# *Forward Assignment; Strongest Postconditions*

*CS 536: Science of Programming, Fall 2019*

9/30 pp.3,7, 10/2

## *A. Why?*

- At times, a forward version of the assignment rule is more appropriate than the backward version.
- The forward assignment rule is part of calculating the *sp* (strongest postcondition) of a loop−free program.

## *B. Outcomes*

At the end of this lecture you should

- Know the basic assignment axioms.
- Know what a strongest postcondition is and how to calculate the *sp* of loop−free programs.

## *C. Forward Assignment Rules*

- We already have a "backwards" assignment rule, $\{P(e)\}\ v := e\ \{P(v)\}$ where $P$ is a predicate function. If we just use the body of $P$ as the predicate, the rule is $\{\,(\text{body of } P)[e/v]\,\}\ v := e\ \{P\}$.
  - Since $p[e/v] \equiv wlp(v := e, p)$, we know this is the most general possible rule.
- What about the other direction, $\{p\}\ v := e\ \{???\}$ — what can we use for the postcondition?
  - Most people's first guess is $\{p\}\ v := e\ \{p \wedge v = e\}$, which can work under certain conditions.

### *New Variable Introduction*

- If $v$ is a new (fresh) variable (doesn't appear free in $p$ and doesn't appear in $e$) then $\{p\}\ v := e\ \{p \wedge v = e\}$.
  - For example, $\{\mathtt{x} > \mathtt{y}\}\ \mathtt{z} := 2\ \{\mathtt{x} > \mathtt{y} \wedge \mathtt{z} = 2\}$
- To justify this, using *wlp*, we know $\{(p \wedge v = e)[e/v]\}\ v := e\ \{p \wedge v = e\}$.
  - Expanding, $(p \wedge v = e)[e/v] \equiv p[e/v] \wedge e = e[e/v]$.
  - Since $v$ is fresh, it doesn't occur in $p$ or $e$, so $p[e/v] \equiv p$ and $e[e/v] \equiv e$. So we need $\{p \wedge e = e\}\ v := e\ \{p \wedge v = e\}$, which definitely holds.

### *Forward Assignment (General Case)*

- As an example of why $\{p\}\ v := e\ \{p \wedge v = e\}$ doesn't work in general, consider $\{\mathtt{x} > 0\}\ \mathtt{x} := \mathtt{x/2}\ \{???\}$.
  - Since division truncates, if $\mathtt{x} = 1$, the final state doesn't satisfy $\mathtt{x} > 0$. We might want to try $\mathtt{x*2} > 0$ as the postcondition, but again this fails if $\mathtt{x} = 1$.
  - What we do know is that the value $\mathtt{x}$ had before the assignment was positive, and the new value of $\mathtt{x}$ is half of the old value of $\mathtt{x}$. If we use $\mathtt{x}_0$ as the name for the value of $\mathtt{x}$ before the assignment, we can say $\{\mathtt{x}_0 = \mathtt{x} \wedge \mathtt{x} > 0\}\ \mathtt{x} := \mathtt{x/2}\ \{\mathtt{x}_0 > 0 \wedge \mathtt{x} = \mathtt{x}_0/2\}$.
  - Note we don't have to actually store $\mathtt{x}_0$ in memory; it's just a name we use for logical reasoning purposes — $\mathtt{x}_0$ is a "fresh logical constant"; fresh in the sense that it doesn't appear in $p$ or $e$, logical because it only appears in the correctness discussion, not the program, and constant because though $\mathtt{x}$ changes, $\mathtt{x}_0$ doesn't. (Note in this context, "logical" doesn't mean "boolean".)

 

- **The general rule** is $\{p \wedge v = v_0\}\ v := e\ \{p[v_0/v] \wedge v = e[v_0/v]\}$. In the precondition, we may be able to omit $v = v_0$ as being understood.
  - **Example 1a**: $\{\mathtt{x} > 0 \wedge \mathtt{x} = \mathtt{x}_0\}\ \mathtt{x := x-1}\ \{\mathtt{x}_0 > 0 \wedge \mathtt{x} = \mathtt{x}_0\mathtt{-1}\}$.
  - **Example 2a**: $\{\mathtt{s} = \mathtt{sum(0,i)} \wedge \mathtt{s} = \mathtt{s}_0\}\ \mathtt{s := s+i+1}\ \{\mathtt{s}_0 = \mathtt{sum(0,i)} \wedge \mathtt{s} = \mathtt{s}_0\mathtt{+i+1}\}$.
- The $\mathtt{x} = \mathtt{x}_0$ and $\mathtt{s} = \mathtt{s}_0$ clauses are a bit annoying; we could drop them by using an existential in the postcondition, but that's no fun either:
  - **Example 1b**: $\{\mathtt{x} > 0\}\ \mathtt{x := x-1}\ \{\exists \mathtt{x}_0 \,.\, \mathtt{x}_0 > 0 \wedge \mathtt{x} = \mathtt{x}_0\mathtt{-1}\}$
  - **Example 2b**: $\{\mathtt{s} = \mathtt{sum(0,i)}\}\ \mathtt{s := s+i+1}\ \{\exists \mathtt{s}_0 \,.\, \mathtt{s}_0 = \mathtt{sum(0,i)} \wedge \mathtt{s} = \mathtt{s}_0\mathtt{+i+1}\}$.
- Let's drop the existential as implied — when a symbol appears in the postcondition but not the precondition, then we're implicitly quantifying it existentially in the postcondition.
  - We've actually been doing something similar with the precondition: Variables free in the precondition are treated as being universally quantified across both the precondition and postcondition.
  - **Example 1c**: (For all $\mathtt{x}$, there is an $\mathtt{x}_0$ such that) $\{\mathtt{x} > 0\}\ \mathtt{x := x-1}\ \{\mathtt{x}_0 > 0 \wedge \mathtt{x} = \mathtt{x}_0\mathtt{-1}\}$
  - **Example 2c**: (For all $\mathtt{s}$ and $\mathtt{i}$, there is an $\mathtt{s}_0$ such that)
    $\{\mathtt{s} = \mathtt{sum(0,i)}\}\ \mathtt{s := s+i+1}\ \{\mathtt{s}_0 = \mathtt{sum(0,i)} \wedge \mathtt{s} = \mathtt{s}_0\mathtt{+i+1}\}$.
  - **Example 3:** $\{\mathtt{s}_0 = \mathtt{sum(0,\ i)} \wedge \mathtt{s} = \mathtt{s}_0\mathtt{+i+1}\}\ \mathtt{i := i+1}\ \{\mathtt{s}_0 = \mathtt{sum(0,\ i}_0\mathtt{)} \wedge \mathtt{s} = \mathtt{s}_0\mathtt{+i}_0\mathtt{+1} \wedge \mathtt{i} = \mathtt{i}_0\mathtt{+1}\}$. The postcondition of this example can be weakened to $\mathtt{s} = \mathtt{sum(0,i)}$, which will be useful later for explaining the loop **`while`** `i < n` **`do`** `s := s+i+1; i := i+1` **`od`**.
    - Using the backward assignment rule on {??} s := s+i+1; i := i+1 {s = sum(0, i)} gets us a logically equivalent triple
      - ?? = wp (s := s+i+1; i := i+1, s = sum(0, i))
        ≡ wp(s := s+i+1, s = sum(0, i+1))
        ≡ s+i+1 = sum(0, i+1)

