Forward Assignment; Strongest Postconditions

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 class 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, \( P(e) \) \( \triangleright\triangleright \) \( e \{ P(v) \} \) where \( P \) is a predicate function. If we just use the body of \( P \) as the predicate, the rule is \( \{ \text{body of } P \}[e/v] \) \( \triangleright\triangleright \) \( e \{ P \} \).
  - Since \( p[e/v] = wlp(v := e, p) \), 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 \( \{ p \} v := e \{ p \land v = e \} \), 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 \( \{ p \} v := e \{ p \land v = e \} \).
  - For example, \( \{ x > y \} z := 2 \{ x > y \land z = 2 \} \)
  - To justify this, using \( wlp \), we know \( \{ p \land v = e \}[e/v] \) \( \triangleright\triangleright \) \( e \{ p \land v = e \} \).
    - Expanding, \( (p \land v = e)[e/v] = p[e/v] \land e = e[e/v] \).
    - Since \( v \) is fresh, it doesn’t occur in \( p \) or \( e \), so \( p[e/v] = p \) and \( e[e/v] = e \). So we need \( \{ p \land e = e \} \) \( v := e \{ p \land v = e \} \), which definitely holds.

Forward Assignment (General Case)
- As an example of why \( \{ p \} v := e \{ p \land v = e \} \) doesn’t work in general, consider \( \{ x > 0 \} x := x/2 \{ ??? \} \).
  - Since division truncates, if \( x = 1 \), the final state doesn’t satisfy \( x > 0 \). We might want to try \( x*2 > 0 \) as the postcondition, but again this fails if \( x = 1 \).
• What we do know is that the value \( x \) had before the assignment was positive, and the new value of \( x \) is half of the old value of \( x \). If we use \( x_0 \) as the name for the value of \( x \) before the assignment, we can say \( \{ x_0 = x \land x > 0 \} \ x := x/2 \ \{ x_0 > 0 \land x = x_0/2 \} \).

• Note we don’t have to actually store \( x_0 \) in memory; it’s just a name we use for logical reasoning purposes — \( x_0 \) 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 \( x \) changes, \( x_0 \) doesn’t. (Note in this context, “logical” doesn’t mean “boolean”.)

• The general rule is \( \{ p \land v = v_0 \} \ v := e \ \{ p[v_0/v] \land v = e[v_0/v] \} \). In the precondition, we may be able to omit \( v = v_0 \) as being understood.

  • Example 1a: \( \{ x > 0 \land x = x_0 \} \ x := x-1 \ \{ x_0 > 0 \land x = x_0 - 1 \} \)

  • Example 2a: \( \{ s = sum(0,i) \land s = s_0 \} \ s := s+i+1 \ \{ s_0 = sum(0,i) \land s = s_0 + i + 1 \} \)

• The \( x = x_0 \) and \( s = s_0 \) clauses are a bit annoying; we could drop them by using an existential in the postcondition, but that’s no fun either:

  • Example 1b: \( \{ x > 0 \} \ x := x-1 \ \{ \exists x_0. \ x_0 > 0 \land x = x_0 - 1 \} \)

  • Example 2b: \( \{ s = sum(0,i) \} \ s := s+i+1 \ \{ \exists s_0. \ s_0 = sum(0,i) \land s = s_0 + i + 1 \} \)

• 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 \( x \), there is an \( x_0 \) such that) \( \{ x > 0 \} \ x := x-1 \ \{ x_0 > 0 \land x = x_0 - 1 \} \)

• Example 2c: (For all \( s \) and \( i \), there is an \( s_0 \) such that)

  \( \{ s = sum(0,i) \} \ s := s+i+1 \ \{ s_0 = sum(0,i) \land s = s_0 + i + 1 \} \)

• Example 3: \( \{ s_0 = sum(0, i) \land s = s_0 + i + 1 \} \ i := i + 1 \ \{ s_0 = sum(0, i_0) \land s = s_0 + i_0 + 1 \land i = i_0 + 1 \} \). The postcondition of this example can be weakened to \( s = sum(0, i) \), which will be useful later for explaining the loop \( \text{while } i < n \ do \ s := s+i+1; \ i := i+1 \ od \).

