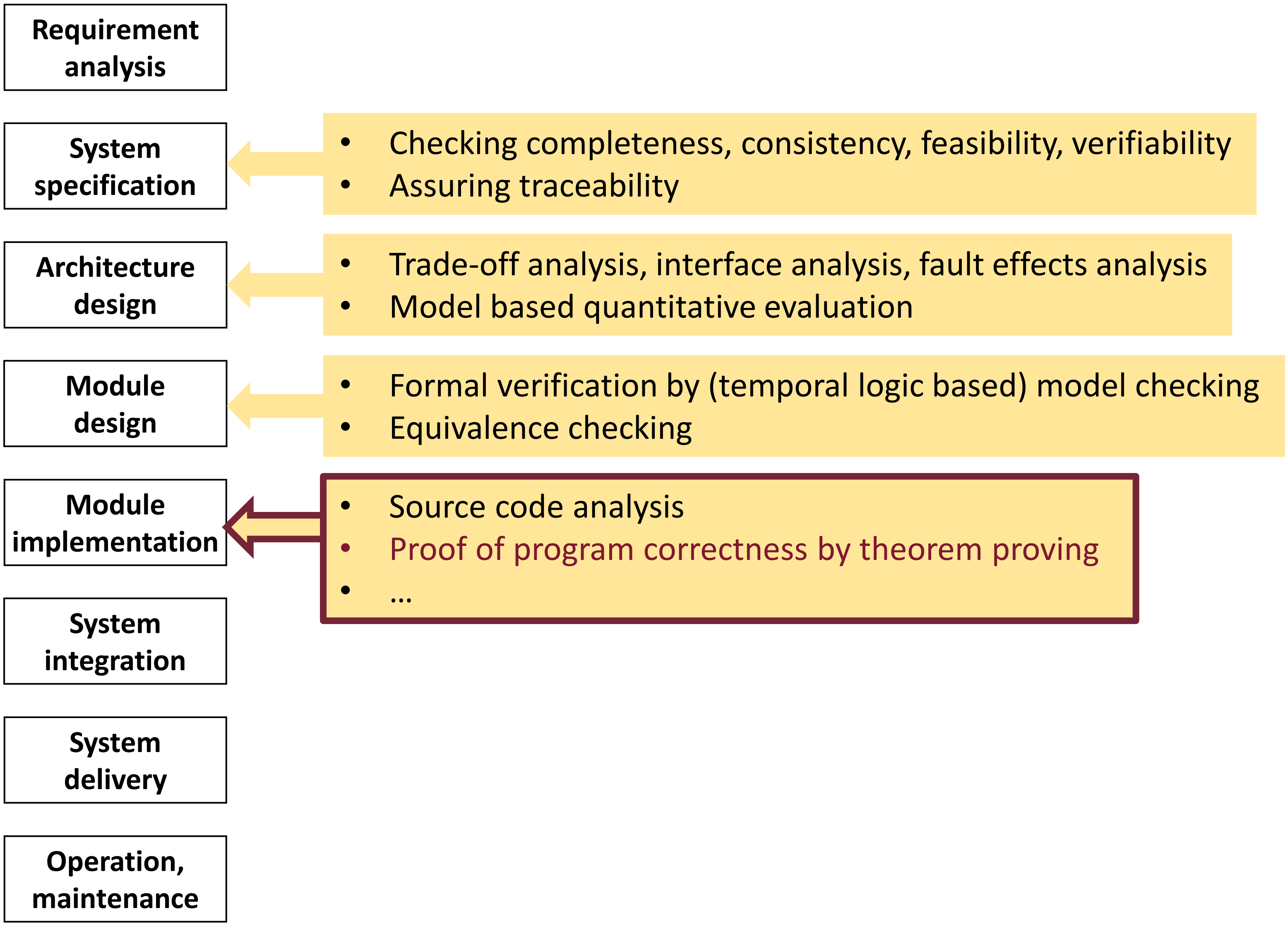
Proof of program correctness

Istvan Majzik
majzik@mit.bme.hu

Budapest University of Technology and Economics
Dept. of Measurement and Information Systems

Typical development steps and V&V tasks



Approach: Theorem proving

Motivation

- Proof of the correctness of critical algorithms
  - Restricted to core functions: safety-critical modules, security related algorithms, communication protocols, …
  - Basis for correctness proof:
    - Detailed design: Algorithm given in pseudo-language (called in the following as "program")
    - Real source code: Subset of real programming languages
- Using a theorem proving approach
  - Property to be verified is a "theorem" to be proven
    - Contract (pre- and post-conditions) can be mapped to theorems: A post-condition is satisfied by the program if the preconditions hold
  - Formal reasoning is applied to prove the theorem
- Challenges:
  - How to derive theorem from a (pseudo) program?
  - What are the efficient proof strategies?

