

Reference Sheet for CO141 Reasoning about Programs

Spring 2017

Stylised Proofs for Reasoning

1. Write out and name each given formula.
2. Write out and name each formula to be shown.
3. Plan out the proof and name intermediate results.
4. Justify each step of the proof.

We use the following methods to plan out a proof for P :

1. *Contradiction* i.e. show $\neg P \rightarrow \text{false}$.
2. If $P = Q \wedge R$ show both Q and R .
3. If $P = Q \vee R$ show either Q or R .
4. If $P = Q \rightarrow R$ assume Q and show R .
5. If $P = \neg Q$ show $Q \rightarrow \text{false}$.
6. If $P = \forall x Q(x)$ take arbitrary c and show $Q(c)$.
7. If $P = \exists x Q(x)$ find some c and show $Q(c)$.

We use the following methods to justify our proof:

1. If false holds then P holds.
2. If $Q \wedge R$ holds then Q and R both hold.
3. If $Q \vee R$ holds we do case analysis assuming each in turn.
4. If $Q \rightarrow R$ holds and Q holds then R holds.
5. If $\forall x Q(x)$ holds then $Q(c)$ holds for any c .
6. If $\exists x Q(x)$ holds then $Q(c)$ holds for some c .
7. We can apply any lemma / equivalence given or proven earlier.

1 Reasoning about Haskell Programs

1.1 Mathematical Induction

Principle of Mathematical Induction: For any $P \subseteq \mathbb{N}$:

$$P(0) \wedge \forall k : \mathbb{N}. [P(k) \rightarrow P(k+1)] \rightarrow \forall n : \mathbb{N}. P(n)$$

i.e. to prove by induction, we prove a base case and an inductive step.

General Technique: For any $P \subseteq \mathbb{Z}$ and any $m : \mathbb{Z}$:

$$P(m) \wedge \forall k \geq m. [P(k) \rightarrow P(k+1)] \rightarrow \forall n \geq m. P(n)$$

1.2 Strong Induction

$$P(0) \wedge \forall k : \mathbb{N}. [\forall j \in \{0..k\}. P(j) \rightarrow P(k+1)] \rightarrow \forall n : \mathbb{N}. P(n)$$

Note: for some problems, it may be necessary to split the inductive step into cases. E.g. $k = 0$ or $k \neq 0$.

Mathematical induction and strong induction are *equivalent*.

1.3 Structural Induction over Haskell Data Types

We generalise the concept of predecessor and successor.

Example 1: Structural Induction Principle over Lists:

$$P([]) \wedge \forall \text{vs} : [\text{T}] \forall \text{v} : \text{T}. [P(\text{vs}) \rightarrow P(\text{v} : \text{vs})] \rightarrow \forall \text{xs} : [\text{T}]. P(\text{xs})$$

Example 2: Structural Induction Principle over `Data BExp = Tr | Fl | Bnt BExp | BAnd BExp BExp`:

$$P(\text{Tr}) \wedge P(\text{Fl}) \wedge \forall \text{b} : \text{BExp}. [P(\text{b}) \rightarrow P(\text{Bnt b})] \wedge \\ \forall \text{b1}, \text{b2} : \text{BExp}. [P(\text{b1}) \wedge P(\text{b2}) \rightarrow P(\text{BAnd b1 b2})] \rightarrow \forall \text{b} : \text{BExp}. P(\text{b})$$

Proof Methods

1. Invent an Auxiliary Lemma.
2. Strengthen the original property. E.g. rewrite $\forall i s : [\text{Int}].\text{sum } i s = \text{sum_tr } i s$ as $\forall k : \text{Int} \forall i s : [\text{Int}].k + \text{sum } i s = \text{sum_tr } i s$.

1.4 Induction over Recursively Defined Structures

Sets, relations and functions can be defined inductively, which leads to inductive principles.

