Forward Assignment; Strongest Postconditions

CS 536: Science of Programming, Fall 2022

2022-10-06: pp. 2 – 5, 7, 2022-11-07: p.7

A. Why?

- Sometimes, the forward version of the assignment rule is preferable to the backward version.
- The forward assignment rule is part of calculating the *sp* (strongest postcondition) of a loop-free program.
- The *sp* is the postcondition that includes all possible results of a program under a precondition.

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, $\{P(e)\}\ v := e\ \{P(v)\}$ where P is a predicate function. If we just use the body of P as the predicate, the rule is $\{(body\_of\_P)[e/v]\}\ v := e\ \{P\}$.
  - Since $p[e/v] \equiv wlp(v := e, p)$, 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, $\{x > y\}\ z := 2\ \{x > y \wedge z = 2\}$
- 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 certainly holds.

Forward Assignment - General Case

- As an example of why $\{p\}\ v := e\ \{p \wedge v = e\}$ doesn't work in general, consider $\{x > 0\}\ x := x - 2\ \{???\}$.
  - We certainly don't have $\{x > 0\}\ x := x - 2\ \{x > 0 \wedge x = x - 2\}$. If we look more carefully, the relationship we're trying to capture with $x > 0 \wedge x = x - 2$ is:

    (value of x before asgt) > 0 ∧ (the current value of x) = (value of x before asgt) – 2

- This example uses subtraction, which is invertible, so we can write $(x+2 > 0 \wedge x = (x+2) - 2)$ for the postcondition.
- But not all assignments are invertible: Consider $\{x > 0\}\ x := x/2\ \{???\}$. Because of truncating integer division, $(2*x > 0 \wedge x = (2*x/2))$ is only true for even values of x.
- What we can do instead is to introduce a name for *(the value of x before the assignment)*. If we use x_0 as this name, we can say $\{x_0 = x \wedge x > 0\}\ x := x/2\ \{x_0 > 0 \wedge x = x_0/2\}$.
- ***Definition: Aging*** x is the process of introducing a logical constant to name the value of x before a change.
- 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 Forward Assignment Rule

- The general rule for forward assignment is $\{p \wedge v = v_0\}\ v := e\ \{p[v_0/v] \wedge v = e[v_0/v]\}$. If it's omitted, the $v = v_0$ part of the precondition is understood.
  - ***Example 1a***: $\{x > 0 \wedge x = x_0\}\ x := x - 1\ \{x_0 > 0 \wedge x = x_0 - 1\}$.
  - ***Example 2a***: $\{s = sum(0, i) \wedge s = s_0\}\ s := s + i + 1\ \{s_0 = sum(0, i) \wedge s = s_0 + i + 1\}$.
- Aging x and s using the $x = x_0$ and $s = s_0$ clauses is a bit annoying; we can drop them by using an existential in the postcondition, but that's no fun either:
  - ***Example 1b***: $\{x > 0\}\ x := x - 1\ \{\exists x_0 . x_0 > 0 \wedge x = x_0 - 1\}$
  - ***Example 2b***: $\{s = sum(0, i)\}\ s := s + i + 1\ \{\exists s_0 . s_0 = sum(0, i) \wedge s = s_0 + i + 1\}$.
- 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) $\{x > 0\}\ x := x - 1\ \{x_0 > 0 \wedge x = x_0 - 1\}$
  - ***Example 2c***: (For all s and i, there is an s_0 such that) $\{s = sum(0, i)\}\ s := s + i + 1\ \{s_0 = sum(0, i) \wedge s = s_0 + i + 1\}$.
- ***Example 3:*** (For all s, s_0, and i, there is an i_0 such that)
  $\{s_0 = sum(0, i) \wedge s = s_0 + i + 1\}\ \ i := i + 1\ \{s_0 = sum(0, i_0) \wedge s = s_0 + i_0 + 1 \wedge i = i_0 + 1\}$.
- ***Discussion:*** *Simplifying the postcondition; Equivalence with wp*
  - The postcondition of Example 3 can be weakened to $s = sum(0, i)$. Combining Examples 2c and 3 gives us $\{s = sum(0, i)\}\ s := s + i + 1\ ;\ i := i+1\ \{s = sum(0, i)\}$. [2022-10-06]
  - Using backward assignment to calculate p in $\{p\}\ s := s+i+1;\ i := i+1\ \{s = sum(0, i)\}$ produces the same triple (after simplification)

