COMS22202: 2015/16

Language Engineering

Dr Oliver Ray
(csxor@Bristol.ac.uk)

Department of Computer Science
University of Bristol

Tuesday 12th April, 2016

Assignment Axiom (Hoare)

"Assignment is undoubtedly the most characteristic feature of programming a digital computer, and one that most clearly distinguishes it from other branches of mathematics.

It is surprising therefore that the axiom governing our reasoning about assignment is quite as simple as any to be found in elementary logic."

CAR. Hoare - An Axiomatic Basis for Computer Programming Comm. ACM 12(10):576-580, 1969

$$\frac{}{\{P(a)\}\ \texttt{x:=a}\ \{P(x)\}} \quad \equiv \quad \frac{}{\{P[x\mapsto a]\}\ \texttt{x:=a}\ \{P\}}$$

Note: these are just two different ways of writing exactly the same axiom schema; but, while the first is lighter on notation, the second allows to more naturally denote the symbolic result of combining multiple assignment statements by composition - i.e. $\{P[z\mapsto c]\ldots[y\mapsto b][x\mapsto a]\}$ `x:=a; y:=b;` … `z=c` $\{P\}$

Assignment Axiom (Floyd)

“This paper attempts to provide an adequate basis for formal definitions of the meanings of programs in appropriately defined programming languages, in such a way that a rigorous standard is established for proofs about computer programs.”

R.W. Floyd - Assigning meanings to programs.
Proc. Symposia in Applied Mathematics 19:19-32, 1967

$$\frac{}{\{P\}\ x:=a\ \{\exists n.P[x\mapsto n]\ \wedge\ x=a[x\mapsto n]\}}$$

where n is a fresh variable that does not appear in P, x or a

Note: before Hoare introduced his “backward” assignment rule (which computes the precondition from the postcondition), logicians used this “forward” rule (which does the opposite); here n denotes the old value of x (the precise identity of which is lost after the assignment)

Alternative Assignment Axioms

$$\frac{}{\{P(a)\}\ x:=a\ \{P(x)\}}$$

backwards

$$\frac{}{\{P\}\ x:=a\ \{\exists n.P[x\mapsto n]\ \wedge\ x=a[x\mapsto n]\}}$$

forwards

$$\frac{}{\{P(x)\ \wedge\ x=n\}\ x:=a\ \{P(n)\ \wedge\ x=a[x\mapsto n]\}}$$

combined

Note: many alternatives are possible and, although all these can be shown to be logically equivalent (in the context of the other rules), it is definitely the case that the backward rule is by far the easiest to use in practice.

Note: in (more) general (than required on this unit) the variable substitution operation only applies to *free* variables and renames *bound* variables to avoid capture – so $(\exists x(x=y))[y\mapsto x]$ is $\exists z(z=x)$.

Examples: backwards rule

inc

$$\frac{\dfrac{}{\{x+1=2\}\ x:=x+1\ \{x=2\}}}{\{x=1\}\ x:=x+1\ \{x=2\}}$$

as x=1 |= x+1=1+1=2

swap

$$\begin{tikzcd}
\{x=m \wedge y=n\} \\
z:=x \\
\{z=m \wedge y=n\} \arrow[uu, bend left] \\
x:=y \\
\{z=m \wedge x=n\} \arrow[uu, bend left] \\
y:=z \\
\{y=m \wedge x=n\} \arrow[uu, bend left]
\end{tikzcd}$$

Examples: forwards rule

inc

$$\dfrac{\dfrac{}{\{x=1\}\ x:=x+1\ \{\exists n(n=1 \wedge x=n+1)\}}}{\{x=1\}\ x:=x+1\ \{x=2\}}$$

using ∃n(n=1 ^ x=n+1) |= x=1+1=2

swap

{x=m ^ y=n}
z:=x
{∃i(x=m ^ y=n ^ z=x)}
x:=y
{∃j(∃i(j=m ^ y=n ^ z=j) ^ x=y)}
y:=z
{∃k(∃j(∃i(j=m^k=n^z=j)^x=k)^y=z)}
{∃ijk(j=m=z=y ^ k=n=x)}
{m=z=y ^ n=x)}
{y=m ^ x=n}

Examples: forwards rule (with cheat!)

inc

$$\frac{\overline{\{x=1\}\ x:=x+1\ \{\exists n(n=1 \wedge x=n+1)\}}}{\{x=1\}\ x:=x+1\ \{x=2\}}$$

using ∃n(n=1 ^ x=n+1) |= x=1+1=2

swap

{x=m ^ y=n}
z:=x
{∃i (x=m ^ y=n ^ z=x)}
{z=m ^ y=n}
x:=y
{∃j (z=m ^ y=n ^ x=y)}
{z=m ^ x=n}
y:=z
{∃k (z=m ^ x=n ^ y=z)}
{y=m ^ x=n}

Note: we can use the conditions from the backwards proof to simplify the forward conditions as we go along – but, if you need to do that, then you might as well use the backwards rule in the first place!

