Predicate Logic for Specifying Programs and Proving their Correctness

Program Specification
- At one level of abstraction, a program is a relation between input and output values.
- It is common to express this relation as a pair of assertions about the state of a program:
  - The input assertion specifies what must be true at the start in order for the program to be applicable.
  - The output assertion specifies what will be true at the end.
- Such assertions are predicates on program state.

Pre- and Post- Conditions
- The input assertion is also called a pre-condition and the output assertion a post-condition.
- We normally work under the assumption that the input assertion is true.
- If the input assertion is not true, then "all bets are off";

Uses of Specification
- Specification can be used before coding, to prove that an already-constructed program is correct.
- Specification can be used before coding, as a specification to which code can be written.

Specification Example
- A program that finds the index of a maximal value in an array $a$ of $n > 0$ elements.
- An array max-finding program:
  - Input assertion: $n > 0$
  - Output assertion:
    \[ (\forall i \in \mathbb{N} \leq n) \ (a[i] \leq a[m]) \]
    ("Index $m$ is index of a maximal value in $a".)
  - Is this adequate as a specification?

The Need for "Anchors"
- Input assertion: $n > 0$ and $a \geq a_0$
- Output assertion: Index $m$ is index of a maximal value in $a$ and $a \geq a_0$
- Without the anchor, the assertion could be satisfied without the program actually computing the maximum; it could simply set all elements of $a$ to $0$ and $m$ to $0$, and the assertion would be satisfied: $a \equiv a_0$ prevents the array from being modified overall.
Specification Examples

- **A greatest-common-divisor program:**
  - **Input assertion:** \( x = x_0, y = y_0, x_0 > 0, y_0 > 0 \)
  - **Output assertion:** \( x \) is the greatest common divisor of \( x_0 \) and \( y_0 \)(notated \( x = \text{gcd}(x_0, y_0) \)).
  
  Implicit here is that all quantities are integers.

- **Values** \( x_0 \) and \( y_0 \) are not program variables, but serve to anchor the original values for later reference.

Examples

- **A searching program:**
  - **Input assertion:** \( a == a_0 \)
  - **Output assertion:** Index \( m \) is such that
    - \( a == a_0 \) and
    - \( a[m] == v \) if \( v \) occurs in array \( a \), and
    - \( m == -1 \) if \( v \) does not occur in array \( a \)

Example with more-formal Logic

- **A searching program:**
  - **Input assertion:**
    - \( a == a_0 \)
  - **Output assertion:**
    - \( a == a_0 \)
    - \( (\forall i) a[i] == v \) \( \Rightarrow a[m] == v \), and
    - \( (\forall i) a[i] != v \) \( \Rightarrow m == -1 \)

Forms of Correctness

The main forms of correctness are:

- **Partial Correctness (PC):**
  - If the input assertion is satisfied when the program starts and the program eventually terminates, then the output assertion will be satisfied upon termination.

- **Termination:**
  - If the input assertion is satisfied, then the program will eventually terminate.

- **Total Correctness (TC):**
  - Defined to be PC + termination

Forms of Correctness

The forms PC and termination are separated because it often is easier to prove them separately.

(Principle of "Separation of concerns")
**Example**

- Construct, and prove correct, a program that squares a number using only addition.

- This technique goes back to Babbage's "Difference Engine", 1832.

**Squaring Program Idea**

- Look at squares:
  
  1, 2, 4, 9, 16, 25, ...

- Look at first differences:
  
  1, 3, 5, 7, 9, ...

- Note that first differences are:
  
  - odd
  
  - differ by 2 each time

**Synthesis of Squaring Program**

- Assert \( n \geq 0 \) is the number to be squared.
- \[
\text{int sum = 0;} \\
\text{// accumulated sum} \\
\text{int i = 0;} \\
\text{// step number} \\
\text{int fd = 1;} \\
\text{// first difference} \\
\text{int sd = 2;} \\
\text{// second difference} \\
\text{while (i < n)} \\
\{
\text{sum = sum + fd;} \\
\text{fd = fd + sd;} \\
\text{i = i + 1;} \\
\}
\]
- Assert \( \text{sum} = n^2 \)
- What/where to add intermediate assertions?

