Using Logic on Computation Systems

Proofs of Performance

Specification of a Computational System

- By computational system, we could mean any of the following:
  - Abstract model (FSA, TM, PDA, …)
  - Single program
  - Interacting collection of programs
  - “Reactive” system, such as an operating system
  - Multi-agent systems, such as in AI and robotics.
- The language of logic can be used to specify desired properties of such systems.
- Aspects of logic such as predicates and functions can be used to specify the system itself.
- The proof aspect of logic can be used to prove that the properties hold.
Why Bother?

- Failure of a computational system can result in:
  - Loss of lives
  - Loss of money
  - Loss of time
  - General chaos

- While proving that a system works according to spec is time-consuming and requires extra effort and knowledge, it may be worth it compared to the value placed on not having the above items.

How Logic Enters (1)

- Model-checking approach (H&R, chapters 3 & 5):
  - Logic formulas are used to specify properties of a system.
  - A fixed logical interpretation ("model") is used to capture the characteristics of the system itself.
  - The properties are verified with respect to the interpretation.
  - Most workable when the interpretation has a finite domain.
  - Some infinite domains can be accommodated (e.g. Petri net models).
How Logic Enters (2)

- Proof-checking approach (H&R, chapter 4):
  - Logic formulas are used to specify properties of a system.
  - Intended interpretation is characterized by axioms.
  - New rules of inference are added to represent program construction.
  - The properties are proved in the proof system relative to the axioms.
  - Works for arbitrary interpretations, not just ones with finite domain.
  - Generally undecidable.

Perspective

- We won’t be able to cover H&R chapters 3&5. Some aspects these have been covered in a past offering of CS 156 (Parallel and Real-Time Computation).
- Chapter 5 discusses temporal logic, which is of interest in its own right. It also discusses Kripke models, which are essentially finite-state machines.
- There is the possibility of a future course dedicated to applications of logic. Let me know if you are interested.
Hoare Logic

Proofs of Programs
Essentially this is what Chapter 4 of H&R discusses.

Assumptions

- CS 60 talked about partial and total correctness with respect to an input/output specification.
- These traits were verified by insertion of assertions between statements and tests.
- A loop invariant is an assertion that goes before the test in a while loop.
Triple Notation
(C.A.R. (Tony) Hoare)

Instead of drawing

\[ \begin{array}{c}
\text{Pre-condition} \\
\text{Statement} \\
\text{Post-condition}
\end{array} \]

write

\{\text{Pre-condition}\} \quad \text{Statement} \quad \{\text{Post-condition}\}

Tony did not use the boxes, but we will, for better clarity.

Composition of Triples
Expressible as *Inference Rules*

\begin{align*}
\text{(from)} & \quad \{P\} \quad \text{Statement 1} \quad \{Q\} \\
& \quad \{Q\} \quad \text{Statement 2} \quad \{R\} \\
\text{(infer)} & \quad \{P\} \quad \text{Statement 1; Statement 2} \quad \{R\}
\end{align*}
Composition Rule
Example

\{x+y > 0\} \underline{z = x+y; \{z > 0\}}
\{z > 0\} \underline{x = z; \{x > 0\}}

\{x+y > 0\} \underline{z = x+y; x = z; \{x > 0\}}

Note:
The composition rule itself does not entail justification of the antecedents (formulas above the line). These are done as separate steps.

Other Inference Rules

- Implication Rule
- Conditional Rule
- One-armed Conditional Rule
- While Rule
- Assignment Rule
Implication Rule

\[
\begin{align*}
\{P\} & \text{Stmt 1} \{Q\} \\
\mathbf{P'} & \implies \mathbf{P} \\
\mathbf{Q} & \implies \mathbf{Q'} \\
\hline
\{P'\} & \text{Stmt 1} \{Q'\}
\end{align*}
\]

In other words, in a forward derivation:
Pre-conditions can always be strengthened; Post-conditions can always be weakened.

