# Proof Outlines for Partial Correctness

## Part 2: Partial and Minimal Proof Outlines

## CS 536: Science of Programming, Fall 2022

2022-10-20: p.2; 2022-10-22: p.4

### 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 provide 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 green. (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 at top of loop body
    {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***
{$p_1 \wedge k \geq n$} // Loop rule requires inv ∧ ¬ loop test after the loop
{s = sum(0, n)}

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

  {n ≥ 0} k := 0; s := 0;
  {***inv*** $p_1 \equiv 0 \leq k \leq n \wedge s = sum(0, k)$}
  ***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 [2022-10-20]
   ++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

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

  all have the same minimal proof outline, {T} k := 0; x := 1 {k ≥ 0 ∧ x = 2^k}

- 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 full outline below, the minimal outline is {y = x} ***if*** x < 0 ***then*** y := –x ***fi*** {y = abs(x)}.

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

- Note this full outline for the same code produces the same minimal outline.

  {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)}

- ***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(...), and preconditions are inferred from postconditions using wp(...). 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 expand 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.

[2022-10-22] Rewrote algorithm

- 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 green.
- ***Notation***:
  - {p} {...} S {...} and {p} {...} S {...} mean p is or is being added as the first precondition of S.
  - {...} {p} S {...} and {...} {p} S {...} mean p is or is being added as the last precondition of S.
  - {...} S {p} {...} and {...} S {p} {...} mean p is or is being added as the first postcondition of S.
  - {...} S {...} {p} and {...} S {...} {p} mean p is or is being added as the 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: {...} {wp(v := e, q)} v := e.
  2. Add wp to a skip: {...} {q} ***skip*** {q} {...}.
  3. Add precondition to second statement of a sequence: $S_1$ ; {...} {p} $S_2$.
  4. Add strongest preconditions to the branches of an ***if-else***:
     {p} ***if*** B ***then*** {p ∧ B} {...} $S_1$ {...} ***else*** {p ∧ ¬B} {...} $S_2$ {...} ***fi***.
  5. Add a precondition to an ***if-else***:
     {...} $(B \rightarrow p_1) \wedge (\neg B \rightarrow p_2)$ ***if B then*** {$p_1$} {...} $S_1$ ... ***else*** {$p_2$} {...} $S_2$ ... ***fi***.

  *B. Or add a postcondition:*

  6. Add sp to an assignment: {p} v := e {sp(p, v := e)} {...}
  7. Add sp to a ***skip***: {...} {p} ***skip*** {p} {...}
  8. Add a postcondition to the first statement of a sequence: ... $S_1$; {q} {...} $S_2$.
  9. Add a postcondition to a conditional statement
     ***if*** B ***then*** $S_1$ {...} {$q_1$} ***else*** $S_2$ {...} {$q_2$} ***fi*** {$q_1 \vee q_2$} {...}
  10. Add postconditions to the branches of a conditional statement:
      ***if*** B ***then*** $S_1$ {...} {$q_1$} ***else*** $S_2$ {...} {$q_2$} ***fi*** {$q_1 \vee q_2$} .

  *C. Or add loop conditions:*

  11. Add loop body pre-and post-conditions and a loop postcondition:
      {***inv*** p} ***while*** B ***do*** {p ∧ B} {...} $S_1$ {...} {p} ***od*** {p ∧ ¬B} {...}

  *D. Or strengthen or weaken some condition:*

  12. Strengthen q: ... {p} {q} ... where p → q.
  13. Weaken p:  ... {p} {q} ... where p → q.

  // End loop [end rewrite]

 

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

  {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)}

- 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 ≤ n ∧ s+k = sum(k, n))[k–1/k]
  ≡ 0 ≤ k–1 ≤ n ∧ s+(k-1) = sum(k-1, n)
  k := k–1;
  {p[s+k/s]} // p[s+k/s] ≡ (0 ≤ k ≤ n ∧ s = sum(k, n))[s+k/s]
  ≡ 0 ≤ k ≤ n ∧ s+k = sum(k, n)
  s := s+k {p}

***od***

{p ∧ k ≤ 0} {s = sum(0, n)}

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

{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;
{p[$k_0$/k] ∧ $k_0$ > 0 ∧ k = $k_0$–1} // 0 ≤ $k_0$ ≤ n ∧ s = sum($k_0$, n) ∧ $k_0$ > 0 ∧ k = $k_0$ – 1
s := s+k
{p[$k_0$/k][$s_0$/s] ∧ $k_0$ > 0 ∧ k = $k_0$–1 ∧ s = $s_0$+ k} [notice: plus k not plus $k_0$]
// 0 ≤ $k_0$ ≤ n ∧ $s_0$ = sum($k_0$, n) ∧ $k_0$ > 0 ∧ k = $k_0$ – 1 ∧ s = $s_0$+k
// s = sum(k, n)? s = $s_0$+($k_0$–1) and k = $k_0$–1 and $s_0$= sum($k_0$, n)
{p}
***od***
{p ∧ k ≤ 0} {s = 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 ≥ 0} k := n; s := n {p ≡ 0 ≤ k ≤ n ∧ s = sum(k, n)}

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

{n ≥ 0} k := n; {n ≥ 0 ∧ k = n} s := n; {n ≥ 0 ∧ k = n ∧ s = n} {p}

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

{n ≥ 0}
{0 ≤ n ≤ n ∧ n = sum(n, n)}
k := n;
{0 ≤ k ≤ n ∧ n = sum(k, n)}
s := n
{0 ≤ k ≤ n ∧ s = sum(k, n)}

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

{n ≥ 0} k := n; {n ≥ 0 ∧ k = n} {0 ≤ k ≤ n ∧ n = sum(k, n)} s := n {p}

- The three versions produce slightly different predicate logic obligations, but they're all about equally easy to prove.

  - sp and sp: $n \geq 0 \wedge k = n \ \wedge \ s = n \rightarrow 0 \leq k \leq n \ \wedge \ s = sum(k, n)$
  - wp and wp: $n \geq 0 \rightarrow 0 \leq n \leq n \ \wedge \ n = sum(n, n)$
  - sp and wp: $n \geq 0 \ \wedge \ k = n \rightarrow 0 \leq k \leq n \ \wedge \ n = sum(k, n)$
- Similarly, with a conditional triple $\{p\}$ ***if*** $B$ ***then*** $\{p_1\}\ S_1$ ***else*** $\{p_2\}\ S_2$ ***fi***, we can get
  - With case 4: $\{p\}$ ***if B then*** $\{p \wedge B\}\ \{p_1\}\ S_1$ ***else*** $\{p \wedge \neg B\}\ \{p_2\}\ S_2$ ***fi***
  - Or with case 5: $\{p\}\ \{(B \rightarrow p_1) \wedge (\neg B \rightarrow p_2)\}$ ***if*** $B$ ***then*** $\{p_1\}\ S_1$ ***else*** $\{p_2\}\ S_2$ ***fi***
- We get different predicate logic obligations for the two approaches:
  - With case 4: $p \wedge B \rightarrow p_1$ and $p \wedge \neg B \rightarrow p_2$
  - With case 5: $p \rightarrow (B \rightarrow p_1) \wedge (\neg B \rightarrow p_2)$
- But the work involved in proving the single second condition is about as hard as the combined work of proving the two first conditions.

 