Software Verification with Satisfiability Modulo Theories
- Software Verification -

Ming-Hsien Tsai

Institute of Information Science
Academia Sinica

FLOLAC 2017

Based on the slides of Yih-Kuen

Outline

- Hoare logic
- Weakest precondition
- Frama-C
- Tools

Hoare Logic

- *Hoare logic* is an axiomatic approach to program correctness
- Properties of programs can be verified in a *deductive* manner: applying *inference rules* to a set of axioms
- Different program languages may need different inference rules
- It is possible to automate the deductive verification

Assertions

- A time snapshot of a program execution is a *state*, which maps program variables to their values at that time.

- A program execution is an evolution of states.

- An *assertion* is a statement about states of a program.

    - $x < 2^{51} \wedge y < 2^{15}$

    - $res \equiv (x \cdot y) \; mod \; 2^{255\text{-}19}$

- Most interesting assertions can be expressed in FOL.

Pre- and Post-conditions

- Put an assertion at the entry point of a program to specify the requirements of inputs: *pre-condition*
- Put an assertion at the exit point of a program to specify the guarantees of outputs: *post-condition*

Hoare Triples

- A program C annotated with pre-condition P and post-condition Q is a *Hoare triple*: $\{\ P\ \}\ C\ \{\ Q\ \}$
- Validity of a Hoare triple
  - *Partial correctness*: If the program starts with a state satisfying P and terminates at a final state, then the final state satisfies Q
  - *Total correctness*: If the program starts with a state satisfying P, then the program must terminate at a final state and the final state satisfies Q
- If a Hoare triple is interpreted as total correctness, it is sometimes written as $\langle\ P\ \rangle\ C\ \langle\ Q\ \rangle$

Specifications

- A program specification can be written as a Hoare triple, plus assertions inserted in the program

- If the Hoare triple can be shown to be valid, then the program satisfies the specification

- For a function that returns a result, we use the variable res to represent the returned result.

Examples

- $\{y \neq 0\}\ div(x,\ y)\ \{res = x\ /\ y\}$
- $\{size(ls) = n\}\ sort(ls,\ n)\ \{sorted(ls) \wedge size(ls) = n\}$
  - $size$ and $sorted$ are first-order functions
- $\{x < y \wedge y < z \wedge z < w \wedge w{+}x = y{+}z \wedge x{+}y{=}z{+}w\}\ C\ \{Q\}$
  - always valid for integer variables x, y, z, and w

Examples

- $\{y \neq 0\}\ div(x,\ y)\ \{res = x\ /\ y\}$
- $\{size(ls) = n\}\ sort(ls,\ n)\ \{sorted(ls) \wedge size(ls) = n\}$
  - $size$ and $sorted$ are first-order functions
- $\{x < y \wedge y < z \wedge z < w \wedge w{+}x = y{+}z \wedge x{+}y{=}z{+}w\}\ C\ \{Q\}$
  - always valid for integer variables x, y, z, and w

Be careful of writing specifications

Exercise

- Let max be a function that returns the maximal number between two input numbers. Write a specification of max as precise as possible.
  - { ? } max(x, y) { ? }
- Write the specification of a function that concatenates two integer lists. You may define other functions of list and use them in the specification.
  - list ::= nil | cons(Int, list)

Assignment

$$x := e$$

- Assume that the evaluation of e does not cause any side-effect
- $P[e/x]$: change x to e in P
- Which one is correct?
  - $\{P\}\ x := e\ \{P[e/x]\}$
  - $\{Q[e/x]\}\ x := e\ \{Q\}$

Assignment

$$x := e$$

- Assume that the evaluation of e does not cause any side-effect
- $P[e/x]$: change x to e in P
- Which one is correct?
  - $\{P\}\ x := e\ \{P[e/x]\}$ ✗ $\{x - 1 = 0\}\ x := 2\ \{2 - 1 = 0\}$
  - $\{Q[e/x]\}\ x := e\ \{Q\}$

Assignment

$$x := e$$

- Assume that the evaluation of e does not cause any side-effect
- $P[e/x]$: change x to e in P
- Which one is correct?
  - $\{P\}\ x := e\ \{P[e/x]\}$ ✗ $\{x - 1 = 0\}\ x := 2\ \{2 - 1 = 0\}$
  - $\{Q[e/x]\}\ x := e\ \{Q\}$ ○ $\{2 - 1 > 0\}\ x := 2\ \{x - 1 > 0\}$

Assignment
More Examples

- $\{x > 5\}\ x := x - 1\ \{x \geq 0\}$
- $\{x - 1 \geq 0\}\ x := x - 1\ \{x \geq 0\}$
- $\{(x+1)+y > \mathrm{z}\}\ x := x + 1\ \{x+y > \mathrm{z}\}$

Assignment Axiom

$$\frac{}{\{\ Q[e/x]\ \}\ x := e\ \{\ Q\ \}}\ \text{Assign}$$

- No side-effect: only x is changed
- x in post-condition has a new value same as e to satisfy Q
- What if x does not have value same as e?
  - Change x to e would satisfy Q

Multiple Assignment

$$x_1,\ x_2,\ \ldots,\ x_n := e_1,\ e_2,\ \ldots,\ e_n$$

where x's are distinct variables

$$\frac{}{\{\ Q[e_1,e_2,\ldots,e_n/x_1,x_2,\ldots,x_n]\ \}\ x_1,\ x_2,\ \ldots,\ x_n := e_1,\ e_2,\ \ldots,\ e_n\ \{\ Q\ \}}\ \text{MultiAssign}$$

- $Q[e_1,e_2,\ldots,e_n/x_1,x_2,\ldots,x_n]$ is the result of simultaneous substitution
- $(x < y)[y,x/x,y] = (y < x)$

Proof Rules

$$\frac{}{\{\ Q[e/x]\ \}\ x := e\ \{\ Q\ \}}\ \text{Assign}$$

$$\frac{\{\ P \wedge B\ \}\ S_1\ \{\ Q\ \} \qquad \{\ \mathrm{P} \wedge \neg B\ \}\ S_2\ \{\ Q\ \}}{\{\ P\ \}\ \textbf{If}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{\ Q\ \}}\ \text{Conditional}$$

$$\frac{}{\{\ Q\ \}\ \textbf{skip}\ \{\ Q\ \}}\ \text{Skip}$$

$$\frac{\{\ P \wedge B\ \}\ S\ \{\ Q\ \} \qquad P \wedge \neg B \rightarrow Q}{\{\ P\ \}\ \textbf{If}\ B\ \textbf{then}\ S\ \textbf{fi}\ \{\ Q\ \}}\ \text{If-Then}$$

$$\frac{\{\ P\ \}\ S_1\ \{\ Q\ \} \qquad \{\ Q\ \}\ S_2\ \{\ R\ \}}{\{\ P\ \}\ S_1;\ S_2\ \{\ R\ \}}\ \text{Sequence}$$

$$\frac{\{\ P \wedge B\ \}\ S\ \{\ P\ \}}{\{\ P\ \}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\ \{\ P \wedge \neg B\ \}}\ \text{While}$$

$$\frac{P \rightarrow P' \qquad \{\ P'\ \}\ S\ \{\ Q'\ \} \qquad Q' \rightarrow Q}{\{\ P\ \}\ S\ \{\ Q\ \}}\ \text{Consequence}$$

Proof Rules (cont'd)

$$\frac{P \rightarrow P' \qquad \{\ P'\ \}\ S\ \{\ Q\ \}}{\{\ P\ \}\ S\ \{\ Q\ \}}$$ Strengthening Precondition

$$\frac{\{\ P\ \}\ S\ \{\ Q'\ \} \qquad Q' \rightarrow Q}{\{\ P\ \}\ S\ \{\ Q\ \}}$$ Weakening Postcondition

$$\frac{\{\ P_1\ \}\ S\ \{\ Q_1\ \} \qquad \{\ P_2\ \}\ S\ \{\ Q_2\ \}}{\{\ P_1 \wedge P_2\ \}\ S\ \{\ Q_1 \wedge Q_2\ \}}$$ Conjunction

$$\frac{\{\ P_1\ \}\ S\ \{\ Q_1\ \} \qquad \{\ P_2\ \}\ S\ \{\ Q_2\ \}}{\{\ P_1 \vee P_2\ \}\ S\ \{\ Q_1 \vee Q_2\ \}}$$ Disjunction

Conditional

$$\frac{\{\ P \wedge B\ \}\ S_1\ \{\ Q\ \} \qquad \{\ P \wedge \neg B\ \}\ S_2\ \{\ Q\ \}}{\{\ P\ \}\ \textbf{If}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{\ Q\ \}}\ \text{Conditional}$$