### *Correctness of The Forward Assignment Rule*

- The forward assignment rule appears to be very different from our earlier "backward" assignment rule, but actually, we can derive the forward assignment rule using the backward assignment rule.
- **Theorem** (**Correctness of Forward Assignment**
  $\vDash \{p \wedge v = v_0\}\ v := e\ \{p[v_0/v] \wedge v = e[v_0/v]\}$, where $v_0$ is a fresh logical constant.
- **Proof**:
  - We'll argue that the precondition $p \wedge v = v_0$ implies the *wlp* of the assignment and postcondition.
  - We need $(p \wedge v = v_0) \rightarrow wlp(v := e, q)$, where $q \equiv (p[v_0/v] \wedge v = e[v_0/v])$. Simplifying,

    $$\begin{aligned} wlp(v := e, q) &\equiv q[e/v] \\ &\equiv (p[v_0/v] \wedge v = e[v_0/v])[e/v] \\ &\equiv p[v_0/v][e/v] \wedge v[e/v] = e[v_0/v][e/v] \end{aligned}$$

  - We can simplify $p[v_0/v][e/v] \wedge v[e/v] = e[v_0/v][e/v]$ by looking at its three parts.
    - The easiest one is $v[e/v] \equiv e$.
    - For $p[v_0/v]$, since $v$ doesn't appear in it, we have $p[v_0/v][e/v] \equiv p[v_0/v]$, which is $\Leftrightarrow p \wedge v = v_0$.

 

    - Similarly, since $v$ doesn't appear in $e[v_0/v]$, we have $e[v_0/v][e/v] = e[v_0/v] = e$ when $v = v_0$.
  - So $(p[v_0/v][e/v] \wedge v[e/v] = e[v_0/v][e/v]) \Leftrightarrow p \wedge v = v_0 \wedge e = e \Leftrightarrow p \wedge v = v_0$.
  - $p[v_0/v][e/v] \wedge v[e/v] = e[v_0/v][e/v]) \equiv$
    $p[v_0/v] \wedge e = e[v_0/v]$
    and this is $\Leftrightarrow p \wedge v = v_0$ [9/30]
  - For a particular example, with $\{\mathtt{x} > 0 \wedge \mathtt{x} = \mathtt{x}_0\}\ \mathtt{x := x-1}\ \{\mathtt{x}_0 > 0 \wedge \mathtt{x} = \mathtt{x}_0\mathtt{-1}\}$, we find $wlp(\mathtt{x := x-1}, \mathtt{x}_0 > 0 \wedge \mathtt{x} = \mathtt{x}_0\mathtt{-1}) \equiv \mathtt{x}_0 > 0 \wedge \mathtt{x-1} = \mathtt{x}_0\mathtt{-1}$, which is implied by $\mathtt{x} > 0 \wedge \mathtt{x} = \mathtt{x}_0$.
- Note *that* the simpler rule for introducing a new variable is a special case of the general rule:
  - We want $\{p\}\ v := e\ \{p \wedge v = e\}$ if $v$ doesn't occur in $e$ or and $v$ is not free in $p$. The forward assignment rule says $\{p\}\ v := e\ \{p[v_0/v] \wedge v = e[v_0/v]\}$, where $v_0$ is a fresh logical constant. Since $v$ does not occur in $e$, we know $e[v_0/v] \equiv e$. Similarly, since $v$ isn't free in $p$, we know $p[v_0/v] \equiv p$. Substituting into $\{p\}\ v := e\ \{p[v_0/v] \wedge v = e[v_0/v]\}$ gives us $\{p\}\ v := e\ \{p \wedge v = e\}$.
- Not shown: So we've shown that the forward assignment rule can be derived if we have the backward assignment rule. Turns out the other direction is also true: You can derive the backward assignment rule from the forward assignment rule. [9/30]