**Prove the Squaring Program**

- Using assertion-based reasoning ("loop invariant") for PC.

- Use "energy function" or "variant" for termination.
Using Primed Variables

- A useful technique to avoid confusion:
  - Each variable has a primed and un-primed version.
  - Unprimed: indicates the value of variable before the step or iteration.
  - Primed: indicates the value after the step or iteration.

Using Primed Variables

- Instead of assignment statements:
  \[ \text{sum} = \text{sum} + \text{fd}; \]
  \[ \text{fd} = \text{fd} + \text{sd}; \]
  \[ i = i+1; \]

  use mathematical equations:
  \[ \text{sum}' = \text{sum} + \text{fd} \]
  \[ \text{fd}' = \text{fd} + \text{sd} \]
  \[ \text{sd}' = \text{sd} \quad \text{(no change)} \]

  \[ i' = i+1 \]

- It is easier to reason about equations:

  \[ \text{sum} = i \quad \forall i \leq n \]
  \[ \text{fd} = 2i + 1 \quad \forall i \leq n \]
  \[ \text{sd} = 2 \quad \forall i \leq n \]

Transition Induction

- To prove that an assertion (e.g. a loop invariant) is true whenever the program reaches the point to which the assertion is attached:
  - Basis: Show that the assertion is true the first time.
  - Induction: Show that if the assertion is true the current time (unprimed variables), then it is true the next time (primed variables), provided there is a next time.

Loop Invariant Positioning

- The loop invariant always goes right before the test for continuing to the next iteration.

  - In particular, it is "tested":
    - Before the first iteration, if any.
    - Before each additional iteration.
    - Before exit.

Proof Write-Up (1)

- Input denotes the input assertion.
- Output denotes the output assertion.
- Test denotes the test in the while or for loop.
- Body denotes the equations describing the body of the loop. It expresses primed variables in terms of unprimed ones.

Proof Write-Up (2)

1. Identify the input and output assertions.
2. Clearly state your loop invariant Inv.
3. Express the three verification conditions:
   a. (Inv and !Test) => Output  \( \text{("Base")} \)
   b. (Input and Initialization) => Inv  \( \text{("Base")} \)
   c. (Inv and Test and Body) => Inv'  \( \text{("Induction Step")} \)
4. Prove the verification conditions.

Note that Inv' is Inv with the non-changing variables primed.
Example: Proof Write-Up for the Squaring Program (1: Assertions)

- Input: \( n \geq 0 \)
- Output: \( \text{sum} \equiv n^2 \)
- Initialization:
  \[ \text{sum} = 0 \text{ and } i = 0 \text{ and } \text{fd} = 1 \text{ and } \text{sd} = 2 \]
- Test: \( i \leq n \)
- Body:
  \[ \text{sum}' = \text{sum} + \text{fd} \]
  \[ \text{fd}' = \text{fd} + \text{sd} \]
  \[ \text{sd}' = \text{sd} \]
  \[ i' = i + 1 \]

Example: Proof Write-Up for the Squaring Program (2: Invariant)

Invariant (created):
\[ i \leq n \]
\[ \text{and } \text{sum} \equiv i^2 \]
\[ \text{and } \text{fd} = 2i + 1 \]
\[ \text{and } \text{sd} = 2 \]

Example: Proof Write-Up for the Squaring Program (3: Verification Conditions)

- Assume: \( [i \leq n \text{ and } \text{sum} \equiv i^2 \text{ and } \text{fd} = 2i + 1 \text{ and } \text{sd} = 2] \) and \( \{i < n\} \)
- To show: \( \text{sum} \equiv n^2 \)
  - From \( i < n \) and \( i \equiv n \) we have \( i \equiv n \).
  - From \( \text{sum} \equiv i^2 \) we have \( \text{sum} \equiv n^2 \).