$\{\ true\ \}$

$\textbf{If}\ x < y\ \textbf{then}$

$\quad res := y$

$\textbf{else}$

$\quad res := x$

$\textbf{fi}$

$\{res \geq x \wedge res \geq y\}$

Proof Outline

$\{\ true\ \}$

$\textbf{If}\ x < y\ \textbf{then}$

$\quad res := y$

$\textbf{else}$

$\quad res := x$

$\textbf{fi}$

$\{res \geq x \wedge res \geq y\}$

Conditional

$$\frac{\{\ P\wedge B\ \}\ S_1\ \{\ Q\ \} \qquad \{\ P\wedge\neg B\ \}\ S_2\ \{\ Q\ \}}{\{\ P\ \}\ \textbf{If}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{\ Q\ \}}\ \text{Conditional}$$

$\{\ true\ \}$

$\textbf{If}\ x < y\ \textbf{then}$

$res := y$

$\textbf{else}$

$res := x$

$\textbf{fi}$

$\{res \geq x \wedge res \geq y\}$

Proof Outline

$\{\ true\ \}$

$\textbf{If}\ x < y\ \textbf{then}$

$\{x < y\}$

Conditional

$res := y$

$\{res \geq x \wedge res \geq y\}$

$\textbf{else}$

$res := x$

$\textbf{fi}$

$\{res \geq x \wedge res \geq y\}$

Conditional

$$\frac{\{\ P \wedge B\ \}\ S_1\ \{\ Q\ \} \qquad \{\ P \wedge \neg B\ \}\ S_2\ \{\ Q\ \}}{\{\ P\ \}\ \textbf{If}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{\ Q\ \}}\ \text{Conditional}$$

$\{\ true\ \}$

If $x < y$ **then**

$res := y$

else

$res := x$

fi

$\{res \geq x \wedge res \geq y\}$

Proof Outline

$\{\ true\ \}$

If $x < y$ **then**

$\{x < y\}$

$res := y$

$\{res \geq x \wedge res \geq y\}$

else

$\{\neg(x < y)\}$

Conditional

$res := x$

$\{res \geq x \wedge res \geq y\}$

fi

$\{res \geq x \wedge res \geq y\}$

Conditional

$$\frac{\{\ P \wedge B\ \}\ S_1\ \{\ Q\ \} \qquad \{\ P \wedge \neg B\ \}\ S_2\ \{\ Q\ \}}{\{\ P\ \}\ \textbf{If}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{\ Q\ \}}$$ Conditional

Strengthening Precondition

$\{\ true\ \}$

$\textbf{If}\ x < y\ \textbf{then}$

$res := y$

$\textbf{else}$

$res := x$

$\textbf{fi}$

$\{res \geq x \wedge res \geq y\}$

Proof Outline

$\{\ true\ \}$

$\textbf{If}\ x < y\ \textbf{then}$

$\{x < y\}$

$\{y \geq x \wedge y \geq y\}$

$res := y$

$\{res \geq x \wedge res \geq y\}$

$\textbf{else}$

$\{\neg(x < y)\}$

$res := x$

$\{res \geq x \wedge res \geq y\}$

$\textbf{fi}$

$\{res \geq x \wedge res \geq y\}$

Conditional

$$\frac{\{\ P \wedge B\ \}\ S_1\ \{\ Q\ \} \qquad \{\ P \wedge \neg B\ \}\ S_2\ \{\ Q\ \}}{\{\ P\ \}\ \textbf{If}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{\ Q\ \}}\ \text{Conditional}$$

$\{\ true\ \}$

If $x < y$ **then**

$res := y$

else

$res := x$

fi

$\{res \geq x \wedge res \geq y\}$

Strengthening Precondition

Proof Outline

$\{\ true\ \}$

If $x < y$ **then**

$\{x < y\}$

$\{y \geq x \wedge y \geq y\}$

$res := y$

$\{res \geq x \wedge res \geq y\}$

else

$\{\neg(x < y)\}$

$\{x \geq x \wedge x \geq y\}$

$res := x$

$\{res \geq x \wedge res \geq y\}$

fi

$\{res \geq x \wedge res \geq y\}$

Conditional

$$\frac{\{\ P \wedge B\ \}\ S_1\ \{\ Q\ \} \qquad \{\ P \wedge \neg B\ \}\ S_2\ \{\ Q\ \}}{\{\ P\ \}\ \textbf{If}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{\ Q\ \}}\ \text{Conditional}$$

$\{\ true\ \}$

If $x < y$ **then**

$res := y$

else

$res := x$

fi

$\{res \geq x \wedge res \geq y\}$

Proof Outline

$\{\ true\ \}$

If $x < y$ **then**

$\{x < y\}$

$\{y \geq x \wedge y \geq y\}$

Assign $res := y$

$\{res \geq x \wedge res \geq y\}$

else

$\{\neg(x < y)\}$

$\{x \geq x \wedge x \geq y\}$

Assign $res := x$

$\{res \geq x \wedge res \geq y\}$

fi

$\{res \geq x \wedge res \geq y\}$

While

$$\frac{\{\ P \wedge B\ \}\ S\ \{\ P\ \}}{\{\ P\ \}\ \mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{od}\ \{\ P \wedge \neg B\ \}}\ \text{While}$$

- P in the While rule is a *loop invariant*
- Invariant: an assertion that always holds whenever the program reaches it
- Loop invariants are usually specified manually
- For some classes of assertions, loop invariants can be synthesized

While Example

$$\frac{\{\ P \land B\ \}\ S\ \{\ P\ \}}{\{\ P\ \}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\ \{\ P \land \neg B\ \}}\ \text{While}$$

$\{\ x \geq 0 \land y > 0 \land (x \equiv m\ (mod\ y)\}$

While $x \geq y$ **do**

$x := x - y$

od

$\{\ x \geq 0 \land y > 0 \land (x \equiv m\ (mod\ y) \land x < y\}$

$\{\ x \geq 0 \land y > 0 \land (x \equiv m\ (mod\ y)\}$

While $x \geq y$ **do**

$x := x - y$

od

$\{\ x \geq 0 \land y > 0 \land (x \equiv m\ (mod\ y) \land x < y\}$

While

Example

$$\frac{\{\ P \wedge B\ \}\ S\ \{\ P\ \}}{\{\ P\ \}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\ \{\ P \wedge \neg B\ \}}\ \text{While}$$

$\{\ x \geq 0 \wedge y > 0 \wedge (x \equiv m\ (mod\ y)\}$

While $x \geq y$ **do**

$x := x - y$

od

$\{\ x \geq 0 \wedge y > 0 \wedge (x \equiv m\ (mod\ y) \wedge x < y\}$

$\{\ x \geq 0 \wedge y > 0 \wedge (x \equiv m\ (mod\ y)\}$

While $x \geq y$ **do**

$\{\ x \geq 0 \wedge y > 0 \wedge (x \equiv m\ (mod\ y) \wedge x \geq y\}$

While $x := x - y$

$\{\ x \geq 0 \wedge y > 0 \wedge (x \equiv m\ (mod\ y)\}$

od

$\{\ x \geq 0 \wedge y > 0 \wedge (x \equiv m\ (mod\ y) \wedge x < y\}$

While
Example

$$\frac{\{\ P \wedge B\ \}\ S\ \{\ P\ \}}{\{\ P\ \}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\ \{\ P \wedge \neg B\ \}}\ \text{While}$$

$\{\ x \geq 0 \wedge y > 0 \wedge (x \equiv m\ (mod\ y)\}$

While $x \geq y$ **do** Strengthening Precondition

$x := x - y$

od

$\{\ x \geq 0 \wedge y > 0 \wedge (x \equiv m\ (mod\ y) \wedge x < y\}$

$\{\ x \geq 0 \wedge y > 0 \wedge (x \equiv m\ (mod\ y)\}$

While $x \geq y$ **do**

$\{\ x \geq 0 \wedge y > 0 \wedge (x \equiv m\ (mod\ y) \wedge x \geq y\}$

$\{\ x - y \geq 0 \wedge y > 0 \wedge (x - y \equiv m\ (mod\ y)\}$

$x := x - y$

$\{\ x \geq 0 \wedge y > 0 \wedge (x \equiv m\ (mod\ y)\}$

od

$\{\ x \geq 0 \wedge y > 0 \wedge (x \equiv m\ (mod\ y) \wedge x < y\}$

While

Example

$$\frac{\{\ P \wedge B\ \}\ S\ \{\ P\ \}}{\{\ P\ \}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\ \{\ P \wedge \neg B\ \}}\ \text{While}$$

$\{\ x \geq 0 \wedge y > 0 \wedge (x \equiv m\ (mod\ y)\}$

While $x \geq y$ **do**

$x := x$ - y

od

$\{\ x \geq 0 \wedge y > 0 \wedge (x \equiv m\ (mod\ y) \wedge x < y\}$

$\{\ x \geq 0 \wedge y > 0 \wedge (x \equiv m\ (mod\ y)\}$

While $x \geq y$ **do**

$\{\ x \geq 0 \wedge y > 0 \wedge (x \equiv m\ (mod\ y) \wedge x \geq y\}$

$\{\ x\text{-}y \geq 0 \wedge y > 0 \wedge (x\text{-}y \equiv m\ (mod\ y)\}$

Assign $x := x$ - y

$\{\ x \geq 0 \wedge y > 0 \wedge (x \equiv m\ (mod\ y)\}$

od

$\{\ x \geq 0 \wedge y > 0 \wedge (x \equiv m\ (mod\ y) \wedge x < y\}$

While
Total Correctness

- For total correctness, loops must terminate
- How to ensure this in annotations?
  - specify a rank function that decreases after every loop body

$$\frac{\{\ P \wedge B\ \}\ S\ \{\ P\ \} \qquad \{\ P \wedge B \wedge t = Z\ \}\ S\ \{\ t < Z\ \} \qquad P \wedge B \rightarrow t \geq 0}{\{\ P\ \}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\ \{\ P \wedge \neg B\ \}}\ \text{While Total}$$

t is a rank function

Rank Function Example

- What is the rank function?