### *D. The Strongest Postcondition (sp)*

- **Definition**: Given a precondition $p$ and program $S$, the **strongest postcondition** of $p$ and $S$ is (the predicate that stands for) the set of states we can terminate in if we run $S$ starting in a state that satisfies $p$. In symbols,
  - $sp(p, S) = \{\tau \mid \tau \in M(S, \sigma) \cap \Sigma$ for some $\sigma$ where $\sigma \vDash p\}$.
  - Equivalently, $sp(p, S) = \bigcup (M(S, \sigma) \cap \Sigma)$ where $\sigma \vDash p$.
- I.e., $sp(p, S)$ is the image of $M(S, \ldots)$ (considered as a function) over the set of states that satisfy $p$.
- Figures 1 and 2 show the relationship between $p$, $S$, and $sp(p, S)$ and $sp(\neg p, S)$
  - If $\sigma \vDash p$, then every state in $M(S, \sigma) \cap \Sigma$ is by definition in $sp(p, S)$, so $\vDash \{p\}\ S\ \{sp(p, S)\}$.
    - This is only valid for **partial correctness**: Starting in a state that satisfies $p$ might yield an error.
    - (To get total correctness, we need termination: $\vDash_{tot} \{p\}\ S\ \{sp(p, S)\}$ iff $\vDash_{tot} \{p\}\ S\ \{\mathtt{T}\}$.)
  - For $sp(\neg p, S)$, the situation is symmetric:
    - If $\sigma \vDash \neg p$, then every state in $M(S, \sigma) \cap \Sigma$ is by definition in $sp(\neg p, S)$.
    - We have partial correctness for $\{p\}\ S\ \{sp(\neg p, S)\}$.
    - We only get total correctness if $S$ always terminates starting in $\neg p$.
  - Notice that $sp(p, S)$ and $sp(\neg p, S)$ can have states in common: $sp(p, S) \cap sp(\neg p, S) \neq \varnothing$, just means it's possible for states satisfying $p$ and $\neg p$ to map to a common final state. (E.g., $\mathtt{x := 0}$ sets $\mathtt{x}$ to 0 regardless of $p$ or $\neg p$.)

 

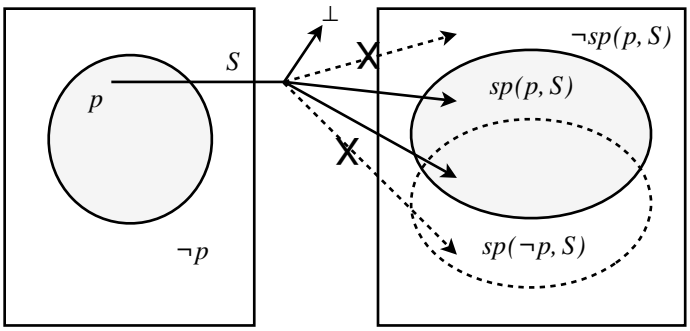


*Figure 1: sp(p, S) is the set of states reachable via S from p*

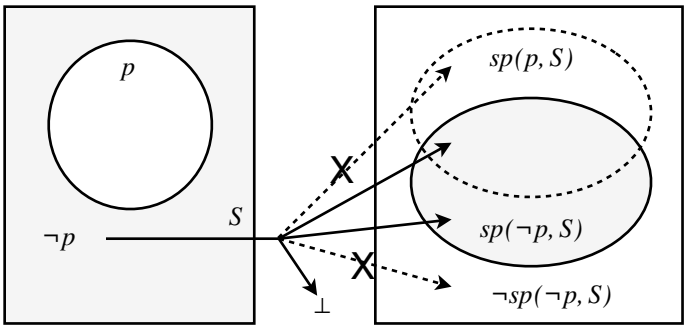


*Figure 2: sp(¬p, S) is the set of states reachable via S from ¬p*

- **Example 4**: For an example where $sp(p, S)$ and $sp(\neg p, S)$ overlap, since $\mathtt{x} := \mathtt{x}/4$ maps $\mathtt{x} = 0$, 1, 2, and 3 to $\mathtt{x} = 0$, the state $\{\mathtt{x} = 0\}$ can be reached using $\mathtt{x} := \mathtt{x}/4$ from states where $\mathtt{x} \geq 2$ and also from states where $\mathtt{x} < 2$, so $\mathtt{x} = 0 \rightarrow sp(\mathtt{x} \geq 2, \mathtt{x} := \mathtt{x}/4) \wedge sp(\mathtt{x} < 2, \mathtt{x} := \mathtt{x}/4)$.
- **Example 5**: Let $W \equiv$ **while** $\mathtt{i} \neq 0$ **do** $\mathtt{i} := \mathtt{i}-1$ **od**. Clearly, if $W$ terminates, it's with $\mathtt{i} = 0$, so for any precondition $p$, if i isn't free in $p$, we have $sp(p, W) \Leftrightarrow p \wedge \mathtt{i} = 0$, so $\{p\}\ W\ \{p \wedge \mathtt{i} = 0\}$ is partially correct. However, it's not totally correct for every $p$: It terminates iff we start in a state with $\mathtt{i} \geq 0$ (i.e., $\vDash_{tot} \{\mathtt{i} \geq 0\}\ W\ \{\mathtt{T}\}$). So $\{\mathtt{i} \geq 0\}\ W\ \{\mathtt{i} = 0\}$ is totally correct. However, $\{\mathtt{i} < 0\}\ W\ \{\mathtt{i} = 0\}$ is partially correct but never totally correct.
- **Strongest** postcondition: For partial correctness, $\vDash \{p\}\ S\ \{sp(p, S)\}$ (i.e., $sp(p, S)$ is a postcondition). What makes it the strongest postcondition is that it implies any other postcondition: for general $q$, $\vDash \{p\}\ S\ \{q\}$ iff $\vDash sp(p, S) \rightarrow q$. The proof of strength is easy: By definition of *sp,* every state $\tau \in sp(p, S)$ comes from some $M(S, \sigma)$ where $\sigma \vDash p$. Since $\{p\}\ S\ \{q\}$ is partially correct, we also know $\tau \vDash q$. Since every state satisfying $sp(p, S)$ also satisfies $q$, we know that $sp(p, S) \rightarrow q$ is valid.

