7 Weakest precondition semantics

In axiomatic semantics the formulae take the form $\{P\}S\{Q\}$. For a given postcondition, Q, and a given statement, S, there are many preconditions, P, that make the formula true or provable. For the **While** language there is always a weakest precondition with this property. For the assignment statement this precondition is present in

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

We shall define the semantics of **While** using such weakest preconditions. We define the semantic function

$$\mathcal{S}_{\text{wp}} \in \textbf{Stm} \to \textbf{Lexp} \to \textbf{Lexp}$$

where **Lexp** is a set of logical expressions. We do not specify the grammar and will freely use any logical expression containing the arithmetic and boolean expressions of **While** as well as standard mathematical expressions of numerical and boolean. There are two disjoint sets of variables, **Var** with programming variables, and **Const** with "mathematical" variables. A mathematical variable has a value that cannot be changed by a program, hence the name **Const**. Formally we consider an element of **Lexp** to be a function that, given a state and an 'assignment' for all (free) mathematical variables, will return a boolean value. In the concrete representation we just use the expression without indicating that it is a function and we may freely change the representation as long as it doesn't change the function.

In particular we will use $\forall n \in S\ .\ e$ and $\exists n \in S\ .\ e$ where $n \in$ **Const**, S is a set, and e is a logical expression in **Lexp**. e will usually contain free occurrences of n. $\forall n \in S\ .\ e$ will be true iff e is true when n is replaced by all values in S. An example $\forall n \in \mathrm{N}\ .\ (n+1)^2 = n^2 + 2n + 1$.

$\exists n \in S\ .\ e$ will be true iff e is true if n is replaced by at least one value in S. An example $\exists n \in \mathrm{N}\ .\ (n+1)^2 = 25$. When there is no doubt about which set n should belong to this information may be omitted, e.g. $\exists n\ .\ (n+1)^2 = 25$.

We may perform substitutions in **Lexp** using the notation $e[x \mapsto a]$ where $x \in$ **Var** and $a \in$ **Aexp**. Again we omit the formal definition. It should be noted that n in $\forall n \in S\ .\ e$ is a bound variable and substitution may require name changes as in the λ-calculus.

The semantic function is defined over the structure of **Stm**.

$$\mathcal{S}_{\text{wp}}[\![x := a]\!]P \triangleq P[x \mapsto a]$$

$$\mathcal{S}_{\text{wp}}[\![\texttt{skip}]\!]P \triangleq P$$

$$\mathcal{S}_{\text{wp}}[\![S_1; S_2]\!]P \triangleq \mathcal{S}_{\text{wp}}[\![S_1]\!](\mathcal{S}_{\text{wp}}[\![S_2]\!]P)$$

$$\mathcal{S}_{\text{wp}}[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]P \triangleq (b \Rightarrow \mathcal{S}_{\text{wp}}[\![S_1]\!]P) \wedge (\neg b \Rightarrow \mathcal{S}_{\text{wp}}[\![S_2]\!]P)$$

$$\mathcal{S}_{\text{wp}}[\![\texttt{while}\ b\ \texttt{do}\ S]\!]P \triangleq \exists n \in \mathrm{N}\ .\ W_n(P)$$

$$\text{where } W_0(P) \triangleq \neg b \wedge P$$

$$W_{n+1}(P) \triangleq W_0(P) \vee \mathcal{S}_{\text{wp}}[\![\texttt{if}\ b\ \texttt{then}\ S\ \texttt{else skip}]\!](W_n(P))$$

The first two lines require no explanations. The case for composition is intuitive. To find the weakest precondition for $S_1; S_2$ to establish P we first compute the weakest precondition for S_2 to establish P, i.e. $\mathcal{S}_{\text{wp}}[\![S_2]\!]P$. This must in turn be established by S_1, so the execution must start in a state where $\mathcal{S}_{\text{wp}}[\![S_1]\!](\mathcal{S}_{\text{wp}}[\![S_2]\!]P)$ holds.

Exempel.

$$
\begin{aligned}
&\mathcal{S}_{\text{wp}}[\![\texttt{x:=x+1; x:=x*x}]\!](\mathtt{x} = 4) = \\
&\mathcal{S}_{\text{wp}}[\![\texttt{x:=x+1}]\!](\mathcal{S}_{\text{wp}}[\![\texttt{x:=x*x}]\!](\mathtt{x} = 4)) = \\
&\mathcal{S}_{\text{wp}}[\![\texttt{x:=x+1}]\!](\mathtt{x} * \mathtt{x} = 4) = \\
&((\mathtt{x} + \mathtt{1}) * (\mathtt{x} + \mathtt{1}) = 4) = \\
&\mathtt{x} = 1 \lor \mathtt{x} = -3
\end{aligned}
$$

$\Diamond$

The definition for `if` statement looks natural. If b is true then S_1 must establish P which it will do with the precondition $\mathcal{S}_{\text{wp}}[\![S_1]\!]P$ and if b is false then S_2 must establish P.