$\{\ x \geq 0 \wedge y > 0 \wedge (x \equiv m\ (mod\ y)\}$

While $x \geq y$ **do**

  $x := x - y$

od

$\{\ x \geq 0 \wedge y > 0 \wedge (x \equiv m\ (mod\ y) \wedge x < y\}$

Functions

$$\textbf{fun } p(\textbf{in } x, \textbf{ inout } y, \textbf{ out } z);\ S;$$

- p is the name of the function
- x is a sequence of input variables, y is a sequence of input and output variables, and z is a sequence of output variables
- S is the function body
- Assume there is not global variables
- Functions are call-by-value

Non-recursive Functions Inference Rule

$$\frac{\{\ P\ \}\ S\ \{\ Q\ \}}{\{\ P[a,b/x,y] \wedge I\ \}\ p(a,b,c)\ \{\ Q[b,c/y,z] \wedge I\ \}}\ \textsf{Fun}$$

where p is a function **fun** p(**in** x, **inout** y, **out** z); S; and I does not refer to variables changed by p

Recursive Functions Inference Rule

$$\frac{\forall s,t,u.\ \{\ P[s,t/x,y]\ \}\ p(s,t,u)\ \{\ Q[t,u/y,z]\ \} \vdash \{\ P\ \}\ S\ \{\ Q\ \}}{\{\ P[a,b/x,y] \wedge I\ \}\ p(a,b,c)\ \{\ Q[b,c/y,z] \wedge I\ \}}\ \textsf{Rec}$$

where p is a function **fun** p(**in** x, **inout** y, **out** z); S; and I does not refer to variables changed by p

Exercise

- Complete the proof outline.

$\{x \geq 0 \wedge y \geq 0 \wedge gcd(x, y) = gcd(m, n)\}$

while $\mathrm{x} \neq 0 \wedge \mathrm{y} \neq 0$ **do**

  if $\mathrm{x} < \mathrm{y}$ **then**

    $\mathrm{x}, \mathrm{y} := \mathrm{y}, \mathrm{x}$

  fi;

  $\mathrm{x} := \mathrm{x} - \mathrm{y}$

od

$\{(x = 0 \wedge y \geq 0 \wedge y = gcd(x, y) = gcd(m, n)) \vee$

  $(x \geq 0 \wedge y = 0 \wedge x = gcd(x, y) = gcd(m, \mathrm{n}))\}$

Weakest Precondition

- Weakest precondition: the weakest precondition that guarantees termination of the program in a state satisfying the postcondition
- $wp(S,\ Q)$ is the weakest precondition of a program S and a postcondition Q
- $wp(S,\ \cdot)$ is a predicate transformer that transforms a postcondition to a weakest precondition
- $wp(S,\ \cdot)$ can be seen as the semantics of S

Hoare Triple as wp

- When total correctness is meant, $\{P\}\ S\ \{Q\}$ is another notation for $P \Rightarrow wp(S,\ Q)$

  - $P \Rightarrow wp(S,\ Q)$: P entails $wp(S,\ Q)$

Properties of wp

- Axioms:
  - Law of the Excluded Miracle: $wp(S, \mathit{false}) \equiv \mathit{false}$
  - Distributivity of Conjunction: $wp(S, Q_1) \wedge wp(S, Q_2) \equiv wp(S, Q_1 \wedge Q_2)$
  - Distributivity of Disjunction for deterministic S: $wp(S, Q_1) \vee wp(S, Q_2) \equiv wp(S, Q_1 \vee Q_2)$
- Derived:
  - Law of Monotonicity: if $Q_1 \Rightarrow Q_2$, then $wp(S, Q_1) \Rightarrow wp(S, Q_2)$
  - Distributivity of Disjunction for nondeterministic S: $wp(S, Q_1) \vee wp(S, Q_2) \equiv wp(S, Q_1 \vee Q_2)$

Some Laws for Predicate Calculation

- $A \leftrightarrow B \equiv B \leftrightarrow A$
- $A \leftrightarrow (B \leftrightarrow C) \equiv (A \leftrightarrow B) \leftrightarrow C$
- $false \vee A \equiv A \vee false \equiv A$
- $\neg A \wedge A \equiv false$
- $A \rightarrow B \equiv \neg A \vee B$
- $A \rightarrow false \equiv \neg A$
- $(A \vee B) \rightarrow C \equiv$
  $(A \rightarrow C) \wedge (B \rightarrow C)$
- $A \rightarrow (B \rightarrow C) \equiv (A \wedge B) \rightarrow C$
- $A \rightarrow B \equiv A \leftrightarrow (A \wedge B)$
- $A \wedge B \Rightarrow A$

Some Laws for Predicate Calculation (cont'd)

- $\forall x (x = e \to A) \equiv A[e/x] \equiv \exists x (x = e \wedge A)$, if x is not free in e
- $\forall x (A \wedge B) \equiv \forall x A \wedge \forall x B$
- $\forall x (A \to B) \Rightarrow \forall x A \to \forall x B$
- $\forall x (A \to B) \equiv A \to \forall x B$, if x is not free in A
- $\exists x (A \wedge B) \equiv A \wedge \exists x B$, if x is not free in A

wp: Skip and Abort

- $wp(\mathbf{skip}\ ,\ Q) = Q$
- $wp(\mathbf{abort},\ Q) = \mathit{false}$

wp: Assignment and Sequence

- $wp(x := e,\ Q) = Q[e/x]$
- $wp(S_1;\ S_2,\ Q) = wp(S_1,\ wp(S_2,\ Q))$

Example

$$wp(x := x \text{ - } 5;\ x := x * 2,\ x > 20)$$

$$= wp(x := x \text{ - } 5,\ wp(x := x * 2,\ x > 20))$$

$$= wp(x := x \text{ - } 5,\ x * 2 > 20)$$

$$= (x \text{ - } 5) * 2 > 20$$

$$= x > 15$$

wp: Conditional

- $wp(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi},\ Q) = (B \rightarrow wp(S_1,\ Q)) \wedge (\neg B \rightarrow wp(S_2,\ Q))$

- $wp(\textbf{if } B \textbf{ then } S \textbf{ fi},\ Q) = (B \rightarrow wp(S,\ Q)) \wedge (\neg B \rightarrow Q)$

Example

$wp(\textbf{if } x < y \textbf{ then } x := y \textbf{ fi}, x \geq y)$

$= (x < y \rightarrow wp(x := y, x \geq y)) \wedge (\neg(x < y) \rightarrow x \geq y)$

$= (x < y \rightarrow y \geq y) \wedge (\neg(x < y) \rightarrow x \geq y)$

$\Leftrightarrow true$

wp: While

- **while** B **do** S **od** is equivalent to
  - **if** B **then** (S; **if** B **then** (S; **if** B **then** (...) **fi**) **fi**) **fi**
- Thus, $wp(\textbf{while } B \textbf{ do } S \textbf{ od}, \mathrm{Q}) = (\neg B \rightarrow Q) \wedge (B \rightarrow wp(S, (\neg B \rightarrow Q) \wedge (B \rightarrow wp(S, ...))))$
- Define
  - $H_0(Q) \triangleq \neg B \rightarrow Q$
  - $H_k(Q) \triangleq wp(S, H_{k\text{-}1}(Q))$
- $wp(\textbf{while } B \textbf{ do } S \textbf{ od}, \mathrm{Q}) = \exists \mathrm{k}.\ 0 \leq \mathrm{k} \wedge H_k(Q)$

wp: Theorem for While

- Suppose there exist a predicate P and an integer-valued expression t such that
  - $P \wedge B \Rightarrow wp(S,\ P)$,
  - $P \Rightarrow (t \geq 0)$, and
  - $P \wedge B \wedge (t = t_0) \Rightarrow wp(S,\ t < t_0)$, where t_0 is a rigid variable.
- Then, $P \Rightarrow wp(\mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{od},\ P \wedge \neg B)$