### *E. Strongest Postconditions for Loop-Free Programs*

- Recall the forward assignment rule:
  - $\{p \wedge v = v_0\}\ v := e\ \{p[v_0/v] \wedge v = e[v_0/v]\}$, where $v_0$ is a fresh logical constant and the $v = v_0$ conjunct is optional and implicit.

 

- It turns out (we won't prove this) that this rule gives the strongest postcondition of the precondition and assignment, and we can use it as the base for calculating the strongest postcondition of a loop–free program:
  - $sp(p, \texttt{skip}) \equiv p$
  - $sp(p, v := e) \equiv p[v_0/v] \wedge v = e[v_0/v]$, where $v_0$ is a fresh constant.
  - $sp(p, S_1; S_2) \equiv sp(sp(p, S_1), S_2)$
    - The most we can know after $S_1$ ; $S_2$ is the most we know after executing $S_2$ in the state that is the most we know after $S_1$.
  - $sp(p, \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi}) \equiv sp(p \wedge B, S_1) \vee sp(p \wedge \neg B, S_2)$
    - After executing an **if–else**, we know what we know after executing the true branch (after the test succeeded) or the false branch (after the test failed), but we don't know which branch was taken.
    - For specific $p$, $B$, $S_1$, and $S_2$, we might be able to infer which branch was taken, but in the general case, we can't.
  - $sp(p, \texttt{if } B_1 \rightarrow S_1 \,\square\, B_2 \rightarrow S_2 \texttt{ fi}) \equiv sp(p \wedge B_1, S_1) \vee sp(p \wedge B_2, S_2)$. As with deterministic **if–else**, once the nondeterministic **if–fi** finishes, we know it evaluated one of the guarded commands but not necessarily which one.

### *F. Examples of Strongest Postconditions*

**Example 6:**

- Let's use *sp* to fill in the postcondition of $\{\texttt{x} > \texttt{y}\}\ \texttt{x := x+k; y := y+k}\ \{???\}$
- (Presumably, the postcondition will imply that x is still > y.)
- Let $p \equiv \texttt{x} > \texttt{y}$, then $sp(p, \texttt{x := x + k; y := y + k}) \equiv sp(sp(p, \texttt{x := x + k}), \texttt{y := y + k})$.
- Let's calculate the inner *sp* first:

$$\begin{aligned} & sp(p, \texttt{x} := \texttt{x} + \texttt{k}) \\ & \quad \equiv sp(\texttt{x} > \texttt{y}, \texttt{x} := \texttt{x} + \texttt{k}) \\ & \quad \equiv (\texttt{x} > \texttt{y})[\texttt{x}_0/\texttt{x}] \wedge \texttt{x} = (\texttt{x} + \texttt{k})[\texttt{x}_0/\texttt{x}] \\ & \quad \equiv \texttt{x}_0 > \texttt{y} \wedge \texttt{x} = \texttt{x}_0 + \texttt{k} \end{aligned}$$

- Then going back to our original problem and the outer *sp*, we get

$$\begin{aligned} & sp(p, \texttt{x} := \texttt{x} + \texttt{k; y} := \texttt{y} + \texttt{k}) \\ & \quad \equiv sp(sp(p, \texttt{x} := \texttt{x} + \texttt{k}), \texttt{y} := \texttt{y} + \texttt{k}) \\ & \quad \equiv sp(\texttt{x}_0 > \texttt{y} \wedge \texttt{x} = \texttt{x}_0 + \texttt{k}, \texttt{y} := \texttt{y} + \texttt{k}) \\ & \quad \equiv (\texttt{x}_0 > \texttt{y} \wedge \texttt{x} = \texttt{x}_0 + \texttt{k})[\texttt{y}_0/\texttt{y}] \wedge \texttt{y} = (\texttt{y} + \texttt{k})[\texttt{y}_0/\texttt{y}] \\ & \quad \equiv\ \texttt{x}_0 > \texttt{y}_0 \wedge \texttt{x} = \texttt{x}_0 + \texttt{k} \wedge \texttt{y} = \texttt{y}_0 + \texttt{k} \end{aligned}$$

- So our original triple can be filled in as

$$\{\texttt{x} > \texttt{y}\}\ \ \texttt{x := x + k; y := y + k}\ \ \{\texttt{x}_0 > \texttt{y}_0 \wedge \texttt{x} = \texttt{x}_0 + \texttt{k} \wedge \texttt{y} = \texttt{y}_0 + \texttt{k}\}$$

**Example 7:**

- Using the same program, let's compare the triple we got using *sp* with a triple we get using *wp*. For the postcondition, let's use x > y:

$wp(\texttt{x := x + k; y := y + k}, \texttt{x > y})$

$\equiv wp(\texttt{x := x + k}, wp(\texttt{y := y + k}, \texttt{x > y}))$

$\equiv wp(\texttt{x := x + k}, (\texttt{x > y})[\texttt{y+k/y}])$

$\equiv wp(\texttt{x := x + k}, (\texttt{x > y + k}))$

$\equiv (\texttt{x > y + k})[\texttt{x+k/x}]$

$\equiv \texttt{x + k > y + k}$

- Which gives us $\{\texttt{x+k > y+k}\}$ `x := x+k; y := y+k` $\{\texttt{x > y}\}$

