

Proof Outlines for Partial Correctness, v.2

Part 2: Partial and Minimal Proof Outlines

CS 536: Science of Programming, Spring 2023

2023-03-20, 20:00 - v.2

A. Why

- A formal proof lets us write out in detail the reasons for believing that something is valid.
- Proof outlines condense the same information as a proof.

B. Objectives

At the end of this class you should

- Know the structure of full proof outlines and formal proofs and how they are related.
- Know the difference between full, partial, and minimal proof outlines and how they are related.

C. Minimal Proof Outlines

- In a **full proof outline** of correctness, we include all the triples found in a formal proof of correctness, but we omit much of the redundant text, which makes them **much** easier to work with than formal proofs. But if you think about it, you'll realize that we can shorten the outline by omitting conditions that can be inferred to exist from the structure of the program.
- In a **minimal proof outline**, we have the minimum amount of program annotation that allows us to infer the rest of the formal proof outline. In general, we can't infer the initial precondition and initial postcondition, nor can we infer the invariants of loops, so a minimal outline will include those conditions and possibly no others.
- A **partial proof outline** is somewhere in the middle: More filled-in than a minimal outline but not completely full.
- **Example 1:** Here's a full proof outline from the previous class, with the removable parts *in blue*. (The outline comes from the previous class.)

```

{ n ≥ 0 }
k := 0; { n ≥ 0 ∧ k = 0 }           // Inferred as the sp of k := 0
s := 0; { n ≥ 0 ∧ k = 0 ∧ s = 0 }    // Inferred as the sp of s := 0
{ inv p1 ≡ 0 ≤ k ≤ n ∧ s = sum(0, k) } // The invariant remains — it can't be inferred
while k < n do
    { p1 ∧ k < n }                  // Loop rule requires inv ∧ loop test here
    { p1 [k+1/k] [s+k+1/s] }         // Inferred as the wp of s:=s+k+1
    s := s+k+1; { p1 [k+1/k] }       // Inferred as the wp of k:=k+1

```

```

    k := k + 1 {p1}           // Loop rule requires invariant at end of loop body
od
{p1 ∧ k ≥ n}                // Loop rule requires inv ∧ ¬ loop test after the loop
{s = sum(0, n)} [Mon 2023-03-20, 14:04 not inferable]

```

- Dropping the inferable parts leaves us with the minimal outline:

```

{n ≥ 0} k := 0; s := 0;
{inv p1 ≡ 0 ≤ k ≤ n ∧ s = sum(0, k)}
while k < n do
    s := s + k + 1; k := k + 1
od
{s = sum(0, n)}

```

- In a language like C or Java, the conditions become comments; something like::

```

// Assume: n ≥ 0
int k, s;           // 0 ≤ k ≤ n and s = sum(0, k)
k = s = 0;          // establish k, s
while (k < n) {
    s += k + 1;      // reset s
    ++k;             // Get closer to termination; reestablish k, s
}
// Established: s = sum(0, n)

```

- The following example shows how different total proof outlines can all have the same minimal proof outline.

- **Example 2:** The three full proof outlines below all have the same minimal proof outline, namely, $\{T\} k := 0; x := 1 \{k \geq 0 \wedge x = 2^k\}$

```

{T} {0 ≥ 0 ∧ 1 = 20} k := 0; {k ≥ 0 ∧ 1 = 2k} x := 1 {k ≥ 0 ∧ x = 2k}
{T} k := 0; {k = 0} x := 1 {k = 0 ∧ x = 1} {k ≥ 0 ∧ x = 2k}
{T} k := 0; {k = 0} {k ≥ 0 ∧ 1 = 2k} x := 1 {k ≥ 0 ∧ x = 2k}

```

- The reason multiple full proof outlines can have the same minimal outline is because different organizations of *wp* and *sp* can have the same minimal outline. There can also be differences in whether and where preconditions are strengthened or postconditions are weakened.

- **Example 3:** For the three full outlines below, the minimal outline is the same:

```

{y = x} if x < 0 then y := -x fi {y = abs(x)}.

