Auxiliary and Logical Variables

CS 536: Science of Programming, Spring 2023

2023-04-24 pp. 3, 6

A. Why

- Auxiliary variables help us reason about our programs without adding unnecessary computations.

B. Objectives

At the end of this class you should

- Recognize whether or not a set of variables is auxiliary for a program.
- Be able to add auxiliary variables to a program or remove auxiliary variables from a program, consistently.

C. Why Auxiliary Variables?

- We've used logical variables, which only appear in the correctness proof in:
  - The forward assignment rule to name the value a variable had before the assignment statement.
  - Program specifications to name the value a variable had when the program began.
- Since they only appear in proofs, we haven't been calculating the values of logical variables because it's clearly unnecessary to do so.
- Auxiliary variables are an extension of the notion of logical variables. Normally, we calculate the values of all of our program variables; with auxiliary variables, we won't.
- Auxiliary variables added to the program to enable a correctness proof but aren't relevant to the calculation of the values of variables we're actually interested in: Their actual values at runtime, however, don't affect the calculations that we're interested in. It's in that sense that auxiliary variables are unnecessary.
- To illustrate, consider forward assignment: $\{ p \wedge x = x_0 \}\ x := e\ \{ p[x_0 / x] \wedge x = e[x_0 / x] \}$.
  - Without introducing x_0, we're kind of stuck for how to describe forward assignment.
- Now consider $\{ p \}\ x_0 := x ; \{ p \wedge x = x_0 \}\ x := e\ \{ p[x_0 / x] \wedge x = e[x_0 / x] \}$
  - The assignment $x_0 := x$ sets our "logical" variable but doesn't affect the calculation of $x := e$.
  - We could calculate x_0 at runtime, but why bother if all we're interested in is x?
  - So we can argue that in some sense the assignment $x_0 := x$ doesn't really need to be executed because it doesn't affect $x := e$.
  - We've had an implicit quantifier over x_0 where the range of the quantifier is both conditions.

$$\{ \exists\, x_0 \,.\, p \wedge x = x_0 \}\; x := e \;\{ p\,[\,x_0 \,/\, x\,] \wedge x = e\,[\,x_0 \,/\, x\,] \}$$

- Here, x_0 doesn't change once we set it. Using auxiliary variables will let us change variables like x_0 as long as those changes don't affect the calculations we're interested in, so we'll still be able to avoid calculating their values.

Example 1:

- In the program below, we search through $x, f(x), f(f(x)), f(f(f(x)))$, ... for the first value that meets property $P(x)$. For termination, let's assume that in this sequence, the difference between adjacent values decreases: $|x - f(x)| > |f(x) - f(f(x))| > |f(f(x)) - f(f(f(x)))|$ ... ≥ 0.

```
x0 := x ;                          // Previous value of x
x := f(x) ;                        // New value of x
delta_x := x – x0 ;
{ inv ... } { bd | delta_x | }     // Absolute value of delta_x
while ¬ P(x) do
    ( ... computations that don't use x0 or delta_x ...)
    // Update old x, current x, and delta_x
        x0 := x ;
        x := f(x) ;
        delta_x := x – x0
od
// After the loop, we don't use delta_x or x0.
```

- If $delta\_x$ isn't used anywhere (except in the bound function), then calculating its actual value doesn't really serve any purpose. Similarly, if x_0 is used only to calculate $delta\_x$, then its value doesn't serve any useful purpose. We use x in $delta\_x := x - x_0$, but since we're not calculating $delta\_x$, we can ignore the value of x here.
- We can't just treat x_0 and $delta\_x$ as named logical constants because they change over time. We can't just write the program without them, since we need $delta\_x$ for the bound function and x_0 for $delta\_x$.
- $delta\_x$ and x_0 will be ***auxiliary variables***: They're program variables, so we can discuss their logical properties, but they're like logical variables in that we don't compute their values.

