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 is 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**:
    (□i) (i ≥ 0 □ i < n) □ 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.

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 == 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.

Specification Examples

- 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

Example with more-formal Logic

- A searching program:
  - Input assertion:
    - a == a_0
  - Output assertion:
    - a == a_0 and
    - ((□i) a[i] == v) □ a[m] == v, and
    - ((□i) a[i] != v) □ 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

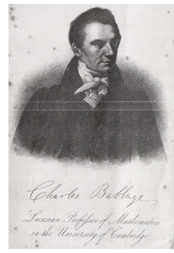
Forms of Correctness

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

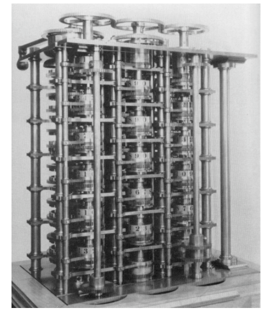
(Principle of "Separation of concerns")

Example

- Construct, and prove correct, a program that squares a number using only addition.
- This technique goes back to Babbage's "Difference Engine", 1832.



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 $\geq$ 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 $\geq$ 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 == i² □ 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:**

```
sum = sum + fd;
fd = fd + sd;
i = i+1;
```

  use mathematical equations:

  sum' == sum + fd
  fd' == fd + sd
  sd' == sd **(no change)**
  i' == i+1

- It is easier to reason about equations:
- **sum == i^2 □ i $\le$ n □ fd == 2*i+1 □ sd == 2 □ i < n**
  implies
- **sum' == i'^2 □ i' $\le$ n □ fd' == 2*i'+1 □ sd' == 2**

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 ("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 >= 0
- **Output** is: sum == n^2
- **Initialization** is:
  sum == 0 and i == 0 and fd == 1 and sd == 2
- **Test** is: i < n
- **Body** is: sum' == sum + fd
  and fd' == fd + sd
  and sd' == sd
  and i' == i + 1

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

Invariant (created):

i <= n
and sum == i^2
and fd == 2*i + 1
and sd == 2

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

```
a. (Inv and !Test) => Output:
      [i <= n and sum == i^2 and fd == 2*i + 1 and sd == 2] and [ !(i < n) ]
                   Inv                                            !Test
   => [sum == n^2]
        Output
```

```
b. (Input and Initialization) => Inv:
      [N >= 0] and [sum == 0 and i == 0 and fd == 1 and sd == 2]
       Input              Initialization

   => [i <= n and sum == i^2 and fd == 2*i + 1 and sd == 2]
                    Inv
```

```
c. (Inv and Test and Body) => Inv'
      { [i <= n and sum == i^2 and fd == 2*i + 1 and sd == 2] and [i < n]
                     Inv                                          Test

   and [sum' == sum + fd and fd' == fd + sd  and  sd' == sd  and  i' == i + 1] }
                     Body

   => [i' <= n and sum' == i'^2 and fd' == 2*i' + 1 and sd' == 2]
                     Inv'
```

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

- Assume:
  [i <= n and sum == i^2 and fd == 2*i + 1 and sd == 2]
  and [!(i < n)]
- To show: sum == n^2
  - From i <= n and !(i < n) we have i == n.
  - From sum == i^2, we have sum == n^2.

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

- Assume: [n >= 0] and [sum == 0 and i == 0 and fd == 1 and sd == 2]
- To show: [i <= n and sum == i^2 and fd == 2*i + 1 and sd == 2]
  - show: i <= n
    - From the assumptions, i == 0 and n >= 0. Therefore i <= n.
  - show: sum == i^2
    - From the assumptions, sum == 0 and i == 0. Therefore sum == i^2.
  - show: fd == 2*i + 1
    - From the assumptions, i == 0 and fd == 1. Therefore fd == 2*i + 1.
  - show: sd == 2
    - From the assumptions, sd == 2.

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

- Assume: [i <= n and sum == i^2 and fd == 2*i + 1 and sd == 2] and [i < n]
  and [sum' == sum + fd and fd' == fd + sd and sd' == sd and i' == i + 1]
- To show: [i' <= n and sum' == i'^2 and fd' == 2*i' + 1 and sd' == 2]
  - show: i' <= n
    - From i' == i + 1 and i < n we have i' <= n.
  - show: sum' == i'^2
    - From i' =- i + 1, we have i'^2 == i^2 + 2*i + 1.
    - But sum' == sum + fd and sum == i^2 and fd == 2*i + 1. So sum' == i'^2.
  - show: fd' == 2*i' + 1
    - From fd' == fd + sd and fd == 2*i + 1 and sd == 2, we get
      fd' == 2*(i+1) + 1, but i' == i+1, so fd' == 2*i' + 1.
  - show: sd' == 2
    - From sd ' == sd and sd == 2 we get sd' == 2.

Energy Function Principle for Proving Termination

- An **energy function** (also called a "variant") is a function that one **fabricates**:
  f:States → 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

```
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
```

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:
  - $a == a_0$, and
  - $((\exists i)\ a[i] == v) \Rightarrow a[r] == v$, and
  - $((\forall i)\ a[i] != v) \Rightarrow r == -1$
- where r represents the *returned value*
- For simplicity, we assume that the quantifiers range over the valid indices of the array only.