Theorem proving systems

- Parts of theorem proving systems:
  - Deduction system: Description of the problem space
    - Theorem (to be proven): The property to be checked
    - (Logic) axioms: Premise or starting statements for further reasoning
    - Inference rules: Induction, deduction, unification, …
  - Problem description language
    - E.g., first order logic (FOL), FOL extended with types, higher order logic (HOL), …
- Components:
  - Algorithmic: Application of the inference rules
  - Search: Strategy or tactic for selecting inference rules
    - Goal-driven (backward) search
    - Depth-first or breadth-first search
    - Interactive (with hints from the user)
- Popular theorem proving tools
  - HOL, PVS, ACL2, …

Application of theorem proving systems

- Use cases
  - Theorem proving: Deriving automatically the proof of the theorem
  - Proof checking: Automatic checking of a manual proof
  - Interactive proving: Supporting manual proof steps (application of rules)
- Typical tasks for theorem proving
  - Verifying data-intensive algorithms (using theories for the data types)
  - Verifying parameter dependency (e.g., number of participants in a protocol)
    - (Mathematical) induction can be used
  - Using together with model checking for parameterized systems
    - Initially, verifying the property for the smallest parameter: model checking
    - Proof of preserving the property when the parameter increases: by induction
- Automatic theorem proving is a complex task
  - In general, varies from trivial to impossible (depending on the underlying logic)
  - Propositional logic: Decidable, but only exponential-time algorithms are believed to exist for general proof tasks
  - It is important to have a good proof strategy

Properties of theorem proving systems

D deduction system, c property (theorem) to be proven

- Semantic soundness:
  - What can be deduced in D, it is true (it holds)
  - Necessary property for usability
  - Formally: $\forall c$: if $|\!-_D c$ (it can be deduced) then $|\!= c$ (it holds)
- Semantic completeness:
  - What is true, that can be deduced in D
  - Useful property, but not always possible
  - Formally: $\forall c$: if $|\!= c$ then $|\!-_D c$
- Consistency:
  - It is not possible to deduce a theorem and its opposite
- Total soundness and completeness:
  - Sound and complete for all interpretation (of variables)

Mapping the verification task to theorem proving

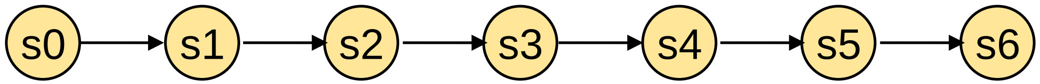
Sources for the parts of deduction systems:

- For the axioms (= starting statements for reasoning):
  - Program domain axioms (e.g., integer, string, list theories)
  - Program statements (e.g., value assignments) – depending on the theorem proving approach
- For the inference rules:
  - Semantics of the programming language
  - Semantics of the program domain
- For the theorem to be proven:
  - Program and its specification (pre- and post-conditions)

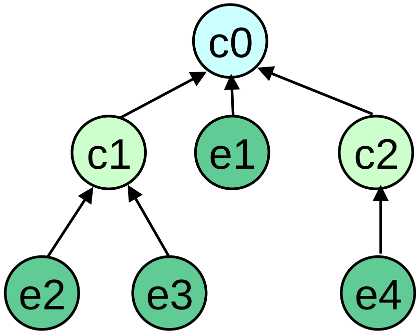
- What is the proper proof strategy?

Inductive strategies for correctness proof

- Computational induction: Based on operational semantics
  - For states in program paths:
    If the properties of the initial state are known then the properties of the terminal state of a program path can be deduced by following the semantics of state transitions



- Structural induction: Based on axiomatic semantics
  - For syntactic constructs:
    If the properties of components are known then the properties of the composite constructs can be derived on the basis of the semantics of the syntactic composition



Goals of this lecture

- Proposing proof strategies for proving program correctness
  - The proof strategy may require manual steps
  - In general, there is no fully automated efficient proof technique
- The strategy is not for a concrete programming language
  - Pseudo-languages are used (for algorithm description)
    - In the following, it is called as "programming language"
  - E.g., domain-specific languages may also be supported
- Assumptions for provability
  - Programming language: Formal semantics is defined (operational or axiomatic semantics)
  - Specification language: First order logic

Specifying program correctness

Programming language with operational semantics

- "State": Configuration $C(\sigma,\lambda)$
  - σ observable state (included in the output of the program)
    - $\sigma[x]$ is the value of variable x in observable state σ
    - $\sigma[\underline{x}]$ is the value of variable vector $\underline{x}$ in observable state σ
  - Unobservable (hidden) state (not relevant for correctness)
  - Syntactic continuation λ: Defines the further computation
    - Analogy: "program counter"
    - Defines the statements to be executed (e.g., in the source code)
- Transition relation among configurations: $\rightarrow$
  - $\pi(P, \sigma_0)$ is the computation of program P from initial observable state σ_0
    - $C_0 \rightarrow C_1 \rightarrow C_2 \rightarrow \ldots$ maximal sequence (to the terminal state, or infinite)
    - $\sigma_0 \rightarrow \sigma_1 \rightarrow \sigma_2 \rightarrow \ldots$ observable state sequence
  - $val(\pi(P, \sigma)) = \sigma_n$ terminal state in case of finite computation
- Domain I: Computations (variables) are interpreted here

Specifying program properties

- Restrictions for the program:
  - Deterministic
  - Terminating (not continuously operating): Performs value (or state) transformation
- Specification of program properties: Predicates
  - Precondition: $p(\underline{x})$ – specifies the allowed initial states
    - $\underline{x}$ variables in the observable state
    - $\sigma_0 \models p(\underline{x})$ means: in the initial state $p(\underline{x})$ holds
  - Postcondition: $q(\underline{x})$ – specifies the acceptable terminal states
    - true – holds in all terminating computations
    - false – does not hold in any terminal state
    - $val(\pi(P,\sigma_0)) \models q(\underline{x})$ means: $q(\underline{x})$ holds in terminal state of π
- Construction of pre- and postconditions
  - Using (existentially quantified) bound auxiliary variables
  - Using specification variables

Examples for specifications (in the integer domain)

- The program outputs x and y where y is greater than x:

  Precondition $p(x,y) = true$, postcondition $q(x,y) = y>x$

  In other form, pre- and postcondition together: $(true, y>x)$

- The program outputs x that is an even number:

