DM8XX - Advanced Topics in Programming Languages

Spec#

Jakob Lykke Andersen

IMADA

May 25, 2009

The goal of Spec#

- Help us detect bugs.
- Help us prevent bugs.
- Incorporate some of the specifications into the code.

Spec# in general

- An extension of C#.
- Runtime checks of contracts.
- Static verification of contracts.
- None-null types, pre- and post-conditions, invariants, object ownership...

The project

- Static verification of SelectionSort and almost all af Quicksort.
- Small examples with the use of object ownership and states.

Pre- and post-conditions

- Part of the method signature.
- Pre-condition: *requires B*
- Post-condition: *ensures B*

```
static int Incr(int i)
    requires i > 42 otherwise ArgumentOutOfRangeException;
    ensures result == i + 1;
{
    return i+1;
}
```

Assert and assume

- Assert:
  Compile time: This should be true, please try to verify it.
  Runtime: This must be true, crash if it is not.
- Assume:
  Compile time: This is the truth, use it for whatever you want.
  Runtime: This must be true, crash if it is not.

Modifies

- What parts of the heap may be modified by the method.
- Default is `modifies this.*`
- `modifies this.random` disables the default.

Loop invariants

```
for(int i = 0; i < numbers.Length; i++)
    invariant B;
{

}
```

```
int i = 0;
assert(B);
while(i < numbers.Length) {

    i++;
    assert(B);
}
```

Intervals

- $(a : b) \equiv [a; b[$
- $(a..b) \equiv [a; b]$

Exchange

```
static void Exchange(int[]! numbers, int a, int b)
    requires 0 <= a && a < numbers.Length;
    requires 0 <= b && b < numbers.Length;
    // modifies[a], modifies[b] makes the translator throw an
    // exception.
    modifies numbers[*];
    ensures numbers[a] == old(numbers[b]);
    ensures numbers[b] == old(numbers[a]);
    ensures forall(int k in (0:numbers.Length), k != a, k != b;
        numbers[k] == old(numbers[k]));
{
    int temp = numbers[a];
    numbers[a] = numbers[b];
    numbers[b] = temp;
}
```

SelectionSort

```
static void SelectionSort(int[]! numbers) {
    for(int i = 0; i < numbers.Length; i++) {
        int bestIndex = i;
        for(int j = i; j < numbers.Length; j++) {
            if(numbers[j] < numbers[bestIndex]) {
                bestIndex = j;
            }
        }
        Exchange(numbers, i, bestIndex);
    }
}
```

SelectionSort

```
static void SelectionSort(int[]! numbers)
    modifies numbers[*]; // #1 we modify more than the default set
{
    for(int i = 0; i < numbers.Length; i++) {
        int bestIndex = i;
        for(int j = i; j < numbers.Length; j++) {
            if(numbers[j] < numbers[bestIndex]) {
                bestIndex = j;
            }
        }
        Exchange(numbers, i, bestIndex);
    }
}
```

SelectionSort

```
static void SelectionSort(int[]! numbers)
    modifies numbers[*]; // #1 we modify more than the default set
{
    for(int i = 0; i < numbers.Length; i++) {
        int bestIndex = i;
        for(int j = i; j < numbers.Length; j++)
            // #2 prove upper bound so Exchange is happy
            invariant bestIndex < numbers.Length;
        {
            if(numbers[j] < numbers[bestIndex]) {
                bestIndex = j;
            }
        }
        Exchange(numbers, i, bestIndex);
    }
}
```

SelectionSort

```
static void SelectionSort(int[]! numbers)
    modifies numbers[*]; // #1 we modify more than the default set
    // #3 introduce post condition
    ensures forall{int k in (0:numbers.Length), int l in (k:numbers.Length);
        numbers[k] <= numbers[l]};
{
    for(int i = 0; i < numbers.Length; i++) {
        int bestIndex = i;
        for(int j = i; j < numbers.Length; j++)
            // #2 prove upper bound so Exchange is happy
            invariant bestIndex < numbers.Length;
        {
            if(numbers[j] < numbers[bestIndex]) {
                bestIndex = j;
            }
        }
        Exchange(numbers, i, bestIndex);
    }
}
```

SelectionSort

```
static void SelectionSort(int[]! numbers)
    modifies numbers[*]; // #1 we modify more than the default set
    // #3 introduce post condition
    ensures forall{int k in (0:numbers.Length), int l in (k:numbers.Length);
        numbers[k] <= numbers[l]};
{
    for(int i = 0; i < numbers.Length; i++)
        // #4 use post condition as invariant to prove it
        invariant forall{int k in (0:i), int l in (k:numbers.Length);
            numbers[k] <= numbers[l]};
    {
        int bestIndex = i;
        for(int j = i; j < numbers.Length; j++)
            // #2 prove upper bound so Exchange is happy
            invariant bestIndex < numbers.Length;
        {
            if(numbers[j] < numbers[bestIndex]) {
                bestIndex = j;
            }
        }
        Exchange(numbers, i, bestIndex);
    }
}
```