Sets *Example:* Consider the set of ordered lists, $OL \subseteq \mathbb{N}^*$:

1. $[] \in OL$
2. $\forall i \in \mathbb{N}. i : [] \in OL$
3. $\forall i, j \in \mathbb{N} \forall j s \in \mathbb{N}^*. [i \leq j \wedge j : j s \in OL \rightarrow i : j : j s \in OL]$

For a property $Q \subseteq \mathbb{N}^*$, we get the inductive principle

$$Q([]) \wedge \forall i \in \mathbb{N}. Q(i : []) \wedge \forall i, j \in \mathbb{N} \forall j s \in \mathbb{N}^*. [i \leq j \wedge j : j s \in OL \wedge Q(j : j s) \rightarrow Q(i : j : j s)] \rightarrow \forall n s \in OL. Q(n s)$$

Relations *Example 1:* Consider the strictly less than relation, $SL \subseteq \mathbb{N} \times \mathbb{N}$:

1. $\forall k \in \mathbb{N}. SL(0, k + 1)$
2. $\forall m, n \in \mathbb{N}. [SL(m, n) \rightarrow SL(m + 1, n + 1)]$

For a property $Q \subseteq \mathbb{N} \times \mathbb{N}$, we get the inductive principle

$$\forall k \in \mathbb{N}. Q(0, k + 1) \wedge \forall m, n \in \mathbb{N}. [SL(m, n) \wedge Q(m, n) \rightarrow Q(m + 1, n + 1)] \rightarrow \forall m, n \in \mathbb{N}. [SL(m, n) \rightarrow Q(m, n)]$$

Example 2: Consider the set of natural numbers, $S_{\mathbb{N}}$:

1. $\text{Zero} \in S_{\mathbb{N}}$
2. $\forall n. [n \in S_{\mathbb{N}} \rightarrow \text{Succ } n \in S_{\mathbb{N}}]$

and the predicate $\text{Odd}(S_{\mathbb{N}})$:

1. $\text{Odd}(\text{Succ Zero})$
2. $\forall n \in S_{\mathbb{N}}. [\text{Odd}(n) \rightarrow \text{Odd}(\text{Succ } (\text{Succ } n))]$

Here it is much simpler to derive the inductive principle from the definition of Odd , rather than from the definition of $S_{\mathbb{N}}$:

$$Q(\text{Succ Zero}) \wedge \forall n \in S_{\mathbb{N}}. [\text{Odd}(n) \wedge Q(n) \rightarrow Q(\text{Succ } (\text{Succ } n))] \rightarrow \forall n \in S_{\mathbb{N}}. [\text{Odd}(n) \rightarrow Q(n)]$$

Functions *Example:* Consider the Haskell function:

```
DM (i, j) = DM' (i, j, 0, 0)
DM' (i, j, cnt, acc)
  | acc + j > i = (cnt, i - acc)
  | otherwise = DM' (i, j, cnt + 1, acc + j)
```

We can define this inductively as follows:

1. $\forall i, j \in \mathbb{Z}. [\text{DM}(i, j) = \text{DM}'(i, j, 0, 0)]$
2. $\forall i, j, cnt, acc \in \mathbb{Z}. [acc + j > i \rightarrow \text{DM}'(i, j, cnt, acc) = (cnt, i - acc)]$
3. $\forall i, j, cnt, acc \in \mathbb{Z}. [acc + j \leq i \wedge \text{DM}'(i, j, cnt + 1, acc + j) = (k1, k2) \rightarrow \text{DM}'(i, j, cnt, acc) = (k1, k2)]$

For a predicate $Q \in \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$, we get the following inductive principle for DM' :

$$\forall i, j, cnt, acc \in \mathbb{Z}. [acc + j > i \rightarrow Q(i, j, cnt, acc, cnt, i - acc)] \wedge \forall i, j, cnt, acc, k1, k2 \in \mathbb{Z}. [acc + j \leq i \wedge \text{DM}'(i, j, cnt + 1, acc + j) = (k1, k2) \wedge Q(i, j, cnt + 1, acc + j, k1, k2) \rightarrow Q(i, j, cnt, acc, k1, k2)] \rightarrow \forall i, j, cnt, acc, k1, k2 : \mathbb{Z}. [\text{DM}'(i, j, cnt, acc) = (k1, k2) \rightarrow Q(i, j, cnt, acc, k1, k2)]$$

2 Reasoning about Java Programs

2.1 Program Specifications

Pre-Conditions, Mid-Conditions, Post-Conditions

1. *Pre-condition:* Must be proven in order to call function, an assumption that code in method can make.
2. *Mid-condition:* Assumption made at specific point in code, must be guaranteed by preceeding code and can be assumed by subsequent code.
3. *Post-condition:* Expected to hold after the code has been executed (assuming termination and that precondition held).