Verification Condition Generation

$\{\ P\ \}$

S_1

$\{\ R\ \}$

Verification Condition:

S_2

S_3

$\{\ Q\ \}$

Verification Condition Generation

$\{ \ P \ \}$

S_1

$\{ \ R \ \}$

Verification Condition:

S_2

$\{ \ wp(S_3, \ Q) \ \}$

S_3

$\{ \ Q \ \}$

Verification Condition Generation

$\{\ P\ \}$

S_1

$\{\ R\ \}$

$\{\ wp(S_2,\ wp(S_3,\ Q))\ \}$

S_2

$\{\ wp(S_3,\ Q)\ \}$

S_3

$\{\ Q\ \}$

Verification Condition:

Verification Condition Generation

$\{\ P\ \}$

S_1

$\{\ R\ \}$

$\{\ wp(S_2,\ wp(S_3,\ Q))\ \}$

S_2

$\{\ wp(S_3,\ Q)\ \}$

S_3

$\{\ Q\ \}$

Verification Condition:

1. $R \rightarrow wp(S_2,\ wp(S_3,\ Q))$

Verification Condition Generation

$\{\ P\ \}$

$\{\ wp(S_1,\ R)\ \}$

S_1

$\{\ R\ \}$

$\{\ wp(S_2,\ wp(S_3,\ Q))\ \}$

S_2

$\{\ wp(S_3,\ Q)\ \}$

S_3

$\{\ Q\ \}$

Verification Condition:

1. $R \rightarrow wp(S_2,\ wp(S_3,\ Q))$

Verification Condition Generation

$\{\ P\ \}$

$\{\ wp(S_1,\ R)\ \}$

S_1

$\{\ R\ \}$

$\{\ wp(S_2,\ wp(S_3,\ Q))\ \}$

S_2

$\{\ wp(S_3,\ Q)\ \}$

S_3

$\{\ Q\ \}$

Verification Condition:

1. $R \rightarrow wp(S_2,\ wp(S_3,\ Q))$
2. $P \rightarrow wp(S_1,\ R)$

Verification Condition Generation

$\{\ P\ \}$

S_1

$\{\ R\ \}$

Verification Condition:

S_2

S_3

$\{\ Q\ \}$

Verification Condition Generation

$\{\ P\ \}$

S_1

$\{\ R\ \}$

Verification Condition:

S_2

$\{\ wp(S_3,\ Q)\ \}$

S_3

$\{\ Q\ \}$

Verification Condition Generation

$\{ \ P \ \}$

S_1

$\{ \ R \ \}$

$\{ \ wp(S_2, \ wp(S_3, \ Q)) \ \}$

S_2

$\{ \ wp(S_3, \ Q) \ \}$

S_3

$\{ \ Q \ \}$

Verification Condition:

Verification Condition Generation

$\{ P \}$

S_1

$\{ R \}$

$\{ wp(S_2, wp(S_3, Q)) \}$

S_2

$\{ wp(S_3, Q) \}$

S_3

$\{ Q \}$

Verification Condition:

1. $P \rightarrow wp(S_1, R \wedge wp(S_2, wp(S_3, Q)))$

Verification Condition Generation

$\{\ P\ \}$

$\{\ wp(S_1,\ R \wedge wp(S_2,\ wp(S_3,\ Q)))\ \}$

S_1

$\{\ R\ \}$

$\{\ wp(S_2,\ wp(S_3,\ Q))\ \}$

S_2

$\{\ wp(S_3,\ Q)\ \}$

S_3

$\{\ Q\ \}$

Verification Condition:

1. $P \rightarrow wp(S_1,\ R \wedge wp(S_2,\ wp(S_3,\ Q)))$

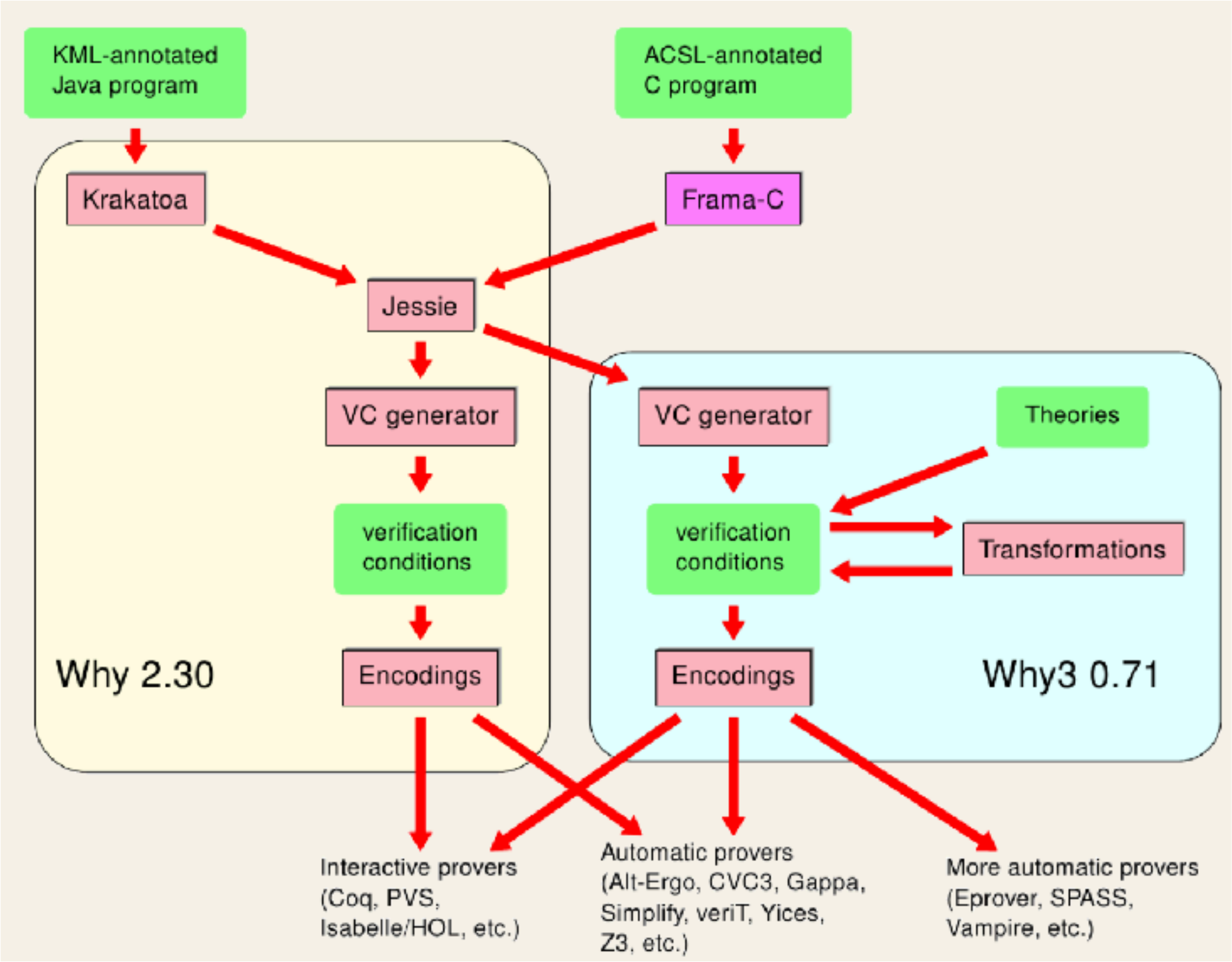
Exercise

- Compute $wp(x := x{+}2;\ y := y\text{-}2,\ x{+}y{=}0)$
- Compute $wp(\textbf{If}\ x < y\ \textbf{then}\ res := y\ \textbf{else}\ res := x\ \textbf{fi},\ res \geq x \wedge res \geq y)$

Frama-C

- Frama-C is an extensible and collaborative platform dedicated to source-code analysis of C software
- Available on http://frama-c.com
- Various plugins
  - Value analysis
  - Weakest precondition computation
  - Verification

Frama-C + Jessie + Why



ACSL

- ACSL is an acronym for "ANSI/ISO C Specification Language"

- A Behavioral Interface Specification Language (BISL) implemented in the Frama-C framework

- Inspired from the Caduceus tool, of which its language is inspired from the Java Modeling Language (JML)

Jessie

- a plugin for the Frama-C environment, aimed at performing deductive verification of C programs, annotated using the ACSL language, using the Why tool for generating proof obligations

Why

- A software verification platform containing
  - a general-purpose verification condition generator
  - a tool Krakatoa for the verification of Java programs
  - a tool Caduceus for the verification of C programs (obsolete)
- Why is integrated with many provers
- Why language is closed to OCaml