• Using the backward assignment rule on \( \{ p \} \ s := s+i+1; \ i := i+1 \ \{ s = sum(0, i) \} \) gets us a logically equivalent triple:

  \[
  p = wp(s := s+i+1; \ i := i+1, \ s = sum(0, i))
  \]

  \[
  = wp(s := s+i+1, \ wp(i := i+1, \ s = sum(0, i)))
  \]

  \[
  = wp(s := s+i+1, \ s = sum(0, i+1))
  \]

  \[
  = s+i+1 = sum(0, i+1)
  \]

  \[
  \iff s = sum(0, i)
  \]

**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)**
  \[ \vdash \{ p \land v = v_0 \} v := e \{ p[v_0/v] \land v = e[v_0/v] \}, \text{where } v_0 \text{ is a fresh logical constant.} \]

- **Proof:**
  - We'll argue that the precondition \( p \land v = v_0 \) implies the \( wp \) of the assignment and postcondition.
  - We need \( (p \land v = v_0) \rightarrow wp(v := e, q) \), where \( q = (p[v_0/v] \land v = e[v_0/v]) \). Simplifying,
    \[ wp(v := e, q) = q[e/v] = (p[v_0/v] \land v = e[v_0/v])[e/v] = p[v_0/v][e/v] \land v[e/v] = e[v_0/v][e/v] \]
    
  - We can simplify \( p[v_0/v][e/v] \land v[e/v] = e[v_0/v][e/v] \) by looking at its three parts.
    - The easiest one is \( v[e/v] = e \).
    - For \( p[v_0/v] \), since \( v \) doesn't appear in it, we have \( p[v_0/v][e/v] = p[v_0/v] \), which is \( \iff p \land v = v_0 \).
    - 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 \( (p[v_0/v][e/v] \land v[e/v] = e[v_0/v][e/v]) \iff p[v_0/v] \land e = e[v_0/v] \)
  - So \( (p[v_0/v][e/v] \land v[e/v] = e[v_0/v][e/v]) \iff p \land v = v_0 \land e = e \iff p \land v = v_0 \).
  - \( p[v_0/v][e/v] \land v[e/v] = e[v_0/v][e/v] = p[v_0/v] \land e = e[v_0/v] \)
    and this is \( \iff p \land v = v_0 \).
  - For a particular example, with \( \{ x > 0 \land x = x_0 \} x := x-1 \{ x_0 > 0 \land x = x_0 - 1 \} \), we find \( wp(x := x-1, x_0 > 0 \land x = x_0 - 1) = x_0 > 0 \land x = x_0 - 1 \), which is implied by \( x > 0 \land x = x_0 \).
  
  - Note that the simpler rule for introducing a new variable is a special case of the general rule:
    - We want \( \{ p \} v := e \{ p \land v = e \} \) if \( v \) doesn't occur in \( e \) or and \( v \) is not free in \( p \). The forward assignment rule says \( \{ p \} v := e \{ p[v_0/v] \land v = e[v_0/v] \} \), where \( v_0 \) is a fresh logical constant.
      Since \( v \) does not occur in \( e \), we know \( e[v_0/v] = e \). Similarly, since \( v \) isn't free in \( p \), we know \( p[v_0/v] = p \). Substituting into \( \{ p \} v := e \{ p[v_0/v] \land v = e[v_0/v] \} \) gives us \( \{ p \} v := e \{ p \land v = e \} \).
  
  - 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.

**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,
- \( sp(p, S) = \{ \tau \mid \tau \in M(S, \sigma) \cap \Sigma \text{ for some } \sigma \text{ where } \sigma \models p \} \).
- Equivalently, \( sp(p, S) = \cup (M(S, \sigma) \cap \Sigma) \text{ where } \sigma \models p \).

I.e., \( sp(p, S) \) is the image of \( M(S, ...) \) (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 \( \sigma \models p \), 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: \( \models_{\text{tot}} \{ p \} S \{ sp(p, S) \} \) iff \( \models_{\text{tot}} \{ p \} S \{ T \} \).)