$p \equiv wp(s := s+i+1;\ i := i+1,\ s = sum(0, i))$

$\equiv wp(s := s+i+1,\ wp(i := i+1,\ s = sum(0, i)))$

$\equiv wp(s := s+i+1,\ s = sum(0, i+1))$

$\equiv s+i+1 = sum(0, i+1)$ Finishes calculation of *sp*

$\Leftrightarrow s = sum(0, i)$ Logical simplification

D. Correctness of the Assignment Rules

- This section is mostly technical. The key takeaway is that the forward and backward assignment rules are equally strong because you can derive each from the other. In addition, new variable introduction is just a special case of forward assignment.
- ***Discussion:***
  - Combining Examples 2c and 3 above and weakening the postcondition gives us the triple
    $\{s = sum(0, i)\}\ s := s+i+1;\ i := i+1\ \{s = sum(0, i)\}$
  - It turns out that *wp* can be used on the same program and postcondition to produce the same triple after precondition strengthening.
  - This is not accidental: The forward and backward assignment rules are equivalent in power in the sense that anything proved using forward assignment can also be proved using backward assignment, and vice versa.
  - The standard way to argue this is to show how a triple obtained using one assignment rule can be derived using the other assignment rule./

Derivation of the Forward Assignment Rule from the Backward 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*** (***Forward Assignment):*** $\models \{p \wedge v = v_0\}\ v := e\ \{p[v_0/v] \wedge v = e[v_0/v]\}$, where v_0 is a fresh logical constant.
- ***Proof***: [2022-10-06] Forward assignment tells us the triple $\{p \wedge v = v_0\}\ v := e\ \{p[v_0/v] \wedge v = e[v_0/v]\}$ is correct. We'd like to prove the same triple using backward assignment. Applying the backward assignment rule to the given postcondition $p[v_0/v] \wedge v = e[v_0/v]$ will gives us a precondition that makes for a correct triple, but it won't be syntactically the same as $p \wedge v = v_0$. It will, however, be logically equivalent to $p \wedge v = v_0$, and that will let us conclude that forward assignment tells us that our same original triple, $\{p \wedge v = v_0\}\ v := e\ \{p[v_0/v] \wedge v = e[v_0/v]$, is correct. [end]
- With backward assignment, we know $\{wlp(v := e, q)\}\ v := e\ \{q\}$ where $q \equiv p[v_0/v] \wedge v = e[v_0/v]$. The only occurrence of v within $p[v_0/v] \wedge v = e[v_0/v]$ is the v in $v = \ldots$. This makes $wlp(v := e, q) \equiv q[e/v]$ $\Leftrightarrow p[v_0/v] \wedge e = e[v_0/v]$. Is this implied by $p \wedge v = v_0$? Yes: $p \wedge v = v_0$ implies $p[v_0/v]$, and $v = v_0$ implies $e = e[v/v] = e[v_0/v]$. Since $(p \wedge v = v_0) \rightarrow wlp(v := e, q)$, precondition strengthening tells us $\{p \wedge v = v_0\}\ v := e\ \{q\}$. So the backward assignment rule justifies the forward assignment rule.

- For a particular example, with $\{x>0 \wedge x=x_0\}\ x:=x-1\ \{x_0>0 \wedge x=x_0-1\}$, we find $wlp(x := x-1,\ x_0>0 \wedge x=x_0-1) \equiv x_0>0 \wedge x-1=x_0-1$, which is implied by $x>0 \wedge x=x_0$.

Derivation of New Variable Introduction

- The simpler rule for introducing a new variable is a special case of forward assignment.
- We want $\{p\}\ v := e\ \{p \wedge v=e\}$ if v doesn't occur in e or and v is not free in p. By forward assignment, $\{p\}\ v:=e\ \{p[v_0/v] \wedge v=e[v_0/v]\}$, 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 $\{p\}\ v:=e\ \{p \wedge v=e\}$.

