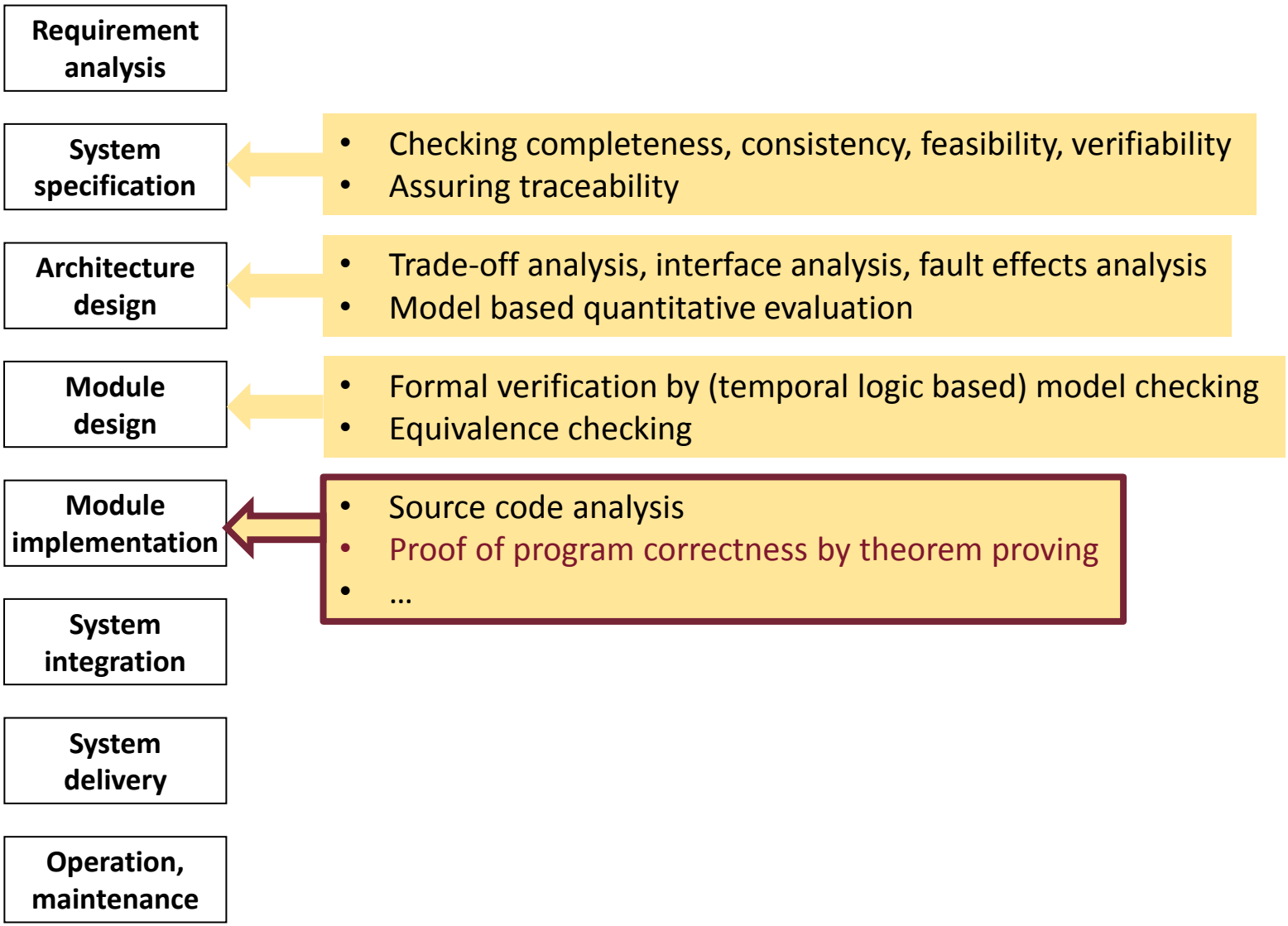
Proof of program correctness

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 statement 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 (applications 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 $\vdash_D c$ (it can be deduced) then $\models c$ (it holds)
- Semantic completeness:
  - What is true, that can be deduced in D
  - Useful property, but not always possible
  - Formally: $\forall c$: if $\models c$ then $\vdash_D c$
- 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:

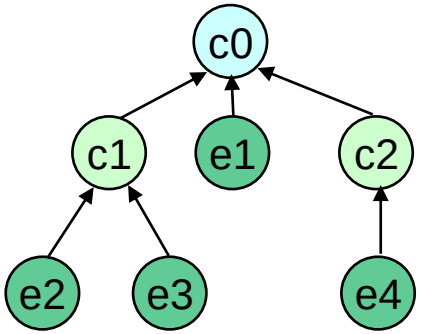
- For the axioms (= starting statements for reasoning):
  - Program domain axioms (e.g., integer, string, list theories)
  - Program statements (e.g., value assignments) – depending on the theorem proving approach
- For the inference rules:
  - Semantics of the programming language
  - Semantics of the 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
  - σ observable state (included in the output if 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 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 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

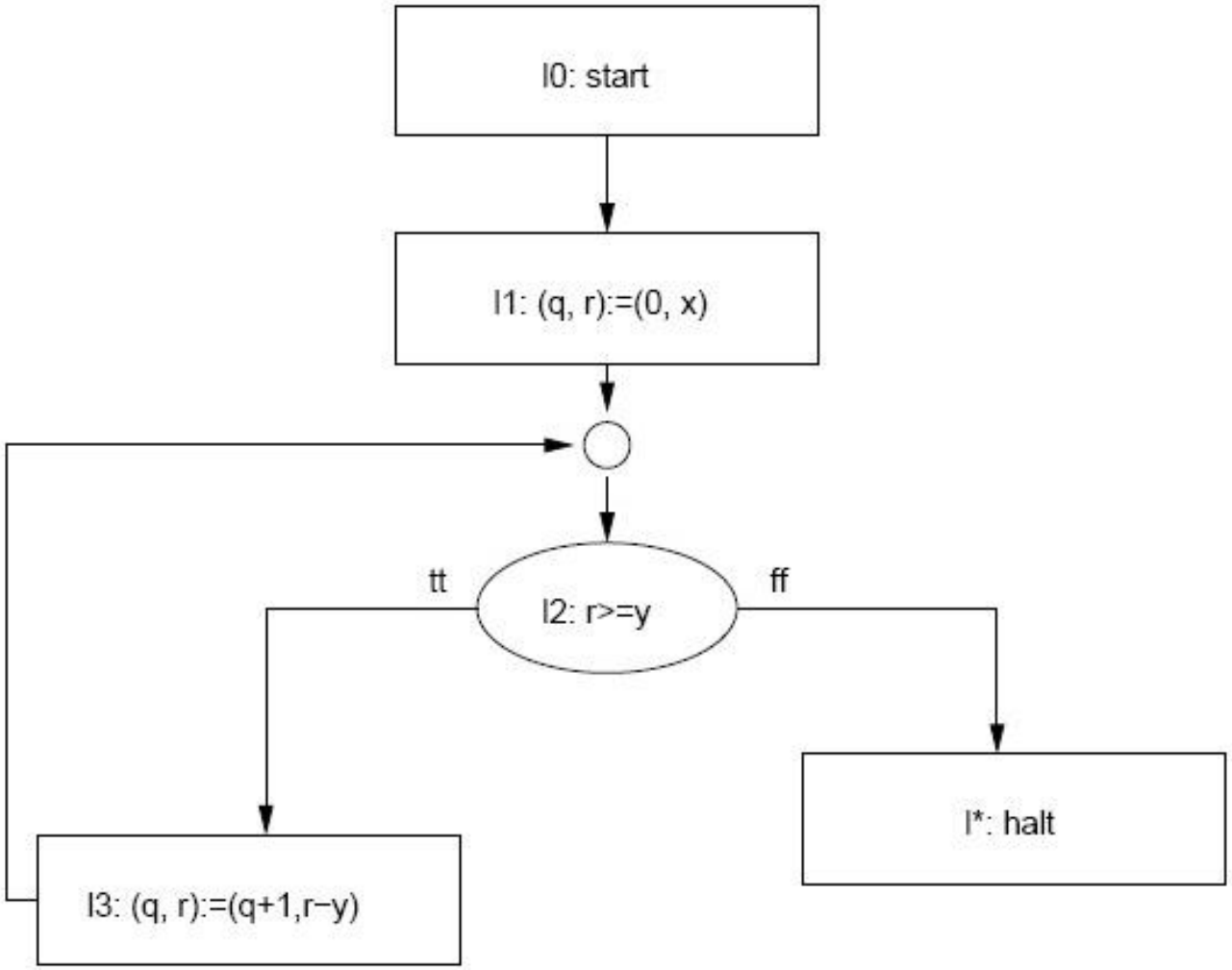
Flow language for simple deterministic programs