```

- The first outline uses *wp* on the branches of the conditional:

```

{y = x}
if x < 0 then

```

```

    {y = x ∧ x < 0} {-x = abs(x)} y := -x {y = abs(x)}
else
    {y = x ∧ x ≥ 0} {y = x ∧ x ≥ 0} skip {y = abs(x)}
fi
{y = abs(x)}

```

- The second outline uses *sp* on the branches of the conditional. Using *sp* makes the predicate logic obligation for the true branch a bit more complicated.

```

{y = x}
if x < 0 then
    {y = x ∧ x < 0} y := -x {y0 = x ∧ x < 0 ∧ y = -x} {y = abs(x)}
else
    {y = x ∧ x ≥ 0} skip {y = x ∧ x ≥ 0} {y = abs(x)}
fi
{y = abs(x)}

```

- The third full outline calculates the *wp* of the conditional. There's only one predicate logic obligation, but it contains the same information as the two obligations of the previous outlines.

```

{y = x}
{(x < 0 → -x = abs(x)) ∧ (x ≥ 0 → y = abs(x))} // wp of the if-else
if x < 0 then
    {-x = abs(x)} y := -x {y = abs(x)}
else
    {y = abs(x)} skip {y = abs(x)}
fi
{y = abs(x)}
// end of example e

```

- Example 4:** The minimal proof outline for

```

{n ≥ 0} k := n; {n ≥ 0 ∧ k = n} s := n; {n ≥ 0 ∧ k = n ∧ s = n}
{inv p ≡ 0 ≤ k ≤ n ∧ s = sum(k, n)}
while k > 0 do
    {p ∧ k > 0} {p[s+k/s]} [k-1/k]}
    k := k-1; {p[s+k/s]}
    s := s+k {p}
od
{p ∧ k ≤ 0} {s = sum(0, n)}

```

is

```

{ n ≥ 0 } k := n; s := n;
{ inv p ≡ 0 ≤ k ≤ n ∧ s = sum(k, n) }
while k > 0 do
    k := k - 1; s := s + k
od
{ s = sum(0, n) }

```

D. Expanding Partial Proof Outlines

- To expand a partial proof outline into a full proof outline, basically we need to infer all the missing conditions. Postconditions are inferred from preconditions using $sp(\dots)$, and preconditions are inferred from postconditions using $wp(\dots)$. Loop invariants tell us how to annotate the loop body and postcondition, and the test for a conditional statement can become part of a precondition.
- Expanding a partial outline can lead to a number of different full outlines, but all the full outlines will be correct, and the differences between them are generally stylistic. Expansion can have different results because multiple full outlines can have the same minimal outline.
- For example, $\{p\} v := e \{q\}$ can be expanded to $\{p\} \{wp(v := e, q)\} v := e \{q\}$ or $\{p\} v := e \{sp(p, v := e)\} \{q\}$.
- The situation similar to how a full proof outline can expand to various formal proofs, all of which are correct but can be slightly different. The different full outlines here are actually different, though generally only in small ways.
- So we can't have a deterministic algorithm for expanding minimal outlines, but with that warning, here's an informal nondeterministic algorithm. Added conditions are shown *in blue*.
- **Notation:**
 - In $\{p\} \{ \dots \} S \{ \dots \}$ and $\{ \dots \} \{p\} S \{ \dots \}$, p is the first or last precondition of S respectively.
 - In $\{ \textcolor{blue}{p} \} \{ \dots \} S \{ \dots \}$ and $\{ \dots \} \{ \textcolor{blue}{p} \} S \{ \dots \}$, we add p as the new first or last precondition of S .
 - In $\{ \dots \} S \{p\} \{ \dots \}$ and $\{ \dots \} S \{ \dots \} \{p\}$, p is the first or last postcondition of S .
 - In $\{ \dots \} S \{ \textcolor{blue}{p} \} \{ \dots \}$ and $\{ \dots \} S \{ \dots \} \{ \textcolor{blue}{p} \}$, we add p as the new first or last postcondition of S .

- **The algorithm:**

Until every statement can be proved by a triple, apply one of the cases below:

A. Add a precondition:

1. Add wp to an assignment: $\{ \dots \} \{ wp(v := e, q) \} v := e \{ q \} \{ \dots \}$.
2. Add wp to a *skip*: $\{ \dots \} \{ q \} \text{ skip } \{ q \} \{ \dots \}$.
3. Add precondition to second statement of a sequence: $S_1; \{ \dots \} \{ \textcolor{blue}{p} \} S_2 \{ \dots \}$.
4. Add strongest preconditions to the branches of an *if-else*:

$\{p\} \text{ if } B \text{ then } \{ \textcolor{blue}{p} \wedge B \} \{ \dots \} S_1 \{ \dots \} \text{ else } \{ \textcolor{blue}{p} \wedge \neg B \} \{ \dots \} S_2 \{ \dots \} \text{ fi.}$

The nondeterministic version of this is

$$\{p\} \text{ if } B_1 \rightarrow \{p \wedge B_1\} \{ \dots \} S_1 \{ \dots \} \square B_2 \rightarrow \{p \wedge B_2\} \{ \dots \} S_2 \{ \dots \} \text{ fi.}$$

5. Add a precondition to an *if-else*.

$$\{ \dots \} (B \rightarrow p_1) \wedge (\neg B \rightarrow p_2) \text{ if } B \text{ then } \{p_1\} \{ \dots \} S_1 \dots \text{ else } \{p_2\} \{ \dots \} S_2 \dots \text{ fi.}$$

If p_1 and p_2 are the *wp* of S_1 and S_2 respectively, then the new addition is the *wp* of the conditional.

