Induction

Mathematical induction (side condition: n not free in Γ):

$$\frac{\Gamma \vdash \underline{P}(0) \quad \Gamma, \underline{P}(n) \vdash \underline{P}(n+1)}{\Gamma \vdash \forall n \cdot \underline{P}(n)} \qquad \frac{\Gamma, (\forall m < n \cdot \underline{P}(m)) \vdash \underline{P}(n)}{\Gamma \vdash \forall n \cdot \underline{P}(n)}$$

The two rules above represent weak and strong induction, respectively.

Structural induction (side condition: xs, x not free in Γ):

$$\frac{\Gamma \vdash \underline{P}([]) \quad \Gamma, \underline{P}(xs) \vdash \underline{P}(x : xs)}{\Gamma \vdash \forall xs \cdot \underline{P}(xs)} \qquad \frac{\Gamma, (\forall xs' \sqsubset xs \cdot \underline{P}(xs')) \vdash \underline{P}(xs)}{\Gamma \vdash \forall xs \cdot \underline{P}(xs)}$$

The two rules above represent weak and strong structural induction, respectively.

The subterm relation for lists, denoted by $\sqsubset$, is defined as follows:

$\forall x, xs \cdot xs \sqsubset x : xs$
$\forall xs, ys, zs \cdot xs \sqsubset ys \wedge ys \sqsubset zs \Rightarrow xs \sqsubset zs$

- Strong structural induction
  - We assume that I.H. holds for all subterms
  1. We prove $\forall xs. P(xs)$ by strong structural induction on xs
  2. We assume, the induction hypothesis holds for all proper subterms of xs
  3. Let ..... be arbitrary
  4. We proceed by a case analysis on xs
  ↪ Für jeden Typ, den xs annehmen kann, müssen wir nun die Gleichheit zeigen
  5. Nun, Terme so umformen, dass wir möglichst viel I.H. für subterms einsetzen können.

- Subterms can be: · Subformulas
  · Subtrees

IMP

n	for numerals (Numeral)
x, y, z	for variables (Var)
e, e', e_1, e_2	for arithmetic expressions (Aexp)
b, b_1, b_2	for boolean expressions (Bexp)
s, s', s_1, s_2	for statements (Stm)

- Numeral -> value
- Variable -> Numeral
  - $\sigma(x)$

- States: · Associates a value with each variable
  · $\sigma(x)$ -> value
- Arithmetic Expression: · $\mathcal{A}[\![x]\!]\sigma = \sigma(x)$
  · $\mathcal{A}[\![n]\!]\sigma = \mathcal{N}[\![n]\!]$
  · $\mathcal{A}[\![e_1 \text{ op } e_2]\!]\sigma$
  $= \mathcal{A}[\![e_1]\!]\sigma \ \overline{op}\ \mathcal{A}[\![e_2]\!]\sigma$
- Boolean Expression:
  · $\mathcal{B}[\![e_1 \text{ op } e_2]\!]\sigma = tt$ if $\mathcal{A}[\![e_1]\!]\sigma \ \overline{op}\ \mathcal{A}[\![e_2]\!]\sigma$
  · $\mathcal{B}[\![b_1 \oplus b_2]\!]$ ($\oplus \in$ and/or)
  · $\mathcal{B}[\![\text{not } b]\!]\sigma = tt$ if $\mathcal{B}[\![b]\!]\sigma = ff$
- Free variables

Arithmetic expressions

$$
\begin{array}{ll}
FV(e_1 \text{ op } e_2) & = FV(e_1) \cup FV(e_2) \\
FV(n) & = \varnothing \\
FV(x) & = \{x\}
\end{array}
$$

Boolean expressions

$$
\begin{array}{ll}
FV(e_1 \text{ op } e_2) & = FV(e_1) \cup FV(e_2) \\
FV(\texttt{not } b) & = FV(b) \\
FV(b_1 \texttt{ or } b_2) & = FV(b_1) \cup FV(b_2) \\
FV(b_1 \texttt{ and } b_2) & = FV(b_1) \cup FV(b_2)
\end{array}
$$

Statements

$$
\begin{array}{ll}
FV(\texttt{skip}) & = \varnothing \\
FV(x\texttt{:=}e) & = \{x\} \cup FV(e) \\
FV(s_1\texttt{;}s_2) & = FV(s_1) \cup FV(s_2) \\
FV(\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end}) & = FV(b) \cup FV(s_1) \cup FV(s_2) \\
FV(\texttt{while } b \texttt{ do } s \texttt{ end}) & = FV(b) \cup FV(s)
\end{array}
$$