**Example 8**:

- Let $p \equiv \texttt{y} = \texttt{y}_0$ and $S \equiv$ **if** $\texttt{x} \geq 0$ **then** `y := x` **else** `y := -x` **fi**. Then
  - $sp(p \wedge \texttt{x} \geq 0, \texttt{y := x}) \equiv (\texttt{x} \geq 0)[\texttt{y}_0/\texttt{y}] \wedge \texttt{y} = \texttt{x}[\texttt{y}_0/\texttt{y}] \equiv \texttt{x} \geq 0 \wedge \texttt{y} = \texttt{x}$
  - $sp(p \wedge \texttt{x} < 0, \texttt{y := -x}) \equiv (\texttt{x} < 0)[\texttt{y}_0/\texttt{y}] \wedge \texttt{y} = \texttt{-x}[\texttt{y}_0/\texttt{y}] \equiv \texttt{x} < 0 \wedge \texttt{y} = \texttt{-x}$
- So $sp(\texttt{y} = \texttt{y}_0,$ **if** $\texttt{x} \geq 0$ **then** `y := x` **else** `y := -x` **fi**$)$

  $\equiv sp(p \wedge \texttt{x} \geq 0, \texttt{y := x}) \vee sp(p \wedge \texttt{x} < 0, \texttt{y := -x})$

  $\equiv \texttt{x} \geq 0 \wedge \texttt{y} = \texttt{x} \ \vee \ \texttt{x} < 0 \wedge \texttt{y} = \texttt{-x}$
- Note that since the assignments to `y` don't depend on $\texttt{y}_0$. we don't find $\texttt{y}_0$ occurring in the result.

**Example 9**:

- Let $S \equiv$ **if** `even(x)` **then** `x := x+1` **else** `x := x+2` **fi** and let $p \equiv \texttt{x} \geq 0$, then

  $sp(p, S) \equiv sp(p,$ **if** `even(x)` **then** `x := x+1` **else** `x := x+2` **fi**$)$

  $\equiv sp(p \wedge \texttt{even(x)}, \texttt{x := x+1}) \vee sp(p \wedge \texttt{odd(x)}, \texttt{x := x+2})$
- For the true branch,

  $sp(p \wedge \texttt{even(x)}, \texttt{x := x+1})$

  $\equiv (p \wedge \texttt{even(x)})[\texttt{x}_0/\texttt{x}] \wedge \texttt{x} = (\texttt{x+1})[\texttt{x}_0/\texttt{x}]$

  $\equiv (\texttt{x} \geq 0 \wedge \texttt{even(x)})[\texttt{x}_0/\texttt{x}] \wedge \texttt{x} = (\texttt{x+1})[\texttt{x}_0/\texttt{x}]$

  $\equiv \texttt{x}_0 \geq 0 \wedge \texttt{even}(\texttt{x}_0) \wedge \texttt{x} = \texttt{x}_0\texttt{+1}$
- For the false branch,

  $sp(p \wedge \texttt{odd(x)}, \texttt{x := x+2})$

  $\equiv (p \wedge \texttt{odd(x)})[\texttt{x}_0/\texttt{x}] \wedge \texttt{x} = (\texttt{x+2})[\texttt{x}_0/\texttt{x}]$

  $\equiv (\texttt{x} \geq 0 \wedge \texttt{odd(x)})[\texttt{x}_0/\texttt{x}] \wedge \texttt{x} = \texttt{x}_0\texttt{+2}$

  $\equiv \texttt{x}_0 \geq 0 \wedge \texttt{odd}(\texttt{x}_0) \wedge \texttt{x} = \texttt{x}_0\texttt{+2}$
- So $sp(p, S) \equiv sp(p \wedge \texttt{even(x)}, \texttt{x := x+1}) \vee sp(p \wedge \texttt{odd(x)}, \texttt{x := x+2})$

  $\equiv (\texttt{x}_0 \geq 0 \wedge \texttt{even}(\texttt{x}_0) \wedge \texttt{x} = \texttt{x}_0\texttt{+1}) \vee (\texttt{x}_0 \geq 0 \wedge \texttt{odd}(\texttt{x}_0) \wedge \texttt{x} = \texttt{x}_0\texttt{+2})$
- If we want to logically simplify, we get $\texttt{x}_0 \geq 0 \wedge (\texttt{even}(\texttt{x}_0) \wedge \texttt{x} = \texttt{x}_0\texttt{+1} \vee \texttt{odd}(\texttt{x}_0) \wedge \texttt{x} = \texttt{x}_0\texttt{+2})$.
- That's about as far as we can simplify without losing information. For example, we *could* simplify to $(\texttt{x}_0 \geq 0 \wedge (\texttt{x} = \texttt{x}_0 + 1 \vee \texttt{x} = \texttt{x}_0 + 2))$ or to $(\texttt{x} \geq 1 \wedge \texttt{odd(x)})$, but whether we *want* to simplify or not depends on where we're using the *sp* in the overall proof of correctness for the program.