The nondeterministic version of this is

$$\{ \dots \} (B_1 \rightarrow p_1) \wedge (B_2 \rightarrow p_2) \text{ if } B_1 \rightarrow \{p_1\} \{ \dots \} S_1 \dots \square B_2 \rightarrow \{p_2\} \{ \dots \} S_2 \dots \text{ fi.}$$

B. Or add a postcondition:

6. Add *sp* to an assignment: $\{p\} v := e \{sp(p, v := e)\} \{ \dots \}$

7. Add *sp* to a *skip*: $\{ \dots \} \{p\} \text{ skip } \{p\} \{ \dots \}$

8. Add a postcondition to the first statement of a sequence: $\{ \dots \} S_1 \{q\}; \{ \dots \} S_2$.

We add q just after S_1 ; if we want q just before S_2 , we can add it as a new precondition of S_2 (step 3 above).

9. Add a postcondition to a conditional statement:

$$\text{if } B \text{ then } S_1 \{ \dots \} \{q_1\} \text{ else } S_2 \{ \dots \} \{q_2\} \text{ fi } \{q_1 \vee q_2\} \{ \dots \}.$$

If q_1 and q_2 are the *sp* of S_1 and S_2 , then we are adding the *sp* of the conditional.

The nondeterministic version of this is

$$\text{if } B_1 \rightarrow S_1 \{ \dots \} \{q_1\} \square B_2 \rightarrow S_2 \{ \dots \} \{q_2\} \text{ fi } \{q_1 \vee q_2\} \{ \dots \}.$$

10. Add postconditions to the branches of a conditional statement:

$$\text{if } B \text{ then } S_1 \{ \dots \} \{q_1\} \text{ else } S_2 \{ \dots \} \{q_2\} \text{ fi } \{q_1 \vee q_2\} \{ \dots \}.$$

The nondeterministic version of this is

$$\text{if } B_1 \rightarrow S_1 \{ \dots \} \{q_1\} \square B_2 \rightarrow S_2 \{ \dots \} \{q_2\} \text{ fi } \{q_1 \vee q_2\} \{ \dots \}.$$

C. Or add loop conditions:

11. Add loop body pre-and post-conditions and a loop postcondition:

$$\{inv\ p\} \text{ while } B \text{ do } \{p \wedge B\} \{ \dots \} S_1 \{ \dots \} \{p\} \text{ od } \{p \wedge \neg B\} \{ \dots \}.$$

D. Or strengthen or weaken some condition:

12. Strengthen q : $\dots \{p\} \{q\} \dots$ where $p \rightarrow q$.

13. Weaken p : $\dots \{p\} \{q\} \dots$ where $p \rightarrow q$.

// End loop

• **Example 4 reversed:** Let's expand

$\{n \geq 0\} k := n; s := n;$

$\{inv\ p \equiv 0 \leq k \leq n \wedge s = \text{sum}(k, n)\}$

while $k > 0$ **do**

```

    k := k - 1;
    s := s + k
od
{ s = sum(0, n) }

```

- First, we can apply case 6 (sp of an assignment) to $k := n$ and to $s := n$ to get

```

{ n ≥ 0 } k := n; { n ≥ 0 ∧ k = n } s := n; { n ≥ 0 ∧ k = n ∧ s = n }
{ inv p ≡ 0 ≤ k ≤ n ∧ s = sum(k, n) }
while k > 0 do
    k := k - 1;
    s := s + k
od
{ s = sum(0, n) }

```

- The next three steps are independent of the first two steps we took: First, apply case 11 to the loop:

```

{ n ≥ 0 } k := n; { n ≥ 0 ∧ k = n } s := n; { n ≥ 0 ∧ k = n ∧ s = n }
{ inv p ≡ 0 ≤ k ≤ n ∧ s = sum(k, n) }
while k > 0 do
    { p ∧ k > 0 }
    k := k - 1;
    s := s + k
    { p }
od
{ p ∧ k ≤ 0 }
{ s = sum(0, n) }

```

- Then apply case 1 (wp of an assignment) to $s := s + k$ and to $k := k - 1$:

```

{ n ≥ 0 } k := n; { n ≥ 0 ∧ k = n } s := n; { n ≥ 0 ∧ k = n ∧ s = n }
{ inv p ≡ 0 ≤ k ≤ n ∧ s = sum(k, n) }
while k > 0 do
    { p ∧ k > 0 } // ≡ 0 ≤ k ≤ n ∧ s = sum(k, n) ∧ k > 0
    { p[s+k/s][k-1/k] } // ≡ 0 ≤ k-1 ≤ n ∧ s+(k-1) = sum(k-1, n)
    k := k - 1;
    { p[s+k/s] } // ≡ 0 ≤ k ≤ n ∧ s+k = sum(k, n)
    s := s + k
    { p }
od

