Strongest postcondition

Farzaneh Derakhshan
based on material by Stefan Muller, Jim Sasaki, and Mike Gordon

CS 536: Science of Programming, Fall 2023
Lecture 13

In the last two lectures, we discussed the weakest preconditions and learned that they are calculated *backward*, i.e., given a postcondition and a program, we constructed the weakest precondition. There is a dual concept called the *strongest postcondition*, which we will construct *forward*, i.e., given a precondition and a program, we will constrict the strongest postcondition.

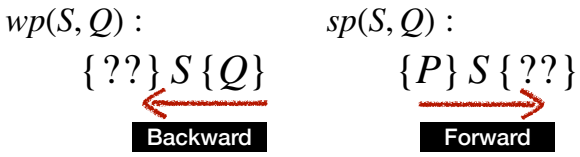

Figure 1: Weakest precondition: backward, Strongest postcondition: forward

Before introducing the strongest postcondition and an algorithm to construct it, let's take a different approach to look at the weakest precondition. Consider the set of all states that satisfy the weakest precondition, i.e., $\texttt{WP} - \texttt{state}(S, Q) = \{\sigma \in \mathsf{states} \mid \sigma \vDash wp(S, Q)\}$. Here, we write states for the set of all states σ, and use the set builder notation to collect those elements of state that satisfy the weakest precondition in a smaller set $\texttt{WP} - \texttt{state}$. On the other hand, we know that the weakest precondition $wp(S, Q)$ holds in precisely those initial states for which there exists an execution of S that terminates in a final state satisfying Q. By this observation, we can build an equivalent definition for the set $\texttt{WP} - \texttt{state}$ as follows (in blue):

$$\begin{aligned} \texttt{WP} - \texttt{state}(S, Q) &= \{\sigma \in \mathsf{states} \mid \sigma \vDash wp(S, Q)\} \\ &= \{\sigma \in \mathsf{states} \mid \exists \sigma'.\, M(S, \sigma) = \sigma' \wedge \sigma' \vDash Q\} \end{aligned}$$

We can take a similar approach to build the set for the strongest postcondition. The define the set $\texttt{SP} - \texttt{state}$, as all states that satisfy the strongest postcondition, i.e., $\texttt{SP} - \texttt{state}(S, P) = \{\sigma' \in \mathsf{states} \mid \sigma' \vDash sp(S, P)\}$. The strongest postcondition $sp(S, P)$ holds in precisely those final states for which there exists an execution of S that starts from an initial state satisfying P. So we can build an equivalent definition for the set $\texttt{SP} - \texttt{state}$ (in blue):

$$\begin{aligned} \texttt{SP} - \texttt{state}(S, P) &= \{\sigma' \in \mathsf{states} \mid \sigma' \vDash sp(S, P)\} \\ &= \{\sigma' \in \mathsf{states} \mid \exists \sigma.\, \sigma \vDash P \wedge M(S, \sigma) = \sigma'\} \end{aligned}$$

Example 1. What are the elements of the set $\texttt{SP} - \texttt{state}(S, F)$? Can we find $sp(S, F)$ from the set $\texttt{SP} - \texttt{state}(S, F)$?

Let's rewrite the above definition for program S and precondition F:

$$\begin{aligned} \texttt{SP} - \texttt{state}(S, F) &= \{\sigma' \in \mathsf{states} \mid \sigma' \vDash sp(S, F)\} \\ &= \{\sigma' \in \mathsf{states} \mid \exists \sigma.\, \sigma \vDash F \wedge M(S, \sigma) = \sigma'\} \end{aligned}$$

We know that no state that satisfies F – it is a contradiction. So the condition $\sigma \vDash$ (in red) is never satisfied, meaning $\texttt{SP} - \texttt{state}(S, F) = \{\}$. The set $\texttt{SP} - \texttt{state}(S, F)$ being empty means that no σ' satisfies the

strongest postcondition $sp(S, F)$, and thus $sp(S, F) = F$. (**Exercise 1.** Why?—Hint: use line 1 of the definition.)