Example: Proof Write-Up for the Squaring Program (4a: Proof of Verification Conditions)

- Assume: \( [i \equiv n \text{ and } \text{sum} \equiv i^2 \text{ and } \text{fd} = 2i + 1 \text{ and } \text{sd} = 2] \) and \( \{i \equiv n\} \)
- To show: \( \text{sum} \equiv n^2 \)
  - From \( i \equiv n \) and \( i \equiv n \) we have \( i \equiv n \).
  - From \( \text{sum} \equiv i^2 \) we have \( \text{sum} \equiv n^2 \).

Example: Proof Write-Up for the Squaring Program (4b: Proof of Verification Conditions)

- Assume: \( \{i \equiv 0\} \text{ and } \{\text{sum} \equiv 0 \text{ and } i \equiv 0 \text{ and } \text{fd} = 1 \text{ and } \text{sd} = 2\} \)
- To show: \( \{i \equiv n \text{ and } \text{sum} \equiv i^2 \text{ and } \text{fd} = 2i + 1 \text{ and } \text{sd} = 2\} \)
  - \( i \equiv n \)
    - From the assumption, \( i \equiv 0 \text{ and } n \equiv 0 \). Therefore \( i \equiv n \).
  - \( \text{sum} \equiv i^2 \)
    - From the assumption, \( \text{sum} \equiv 0 \text{ and } i \equiv 0 \). Therefore \( \text{sum} \equiv i^2 \).
  - \( \text{fd} = 2i + 1 \)
    - From the assumption, \( \text{fd} = 1 \text{ and } i \equiv 0 \). Therefore \( \text{fd} = 2i + 1 \).
  - \( \text{sd} = 2 \)
    - From the assumption, \( \text{sd} = 2 \).

Example: Proof Write-Up for the Squaring Program (4c: Proof of Verification Conditions)

- Assume: \( \{i \equiv n \text{ and } \text{sum} \equiv i^2 \text{ and } \text{fd} = 2i + 1 \text{ and } \text{sd} = 2\} \) and \( \{i \equiv n\} \)
- To show: \( \{i \equiv n \text{ and } \text{sum} \equiv i^2 \text{ and } \text{fd} = 2i + 1 \text{ and } \text{sd} = 2\} \)
  - \( i \equiv n \)
    - From \( i \equiv n \) and \( i \equiv n \) we have \( i \equiv n \).
  - \( \text{sum} \equiv i^2 \)
    - From \( i \equiv n \) and \( i \equiv n \) we have \( i \equiv n \).
  - \( \text{fd} = 2i + 1 \)
    - From \( i \equiv n \) and \( i \equiv n \) we have \( i \equiv n \).
  - \( \text{sd} = 2 \)
    - From \( i \equiv n \) and \( i \equiv n \) we have \( i \equiv n \).
  - \( \text{sd} = 2 \)
Energy Function Principle for Proving Termination

- An energy function (also called a "variant") is a function that one fabricates:
  \[ f : \text{States} \rightarrow \text{Natural Numbers} \]
  (therefore always non-negative)

- If the same point in the program is visited with state \( s_1 \) after state \( s_2 \), then necessarily:
  \[ f(s_2) < f(s_1) \]

Exercise: Prove this "cubing" program

```java
int n, i, sum, fd, sd;
// assert n > 0
sum = 0;
sd = 6;
for( i = 0; i < n; i++ )
{
    sum = sum + fd;
    fd = fd + sd;
    sd = sd + 6;
}
return sum;
// assert sum == n^3
```

Example

- Construct, and prove correct, a searching program.
- The need to prove that a program is correct can have beneficial effects on the structure of the program:
  - You need to be able to explain (in logic) what all the pieces do.

