1. Introduction (ch. 1)

Predicate logic fundamentals

- See separate class notes **(required self-study)**
- **Illustration:** truth tables for not, and, or, imply, etc.

System specifications in context

- programming language semantics, e.g., sequence construct "c1 ; c2"
  - denotational: E(c2, E(c1, state0))
  - operational: exec(c1), exec(c2)
  - axiomatic: given {P} c1 {Q}, {R} c2 {W}, $Q \Rightarrow R$ conclude {P} c1 ; c2 {W}
- software engineering
  - program design and coding, e.g., array sorting
  - component specifications as part of the software architecture design
  - requirements engineering
- formal verification and analysis, e.g., security

Proof systems — a historical perspective

Sequential programming tradition

Floyd — flowchart annotations and backward substitution

- partial and total correctness

Hoare triple — structured programs

- {P} s {Q} is true if whenever s is started in a state satisfying P, the resulting state satisfies Q, if s terminates
- {P} skip {P} — skip axiom
- $\{P^{x}_{e}\}$ x := e {P} — assignment axiom, e.g., {true} x := 5 {x=5}
- $P' \Rightarrow P$, {P} s {Q}, $Q \Rightarrow Q' \vdash$ {P'} s {Q'} — consequence
- {P} s_1 {Q}, {Q} s_2 {R} $\vdash$ {P} s_1 ; s_2 {R} — sequential composition
- $P \wedge \neg(B_1 \vee ... \vee B_n) \Rightarrow Q$, for $1 \le i \le n$ $\{P \wedge B_i\}$ s_i {Q} $\vdash$ {P} IF {Q}
  — alternative rule
- for $1 \le i \le n$ $\{I \wedge B_i\}$ s_i {I} $\vdash$ {I} DO $\{I \wedge \neg(B_1 \vee ... \vee B_n)\}$
  — iterative rule

- **Illustration**: zero an array of integers

  - use deterministic sequential processing inside a loop

  - use non-deterministic selection inside a loop

Dijkstra — *wp*-calculus

- statements are viewed as predicate transformers
- the programmer usually knows what is the desired result
- *wp*(s,Q) — the largest set of states such that, if s is started in one of these states, s is guaranteed to terminate in a state satisfying Q

- $\{wp(s,Q)\}\ s\ \{Q\}$
- $P \Rightarrow wp(s,Q) \vdash \{P\}\ s\ \{Q\}$
- $wp(s,false) = false$ — law of excluded miracle
- $wp(s,Q) \wedge wp(s,R) = wp(s,Q \wedge R)$ — distributive law of conjunction
- $wp(s,Q) \vee wp(s,R) \Rightarrow wp(s,Q \vee R)$ — distributive law of disjunction

  Note: this is due to the presence of nondeterminism

  wp(flip_coin, head) = false and *wp*(flip_coin, tail) = false

  but

  wp(flip_coin, head $\vee$ tail) = true
- $wp(skip,Q) = Q$
- $wp(x:=e,Q) = Q^{x}_{e}$
- $wp(s_1 ; s_2 ,Q) = wp(s_1,wp(s_2 ,Q))$
- $wp(IF,Q) = \neg (B_1 \vee ... \vee B_n) \Rightarrow Q \wedge \text{for } 1 \le i \le n\ (B_i \Rightarrow wp(s_i ,Q))$
- $wp(DO,Q) = (\exists\ k: 0 \le k : H_k(Q))$

  $H_0(Q) = \neg BB \wedge Q;\ H_k(Q) = H_0(Q) \vee wp(IF,H_{k-1}(Q))$

  terminate after k or fewer iterations

  $H_1(Q) = H_0(Q) \vee wp(IF,H_0(Q))$

Concurrent programming tradition

- Note: $\vdash$ is logical deduction while $\models$ is satisfiability in this model

Owiki and Gries

the idea is to build upon the sequential programming tradition and add a non-interference requirement

