Reasoning Notes

Hashan Punchihewa

https://hashanp.xyz

Contents

Chapter 1

Reasoning about Programs

1.1 Stylised Proofs

Here is an example of a proof schema:

- Give each a reason for each deduction
- If you have to show $A \to B$, then A can become a given.
- If you have to show $A \wedge B$, then A and B can become two different things to show.
- If you do something using arithmetic, say 'by arithmetic'

Proofs are essentially written in a looser form of natural deduction but with the same reasoning rules:

Rule	Proposition	How to prove
$\wedge I$	$P = Q \wedge R$	Prove Q and then prove R
$\vee I$	$P = Q \vee R$	Prove Q or prove R
$\to I$	$P = Q \to R$	Assume Q and prove R
$\neg I$	$P = \neg Q$	Show $Q \to \bot$
$\forall I$	$P = \forall x\ Q(x)$	Take arbitrary c and show $Q(c)$
$\exists I$	$P = \exists x\ Q(x)$	Find some c and show $Q(c)$
PC		Show $\neg P \to \bot$

Rule	Proposition	Having proven
$\wedge E$	$P = Q \wedge R$	Q and R hold
$\vee I$	$P = Q \vee R$	Case Analysis
$\to E$	$P = Q \to R \wedge Q$	R holds
$\neg E$	$P = \neg Q$	$Q \to \bot$
$\bot E$	$P = \bot$	Any Q holds
$\forall E$	$P = \forall x\ Q(x)$	$Q(c)$ for all c
$\exists E$	$P = \exists x\ Q(x)$	Let c be the x such that $Q(x)$ holds
LEM		Use lemmata

1.2 Reasoning about Haskell

Let $P(x)$ be **some proposition**. We will prove $\forall x \in \mathbf{S}\ P(x)$ by **mathematical / strong / structural** induction (over **S** if structural).

Base Case:
(Take arbitrary **something** if necessary)
To Show: **Something**
Proof:
Justification

Inductive Case:
Take arbitrary **something**
Inductive Hypothesis (IH): **Something**
To Show: **Something**
Proof:
Justification

Notes:

- Label the inductive hypothesis explicitly as an inductive hypothesis, not just another assumption.
- Use different letters for your inductive hypothesis, and the variables you take arbitrarily. This is so when you apply your inductive hypothesis it is clear what you are doing.
- When applying definitions, state which direction you are applying the definitions.
- An example of where you have to take variables arbitrarily in the base case, is if you were doing structural induction over a tree data structure with values `Leaf Int`, so you have to take arbitrarily over the integers.
- If asked to show something will terminate in Haskell, think induction and try and find some arithmetic expression to do induction on.
- When writing out 'to show' and the 'inductive hypothesis', always write it out in full.

Definition 1.2.1

- *Principle of Mathematical Induction:*

$$P(0) \land \forall k : \mathbb{N}\ [P(k) \to P(k+1)] \to \forall n : \mathbb{N}\ P(n)$$

- *For all $P \subseteq \mathbb{Z}$ and $m \in \mathbb{Z}$:*

$$P(m) \land \forall k \geq m\ [P(k) \to P(k+1)] \to \forall n \geq m\ P(n)$$

- *Principle of Strong Induction:*

$$P(0) \land \forall k : \mathbb{N}\ [(\forall j \in \{0 \ldots k\}\ P(j)) \to P(k+1)] \to \forall n : \mathbb{N}\ P(n)$$

Note that the principal of mathematical induction and the principle of strong induction are equivalent, and can be proven from each other. Also note, strong induction may necessitate breaking the inductive case into stages, e.g. considering $k = 0$ and $k \neq 0$ separately.

