**Using Logic on Computation Systems**

**Proofs of Performance**

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

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

write

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

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

Composition of Triples
Expressible as Inference Rules

(from)

\begin{align*}
\{ P \} & \text{Statement 1} \{ Q \} \\
\{ Q \} & \text{Statement 2} \{ R \}
\end{align*}

(infer)

\{ P \} \text{Statement 1; Statement 2} \{ R \}

Composition Rule
Example

\[
\begin{align*}
(x+y > 0) & \quad z = x+y \quad (z > 0) \\
(z > 0) & \quad x = z \quad (x > 0)
\end{align*}
\]

\[
(x+y > 0) \quad x = x+y; \quad x = z; (x > 0)
\]

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

\[(P) \implies \text{Stmt 1} \implies (Q)\]
\[P' \implies P\]
\[Q \implies Q'\]

\[\vdash (P') \implies \text{Stmt 1} \implies (Q)\]

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

Implication Rule Example

\[(x + y > 0) \implies z = x + y \implies (z > 0)\]
\[(x > 0 \implies y > 0) \implies x + y > 0\]
\[\vdash z > 0 \implies z + 5 > 0\]

Conditional Rule

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

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

Conditional Rule Example

\[(z = x \implies x \geq 0) \implies y = x \implies (z = x \implies y \geq 0)\]
\[(z = x \implies x < 0) \implies y = -x \implies (z = x \implies y \geq 0)\]

\[\vdash (z = x)\]
\[\text{if}(x > 0) \implies y = x \text{ else } y = -x;\]
\[\{z = x \implies y \geq 0\}\]

One-Armed Conditional Rule

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

While Rule

\[(Q \implies P) \implies \text{Stmt 1} \implies (Q)\]
\[\vdash (Q) \implies \text{while}(P) \implies \text{Stmt 1} \implies (Q \implies P)\]

\[Q \text{ is the "loop invariant"}\]
While Rule Example

\[
\begin{array}{c}
\{x \geq 0 \} [x \times 0] [x = x - 1] [x \geq 0] \\
\{x \geq 0\} \text{ while} (x > 0) [x = x - 1] [x \geq 0; x \leq 0] \\
\{x \geq 0\}
\end{array}
\]

Assignment Rule

\[
\{Q \forall x \} x = \exists; (Q)
\]

read "Q with each occurrence of x replaced with E"

Assignment Rule Example

(no antecedent)

\[
\begin{array}{c}
\{x+y > 5\} x = x+y; (x \times 5) \\
\{x \geq 0\} \text{ Q}[x \forall x] x = \exists; Q
\end{array}
\]

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*(n-1)*...*1 and 0! == 1):

\{ f == 1 \} \land \{ k == n \} \land \{ n > 0 \} \implies \{ f == k! \} \land \{ k > 0 \} \land \{ 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!

for then (Q \land P) is f == n! / k! \land k > 0 which is equivalent to k == 0 \land f == n! which implies f == n!.

Approach:

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

\{ f == n! / (k-1)! \} \land (k-1) > 0 \land k = k-1;

\{ f == n! / k! \} \land k > 0

The left-hand formula simplifies, to get:

\{ f == n! / k! \} \land k > 0

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

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

which is n! / k!.

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

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 == 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 \( I \) such that:
  - \( I \geq 0 \) is invariant, and
  - The value of \( I \) decreases at every iteration.
- If a loop has a variant, then the loop must terminate.

Variant Example

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

\[
\text{while}(x > 0)
\]

\[
\{
    \quad y = y + k;
    \quad x = x - 1;
\}
\]

\[
\{ y = y_0 + k \cdot x_0 \}
\]

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

Variant as a Triple

A sufficient condition for \( I \) to be a variant of

\[
\text{while}(P) \quad \text{Stmt}
\]

is that we be able to derive a triple:

\[
\{ I_0 = I \} \quad \{ I > 0 \} \quad \text{Stmt} \quad \{ I_0 > I \} \quad \{ I \geq 0 \}
\]

where \( I_0 \) is a free variable.

Variant as a Triple: Example

\[
\{ I_0 = I \} \quad \{ I > 0 \} \quad \text{Stmt} \quad \{ I_0 > I \} \quad \{ I \geq 0 \}
\]

Consider the previous while program:

\[
\text{while}(x > 0)
\]

\[
\{
    \quad y = y + k;
    \quad x = x - 1;
\}
\]

\[
\quad \{ y = y_0 + k \cdot x_0 \}
\]

is the triple to be derived.

Variant as a Triple: Example

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

is the triple to be derived.

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

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

which follows from the implication rule if we can derive:

\[
\{ x_0 = x \} \quad \{ x > 0 \} \quad \{ x_0 > x - 1 \} \quad \{ x \geq 0 \}
\]

which follows directly (assuming \( x \) integer).
Exercise

Derive a variant for the gcd program introduced earlier:

\[
\begin{align*}
&\text{while}( x \neq y ) \\
&\text{if}( x > y ) x = x - y; \\
&\text{else } y = y - x; \\
&\{ x = \text{gcd}(x_0, y_0) \}
\end{align*}
\]

Exercise

Can a variant be derived for the similar triple:

\[
\begin{align*}
&\text{while}( x \neq y ) \\
&\text{if}( x > y ) x = x - y; \\
&\text{else } y = y - x; \\
&\{ x = \text{gcd}(x_0, y_0) \}
\end{align*}
\]

WP Calculus

wp obeys some fairly obvious rules:

- \( wp(x = E, Q) = Q[x/E] \) as already stated
- \( wp(B_1; B_2, Q) = wp(B_1; B_2, Q) \)

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

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

wp is

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

Example 2: WP for a Test

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

wp is

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

which simplifies to \( \text{true} \).

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 \( \text{if}(P) S \)
  is then
  \[
  P \implies wp(S, Q)
  \]
  \[
  P \implies Q
  \]
Example: WP for a Test without else

- if\((x > y)\) \(y = x;\)
- \(\{y = \text{max}(x, y)\}\)
- \(wp\) is \(wp\) of the assignment statement
- \((x > y) \implies x = \text{max}(x, x)\)
- \(\square(x > y) \implies y = \text{max}(x, y)\)
- which simplifies to \text{true}.

Alternate WP for a Test

- \(wp\) of \(\text{if}(P)\) \(S\) else \(T, Q) =\\
- \((P \land wp(S, Q)) /\wp(P \land wp(T, Q))\)
- 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\n- \((P \land A) /\wp(P \land B)\)
- is equivalent to\n- \((P \implies A) /\wp(P \implies B)\)

Alternate WP for a Test

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