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.
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:**
  \[(\exists i)(i > 0 \land i < n) \land a[i] < a[m]\]

  ("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 $a == a_0$
- Output assertion: Index $m$ is index of a maximal value in $a$ and $a == a_0$

Why is this necessary?

- 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; $a == a_0$ prevents the array from being modified overall.
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 as $x = \text{gcd}(x_0, y_0)$].

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.
An array-sorting program:

- **Input assertion**: $a = a_0$
- **Output assertion**:
  - The elements of $a$ are those of $a_0$ as a “bag” (or “multi-set”) i.e. the same number of each element.
  - The elements of $a$ are in increasing order

Implicit here is that the notion that order makes sense for the elements.
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 \( a \)
A searching program:

Input assertion:
- \( a = a_0 \)

Output assertion:
- \( a = a_0 \) and
- \( (\forall i) \ a[i] = v \implies a[m] = v, \) and
- \( (\forall i) \ a[i] \neq v \implies m = -1 \)
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.
Babbage

“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 ≥ 0 is the number to be squared.
- int sum = 0; // accumulated sum
  int i = 0; // step number
  int fd = 1; // first difference
  int sd = 2; // second difference
  while( i < n )
  {
    sum = sum + fd;
    fd = fd + sd;
    i = i+1;
  }
- Assert sum == n^2
- What/where to add intermediate assertions?
Synthesis of Squaring Program

- Assert $n > 0$ is the number to be squared.
- ```
    int sum = 0;               // accumulated sum
    int i = 0;                 // step number
    int fd = 1;                // first difference
    int sd = 2;                // second difference
    while( i < n ) {
        // sum == i^2 [] i <= n [] fd == 2*i+1 [] sd == 2
        sum = sum + fd;
        fd = fd + sd;
        i = i+1;
    }
    Assert sum == n^2
```
- The arrow-ed assertion is called a loop invariant.
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
Using Primed Variables

- Instead of assignment statements:
  \[
  \begin{align*}
  \text{sum} &= \text{sum} + \text{fd}; \\
  \text{fd} &= \text{fd} + \text{sd}; \\
  i &= i+1;
  \end{align*}
  \]
  
  use mathematical equations:
  \[
  \begin{align*}
  \text{sum}' &= \text{sum} + \text{fd} \\
  \text{fd}' &= \text{fd} + \text{sd} \\
  \text{sd}' &= \text{sd} \quad \text{(no change)} \\
  i' &= i+1
  \end{align*}
  \]

- It is easier to reason about equations:

  \[
  \begin{align*}
  \text{sum} &= i^2 \quad i < n \quad \text{fd} &= 2i+1 \quad \text{sd} = 2 \\
  \text{i < n} \quad \text{implies} \\
  \text{sum'} &= i'^2 \quad i' < n \quad \text{fd'} &= 2i'+1 \quad \text{sd'} = 2
  \end{align*}
  \]
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.
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  ("Final")
   
   b. (Input and Initialization) => Inv  ("Basis")
   
   c. (Inv and Test and Body) => Inv'  ("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** is: \( n \geq 0 \)

- **Output** is: \( \text{sum} = n^2 \)

- **Initialization** is:
  
  \[
  \text{sum} = 0 \text{ and } i = 0 \text{ and } fd = 1 \text{ and } sd = 2
  \]

- **Test** is: \( i < n \)

- **Body** is:
  
  \[
  \begin{align*}
  \text{sum}' &= \text{sum} + fd \\
  \text{and } fd' &= fd + sd \\
  \text{and } sd' &= sd \\
  \text{and } i' &= i + 1
  \end{align*}
  \]
Example: Proof Write-Up for the Squaring Program (2:Invariant)