The principles of induction can be generalised to apply to Haskell data structures, i.e. structural induction and Haskell functions. For instance, consider a binary tree:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

This results in the following inductive principle:

$$\forall \texttt{x} : \texttt{a}\ P(\texttt{Leaf x}) \land (\forall \texttt{x}, \texttt{y} : \texttt{Tree a}\ P(x) \land P(y) \to P(\texttt{Node x y})) \to \forall \texttt{t} : \texttt{Tree a}\ P(t)$$

This can be generalised for any number of base cases, any number of inductive cases, and regardless of whether the data type is polymorphic. If the inductive hypothesis isn't strong enough, at first, there are two things that can be done: strengthen the hypothesis; take advantage of an auxiliary lemma.

More generally, sets can be defined inductively. E.g. by the rules:

1. $\texttt{Zero} \in S$
2. $\forall \texttt{n}\ [\texttt{n} \in S \to \texttt{Succ n} \in S]$

Sets that are too complex to be defined using Haskell data types can be defined in this way:

1. $\texttt{[]} \in S$
2. $\forall \texttt{i} \in \mathbb{N}\ [\texttt{i:[]} \in S]$
3. $\forall \texttt{i}, \texttt{j} \in \mathbb{N}, \texttt{js} \in [\mathbb{N}]\ [i \leq j \land \texttt{j:js} \in S \to \texttt{i:j:js} \in S]$

This gives the inductive principle:

$$\begin{aligned} & P(\texttt{[]}) \\ \land & (\forall \texttt{i} \in \mathbb{N}\ P(\texttt{i:[]})) \\ \land & (\forall \texttt{i}, \texttt{j} \in \mathbb{N}, \texttt{js} \in [\mathbb{N}]\ [i \leq j \land \texttt{j:js} \in S \land P(\texttt{j:js}) \to P(\texttt{i:j:js})]) \\ \to & \forall \texttt{xs} \in S\ [P(\texttt{xs})] \end{aligned}$$

Relations can equally be defined inductively (since relations are just sets):

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

This leads to the following inductive principle:

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

Similarly functions can be defined inductively:

```
f 0 = 0
f i = 1 + f (i - 3)
```

Then this gives us the following inductive hypothesis:

$$\begin{aligned}&Q(0,0)\\ \wedge&[\forall i, j \in \mathbb{N}\ (i \neq 0) \wedge (f\ (i-3) = j) \wedge Q(i-3, j) \rightarrow Q(i, 1+j)]\\ \rightarrow&\forall i, j \in \mathbb{N}\ [f\ i = j \rightarrow Q(i,j)]\end{aligned}$$

Observe that this guards against partiality.

1.3 Reasoning about Java Programs

Proof setup:

Given:
1) **Something**
2) **Something**
3) etc.

To Show:
α) **Something**
β) **Something**
etc.

Proof:
n) **Something** **Justification**
etc.

- Reasoning about Java programs, is based on pre-conditions, mid-conditions and post-conditions of methods.
- Pre-conditions must be established before calling a method, and state what must hold at the point the method is called.
- Mid-conditions state properties that hold at a particular point in a method.

- Post-conditions are established after the method finishes, and states what can be assumed about the behaviour of a method, provided its pre-conditions hold.
- Three sections: *Given*, *To Show*, *Proof*
- Number each given, intermediate result and assumption
- Given each proposition to be shown a (Greek) letter

Thus a method would be described as follows:

```
// PRE: P
code0
// MID: M1
code1
// MID: M2
code2
// POST: Q
```

This leads to the following proof obligations:

- $P \land \texttt{code0} \rightarrow M_1$
- $M_1 \land \texttt{code1} \rightarrow M_2$
- $M_2 \land \texttt{code2} \rightarrow Q$

This can be equally formalised as Hoare triples:

- $\{P\}\ \texttt{code0}\ \{M_1\}$
- $\{M_1\}\ \texttt{code1}\ \{M_2\}$
- $\{M_2\}\ \texttt{code2}\ \{Q\}$

Here are the conventions for variables and assignments:

- x is the value of x before the code is executed.
- x' is the value of x after the code is executed.
- x_0 is the original value of x that was passed to the method.
- $Q[x \rightarrow x']$ refers to Q with x substituted for x'.
- The following axiom deals with assignment $P \land x' = u \rightarrow Q[x \rightarrow x']$ allows us to conclude: $\{P\}\ x = u; \{Q\}$.
- r refers to the return value of a method.
- In all method calls you have to assume parameters are changed. If x is passed as a parameter, and called y in the method, then the substitutions $y_0 \rightarrow x$ and $y \rightarrow x'$ are used.
- Note we don't use the x' notation in specifications; rather the proof obligations are changed.