• For \( sp(\neg p, S) \), the situation is symmetric:

  • If \( \sigma \models \neg p \), 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: \( sp(p, S) \cap sp(\neg p, S) \neq \emptyset \), just means it's possible for states satisfying \( p \) and \( \neg p \) to map to a common final state. (E.g., \( x := 0 \) sets \( x \) to 0 regardless of \( p \) or \( \neg p \).)

\[ \begin{align*}
\text{Figure 1: } sp(p, S) \text{ is the set of states reachable via } S \text{ from } p \\
\text{Figure 2: } sp(\neg p, S) \text{ is the set of states reachable via } S \text{ from } \neg p
\end{align*} \]

• **Example 4**: For an example where \( sp(p, S) \) and \( sp(\neg p, S) \) overlap, take \( S = x := 1 \), which produces the same final state regardless of the start state: \( sp(p, x := 1) = sp(\neg p, x := 1) = x = 1 \).

• **Example 5**: Let \( W = \text{while } i \neq 0 \text{ do } i := i - 1 \text{ od} \). Clearly, if \( W \) terminates, it's with \( i = 0 \), so for any precondition \( p \), if \( i \) isn't free in \( p \), we have \( sp(p, W) \Leftrightarrow p \land i = 0 \), so \( \{ p \} W \{ p \land i = 0 \} \) is partially correct. However, it's not totally correct for every \( p \): It terminates iff we start in a state with \( i \geq 0 \) (i.e., \( \models_{\text{tot}} \{ i \geq 0 \} W \{ T \} \)). So \( \{ i \geq 0 \} W \{ i = 0 \} \) is totally correct. However, \( \{ i < 0 \} W \{ i = 0 \} \) is partially correct but never totally correct.

• **Strongest** postcondition: For partial correctness, \( \models \{ p \} S \{ sp(p, S) \} \) (i.e., \( sp(p, S) \) is a postcondition). What makes it the strongest postcondition is that it implies any other
Strongest Postconditions for Loop-Free Programs

Recall the forward assignment rule:

\[ \{ p \land v = v_0 \} v := e \{ p[v_0/v] \land v = e[v_0/v] \} \]

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:

\[ \text{sp}(p, \text{skip}) = p \]
\[ \text{sp}(p, v := e) = p[v_0/v] \land v = e[v_0/v], \text{ where } v_0 \text{ is a fresh constant.} \]
\[ \text{sp}(p, S_1; S_2) = \text{sp}(\text{sp}(p, S_1), S_2) \]

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 \( S_1 \).

\[ \text{sp}(p, \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}) = \text{sp}(p \land B, S_1) \lor \text{sp}(p \land \neg B, S_2) \]

After executing an \textit{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 \( p, B, S_1, \text{ and } S_2 \), we might be able to infer which branch was taken, but in the
general case, we can’t.

\[ \text{sp}(p, \text{if } B_1 \rightarrow S_1 \square B_2 \rightarrow S_2 \text{ fi}) = \text{sp}(p \land B_1, S_1) \lor \text{sp}(p \land B_2, S_2). \]

As with deterministic \textit{if-else}, once the nondeterministic \textit{if-fi} finishes, we know it evaluated one of the guarded commands but
not necessarily which one.

\textbf{Explicitly naming old variables before conditionals:}

\textbullet{} E.g., consider \textit{if } \( B \text{ then } x := x + 1 \text{ else skip fi}. \) (Assume \( x \) doesn’t appear in \( B. \))

\textbullet{} We definitely need to include \( x_0 \) when taking the sp of the true branch to get \((B \rightarrow x = x_0 + 1).\)

\textbullet{} But we should also include \( x_0 \) in the sp of the false branch to get \((\neg B \rightarrow x = x_0)\) so that after
the conditional, we know what \( x \) looks like after the conditional, regardless of \( B. \) Without
including \( x = x_0 \) in the false branch sp, we’d just know \( \neg B. \)

\textbullet{} In short, for \( \text{sp}(p, \text{if ... fi}), \) we should add old variable clauses for all variables modified by the
branches.