Note

  - $((\exists i)\ a[i] == v)$
  - $((\forall i)\ a[i] != v)$
- 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 == a0
static int search(int a[], int v)
  {
  for( int i = 0; i < a.length; i++ )
    {
    if( a[i] == v )
      {r = i;          r represents the returned value
      return i;
      }
    } r = -i;          r represents the returned value
  return -1;
  }
[a == a0] ^ [((∃ i) a[i] == v) ⇒ a[r] == v] ^ [((∀i) a[i] != v) ⇒ r == -1]
```

Globalizing an Assertion

```
global: a == a0
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;
  }
[((∃ i) a[i] == v) ⇒ a[r] == v] ^ [((∀i) a[i] == v) ⇒ r == -1]
```

Adding Intermediate Assertions (1)

```
global: a == a0
static int search(int a[], int v)
  {          ((∀k < i) a[k] != v)
  for( int i = 0; i < a.length; i++ )
    {
    if( a[i] == v )
      { r = i;
      return i;
      }
    } r = -1;
  return -1;
  }
[((∃ i) a[i] == v) ⇒ a[r] == v] ^ [((∀i) a[i] != v) ⇒ r == -1]
```

Adding Intermediate Assertions (2)

```
global: a == a0
static int search(int a[], int v)
  {          ((∀k < i) a[k] != v)
  for( int i = 0; i < a.length; i++ )
    {
    if( a[i] == v )
      { r = i;
      return i;  a[r] == v
      }
    } r = -1;
  return -1;
  }
[((∃ i) a[i] == v) ⇒ a[r] == v] ^ [((∀i) a[i] != v) ⇒ r == -1]
```

Adding Intermediate Assertions (3)

```
global: a == a0
static int search(int a[], int v)
  {          ((∀k < i) a[k] != v)
  for( int i = 0; i < a.length; i++ )
    {
    if( a[i] == v )
      { r = i;
      return i;  a[r] == v
      }  ((∀k < i + 1) a[k] != v)
    } r = -1;
  return -1;   note
  }
[((∃ i) a[i] == v) ⇒ a[r] == v] ^ [((∀i) a[i] != v) ⇒ r == -1]
```

Adding Intermediate Assertions (4)

```
global: a == a0
static int search(int a[], int v)
  {          ((∀k < i) a[k] != v)
  for( int i = 0; i < a.length; i++ )
    {
    if( a[i] == v )
      { r = i;
      return i;  a[r] == v
      }  ( (∀k < i + 1) a[k] == v)
    } r = -1;
  return -1;  ((∀k) a[k] != v)  ^ r == -1
  }
[((∃ i) a[i] == v) ⇒ a[r] == v] ^ [((∀i) a[i] != v) ⇒ r == -1]
```

Proving the Assertions (1)

global: a == a_0

```
static int search(int a[], int v)
  {              ((□k < i) a[k] != v □ i ≤ a.length)
  for( int i = 0; i < a.length; i++ )
    {
    if( a[i] == v )
      {  r = i;
      return i;  a[r] == v
      }  ((∀k < i + 1) a[k] != v □ i < a.length)
    }  r = -1;
  return -1;  ((□k) a[k] != v)  ^  r == -1
  }
```

[((□i) a[i] == v) □ a[r] == v] ^ [((□i) a[i] != v) □ r == -1]

Proving the Assertions (2)

We want 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).

Proving the Assertions (3)

((□k < i) a[k] != v □ i ≤ a.length)

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

((□k < 0) a[k] != v □ i ≤ a.length)

Is this assertion true?

Proving the Assertions (4)

((□k < i) a[k] != v □ i ≤ 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?

Proving the Assertions (5)

((□k < i) a[k] != v □ i ≤ 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.

In between those two times:
It was established that a[i] != v and i < a.length
then i became the value i +1.
Thus ((□k < i) a[k] != v □ i ≤ a.length) still holds!!

In more detail (pay careful attention to ≤ vs. <)

((□k < i) a[k] != v □ i ≤ a.length) before the iteration

a[i] != v □ i < a.length   the iteration reveals

((□k ≤ i) a[k] != v □ i < a.length) after the body

But then i = i +1;

((□k < i) a[k] != v □ i ≤ a.length) starting the next iteration

Proving the Assertions (6)

So far we have established the loop invariant: $((\forall k < i)\ a[k] \mathrel{!=} v \land i \leq a.length)$

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

$$\underbrace{((\forall k < a.length)\ a[k] \mathrel{!=} v}_{\text{How do we show the first conjunct?}} \land \underbrace{r == -1}_{\text{This conjunct is obvious.}}$$

Proving the Assertions (8)

We use the established the loop invariant: $((\forall k < i)\ a[k] \mathrel{!=} v \land i \leq a.length)$

The program can get to the second exit only if $\boxed{i \geq a.length.}$
Substituting in the invariant:

$$((\forall k < a.length)\ a[k] \mathrel{!=} v)$$

But this is the same as

$$((\forall k)\ a[k] \mathrel{!=} v)$$

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.

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([]) // Basis
  - $(\forall L)\ (\forall A)$ P(L) => P([A | L]) // Induction step

Structural Induction Example

- append([], M) => M;
- append([A | L], M) => [A | append(L, M)];
- Show
  $(\forall L)\ (\forall M)$
  length(append(L, M)) = length(L) + length(M)
- Here P(L) is the property:

  $(\forall M)$ length(append(L, M)) = length(L) + length(M)

Structural Induction Proof: Basis

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

Structural Induction Proof: Induction Step

- To show (∀L) (∀A) P(L) => P([A | L])
- Assume P(L):
  (∀M) length(append(L, M)) = length(L) + length(M)
- To show P([A | L]):
  (∀M) length(append([A | L], M)) = length([A | L]) + length(M)
- From the definition of append, we know that the "to show" part is the same as:
  (∀M) length([A | append(L, M)]) = length([A | L]) + length(M)
- Since length([A | X]) == 1 + length(X), the "to show" part is equivalent to:
  (∀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.

Fine Points

- To be perfectly rigorous, we'd need to axiomatize information such as:

  length([A | X]) == 1 + 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.