 

**Old values before conditionals**

- If we modify a variable in both arms of a conditional statement, then we need an implicit "old value" clause before the conditional: $\{\ldots \wedge \mathtt{x} = \mathtt{x}_0\}$ **if** … **then** x := … **else** x := … **fi**. Even if the variable is modified in only one arm of the conditional, to avoid losing information, it might still be necessary to track the old value before the conditional, not just in the arm that changes it
  - I.e., $\{\ldots \wedge \mathtt{x} = \mathtt{x}_0\}$ **if** … **then** x := … **else** x := … **fi**
  - Versus $\{\ldots\}$ **if** … **then** $\{\ldots \wedge \mathtt{x} = \mathtt{x}_0\}$ x := … **else** … no change to x … **fi**

**Example 10**:

- Let $S \equiv$ **if** $\mathtt{x} \geq \mathtt{y}$ **then** $\mathtt{x} := \mathtt{x}-\mathtt{y}$ **else** $\mathtt{y} := \mathtt{y}-\mathtt{x}$ **fi**
- Let $q_1 \equiv sp(\mathtt{x} = \mathtt{x}_0 \wedge \mathtt{y} = \mathtt{y}_0, S)$
  $\equiv sp(\mathtt{x} = \mathtt{x}_0 \wedge \mathtt{y} = \mathtt{y}_0 \wedge \mathtt{x} \geq \mathtt{y}, \mathtt{x} := \mathtt{x}-\mathtt{y}) \vee sp(\mathtt{x} = \mathtt{x}_0 \wedge \mathtt{y} = \mathtt{y}_0 \wedge \mathtt{x} < \mathtt{y}, \mathtt{y} := \mathtt{y}-\mathtt{x})$
  $\equiv (\mathtt{y} = \mathtt{y}_0 \wedge \mathtt{x}_0 \geq \mathtt{y} \wedge \mathtt{x} = \mathtt{x}_0-\mathtt{y}) \vee (\mathtt{x} = \mathtt{x}_0 \wedge \mathtt{x} < \mathtt{y}_0 \wedge \mathtt{y} = \mathtt{y}_0-\mathtt{x})$
- With $q_1$, if the first disjunct holds, then $\mathtt{x}_0 \geq \mathtt{y}_0$; if the second disjunct holds, then $\mathtt{x}_0 < \mathtt{y}_0$.
- If we only introduce $\mathtt{x}_0$ and $\mathtt{y}_0$ just before the assignments to x and y, we get
  - For the true branch, $sp(\mathtt{x} = \mathtt{x}_0 \wedge \mathtt{x} \geq \mathtt{y}, \mathtt{x} := \mathtt{x}-\mathtt{y}) \equiv (\mathtt{x}_0 \geq \mathtt{y} \wedge \mathtt{x} = \mathtt{x}_0-\mathtt{y})$
  - For the false branch, $sp(\mathtt{y} = \mathtt{y}_0 \wedge \mathtt{x} < \mathtt{y}, \mathtt{y} := \mathtt{y}-\mathtt{x}) \equiv (\mathtt{x} < \mathtt{y}_0 \wedge \mathtt{y} = \mathtt{y}_0-\mathtt{x})$
- Let $q_2$ be their disjunction, $(\mathtt{x}_0 \geq \mathtt{y} \wedge \mathtt{x} = \mathtt{x}_0-\mathtt{y}) \vee (\mathtt{x} < \mathtt{y}_0 \wedge \mathtt{y} = \mathtt{y}_0-\mathtt{x})$.
  - With $q_2$, if the first disjunct holds, we know $\mathtt{x}_0 \geq \mathtt{y}$, but not $\mathtt{x}_0 \geq \mathtt{y}_0$. Symmetrically, if the second conjunct holds, we know $\mathtt{x} < \mathtt{y}_0$, but not $\mathtt{x}_0 < \mathtt{y}_0$. (We certainly know $\mathtt{x}_0 \geq \mathtt{y}_0$ or $\mathtt{x}_0 < \mathtt{y}_0$, but we've lost the connections with them and the two disjuncts.)
- Since $q_1$ implies $q_2$ but not vice versa, we know $q_1$ is the stronger of the two, so it, not $q_2$, should be the strongest postcondition.

[9/30 added example]

**Example 11:**

- If we have a sequence of assignments to one variable, then we introduce multiple logical variables to talk about its values at different times in the sequence.
- To complete $\{\mathtt{x} > \mathtt{f}(\mathtt{x}, \mathtt{y})\}$ $\mathtt{x} := \mathtt{x}+1\mathtt{;}\ \mathtt{x} := \mathtt{x}*\mathtt{x}$ {???}, we'll calculate the strongest postcondition.
  - To talk about the original value of x, and the value of x between the two assignments, we need two different variables.
- We need $sp(\mathtt{x} > \mathtt{f}(\mathtt{x}, \mathtt{y}), S_1\mathtt{;}\ S_2) \equiv sp\,(sp(\mathtt{x} > \mathtt{f}(\mathtt{x}, \mathtt{y}), S_1), S_2)$ where $S_1 \equiv \mathtt{x} := \mathtt{x}+1$ and $S_2 \equiv \mathtt{x} := \mathtt{x}*\mathtt{x}$.