Implication Rule

Example

\[
\begin{align*}
\{x+y > 0\} & \underline{z = x+y} \{z > 0\} \\
(x > 0 \land y > 0) & \implies x+y > 0 \\
z > 0 & \implies z+5 > 0 \\
\hline
\{x > 0 \land y > 0\} & \underline{z = x+y} \{z+5 > 0\}
\end{align*}
\]
Conditional Rule

\[ \{Q \land P\} \text{Stmt 1} \{R\} \]
\[ \{Q \land \neg P\} \text{Stmt 2} \{R\} \]

\[ \{Q\} \text{if}(P)\text{ Stmt 1 else Stmt 2} \{R\} \]

Conditional Rule Example

\[ \{z == x \land x \geq 0\} y = x; \{z == x \land y \geq 0\} \]
\[ \{z == x \land x < 0\} y = -x; \{z == x \land y \geq 0\} \]

\[ \{z == x\} \]
\[ \text{if}(x \geq 0) y = x; \text{ else } y = -x; \]
\[ \{z == x \land y \geq 0\} \]}
One-Armed Conditional Rule

\[
\{ Q \land P \} \text{Stmt 1} \{ R \}
\]

\[
(Q \implies P) \implies R
\]

\[
\{ Q \} \text{if}(P) \text{Stmt 1} \{ R \}
\]

While Rule

\[
\{ Q \land P \} \text{Stmt 1} \{ Q \}
\]

\[
\{ Q \} \text{while}(P) \text{Stmt 1} \{ Q \land \neg P \}
\]

Q is the "loop invariant"
While Rule
Example

\[ \{ x \geq 0 \} \text{ while } ( x > 0 ) \quad x = x - 1; \quad \{ x > 0 \} \]

\[ \{ x \geq 0 \} \text{ while } ( x > 0 ) \quad x = x - 1; \quad \{ x \geq 0 \} \]

Assignment Rule

\[ \{ x \geq 0 \} \text{ while } ( x > 0 ) \quad x = x - 1; \quad \{ x \geq 0 \} \]

\[ \{ x \geq 0 \} \text{ while } ( x > 0 ) \quad x = x - 1; \quad \{ x \geq 0 \} \]

read “Q with each occurrence of x replaced with E”
Assignment Rule
Example

(no antecedent)

\[
\begin{align*}
\{x+y > 5\} & \Rightarrow x = x+y; \{x > 5\} \\
Q[x \not\in \mathfrak{E}] & \Rightarrow x = \mathfrak{E}; \quad Q
\end{align*}
\]

Typically, we use the assignment rule by starting with a desired post-condition, which then determines the pre-condition mechanically.

Weakest Precondition

- Given a block B with a post-condition Q, the **weakest-precondition** for block B with post-condition Q is the pre-condition P that is implied-by any other pre-condition:

\[
\{P\} B \{Q\}
\]

- This formula is sometimes written wp(B, Q).
- The given formula for the assignment rule in fact constructs the WP for Q.
Exercises
WP for Assignment Statements

- Examples:
  - \{??\} \ x = x+y; \{y > x\}
  - \{??\} \ y = 2^y; \{y < 5\}
  - \{??\} \ y = 2^y; \{even(y)\}

Composition Rule Example

- Consider
  \{??\} \ x = z+1; \ y = x+y; \{y > 5\}
- wp(y = x+y;, y > 5) = x+y > 5
- wp(x = z+1;, x+y > 5) = z+1+y > 5
- So ?? is
  \ z+1+y > 5
Derivation Example

Derive the following triple (where \( n! = n \times (n-1) \times \cdots \times 1 \) and \( 0! = 1 \)):

\[
\{ f = 1 \land k = n \land n \geq 0 \} \text{ while } (k > 0) \{ f = k \times f; \ k = k-1; \} \{ f = n! \}
\]

Approach:

Work backward.
Use the while rule, the composition rule, and the assignment rule (twice),
and the implication rule.

While rule: What is the invariant \( Q \)?

The exit condition is of the form \( (Q \land P) \) where \( P \) is \( k > 0 \)
suggesting that invariant \( Q \) might be: \( f = n! / k! \land k \geq 0 \)
for then \( (Q \land P) \) is \( f = n! / k! \land k \geq 0 \land k > 0 \)
which is equivalent to \( k = 0 \land f = n! \) which implies \( f = n! \).

It is necessary to provide some formulas for dealing with factorial (!).