\textbullet{} For nondeterministic \textit{if... fi}, the principle is the same: Before the conditional, add old
variable clauses for all variables modified in any of the guarded commands.
F. Examples of Strongest Postconditions

Example 6:

- Let's use \( sp \) to fill in the postcondition of \( \{ x > y \} \): 
  \( x := x + k; \ y := y + k \). 
- (Presumably, the postcondition will imply that \( x \) is still > \( y \).)
- Let \( p = x > y \), then \( sp(p, x := x + k; \ y := y + k) = sp(sp(p, x := x + k), y := y + k) \).
- Let's calculate the inner \( sp \) first:
  \[
  sp(p, x := x + k) \\
  = sp(x > y, x := x + k) \\
  = (x > y)[x_0/x] \land x = (x + k)[x_0/x] \\
  = x_0 > y \land x = x_0 + k
  \]
- Then going back to our original problem and the outer \( sp \), we get
  \[
  sp(p, x := x + k; \ y := y + k) \\
  = sp(sp(p, x := x + k), y := y + k) \\
  = sp(x_0 > y \land x = x_0 + k, y := y + k) \\
  = (x_0 > y \land x = x_0 + k)[y_0/y] \land y = (y + k)[y_0/y] \\
  = x_0 > y_0 \land x = x_0 + k \land y = y_0 + k
  \]
- So our original triple can be filled in as
  \[
  \{ x > y \} \ x := x + k; \ y := y + k \ \{ x_0 > y_0 \land x = x_0 + k \land y = y_0 + k \}
  \]

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 \( x > y \):
  \[
  wp(x := x + k; \ y := y + k, x > y) \\
  = wp(x := x + k, wp(y := y + k, x > y)) \\
  = wp(x := x + k, (x > y)[y+k/y]) \\
  = wp(x := x + k, (x > y + k)) \\
  = (x > y + k)[x+k/x] \\
  = x + k > y + k
  \]
- Which gives us \( \{ x+k > y+k \} \ x := x+k; \ y := y+k \ \{ x > y \} \)

Example 8:

- Let \( p = y = y_0 \) and \( S = if \ x \geq 0 \ then \ y := x \ else \ y := -x fi \). Then
  - \( sp(p \land x \geq 0, \ y := x) = (x \geq 0)[y_0/y] \land y = x[y_0/y] = x \geq 0 \land y = x \)
  - \( sp(p \land x < 0, \ y := -x) = (x < 0)[y_0/y] \land y = -x[y_0/y] = x < 0 \land y = -x \)
- So \( sp(y = y_0, if \ x \geq 0 \ then \ y := x \ else \ y := -x fi) \)
  \[
  = sp(p \land x \geq 0, \ y := x) \lor sp(p \land x < 0, \ y := -x) \\
  = x \geq 0 \land y = x \lor x < 0 \land y = -x
  \]
- Note that since the assignments to \( y \) don't depend on \( y_0 \), we don't find \( y_0 \) occurring in the result.

Example 9:
Let $S = \text{if even}(x) \text{ then } x := x+1 \text{ else } x := x+2 \text{ fi}$ and let $p = x \geq 0$, then

$$sp(p, S) = sp(p, \text{if even}(x) \text{ then } x := x+1 \text{ else } x := x+2 \text{ fi})$$

$$= sp(p \land \text{even}(x), x := x+1) \lor sp(p \land \text{odd}(x), x := x+2)$$

