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).
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.
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.
Instead of drawing

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

Tony did not use the boxes, but we will, for better clarity.
Composition of Triples
Expressible as *Inference Rules*

(from)

\{P\} Statement 1 \{Q\}
\{Q\} Statement 2 \{R\}

(infer)

\{P\} Statement 1; Statement 2 \{R\}
Composition Rule

Example

\[ \{x+y > 0\} \quad z = x+y; \quad \{z > 0\} \]

\[ \{z > 0\} \quad x = z; \quad \{x > 0\} \]

---

\[ \{x+y > 0\} \quad z = x+y; \quad x = z; \quad \{x > 0\} \]

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

\{P\}\text{Stmt 1}\{Q\}

\begin{align*}
P' & \implies P \\
Q & \implies Q'
\end{align*}

\{P'\}\text{Stmt 1}\{Q'\}

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

Example

\[\{x+y > 0\} z = x+y; \{z > 0\}\]

\[(x > 0 \land y > 0) \land x+y > 0\]

\[z > 0 \land z+5 > 0\]

\[\{x > 0 \land y > 0\} z = x+y; \{z+5 > 0\}\]
Conditional Rule

\{Q \sqcap P\} \textbf{Stmt 1} \{R\}

\{Q \sqcap \neg P\} \textbf{Stmt 2} \{R\}

\{Q\} \textbf{if}(P) \textbf{Stmt 1} \textbf{else} \textbf{Stmt 2} \{R\}
Conditional Rule
Example

\( \{z == x \land x \geq 0\} \text{y} = x; \{z == x \land y \geq 0\} \)

\( \{z == x \land x < 0\} \text{y} = -x; \{z == x \land y \geq 0\} \)

\( \{z == x\} \)

\( \text{if}(x \geq 0) \ y = x; \text{else} \ y = -x; \)

\( \{z == x \land y \geq 0\} \)
One-Armed Conditional Rule

\{Q \parallel P\} \text{Stmt 1} \{R\}

(Q \parallel P) \parallel R

\{Q\} \text{if}(P) \text{Stmt 1} \{R\}
While Rule

\{Q \land P\} \textbf{Stmt 1} \{Q\}

\{Q\} \textbf{while( P ) Stmt 1} \{Q \land \neg P\}

Q is the “loop invariant”
While Rule
Example

\[
\{x \geq 0 \land x > 0\} \quad x = x - 1; \quad \{x \geq 0\}
\]

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

\[ \{Q[E/x]\} \ x = E; \ \{Q\} \]

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

(no antecedent)

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)}`
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 \( n! = n*(n-1)\cdots*1 \) and \( 0! = 1 \)):

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

**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 \((Q \land \neg P)\) where \( P \) is \( k > 0 \)

suggesting that invariant \( Q \) might be: \( f == n! / k! \land k \geq 0 \)

for then \((Q \land \neg P)\) is \( f == n! / k! \land k > 0 \land k \geq 0 \)

which is equivalent to \( k == 0 \land f == n! \) which implies \( f == n! \).
Derivation Example

The invariant is justified by the while rule, \textit{provided} that we can derive:

\[
\{ f == n! / k! \quad \Box \quad k > 0 \quad \Box \quad k > 0 \} \quad f = k * f; \quad k = k-1; \quad \{ f == n! / k! \quad \Box \quad k > 0 \}
\]

\[(Q \quad \Box \quad P) \{ \text{Stmt 1} \} \quad Q\]

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

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

Work backward from the post-condition using the \textit{assignment rule}:

\[
\{ f == n!/(k-1)! \quad \Box \quad (k-1) \geq 0 \} \quad k = k-1; \quad \{ f == n! / k! \quad \Box \quad k \geq 0 \}
\]
Use the assignment rule again:

\[ \{ k \cdot f = n!/(k-1)! \quad \square \quad (k-1) \geq 0 \} \quad f = k \cdot f; \quad \{ f = n!/(k-1)! \quad \square \quad (k-1) \geq 0 \} \]