Derivation of the Backward Assignment Rule from the Forward Assignment Rule

- We know the forward assignment rule can be derived from the backward assignment rule. The converse is also true: We can derive the forward assignment rule from the backward assignment rule.
- ***Theorem (Backward Assignment)***: $\models \{p[e/v]\}\ v:=e\ \{p\}$ follows from the forward assignment rule.
- ***Proof:*** Using forward assignment on the precondition $p[e/v] \wedge v = v_0$ and assignment $v:=e$ gives us the postcondition $p[e/v][v_0/v] \wedge v = e[v_0/v]$. To justify backward assignment, we need this last predicate to imply $p[e/v]$. *[2022-10-06]*
- In $p[e/v]$, the only occurrences of v are the ones in e, so in $p[e/v][v_0/v]$, the only occurrences of v_0 are the ones that replace the v's in e.
- Thus $p[e/v][v_0/v]$ is logically equivalent to $p[e[v_0/v] / v]$ (where we replace the v's in e with v_0's and then replace the v's in p with the result). Let $e' \equiv e[v_0/v]$. Now, if $v = e'$, then $(p[e'/v] \wedge v = e')$ is equivalent to $p \wedge v = e'$, which implies $p[e/v]$. *[2022-10-06]* So the backward assignment rule can be derived from the forward assignment rule.

E. 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,
  - $sp(p,S) = \{\tau \mid \tau \in M(S,\sigma) - \bot$ for some σ where $\sigma \models p\}$.
  - Equivalently, $sp(p,S) = \cup_\sigma (M(S,\sigma) - \bot)$ where $\sigma \models p$.
- If we treat $M(S,\ldots) - \bot$ as a function over states, then $sp(p,S)$ is the image of this function over the states that satisfy p.
- Figure 1 shows the relationship between p, S, and $sp(p,S)$:
  - If $\sigma \models p$, then every state in $M(S, \sigma) - \bot$ is by definition in $sp(p,S)$, so $\models \{p\}\ S\ \{sp(p,S)\}$.
    - This is only valid for ***partial correctness***: Starting in a state that satisfies p might yield $\bot$.
    - To get total correctness, $\models_{tot} \{p\}\ S\ \{sp(p,S)\}$, we need termination, $\models_{tot} \{p\}\ S\ \{T\}$.
- ***Example 4***: Let $W \equiv$ ***while*** $i \neq 0$ ***do*** $i:=i-1$ ***od***, then $sp(i \geq 0,\ W) \equiv i = 0$, and we find that $\{i \geq 0\}\ W\ \{sp(i \geq 0,\ W)\}$ is not only partially correct, it's totally correct.

- ***Example 5***: For the same W, weakening $i \geq 0$ produces the same sp. At the limit, $sp(T, W) \equiv i = 0$. Here, the sp is partially correct but not totally correct: $\models \{T\}\ W\ \{sp(T, W)\}$ but $\not\models_{tot} \{T\}\ W\ \{sp(T, W)\}$. Of course, this is because W doesn't terminate when one starts with $i < 0$.
- ***Why strongest***? For partial correctness, $sp(p, S)$ is a postcondition. What makes it the *strongest* postcondition is that it implies any other postcondition: for any q, $\models \{p\}S\{q\}$ iff $\models sp(p, S) \rightarrow q$.

[2022-10-06]

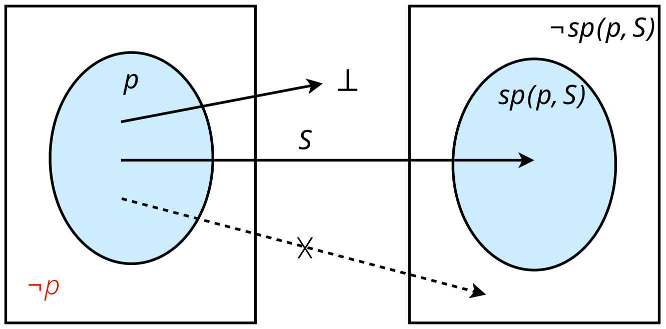

Figure 1: sp(p, S) is the set of states reachable via S from p

- ***Lemma:*** $\models \{p\}S\{q\}$ iff $\models sp(p, S) \rightarrow q$.
  - The $\Leftarrow$ direction holds by postcondition weakening: We have $\models \{p\}\ S\ \{sp(p, S)\}$ and $\models sp(p, S) \rightarrow q$, therefore $\models \{p\}\ S\ \{q\}$.
  - For the $\Rightarrow$ direction, assume $\models \{p\}S\{q\}$ and let $\tau \models sp(p, S)$. Since $\tau \models sp(p, S)$, we have $\tau \in M(S, \sigma) - \perp$ for some $\sigma \models p$. But $\sigma \models \{p\}S\{q\}$ tells us that $M(S, \sigma) - \perp \models q$, so $\tau \models q$. So $\tau \models sp(p, S)$ implies $\tau \models q$, so we have $\models sp(p, S) \rightarrow q$.

