CS403: Programming Languages

Notes on Invariants

Introduction

The liberal use of invariants can greatly reduce the number of bugs in your code. The problem is identifying useful invariants. Here is a typical first stab at a binary search routine. In this version, the high and low limits of the search are required to specify a valid range.

```
function search(items,lo,hi,value)
    {
    // hi >= lo
    // value not below index lo
    // value not above index hi
    // if lo > 0 then value >= items[lo]
    // if high < length(items) - 1 then value <= items[hi]

    var result;
    var middle = (lo + hi) / 2;

    if (items[middle] == value)
        result = True;
    else if (lo == hi)
        result = False;
    else if (items[middle] < value)
        result = search(items,middle + 1,hi,value);
    else
        result = search(items,lo,middle - 1,value);

    // if result == False, then value not in items[lo..hi]
    // if result == True, then value in items[lo..hi]

    return result;
    }
```

Note that this version is incorrect in the case of a 2 element array with the first element smaller than the target value. The invariant $hi \geq lo$ would have caught this error. A fix is to add this if clause after the `if (lo == hi)` clause:

```
else if (lo == middle && items[middle] > value)
    result = False;
```

Sometimes changing an invariant sometimes leads to a cleaner implementation. In this version, we insist that *hi* always be greater than *lo* (we will assume that the value of *hi* is one higher than the index of the last element to be searched).

```
function search(items,lo,hi,value)
    {
    // hi > lo
    // value not below index lo
    // value not above hi-1
    // if lo > 0 then value >= items[lo]
    // if high < length(items) then value <= items[hi -1]

    int result;

    int middle = (lo + hi) / 2;

    if (items[middle] == value)
        result = True;
    else if (lo == middle)
        result = False;
```

```
        else if (value > items[middle])
           result = search(items,middle+1,hi,value);
        else
            result = search(items,lo,middle,value);

        // if result == False, then value not in items[lo..hi-1]
        // if result == True, then value in items[lo..hi-1]

        return result;
        }
```

Here's another example. This time we will search a binary tree:

```
    int search(t,value)
        {
        //t points to an actual Tree (e.g. t is not NULL)

        var result;

        if (t.value == value)
            result = True;
        else if (t.left != NULL && t.value < value)
            result = search(t.left,value);
        else if (t.value < value)
            result = False;
        else if (t.right != NULL)
            result = search(t.right,value);
        else
            result = False;

        // if result == False, value is not in tree t
        // if result == True, value is in tree t

        return result;
        }
```

This time, relaxing an invariant leads to a cleaner implementation: int

```
    int search(Tree *t,int value)
        {
        //t points to an actual Tree or t is NULL

        int result;

        if (t == NULL)
            result = False;
        else if (t.value == value)
            result = True;
        else if (t.value < value)
            result = search(t.left,value);
        else
            result = search(t.right,value);

        // if result == False, value is not in tree t
        // if result == True, value is in tree t

        return result;
        }
```

This second version reduced the number of cases from five to four and reduced the total number of tests from five to three.

Assertions

Assertions are run time checks that ensure an invariant holds. In Scam, one can define a simple *assert* function that enforces a given invariant:

```
(define (assert # $invariant)
    (define passed (eval $invariant #))
    (if (not passed)
        (throw
            'assertionFailure
            (string+ "invariant " (string $invariant) " failed!")
            )
        )
    )
```

The *assert* function delays the evaluation of the invariant so that the string form of the invariant can be obtained. This string form is used to generate a helpful exception should evaluation of the invariant result in a false value.
Rewriting the binary seach routine in Scam and using the *assert* function, yields:

```
(define (search items lo hi value)
    (assert (>  hi lo))
    (assert (eq? (linearSearch items 0 lo value) #f))
    (assert (eq? (linearSearch items hi (length items) value) #f))
    (assert (or (= lo 0) (>= value (getElement items lo)))
    (assert (or (= hi (length items) (<= value (getElement items (- hi 1)))))

    (define result)

    (define middle (/ (+ lo hi) 2))

    (cond
        ((= (getElement items middle) value)
            (set! result #t))
        ((= lo middle)
            (set! result #f))
        ((> value (getElement items middle))
            (set! result (search items (+ middle 1) hi value)))
        (else
            (set! result (search items lo middle value)))
        )

    (if (eq? result #f)
        (assert (eq? (linearSearch items lo hi value) #f))
        (assert (eq? (linearSearch items lo hi value) #t))
        )

    result
    )
```

The initial assertions that begin with *or* employ the transformation of an *if*:

```
if E then S
```

to a logical implication:

$E \rightarrow S$

to a logical disjunction:

$\neg E \vee S$

Note also the use of an alternate search method, *linearSearch*, that is used to enforce some of the preconditions and postconditions. A common programming technique is to use a simple, but inefficient, algorithm to verify a complex, but efficient, one.