  $(true, even(x))$ if there is a function $even(x)$ in the domain

  $(true, \exists y: x=2y)$ here y is a bound auxiliary variable in $q(x)$

- The program doubles its input x:

  $(X=x, x=2X)$ here X is a specification variable

- The program outputs the quotient q and remainder r of the positive integer division x/y:

  $$(X=x \wedge x>0 \wedge Y=y \wedge y>0, \quad X=q \cdot Y+r \wedge 0<=r<Y)$$

  If the value of x and y have to be preserved:

  $$(X=x \wedge x>0 \wedge Y=y \wedge y>0, \quad X=q \cdot Y+r \wedge 0<=r<Y \wedge x=X \wedge y=Y)$$

Program correctness criteria: Partial correctness

- Partial correctness: Notation is $\{p(\underline{x})\}\ P\ \{q(\underline{x})\}$

  A program P is partially correct according to $p(\underline{x})$ and $q(\underline{x})$, if the following statement holds:

  $\forall \pi(P,\sigma_0)$ and $\sigma_0 \models p(\underline{x})$:
  if π terminates then $val(\pi(P,\sigma_0)) \models q(\underline{x})$

- Notes:
  - Statement for the computations that start from an initial state and satisfy the precondition: if the computation terminates, then the postcondition holds in the final state
  - Does not guarantee anything about the computations for which $\sigma_0 \not\models p(\underline{x})$
  - $\{true\}\ P\ \{true\}$ holds for all programs
  - If $\{true\}\ P\ \{false\}$ holds: there is no terminating computation

Program correctness criteria: Total correctness

- (Total) correctness: Notation is $\langle p(\underline{x})\rangle\ P\ \langle q(\underline{x})\rangle$

  Program P is correct according to $p(\underline{x})$ and $q(\underline{x})$,
  if the following statement holds:

  $\forall \pi(P,\sigma_0)$ and $\sigma_0 \models p(\underline{x})$:
  π terminates and $val(\pi(P,\sigma_0)) \models q(\underline{x})$

- Notes:
  - Statement for the computations that start from an initial state and satisfy the precondition: the computation terminates and the postcondition holds in the final state
  - $\langle p(\underline{x})\rangle\ P\ \langle true\rangle$ specifies termination only
  - It can be stated:
    $\langle p(\underline{x})\rangle\ P\ \langle q(\underline{x})\rangle$ iff $\{p(\underline{x})\}\ P\ \{q(\underline{x})\}$ and $\langle p(\underline{x})\rangle\ P\ \langle true\rangle$
    i.e., the program is correct if partially correct and terminates

Proof of correctness for simple flow programs

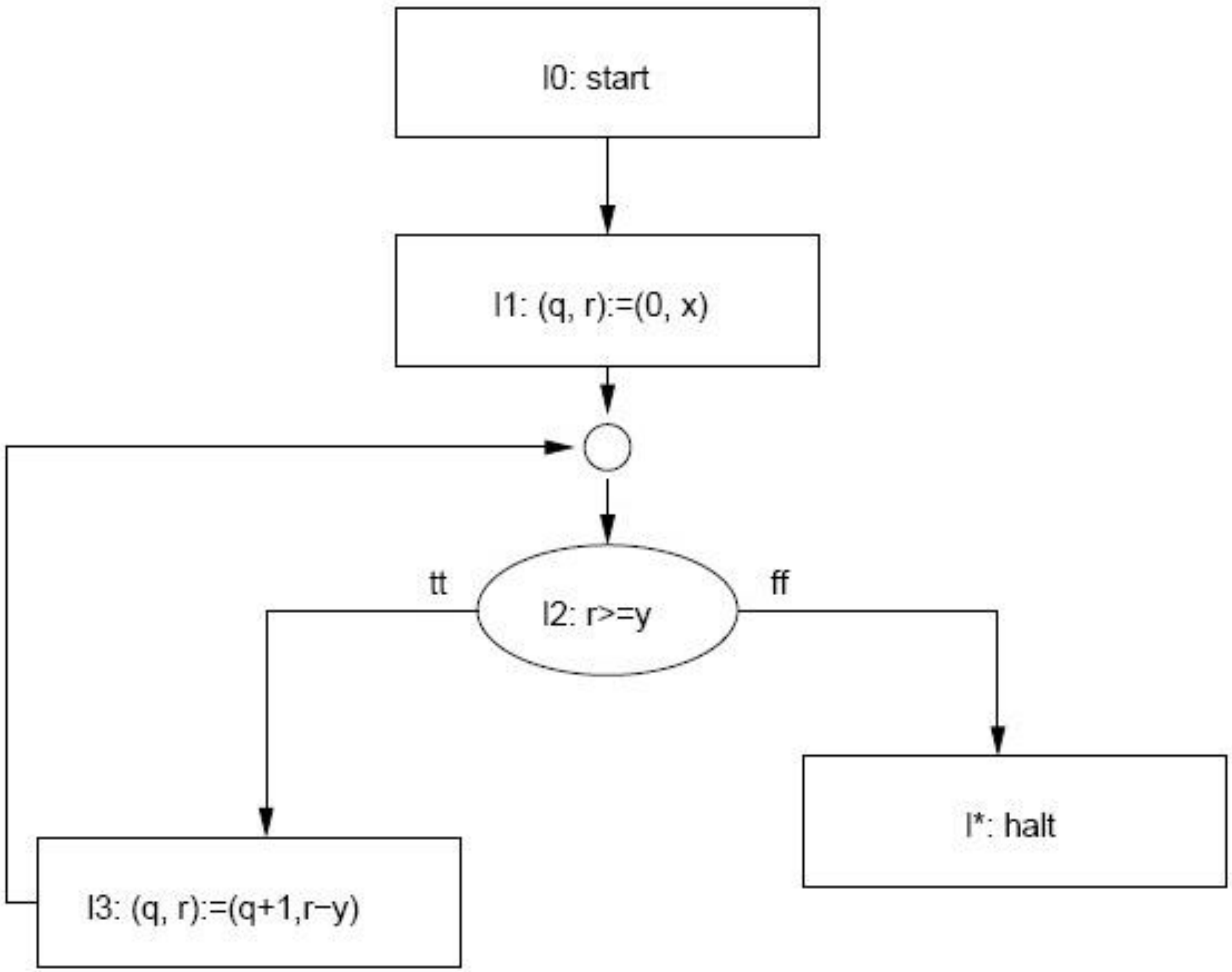
Flow language for simple deterministic programs