Example: Consider the Java code:

```
type method(type  $x_1, \dots, \text{type } x_n$ )
// PRE:  $P(x_1, \dots, x_n)$ 
// POST:  $Q(x_1, \dots, x_n)$ 
{
    code1
    // MID:  $R(x_1, \dots, x_n)$ 
    code2
    // MID:  $S(x_1, \dots, x_n)$ 
    code3
}
```

Note: if we choose to introduce new (value) variables in our conditions, there is an implicit universal quantification over the whole specification.

Here we need to prove:

1. $P(x_1, \dots, x_n) \wedge \text{code1} \rightarrow R(x_1, \dots, x_n)$
2. $R(x_1, \dots, x_n) \wedge \text{code2} \rightarrow S(x_1, \dots, x_n)$
3. $S(x_1, \dots, x_n) \wedge \text{code3} \rightarrow Q(x_1, \dots, x_n)$

Program Variables

1. x refers to the value of x before code is executed.
2. x' refers to its value after code is executed, *shouldn't be present in assertions*.
3. x_0 refers to its original value, as passed into the method.

We use r to refer to the return value of a method.

Arrays

1. $a \sim b$ means a is a permutation of b .
2. $a \approx b$ means a is identical to b .
3. $a[x..y)$ means the elements of a from index x up to (but not including) y .
4. $\text{Sorted}(a)$ means a is sorted.
5. $\text{min}(a)$ is the smallest element in a .
6. $\text{max}(a)$ is the largest element in a .

2.2 Conditional Branches

We can assume the pre-condition and the `if else` condition. Have to show post-condition holds on both branches of the code.

Example: Consider the Java code:

```
// PRE: true
if (x >= y) {
    // MID:  $x_0 \geq y_0$ 
    res = x;
    // MID:  $\text{res} = x_0 \wedge x_0 \geq y_0$ 
} else {
    // MID:  $y_0 > x_0$ 
    res = y;
    // MID:  $\text{res} = y_0 \wedge y_0 > x_0$ 
}
// MID:  $\text{res} = \max\{x_0, y_0\}$ 
```

2.3 Recursion

Method Calls Need to show that the precondition is met before the method call, then can assume postcondition will hold afterwards. We make necessary substitutions in order to prove our assertions.

Example: Consider the java method:

```
1 int sumAux(int[] a, int i)
2 // PRE:  $a \neq \text{null} \wedge 0 \leq i \leq a.length$ 
3 // POST:  $a \approx a_0 \wedge r = \sum a[i..a.length)$ 
4 {
5     if (i == a.length) {
6         // MID:  $a \approx a_0 \wedge i = a.length$ 
7         return 0;
8     } else {
9         // MID:  $a \approx a_0 \wedge a \neq \text{null} \wedge 0 \leq i < a.length$ 
10        int val = a[i] + sumAux(a, i+1);
11        // MID:  $a \approx a_0 \wedge \text{val} = a[i..a.length)$ 
12        return val;
13    }
14 }
```

We need to prove:

1. *Line 6:* Show mid-condition holds: $a_0 \neq \text{null} \wedge 0 \leq i \leq a_0.length \wedge i = a_0.length \wedge a' \approx a_0 \rightarrow a' \approx a_0 \wedge i = a'.length$.
2. *Line 7:* Show post-condition holds: $a \approx a_0 \wedge i = a.length \wedge r = 0 \rightarrow a \approx a_0 \wedge r = \sum a[i..a.length)$.