Derivation Example

The invariant is justified by the while rule, provided that we can derive:

\[
\{ f = n! / k! \land k \geq 0 \land k > 0 \} \text{ while } (k > 0) \{ f = k \times f; \ k = k-1; \} \{ f = n! / k! \land k \geq 0 \}
\]

But \( k > 0 \land k \geq 0 \), so it is sufficient (by the implication rule) to derive the logically equivalent

\[
\{ f = n! / k! \land k > 0 \} \{ f = k \times f; \ k = k-1; \} \{ f = n! / k! \land k \geq 0 \} \quad (*)
\]

Work backward from the post-condition using the assignment rule:

\[
\{ f = n! / (k-1)! \land (k-1) \geq 0 \} \{ k = k-1; \} \{ f = n! / k! \land k \geq 0 \}
\]
Derivation Example

Use the assignment rule again:

\[
\{ k \ast \text{f} == n!/k! \} \quad \{ f == n!/(k-1)! \} \quad \{ k > 0 \} \quad \{ f == n!/(k-1)! \} \quad \{ (k-1) > 0 \}
\]

The left-hand formula simplifies, to get:

\[
\{ \text{f} == n!/k! \} \quad \{ \text{f} == n!/(k-1)! \} \quad \{ \text{k} > 0 \}
\]

since \( n!/k! == n*(n-1)*(n-2)\ldots(k+1)*k \)

and, dividing by this by \( k \) gives \( n*(n-1)*(n-2)\ldots(k+1) \)

which is \( n! / k! \).

So now we have shown that (*) is derivable by the composition rule.

It is necessary to provide some formulas for dealing with factorial (!).

Derivation Example

All that is left to do is use the implication rule, with

\[
f == 1 \quad \text{f} == n! \quad \text{k} == n \quad \text{k} \geq 0 \quad \{ f == n! / k! \} \quad \{ k \geq 0 \}
\]

overall pre-condition invariant

This is plausible, since from the left-hand side \( k == n \), so \( k == n! \), and thus \( n! / k! == 1 \).
Derivation Exercise

Derive the following triple:

\{ x \equiv x_0 \land y \equiv y_0 \land x > 0 \land y > 0 \}

\[
\text{while( } x \neq y \text{ )}
\]
\[
\begin{align*}
\text{if( } x > y \text{ )} \\
\quad &x = x - y; \\
\text{else} \\
\quad &y = y - x;
\end{align*}
\]

\{ x \equiv \gcd(x_0, y_0) \}

where \( \gcd \) is the greatest-common-divisor function,

introducing and justifying any formulas you need for \( \gcd \).

Verifying Termination

- “Partial correctness” means that the program is correct, provided that it terminates.

- “Total correctness” is partial correctness and termination.

- Termination is often verified separately.
Verifying Termination

- The reason that termination is verified separately is that it requires coming up with a different sort of expression than an invariant.

- Such an expression is a “variant”. It describes a program’s inexorable movement toward a stopping point.

Variants

- Clearly the only cause for a (non-recursive) program’s non-termination could lie in while-loops.

- A variant is some expression $E$ such that:
  - $E \geq 0$ is invariant, and
  - The value of $E$ decreases at every iteration.

- If a loop has a variant, then the loop must terminate.
Variant Example

\{x \equiv x_0 \land y \equiv y_0 \land x_0 \geq 0\}

\textbf{while}( x > 0 )
\{ \\
y = y + k; \\
x = x - 1;
\}
\{y \equiv y_0 + k^*x_0\}

Here a \textbf{variant} for the loop would be \( x \), since:
\( x \geq 0 \) is invariant, and
\( x \) decreases on each iteration.

Variant as a Triple

A sufficient condition for \( \mathcal{E} \) to be a variant of
\[\textbf{while}(P) \text{ Stmt;} \]
is that we be able to derive a triple:

\( \{\mathcal{E}_0 \equiv \mathcal{E} \land \mathcal{E} > 0 \land P\} \text{ Stmt } \{\mathcal{E}_0 > \mathcal{E} \land \mathcal{E} \geq 0\} \)

where \( \mathcal{E}_0 \) is a free variable.
Consider the previous while program:

```c
while( x > 0 )
{
    y = y + k;
    x = x-1;
}
```

is the triple to be derived.