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

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

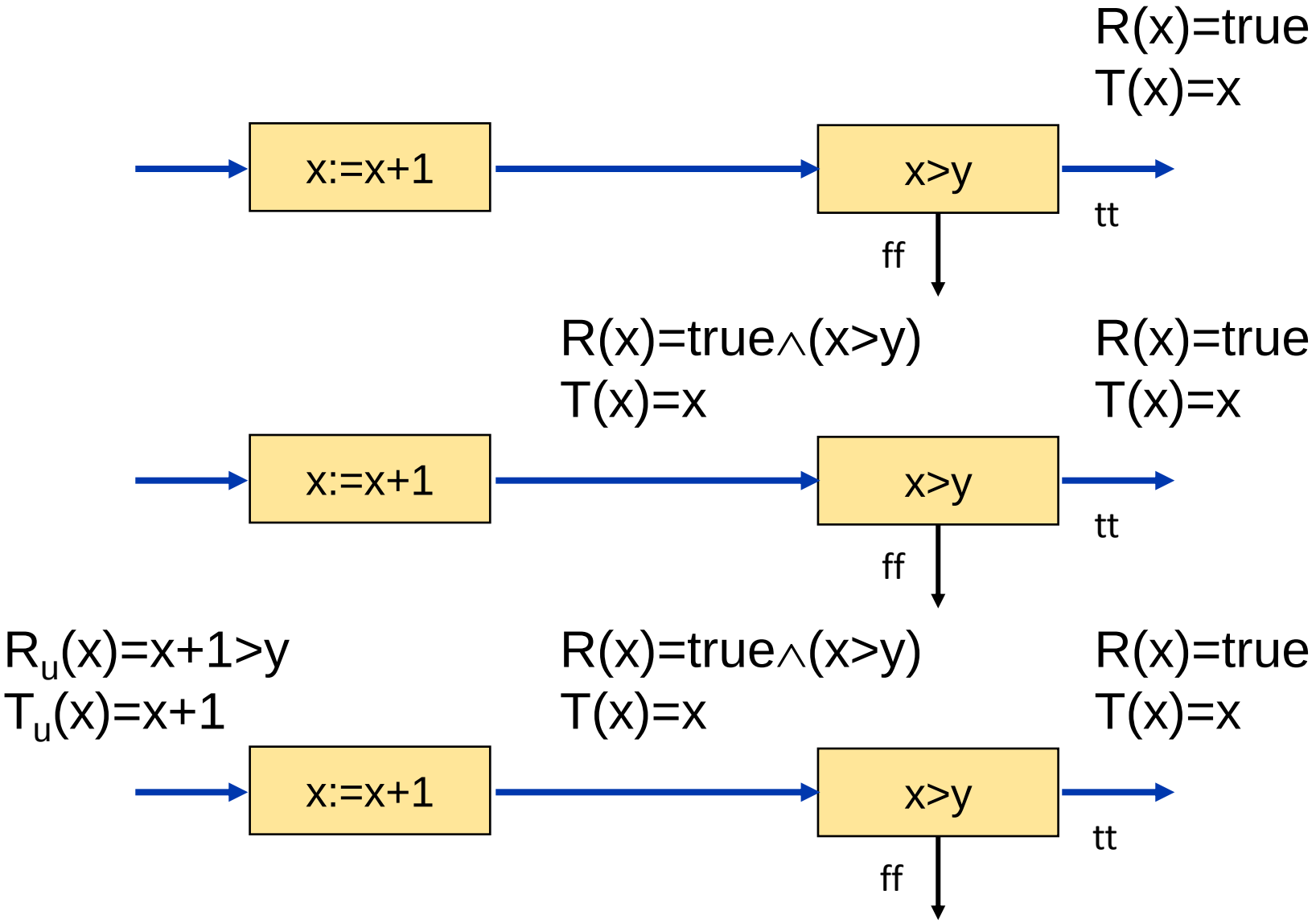
Partial correctness for loop-free programs (1)