- PLF "flow language": Pseudo-language similar to assembly
  - Statements *start*, *x:=e*, *B(x)*, *halt* with unique labels l_0 (start), l_* (halt), l_i, …
- Structure of a PLF program: Finite directed graph
  - Vertices: statements; edges: sequencing of statements
  - Notation: $succ(l_i)$, and $succ^+(l_i)$, $succ^-(l_i)$ in case of branch give the next vertex
  - All statements are on a start $\rightarrow$ halt path
- Semantics of PLF: Defining $C=(\sigma,\lambda)$ configuration and $\rightarrow$ relation: $C(\sigma,\lambda) \rightarrow C'(\sigma',\lambda')$ with λ as label l iff
  - λ is at *start*: $\lambda'=succ(\lambda)$, $\sigma'=\sigma$
  - λ is at statement *x:=e* : $\lambda'=succ(\lambda)$, $\sigma'=\sigma[e/x]$

    here [e/x] denotes that e replaces x
  - λ is at branching condition *B(x)*:
    - If $\sigma|=B(x)$ then $\lambda'=succ^+(\lambda)$, $\sigma'=\sigma$
    - If $\sigma|\neq B(x)$ then $\lambda'=succ^-(\lambda)$, $\sigma'=\sigma$

Example: Integer division

x/y positive integer division, dividend x, divider y, quotient q, remainder r:



Preview of the proof strategies

- Partial correctness for loop-free programs
  - Approach: Backward computational induction
- Partial correctness for programs with loops
  - Approach: Inductive assertions
- Correctness for programs with loops: Proving termination
  - Approach: Parameterized inductive assertions

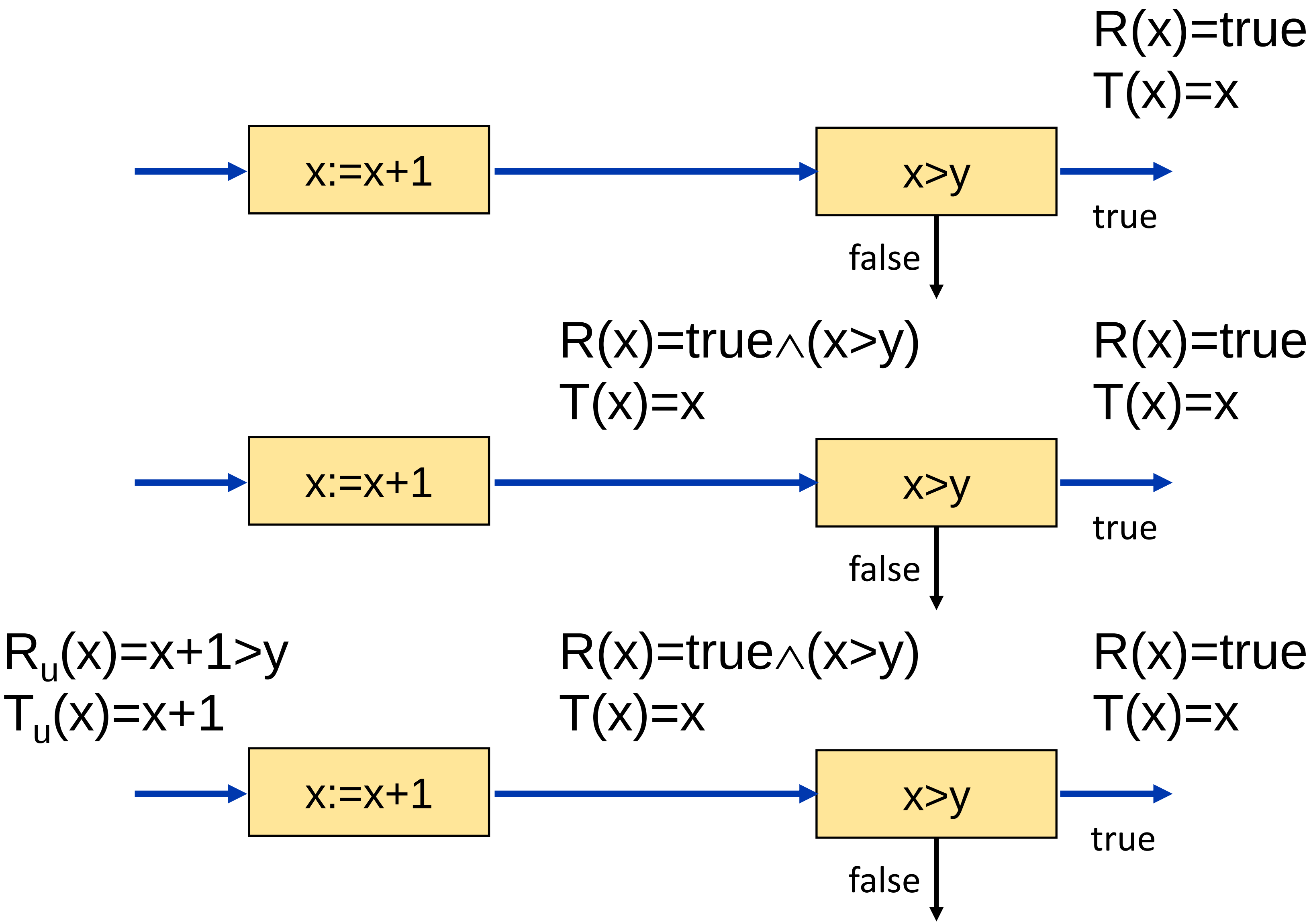
Partial correctness for loop-free programs (1)

