A vueltas con la regla de verificación de la asignación.

En Programación imperativa, la asignación (y sus posibles variantes) es la única vía para hacer progresar los cómputos, por medio del cambio de estado que suponen.
La regla que define su Semántica operacional precisa de forma muy sencilla cuál es ese cambio $\langle x := a, s \rangle \to s[x \mapsto \mathcal{A}[\![a]\!]s]$.
Sin embargo, no es hasta que tenemos su regla de verificación, que comprendemos cuál es el papel que desempeña la asignación, en ese progreso hacia el resultado final perseguido.
Esta es la regla de verificación de la asignación:

$$[ass_p] \quad \{P[x \mapsto \mathcal{A}[\![a]\!]]\} \; x := a \; \{P\}$$

Cuando nos enfrentamos a ella por primera vez es normal que pensemos que la regla "está mal", pues parece ir "al revés" de lo que va el cambio producido por la asignación, y sin embargo esta aparente "definición hacia atrás" no sólo es correcta, sino la única forma de presentarla.
Para disipar dudas conviene examinar el significado del predicado $P' = P[x \mapsto \mathcal{A}[\![a]\!]]$, que es la precondición que decimos que necesitamos para conseguir P, sin más que ejecutar $x := a$.
¿ $P' \Rightarrow P$? Ciertamente no, pues en caso contrario resultaría innecesario hacer nada, y en particular ejecutar la asignación para conseguir la postcondición buscada P.
No lo implica, no ... ¡pero casi! En efecto, tenemos
$P'\,s ::= P(s[x \mapsto \mathcal{A}[\![a]\!]s])$ ¡que justamente nos dice que s es tal que "pasa a cumplir" P si tornamos $s\,x$ en

$\mathcal{A}[\![a]\!]s$, justo lo que hace la asignación $x := a$!
Así que la asignación $x := a$ será el <u>último</u> (y único) paso necesario para cumplir P, si nos encontramos en una situación en la que ya hemos preparado el terreno (en concreto, todo el resto del estado) y aunque el valor sx todavía no hace que se cumpla P, vemos que "cambiándolo" por $\mathcal{A}[\![a]\!]s$ (cuyo valor se "cocina" con los valores (que no cambian) de todas las demás variables, y eventualmente con el valor sx que no nos sirve (en general) para tener P, ¡pero sí podría ser necesario para <u>fabricar</u> el nuevo valor, que sí que nos servirá!) conseguimos que se satisfaga P.
Una pequeña galería de ejemplos sencillos nos mostrará la forma de uso de la regla de verificación de la asignación, y sobre todo nos ofrecerá una <u>guía metodológica</u> del "uso adecuado" de los "distintos tipos" de asignación.
Para que los ejemplos sean más ilustrativos me permitiré el uso de arrays $A[1..n]$ de enteros, que a efectos de la definición de las semánticas se corresponderán con lo que sería una "familia indexada" $\langle A_1, \ldots, A_n \rangle$ de variables (con la capacidad añadida de "acceder" a A_i con las "componentes valoradas" $A[e]$, siempre que $\mathcal{A}[\![e]\!]s \in 1..n$).

- <u>Avance recorriendo un array</u>

$$Q = \{ suma = \sum_{j=1}^{i} A[j] \wedge i \geq 0 \wedge i < n \} \; suma := suma + A[i+1] ;$$

$$i := i+1 \; \{ suma = \sum_{j=1}^{i} A[j] \wedge i \geq 0 \wedge i \leq n \} = P$$

La primera lección al examinar el aserto anterior, es que siendo perfectamente correcto, y capturando el "invariante" que deseamos, sin embargo, resultará luego (si no le añadimos nada) absolutamente inútil y no muy motivador ¡pues si sólo pretendemos que nada cambie, para qué vamos a hacer nada! Evidentemente, la solución pasa por "añadir la necesidad de progreso". En concreto, basta introducir i_0, "reforzando" la precondición con $i = i_0$, y la postcondición con $i = i_0 + 1$. La consecuencia inmediata es que ahora $Q \not\Rightarrow P$... ¡pero casi!. Como $\sum_{j=1}^{i_0+1} A[j] = \left(\sum_{j=1}^{i_0} A[j]\right) + A[i_0+1]$, de Q "sacamos" el primer sumando para suma, por lo que "basta" con incrementarlo en $A[i_0+1]$ vía $suma := suma + A[i+1]$ (¡pues $i = i_0$!); pero con ello pasaríamos a tener $suma = \sum_{j=1}^{i+1} A[j]$, y en cambio queremos $\sum_{j=1}^{i} A[j]$, ¡evidentemente la segunda asignación $i := i+1$, logra justamente el "reequilibrio" necesario! Como hemos hecho, lo "natural" es plantearse las dos asignaciones "en bloque", pero como nuestro lenguaje no las contempla, la formalización completa pasaría por la explicitación de la condición "intermedia" R, que obviamente sería