D. Auxiliary Variables

- ***Definition***: Let S be a program and let $V = Vars(S)$. A set of variables $A \subseteq V$ is an ***auxiliary set*** (for S) if:
  - All computations in S of values in $V - A$ depend only on variables in $V - A$; and
  - All boolean tests in S use only variables from $V - A$.
- The empty set is trivially auxiliary, and if S includes no boolean tests, then V is trivially auxiliary.
- ***Definition***: The ***required variables*** (with respect to A) are the ones in $V - A$.

  - The idea is that when it comes to calculating the values of variables, we're interested only in the values of required variables. Use of required variables can for us to treat other variables as required.
  - E.g., if we're interested in x, then having assignments like $x := y$ and $y := z$ force us to be interested in y and then z too.
  - We can get away with not actually calculating and storing the values of auxiliary variables because their values can't affect the values of required variables.
- ***Definition***: A variable of S is a ***primary variable*** if it is not a member of any auxiliary set of variables for S.
  - All variables that appear in tests are primary, as are the variables needed to calculate their values, directly and indirectly. (I.e, if x is primary, then $x := y$ and $y := z$ force y and z to be primary also.)
- ***Notation:*** To indicate in a program that we intend a variable to be auxiliary, we'll parenthesize it. In Example 1, we would write $(x_0) := x;$ and $(delta\_x) := x - (x_0);$ (We can omit parenthesizing them in conditions.)
- ***Definition***: An ***auxiliary labeling*** for a program tells us which program variables are auxiliary vs required.
- ***Definition***: An auxiliary labeling is ***consistent*** if
  - No auxiliary variable appears in a ***if*** or ***while*** test and
  - For every assignment statement $v := e$, if v is required then all the variables of e are also required. Contrapositively, if any variable in e is auxiliary, then v must be auxiliary.
- A case analysis shows us which usages of auxiliary variables are allowed and which are disallowed. Here, a and a' are auxiliary and r and r' are required. [2023-04-24]
- ***Allowed***: [2023-04-24]
  - $(a) := \ldots r \ldots (a') \ \ldots$ If the rhs contains an auxiliary variable, then the lhs must also be auxiliary. (The assignment forces a dependency from a' to a.) Also, if the lhs is auxiliary, then the rhs can include auxiliary and required variables.
  - $r := \ldots r' \ \ldots$ If the lhs is required, then the rhs can include required variables.
  - ***if*** / ***while*** $\ldots r \ldots$ Required variables can appear in tests.
- ***Disallowed***:
  - $r := \ldots (a) \ldots$ If the lhs is required, then the rhs cannot include auxiliary variables.
  - ***if*** / ***while*** $\ldots (a) \ldots$ Auxiliary variables cannot appear in tests.

Expanding an auxiliary labeling

- Let's call a labeling ***fully expanded*** if it is includes all the variables forced to be auxiliary. I.e., if $v := e$ is an assignment and e includes an auxiliary variable, then v is marked auxiliary.

- A fully expanded labeling is consistent if no ***if / while*** test includes a labeled variable.
- There's a simple algorithm for fully expanding a starting set of variables: If the program contains an assignment notated *y := ... (x) ...* , then mark all occurrences of *y* as *(y)*. Repeat until no such assignment exists.

Example 2:

- Let's expand the initial labeling *{ v }* for the following program. We start with

  x := y ; y := (v) + w ; ***if*** *w ≥ 0* ***then*** *x := x+1 ; w := w–1* ***fi***

- Because of *y := (v) ...* , mark *y*:

  x := (y) ; (y) := (v) + w ; ***if*** *w ≥ 0* ***then*** *x := x+1 ; w := w–1* ***fi***

- Because of *x := (y) ...*, mark *x*:

  (x) := (y) ; (y) := (v) + w ; ***if*** *w ≥ 0* ***then*** *(x) := (x) + 1 ; w := w–1* ***fi***

