Partial and Minimal Proof Outlines

CS 536: Science of Programming, Fall 2022

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 a previous class, with the removable parts *in green*:

```
{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)}           // Can't be inferred
while k < n { {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 bottom of loop body
}
{p₁ ∧ k ≥ n} {s = sum(0, n)}           // Loop rule requires inv ∧ ¬ loop test after the loop
```

- 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 {
        s := s+k+1; k := k+1
}
{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 and 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

- Remember we have two rules for assignment:
  - "Backward": {[e/x]q} x := e {q}
  - "Forward": {p} x := e {[x0/x]p ∧ x = [x0/x]e}
- We use backward assignment to compute wps and forward assignment to compute sps.
- Sometimes we don't have much of a choice, but sometimes we can decide whether to add a condition based on a wp or an sp, and these lead to different full proof outlines for the program, even though they may all have the same minimal proof outline.
- 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}*

Example 3

- The minimal proof outline for

```
    {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)}
    }                     {y = abs(x)}
```

  is *{y = x} if x < 0 then y := –x fi {y = abs(x)}*

- Note this is the same minimal outline for the following full outline for the same code:

```
    {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)}
    }                     {y = abs(x)}
```

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, since *{p} {wp(v := e, q)} v := e {q}* and *{p} v := e {sp(p, v := e)} {q}* have the same minimal outline, *{p} v := e {q}* can expand to either full outline.
- The situation similar to how a full proof outline can expand to various (but all correct) formal proofs, but the different proofs simply shuffle the order of the rule applications. 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 green*.)

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

// do

A. Add a precondition:

1. Prepend $wp(v := e, q)$ to $v := e\ \{q\}$.
2. Prepend q to $skip\ \{q\}$
3. Prepend some p to S_2 in $S_1;\ S_2\ \{q\}$ to get $S_1;\ \{p\}\ S_2\ \{q\}$.
4. Add preconditions to the branches of an *if-else*:
   Turn $\{p\}$ *if e then* $\{s_1\}$ *else* $\{s_2\}$ into
   $\{p\}$ *if e then* $\{\ \{p \wedge e\}\ s_1\ \}$ *else* $\{\{p \wedge \neg e\}\ s_2\ \}$
5. Add a precondition to an *if-else*:
   Prepend $(e \rightarrow p_1) \wedge (\neg e \rightarrow p_2)$ to *if e then* $\{\{p_1\}\ s_1\}$ *else* $\{\{p_2\}\ s_2\ \}$

B. Or add a postcondition:

6. Append $sp(p, v := e)$ to $\{p\}\ v := e$.
7. Append p to $\{p\}\ skip$
8. Append some q to S_1 in $\{p\}\ S_1;\ S_2$ to get $\{p\}\ S_1;\ \{q\}\ S_2$.
9. Add a postcondition to a conditional statement
   Append $q_1 \vee q_2$ to *if e then* $\{s_1\ \{q_1\}\}$ *else* $\{s_2\ \{q_2\}\}$
10. Add postconditions to the branches of a conditional statement:
    Turn *if e then* $\{S_1\}$ *else* $\{S_2\}$ $\{q_1 \vee q_2\}$ into
    if e then $\{S_1\ \{q_1\}\}$ *else* $\{S_2\ \{q_2\}\ \}\ \{q_1 \vee q_2\}$

C. Or add loop conditions:

11. Take a loop and add pre-and post-conditions to the loop body; add a postcondition for the loop:
    Turn $\{inv\ p\}$ *while e* $\{s\}$ into
    $\{inv\ p\}$ *while e* $\{\ \{p \wedge e\}\ s\ \{p\}\ \}\ \{p \wedge \neg e\}$

D. Or strengthen or weaken some condition:

12. Turn … $\{q\}$ … into … $\{p\} \Rightarrow \{q\}$ … for some predicate p where $p \Rightarrow q$.
13. Turn … $\{p\}$ … into … $\{p\} \Rightarrow \{q\}$ … for some predicate q where $p \Rightarrow q$.