For the true branch,

$$sp(p \land \text{even}(x), x := x+1)$$

$$= (p \land \text{even}(x))[x_0/x] \land x = (x+1)[x_0/x]$$

$$= (x \geq 0 \land \text{even}(x))[x_0/x] \land x = (x+1)[x_0/x]$$

$$= x_0 \geq 0 \land \text{even}(x_0) \land x = x_0+1$$

For the false branch,

$$sp(p \land \text{odd}(x), x := x+2)$$

$$= (p \land \text{odd}(x))[x_0/x] \land x = (x+2)[x_0/x]$$

$$= (x \geq 0 \land \text{odd}(x))[x_0/x] \land x = x_0+2$$

$$= x_0 \geq 0 \land \text{odd}(x_0) \land x = x_0+2$$

So $sp(p, S) = sp(p \land \text{even}(x), x := x+1) \lor sp(p \land \text{odd}(x), x := x+2)$

$$= (x_0 \geq 0 \land \text{even}(x_0)) \land x = x_0+1 \lor (x_0 \geq 0 \land \text{odd}(x_0)) \land x = x_0+2$$

If we want to logically simplify, we get $x_0 \geq 0 \land (\text{even}(x_0) \land x = x_0+1 \lor \text{odd}(x_0) \land x = x_0+2)$.

That's about as far as we can simplify without losing information. For example, we could simplify to $(x_0 \geq 0 \land (x = x_0 + 1 \lor x = x_0 + 2))$ or to $(x \geq 1 \land \text{odd}(x))$, 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.

**Example 10:**

Let $S = \text{if } x \geq y \text{ then } x := x-y \text{ else } y := y-x \text{ fi}$

Let $q_1 = sp(x = x_0 \land y = y_0, S)$

$$= sp(x = x_0 \land y = y_0 \land x \geq y, x := x-y) \lor sp(x = x_0 \land y = y_0 \land x < y, y := y-x)$$

$$= (y = y_0 \land x_0 \geq y \land x = x_0+y) \lor (x = x_0 \land x < y_0 \land y = y_0-x)$$

With $q_1$, if the first disjunct holds, then $x_0 \geq y_0$; if the second disjunct holds, then $x_0 < y_0$.

If we only introduce $x_0$ and $y_0$ just before the assignments to $x$ and $y$, we get

• For the true branch, $sp(x = x_0 \land x \geq y, x := x-y) = (x_0 \geq y \land x = x_0+y)$

• For the false branch, $sp(y = y_0 \land x < y, y := y-x) = (x < y_0 \land y = y_0-x)$

Let $q_2$ be their disjunction, $(x_0 \geq y \land x = x_0+y) \lor (x < y_0 \land y = y_0-x)$.

With $q_2$, if the first disjunct holds, we know $x_0 \geq y$, but not $x_0 \leq y_0$. Symmetrically, if the second conjunct holds, we know $x \leq y_0$, but not $x_0 \geq y_0$. (We certainly know $x_0 \geq y_0$ or $x_0 \leq y_0$, 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.
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 \( \{x > f(x, y) \} \ x := x+1; \ x := x*x \) \{??\}, 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 \( sp(x > f(x, y), S_1, S_2) = sp(sp(x > f(x, y), S_1), S_2) \) where \( S_1 = x := x+1 \) and \( S_2 = x := x*x \).

\[
\begin{align*}
sp(x > f(x, y), S_1) &= sp(x > f(x, y), x := x+1) \\
&= (x > f(x, y))[x_0/x] \land x = (x+1)[x_0/x] \\
&= x_0 > f(x_0, y) \land x = x_0+1
\end{align*}
\]

\[
\begin{align*}
sp(sp(x > f(x, y), S_1), S_2) &= sp(x > f(x_0, y) \land x = x_0+1, x := x*x) \\
&= (x_0 > f(x_0, y) \land x = x_0+1)[x_1/x] \land x = (x*x)[x_1/x] \\
&= x_0 > f(x_0, y) \land x_1 = x_0+1 \land x = x_1*x_1
\end{align*}
\]

- 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

A. 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.

B. 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.

C. 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(y ≥ 0, skip)
3. sp(i > 0, i := i+1) [Hint: add an i = i₀ conjunct to i > 0]
4. sp(k ≤ n ∧ s = f(k, n), k := k+1)
5. sp(T, i := 0; k := i) [Go ahead and logically simplify as you go by dropping uses of “T ∧” in the predicates]
6. sp(i ≤ j ∧ j-i < n, i := i+j; j := i+j).
7. sp(0 ≤ i < n ∧ s = sum(0, i), s := s+i+1; i := i+1)
8. Let S = if x < 0 then x := -x fi
   a. Calculate sp(x = x₀, S).
   b. Logically simplify your result from part (a). Feel free to use the function abs(...) or |...|.
   c. Suppose we had calculated sp(T, if x < 0 then x := -x fi) introducing x₀ in the true branch only. What would we get for the sp and what is the problem with it?
Solution to Practice 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. $y \geq 0$ (For the skip rule, the precondition and postcondition are the same.)