- Substitution
  - Replaces each free occurrence of variable x with e
  - $(e_1 \text{ op } e_2)[x \to e] \equiv (e_1[x \to e] \text{ op } e_2[x \to e])$
  - $n[x \to e] \equiv n$
  - $y[x \to e] \equiv \begin{cases} e & \text{if } x \equiv y \\ y & \text{otherwise} \end{cases}$
  - $(\text{not } b)[x \to e] \equiv \text{not } (b[x \to e])$
  - $(b_1 \oplus b_2)[x \to e] \equiv (b_1[x \to e] \oplus b_2[x \to e])$
  - Lemma: $\mathcal{B}[\![b[x \to e]]\!]\sigma \cong \mathcal{B}[\![b]\!]\sigma[x \to \mathcal{A}[\![e]\!]\sigma]$

    first substitute, then evaluate

    first evaluate, then substitute directly with new evaluations
  - ↳ x must be free: $x \in FV(e)$

Operational Semantics

- Describes how the state is modified during the execution of a statement

Big-step semantics

- Transition system $(\Gamma, T, \rightarrow)$ (set of configurations, terminal configurations, transition relation)
- Big-step transitions are of the form $\langle s, \sigma \rangle \rightarrow \sigma'$
- The transition system permits a transition $\langle s, \sigma \rangle \rightarrow \sigma'$, written as $\vdash \langle s, \sigma \rangle \rightarrow \sigma'$, iff there exists a finite derivation tree ending in $root(T) \equiv \langle s, \sigma \rangle \rightarrow \sigma'$
- Beweis zwei Transitions sind gleich
  1. Transition tree für erste Transition bauen
  2. Anderen transition tree so bauen, dass er sich mit Ersten überschneidet
- The big-step semantics of IMP is deterministic
- Non-determinism: $x := 1 \; \square \; (x := 2; \; x := x + 2)$
  - Big-step semantics cannot describe non-terminating computations, if only one non-deterministic branch terminates successfully, we will only see that result
- Parallelism cannot be modeled

Small-step semantics

- Focuses attention on the individual steps of an execution
- k-step execution: $\gamma \to_1^k \gamma'$
  - ↳ There is an execution from γ to γ' in exactly k steps
- $\gamma \to_1^1 \gamma' \triangleq \gamma \to_1 \gamma'$
- Derivation sequence: $\gamma_0, \gamma_1, \ldots$
  - If the derivation sequence is finite, then last configuration is either a terminal or a stuck configuration

---

Difference

- Operational Semantics: - Prove it for specific value
  - Big-step ≙ Natural Semantics
  - Small-step ≙ Structural Sem.
- Axiomatic Semantics: - Prove it for arbitrary values
  - Hoare triple

Program Correctness

- Partial correctness: · If a program terminates then there will be a certain relationship between initial and final state
- Total correctness: partial correctness + termination

Hoare Logic

- $\{P\}\ s\ \{Q\}$ (P = Precondition / Q = Postcondition)
  - 1. If P evaluates to true in an initial state σ
  - 2. If the execution of s from σ terminates in the state σ'
  - 3. Then Q will evaluate to true in σ'
- Logic variables: · Only occur in Pre-/Postcondition
  · Can be used to save values in the initial state, so they can be reffered later in Postcondition
  · $\{x = N\}$ ↑ Logical variable (capital letter)
- Derivation tree: · $\vdash \{P\}\ s\ \{Q\}$
  · There exists a finite derivation tree
  · During proofs, we often need to perform semantic reasoning e.g. applying mathematical properties of arithmetic rules

- $\{P\}$ ← orginal
  ← Hier können wir Precondition strengthen
  $\models \{P'\}$
- $\{Q'\}$ ← stärkere Postcondition
  $\models \{Q\}$ ← orginal

- **While-loop:** $\dfrac{\{b \wedge P\}\ s\ \{P\}}{\{P\}\ \text{while } b \text{ do } s \text{ end } \{\neg b \wedge P\}}$

  ↑ P is the loop invariant, and because it also holds, when the loop terminates

  - finding loop invariant
    - do some loop-executions, write down the states of each execution and try to figure out a pattern

- **Prove termination:** · write as precondition $\{true\}$
  · write as postcondition $\{false\}$
  ↳ If proof goes through then statement is non-terminating

- **Total correctness:** · $\{P\}\ s\ \{\Downarrow Q\}$
  · We only need to change the while-loop: each iteration must decrease the value of the loop-iterator

---

Proof outline

$$\frac{\{b \wedge P\}\ S_1\ \{Q\} \qquad \{\neg b \wedge P\}\ S_2\ \{Q\}}{\{P\}\ \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ end } \{Q\}}\ (\text{If Ax})$$

```
{P}
if b then
{b ∧ P}
S1
{Q}
else
{¬b ∧ P}
S2
{Q}
end
{Q}
```

Assignment:

$\{z+1 \le x\}$	$\{z-1 \le x\}$	$\models \{\dots\}$
$z = z+1$	$z = z-1$	$\{\dots \wedge 1 = 1\}$
		$z = 1$
$\{z \le x\}$	$\{z \le x\}$	$\{\dots \wedge z = 1\}$