- No more variables need to be marked as auxiliary, and there are no disallowed uses of auxiliary variables, so *{ v , x , y }* is a consistent set of auxiliary variables.
- More generally for this program, the assignments *x := y* and *y := v + w* generate the following dependencies: *y* being auxiliary forces *x* to be auxiliary, and *v* forces *y*.
  - The assignment *x := x+1* makes *x* force *x*, which is trivial, and since *w* appears in the test, it's primary, so it doesn't matter that *y := v + w* makes *w* force *y* .)
- Altogether, there are three consistent labelings.
  - *(x) := y ; y := v + w ;* ***if*** *w ≥ 0* ***then*** *(x) := (x) + 1 ; w := w–1* ***fi*** // *{ x }* auxiliary
  - *(x) := (y) ; (y) := v + w ;* ***if*** *w ≥ 0* ***then*** *(x) := (x) + 1 ; w := w–1* ***fi*** // *{ x , y }* auxiliary
  - *(x) := (y) ; (y) := (v) + w ;* ***if*** *w ≥ 0* ***then*** *(x) := (x) + 1 ; w := w–1* ***fi*** // *{ v , x , y }* auxiliary
- In the other direction, since three of the *2⁴ – 1 = 15* nontrivial labelings are consistent, the other twelve are inconsistent:
  - Since *w* appears in the ***if*** test, it's primary, so the 8 labelings that include it are inconsistent.
  - Since *x := (y)* is inconsistent, *{ y }* and *{ y , y }* are inconsistent.
  - Since *y := (v) + w* is inconsistent, *{ v }* and *{ v , y }* are inconsistent.

Example 3:

- Consider the program *y := r ;* ***while*** *t > 1* ***do*** *y := y * t ; t := t – k* ***od***.
- The consistent labelings are
  - *(y) := r ;* ***while*** *t > 1* ***do*** *(y) := (y) * t ; t := t – k* ***od*** // *{ y }* auxiliary
  - *(y) := (r) ;* ***while*** *t > 1* ***do*** *(y) := (y) * t ; t := t – k* ***od*** // *{ r , y }* auxiliary
- For inconsistent labelings, we have
  - From ***while*** *t ...*, we know that no labeling can include *t*.
  - From *t := t – (k)*, we know that no labeling can include *k* . Since no labeling with *t* is consistent, *(t) := (t) – (k)* is also inconsistent.)

- From $y := (r)$, we know that r without y is inconsistent.

Example 4:

- Let's go back to the x_0 and $delta\_x$ program from Example 1. (To save space, I've compressed it and removed the ***inv*** and ***bd*** headers.)

  $x_0 := x;\ x := f(x);\ delta\_x := x - x_0;$
  while $\neg P(x)$ ***do*** $x_0 := x;\ x := f(x);\ delta\_x := x - x_0$ ***od***

- Since x appears in the ***while*** test, it must be primary. The assignment $delta\_x := x - x_0$ forces a dependency from x_0 to $delta\_x$, but $delta\_x$ forces no dependencies because it doesn't appear on the rhs of a any assignment.
- There are two consistent labelings. One is $\{delta\_x\}$ and $\{delta\_x, x_0\}$.

  $x_0 := x;\ x := f(x);\ (delta\_x) := x - x_0;$
  while $\neg P(x)$ ***do*** $x_0 := x;\ x := f(x);\ (delta\_x) := x - x_0$ ***od***

- The other consistent labeling is $\{delta\_x, x_0\}$.

  $(x_0) := x;\ x := f(x);\ (delta\_x) := x - (x_0);$
  while $\neg P(x)$ ***do*** $(x_0) := x;\ x := f(x);\ (delta\_x) := x - (x_0)$ ***od***

Example 5:

- As a general example of using auxiliary variables, let's consider the following disjoint parallel program. Recall for the program to be a DPP, x and y do not in appear e_2 and e_1 respectively. On the other hand, the outline does not have disjoint conditions because p_1 and p_2 depend on y and q_1 and q_2 depend on x.