- Idea: Computational induction in case of proving $\{p\}\ P\ \{q\}$
- Characteristics of a path u belonging to a finite computation
  $u = l_0 \rightarrow l_1 \rightarrow l_2 \rightarrow \ldots \rightarrow l_m \rightarrow \ldots \rightarrow l_k$
  - **Reachability condition**: $R_u(\underline{x})$ predicate for traversing path u
    - If it holds in case of l_0 then the path u is traversed
  - **State transformation**: $T_u(\underline{x})$ the final state after traversing path u
    - Starting from a state vector $\underline{x}$, after traversing u the observable final state is $T_u(\underline{x})$
    - In other words: $\underline{x} := T_u(\underline{x})$ is the state transformation performed by the program on path u
- Notation:
  - $l_m \rightarrow \ldots \rightarrow l_k$ suffix of the path from index m (from vertex l_m)
  - $R_u^m(\underline{x})$ and $T_u^m(\underline{x})$ refer to these path suffix

Partial correctness for loop-free programs (2)

- It is known that for the end vertex l_k of the path (last path suffix):
  - $R_u^k(\underline{x})$ = true - since the end vertex has been reached
  - $T_u^k(\underline{x}) = \underline{x}$ - since there is no further state transformation
- Backward substitution on path u:
  - Assume: $R_u^{m+1}(\underline{x})$ and $T_u^{m+1}(\underline{x})$ are known for a suffix (first: end vertex)
  - Step: Computing $R_u^m(\underline{x})$ and $T_u^m(\underline{x})$ on the basis of the statement at l_m
    - *$\underline{x}$:=$\underline{e}$* assignment:
      $R_u^m(\underline{x}) = R_u^{m+1}(\underline{x})[\underline{e}/\underline{x}]$, $T_u^m(\underline{x}) = T_u^{m+1}(\underline{x})[\underline{e}/\underline{x}]$
    - *$B(\underline{x})$* with true branch:
      $R_u^m(\underline{x}) = R_u^{m+1}(\underline{x}) \wedge B(x)$, $T_u^m(\underline{x}) = T_u^{m+1}(\underline{x})$
    - *$B(\underline{x})$* with false branch:
      $R_u^m(\underline{x}) = R_u^{m+1}(\underline{x}) \wedge \neg B(x)$, $T_u^m(\underline{x}) = T_u^{m+1}(\underline{x})$
    - *start*:
      $R_u(\underline{x}) = R_u^0(\underline{x})$, $T_u(\underline{x}) = T_u^0(\underline{x})$
  - This way $R_u(\underline{x})$ and $T_u(\underline{x})$ can be computed for the path u by backward substitution

Example for backward substitution



Partial correctness for loop-free programs (3)

- Strategy for proving partial correctness:

  $\{p(\underline{x})\}\ P\ \{q(\underline{x})\}$ iff for each complete path u:

  $\forall \underline{x}: p(\underline{x}) \wedge R_u(\underline{x}) \Rightarrow q(T_u(\underline{x}))$ verification condition holds

First order logic expression on the domain;
can be given to a theorem prover for each path u:

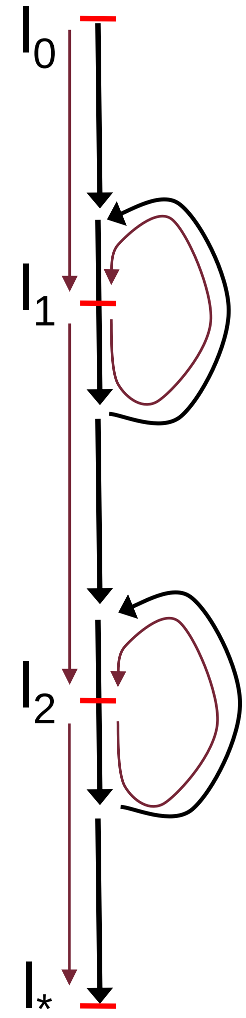
$$\forall \underline{x}: p(\underline{x}) \wedge R_u(\underline{x}) \Rightarrow q(T_u(\underline{x}))$$

$p()$ and $q()$ are given by the specification

$R_u()$ and $T_u()$ are derived for each path of the program source, using backward substitution

Partial correctness for programs with loops (1)

- Idea: Cutting the loops
  - In each loop, a vertex l_i is determined which cuts the loop into loop-free segments
  - To each cut point l_i, a predicate $I_{li}(\underline{x})$, the so-called inductive assertion is assigned
    - It shall be true when first reaching l_i
    - It shall hold when executing the loop (loop invariant)
    - It shall make true the reachability condition of the next segment when exiting the loop,
      or make true the postcondition at the final vertex
  - These segments can be checked as loop-free programs according to the previous strategy
    - Each reachability condition and
    - state transformation can be computed