Search Program Specification

- Input assertion:
  \[ a := a_0 \]
- Output assertion:
  \[ a := a_0 \text{ and } \forall (\text{if } a[i] = v \text{ then } a[r] = v, \text{ and } \forall (\text{if } a[i] \neq v \text{ then } r := -1) \]
  > where \( r \) represents the returned value
  > For simplicity, we assume that the quantifiers range over the valid indices of the array only.

A Search Program

```java
static int search(int a[], int v)
{
    for( int i = 0; i < a.length; i++ )
    {
        if( a[i] == v )
        {
            return i;
        }
    }
    return -1;
}
```

Note

- \( \forall \text{if } a[i] = v \)
- \( \forall \text{if } a[i] \neq v \)
- The above two conditions are complementary; one or the other always holds, but not both.
- This is a variant of DeMorgan's Law.
Adding Input and Output Assertions (shown in red)

\( a = a_0 \)

static int search(int a[], int v)
{
    for( int i = 0; i < a.length; i++ )
    {
        if( a[i] == v )
            return i;
        }  
    return -1;

    ([v \in a] \Rightarrow \forall v \in a \Rightarrow v = v) \land \forall v \in a \Rightarrow v \land \land \land \land \land

Adding Intermediate Assertions (1)

global: \( a = a_0 \)

static int search(int a[], int v)
{
    (\forall k < i) a[k] \Rightarrow v
    for( int i = 0; i < a.length; i++ )
    {
        if( a[i] == v )
            return i;
        }  
    return -1;

    ([\exists i \in a] \Rightarrow \forall v \in a \Rightarrow \forall v \in a \Rightarrow v \land \forall v \in a \Rightarrow v \land \land \land \land \land

Adding Intermediate Assertions (2)

global: \( a = a_0 \)

static int search(int a[], int v)
{
    (\forall k < i) a[k] \Rightarrow v
    for( int i = 0; i < a.length; i++ )
    {
        if( a[i] == v )
            return i;
        }  
    return -1;

    ([\exists i \in a] \Rightarrow \forall v \in a \Rightarrow \forall v \in a \Rightarrow v \land \forall v \in a \Rightarrow v \land \land \land \land \land

Adding Intermediate Assertions (3)

global: \( a = a_0 \)

static int search(int a[], int v)
{
    (\forall k < i) a[k] \Rightarrow v
    for( int i = 0; i < a.length; i++ )
    {
        if( a[i] == v )
            return i;  
        }  
    return -1;

    ([\exists i \in a] \Rightarrow \forall v \in a \Rightarrow \forall v \in a \Rightarrow v \land \forall v \in a \Rightarrow v \land \land \land \land \land

Adding Intermediate Assertions (4)

global: \( a = a_0 \)

static int search(int a[], int v)
{
    (\forall k < i) a[k] \Rightarrow v
    for( int i = 0; i < a.length; i++ )
    {
        if( a[i] == v )
            return i;
        }  
    return -1;

    ([\exists i \in a] \Rightarrow \forall v \in a \Rightarrow \forall v \in a \Rightarrow v \land \forall v \in a \Rightarrow v \land \land \land \land \land

Globalizing an Assertion

\( a = a_0 \)

static int search(int a[], int v)
{
    for( int i = 0; i < a.length; i++ )
    {
        if( a[i] == v )
            return i;
        }  
    return -1;

    ([\exists i \in a] \Rightarrow \forall v \in a \Rightarrow \forall v \in a \Rightarrow v \land \forall v \in a \Rightarrow v \land \land \land \land \land

Intermediate (3)

Intermediate (4)

Intermediate (5)

Intermediate (6)
Assume that $r = i$.

The point.

Is between time

true?

hold!!

The first time, the value of i is 0.

The assertion then is

Is this assertion true?