ACSL: rquires/ensures/result

```
/*@ requires x >= 0;
  @ ensures \result >= 0;
  @ ensures \result * \result <= x;
  @ ensures x < (\result + 1) * (\result + 1); @*/
int isqrt(int x);
```

requires: specify preconditions

ensures: specify postconditions

\result: the returned value

ACSL: valid/assigns/old

```
/*@ requires \valid (p);
  @ assigns *p;
  @ ensures *p == \old(*p) + 1; @*/
void incrstar(int *p);
```

valid: deferencing the pointer will produce a definite value according to the C standard
assigns: specify the set of modified memory locations
\old: the value before function call

ACSL: logic specifications

```
//@ predicate is_positive(integer x) = x > 0;
/*@ logic integer get_sign(real x) =
  @                x>0.0?1:(x<0.0?-1:0);
  @*/
```

```
//@ lemma mean_property: \forall integer x,y; x <= y ==> x <= (x+y)/2 <= y;
```

```
/*@ inductive is_gcd(integer a, integer b, integer d) {
  @   case gcd_zero:
  @     \forall integer n; is_gcd(n,0,n);
  @   case gcd_succ:
  @     \forall integer a,b,d; is_gcd(b, a % b, d) ==> is_gcd(a,b,d);
  @ }
  @*/
```

ACSL: logic specifications (cont'd)

```
/*@ axiomatic IntList {
  @   type int_list;
  @   logic int_list nil;
  @   logic int_list cons(integer n,int_list l);
  @   logic int_list append(int_list l1,int_list l2);
  @   axiom append_nil:
  @     \forall int_list l; append(nil,l) == l;
  @   axiom append_cons:
  @     \forall integer n, int_list l1,l2;
  @     append(cons(n,l1),l2) == cons(n,append(l1,l2));
  @ }
  @*/
```

ACSL: invariants/variants

```
int bsearch(double t[], int n, double v) {
  int l=0,u=n-1;
  /*@ loop invariant 0 <= l && u <= n-1;
    @ for failure: loop invariant
    @ \forall integer k;0<=k<n&&t[k]==v==>l<=k<=u; @*/
  while (l<=u){
    int m = l + (u-l)/2; // better than (l+u)/2 if (t[m]<v)l=m+1;
    else if (t[m]>v)u=m-1;
    else return m;
  }
  return -1;
}
```

```
void f(int x) {
  //@ loop variant x;
  while (x >= 0) {
    x -= 2;
  }
}
```

Forward Reasoning

$$\frac{}{\{\ Q\ \}\ x := e\ \{\ \exists\ y.\ Q[y/x] \wedge x = e[y/x]\ \}}\ y \text{ free in } Q$$

$e[y/x]$: replace x in e with y

$$\{\ x > 0\ \}\ x := x + 1\ \{\ \exists\ z.\ z > 0 \wedge x = z + 1\ \}$$

$$\{\ x > 0\ \}\ x := x \text{ - } 1\ \{\ \exists\ z.\ z > 0 \wedge x = z \text{ - } 1\ \}$$

$$\{\ x = y\ \}\ x := x + y\ \{\ \exists\ z.\ z = y \wedge x = z + y\ \}$$

Strongest Postcondition

- $sp(S,\ Q)$: the strongest postcondition of program S and precondition Q
  - $sp(\text{SKIP},\ Q) = Q$
  - $sp(x := e,\ Q) = \exists\ y.\ Q[y/x] \land x = e[y/x]$
  - $sp(S_1;\ S_2,\ Q) = sp(S_2,\ sp(S_1,\ Q))$
  - $sp(\textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi},\ Q) = sp(S_1,\ Q \land B) \lor sp(S_2,\ Q \land \neg B)$
  - $sp(\textbf{while}\ B\ \textbf{do}\ S\ \textbf{od},\ \text{Q}) = sp(\textbf{while}\ B\ \textbf{do}\ S\ \textbf{od},\ sp(S,\ Q \land B)) \lor (Q \land \neg B)$
- Can we avoid quantifications in forward reasoning?

Symbolic Execution

- Assume an initial symbolic value for each variable
- Execute the program with the symbolic states (formulas describing what the symbolic values are)

Symbolic Execution Example

$x = x_0,\ y = y_0$

```
int main(void) {
  int x, y;
  while(x < y)
    x = x + 1;
  assert(x == y);
}
```

Symbolic Execution Example

$x = x_0,\ y = y_0$

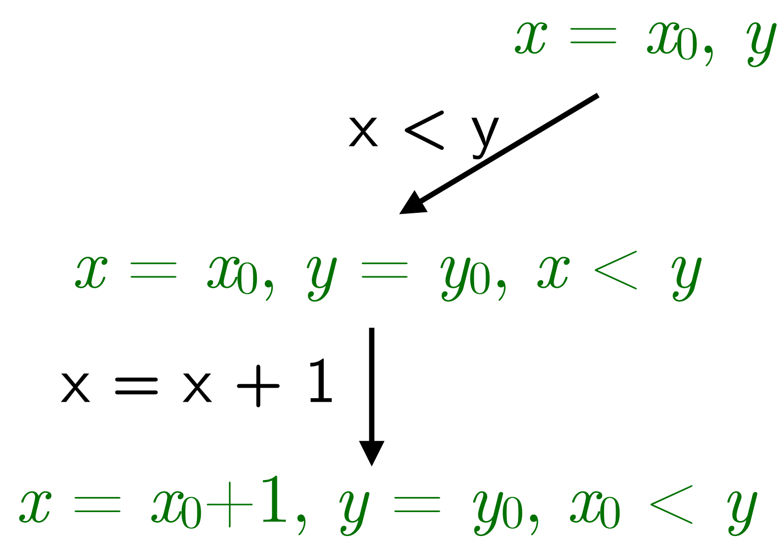
x < y

$x = x_0,\ y = y_0,\ x < y$

```
int main(void) {
  int x, y;
  while(x < y)
    x = x + 1;
  assert(x == y);
}
```

Symbolic Execution Example

```c
int main(void) {
  int x, y;
  while(x < y)
    x = x + 1;
  assert(x == y);
}
```



Symbolic Execution Example

```
int main(void) {
  int x, y;
  while(x < y)
    x = x + 1;
  assert(x == y);
}
```

$$x = x_0,\ y = y_0$$

x < y

$$x = x_0,\ y = y_0,\ x < y$$

x = x + 1

$$x = x_0{+}1,\ y = y_0,\ x_0 < y$$

x < y

$$x = x_0{+}1,\ y = y_0,\ x_0 < y,\ x < y$$

Symbolic Execution Example

```c
int main(void) {
  int x, y;
  while(x < y)
    x = x + 1;
  assert(x == y);
}
```

$x = x_0,\ y = y_0$

↓ x < y

$x = x_0,\ y = y_0,\ x < y$

↓ x = x + 1

$x = x_0{+}1,\ y = y_0,\ x_0 < y$

↓ x < y

$x = x_0{+}1,\ y = y_0,\ x_0 < y,\ x < y$

↓ x = x + 1

$x = x_0{+}2,\ y = y_0,\ x_0 < y,\ x_0{+}1 < y$

Symbolic Execution Example

```c
int main(void) {
  int x, y;
  while(x < y)
    x = x + 1;
  assert(x == y);
}
```

$x = x_0,\ y = y_0$

x < y ↓

$x = x_0,\ y = y_0,\ x < y$

x = x + 1 ↓

$x = x_0{+}1,\ y = y_0,\ x_0 < y$

x < y ↓

$x = x_0{+}1,\ y = y_0,\ x_0 < y,\ x < y$

x = x + 1 ↓

$x = x_0{+}2,\ y = y_0,\ x_0 < y,\ x_0{+}1 < y$

x < y ↓

⋮

Widening

- Symbolic execution may not terminate.
- We need to forget some details.
  - Widening
- If symbolic execution reaches a symbolic state P and we can prove $P \Rightarrow Q$, then symbolic execution can continue with Q.