Working backward from the post-condition, we need to derive:

\[ \{ x_0 = x \land x > 0 \land x > 0 \} \implies y = y + k; \ x = x-1; \ {x_0 > x \land x \geq 0} \]

which follows from the implication rule if we can derive:

\[ \{ x_0 = x \land x > 0 \land x > 0 \} \implies \{ x_0 > x \land x \geq 0 \} \]

which follows directly (assuming x integer).
Exercise

Derive a variant for the gcd program introduced earlier:

\{x = x_0 \land y = y_0 \land x > 0 \land y > 0\}

\textbf{while}( x \neq y )
\textbf{if}( x > y )
\quad x = x - y;
\textbf{else}
\quad y = y - x;
\textbf{end while}

\{x = \text{gcd}(x_0, y_0)\}

Exercise

Can a variant be derived for the similar triple:

\{x = x_0 \land y = y_0 \land x \geq 0 \land y \geq 0\}

\textbf{while}( x \neq y )
\textbf{if}( x > y )
\quad x = x - y;
\textbf{else}
\quad y = y - x;
\textbf{end while}

\{x = \text{gcd}(x_0, y_0)\}
WP Calculus

- wp obeys some fairly obvious rules:
  - \( wp(x = \mathcal{E};, \ Q) = Q [\mathcal{E} / x ] \) as already stated
  - \( wp(B_1; B_2, Q) = \)
  - \( wp(if(P) \ B_1; \ else \ B_2, Q) = \)

- wp for a loop is harder, because it generally requires an infinite formula (unwind the loop as an infinite nest of conditions).

Example: WP for a Test

- \{??\}
  \[
  \text{if( } x > y \text{ ) } x = x-y; \ else \ y = y-x;
  \{gcd(x, y) == z}\]

- wp is \( wp’s \) of the assignment statements
  - \( (x > y) \quad gcd(x-y, y) == z \)
  - \( (x > y) \quad gcd(x, y-x) == z \)
Example 2: WP for a Test

- `{??}
  if( x > y ) z = x; else z = y;
  {z == max(x, y)}
- wp is wp’s of the assignment statements
  (x > y) \implies max(x, y) == x
  \iff (x > y) \implies max(x, y) == y
  which simplifies to true.

When the else part is missing

- If the else part is missing, then T is effectively a “no-op”, “skip”, or trivial assignment x = x;
- Since wp(x = x; Q) = Q
- the wp for
  if(P) S
  is then
  P \implies wp(S, Q)
  \wedge P \implies Q
Example: WP for a Test without else

- `{??}
  if( x > y ) y = x;
  \{y = \max(x, y)\}

- wp is \( wp \) of the assignment statement

  \[
  \begin{align*}
  (x > y) & \implies x = \max(x, x) \\
  \| (x > y) & \implies y = \max(x, y)
  \end{align*}
  \]

- which simplifies to true.

Alternate WP for a Test

- \( wp(if(P) S \text{ else } T, Q) = \)

  \[
  (P \land wp(S, Q)) \\
  (\lnot P \land wp(T, \lnot Q))
  \]

- To see that this is equivalent to the previous version, let \( wp(S, Q) \) be A and \( wp(T, Q) \) be B. Then we are asking whether

  \[
  (P \land A) \land (\lnot P \land B)
  \]

  is equivalent to

  \[
  (P \lor A) \land (\lnot P \lor B)
  \]
Alternate WP for a Test

- \((P \land A) \equiv (\Box P \land B) \equiv (P \implies A) \land (\Box P \implies B)\)
- For \(P = true\), this becomes \(A =? A\).
- For \(P = false\), this becomes \(B =? B\)
- Therefore the two forms are equivalent.

Sorting Example using WP (1)

- Suppose that \(S(x, y)\) stands for
  \[\text{if}(x > y) \{x, y = y, x;\}\]
  where by the assignment statement we mean \textit{parallel assignment}: the RHS’s are evaluated then the values are \textit{simultaneously} assigned to the LHS variables.
Sorting Example (2)

- $S(x, y)$ stands for
  \[
  \text{if}(x > y) \{x, y = y, x;\}
  \]

- Using the WP for $if$, we have for any predicate $P$
  \[
  \text{wp}(\text{if }(x > y) \{x, y = y, x;\}, P) = \\
  [(x > y) \land P[y, x / x, y]] \land [(x > y) \lor P]
  \]
  “Parallel” substitution