Exempel.

$$
\begin{aligned}
&\mathcal{S}_{\text{wp}}[\![\texttt{if 0}{\leq}\texttt{x then y:=x else y:=-x}]\!](\mathtt{y} = |n|) = \\
&(0 \leq \mathtt{x} \Rightarrow \mathcal{S}_{\text{wp}}[\![\texttt{y:=x}]\!](\mathtt{y} = |n|)) \land (\neg(0 \leq \mathtt{x}) \Rightarrow \mathcal{S}_{\text{wp}}[\![\texttt{y:=-x}]\!](\mathtt{y} = |n|)) \\
&(0 \leq \mathtt{x} \Rightarrow \mathtt{x} = |n|) \land (\neg(0 \leq \mathtt{x}) \Rightarrow -\mathtt{x} = |n|) = \\
&(|\mathtt{x}| = |n|)
\end{aligned}
$$

$\Diamond$

The definition for the `while` statement requires some explanation. If we start the execution of `while` b `do` S in a state where $W_0 = \neg b \land P$ is true then statement will terminate without executing S in a state where P will hold. It can be proved by induction that

$$W_n(P) = \mathcal{S}_{\text{wp}}[\![IF; IF; ...; IF]\!](\neg b \land P), \qquad \text{with } n \text{ repetitions}$$

where $IF \triangleq$ `if` b `then` S `else skip`. We understand that $W_n(P)$ is the weakest precondition that will make the `while` statement establish P in at most n iterations. Finally $\exists n \in \mathrm{N} \,.\, W_n(P)$ will be true if the `while` statement establish P in a finite number of iterations. In fact, $\exists n \in \mathrm{N} \,.\, W_n(P)$ will be equal to $W_k(P)$, where k is the smallest number such that $W_k(P)$ is true, when such a k exists.

We observe that $\mathcal{S}_{\text{wp}}[\![]\!]$ specifies total correctness; if the precondition is satisfied then the execution will terminate.

Sats. $\mathcal{S}_{\text{wp}}[\![S]\!]\text{ff} = \text{ff}$ for all $S \in \textbf{Stm}$. ■

Proof. (Exercise). The proof is by induction over the structure of S. To justify this proof method the definition of $\mathcal{S}_{\text{wp}}[\![S]\!]$ must be compositional. Thus the definition of $W_{n+1}(P)$ must be changed

$$W_{n+1}(P) \triangleq W_0(P) \lor (b \Rightarrow \mathcal{S}_{\text{wp}}[\![S]\!](W_n(P)) \land (\neg b \Rightarrow W_n(P)))$$

□

Exempel. From the previous theorem it follows that there is no statement in **While** with the property $\mathcal{S}_{\text{wp}}[\![S]\!]P = \text{tt}$ for all P. If such a statement existed it would put all programmers out of work; it could be used to establish any predicate, even $\mathtt{x} = \mathtt{x} + \mathtt{1}$.
$\Diamond$

The function $\mathcal{S}_{\text{wp}}[\![S]\!]$ is monotone when we use implication, $\Rightarrow$, as the ordering relation.

> **Sats.** If $P \Rightarrow Q$ then $\mathcal{S}_{\text{wp}}[\![S]\!]P \Rightarrow \mathcal{S}_{\text{wp}}[\![S]\!]Q$ for all $S \in \mathbf{Stm}$. $\blacksquare$

> **Sats.** If $P \Rightarrow \mathcal{S}_{\text{wp}}[\![S]\!]R$ then $\{P\}\ S\ \{R\}$. $\blacksquare$

In the present setting an *invariant* for `while` b `do` S is a predicate, P, satisfying $b \wedge P \Rightarrow \mathcal{S}_{\text{wp}}[\![S]\!]P$.

7.1 Finding invariants

In order to compute something that couldn't be computed by hand repetition or recursion is needed. In order to prove something defined by recursion an inductive assumption is needed and for a repetition an invariant plays a similar role.

If a program is constructed without considering the possibility to prove (in principle) that the program satisfies its specification then it will be very hard or impossible to actually prove it. Actually thinking about invariants and formally defining them will improve program quality even if a proof is not constructed.