Widening Example

$x = x_0,\ y = y_0$

```
int main(void) {
  int x, y;
  while(x < y)
    x = x + 1;
  assert(x == y);
}
```

x_0, y_0, and x_1 (logical variables) are implicitly quantified by ∃

Widening Example

$x = x_0,\ y = y_0$

x < y

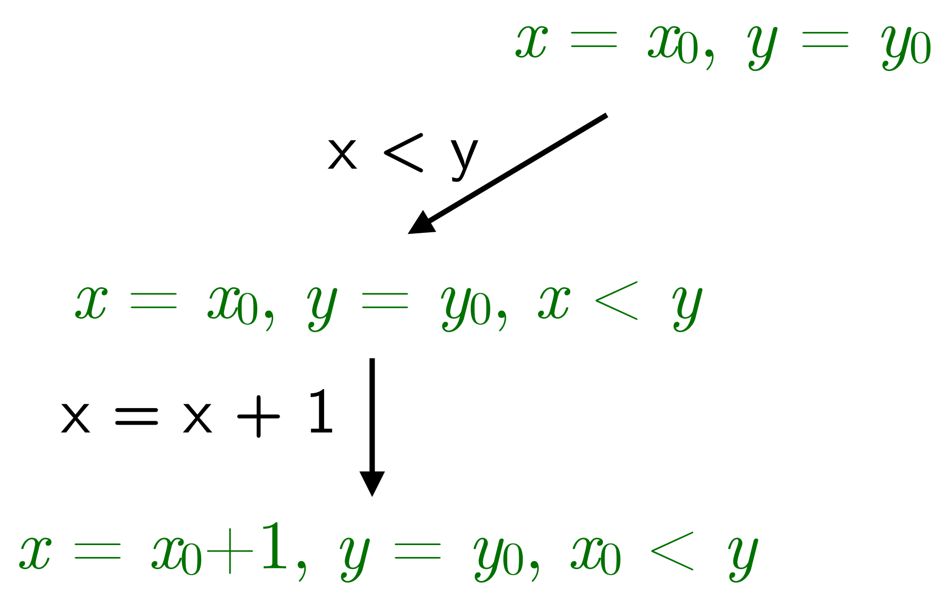
$x = x_0,\ y = y_0,\ x < y$

```
int main(void) {
  int x, y;
  while(x < y)
    x = x + 1;
  assert(x == y);
}
```

x_0, y_0, and x_1 (logical variables) are implicitly quantified by ∃

Widening Example

```c
int main(void) {
  int x, y;
  while(x < y)
    x = x + 1;
  assert(x == y);
}
```



x_0, y_0, and x_1 (logical variables) are implicitly quantified by ∃

Widening Example

```
int main(void) {
  int x, y;
  while(x < y)
    x = x + 1;
  assert(x == y);
}
```

$x = x_0,\ y = y_0$

x < y

$x = x_0,\ y = y_0,\ x < y$

x = x + 1

$x = x_0{+}1,\ y = y_0,\ x_0 < y$

$x = x_1,\ y = y_0,\ x_1 \leqq y$

x_0, y_0, and x_1 (logical variables) are implicitly quantified by $\exists$

Widening Example

```
int main(void) {
  int x, y;
  while(x < y)
    x = x + 1;
  assert(x == y);
}
```

$x = x_0,\ y = y_0$

x < y ↙

$x = x_0,\ y = y_0,\ x < y$

x = x + 1 ↓

$x = x_0{+}1,\ y = y_0,\ x_0 < y$

$x = x_1,\ y = y_0,\ x_1 \leqq y$

x < y ↙

$x = x_1,\ y = y_0,\ x_1 \leqq y,\ x < y$

x_0, y_0, and x_1 (logical variables) are implicitly quantified by ∃

Widening Example

```c
int main(void) {
  int x, y;
  while(x < y)
    x = x + 1;
  assert(x == y);
}
```

$x = x_0,\ y = y_0$

↓ x < y

$x = x_0,\ y = y_0,\ x < y$

↓ x = x + 1

$x = x_0{+}1,\ y = y_0,\ x_0 < y$

$x = x_1,\ y = y_0,\ x_1 \leqq y$

↓ x < y

$x = x_1,\ y = y_0,\ x_1 \leqq y,\ x < y$

↓ x = x + 1

$x = x_1{+}1,\ y = y_0,\ x_1 \leqq y,\ x_1 < y$

x_0, y_0, and x_1 (logical variables) are implicitly quantified by $\exists$

Widening Example

```
int main(void) {
  int x, y;
  while(x < y)
    x = x + 1;
  assert(x == y);
}
```

$x = x_0,\ y = y_0$

x < y ↓

$x = x_0,\ y = y_0,\ x < y$

x = x + 1 ↓

$x = x_0{+}1,\ y = y_0,\ x_0 < y$

$x = x_1,\ y = y_0,\ x_1 \leqq y$

x < y ↓

$x = x_1,\ y = y_0,\ x_1 \leqq y,\ x < y$

x = x + 1 ↓

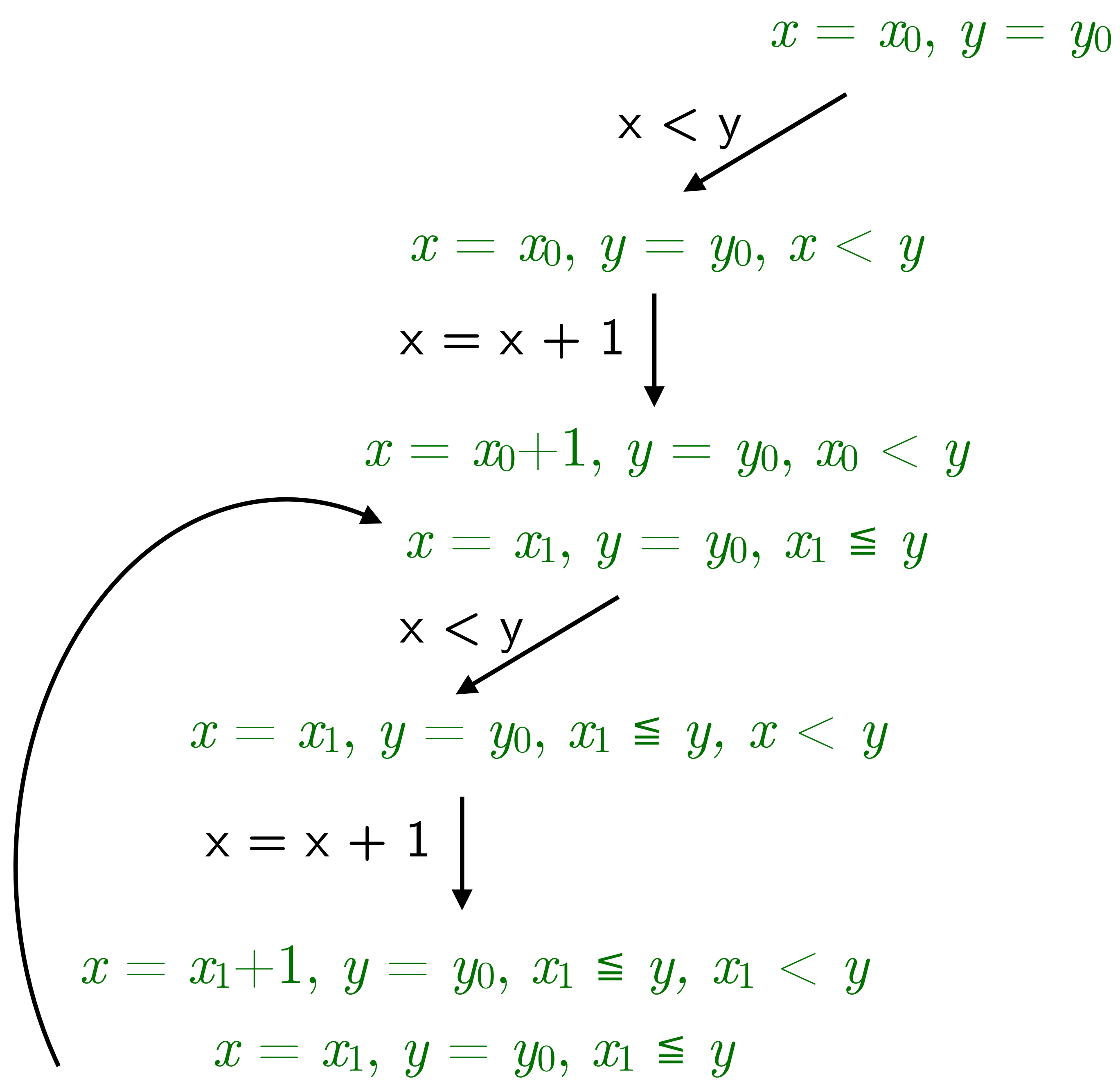
$x = x_1{+}1,\ y = y_0,\ x_1 \leqq y,\ x_1 < y$