Sorting Example (3)

- Suppose that we wish to show:
  \[
  \{\text{true}\}
  \]
  \[
  S(x, y); S(y, z); S(x, y)
  \]
  \[
  \{x \leq y \land y \leq z\}
  \]
  i.e. the sequence shown sorts three numbers.
Sorting Example (4)

- Working backward from the last statement $S(x, y)$:
  \[
  wp(S(x, y), \{x \leq y \land y < z\}) =
  \]
- $wp(if(x>y) \{x, y = y, x;\}, \{x \leq y \land y < z\}) =
  \]
- \[
  [\neg(x > y) \lor (y < x \land x \leq z)]
  \land \neg(x > y) \lor (x < y \land y < z)
  \]
- which simplifies (using reasoning about $\leq$ and $>$) to
  \[
  [(x > y) \lor (x \leq z)] \land [(x < y) \lor (y < z)]
  \]
- which simplifies to
  \[
  (x < z) \lor (y < z)
  \]

Sorting Example (5)

- Working backward from the middle statement $S(y, z)$:
  \[
  wp(S(y, z), (x < z) \lor (y < z)) =
  \]
- \[
  [\neg(y > z) \lor [(x < y) \land (z < y)]]
  \land [\neg(y > z) \lor [(x < z) \land (y < z)]]
  \]
- which is equivalent to
  \[
  [(y > z) \lor (x < y)]
  \land [(y < z) \lor (x < z)]
  \]
## Sorting Example (6)

- Working backward from the first statement \( S(x, y) \):

\[
wp(S(x, y), [(y > z) \land (x \leq y)] \lor [(y \leq z) \land (x \leq y)]) =
\]

\[
[(x > y) \land [(x > z) \land (y \leq x)] \lor [(x \leq z) \land (y \leq z)]]
\]

\[
\lor [(x > y) \land [(y > z) \land (x \leq y)] \lor [(y \leq z) \land (x \leq z)]]
\]

- which simplifies to

\[
[(x > y) \land [(x \leq z) \land (y \leq z)]]
\]

\[
\lor [(x \leq y) \land [(y \leq z) \land (x \leq z)]]
\]

- which simplifies to

true

## Sorting Example (7)

- That derivation was fairly complex.

- However, you did not see my first version.

- The simplifications were motivated by intuition: the observation that the main thing that makes this work is the pre-condition for the final statement \( S(x, y) \):

\[(x \leq z) \lor (y \leq z)\]
Forward Reasoning

Sorting Example by Forward Reasoning (1)

- We want
  
  \{\text{true}\}
  
  \(S(x, y); S(y, z); S(x, y)\)
  
  \(\{x \leq y \land y \leq z\}\)

- This time try forward reasoning.
Sorting Example by Forward Reasoning (2)

- First
  
  \{true\}

  S(x, y);

  \{x \leq y\}

- We can verify this claim by backward reasoning on one step.

- We could also use “primes” for after and before.

Sorting Example by Forward Reasoning (3)

- Second
  
  \{x \leq y\}

  S(y, z);

  \{??\}?}
Sorting Example by Forward Reasoning (4)

- \{x \leq y\}
  
  \text{S}(y, z);
  
  \{x \leq z \land y \leq z\}

- This can also be verified by backward reasoning on this one step, or using primes.

Sorting Example by Forward Reasoning (5)

- Third, we need
  
  \{x \leq z \land y \leq z\}
  
  \text{S}(x, y);
  
  \{x \leq y \land y \leq z\}

  which we already derived in the first step of the original wp discussion.
Sorting Example by Forward Reasoning (6)

- Alternate: Using “prime” analysis on the previous statement:
- We need to show:
  \[(x < z \land y < z) \lor (x > y \land x' = y \land y' = x)\]
  \[\land [x' \leq y' \land y' \leq z']\]
  and
  \[(x < z \land y < z) \lor (x > y)\] \[\lor [x \leq y \land y \leq z]\]
  (If no assignment is done, we don’t need the primes.)

Forward vs. WP

- In the sorting example, forward reasoning seemed to be simpler than WP.
- This won’t always be the case.
- The reason that WP is sometimes simpler is that it only targets proving the desired post-condition, rather than everything that can be derived from the pre-condition.