```

$$\{p \wedge k \leq 0\}$$

$$\{s = \text{sum}(0, n)\}$$

- And this finishes the expansion.
- Note using sp on the loop assignments works too:

$$\{n \geq 0\} k := n; \{n \geq 0 \wedge k = n\} s := n; \{n \geq 0 \wedge k = n \wedge s = n\}$$

$$\{inv\ p \equiv 0 \leq k \leq n \wedge s = \text{sum}(k, n)\}$$

$$\text{while } k > 0 \text{ do}$$

$$\quad \{p \wedge k > 0\}$$

$$\quad k := k - 1;$$

$$\quad \{p[k_0/k] \wedge k_0 > 0 \wedge k = k_0 - 1\} \quad // \quad 0 \leq k_0 \leq n \wedge s = \text{sum}(k_0, n) \wedge k_0 > 0 \wedge k = k_0 - 1$$

$$\quad s := s + k$$

$$\quad \{p[k_0/k][s_0/s] \wedge k_0 > 0 \wedge k = k_0 - 1 \wedge s = s_0 + k\} \quad // \text{ Notice: } s_0 + k \text{ not } s_0 + k_0$$

$$\quad // \quad \equiv 0 \leq k_0 \leq n \wedge s_0 = \text{sum}(k_0, n) \wedge k_0 > 0 \wedge k = k_0 - 1 \wedge s = s_0 + k$$

$$\quad // \quad \Rightarrow 0 \leq k \leq n \wedge s = \text{sum}(k, n) \equiv p$$

$$\quad \{p\}$$

$$\text{od}$$

$$\{p \wedge k \leq 0\}$$

$$\{s = \text{sum}(0, n)\}$$

Other Features of Expansion

- In Example 2, we saw that a number of full proof outlines can have the same minimal proof outline. The inverse is that a partial proof outline might expand into a number of different full proof outlines. Which one to use is pretty much a style issue.
- **Example 5:** In Example 4 reversed, we took

$$\{n \geq 0\} k := n; s := n \{p \equiv 0 \leq k \leq n \wedge s = \text{sum}(k, n)\}$$

and applied case 6 (sp) to both assignments to get

$$\{n \geq 0\} k := n; \{n \geq 0 \wedge k = n\} s := n; \{n \geq 0 \wedge k = n \wedge s = n\} \{p\}$$

- Another possibility would have been to use case 1 (wp) on both assignments; we would have gotten

$$\{n \geq 0\} \{0 \leq n \leq n \wedge n = \text{sum}(n, n)\} k := n; \{0 \leq k \leq n \wedge n = \text{sum}(k, n)\} s := n \{0 \leq k \leq n \wedge s = \text{sum}(k, n)\}$$

- Or we could have used case 6 (sp) on the first assignment and case 1 (wp) on the second:

$$\{n \geq 0\} k := n; \{n \geq 0 \wedge k = n\} \{0 \leq k \leq n \wedge n = \text{sum}(k, n)\} s := n \{p\}$$

- The three versions produce slightly different predicate logic obligations, all easy to prove.
 - sp and sp : $n \geq 0 \wedge k = n \wedge s = n \rightarrow 0 \leq k \leq n \wedge s = \text{sum}(k, n)$
 - wp and wp : $n \geq 0 \rightarrow 0 \leq n \leq n \wedge n = \text{sum}(n, n)$
 - sp and wp : $n \geq 0 \wedge k = n \rightarrow 0 \leq k \leq n \wedge n = \text{sum}(k, n)$

- Similarly, with a conditional triple $\{p\} \text{ if } B \text{ then } \{p_1\} S_1 \text{ else } \{p_2\} S_2 \text{ fi}$, we can get
 - With case 4: $\{p\} \text{ if } B \text{ then } \{p \wedge B\} \{p_1\} S_1 \text{ else } \{p \wedge \neg B\} \{p_2\} S_2 \text{ fi}$
 - Or with case 5: $\{p\} \{(B \rightarrow p_1) \wedge (\neg B \rightarrow p_2)\} \text{ if } B \text{ then } \{p_1\} S_1 \text{ else } \{p_2\} S_2 \text{ fi}$
- We get different predicate logic obligations for the two approaches, but the obligations are basically equally difficult to prove.
 - With case 4: $p \wedge B \rightarrow p_1$ and $p \wedge \neg B \rightarrow p_2$
 - With case 5: $p \rightarrow (B \rightarrow p_1) \wedge (\neg B \rightarrow p_2)$