Proofs and Proof Outlines for Partial Correctness

Part 2: Partial Proof Outlines

CS 536: Science of Programming, Fall 2020

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 and partial proof outlines and how they are related.

C. Minimal Proof Outlines

• Full proof outlines of correctness contain all the information in full formal proofs of correctness, but omit much of the redundant text, so they're much easier to work with than formal proofs.
• But if you think about it, you'll realize that most of a full proof outline can be inferred from the structure of the program. We can shorten the outline by omitting them.
• 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.

Example 1

• Here's the full proof outline from the previous class, with the removable parts in green:

\[
\begin{align*}
\{n \geq 0\} \\
k := 0; \{n \geq 0 \land k = 0\} \quad \text{// Inferred as the sp of } k := 0 \\
s := 0; \{n \geq 0 \land k = 0 \land s = 0\} \quad \text{// Inferred as the sp of } s := 0 \\
\{\text{inv } p_1 = 0 \leq k \leq n \land s = \text{sum}(0, k)\} \quad \text{// Can't be inferred} \\
\text{while } k < n \text{ do} \\
\{p_1 \land k < n\} \quad \text{// Loop rule requires invariant \land loop test at top of loop body} \\
\{p_1[k + 1/k][s+k+1/s]\} \quad \text{// wp of } s := s+k+1 \\
s := s+k+1; \{p_1[k+1/k]\} \quad \text{// wp of } k := k+1 \\
k := k+1 \{p_1\} \quad \text{// Loop rule requires invariant at bottom of loop body} \\
o d \\
\{p_1 \land k \geq n\} \{s = \text{sum}(0, n)\} \quad \text{// Loop rule requires invariant \land \neg loop test after the loop}
\end{align*}
\]
• Dropping the inferable parts leaves us with
\[
\begin{align*}
&\{n \geq 0\} k := 0; s := 0; \\
&\text{inv } p_1 = 0 \leq k \leq n \land s = \text{sum}(0, k) \\
&\textbf{while } k < n \textbf{ do} \\
&\hspace{1em} s := s + k + 1; k := k + 1 \\
&\textbf{od} \\
&\{s = \text{sum}(0, n)\}
\end{align*}
\]

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

```c
// Assume: n \geq 0
int k, s;  // 0 \leq k \leq n and s = \text{sum}(0, k)
k = s = 0;  // establish k, s
while (k < n) {
    ++k;  // Get closer to termination
    s += k;  // and re-establish k, s
}
// Established: s = \text{sum}(0, n)
```

• Just as a full proof outline might not stand for a unique proof, a minimal proof outline might not stand for a unique full proof outline.

**Example 2**

• The three full proof outlines
\[
\begin{align*}
&T \{ 0 \geq 0 \land 1 = 2^0 \} k := 0; \{ k \geq 0 \land 1 = 2^k \} x := 1 \{ k \geq 0 \land x = 2^k \} \text{ and} \\
&T \{ k := 0; \{ k = 0 \} x := 1 \{ k \geq 0 \land x = 2^k \} \text{ and} \\
&T \{ k := 0; \{ k = 0 \} \{ k \geq 0 \land 1 = 2^k \} x := 1 \{ k \geq 0 \land x = 2^k \} \\
\end{align*}
\]
both have the same minimal proof outline: \( T \{ k := 0; x := 1 \{ k \geq 0 \land x = 2^k \} \)

**Example 3**

• The minimal proof outline for
\[
\begin{align*}
&\{ y = x \} \\
&\text{if } x < 0 \text{ then} \\
&\hspace{1em} \{ y = x \land x < 0 \} \{ -x = \text{abs}(x) \} y := -x \{ y = \text{abs}(x) \} \\
&\text{else} \\
&\hspace{1em} \{ y = x \land x \geq 0 \} \{ y = \text{abs}(x) \} \\
&\text{fi} \\
&\{ y = \text{abs}(x) \}
\end{align*}
\]
is \( \{ y = x \} \text{ if } x < 0 \text{ then } y := -x \text{ fi } \{ y = \text{abs}(x) \} \)