Examples: combined rule

inc

$$\frac{\overline{\{x=1 \wedge x=1\}\ x:=x+1\ \{n=1 \wedge x=n+1\}}}{\{x=1\}\ x:=x+1\ \{x=2\}}$$

swap

{x=m ^ y=n}
{x=m ^ y=n ^ z=i}
z:=x
{x=m ^ y=n ^ z=x}
x:=y
{y=n ^ z=m ^ x=y}
{y:=z
{z=m ^ x=n ^ y=z}
{y=m ^ x=n}

Forward |= Backward

Writing P as $P[x \mapsto a]$ in the forward (Floyd) rule gives

$\{ P[x \mapsto a] \}\ \ x{:=}a\ \ \{\exists n.\ (P[x \mapsto a])[x \mapsto n]\ \wedge\ x=a[x \mapsto n]\ \}$

where n is a new variable that does not appear in P, x or a

$\{ P[x \mapsto a] \}\ \ x{:=}a\ \ \{\exists n.\ P[x \mapsto a[x \mapsto n]]\ \wedge\ x=a[x \mapsto n]\ \}$

$\{ P[x \mapsto a] \}\ \ x{:=}a\ \ \{\exists n.\ P[x \mapsto x]\ \}$

$\{ P[x \mapsto a] \}\ \ x{:=}a\ \ \{\exists n.\ P\ \}$

$\{ P[x \mapsto a] \}\ \ x{:=}a\ \ \{P\ \}$

which gives the backward (Hoare) rule

Backward |= Forward

Writing P as $\exists n.P[x\mapsto n] \wedge x=a[x\mapsto n]$ in the backward (Hoare) rule

$\{ \exists n.P[x\mapsto n] \wedge a=a[x\mapsto n] \}\ \ x:=a\ \ \{ \exists n.P[x\mapsto n] \wedge x=a[x\mapsto n] \}$

(where n is a new variable that does not appear in P, x or a)

so the only remaining occurrence of x on the rhs is shown in red

but $P \models P \wedge a=a \models P[x\mapsto x] \wedge a=a[x\mapsto x]$
$\models \exists n.P[x\mapsto n] \wedge a=a[x\mapsto n]$

as we can simply set n=x so, by cons, we have

$\{ P \}\ \ x:=a\ \ \{ \exists n.P[x\mapsto n] \wedge x=a[x\mapsto n] \}$

which gives the forward (Floyd) rule

Backward is Best!

Although the alternative rules are logically equivalent, it is much easier to reason backwards in practice as you only need to write down statements that actually contribute to the property you are trying to prove. The backwards rule facilitates this by avoiding having to write down properties or variable that you are not interested in (and then removing them with the consequence rule). Although programs run forwards it is best to reason about them backwards, guided by the property you are trying to prove. By working systematically backwards the only decisions you need to make concern loop invariants and when to optionally simplify the precondition of a composition of assignments. This approach involves less writing and less thinking!

USE THE BACKWARD ASSIGMENT RULE UNLESS TOLD OTHERWISE!

Read and Write

Imagine we had two program variables IN and OUT that denote lists of strings/integers (either directly or using some sort of integer representation) and we had functions head, tail, init, last, etc that perform list manipulations on these.

If we use IN to represent a stream of integers that can be read by the program and OUT to represent a stream of strings/integers output by the program then we can model read and write(ln) statements as follows:

- read x ≡ x:=head(IN); IN:=tail(IN)
- write e ≡ OUT:= append(OUT,[e])
- writeln ≡ write '\n'

Read and Write

Then we can derive the following axioms:

	{ (P[IN↦tail(IN)])[x↦head(IN)] }
	x:=head(IN)
read x	; { P[IN↦tail(IN)] }
	IN:=tail(IN)
	{ P }
	{ (P[OUT↦append(OUT,[e])] }
write e	OUT:= append(OUT,[e])
	{ P }
	{ (P[OUT↦append(OUT,['\n'])] }
writeln	write '\n'
	{ P }

Example

```
    { IN=[1..100] ^ OUT=[ ] }
|=
    { OUT++[head(IN)]++tail(IN)=[1..100] }
    read x;
    { OUT++[x]++IN=[1..100] }
    while !(x=100) do (
      { OUT++[x]++IN=[1..100] ^ !(x=100)}
|=
      { OUT++[x]++[head(IN)]++tail(IN)=[1..100] }
      write x;
      { OUT++[head(IN)]++tail(IN)=[1..100] }
      read x
      { OUT++[x]++IN=[1..100] }
    )
    { OUT++[x]++IN=[1..100] ^ !!(x=100)}
|=
    { IN=[ ] ^ OUT=[1..99] }
```

Another Example

{ head(IN) = n }

read x;

{ x = n }

{ append(OUT,[x]) = append(OUT,[n]) }

{ append(OUT,[x]) = append(_, [n]) }

write x;

{ OUT = append(_, [n]) }

{append(OUT,['\n']) = append(append(_, [n]),['\n'])
= append(_, [n,'\n']) }

{ append(OUT,['\n']) = append(_, [n, _]) }

writeln

{ OUT = append(_, [n, _]) }

∃xy . OUT=append(x,[n,y])

n=last(init(OUT))