$\models \{\dots\}$

Finding invariance

- while $i < k \Rightarrow i \leq k$

Total correctness

while not x = 1 do

{..... ∧ x = z}

⊨

{..... ∧ x - 1 < z}

x := x - 1

{..... ∧ x < z}

what that we prove, that the iterator variable is decreased every iteration

↳ Loop will terminate

Promela

- Input language of spin-model-checker
- constant: #define N 5
- structure: typedef vector {int x; int y}
- Global channel: chan buf = [2] of {int}
- Global variable: byte counter;
- Process: proctype myProc(int p) {...}
- Active processes: active [N] proctype myProc(..){...}
- Variable declarations: int d[3]
- Channel declaration: chan c1 = [2] of {int}

  ↳ can store 2 messages

  chan c2 = [] of {int}

  ↳ Rendez-vous

- Number of states: #program local-states * ∏ variable * #possible values

- assert(E): aborts execution if expression evaluates to 0 (=false)

- if

```
:: x < max x -> x = x + 1;
:: x > min x -> x = x - 1;
fi
```

} Looks for all true options and chooses one non-deterministically

- do

```
:: x < max x -> x = x + 1;
:: x > min x -> x = x - 1;
od
```

} Chooses repeatedly an option non-deterministic, until break/goto is executed

- atomic {s} executes atomically

```
#define N 5

bit Account_locks[N];

inline lock(n) {
  atomic {
    Account_locks[n] == 0;
    Account_locks[n] = 1;
  }
}
```

Blocking, until it is true

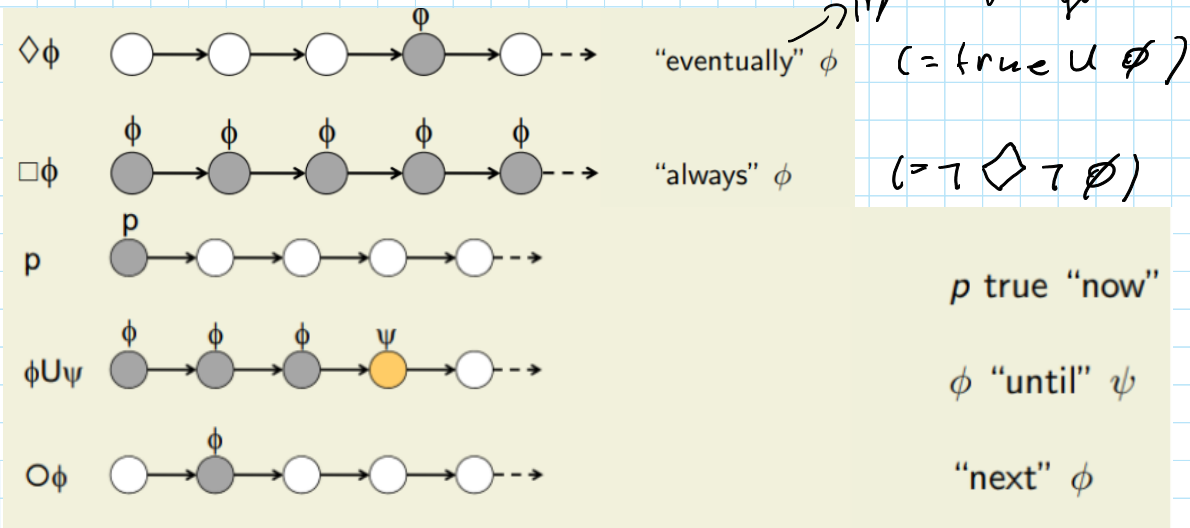
- inline lock(n) can be used everywhere and is later just replaced with the actual code

- Send and Receive channel

  - chan ch = [5] of {mtype, int}
  - Buffered: ch ! req, 7 (sends message)
    ch ? req, n (receives message, if req matches)
  - Unbuffered: Send is only executable, if there is a receive operation that can be executed simultaneously
    ↳ Model synchronous communication

- main: init {...}
- printf("...");
- Own type: mtype = {req, ab}
- Start threads: active [n] proctype myproc (...) {...}

Linear temporal logic



- A file is always open or closed: $\Box$ (open $\vee$ closed)
  (ϕ)
- once ψ is true, then ψ is always true: $\Box(\psi \Rightarrow \Box\psi)$
- Eventually ψ will always hold: $\Diamond\Box\psi$
- Every time ψ holds, ϕ will eventually hold: $\Box(\psi \Rightarrow \Diamond\phi)$

Safety property

- Wenn man einen finite trace (= bad trace) finden kann, der dafür sorgt, dass deine property nie mehr gilt handelt es sich um eine safety property

Liveness property

- Ein infinite trace, die das property erfüllt kann hinten an ein finite trace angehängt werden, und dann erfüllt der neue trace die property