$x = x_1,\ y = y_0,\ x_1 \leqq y$

x_0, y_0, and x_1 (logical variables) are implicitly quantified by $\exists$

Widening Example

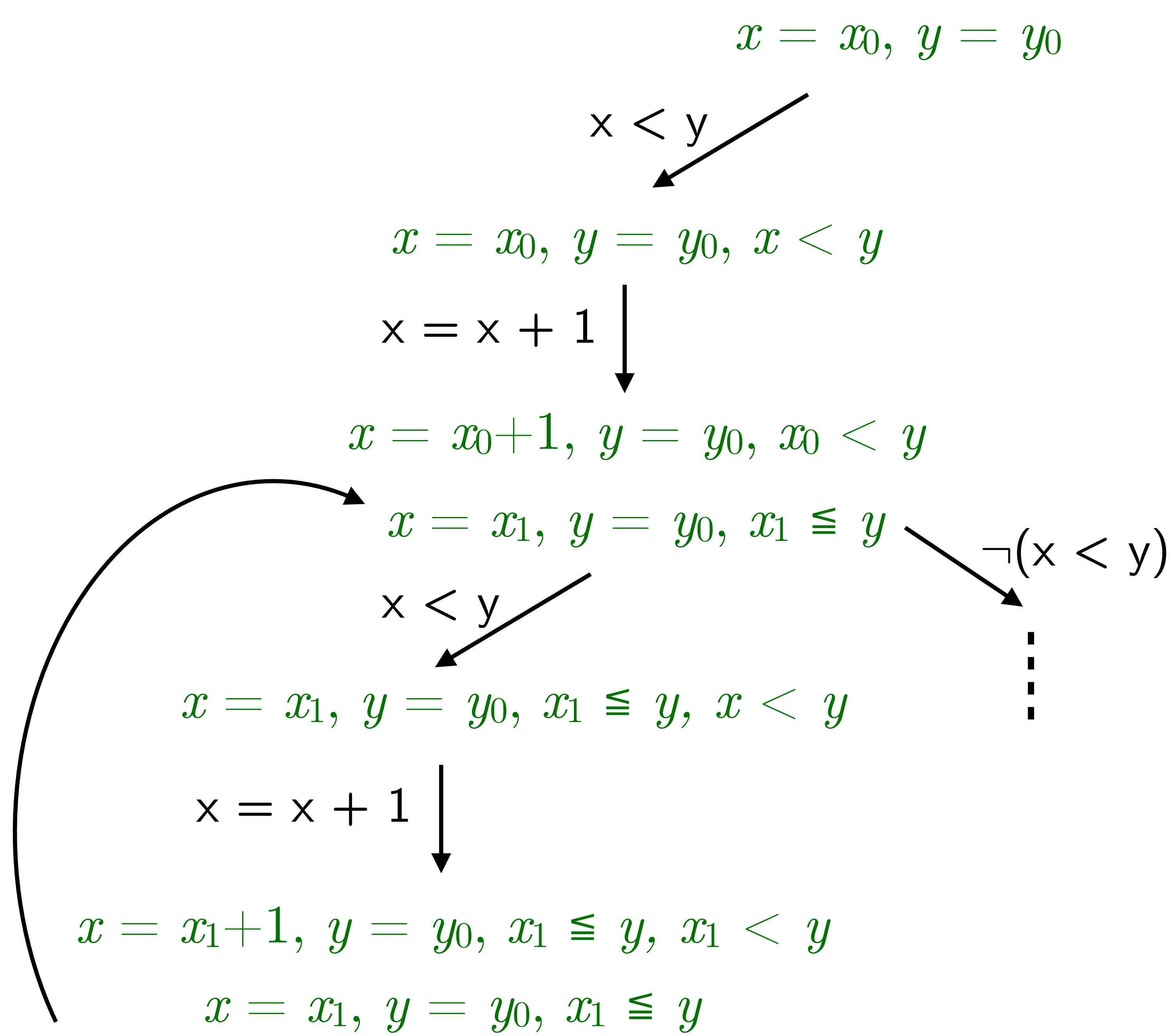
```c
int main(void) {
  int x, y;
  while(x < y)
    x = x + 1;
  assert(x == y);
}
```



x_0, y_0, and x_1 (logical variables) are implicitly quantified by ∃

Widening Example

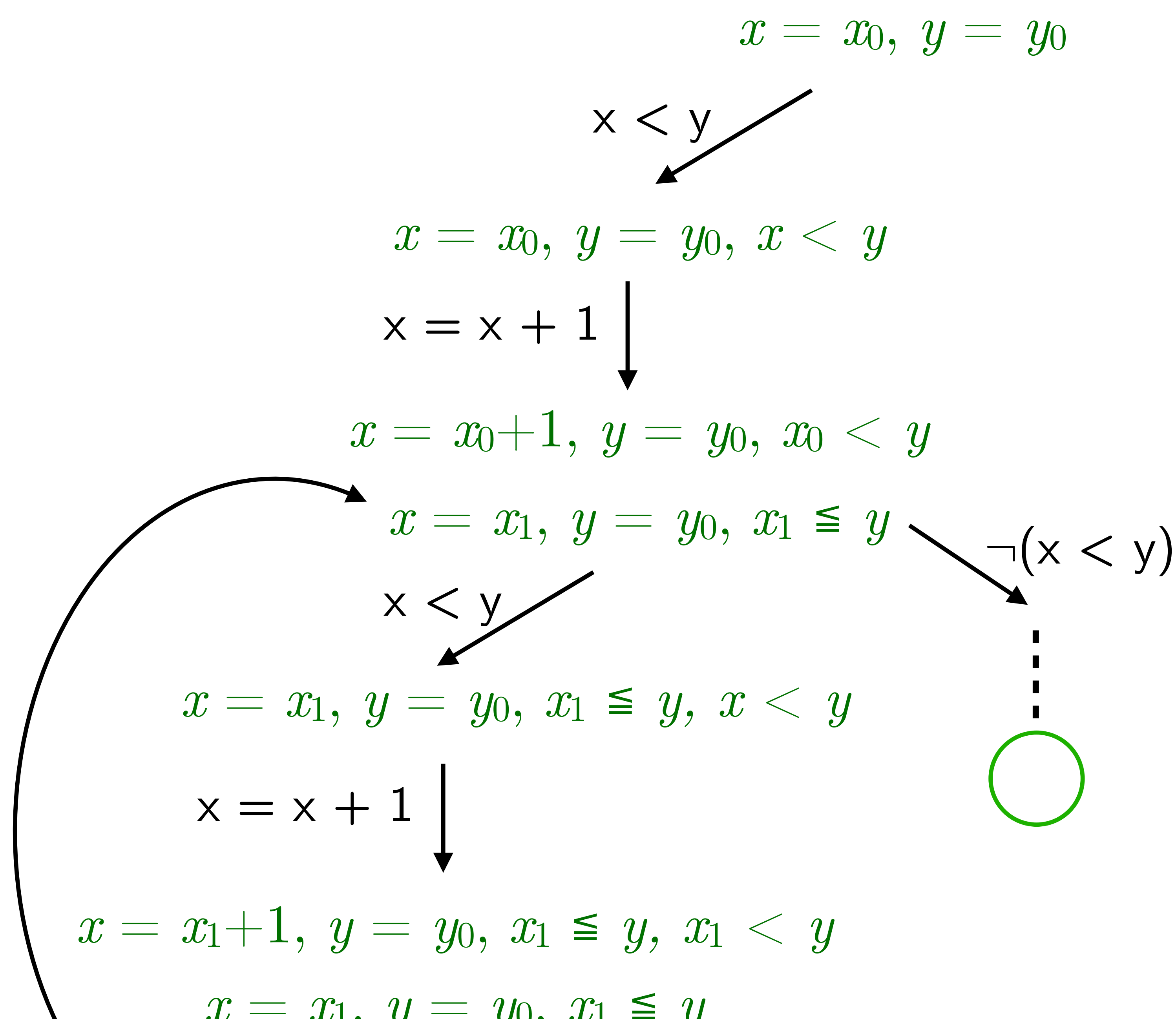
```
int main(void) {
  int x, y;
  while(x < y)
    x = x + 1;
  assert(x == y);
}
```