Example 2. What are the elements of the set $\texttt{SP} - \texttt{state}(x := sqrt(-1), T)$?

Again, we rewrite the definition above for program $x := sqrt(-1)$ and precondition T:

$$\begin{array}{lcl} \texttt{SP} - \texttt{state}(x := sqrt(-1), T) & = & \{\sigma' \in \mathsf{states} \mid \sigma' \vDash sp(x := sqrt(-1), T)\} \\ & = & \{\sigma' \in \mathsf{states} \mid \exists \sigma.\, \sigma \vDash T \wedge M(x := sqrt(-1), \sigma) = \sigma'\} \end{array}$$

We know that the second condition (in red) is always false. After all, the program $x := sqrt(-1)$ always results in a runtime error, i.e., we have $M(x := sqrt(-1), \sigma) = \perp$ for any σ. So $\texttt{SP} - \texttt{state}(x := sqrt(-1), T)$ is an empty set and $sp(x := sqrt(-1), T)$ is the contradictory formula F.

Note 1. In Example 2, we have $\{T\}x := sqrt(-1)\{sp(x := sqrt(-1), T)\}$ which is equal to $\{T\}x := sqrt(-1)\{F\}$. The reason is the program $x := sqrt(-1)$ never terminates successfully (lecture 7). It turns out that, in general, we have

$$\vDash \{P\}\ S\ \{sp(S, P)\}.$$

However, $[\ T\ ]\ x := sqrt(-1)\ [\ sp(x := sqrt(-1), T)\ ]$ does not hold (again because the program never terminates successfully). We have a counterexample to conclude that $[\ P\ ]\ S\ [\ sp(S, P)\ ]$ is not always valid, i.e., $\not\vDash [\ P\ ]\ S\ [\ sp(S, P)\ ]$.

But, if the program S terminates when the precondition P is satisfied, then the total correctness triple $[\ P\ ]\ S\ [\ sp(S, P)\ ]$ is also valid. We can formalize the sentence "if the program S terminates when the precondition P is satisfied" as the Hoare-triple $[\ P\ ]\ S\ [\ T\ ]$. As a result, we have

$$\mathsf{if}\ \vDash [\ P\ ]\ S\ [\ T\ ]\ \mathsf{then}\ \vDash [\ P\ ]\ S\ [\ sp(S, P)\ ].$$

Interestingly, the other direction also holds, i.e., if the total correctness triple $[\ P\ ]\ S\ [\ sp(S, P)\ ]$ is valid, then the program S terminates when the precondition P is satisfied, i.e.,

$$\mathsf{if}\ \vDash [\ P\ ]\ S\ [\ sp(S, P)\ ]\ \mathsf{then}\ \vDash [\ P\ ]\ S\ [\ T\ ]$$

Note 2. In Note 1., we established that $sp(S, P)$ is a postcondition for the partial correctness triple, i.e., $\vDash \{P\}\ S\ \{sp(S, P)\}$. But it is not "any" postcondition for this triple; it is "the strongest" one, meaning that for any Q such that $\vDash \{P\}\ S\ \{Q\}$, we have $sp(S, P) \Rightarrow Q$.

1 Algorithm for strongest postcondition

So far, we have described the meaning of the strongest postcondition, but we haven't explained how to construct the formula. In this section, we introduce an algorithm for constructing the strongest postcondition.

1.1 A forward rule for assignment

It turns out that the algorithm requires another version of the `Assign` rule. The `Assign` rule that we saw earlier, as follows

$$\frac{}{\{[e/x]Q\}\ x := e\ \{Q\}}\ (\textsc{Assign})$$

is a backward rule: To apply this rule in the proof, we have to find the precondition $[e/x]Q$ given a postcondition Q.