- Certain things can be said to be implicit from code, e.g. if an array isn't modified since the entry of a method, then $a \approx a_0$ can be written. Other things that can be said to be implicit code could be `a'.length = a.length`.

Typically method specifications will be given as follows:

```
type someMethod(type x_1, ... , type x_n)
// PRE: P(x_1,...,x_n)
// POST: Q(x_1,...,x_n)
{
    // Some code here
}
```

Also note the following notational conventions for arrays:

- $a \sim b$ means a is a permutation of b
- $a \approx b$ means a and b are identical
- $a[x..y)$ refers to the elements of a from x (inclusive) to y (exclusive), which also gives us $\sum a[x..y)$ and $\prod a[x..y)$.
- $\text{Sorted}(a)$ means a is sorted.
- $\min(a)$ refers to the smallest element in a.
- $\max(a)$ refers to the largest element in a.

When accessing array items, it is important that the array isn't `null`, and the index is within the bounds of the array. This means if doing `a[i]` then it must be shown that `a` $\neq$ `null` and $0 \leq i <$ `a.length`.

To deal with if-statements, the if-statement can be split into two separate Hoare triples:

```
// PRE: P
if (cond) {
    code_0
} else {
    code_1
}
// POST: Q
```

This would give the triple $\{P\}$ `if (cond) code`$_0$ `else code`$_1$ $\{Q\}$. But this can be split into $\{P \wedge \texttt{cond}\}$ `code`$_0$ $\{Q\}$ and $\{P \wedge \neg\texttt{cond}\}$ `code`$_1$ $\{Q\}$

In other words:

```
// PRE: P
if (cond) {
    // MID: P ∧ cond
    code_0
    // MID: Q
```

```
} else {
    // MID: P ∧ ¬cond
    code1
    // MID: Q
}
// POST: Q
```

To reason about method calls, you need to make sure, that the precondition is established before the call, so you can establish the post-condition afterwards. To deal with recursion, you can do the same, but it is important to prove termination, as well e.g. by using mathematical induction.

1.4 Reasoning About Loops

To reason about loops, it is necessary to introduce loop invariants and loop variants. Loop invariants hold before entering the loop and after every execution of the loop. Consider the following:

```
// PRE: P
code0
// INV: I
while(cond) {
    code1
}
// POST: Q
```

It is necessary to show the following proof obligations (modified to accommodate any changes in state):

- $P \wedge \texttt{code}_0 \rightarrow I$
- $I \wedge \texttt{cond} \wedge \texttt{code}_1 \rightarrow I$
- $I \wedge \neg\texttt{cond} \rightarrow Q$

In addition, it is necessary to show the the loop terminates, by showing there is a loop variant, i.e. an integer expression that is bounded below, and decreases at every iteration. For instance if the integer expression is a variable x, then there is the following proof obligation:

$$I \wedge \texttt{cond} \wedge \texttt{code}_1 \rightarrow (x' < x)$$

Remember: array accesses must also be shown to be legal. So if you have the array access `a[x]`, then you have the proof obligation that:

$$I \wedge \texttt{cond} \rightarrow (\texttt{a} \neq \texttt{null}) \wedge (0 \leq i < \texttt{a.length})$$

Usually $\texttt{a} \neq \texttt{null}$ is included in the invariant so you only have to show:

$$I \wedge \texttt{cond} \rightarrow (0 \leq i < \texttt{a.length})$$

In short, the five things proof obligations to remember: beginning, middle, end, variant and array accesses. For the course, we assume `int` isn't bounded.