- 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 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:
  - Assume: $R_u^{m+1}(\underline{x})$ and $T_u^{m+1}(\underline{x})$ are known for a suffix
  - 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:

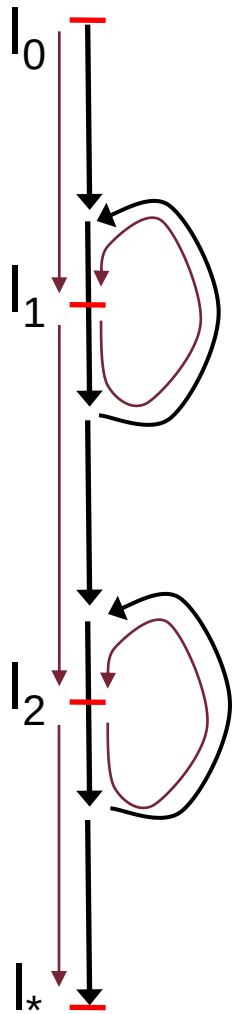
$$\forall \underline{x}: p(\underline{x}) \wedge R_u(\underline{x}) \Rightarrow q(T_u(\underline{x}))$$

$p()$ 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 $l_{li}(\underline{x})$, 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
    - 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: $I_{l_i}(\underline{x})$
    - For the initial vertex: $I_{l_0}(\underline{x}) = p(\underline{x})$ or $p(\underline{x}) \Rightarrow I_{l_0}(\underline{x})$
    - For the final vertex: $I_{l*}(\underline{x}) = q(\underline{x})$ or $I_{l*}(\underline{x}) \Rightarrow q(\underline{x})$
    - 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':