For building the strongest postcondition, we need to introduce a forward rule for the assignment, i.e., a rule that, given a precondition, gives us the postcondition. A naive solution (which is not always correct) is to write the rule as

$$\frac{}{\{P\}\ x := e\ \{P \wedge x = e\}}\ (\textsc{Assign}^*)$$

This rule works only when x does not occur as a free variable in P (see Example 3). However, `Assign`* does not give us a correct postcondition if the variable we are assigned to already exists as a free variable in the precondition (See Example 4).

Example 3. We prove the valid partial triple $\{y = 1\}\, x := 2\, \{y = 1 \wedge x = 2\}$ using the `Assign`* rule:

$$\frac{}{\{y = 1\}\, x := 2\, \{y = 1 \wedge x = 2\}}\ (\textsc{Assign}^*)$$

Example 4. With `Assign`* rule we can prove an invalid triple $\{y = 1\}\, y := 2\, \{y = 1 \wedge y = 2\}$!!

$$\frac{}{\{y = 1\}\, y := 2\, \{y = 1 \wedge y = 2\}}\ (\textsc{Assign}^*)$$

The program always terminates successfully, with the value 2 assigned to y, but the triple states that if the program terminates, we have both $y = 1$ and $y = 2$ in the final state. Hence, `Assign`* is an unsound rule.

To make the rule sound, we need to rename any free occurrence of x in P and e to a fresh variable x_0, which is existentially quantified.

$$\frac{}{\{P\}\, x := e\, \{\exists x_0.[x_0/x]P \wedge x = [x_0/x]e\}}\ (\textsc{Assign-fwd})$$

The existentially quantified variable x_0 represents the value of x in the state before executing the assignment, i.e., the value of x_0 in the initial state. The postcondition expresses that after the assignment, the value of x is the value of e evaluated in the initial state (hence $[x_0/x]e$) and the precondition evaluated in the initial state (hence $[x_0/x]P$) continue to hold. Alternatively, we can understand this rule as the initial state is represented by the statement $(x = x_0) \wedge P$, i.e., x_0 is a name for the value of x before the assignment. Then the state after executing $x := e$ is represented by $(x = [x_0/x]e) \wedge [x_0/x]P$.

Example 5. using `Assign` $-$ `fwd` rule for the program $y := 2$ and precondition $y = 1$ in Example 4, we get:

$$\frac{}{\{y = 1\}\, y := 2\, \{\exists y_0.[y_0/y]y = 1 \wedge y = [y_0/y]2\}}\ (\textsc{Assign-fwd})$$

The postcondition $\{\exists y_0.[y_0/y]y = 1 \wedge y = [y_0/y]2\}$ can be simplified further to $\exists y_0.y_0 = 1 \wedge y = 2$. Since $\exists y_0.y_0 = 1 \wedge y = 2 \Rightarrow y = 2$, by one application of the consequence rule, we get $\{y = 1\}\, y := 2\, \{y = 2\}$.

Example 6.

$$\frac{}{\{x > 0\}\ x := x - 1\ \{\exists x_0.[x_0/x]x > 0 \wedge x = [x_0/x]x - 1\}}\ (\textsc{Assign-fwd})$$

And $\exists x_0.[x_0/x]x > 0 \wedge x = [x_0/x]x - 1$ can be simplified to $\exists x_0.x_0 > 0 \wedge x = x_0 - 1$, giving us the triple $\{x > 0\}\ x := x - 1\ \{\exists x_0.x_0 > 0 \wedge x = x_0 - 1\}$. Moreover, we know that $\exists x_0.x_0 > 0 \wedge x = x_0 - 1 \Rightarrow x \geq 0$, which by the consequence rule gives us $\{x > 0\}\ x := x - 1\ \{x \geq 0\}$.