When a programmer wants to establish some postcondition, P, and a `while` statement may solve the problem, P itself cannot be an invariant since it will not be satisfied before the execution of the while statement. A natural way to find an invariant is to find a weaker condition than P.

If the postcondition is a conjunction, $P = P_1 \wedge P_2$, then P_1 may be a suitable invariant and the negation of P_2 should be used to construct b in `while` b `do` S.

> **Exempel.** Suppose that we want to develop a program that computes the integer part of the square root of a given natural number n, i.e a program that establishes
>
> $$R \triangleq \mathtt{x}^2 \leq n \wedge n < (\mathtt{x}+1)^2$$
>
> Delete the second conjunct and let
>
> $$P \triangleq \mathtt{x}^2 \leq n$$
>
> be an invariant. The statement `x:=0` will establish P. In order to establish R we need a statement that preserves P and gets us closer to R. Being slightly informal we use n in the program rather than a program variable that has the same value as n.
>
> ```
> while ((x+1)*(x+1) ≤ n) do x:=x+1
> ```
>
> We observe that $\mathcal{S}_{\text{wp}}[\![\mathtt{x:=x+1}]\!]P = ((\mathtt{x}+1)^2 \leq n)$ so P is an invariant. $\Diamond$

Another way to make the postcondition weaker is to replace some "constant" with a program variable with bounds.

Exempel. We consider the same problem as above, but now we use

$$P \triangleq \mathtt{x}^2 \leq n \wedge n < (\mathtt{y}+1)^2 \wedge \mathtt{x} \leq \mathtt{y} \leq n$$

To establish P we use `x:=0; y:=`n. In order to get closer to R we have to decrease `y-x` to 0. Dividing something in two equal parts sometimes leads to efficient algorithms. Let's try it:

```
while $\neg(\mathtt{x} = \mathtt{y})$ do
  d := (x+y)/2;
  if d*d $\leq n$ then x:=d
  else y:=d
```

where we can prove that P is an invariant. We have extended arithmetic expressions with division by 2. This cannot give rise to an runtime error which general division could. $\Diamond$

Our next example deals with arrays. We will use the notation $v[a]$, where v is an array name and a is an integer expression, to denote an array element with a given index. We ignore the fact that arrays should have fixed index bounds and errors should occur if we use an index outside the bounds. We have to define

$$\mathcal{S}_{\mathrm{wp}}[\![v[a_1] := a_2]\!]R \triangleq R[v \mapsto v[a_1 \mapsto a_2]]$$

The intention is that we are to replace every occurrence of the array name v in R by an array expression $v[a_1 \mapsto a_2]$. When evaluated in a state, σ, the value of every element of $v[a_1 \mapsto a_2]$ has the same value in v except for the element with index $\mathcal{A}[\![a_1]\!]\sigma$ which has the value $\mathcal{A}[\![a_2]\!]\sigma$.

If `v` is an array name $\mathtt{v}[k..l]$ will denote the subarray with indices from k to l.

Exempel. Let $\mathtt{v}[0..n-1]$ be a sorted array. We say that v has a *plateau* with length `p` if there is an index i such that $\mathtt{v}[i] = \mathtt{v}[i+\mathtt{p}-1]$ (and equal to all the elements in between). We are going to construct a program to compute the length of the longest plateau in $\mathtt{v}[0..n-1]$. We want to establish

$$R \triangleq (\exists k \in \{0..n-\mathtt{p}\} \,.\, v[k] = v[k+\mathtt{p}-1]) \wedge (\forall k \in \{0..n-\mathtt{p}-1\} \,.\, v[k] \neq v[k+\mathtt{p}])$$

As an invariant we use

$$P \triangleq (1 \leq i \leq n) \wedge \mathtt{p} \text{ is the length of the longest plateau in } \mathtt{v}[0..i-1]$$

P is established by `i:=1; p:=0`. To get closer to R we increment i. There are two cases; either p remains the same or it is incremented:

```
while $\neg$(i=n) do
  if $\neg(\mathtt{v[i]} = \mathtt{v[i-p]})$ then i:=i+1
  else (i:=i+1; p:=p+1)
```

Constructing a program for this problem without considering invariants would probably lead to a more complicated solution. $\Diamond$

7.2 References

These notes are inspired by

1. D. Gries: *The Science of Programming*, Springer Verlag, 1981.
2. E.C.R Hehner: *The Logic of Programming*, Prentice-Hall, 1984.