$$R = \{ suma = \sum_{j=1}^{i+1} A[j] \wedge i \geq 0 \wedge i < n \}.$$

- Una operación aritmética sencilla

$$Q = \{ x = x_0 \wedge y = y_0 \} \quad z := x + y \quad \{ x = x_0 \wedge y = y_0 \wedge z = x + y \} = P$$

De nuevo, si retiramos las constantes introducidas nos quedamos con la postcondición más liberal $P' = \{ z = x + y \}$, que naturalmente sigue valiendo aquí, pero que "no dice lo que queremos", pues se podría conseguir sin más que hacer $z := 0$, con tal de que también hagamos $x := 0$ y $y := 0$.

La diferencia con el caso anterior estriba en que la expresión $e = x + y$ sigue recogiendo información del estado s (¡justamente el que $x = x_0$, $y = y_0$, y por tanto $\mathcal{A}[\![e]\!]s = x_0 + y_0$!) pero no se utiliza "el valor" de z. En este caso no se pierde ninguna "información", pues z no se menciona en Q, pero de haberla habido, en principio se perdería (a menos que la "tuviesemos también en otra parte", por ejemplo, si en Q se nos dijera que $z = x$)

- Inicialización de variables

Por ejemplo, el bucle para calcular $suma = \sum_{j=1}^{n} A[j]$, normalmente iría precedido por $i := 0$; $suma := 0$

Con $i := 0$ conseguimos $\{i = 0\}$ a partir de ¡nada! ($\{true\}$)

"Astutamente" lo podemos "revestir" introduciendo $\sum_{j=1}^{i} A[j] = 0$, de manera que $\{i = 0\}$ $suma := 0$ $\{i = 0 \wedge suma = \sum_{j=1}^{i} A[j]\}$.

Observese que la "información útil" $suma = \sum_{j=1}^{i} A[i]$ de cara a realizar las iteraciones del bucle es en realidad "gratuita", y en contra de las apariencias ¡no liga en absoluto a suma con A!; obviamente, sólo gracias a ello podemos concluir esto ¡sin ninguna asunción, ni referencia a A!

- Un ejemplo final: reutilización de "variables auxiliares".

Simplemente considerad que hemos de calcular la suma de las componentes de una "batería" $< A^j \mid j \in 1..m >$ de arrays $A^j[1..n]$.

Ejercicio: "Pegando" el "texto de inicialización" que acabamos de ver, y el "cuerpo" que "preserva" el "invariante" $I = \{ suma = \sum_{j=1}^{i} A[j] \wedge i \geq 0 \wedge i \leq n \}$, compaginad un bucle while que "calcule" $suma = \sum_{j=1}^{n} A[j]$. Completad la verificación detallada del bucle utilizando los axiomas de verificación.

Obviamente, podríamos pensar en "limitarnos" a utilizar la solución del ejercicio "a modo de procedimiento", reiterando su ejecución para cada array de la batería.

Pero, evidentemente, la reutilización de la variable suma en cada una de las iteraciones haría que "lamentablemente" cada "reinicialización" $suma := 0$ "echara a perder" cada suma correctamente calculada, al iniciarse el cálculo de la siguiente. Para evitarlo hemos de conseguir que la asignación destructiva $suma := 0$ no suponga la perdida de su valor, que sería la suma total anterior correctamente computada.

La solución pasaría por "replicar" dicha información "en lugar seguro" (que no volvamos a manipular). Al efecto basta con añadir un vector $sumas[1..m]$, y tras "sumar" $\sum_{k=1}^{n} A^j[k]$ en la variable suma, hacemos $sumas[j] := suma$, tras lo que podemos pasar al cálculo de la suma de A^{j+1}, obviamente ejecutando $j := j+1$, hecho lo cual podemos "tranquilamente" reiniciar suma. Ejercicio: Completar la verificación de este nuevo bucle.