- ⟨ atomic statement ⟩
- **co** s_1 // ... // s_n **oc**

- $\{P \wedge B\}\ s\ \{Q\} \vdash \{P\}\ \langle \textbf{await}\ B \rightarrow s \rangle\ \{Q\}$ — synchronization rule
  (Note: s may be skip; B may be true)
- critical assertions are assertions that must hold in the sequential program before each atomic statement (programs are annotated with assertions of this type)
- NI(a,C) holds, i.e., assignment a does not interfere with critical assertion C if $\{C \wedge pre(a)\}\ a\ \{C\}$ where pre(a) is the annotation for *a*
- Interference freedom: $\{P_i\}\ s_i\ \{Q_i\}$ are interference-free if for all assignments a in s_i and for all critical assertions C in s_j $(j \neq i)$ NI(a,C) holds
- $\{P_i\}\ s_i\ \{Q_i\}$ are interference-free $\vdash$ $\{\text{AND } P_i\}$ **co** s_1 // ... // s_n **oc** $\{\text{AND } Q_i\}$

Temporal logic (Manna and Pnueli)

computations generate behaviors, sequences of states

- state formulas: given the sequence of states σ, P holds in the j'th state $(\sigma,j) \models P$
- temporal formulas

$(\sigma,j) \models \Theta P$ iff $(\sigma,j+1) \models P$ — next

$(\sigma,j) \models \Box P$ iff for all $k \geq j$ $(\sigma,k) \models P$ — henceforth

$(\sigma,j) \models \Diamond P$ iff for some $k \geq j$ $(\sigma,k) \models P$ — eventually

$(\sigma,j) \models P\ \mathcal{U}\ Q$
iff for some $k \geq j$ $(\sigma,k) \models Q$ and for all $j \leq i < k$ $(\sigma,i) \models P$ — until

$(\sigma,j) \models P\ \mathcal{W}\ Q$ iff $(\sigma,j) \models P\ \mathcal{U}\ Q$ or $(\sigma,j) \models \Box P$ — unless

UNITY Perspective

- understand and manage the complexity of the programming task
- extract what is common to the programming task—avoid focusing on specific languages and architectures, yet accommodate all when necessary
- show that a small theory is adequate for a variety of tasks
  - specification
  - derivation
  - verification
- the elements of the UNITY theory
  - computational model—minimalist
  - proof logic—simple, assertional, avoids operational thinking
  - design heuristics and methodology
- UNITY strategy on proofs
  - extricate the proofs from the text
  - avoid the need to concern oneself with non-interference
  - avoid reasoning about computational histories

- reason about properties which are true in every state

Model

The basic question is *"what is actually fundamental?"*

State

- there are models that focus on states (TLA, IOA, shared variables)
  —logic based reasoning, specification refinement
- there are models that focus on events (CSP, CCS, π-Calculus)
  —algebra of events, composition
- "state transition systems" are common to many fields including computer science, control theory, circuit design, communication theory, operations research, etc.
- Example:

  a* composed with b* produces $(a \vee b)^*$

  a := $\bar{a}$ [] b := $\bar{b}$ may not be as clean, but …

  n, m := n*m, (m-1) if m>0 with n initially set to 1 is rather simple (factorial m!)

Deterministic atomic assignment

- all or nothing effect
- common assumption in many areas
  (database, operating systems, programming languages)
- other options (safe or regular vs atomic) complicate programming and can be simulated if necessary
- various granularities may be considered
- determinate (predictable, unique) effect (function-like behavior) simplifies the theory

Non-deterministic flow of control

- it is intrinsic to many problem areas
- often offers simple abstract solutions

Flow of control constructs are not fundamental

- it is a historical accident
  (Turing Machine, von Neumann computer, flowcharts, structured programming)
- perpetuates sequential programming biases
  (the process concept, e.g., CSP)
- creates an unnecessary tie between modularity (placing related concerns together, having a clean interface, compact abstract specification) and one-way-in/one-way-out flow
- a number of models raised objections about flow of control (dataflow, logic programming)

Synchrony vs asynchrony

- these concepts are at the core of any unified theory
- many instances of both cases: systolic arrays, circuits, multiprocessors, networks

Methodology

- after the fact proofs are too difficult
- emphasis on program derivation
  - formal specification
  - solution strategy leading to refinement (strengthen the specification)
  - target architecture considerations leading to refinement (strengthen the specification)
  - mapping to specific architecture
    - program → WHAT should be done
    - mapping → WHEN, WHERE, and HOW should the assignments be performed or should the program halt
  - assessment of the complexity (time and space) w.r.t. the mapping!!!
- this approach promises to reshape the way we do design