Invariant (created):

\[ i \leq n \]
\[ \text{and } \text{sum} = i^2 \]
\[ \text{and } \text{fd} = 2i + 1 \]
\[ \text{and } \text{sd} = 2 \]
Example: Proof Write-Up for the Squaring Program (3: Verification Conditions)

a. (Inv and !Test) => Output:
   \[ i \leq n \text{ and } \sum = i^2 \text{ and } fd = 2i + 1 \text{ and } sd = 2 \] \[ \text{ and } [ i < n ] \]
   \[ \text{Inv} \]
   \[ \text{!Test} \]
   \[ \Rightarrow [\sum = n^2] \]
   \[ \text{Output} \]

b. (Input and Initialization) => Inv:
   \[ N \geq 0 \] \[ \text{and} \] \[ \sum = 0 \text{ and } i = 0 \text{ and } fd = 1 \text{ and } sd = 2 \]
   \[ \text{Input} \]
   \[ \text{Initialization} \]
   \[ \Rightarrow [i \leq n \text{ and } \sum = i^2 \text{ and } fd = 2i + 1 \text{ and } sd = 2] \]
   \[ \text{Inv} \]


c. (Inv and Test and Body) => Inv'
   \{ [i \leq n \text{ and } \sum = i^2 \text{ and } fd = 2i + 1 \text{ and } sd = 2] \text{ and } [i < n] \}
   \[ \text{Inv} \]
   \[ \text{Test} \]
   \[ \text{and} \] \[ \sum' = \sum + fd \text{ and } fd' = fd + sd \text{ and } sd' = sd \text{ and } i' = i + 1 \]
   \[ \text{Body} \]
   \[ \Rightarrow [i' \leq n \text{ and } \sum' = i'^2 \text{ and } fd' = 2i' + 1 \text{ and } sd' = 2] \]
   \[ \text{Inv'} \]
Example: Proof Write-Up for the Squaring Program
(4a: Proof of Verification Conditions)

● Assume:

\[ \begin{align*}
&i \leq n \text{ and } \text{sum} = i^2 \text{ and } \text{fd} = 2i + 1 \text{ and } \text{sd} = 2 \\
&\text{and } [ !(i < n) ]
\end{align*} \]

● To show: \text{sum} = n^2

- From \( i \leq n \) and \( !(i < n) \) we have \( i = n \).

- From \( \text{sum} = i^2 \), we have \( \text{sum} = n^2 \).
Example: Proof Write-Up for the Squaring Program
(4b: Proof of Verification Conditions)

- Assume: \([n \geq 0] \text{ and } [\text{sum} = 0 \text{ and } i = 0 \text{ and } \text{fd} = 1 \text{ and } \text{sd} = 2]\)

- To show: \([i \leq n \text{ and } \text{sum} = i^2 \text{ and } \text{fd} = 2i + 1 \text{ and } \text{sd} = 2]\)
  
  - show: \(i \leq n\)
    - From the assumptions, \(i = 0\) and \(n \geq 0\). Therefore \(i \leq n\).

  - show: \(\text{sum} = i^2\)
    - From the assumptions, \(\text{sum} = 0\) and \(i = 0\). Therefore \(\text{sum} = i^2\).

  - show: \(\text{fd} = 2i + 1\)
    - From the assumptions, \(i = 0\) and \(\text{fd} = 1\). Therefore \(\text{fd} = 2i + 1\).

  - show: \(\text{sd} = 2\)
    - From the assumptions, \(\text{sd} = 2\).
Example: Proof Write-Up for the Squaring Program

(4c: Proof of Verification Conditions)

- Assume: \([i \leq n \text{ and } \text{sum} == i^2 \text{ and } \text{fd} == 2*i + 1 \text{ and } \text{sd} == 2] \text{ and } [i < n]\)

- To show: \([i' \leq n \text{ and } \text{sum}' == i'^2 \text{ and } \text{fd}' == 2*i' + 1 \text{ and } \text{sd}' == 2]\)

  - show: \(i' \leq n\)
    - From \(i' == i + 1 \text{ and } i < n\) we have \(i' \leq n\).
  