• Note this is the same minimal outline for the following full outline for the same code:
\[
\begin{align*}
&\{ y = x \} \\
&\{ (x < 0 \rightarrow -x = \text{abs}(x)) \land (x \geq 0 \rightarrow y = \text{abs}(x)) \} \quad \text{ // wp of the if-else}
\end{align*}
\]
if $x < 0$ then
   \{ -x = \text{abs}(x) \} \ y := -x \ \{ y = \text{abs}(x) \}
else
   \{ y = \text{abs}(x) \} \ \textbf{skip} \ \{ y = \text{abs}(x) \}
fi
\{ y = \text{abs}(x) \}

Example 4

- The minimal proof outline for
  \{ n \geq 0 \} j := n; \ (n \geq 0 \land j = n) \ s := n; \ (n \geq 0 \land j = n \land s = n) \}
  \{ \text{inv } p = 0 \leq j \leq n \land s = \text{sum}(j, n) \}
  \textbf{while} \ j > 0 \ \textbf{do}
    \{ p \land j > 0 \} \{ p[s+j/s][j-1/j] \}
    j := j-1; \ \{ p[s+j/s] \}
    s := s+j \ \{ p \}
  \textbf{od}
\{ p \land j \leq 0 \} \{ s = \text{sum}(0, n) \}

is
\{ n \geq 0 \} j := n; \ s := n;
\{ \text{inv } p = 0 \leq j \leq n \land s = \text{sum}(j, n) \}
\textbf{while} \ j > 0 \ \textbf{do}
  j := j-1; \ s := s+j
\textbf{od}
\{ s = \text{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.
- A deterministic algorithm isn't possible because a partial proof outline can stand for different (but
  equally valid) full proof outlines.
- For example, $\{ p \} v := e \ \{ q \}$ can become
  - $\{ p \} \{ wp(v := e, q) \} v := e \ \{ q \}$ or
  - $\{ p \} v := e \ \{ sp(p, v := e) \} \{ q \}$
• With that warning, here’s an informal algorithm:

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

_A. Add a precondition:_

1. Prepend \( wp(v := e, q) \) to \( v := e \{ q \} \).
2. Prepend \( q \) to \( \text{skip} \{ q \} \).
3. Prepend some \( p \) to \( S_2 \) in \( S_i; S_2 \{ q \} \) to get \( S_i; \{ p \} S_2 \{ q \} \).
4. Add preconditions to the branches of an _if-else:_
   
   Turn \( \{ p \} \text{ if } B \then S_1 \else S_2 \fi \) into \( \{ p \} \text{ if } B \land \neg \{ p \} \land \neg B \} S_2 \fi \)
5. Add a precondition to an _if-else:_
   
   Prepend \( \{ B \rightarrow p_1 \} \land \neg \{ B \rightarrow p_2 \} \) to \( \{ p_1 \} S_1 \else \{ p_2 \} S_2 \fi \)

_B. Or add a postcondition:_

6. Append \( sp(p, v := e) \) to \( \{ p \} v := e \).
7. Append \( p \) to \( \{ p \} \text{skip} \).
8. Append some \( q \) to \( S_1 \) in \( \{ p \} S_i; S_2 \) to get \( \{ p \} S_i; \{ q \} S_2 \).
9. Add a postcondition to a conditional statement
   
   Append \( q_1 \lor q_2 \) to \( \text{if } B \then S_1 \{ q_1 \} \else S_2 \{ q_2 \} \fi \)
10. Add postconditions to the branches of a conditional statement:
    
    Turn \( \text{if } B \then S_1 \else S_2 \fi \{ q \} \lor q_2 \) into \( \text{if } B \then S_1 \{ q_1 \} \else S_2 \{ q_2 \} \fi \{ q_1 \lor q_2 \}
    
    If \( q_1 = q_2 \), then \( \text{if } B \then S_1 \else S_2 \fi \{ q \} \) becomes \( \text{if } B \then S_1 \{ q \} \else S_2 \{ q \} \fi \{ q \} \).