3. Let's implicitly add $i = i_0$ to the precondition, to name the starting value of $i$. Then
   
   
   $sp(i > 0, i := i+1)$
   
   $\equiv (i > 0)[i_0/i] \land i = (i+1)[i_0/i]$  
   
   $\equiv i_0 > 0 \land i = i_0 + 1$

4. As in the previous problem, let's introduce a variable $k_0$ to name the starting value of $k$. Then
   
   $sp(k \leq n \land s = f(k, n), k := k+1)$
   
   $\equiv (k \leq n \land s = f(k, n))[k_0/k] \land k = (k+1)[k_0/k]$  
   
   $\equiv k_0 \leq n \land s = f(k_0, n) \land k = k_0 + 1$

5. We don't need to introduce names for the old values of $i$ and $k$ (they're irrelevant).
   
   $sp(T, i := 0; k := i)$
   
   $\equiv sp(sp(T, i := 0), k := i)$

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

   $\equiv i = 0 \land k = i$

6. Let's introduce $i_0$ and $j_0$ as we need them, then
   
   $sp(i \leq j \land j-i < n, i := i+j; j := i+j)$
   
   $\equiv sp(sp(i \leq j \land j-i < n, i := i+j), j := i+j)$

   $\equiv sp(i_0 \leq j \land j-i_0 < n \land i = i_0 + j, j := i+j)$

   $\equiv i_0 \leq j_0 \land j_0-i_0 < n \land i = i_0 + j_0 \land j = i + j_0$

7. $sp(0 \leq i < n \land s = sum(0, i), s := s+i+1; i := i+1)$ [10/27 - had swapped the arguments]
   
   $\equiv sp(sp(0 \leq i < n \land s = sum(0, i), s := s+i+1, i := i+1))$

   For the inner sp,

   $sp(0 \leq i < n \land s = sum(0, i), s := s+i+1)$

   $\equiv 0 \leq i < n \land s_0 = sum(0, i) \land s = s_0+i+1$  Using $s_0$ to name the old value of $s$

   Returning to the outer sp,

   $sp(sp(0 \leq i < n \land s = sum(0, i), s := s+i+1), i := i+1)$

   $\equiv sp(0 \leq i < n \land s_0 = sum(0, i) \land s = s_0+i+1, i := i+1)$
\[ 0 \leq i' < n \land s_0 = \text{sum}(0, i') \land s = s_0 + i' + 1 \land i = i' + 1 \]

Using \( i' \) to name the old value of \( i \)

(There's no particular reason I used \( i' \) here except that; any other name like \( i_0 \) 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. \( sp(x = x_0, \text{if } x < 0 \text{ then } x := -x \text{ fi}) \)
      \[ = sp(x = x_0 \land x < 0, x := -x) \lor sp(x = x_0 \land x \geq 0, \text{skip fi}) \]
      \[ = (x_0 < 0 \land x = -x_0) \lor (x \geq 0 \land x = x_0) \]
   b. We can simplify \((x_0 < 0 \land x = -x_0) \lor (x \geq 0 \land x = x_0) \iff x = |x_0| \cdot \)
   c. If we had calculated
      \[ sp(T, \text{if } x < 0 \text{ then } x := -x \text{ fi}) \]
      \[ = sp(T \land x < 0, x := -x) \lor sp(T \land x \geq 0, \text{skip fi}) \]
      \[ = (x_0 < 0 \land x = -x_0) \lor x \geq 0 \]

Then we would have lost the information about the else clause not changing \( x \), so we wouldn't have been able to conclude \( x = |x_0| \cdot \)