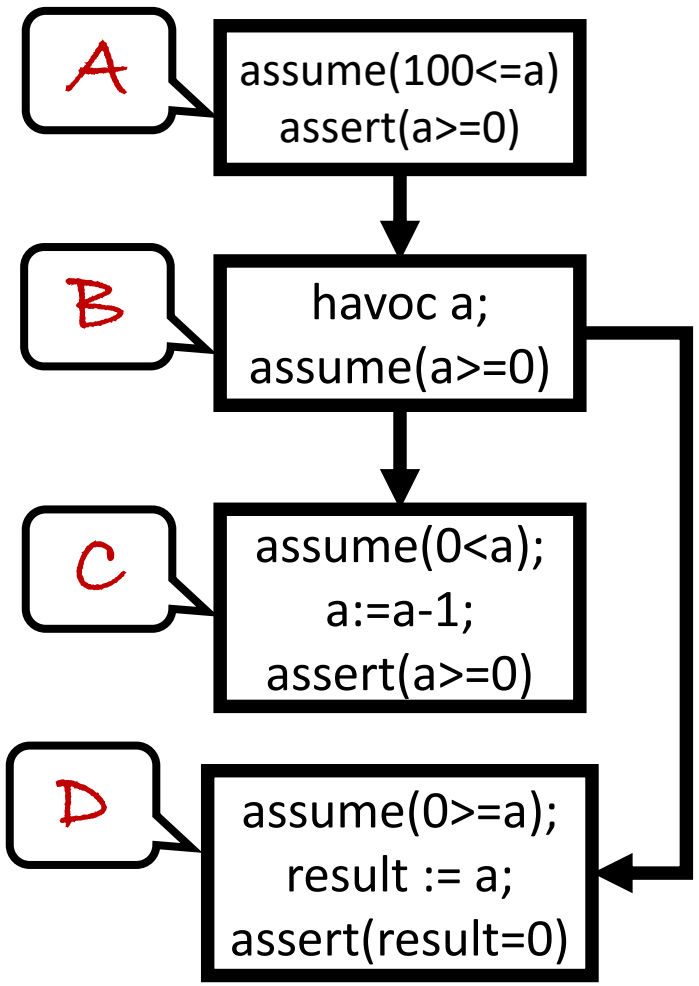
Automated Verifier



$$D_{ok} \equiv (0 \geq a) \rightarrow a = 0$$

$$\begin{aligned} C_{ok} &\equiv wlp(assume(0 < a); a := a - 1; assert(a \geq 0), T) \\ &\equiv wlp(assume(0 < a); a := a - 1, a \geq 0) \\ &\equiv wlp(assume(0 < a), (a - 1) \geq 0) \\ &\equiv (0 < a) \rightarrow (a - 1) \geq 0 \end{aligned}$$