  $\{p_1(x,y) \wedge q_1(x,y)\}$
  $[\ \{p_1(x,y)\}\, x := e_1\, \{p_2(x,y)\}$ // y does not appear in e_1
  $\|\ \{q_1(x,y)\}\, y := e_2\, \{q_2(x,y)\}$ // x does not appear in e_2
  $]\, \{p_2(x,y) \wedge q_2(x,y)\}$

- If we modify the outlines to have disjoint conditions, we can use disjoint parallelism to prove correctness. We'll introduce auxiliary variables x_0 and y_0, change the uses of x in thread 2 to x_0, and change the uses of y in thread 1 to y_0.

  $\{p_1(x,y) \wedge q_1(x,y)\}$
  $x_0 := x;\ y_0 := y;$
  $\{p_1(x,y_0) \wedge q_1(x_0,y)\}$
  $[\ \{p_1(x,y_0)\}\, x := e_1\, \{p_2(x,y_0)\}$
  $\|\ \{q_1(x_0,y)\}\, y := e_2\, \{q_2(x_0,y)\}$
  $]\, \{p_2(x,y_0) \wedge q_2(x_0,y)\}$

- This modified outline concludes $p_2(x,y_0) \wedge q_2(x_0,y)$. We can get the original conclusion $p_2(x,y) \wedge q_2(x,y)$ only if this modified conclusion implies the original conclusion.

Example 6:

- Let's look at a concrete instance of Example 5, starting with

  $\{ x–y = d \}\, [\, x := x+1 \;||\; y := y+1 \,]\, \{ x–y = d \}$

- Neither thread itself maintains $x–y = d$ by itself; it's only the combination that does, so we cannot just make $x–y = d$ the preconditions and postconditions of the threads.
- We can start following the pattern of Example 5:

  $\{ x–y = d \}\, x_0 := x ;\, y_0 := y\, \{ x_0–y_0 = d \wedge x_0 – y_0 = d \}$ [2023-04-24]
  $[\, \{ x_0–y_0 = d \}\, x := x+1\, \{ ??? \}$
  $\;||\; \{ x_0 – y_0 = d \}\, y := y+1\, \{ ??? \}$
  $\,]\; \{ ??? \wedge ??? \}\, \{ x–y = d \}$

- We need to figure out the missing conditions. If we use *sp* on each thread, we get
  - $\{ x_0 = x \wedge x_0 – y_0 = d \}\; x := x+1\; \{ x = x_0+1 \wedge x_0 – y_0 = d \}$
  - $\{ y_0 = y \wedge x_0 – y_0 = d \}\; y := y+1\; \{ y = y_0+1 \wedge x_0 – y_0 = d \}$
- Since $x = x_0 + 1 \wedge x_0 – y_0 = d$ and $y = y_0 + 1 \wedge x_0 – y_0 = d$ implies $x–y = (x_0 + 1) – (y_0 + 1) = x_0 – y_0 = d$, we can combine the two threads and get

  $\{ x–y = d \}\, x_0 := x ;\, y_0 := y\, \{ x_0 = x \wedge x_0 – y_0 = d \wedge y_0 = y \wedge x_0 – y_0 = d \}$
  $[\, \{ x_0 = x \wedge x_0 – y_0 = d \}\, x := x+1\, \{ x = x_0 + 1 \wedge x_0 – y_0 = d \}$
  $\;||\; \{ y_0 = y \wedge x_0 – y_0 = d \}\, y := y+1\, \{ y = y_0 + 1 \wedge x_0 – y_0 = d \}$
  $\,]\, \{ x = x_0+1 \wedge x_0 – y_0 = d \wedge y = y_0+1 \wedge x_0 – y_0 = d \}$
  $\{ x–y = d \}$ // x and y have been modified in the same way [2023-04-24]

- We use x_0 and y_0 here in the conditions of the threads but not the code, so they can be seen as logical variables:

  $\{ x_0 = x \wedge y_0 = y \wedge x–y = d \}\; \ldots \textit{program} \ldots\, \{ x–y = d \}$