The left-hand formula simplifies, to get:

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

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

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

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

So now we have shown that (*) is derivable by the composition rule.
All that is left to do is use the implication rule, with

\[
\begin{align*}
\text{overall pre-condition} & : \ f = 1 \land k = n \land n \geq 0 \\
\text{invariant} & : \ f = n! / k! \land k \geq 0
\end{align*}
\]

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

while (x != y)
    if (x > y)
        x = x - y;
    else
        y = y - x;

\{ x = \text{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 == x_0 \quad y == y_0 \quad x_0 \geq 0 \} 

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

\{ y == y_0 + k*x_0 \}

Here a variant for the loop would be \( x \), since:
- \( x \geq 0 \) is invariant, and
- \( x \) decreases on each iteration.
A sufficient condition for $\mathcal{E}$ to be a variant of

\texttt{while}(P) Stmt;

is that we be able to derive a triple:

\[
\{ \mathcal{E}_0 == \mathcal{E} \quad \mathbb{E} > 0 \quad \mathbb{P}\} \quad \text{Stmt} \quad \{ \mathcal{E}_0 > \mathcal{E} \quad \mathbb{E} \geq 0 \}
\]

where $\mathcal{E}_0$ is a free variable.
Variant as a Triple: Example

\{E_0 == E \land E > 0 \land P\} \text{ Stmt } \{E_0 > E \land E \geq 0\}

Consider the previous while program:

\begin{verbatim}
while( x > 0 )
{
  y = y + k;
  x = x-1;
}
\end{verbatim}

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

is the triple to be derived.
Variant as a Triple: Example

\{x_0 \equiv x \geq x > 0 \land x > 0\} y = y + k; \quad x = x-1; \quad \{x_0 > x \land x \geq 0\}

is the triple to be derived.

Working backward from the post-condition, we need to derive:
\{x_0 \equiv x \geq x > 0 \land x > 0\} y = y + k; \quad \{x_0 > x-1 \land x-1 > 0\}

which follows from the implication rule if we can derive:
\{x_0 \equiv x \geq x > 0 \land x > 0\} \implies \{x_0 > x-1 \land x-1 > 0\}

which follows directly (assuming x integer).
Exercise

- Derive a variant for the gcd program introduced earlier:

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

```
while( x != y )
    if( x > y )
        x = x - y;
    else
        y = y - x;

\{ x == gcd(x_0, y_0) \}
```
Exercise

Can a variant be derived for the similar triple:

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

while( x != y )
    if( x > y )
        x = x - y;
    else
        y = y - x;

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

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

- 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

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

- \text{wp is} \quad \text{wp's of the assignment statements}

  \[
  \begin{align*}
  &\quad (x > y) \quad \quad \gcd(x-y, y) == z \\
  &\quad (x > y) \quad \quad \gcd(x, y-x) == z
  \end{align*}
  \]
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

  \[(x > y) \Downarrow \max(x, y) == x \Downarrow\]

- \[(x > y) \Downarrow \max(x, y) == y\]

  which simplifies to \textit{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
    - $P \not\in wp(S, Q)$
    - $\not\in P \not\in Q$
Example: WP for a Test without else

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

- wp is
  \[
  \wp \text{ of the assignment statement}
  \]
  \[
  (x > y) \land\land x = \max(x, x)
  \]
  \[
  \land\land (x > y) \land\land y = \max(x, y)
  \]

- which simplifies to \textit{true}. 
Alternate WP for a Test

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

\[ \begin{align*}
(P \land \text{wp}(S, Q)) \\
(\Box P \land \text{wp}(T, !Q))
\end{align*} \]

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

\[ (P \land A) \quad (\Box P \land B) \]

is equivalent to

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

\[(P \land A) \land (\Box P \land B) =? (P \Box A) \land (\Box P \Box B)\]

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