F. Calculating Strongest Postconditions of Loop-Free Programs

Definition: (Calculation of sp, part 1):

- As with wlp, the sp of a program can be textually calculated for loop-free programs.
- The simplest cases for calculating sp are for the ***skip***, assignment, and sequence statements.
- $sp(p, \textbf{skip}) \equiv p$
  - Since ***skip*** doesn't change the state, whatever was true before the ***skip*** is true after it.
- $sp(p, v := e) \equiv p[v_0/v] \wedge v = e[v_0/v]$, where v_0 is a fresh constant (the "aged" version of v)
  - The forward assignment rule turns out to give the strongest description of the state after an assignment. We won't prove this formally, but intuitively, the value of v before the assignment isn't changed by an assignment to v: It's still the value of v before the assignment. So everything that was true about v_0 before the assignment is still true after the assignment. Similarly, the new value of v, when described relative to v_0, is the same before and after the assignment.
  - ***Example 6:*** $sp(x > y, x := x + k) \equiv (x > y)[x_0/x] \wedge x = (x + k)[x_0/x] \equiv x_0 > y \wedge x = x_0 + k$.

- ***Example 7:*** Here's the conclusion of Example 6 used as a postcondition for a different assignment: $sp(x_0 > y \wedge x = x_0 + k, y := y + k) \equiv (x_0 > y \wedge x = x_0 + k)[y_0/y] \wedge y = (y+k)[y_0/y]$

  $\equiv x_0 > y_0 \wedge x = x_0 + k \wedge y = y_0 + k$

- $sp(p, S_1; S_2) \equiv sp(sp(p, S_1), S_2)$
  - 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.
- ***Example 8:*** Combining Examples 6 and 7,

  $sp(p, x := x + k; y := y + k)$

$\equiv sp(sp(p, x := x + k), y := y + k)$	Defn sp of sequence
$\equiv sp(x_0 > y \wedge x = x_0 + k, y := y + k)$	Example 6
$\equiv x_0 > y_0 \wedge x = x_0 + k \wedge y = y_0 + k$	Example 7

  - If we don't want to keep the old values x_0 and y_0, we can weaken the *sp* to $x > y$ instead.
- 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.
- ***Example 9:*** To complete $\{x > f(x, y)\}\ x := x+1;\ x := x*x\ \{???\}$, we'll calculate the strongest postcondition.
- We need $sp(x > f(x, y), S_1; S_2) \equiv sp(sp(x > f(x, y), S_1), S_2)$ where $S_1 \equiv x := x+1$ and $S_2 \equiv x := x*x$. Because x is assigned to twice, there will be three versions of x: x_0 names the value x had before the first assignment, x_1 names the value x had between the two assignments, and x will end being the name of the value after the two assignments.

  $sp(x > f(x, y), S_1)$

  $\equiv sp(x > f(x, y), x := x+1)$

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

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

  $sp(sp(x > f(x, y), S_1), S_2)$

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

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

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

Strongest postconditions of conditional statements

- The *sp* of a conditional is the disjunction of the *sp*'s of its branches*. Disjunction is needed because, though execution will make one of those *sp*'s hold, when the conditional statement ends, we lose track of which branch was executed.
- Since the branches of a conditional can include assignments, bindings for initial values of variables ($x = x_0$, etc.) will be needed somewhere when calculating the *sp* of the conditional.
- However, instead of having these bindings turn up recursively, as we analyze the branches, they need to be part of the top level of calculation.

* Since they mean the same thing, I'm going to shorten "arms / branches" to just "branches".