In between those two times:

It was established that $a[i] = v$ and $i < a.length$

then i became the value $i+1$.

Thus $a[i] = v$ and $i < a.length$ still holds!!
Proving the Assertions (6)

So far we have established the loop invariant: \( (i < k) \land a[k] \neq v \lor i < a.length) \)

Now concentrate on establishing the output assertion:

There are two places the program can exit:

In the first place, it is obvious that \( a[r] = v \), since we just tested \( a[i] = v \) and set \( r = i \).

In the second place, we claim that \( (i < k \land a.length) \land a[k] \neq v \land r = -1 \)

How do we show the first conjunct? This conjunct is obvious.

Thought Exercise

- What if the search program were instead a binary search of an ordered array?
- What would the loop invariant be?
- How would you prove that it holds?

Structural Induction

- Complementary approach to correctness.
- Induction on data values or structure.
  - Resembles classical mathematical induction
  - Proves PC + termination at the same time.
- Useful for functional programs.
  - due to McCarthy transformation, this means all programs
  - Set-up can be trickier.

Structural Induction for Lists

- Let \( P \) be a property of (open) lists
- To show \( (\forall L) \ P(L) \)
  - it is sufficient to show:
    - \( P(\) \)
    - \( (\forall L) \ (\forall A) \ P(L) \Rightarrow P([A \mid L]) \) // Induction step

Structural Induction Example

- \( \text{append}([], M) \Rightarrow M \):
- \( \text{append}(A \mid L), M) \Rightarrow [A \mid \text{append}(L, M)] \):

  Show
  - \( (\forall L) \ (\forall M) \)
    - \( \text{length(append}(L, M)) = \text{length}(L) + \text{length}(M) \)
  - Here \( P(L) \) is the property:
    - \( (\forall M) \text{length}(\text{append}(L, M)) = \text{length}(L) + \text{length}(M) \)
Structural Induction Proof: Basis

- To show \( P([]) \): 
  \[
  \text{length}(\text{append}([], M)) = \text{length}([]) + \text{length}(M)
  \]
- We know from the definition of append that 
  \( \text{append}([], M) = M \).
- We also know that 
  \( \text{length}([]) = 0 \).
- So the thing to be shown reduces to:
  \[
  \text{length}(M) = 0 + \text{length}(M)
  \]
- which is true since the equation reduces to an identity.

Structural Induction Proof: Induction Step

- To show \( P(L) \Rightarrow P([A \mid L]) \):
- Assume \( P(L) \):
  \[
  \text{length}(\text{append}(L, M)) = \text{length}(L) + \text{length}(M)
  \]
- To show \( P([A \mid L]) \):
  \[
  \text{length}(\text{append}([A \mid L], M)) = \text{length}([A \mid L]) + \text{length}(M)
  \]
- From the definition of append, we know that the “to show” part is the same as:
  \[
  \text{length}([A \mid \text{append}(L, M)]) + \text{length}(L) = \text{length}([A \mid L]) + \text{length}(M)
  \]
- Since \( \text{length}([A \mid X]) = 1 + \text{length}(X) \), the “to show” part is equivalent to:
  \[
  1 + \text{length}(\text{append}(L, M)) = 1 + \text{length}(L) + \text{length}(M)
  \]
- By the induction hypothesis, we can substitute for \( \text{length}(\text{append}(L, M)) \) an expression that renders this as an identity.

Fine Points

- To be perfectly rigorous, we’d need to axiomatize information such as:
  \[
  \text{length}([A \mid X]) = 1 + \text{length}(X)
  \]
- We’d want to formalize the definition of length, addition, etc.
- There are software systems such as **ACL2** that automate many proofs of this nature.

Undecidability

- There is no tool that can determine whether an arbitrary predicate logic formula is valid.
- If there were, the halting problem would be solvable.
- This is because the computation by a Turing machine can be captured as an appropriate logical formula, one which is valid iff the Turing machine fails to halt.