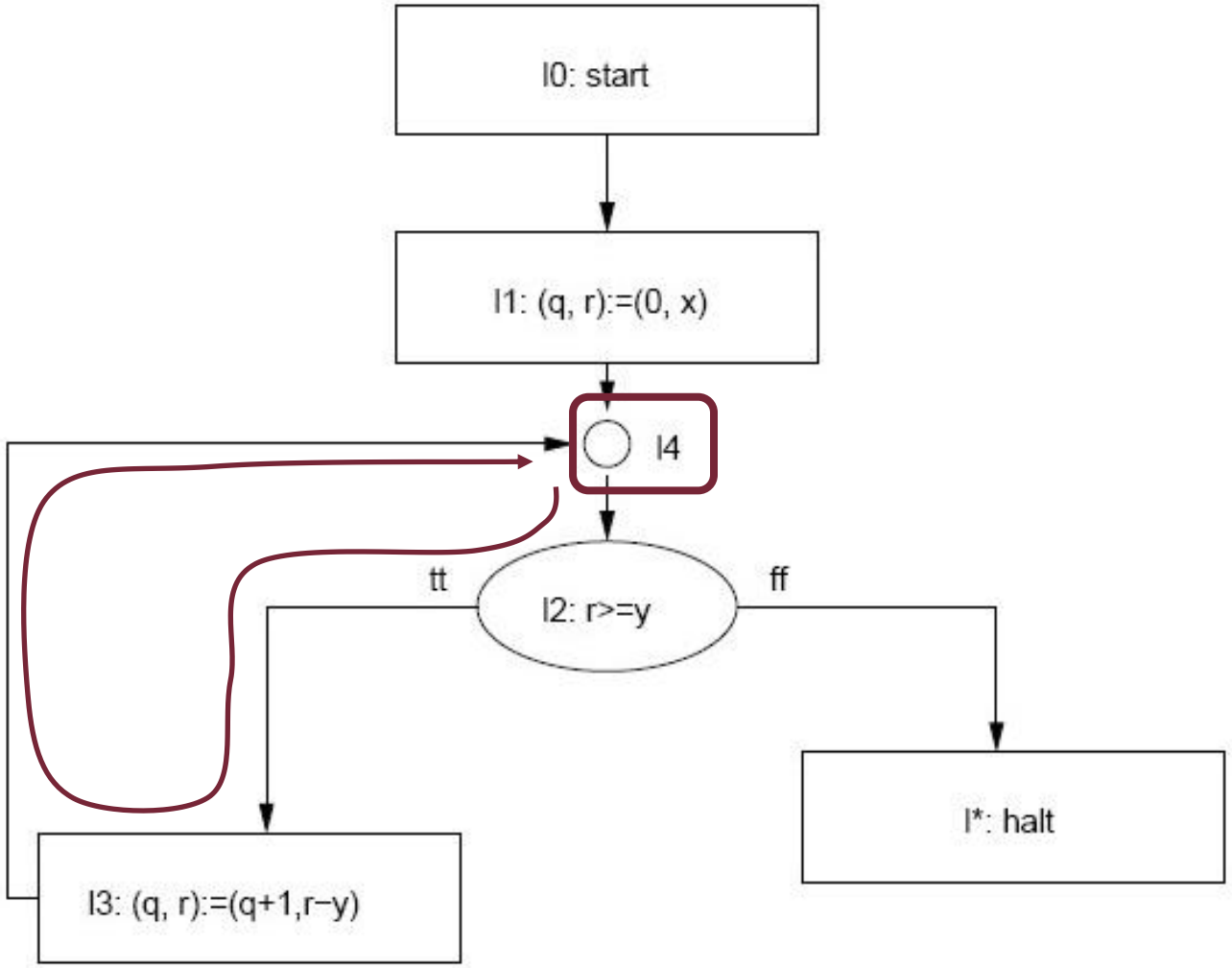
$$\forall \underline{x}: I_l(\underline{x}) \wedge R_u(\underline{x}) \Rightarrow I_{l'}(T_u(\underline{x}))$$

    - 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:

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



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

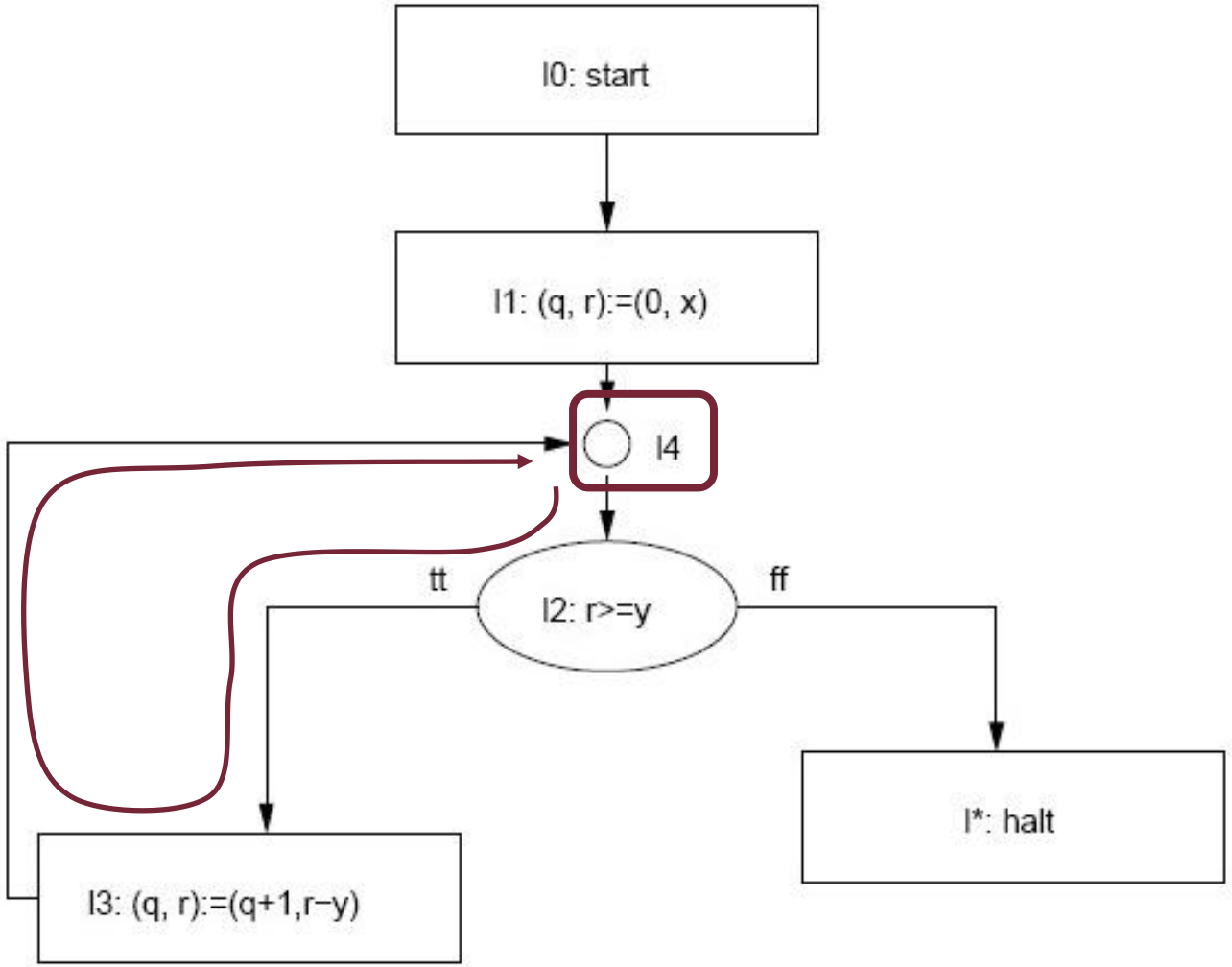
      $$\forall \underline{x}: \; l_l(\underline{x},w) \wedge R_u(\underline{x}) \Rightarrow \exists w' < w: l_{l'}(T_u(\underline{x}),w')$$

      Here $R_u(\underline{x})$ and $T_u(\underline{x})$ can be computed for the segments
- Correct strategy for proving $\langle p(\underline{x}) \rangle \; P \; \langle true \rangle$
  - 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 pos. integer



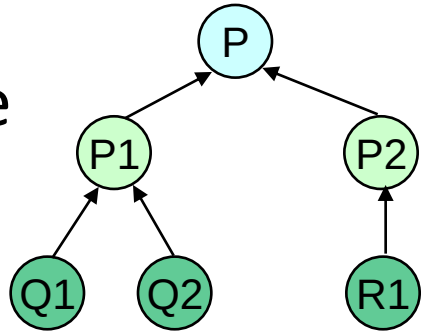
Summary for low-level flow languages

- 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

Proof of correctness for structured programs

Proving correctness for structured programs

- "Composition" of properties:
  - If a program P consists of syntactic units P_1 and P_2 then the properties of P can be derived on the basis of the properties of the syntactic units P_1 and P_2
  - The principle of structural induction



- Structured programs: PLW language

P::= x:=e | skip | P_1; P_2 | if B then P_1 else P_2 fi | while B do P od

- Example (positive integer division):