  - show: \(\text{sum}' == i'^2\)
    - From \(i' = i + 1\), we have \(i'^2 == i^2 + 2*i + 1\).
    - But \(\text{sum}' == \text{sum} + \text{fd} \text{ and } \text{sum} == i^2 \text{ and } \text{fd} == 2*i + 1\). So \(\text{sum}' == i'^2\).
  
  - show: \(\text{fd}' == 2*i' + 1\)
    - From \(\text{fd}' == \text{fd} + \text{sd} \text{ and } \text{fd} == 2*i + 1 \text{ and } \text{sd} == 2\), we get \(\text{fd}' == 2*(i+1) + 1\). but \(i' == i+1 \), so \(\text{fd}' == 2*i' + 1\).
  
  - show: \(\text{sd}' == 2\)
    - From \(\text{sd}' == \text{sd} \text{ and } \text{sd} == 2\) we get \(\text{sd}' == 2\).
Energy Function Principle for Proving Termination

- An energy function (also called a “variant”) is a function that one fabricates:
  \[ f: \text{States} \mapsto \text{Natural Numbers} \]
  (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

```c
int n, i, sum, fd, sd;

// assert n > 0
  sum = 0;
  fd = 1;
  sd = 6;

for( i = 0; i < n; i++ )
  {
    sum = sum + fd;
    fd = fd + sd;
    sd = sd + 6;
  }
return sum;

// assert sum == n^3
```
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:**
  - $a == a_0$, and
  - $((\exists i) a[i] == v) \implies a[r] == v$, and
  - $((\exists i) a[i] ! = v) \implies r == -1$

- where $r$ represents the *returned value*

- For simplicity, we assume that the quantifiers range over the valid indices of the array only.
The above two conditions are complementary; one or the other always holds, but not both.

This is a variant of DeMorgan’s Law.
A Search Program

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

Is this program correct?
If so, how shall we prove it?
Adding Input and Output Assertions *(shown in red)*

\[
a == a_0
\]

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

\[
[a == a_0] ^ [((\Box i) a[i] == v) \Box a[r] == v] ^ [((\Box i) a[i] != v) \Box r == -1]
\]
Globalizing an Assertion

global: $a = a_0$

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

$[(\exists i) a[i] = v] \land a[r] = v] \land [((\forall i) a[i] = v) \land r = -1]$
Adding Intermediate Assertions (1)

global: \( a = a_0 \)

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

\[ \left\langle \begin{array}{c}
    \exists \; i \; a[i] = v \\
    a[r] = v \\
    \forall \; i \; a[i] \neq v \\
    r = -1
\end{array} \right\rangle \]


Adding Intermediate Assertions (2)

global: \( a = a_0 \)

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

\[((\exists i) a[i] = v) \land a[r] = v \land ((\forall i) a[i] \neq v) \land r = -1\]
Adding Intermediate Assertions (3)

\[
\begin{align*}
\text{global: } & a == a_0 \\
\text{static int search(int a[], int v)} & \\
\{ & (\forall k < i) a[k] != v \\
\text{for( int i = 0; } & i < a.length; i++ ) \\
\{ & \quad \text{if( a[i] == v )} \\
\{ & \quad \quad r = i; \\
\quad & \quad \quad \text{return i; } a[r] == v \\
\} & (\forall k < i + 1) a[k] != v \\
\} & r = -1; \quad \text{note} \\
\text{return -1; } & \end{align*}
\]

\[((\exists i) a[i] == v) \quad a[r] == v \quad [((\forall i) a[i] != v) \quad r == -1]
Adding Intermediate Assertions (4)

```java
public class SearchExample {
    // Initial global assertion
    global: a == a_0

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

    // Intermediate assertions
    [((\exists i) a[i] == v) □ a[r] == v] ^ [((\forall i) a[i] != v) □ r == -1]
}
```
Proving the Assertions (1)

global: \( a == a_0 \)

static int search(int a[], int v)
{
    \( \forall k < i \) a[k] != v \quad i < a.length