1.2 The algorithm

Having the forward assignment rule, it is straightforward to define the algorithm for building the strongest postcondition. We define the algorithm recursively as:

$$\begin{array}{lll}
(1) & sp(\mathsf{skip}, P) & ::= \quad P \\
(2) & sp(x := e, P) & ::= \quad \exists x_0.[x_0/x]P \wedge x = [x_0/x]e \\
(3) & sp(S_1; S_2, P) & ::= \quad sp(S_2, sp(S_1, P)) \\
(4) & sp(\mathsf{if}\ e\ \mathsf{then}\ S_2\ \mathsf{else}\ S_2\ \mathsf{fi}, P) & ::= \quad sp(S_1, P \wedge e) \vee sp(S_2, P \wedge \neg e)
\end{array}$$

The first two lines for skip and $x := e$, following the rules for these constructs, return the postcondition in the rules when the precondition is P.

Line 3 for sequences states that to find the strongest postcondition of $S_1; S_2$ for a given precondition P, first find the strongest postcondition of S_1 for P, then feed the result as the precondition for S_2, and find the strongest postcondition of S_2 with respect to $sp(S_1, P)$.

Line 4 is similar to the alternative if rule that we saw in Lecture 9:

$$\frac{\{P \wedge e\}\, S_1\, \{Q_1\} \qquad \{P \wedge \neg\, e\}\, S_2\, \{Q_2\}}{\{P\}\, \mathsf{if}\ e\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}\, \{Q_1 \vee Q_2\}}\ (\textsc{If}^*)$$

It says that for each branch, find the strongest postcondition, with an extra assumption that e holds in the first branch and $\neg e$ holds in the second branch. The strongest postcondition of the if-clause is the strongest postcondition of either of its branches (hence using $\vee$ to connect the sp of two branches).

Example 7. We want to construct $sp(x := x + k; y := y + k, x > y)$:

$sp(x := x + k; y := y + k, x > y) :=$	$sp(y := y + k, sp(x := x + k, x > y))$	**by Line(3)**
	$sp(y := y + k, \exists x_0.[x_0/x](x > y) \wedge x = [x_0/x](x + k))$	**by Line(2)**
	$sp(y := y + k, \exists x_0.x_0 > y \wedge x = x_0 + k)$	**by substitution**
	$\exists y_0.([y_0/y](\exists x_0.x_0 > y \wedge x = x_0 + k) \wedge y = [y_0/y](y + k))$	**by Line(2)**
	$\exists y_0.((\exists x_0.x_0 > y_0 \wedge x = x_0 + k) \wedge (y = y_0 + k))$	**by subsetitution**

Example 8. We want to construct $sp(\mathsf{if}\ x \geq 0\ \mathsf{then}\ y := x\ \mathsf{else}\ y := -x\ \mathsf{fi}, x \geq 0)$:

$sp(\mathsf{if}\ x \geq 0\ \mathsf{then}\ y := x\ \mathsf{else}\ y := -x\ \mathsf{fi}, x \geq 0) :=$	
$sp(y := x, x \geq 0 \wedge x \geq 0) \wedge sp(y := -x, x \geq 0 \wedge \neg(x \geq 0))$	**by Line(4)**
$\exists y_0.([y_0/y](x \geq 0 \wedge x \geq 0) \wedge y = [y_0/y]x) \wedge \exists y_0.([y_0/y](x \geq 0 \wedge \neg(x \geq 0)) \wedge y = [y_0/y] - x)$	**by Line(2)**
$\exists y_0.((x \geq 0 \wedge x \geq 0) \wedge y = x) \wedge \exists y_0.((x \geq 0 \wedge \neg(x \geq 0)) \wedge y = -x)$	**by substitution**

We can simplify the right-hand side of the last line to F, so we can simplify the whole expression to

$$\exists y_0.((x \geq 0 \wedge x \geq 0) \wedge y = x)$$

and, because, y_0 doesn't occur free in the expression, we can further simplify to

$$x \geq 0 \wedge y = x$$