- Let's look at an illustrative example before seeing how to calculate the *sp* of a conditional.
- ***Notation:*** An alternate style for indicating logical constants is to use capital letters. E.g., X instead of x_0. Which notation to use is a style issue[†]; let's try it for an example or two.
- *[2022-10-06]* ***Example 10:*** Let $IF \equiv$ ***if*** $x \geq y+z$ ***then*** $x := x-1$ ***else*** $y := y+2$ ***fi*** and let $p \equiv T$ be our precondition, then the *sp* of the true branch and false branch are

  $sp(T \wedge x \geq y+z,\ x := x-1) \equiv X \geq y+z \wedge x = X-1$ and $sp(T \wedge x < y+z,\ y := y+2) \equiv x < Y+z \wedge y = Y+2$ *[2022-11-07].*
- The disjunction of these two is $(X \geq y+z \wedge x = X-1) \vee (x < Y+z \wedge y = Y+2$ *[2022-11-07]*$)$, which doesn't include the information that the true branch doesn't modify y and the false branch doesn't modify x. So though it is a postcondition for T and IF, it's not the strongest one.
- To define $sp(p, IF)$, it will be handy to pre-calculate some things.
- ***Definitions:***
  - $lhs(S)$ = the set of variables that appear as the lhs of assignments in statement S.
  - $rhs(S)$ = the set of variables that appear in the rhs of assignments in S or in tests in S.
  - $free(p)$ = the set of variables that are free in predicate p.
  - $aged(p, S) = lhs(S) \cap (rhs(S) \cup free(p))$ is the subset of variables of S whose assignments cause aging.

Definition: (Calculation of sp, part 2):

- $sp(p, IF)$: Let $IF \equiv$ ***if*** B ***then*** S_1 ***else*** S_2 ***fi*** and let $aged(p, IF) = \{x, y, \ldots, z\}$, then

  $sp(p, IF) \equiv sp(p_0 \wedge B, S_1) \vee sp(p_0 \wedge \neg B, S_2)$ where $p_0 = p \wedge x = X \wedge y = Y \wedge \ldots \wedge z = Z$ *[2022-10-06]*
- The nondeterministic case is very similar. For $NF \equiv$ ***if*** $B_1 \rightarrow S_1 \square B_2 \rightarrow S_2$ ***fi***, p_0 is the same but $sp(p, NF) \equiv sp(p_0 \wedge B_1, S_1) \vee sp(p_0 \wedge B_2, S_2)$.

- *[2022-10-06]* ***Example 10 revisited****:* $p \equiv T$ and $IF \equiv$ ***if*** $x \geq y+z$ ***then*** $x := x-1$ ***else*** $y := y+2$ ***fi***, so $lhs(IF) = \{x, y\}$, $rhs(IF) = \{x, y, z\}$, and $free(p) \equiv \varnothing$, so $p \equiv x \geq y+z$, so $free(p) = \{x, y, z\}$. This makes $aged(x \geq y+z, IF) = \{x, y\} \cap (\{x, y, z\} \cup \varnothing) = \{x, y\}$, *so we'll add* $x = X \wedge y = Y$ as a conjunct to p to get $p_0 \equiv T \wedge x = X \wedge y = Y$, or simply $x = X \wedge y = Y$.

  $$sp(x = X \wedge y = Y, IF) \equiv (X \geq y+z \wedge y = Y \wedge x = X+1) \vee (x < Y+z \wedge x = X \wedge y = Y-3)$$
- This postcondition does include the information that the true branch modifies x but not y and the false branch modifies y but not x. This makes it stronger than Example 10's condition. *[end]*

- ***Fresh variables, generalized:*** Given a predicate p and an assignment $v := e$, we've said that v is a fresh variable if it doesn't appear in p or e. The definition $aged(p, S) = lhs(S) \cap (rhs(S) \cup$

[†] A typeset X is easier to read than x_0, but on paper, handwritten X and x can be confused if you're not careful. Also, if you need multiple logical names based on x, using x, x_0, x_1, x_2, … is easy but x, X, **X**, … gets out of hand very quickly.

free(p)) generalizes this. If all the assignments in *S* are to variables that aren't otherwise used in *S* and don't appear free in *p*, then the assignments are all to fresh variables.

- ***Example 11:*** $sp(T, \textbf{if } y \geq 1 \textbf{ then } x := 1 \textbf{ else } z := 0 \textbf{ fi})$
  $\equiv sp(y \geq 1, x := 1) \vee sp(y < 1, z := 0)$
  $\equiv (y \geq 1 \wedge x = 1) \vee (y < 1 \wedge z = 0)$.
  Here, $lhs(IF) = \{x, z\}$, $rhs(IF) = \{y\}$, $free(p) = \{y\}$, and $aged(y \geq 1, S) = \{x, z\} \cap (\{y\} \cup \{y\}) = \{x, z\} \cap \{y\} = \emptyset$.