  $sp(\mathtt{x} > \mathtt{f}(\mathtt{x}, \mathtt{y}), S_1)$
  $\equiv sp(\mathtt{x} > \mathtt{f}(\mathtt{x}, \mathtt{y}), \mathtt{x} := \mathtt{x}+1)$
  $\equiv (\mathtt{x} > \mathtt{f}(\mathtt{x}, \mathtt{y}))[\mathtt{x}_0/\mathtt{x}] \wedge \mathtt{x} = (\mathtt{x}+1)[\mathtt{x}_0/\mathtt{x}]$ (using $\mathtt{x}_0$ as the fresh variable)
  $\equiv \mathtt{x}_0 > \mathtt{f}(\mathtt{x}_0, \mathtt{y}) \wedge \mathtt{x} = \mathtt{x}_0+1$

  $sp\,(sp(\mathtt{x} > \mathtt{f}(\mathtt{x}, \mathtt{y}), S_1), S_2)$
  $\equiv sp\,(\mathtt{x}_0 > \mathtt{f}(\mathtt{x}_0, \mathtt{y}) \wedge \mathtt{x} = \mathtt{x}_0+1, \mathtt{x} := \mathtt{x}*\mathtt{x})$

$\equiv (x_0 > f(x_0, y) \wedge x = x_0+1)[x_1/x] \wedge x = (x*x)[x_1/x]$ (using $x_1$ as the fresh variable)

$\equiv x_0 > f(x_0, y) \wedge x_1 = x_0+1 \wedge x = x_1*x_1$

- We have to be careful with which variables we use where in the answer. In $x_0 > f(x_0, y)$, we need $x_0$ to talk about the value of x before the first assignment. Between the two assignments, we have the then-current $x = x_0+1$, but after the second assignment, we use $x_1$ to name the value of $x$ between the two assignments because the now-current $x \neq x_1$. So after the second assignment we have $x_1 = x_0+1$ and $x = x_1*x_1$. But the $x_0 > f(x_0, y)$ clause remains the same because it talks about something that was true before the first assignment.

 

# *Forward Assignment; Strongest Postconditions*

*CS 536: Science of Programming*

## *G. Why*

- Sometimes the forward assignment rule is more helpful than the backward assignment rule.
- The strongest postcondition of a program is the most we can say about the state a program ends in.

## *H. Objectives*

At the end of this activity you should be able to

- Calculate the *sp* of a simple loop-free program.
- Fill in a missing postcondition of a simple loop–free program.

## *I. Questions*

1. What basic properties does $sp(p, S)$ have?

For Questions 2 - 7, syntactically calculate the following, including intermediate *sp* calculation steps. Don't logically simplify the result unless asked to.

2. $sp(\mathtt{y} \geq 0, \textbf{skip})$
3. $sp(\mathtt{i} > 0, \mathtt{i} := \mathtt{i}+1)$ [Hint: add an $\mathtt{i} = \mathtt{i}_0$ conjunct to $\mathtt{i} > 0$]
4. $sp(\mathtt{k} \leq \mathtt{n} \wedge \mathtt{s} = \mathtt{f(k, n)}, \mathtt{k} := \mathtt{k}+1)$
5. $sp(\mathtt{T}, \mathtt{i} := 0; \mathtt{k} := \mathtt{i})$ [Go ahead and logically simplify as you go by dropping uses of "$\mathtt{T} \wedge$" in the predicates]
6. $sp(\mathtt{i} \leq \mathtt{j} \wedge \mathtt{j}-\mathtt{i} < \mathtt{n}, \mathtt{i} := \mathtt{i}+\mathtt{j}; \mathtt{j} := \mathtt{i}+\mathtt{j})$.
7. $sp(\mathtt{s} := \mathtt{s}+\mathtt{i}+1; \mathtt{i} := \mathtt{i}+1, 0 \leq \mathtt{i} < \mathtt{n} \wedge \mathtt{s} = \mathtt{sum}(0, \mathtt{i}))$
8. Let $S \equiv$ **if** $\mathtt{x} < 0$ **then** $\mathtt{x} := -\mathtt{x}$ **fi**
   a. Calculate $sp(\mathtt{x} = \mathtt{x}_0, S)$.
   b. Logically simplify your result from part (a). Feel free to use the function $\mathtt{abs}(\ldots)$ or $|\ldots|$.
   c. Suppose we had calculated $sp(\mathtt{T}, \textbf{if } \mathtt{x} < 0 \textbf{ then } \mathtt{x} := -\mathtt{x} \textbf{ fi})$ introducing $\mathtt{x}_0$ in the true branch only. What would we get for the *sp* and what is the problem with it?

 

***Solution to Activity 13 (Forward Assignment; Strongest Postconditions)***

1. The *sp* has two properties:
   - $sp(p, S)$ is a partial correctness postcondition: $\vDash \{p\}\ S\ \{sp(p, S)\}$.
   - $sp(p, S)$ is strongest amongst the partial correctness postconditions: $\vDash \{p\}\ S\ \{q\}$ iff $sp(p, S) \rightarrow q$.

     (The second property actually implies the first, since $sp(p, S) \rightarrow sp(p, S)$.)

2. $\texttt{y} \geq 0$ (For the **`skip`** rule, the precondition and postcondition are the same.)

3. Let's implicitly add $\texttt{i} = \texttt{i}_0$ to the precondition, to name the starting value of `i`. Then

   $sp(\texttt{i} > 0, \texttt{i := i+1})$

   $\equiv (\texttt{i} > 0)[\texttt{i}_0/\texttt{i}] \wedge \texttt{i} = (\texttt{i+1})\ [\texttt{i}_0/\texttt{i}]$

   $\equiv \texttt{i}_0 > 0 \wedge \texttt{i} = \texttt{i}_0\texttt{+1}$

4. As in the previous problem, let's introduce a variable $\texttt{k}_0$ to name the starting value of k. Then

   $sp(\texttt{k} \leq \texttt{n} \wedge \texttt{s} = \texttt{f(k, n)}, \texttt{k := k+1})$

   $\equiv (\texttt{k} \leq \texttt{n} \wedge \texttt{s} = \texttt{f(k, n)})[\texttt{k}_0/\texttt{k}] \wedge \texttt{k} = (\texttt{k+1})[\texttt{k}_0/\texttt{k}]$