E. Removing Auxiliary Variables

- We need to connect the behavior of programs with and without auxiliary variables. It turns out to be easier to discuss the behavior of removing auxiliary variables instead of adding them, so we'll do it that way.
- ***Definition***: Let S be a program and A be a set of auxiliary variables. Then $S – A$ (*"S with A removed"*) is S where where each assignment to a variable in A has been replaced by a ***skip*** statement.
- It's easy to optimize $S – A$ by replacing ***skip***; S' and S '; ***skip*** with just S' and repeating until this can't be done. If B cannot cause a runtime error, then there's also the optimization of replacing ***if*** B ***then skip else skip fi*** with just ***skip***.
  - Note it's possible to cycle through the pair of optimizations.

Example 7:

- Going back to the program and labelings of Examples 1 and 4, we had two consistent labelings: $\{ delta\_x \}$ gave

  $x_0 := x ;\ x := f(x) ;\ (delta\_x) := x - x_0 ;$ **while** $\neg P(x)$ **do** $x_0 := x ;\ x := f(x) ;\ (delta\_x) := x - x_0$ **od**

  - Removal of $\{ delta\_x \}$ yields

    $x_0 := x ;\ x := f(x) ;$ **skip;** **while** $\neg P(x)$ **do** $x_0 := x ;\ x := f(x) ;$ **skip od**

  - This optimizes to

    $x_0 := x ;\ x := f(x) ;$ **while** $\neg P(x)$ **do** $x_0 := x ;\ x := f(x)$ **od**

- The other consistent labeling was $\{ delta\_x , x_0 \}$:

  $(x_0) := x ;\ x := f(x) ;\ (delta\_x) := x - (x_0) ;$
  while $\neg P(x)$ **do** $(x_0) := x ;\ x := f(x) ;\ (delta\_x) := x - (x_0)$ **od**

  - Removal gives

    skip; $x := f(x) ;$ **skip;** **while** $\neg P(x)$ **do skip;** $x := f(x) ;$ **skip od**

  - This optimizes to

    $x := f(x) ;$ **while** $\neg P(x)$ **do** $x := f(x)$ **od**

Example 8:

- Let's go back to Example 2, where we had a program with three auxiliary labelings.
- First was the labeling $\{ x \}$. Marking, removing, and optimizing gives
  - $S \equiv (x) := y ;\ y := v + w ;$ **if** $w \geq 0$ **then** $(x) := (x) + 1 ;\ w := w - 1$ **fi**
  - $S - \{ x \} \equiv$ **skip;** $y := v + w ;$ **if** $w \geq 0$ **then skip;** $w := w - 1$ **fi**
  - $S - \{ x \}$ after optimization: $y := v + w ;$ **if** $w \geq 0$ **then** $w := w - 1$ **fi**
- For $\{ x , y \}$ we get
  - $S \equiv (x) := (y) ;\ (y) := v + w ;$ **if** $w \geq 0$ **then** $(x) := (x) + 1 ;\ w := w - 1$ **fi**
  - $S - \{ x , y \} \equiv$ **skip;** **skip;** **if** $w \geq 0$ **then skip;** $w := w - 1$ **fi** [2023-04-24]
  - $S - \{ x , y \}$ after optimization: **if** $w \geq 0$ **then** $w := w - 1$ **fi**
- For $\{ v , x , y \}$, we get a different marking from $\{ x , y \}$ but the same results after removal and optimization:
  - $S \equiv (x) := (y) ;\ (y) := (v) + w ;$ **if** $w \geq 0$ **then** $(x) := (x) + 1 ;\ w := w - 1$ **fi**
  - $S - \{ v , x , y \} \equiv$ **skip;** **skip;** **if** $w \geq 0$ **then skip;** $w := w - 1$ **fi** [2023-04-24]
  - $S - \{ v , x , y \}$ after optimization: **if** $w \geq 0$ **then** $w := w - 1$ **fi**

- ***Know this for the exam***[1]: You should be able to fully expand a labeling, be able to verify that a labeling is consistent, and be able to remove the auxiliary variables from a program. (The practice will help with these skills.)