Partial correctness for programs with loops (2)

- Proof strategy:
  - Finding (at least one) cut point in each loop
  - Assigning inductive assertions: $I_{li}(\underline{x})$
    - For the initial vertex: $I_{l_0}(\underline{x}) = p(\underline{x})$ or $\forall \underline{x}: p(\underline{x}) \Rightarrow I_{l_0}(\underline{x})$
    - For the final vertex: $I_{l*}(\underline{x}) = q(\underline{x})$ or $\forall \underline{x}: I_{l*}(\underline{x}) \Rightarrow q(\underline{x})$
    - In loops: loop invariants as given above
  - Verification conditions (to be proven): For each loop-free segment u given by subsequent cut points l and l':

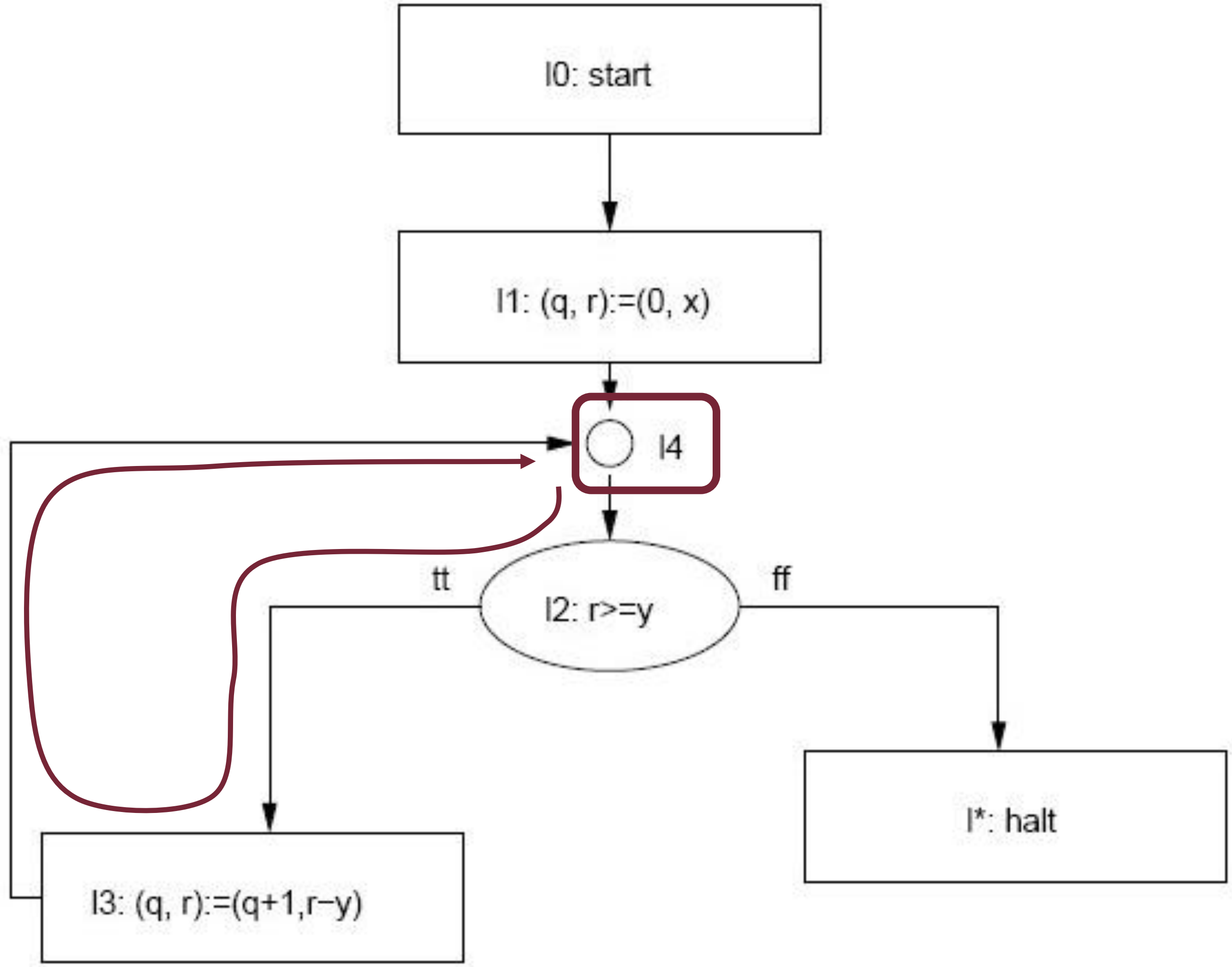
$$\forall \underline{x}: I_l(\underline{x}) \wedge R_u(\underline{x}) \Rightarrow I_{l'}(T_u(\underline{x}))$$

    - Here $R_u(\underline{x})$ and $T_u(\underline{x})$ can be computed for the segments
- Correct and complete strategy
  - Cut points and inductive assertions can always be found (the proof is not constructive ☹)
  - The assignment of inductive assertions is a heuristic procedure

Example: Inductive assertion (loop invariant)

x/y positive integer division, dividend x, divider y, quotient q, remainder r:

$$I_{l4}(x,y,q,r) = (x \geq 0 \land y > 0 \land x = q \cdot y + r \land r \geq 0)$$



Proving termination in case of loops (1)