   $\equiv \texttt{k}_0 \leq \texttt{n} \wedge \texttt{s} = \texttt{f}(\texttt{k}_0, \texttt{n}) \wedge \texttt{k} = \texttt{k}_0\texttt{+1}$

5. We don't need to introduce names for the old values of `i` and `k` (they're irrelevant).

   $sp(\texttt{T}, \texttt{i := 0; k := i})$

   $\equiv sp(sp(\texttt{T}, \texttt{i := 0}), \texttt{k := i})$

   $\Leftrightarrow sp(\texttt{i = 0}, \texttt{k := i})$ // We've dropped the "$\texttt{T} \wedge$ " part of $\texttt{T} \wedge \texttt{i} = 0$)

   $\equiv \texttt{i} = 0 \wedge \texttt{k} = \texttt{i}$

6. Let's introduce $\texttt{i}_0$ and $\texttt{j}_0$ as we need them, then

   $sp(\texttt{i} \leq \texttt{j} \wedge \texttt{j-i} < \texttt{n}, \texttt{i := i+j; j := i+j})$

   $\equiv sp(sp(\texttt{i} \leq \texttt{j} \wedge \texttt{j-i} < \texttt{n}, \texttt{i := i+j}), \texttt{j := i+j})$

   $\equiv sp(\texttt{i}_0 \leq \texttt{j} \wedge \texttt{j-i}_0 < \texttt{n} \wedge \texttt{i} = \texttt{i}_0\texttt{+j}, \texttt{j := i+j})$

   $\equiv \texttt{i}_0 \leq \texttt{j}_0 \wedge \texttt{j}_0\texttt{-i}_0 < \texttt{n} \wedge \texttt{i} = \texttt{i}_0\texttt{+j}_0 \wedge \texttt{j} = \texttt{i+j}_0$

7. $sp(\texttt{s := s+i+1; i := i+1}, 0 \leq \texttt{i} < \texttt{n} \wedge \texttt{s} = \texttt{sum}(0, \texttt{i}))$

   $\equiv sp(\texttt{i := i+1}, sp(\texttt{s := s+i+1}, 0 \leq \texttt{i} < \texttt{n} \wedge \texttt{s} = \texttt{sum}(0, \texttt{i}))$

   For the inner *sp*,

   $sp(\texttt{s := s+i+1}, 0 \leq \texttt{i} < \texttt{n} \wedge \texttt{s} = \texttt{sum}(0, \texttt{i})$

   $\equiv 0 \leq \texttt{i} < \texttt{n} \wedge \texttt{s}_0 = \texttt{sum}(0, \texttt{i}) \wedge \texttt{s} = \texttt{s}_0\texttt{+i+1}$ Using $\texttt{s}_0$ to name the old value of `s`

   Returning to the outer *sp*,

   $sp(\texttt{i := i+1}, sp(\texttt{s := s+i+1}, 0 \leq \texttt{i} < \texttt{n} \wedge \texttt{s} = \texttt{sum}(0, \texttt{i}))$

   $\equiv sp(\texttt{i := i+1}, 0 \leq \texttt{i} < \texttt{n} \wedge \texttt{s}_0 = \texttt{sum}(0, \texttt{i}) \wedge \texttt{s} = \texttt{s}_0\texttt{+i+1})$

 

$\equiv 0 \le \mathtt{i'} < \mathtt{n} \wedge \mathtt{s}_0 = \mathtt{sum}(0, \mathtt{i'}) \wedge \mathtt{s} = \mathtt{s}_0 + \mathtt{i'} + 1 \wedge \mathtt{i} = \mathtt{i'} + 1$

Using `i'` to name the old value of `i`

(There's no particular reason I used `i'` here except that; any other name like $\mathtt{i}_0$ or `j` or `w` works fine as long as it's not already being used in the predicate.)

8. (Old value before an **`if-else`**)

   a. $sp(\mathtt{x} = \mathtt{x}_0, \textbf{if } \mathtt{x} < 0 \textbf{ then } \mathtt{x} := -\mathtt{x} \textbf{ fi})$

   $\equiv sp(\mathtt{x} = \mathtt{x}_0 \wedge \mathtt{x} < 0, \mathtt{x} := -\mathtt{x}) \vee sp(\mathtt{x} = \mathtt{x}_0 \wedge \mathtt{x} \ge 0, \textbf{skip fi})$

   $\equiv (\mathtt{x}_0 < 0 \wedge \mathtt{x} = -\mathtt{x}_0) \vee (\mathtt{x} \ge 0 \wedge \mathtt{x} = \mathtt{x}_0)$

   b. We can simplify $(\mathtt{x}_0 < 0 \wedge \mathtt{x} = -\mathtt{x}_0) \vee (\mathtt{x} \ge 0 \wedge \mathtt{x} = \mathtt{x}_0) \Leftrightarrow \mathtt{x} = |\mathtt{x}_0|$.

   c. If we had calculated

   $sp(\mathtt{T}, \textbf{if } \mathtt{x} < 0 \textbf{ then } \mathtt{x} := -\mathtt{x} \textbf{ fi})$

   $\equiv sp(\mathtt{T} \wedge \mathtt{x} < 0, \mathtt{x} := -\mathtt{x}) \vee sp(\mathtt{T} \wedge \mathtt{x} \ge 0, \textbf{skip fi})$

   $\equiv (\mathtt{x}_0 < 0 \wedge \mathtt{x} = -\mathtt{x}_0) \vee \mathtt{x} \ge 0$

   Then we would have lost the information about the **`else`** clause not changing `x`, so we wouldn't have been able to conclude $\mathtt{x} = |\mathtt{x}_0|$.

 