P_{div}: r:=x; q:=0; while r≥y do r:=r-y; q:=q+1 od

Operational semantics of PLW

- Configuration: $C_i=(P_i, \sigma_i)$ where
  - P_i is the syntactic continuation (E denotes empty cont.)
  - σ_i is the observable state (variables)
- Transition relation: $C \rightarrow C'$
  - $(x:=e, \sigma) \rightarrow (E, \sigma[e/x])$
  - $(skip, \sigma) \rightarrow (E, \sigma)$
  - $(P_1; P_2, \sigma) \rightarrow (P_1'; P_2, \sigma')$ if $(P_1, \sigma) \rightarrow (P_1', \sigma')$
  - $(\text{if } B \text{ then } P_1 \text{ else } P_2 \text{ fi}, \sigma) \rightarrow (P_1, \sigma)$ if $\sigma[B]=true$
    $\rightarrow (P_2, \sigma)$ if $\sigma[B]=false$
  - $(\text{while } B \text{ do } P \text{ od}, \sigma) \rightarrow (P; \text{while } B \text{ do } P \text{ od}, \sigma)$ if $\sigma[B]=true$
    $\rightarrow (E, \sigma)$ if $\sigma[B]=false$

Here $E;P \equiv P$ is applied at the end

D deduction system for proving partial correctness (1)

- Axioms:
  - ASS: $\{p[e/x]\}\ x:=e\ \{p\}$
  - SKIP: $\{p\}\ \text{skip}\ \{p\}$

p holds as postcondition, if p[e/x] holds as precondition

- Rules for the syntactic constructs:

Rule format:

$$\frac{\text{Condition}}{\text{Consequence}}$$

  - SEQ:
$$\frac{\{p\}\ P_1\ \{r\} \quad \text{and} \quad \{r\}\ P_2\ \{q\}}{\{p\}\ P_1; P_2\ \{q\}}$$
  - COND:
$$\frac{\{p \wedge B\}\ P_1\ \{q\} \quad \text{and} \quad \{p \wedge \neg B\}\ P_2\ \{q\}}{\{p\}\ \text{if } B \text{ then } P_1 \text{ else } P_2 \text{ fi}\ \{q\}}$$
  - REP:
$$\frac{\{p \wedge B\}\ P\ \{p\}}{\{p\}\ \text{while } B \text{ do } P \text{ od}\ \{p \wedge \neg B\}}$$

p is a loop invariant

D deduction system for proving partial correctness (2)

- General rules:
  - CONS: $$\frac{p \Rightarrow p_1 \text{ and } \{p_1\}\ P\ \{q_1\} \text{ and } q_1 \Rightarrow q}{\{p\}\ P\ \{q\}}$$
  - AND: $$\frac{\{p\}\ P\ \{q_1\} \text{ and } \{p\}\ P\ \{q_2\}}{\{p\}\ P\ \{q_1 \wedge q_2\}}$$
  - OR: $$\frac{\{p_1\}\ P\ \{q\} \text{ and } \{p_2\}\ P\ \{q\}}{\{p_1 \vee p_2\}\ P\ \{q\}}$$

Strengthening precondition and weakening postcondition

Separated proof of conjunctive postcondition

Separating cases of disjunctive precondition

- Domain axioms and rules:
  To be included in the deduction system

Example: Proving partial correctness

$\{x \geq 0 \wedge y \geq 0\}$ r:=x; q:=0; while r≥y do r:=r-y; q:=q+1 od $\{x = q \cdot y + r \wedge 0 \leq r < y\}$