F. Programs With Auxiliary Variables: Execution and Proof Rules

- ***Know this for the exam***: The goal is to argue that removing auxiliary variables from a program does not change how the program works on required variables.
- To phrase this, it helps to start with a lemma about a single operational semantics step ($\rightarrow$), which makes it easy to go to overall operational semantics ($\rightarrow^*$).
- ***Theorem*** (***Preservation of State Changes***):
  - ***Know this for the exam:*** Removing a program's auxiliary variables yields a program that modifies the non-auxiliary variables in exactly the same way as the original program.
- More formally, let S be a program with auxiliary and required variables A and R. Let $\sigma \cup \tau$ be a state for S where σ covers R and τ covers A. (I.e., their domains are A and R respectively) If $\langle S, \sigma \cup \tau \rangle \rightarrow^* \langle S', \sigma' \cup \tau' \rangle$, then $\langle S - A, \sigma \rangle \rightarrow^* \langle S' - A, \sigma' \rangle$.
- ***Proof***: It's sufficient to verify that single-step execution of S and $S - A$ behave the same on non-auxiliary variables. (We can iterate correctness of $\rightarrow$ to get correctness of $\rightarrow^*$.) Since S and $S - A$ differ only in $S - A$ having ***skip*** where S has assignments to auxiliary variables, this is the important case; ***if*** and ***while*** also have to be discussed; ***skip*** is trivial and omitted.

  Say S includes $v := e$, then $\langle v := e, \sigma \cup \tau \rangle \rightarrow \langle E, (\sigma \cup \tau)[v \mapsto a] \rangle$ where $a = (\sigma \cup \tau)(e)$. If v is auxiliary, the update to $\sigma \cup \tau$ can only affect σ, so we have $\langle v := e, \sigma \cup \tau \rangle \rightarrow \langle E, \sigma \cup \tau[v \mapsto a] \rangle$. The corresponding execution in $S - A$ is $\langle$ ***skip***$, \sigma \rangle \rightarrow \langle E, \sigma \rangle$, so the programs behave the same way on R.

  For ***if*** and ***while*** statements, removing A does not change the tests in ***if*** B and ***while*** B, so S jumps depend on $(\sigma \cup \tau)(B)$ and $S - A$ jumps depend on $\sigma(B)$. But B contains only required variables, so $(\sigma \cup \tau)(B) = \sigma(B)$, so the behavior in S and $S - A$ are the same on R. //
- Now that we understand the semantics of adding and removing auxiliary variables, we can formalize these as sound proof rules.
- ***Theorem (Preservation of Validity)***:
  - ***Know this for the exam***: If a program's specification doesn't involve auxiliary variables, then we can remove the auxiliary variables from the program without changing the specification.
- More formally, let $\models \{p\}\, S'\, \{q\}$ with auxiliary and required variables A and R. If no variables of A are free in p and q, then $\models \{p\}\, S' - A\, \{q\}$.
  Proof: Let $\sigma \models p$ be a state that covers R, and let τ cover A so that $\sigma \cup \tau$ is a state for S. Say $\langle S, \sigma \cup \tau \rangle \rightarrow^* \langle E, \sigma' \cup \tau' \rangle$ where σ' and τ' cover A and R. By the preservation theorem, we

[1] Things labeled "Know this for the exam" are important. Unimportant parts, like the proofs of the theorems in section F, are here because I had to write them out to convince myself they were correct.

know $\langle S - A, \sigma \rangle \to^* \langle E - A, \sigma' \rangle$. For satisfaction of q, validity of $\{ p \} S \{ q \}$ implies $\sigma' \cup \tau' \vDash q$. Since q depends only on R, this implies $\sigma' \vDash q$. So $\sigma \vDash \{ p \} S - A \{ q \}$.

Auxiliary Variable Removal

1. $\{ p \} S \{ q \}$
2. $\{ p \} S - A \{ q \}$ Auxiliary variable removal, *1*, A
   where no free variables of p or q appear in A.