x_0, y_0, and x_1 (logical variables) are implicitly quantified by $\exists$

Widening Example

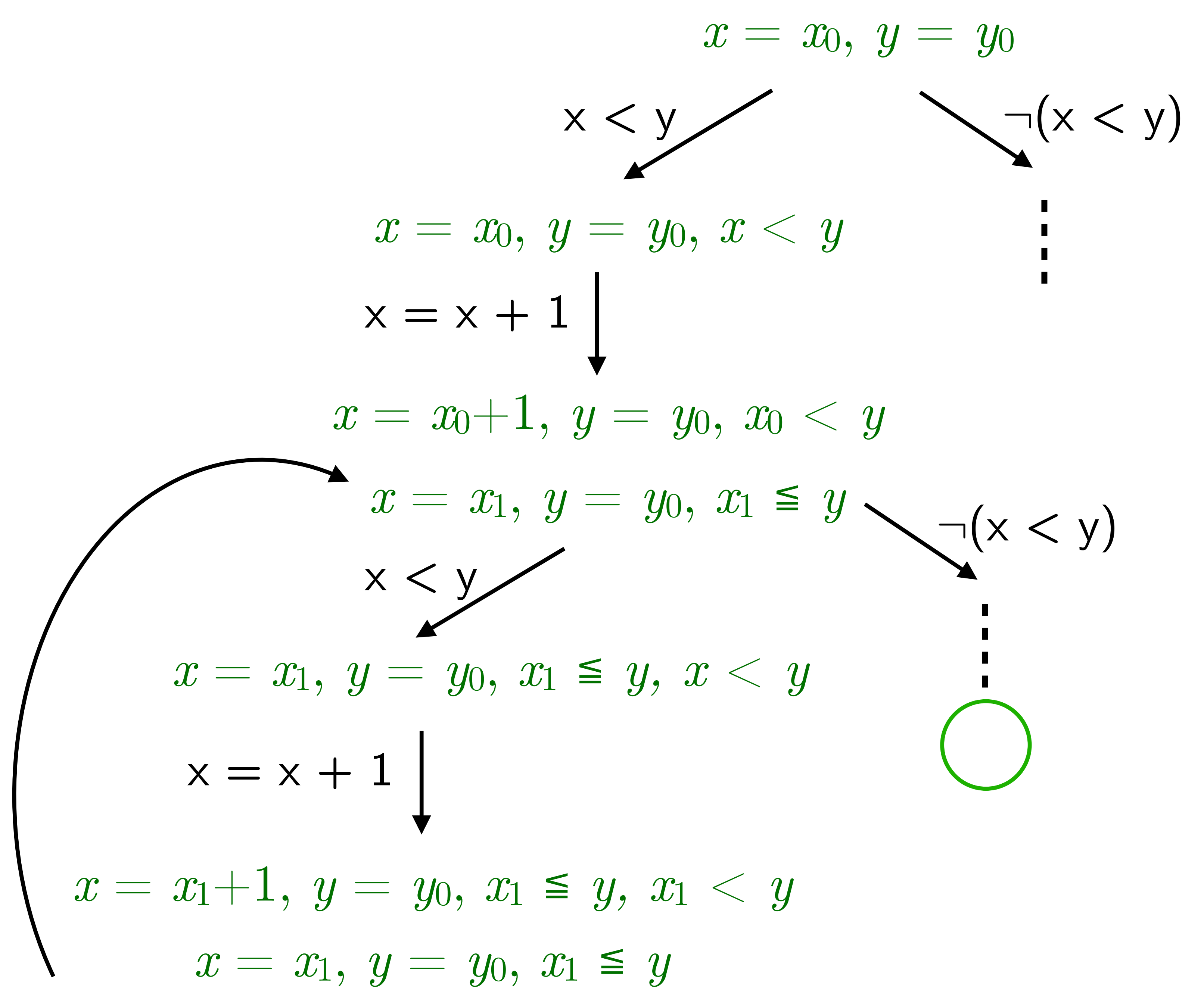
```
int main(void) {
  int x, y;
  while(x < y)
    x = x + 1;
  assert(x == y);
}
```



x_0, y_0, and x_1 (logical variables) are implicitly quantified by ∃

Widening Example

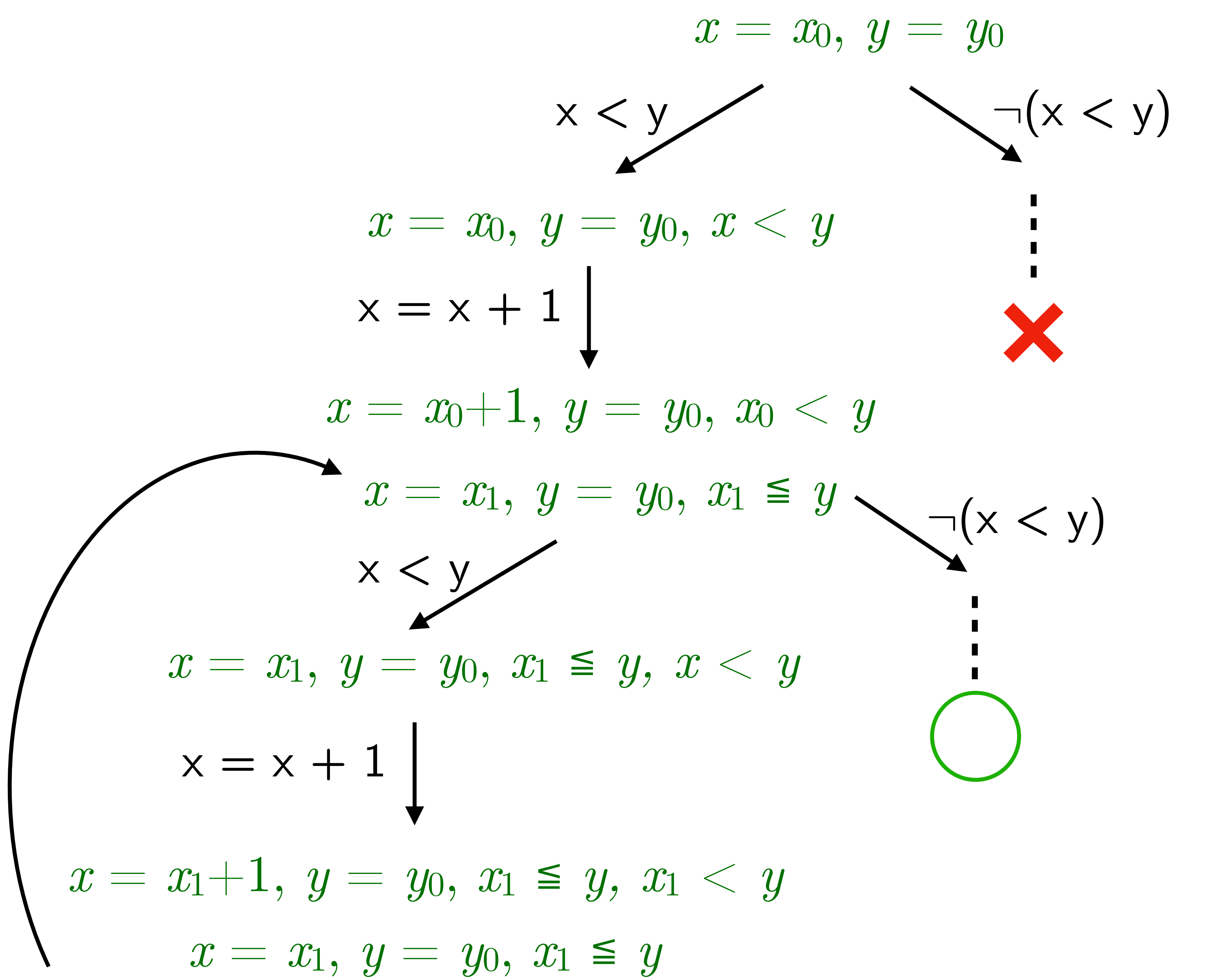
```
int main(void) {
  int x, y;
  while(x < y)
    x = x + 1;
  assert(x == y);
}
```



x_0, y_0, and x_1 (logical variables) are implicitly quantified by ∃

Widening Example

```
int main(void) {
  int x, y;
  while(x < y)
    x = x + 1;
  assert(x == y);
}
```



x_0, y_0, and x_1 (logical variables) are implicitly quantified by ∃

Widening Example (cont'd)

Widening

$x = x_0{+}1,\ y = y_0,\ x_0 < y \longrightarrow x = x_1,\ y = y_0,\ x_1 \leqq y$

$(\exists x_0.\exists y_0.x = x_0{+}1,\ y = y_0,\ x_0 < y) \Rightarrow (\exists x_1.\exists y_0.x = x_1,\ y = y_0,\ x_1 \leqq y)$

Widening

$x = x_0{+}1,\ y = y_0,\ x_0 < y \longrightarrow x \leqq y$

$(\exists x_0.\exists y_0.x = x_0{+}1,\ y = y_0,\ x_0 < y) \Rightarrow (x \leqq y)$

Both widening results allow us to find assertion violation

In our example, logical variables (variables not occurring in the program) are implicitly quantified by ∃

Exercise

$list(n, x, y)$: x points to a list ended at y with length n

- Assume $n \geq 0$ and
  - $list(0, x, x)$ for all x
  - $list(0, x, z) \rightarrow x = z$
  - $x = cons(a, b) \wedge list(n, b, z) \leftrightarrow list(n+1, x, z)$
  - $list(n, x, z) \wedge y = del(x) \wedge n > 0 \rightarrow list(n\text{-}1, y, z)$
  - $list(n, x, z) \wedge n > 0 \rightarrow x \neq nil$
- Either show that the assertion won't be violated or find a counterexample that violates the assertion.

```
x = nil;
i = 0;
while(i < n) {
  x = cons(i, x);
  i = i + 1;
}
j = 0
while(j < n) {
  assert(x != nil)
  x = del(x);
  j = j + 1;
}
```

Tools

- Various tools implementing different algorithms
  - AProVE (http://aprove.informatik.rwth-aachen.de)
  - CPAchecker (https://cpachecker.sosy-lab.org)
  - CBMC (http://www.cprover.org/cbmc/)
  - JavaPathFinder (http://javapathfinder.sourceforge.net)
  - SMACK (http://smackers.github.io)
  - Ultimate Automizer (https://monteverdi.informatik.uni-freiburg.de/tomcat/Website/?ui=tool&tool=automizer)
  - ...
- SV-COMP: competition on software verification