1. $\{x = 0 \cdot y + x \wedge x \geq 0\}\ q := 0\ \{x = q \cdot y + x \wedge x \geq 0\}$ (ASS)
2. $\{x = q \cdot y + x \wedge x \geq 0\}\ r := x\ \{x = q \cdot y + r \wedge r \geq 0\}$ (ASS)
3. $\{x = 0 \cdot y + x \wedge x \geq 0\}\ q := 0; r := x\ \{x = q \cdot y + r \wedge r \geq 0\}$ (1)(2)(SEQ)
4. $x \geq 0 \wedge y \geq 0 \Rightarrow x = 0 \cdot y + x \wedge x \geq 0$ (ARITHMETIC)
5. $\{x \geq 0 \wedge y \geq 0\}\ q := 0; r := x\ \{x = q \cdot y + r \wedge r \geq 0\}$ (3)(4)(CONS)
6. $\{x = (q+1) \cdot y + r - y \wedge r - y \geq 0\}\ r := r - y\ \{x = (q+1) \cdot y + r \wedge r \geq 0\}$ (ASS)
7. $\{x = (q+1) \cdot y + r \wedge r \geq 0\}\ q := q + 1\ \{x = q \cdot y + r \wedge r \geq 0\}$ (ASS)
8. $\{x = (q+1) \cdot y + r - y \wedge r - y \geq 0\}\ r := r - y; q := q + 1\ \{x = q \cdot y + r \wedge r \geq 0\}$ (6)(7)(SEQ)
9. $x := q \cdot y + r \wedge r \geq 0 \wedge r \geq y \Rightarrow x = (q+1) \cdot y + r - y \wedge r - y \geq 0$ (ARITHMETIC)
10. $\{x = q \cdot y + r \wedge r \geq 0 \wedge r \geq y\}\ r := r - y; q := q + 1\ \{x = q \cdot y + r \wedge r \geq 0\}$ (8)(9)(CONS)
11. $\{x = q \cdot y + r \wedge r \geq 0\}$ while $r \geq y$ do $r := r - y; q := q + 1$ od $\{x = q \cdot y + r \wedge 0 \leq r < y\}$ (10)(REP)
12. $\{x \geq 0 \wedge y \geq 0\}\ q := 0; r := x;$ while $r \geq y$ do $r := r - y; q := q + 1$ od $\{x = q \cdot y + r \wedge 0 \leq r < y\}$ (5)(11)(SEQ)

D* deduction system for proving correctness

- Goal: Proving termination of loops
  - while B do P od constructs
- Basic idea: Parametric assertions
  - Parameters from well-founded set
  - E.g., selecting n natural number: arithmetic extension of the specification language is needed
  - pi(x,n) parameterized loop invariant
- Modified REP rules for proving correctness:

Loop invariant, decreases n

  - REP*:

$$\frac{pi(x,n) \Rightarrow B \text{ and } \langle pi(x,n)\rangle P \langle pi(x,n-1)\rangle \text{ and } pi(x,0) \Rightarrow \neg B}{\langle \exists n: pi(x,n)\rangle \text{ while } B \text{ do } P \text{ od } \langle pi(x,0)\rangle}$$

  - All other rules are the same, writing <...> instead of {...}

Properties of the deduction systems

- Notation for the proof of a statement C: $Tr_I \vdash_D C$ where
  - I domain, Tr_I the axioms and deduction rules of the domain
  - D the deduction system
- Properties:
  - The correctness of D defined above can be proven
    - $Tr_I \vdash_D \{p\}P\{q\}$ results in $\models_I \{p\}P\{q\}$
  - The completeness of D cannot be proven
    - If the axioms and rules of the domain are complex enough (e.g., contain the arithmetic of natural numbers): Gödel's first incompleteness theorem holds, i.e., there are statements that are not provable
- Practical implementation:
  - The semantics of the programming language (syntactic constructs) have to be mapped to axioms and rules
  - The theorem prover shall include the axioms and rules of the domain
  - Strategy (or search) is needed for selecting proper domain rules
  - The specification language shall be expressive enough

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
    - Inductive assertions with a decreasing parameter from a well-founded set
- Structured languages (while programs):
  - Partial correctness:
    - Deduction system with structural induction
  - Correctness:
    - Deduction system with parameterized inductive assertions
    - Arithmetic extension to have a well-founded set

Proving program correctness in practice

Classic examples:

- Spec# Programming System: C# extension
  - Preconditions, postconditions (for methods) can be specified
  - Object level invariants (e.g., ranges for variables) can be given
  - Boogie2: To prove postconditions in an automated way
- JML: Java Modelling Language
  - Preconditions, postconditions, invariants can be specified
  - ESC/Java2: Proof of postconditions for a JML subset
- SPARK: Ada language subset
  - Proof by using an interactive theorem prover
- B method: Specific modelling language and approach
  - B4Free, Rodin: The derivation of verification conditions (to be proven) and theorem proving are automated (with some manual support)
  - Focus: Consistent refinement of a specification