_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 \( \{ \text{inv } p \} \text{while } B \do S_1 \od \) into \( \{ \text{inv } p \} \text{while } B \do \{ p \land B \} S_1 \{ p \} \od \{ p \land B \} \)

_D. Or strengthen or weaken some condition:_

12. Turn \( \ldots \{ q \} \ldots \) into \( \ldots \{ p \} \{ q \} \ldots \) for some predicate \( p \) where \( p \rightarrow q \).
13. Turn \( \ldots \{ p \} \ldots \) into \( \ldots \{ p \} \{ q \} \ldots \) 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 reversed:**

• Let’s expand

\[
\{ n \geq 0 \} j := n; s := n; \\
\{ \text{inv } p = 0 \leq j \leq n \land s = \text{sum}(j, n) \} \\
\text{while } j > 0 \do
\]

© James Sasaki, 2020
\[ j := j - 1; \]
\[ s := s + j \]
\[ \text{od} \]
\[ \{ s = \text{sum}(0, n) \} \]

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

\[ \{ n \geq 0 \} j := n; \{ n \geq 0 \land j = n \} s := n; \{ n \geq 0 \land j = n \land s = n \} \]
\[ \{ \text{inv } p = 0 \leq j \leq n \land s = \text{sum}(j, n) \} \]

while \( j > 0 \) do

\[ j := j - 1; \]
\[ s := s + j \]
\[ \text{od} \]
\[ \{ s = \text{sum}(0, n) \} \]

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

\[ \{ n \geq 0 \} j := n; \{ n \geq 0 \land j = n \} s := n; \{ n \geq 0 \land j = n \land s = n \} \]
\[ \{ \text{inv } p = 0 \leq j \leq n \land s = \text{sum}(j, n) \} \]

while \( j > 0 \) do

\[ \{ p \wedge j > 0 \} \]
\[ j := j - 1; \]
\[ s := s + j \{ p \} \]
\[ \text{od} \]
\[ \{ p \wedge j \leq 0 \} \{ s = \text{sum}(0, n) \} \]

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

\[ \{ n \geq 0 \} j := n; \{ n \geq 0 \land j = n \} s := n; \{ n \geq 0 \land j = n \land s = n \} \]
\[ \{ \text{inv } p = 0 \leq j \leq n \land s = \text{sum}(j, n) \} \]

while \( j > 0 \) do

\[ \{ p \wedge j > 0 \} \{ p[s + j/s][j-1/j] \} \]
\[ j := j - 1; \{ p[s + j/s] \} \]
\[ s := s + j \{ p \} \]
\[ \text{od} \]
\[ \{ p \wedge j \leq 0 \} \{ s = \text{sum}(0, n) \} \]

- And this finishes the expansion.

**Other Features of Expansion**

- When we have a sequence of assignments, we can get a number of different proof outlines. Which one to use is pretty much a style issue.

**Example 5**

- In Example 4 reversed, we took
\{(n \geq 0) \ j := n; \ s := n \ \{p = 0 \leq j \leq n \land s = sum(j, n)\}\}

and applied case 6 (sp) to both assignments to get
\{(n \geq 0) \ j := n; \ \{n \geq 0 \land j = n\} \ s := n; \ \{n \geq 0 \land j = n \land s = n\}\ \{p\}\}

• Another possibility would have been to use case 1 (wp) on both assignments; we would have gotten
\{(n \geq 0) \ j := n; \ \{n \geq 0 \land j = n\} \ s := n; \ \{n \geq 0 \land j = n \land s = n\}\ \{p\}\}

• 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 \land j = n\} \ \{0 \leq j \leq n \land s = 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 \land j = n \land s = n \rightarrow 0 \leq j \leq n \land s = sum(j, n)
  • wp and wp: \ n \geq 0 \rightarrow 0 \leq n \land n = sum(n, n)
  • sp and wp: \ n \geq 0 \land j = n \rightarrow 0 \leq j \land n = sum(j, n)

• Similarly, with a conditional triple \{p\} \ if \ B \ \{p_1\} \ S_1 \ else \ \{p_2\} \ S_2 \ fi, we can get
  • By case 4: \{p\} \ if \ B \ \{p \land B\} \{p_1\} \ S_1 \ else \ \{p \land \neg B\} \ \{p_2\} \ S_2 \ fi
  • By case 5: \{p\} \ \{(B \rightarrow p_1) \land (\neg B \rightarrow p_2)\} \ if \ B \ \{p_1\} \ S_1 \ else \ \{p_2\} \ S_2 \ fi

• We get different predicate logic obligations for the two approaches:
  • With case 4: \ p \land B \rightarrow p_1 \ and \ p \land \neg B \rightarrow p_2
  • With case 5: \ p \rightarrow (B \rightarrow p_1) \land (\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.