3. *Line 9:* Show mid-condition holds: $a_0 \neq \text{null} \wedge 0 \leq i \leq a_0.\text{length} \wedge i \neq a'.\text{length} \wedge a' \approx a_0 \rightarrow a' \approx a_0 \wedge a' \neq \text{null} \wedge 0 \leq i < a'.\text{length}$.
4. *Line 10:* Show pre-condition for called method holds: $a \approx a_0 \wedge a \neq \text{null} \wedge 0 \leq i < a.\text{length} \rightarrow a \neq \text{null} \wedge 0 \leq i + 1 \leq a.\text{length}$.
5. *Line 11:* Show mid-condition holds: $a \approx a_0 \wedge a \neq \text{null} \wedge 0 \leq i < a.\text{length} \wedge a' \approx a \wedge r = \sum a'[i+1..a'.\text{length}] \wedge \text{val}' = a[i] + r \rightarrow a' \approx a_0 \wedge \text{val}' = \sum a'[i..a'.\text{length}]$.
6. *Line 12:* Show post-condition holds: $a \approx a_0 \wedge \text{val} = \sum a[i..a.\text{length}] \wedge r = \text{val} \rightarrow a \approx a_0 \wedge r = \sum a[i..a.\text{length}]$.

Blue statements come from the **pre-condition or previous mid-condition**, green statements **implicitly from code**, red statements **explicitly from code** and purple statements from the **post-condition of a called method**.

2.4 Iteration

Invariant To prove a property holds throughout the loop, we need to prove that the *invariant* holds before entering the loop, and is preserved by the loop body (including at termination). The invariant and $\neg \text{cond}$ can be used to prove the following mid-condition.

Variante To prove a loop will terminate, we find an integer expression which is bounded below, and decreases in *every* loop iteration.

Example: Consider the java method:

```

1  int culSum(int[] a)
2  // PRE P: a ≠ null
3  // POST Q: a.length = a0.length ∧ r = ∑ a0[0..a.length) ∧
4             ∀k ∈ [0..a.length). [a[k] = ∑ a0[0..k+1)]
5  {
6      int res = 0;
7      int i = 0;
8      // INV I: a ≠ null ∧ a.length = a0.length ∧ 0 ≤ i ≤ a.length ∧
9              res = ∑ a0[0..i) ∧ ∀k ∈ [0..i). [a[k] = ∑ a0[0..k+1)] ∧
10             ∀k ∈ [i..a.length). [a[k] = a0[k]]
11      // VAR V: a.length - i
12      while (i < a.length) {
13          res = res + a[i];
14          a[i] = res; i++;
15      }
16      // MID M: a.length = a0.length ∧ res = ∑ a0[0..a.length) ∧
17              ∀k ∈ [0..a.length). [a[k] = ∑ a0[0..k+1)]
18      return res;
19  }
```

We need to prove:

1. Invariant holds before loop is entered.

$$P[a \mapsto a_0] \wedge \text{res} = 0 \wedge i = 0 \wedge a \approx a_0 \rightarrow I$$

2. Loop body re-establishes invariant.

$$I \wedge i < a.\text{length} \wedge \text{res}' = \text{res} + a[i] \wedge a'[i] = \text{res}' \wedge i' = i + 1 \wedge \forall k \in [0..a.\text{length}) \setminus \{i\}. [a'[k] = a[k]] \rightarrow I[a \mapsto a', i \mapsto i', \text{res} \mapsto \text{res}']$$

3. Mid-condition holds straight after loop.

$$I \wedge i \geq a.\text{length} \rightarrow M$$

4. Loop terminates.

$$I \wedge i < a.\text{length} \wedge \text{res}' = \text{res} + a[i] \wedge a'[i] = \text{res}' \wedge i' = i + 1 \wedge \forall k \in [0..a.\text{length}) \setminus \{i\}. [a'[k] = a[k]] \rightarrow V \geq 0 \wedge V[a \mapsto a', i \mapsto i', \text{res} \mapsto \text{res}'] < V$$

5. Post-condition established.

$$M \wedge r = \text{res} \rightarrow Q$$

6. Array accesses are legal.

$$I \wedge i < a.\text{length} \rightarrow 0 \leq i < a.\text{length}$$