    for( int i = 0; i < a.length; i++ )
    {
        if( a[i] == v )
        {
            \( r = i; \)
            return i; \( a[r] == v \)
        } \quad (\forall k < i + 1) a[k] != v \quad i < a.length
    }
    \( r = -1; \) \quad (\forall k) a[k] != v \quad r == -1
}
[((\forall i) a[i] == v) \quad a[r] == v \quad [((\forall i) a[i] != v) \quad r == -1]
We want to establish that the loop invariant is true every time the program gets to the point shown (right before the test \( i < a\.length \)).

Use transition induction:

**Basis:** The assertion is true the first time the program gets to that point.

**Induction Step:** Assume it is true for an arbitrary time; show it is true the next time (if there is a next time).
To show: Basis: The assertion is true the first time the program gets to that point.

The first time, the value of i is 0. The assertion then is

\[ ((\exists k < i) \ a[k] \neq v) \land i \leq a.length) \]

Is this assertion true?
Proving the Assertions (4)

\(((\forall k < i) a[k] \neq v \cap i \leq a.length)\)

Induction Step: Assume it is true for an arbitrary time; show it is true the next time (if there is a next time).

Assume the assertion is true now. Suppose there is a next time the program gets to that point.

What happened in between?
Induction Step: Assume it is true for an arbitrary time; show it is true the next time (if there is a next time).
Assume the assertion is true now.
Suppose there is a next time the program gets to that point.

In between those two times:
It was established that \( a[i] \neq v \) and \( i < a.length \) then \( i \) became the value \( i + 1 \).
Thus \( ((\forall k < i) a[k] \neq v \land i \leq a.length) \) still holds!!
In more detail (pay careful attention to $\leq$ vs. $<$)

$$((k < i) \ a[k] \neq v \ \Box i \leq a.length) \text{ before the iteration}$$

$$a[i] \neq v \ \Box i < a.length \text{ the iteration reveals}$$

$$((k \leq i) \ a[k] \neq v \ \Box i < a.length) \text{ after the body}$$

But then $i = i + 1$;

$$((k < i) \ a[k] \neq v \ \Box i \leq a.length) \text{ starting the next iteration}$$
So far we have established the loop invariant: $$((\forall k < i) \ a[k] \neq v \land i \leq a\.length)$$

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

$$((\forall k < a\.length) \ a[k] \neq v \land r == -1$$

How do we show the first conjunct? This conjunct is obvious.
We use the established the loop invariant: \(((\forall k < i) \text{ a}[k] \neq v \land i \leq \text{a}.\text{length})\)

The program can get to the second exit only if \(i > \text{a}.\text{length}\).

Substituting in the invariant:

\[((\forall k < \text{a}.\text{length}) \text{ a}[k] \neq v)\]

But this is the same as

\[((\forall k) \text{ a}[k] \neq v)\]

The program has now been established to be **partially correct**.

To show **total correctness**, we must establish termination as well. Termination is “obvious” because the number of steps in the loop is bounded by the length of the array.
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
  \[(L) \ P(L)\]

- it is sufficient to show:
  - \( P([],) \) // Basis
  - \((L) (A) \ P(L) \Rightarrow P([A \mid L])\) // Induction step
Structural Induction Example

- \( \text{append}([], M) \Rightarrow M; \)
- \( \text{append}([A | L], M) \Rightarrow [A | \text{append}(L, M)]; \)

- Show
  \( (\forall L)(\forall M) \)
  \[ \text{length} (\text{append}(L, M)) = \text{length}(L) + \text{length}(M) \]

- 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 $P([])$:
  $$(\forall M) \text{length}(\text{append}([], M)) = \text{length}([]) + \text{length}(M)$$

- 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:
  $$(\forall M) \text{length}(M) == 0 + \text{length}(M)$$
  which is true since the equation reduces to an identity.
Structural Induction Proof: Induction Step

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