Idea: Parameterized inductive assertions

- The parameter is from a (W, >) well-founded set
  - There is no infinite decreasing $w_0 > w_1 > \ldots$ sequence of $w_i \in W$
  - Examples for well-founded sets:
    - Natural numbers, with the common > relation
    - Strict subsets of a finite set, with the inclusion relation
    - Finite list, with the prefix relation
- The loop terminates if it can be shown that the parameter decreases in each execution of the loop
  - There is no infinite decreasing sequence $\rightarrow$ termination
- The parameter in most cases can be the loop variable, but (computed) auxiliary variables can also be used
  - However, finding parameters is a heuristic procedure

Proving termination in case of loops (2)

- Proof strategy:
  - Finding cut point in each loop: l_i, with l_0 and l_*
  - Finding well-founded set(s) for the cut points: $(W,<)$
  - Assigning parameterized inductive assertions: $I_l(\underline{x},w)$ where $w \in W$
  - Verification conditions (to be proven) :
    - At the initial vertex: $\forall \underline{x}:\ p(\underline{x}) \Rightarrow \exists w:\ I_{l0}(\underline{x},w)$
    - At the terminal vertex: $\forall \underline{x}:\ I_{l*}(\underline{x},w) \Rightarrow q(\underline{x})$
    - For each loop-free segment u given by subsequent l and l':

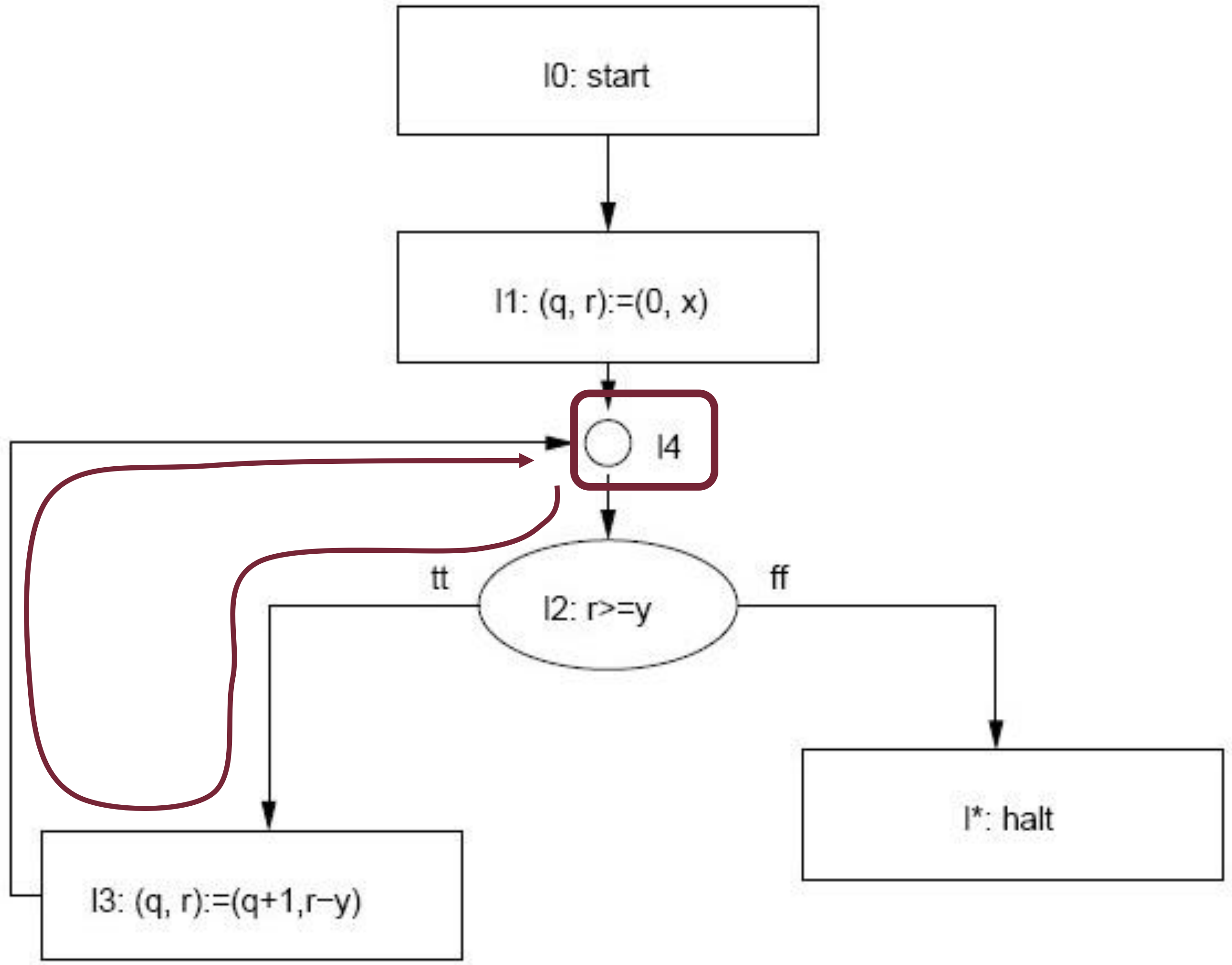
$$\forall \underline{x}:\ I_l(\underline{x},w) \wedge R_u(\underline{x}) \Rightarrow \exists w'<w:\ I_{l'}(T_u(\underline{x}),w')$$

    Here $R_u(\underline{x})$ and $T_u(\underline{x})$ can be computed for the segments
- Correct strategy for proving $<p(\underline{x})>\ P\ <true>$
  - However, the assignment of parameterized inductive assertions is a heuristic procedure

Example: Parameterized inductive assertion

x/y positive integer division, dividend x, divider y, quotient q, remainder r:

$I_{l4}(x,y,q,r,n) = (x \geq 0 \wedge y > 0 \wedge x = q \cdot y + r \wedge r \geq 0 \wedge n = r)$, n positive integer



Summary for low-level flow languages

- Partial correctness for loop-free programs
  - Backward computational induction
- Partial correctness for programs with loops
  - Inductive assertions
- Correctness for programs with loops: Proving termination
  - Parameterized inductive assertions,
    with a decreasing parameter from a well-founded set
    in each loop segment

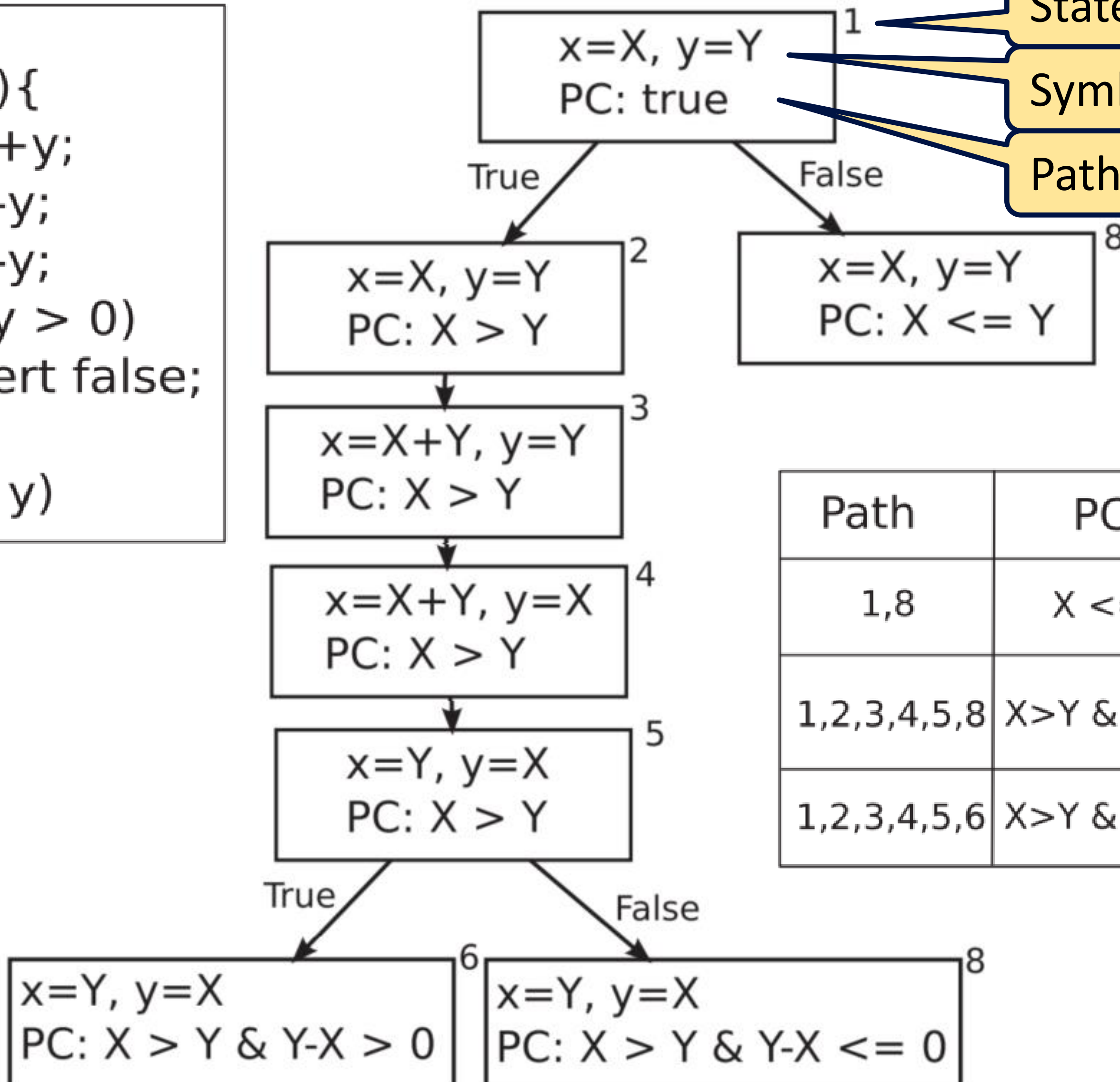
Outlook: Symbolic execution

Basic idea

- Static program analysis technique
- Basic idea
  - Following computation of paths with symbolic variables
  - Deriving reachability conditions as path constraints
  - Constraint solving (e.g., SMT solver):
    A solution yields an input to execute a given path
- Popular nowadays:
  - Efficient SMT solvers exist
  - Used to generate test inputs for covering given paths
  - Mixing symbolic and concrete execution: “Concolic”

Example for deriving path constraints

```
   int x, y;
1  if(x > y){
2     x = x+y;
3     y = x-y;
4     x = x-y;
5     if(x - y > 0)
6        assert false;
7  }
8  print(x, y)
```



Path	PC	Program Input
1,8	X <= Y	X=1 Y=1
1,2,3,4,5,8	X>Y & Y-X<=0	X=2 Y=1
1,2,3,4,5,6	X>Y & Y-X>0	none

Tools for symbolic execution and test generation

Name	Platform	Language	Notes
KLEE	Linux	C (LLVM bitcode)	
Pex	Windows	.NET assembly	VS2015: IntelliTest
SAGE	Windows	x86 binary	Security testing, SaaS model
Jalangi		JavaScript	
Symbolic PathFinder		Java	