Automated Verifier



$$D_{ok} \equiv (0 \geq a) \rightarrow a = 0$$

$$C_{ok} \equiv (0 < a) \rightarrow (a - 1) \geq 0$$

$$B_{ok} \equiv wlp(havoc\ \ a; assume(a \geq 0), C_{ok} \wedge D_{ok})$$

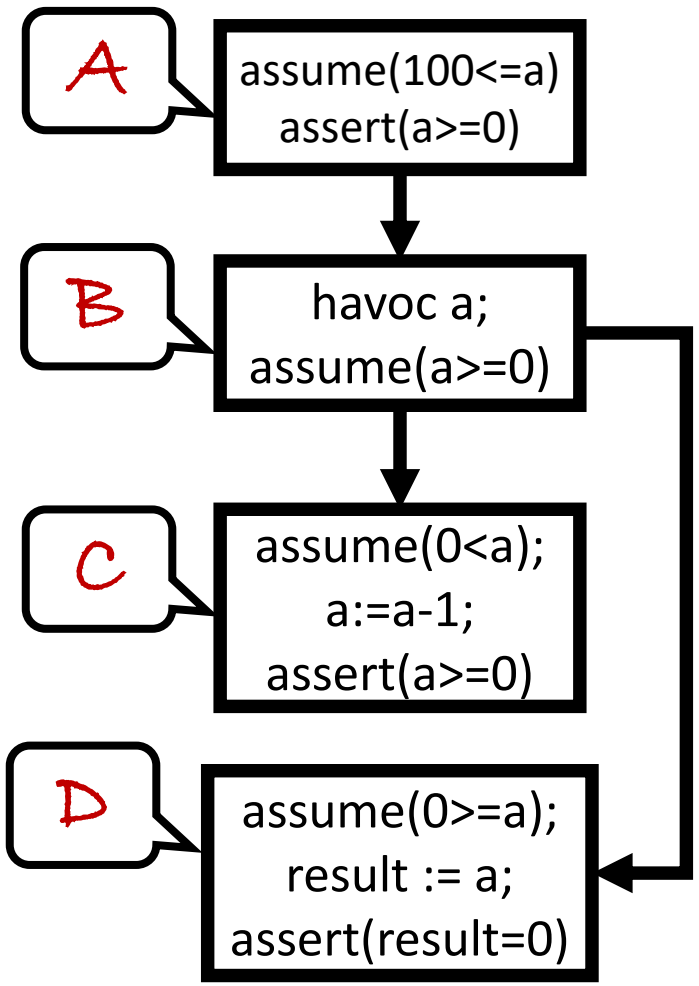
$$\equiv wlp(havoc\ \ a, (a \geq 0) \rightarrow C_{ok} \wedge D_{ok})$$

$$\equiv (\forall v)((a \geq 0) \rightarrow C_{ok} \wedge D_{ok})[v/a]$$

$$\equiv (\forall v)((a \geq 0) \rightarrow ((0 < a) \rightarrow (a - 1) \geq 0) \wedge ((0 \geq a) \rightarrow a = 0))[v/a]$$

$$\equiv (\forall v)((v \geq 0) \rightarrow ((0 < v) \rightarrow (v - 1) \geq 0) \wedge ((0 \geq v) \rightarrow v = 0))$$

Automated Verifier



$$D_{ok} \equiv (0 \geq a) \rightarrow a = 0$$

$$C_{ok} \equiv (0 < a) \rightarrow (a - 1) \geq 0$$

$$B_{ok} \equiv (\forall v)((v \geq 0) \rightarrow ((0 < v) \rightarrow (v - 1) \geq 0) \wedge ((0 \geq v) \rightarrow v = 0))$$

$$A_{ok} \equiv wlp(assume(100 \leq a); assert(a \geq 0), B_{ok})$$

$$\equiv wlp(assume(100 \leq a), (a \geq 0) \wedge B_{ok})$$

$$\equiv (100 \leq a) \rightarrow ((a \geq 0) \wedge B_{ok})$$

$$\equiv (100 \leq a) \rightarrow$$

$$((a \geq 0) \wedge$$

$$((\forall v)((v \geq 0) \rightarrow ((0 < v) \rightarrow (v - 1) \geq 0) \wedge ((0 \geq v) \rightarrow v = 0))))$$

Dafny

post-condition

```
method m(n: nat) returns (result: nat)
  ensures n == result
{
    var i: int := 0;
    while i < n
        invariant i <= n
    {
        i := i + 1;
    }
    return i;
}
```

loop invariant

'►' shortcut: Alt+B

https://rise4fun.com/Dafny/tutorial

```
Dafny 2.3.0.10506

Dafny program verifier finished with 1 verified, 0 errors
Program compiled successfully
```

Dafny

Komodo: Using verification to disentangle secure-enclave hardware from software

Andrew Ferraiuolo
Cornell University

Andrew Baumann
Microsoft Research

Chris Hawblitzel
Microsoft Research

Bryan Parno
Carnegie Mellon University

5 SPECIFICATION

We specify and verify Komodo using Dafny [51], a general-purpose verification language. This section describes our trusted Dafny specifications of ARM assembly language (§5.1) and of Komodo's overall correctness (§5.2). We increase our confidence in the Komodo specification by proving several high-level lemmas: that it maintains consistency

Thanks & Questions