// End loop

- Using the rules above, any newly added precondition gets added to the right of the old precondition; any newly added postcondition gets added to the left of the old postcondition:
  - E.g., taking the *wp* of the assignment $\{p\}\ v := e\ \{q\}$ gives us $\{p\}\ \{wp(v := e, q)\}\ v := e\ \{q\}$, not $\{wp(v := e, q)\}\ \{p\}\ v := e\ \{q\}$.

Example 4:

- Let's expand

```
                                        {$n \ge 0$}
    $j := n;$
    $s := n;$
    {inv $p \equiv 0 \le j \le n \wedge s = sum(j, n)$}
    while $j > 0$ {
        $j := j-1;$
        $s := s+j$
    }
                                        {$s = sum(0, n)$}
```

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

```
                                        {$n \ge 0$}
    $j := n;$                           {$n \ge 0 \wedge j = n$}
    $s := n;$                           {$n \ge 0 \wedge j = n \wedge s = n$}
    {inv $p \equiv 0 \le j \le n \wedge s = sum(j, n)$}
    while $j > 0$ {
        $j := j-1;$
        $s := s+j$
    }
                                        {$s = sum(0, n)$}
```

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

```
                                        {$n \ge 0$}
    $j := n;$                           {$n \ge 0 \wedge j = n$}
    $s := n;$                           {$n \ge 0 \wedge j = n \wedge s = n$}
    {inv $p \equiv 0 \le j \le n \wedge s = sum(j, n)$}
    while $j > 0$ {                     {$p \wedge j > 0$}
        $j := j-1;$
        $s := s+j$                      {$p$}
    }                                   {$p \wedge j \le 0$}
                                        {$s = sum(0, n)$}
```

- Then apply case 1 (*wp* of an assignment) to *s := s+j* and to *j := j–1*:

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

- Every line has an annotation, but we may not be done. We need to check that all of our weakenings actually apply.
  - These are referred to as "predicate logic proof obligations": they're things we need to prove in predicate logic to make sure our proof is actually right.
  - We have: $p \wedge j > 0 \Leftrightarrow 0 < j \leq n \wedge s = sum(j, n) \Leftrightarrow 0 \leq j - 1 \leq n \wedge s + j = sum(j - 1, n) \Leftrightarrow$ *[s+j/s][j–1/j]p*
  - And: $p \wedge j \leq 0 \Leftrightarrow 0 \leq j \leq n \wedge s = sum(j, n) \wedge j \leq 0 \Leftrightarrow j = 0 \wedge s = sum(j, n) \Leftrightarrow s = sum(j, n)$

```
                                        {n ≥ 0}
    j := n;                             {n ≥ 0 ∧ j = n}
    s := n;                             {n ≥ 0 ∧ j = n ∧ s = n}
    {inv p ≡ 0 ≤ j ≤ n ∧ s = sum(j, n)}
    while j > 0 do                      {p ∧ j > 0} => {[s+j/s][j–1/j]p}
        j := j–1;                       {[s+j/s]p}
        s := s+j                        {p}
    }                                   {p ∧ j ≤ 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, we took

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

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

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

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

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

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

  $\{n \geq 0\}$ $j := n$; $\{n \geq 0 \wedge j = n\}$ $\{0 \leq j \leq n \wedge n = sum(j, n)\}$ $s := n$ $\{p\}$
- The three versions produce slightly different predicate logic obligations, but they're all about equally easy to prove.
  - *sp* and *sp*: $n \geq 0 \wedge j = n \;\wedge\; s = n \rightarrow 0 \leq j \leq n \;\wedge\; s = sum(j, n)$
  - *wp* and *wp*: $n \geq 0 \rightarrow 0 \leq n \leq n \;\wedge\; n = sum(n, n)$
  - *sp* and *wp*: $n \geq 0 \;\wedge\; j = n \rightarrow 0 \leq j \leq n \;\wedge\; n = sum(j, n)$
- 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.