SelectionSort

```
static void SelectionSort(int[]! numbers)
    modifies numbers[*]; // #1 we modify more than the default set
    // #3 introduce post condition
    ensures forall{int k in (0:numbers.Length), int l in (k:numbers.Length);
        numbers[k] <= numbers[l]};
{
    for(int i = 0; i < numbers.Length; i++)
        // #4 use post condition as invariant to prove it
        invariant forall{int k in (0:i), int l in (k:numbers.Length);
            numbers[k] <= numbers[l]};
    {
        int bestIndex = i;
        for(int j = i; j < numbers.Length; j++)
            // #2 prove upper bound so Exchange is happy
            invariant bestIndex < numbers.Length;
            // #6b help prove that everything below i isn't modified
            invariant bestIndex >= i;
        {
            if(numbers[j] < numbers[bestIndex]) {
                bestIndex = j;
            }
        }
        Exchange(numbers, i, bestIndex);
        // #6a another inner "post condition"
        // this and #6b proves #5/the outer loop invariant
        assert(forall{int k in (i:numbers.Length); numbers[i] <= numbers[k]});
        // #5 set an inner "post condition" to prove the outer loop invariant
        // this is redundant
        //assert(forall{int k in (0..i), int l in (k:numbers.Length);
        //  numbers[k] <= numbers[l]});
    }
}
```

SelectionSort

```
static void SelectionSort(int[]! numbers)
    modifies numbers[*]; // #1 we modify more than the default set
    // #3 introduce post condition
    ensures forall{int k in (0:numbers.Length), int l in (k:numbers.Length);
        numbers[k] <= numbers[l]};
{
    for(int i = 0; i < numbers.Length; i++)
        // #4 use post condition as invariant to prove it
        invariant forall{int k in (0:i), int l in (k:numbers.Length);
            numbers[k] <= numbers[l]};
    {
        int bestIndex = i;
        for(int j = i; j < numbers.Length; j++)
            // #2 prove upper bound so Exchange is happy
            invariant bestIndex < numbers.Length;
            // #6b help prove that everything below i isn't modified
            invariant bestIndex >= i;
            // #7 help prove #6b
            invariant j >= i;
        {
            if(numbers[j] < numbers[bestIndex]) {
                bestIndex = j;
            }
        }
        Exchange(numbers, i, bestIndex);
        // #6a another inner "post condition"
        // this and #6b proves #5/the outer loop invariant
        assert(forall{int k in (i:numbers.Length); numbers[i] <= numbers[k]});
        // #5 set an inner "post condition" to prove the outer loop invariant
        // this is redundant
        //assert(forall{int k in (0..i), int l in (k:numbers.Length);
        //  numbers[k] <= numbers[l]});
    }
}
```

SelectionSort

```
static void SelectionSort(int[]! numbers)
    modifies numbers[*]; // #1 we modify more than the default set
    // #3 introduce post condition
    ensures forall(int k in (0:numbers.Length), int l in (k:numbers.Length);
        numbers[k] <= numbers[l]);
{
    for(int i = 0; i < numbers.Length; i++)
        // #4 use post condition as invariant to prove it
        invariant forall(int k in (0:i), int l in (k:numbers.Length);
            numbers[k] <= numbers[l]);
    {
        int bestIndex = i;
        for(int j = i; j < numbers.Length; j++)
            // #2 prove upper bound so Exchange is happy
            invariant bestIndex < numbers.Length;
            // #6b help prove that everything below i isn't modified
            invariant bestIndex >= i;
            // #7 help prove #6b
            invariant j >= i;
            // #8 use the post condition (#6a) as invariant
            // # SelectionSort works!
            invariant forall(int k in (i:j);
                numbers[bestIndex] <= numbers[k]);
        {
            if(numbers[j] < numbers[bestIndex]) {
                bestIndex = j;
            }
        }
        Exchange(numbers, i, bestIndex);
        // #6a another inner "post condition"
        // this and #6b proves #5/the outer loop invariant
        assert(forall(int k in (i:numbers.Length); numbers[i] <= numbers[k]));
        // #5 set an inner "post condition" to prove the outer loop invariant
        // this is redundant
        //assert(forall(int k in (0..i), int l in (k:numbers.Length);
        //  numbers[k] <= numbers[l]));
    }
}
```

Partition

```
static int Partition(int[]! numbers, int left, int right) {
    //int pivotValue = numbers[right];
    int storeIndex = left;
    for(int i = left; i < right; i++) {
        if(numbers[i] <= numbers[right]) {
            Exchange(numbers, i, storeIndex);
            storeIndex++;
        }
    }
    Exchange(numbers, storeIndex, right);
    return storeIndex;
}
```

Partition

```
static int Partition(int[]! numbers, int left, int right)
    // #1 pre conditions and modifies clause
    requires 0 <= left && left < numbers.Length;
    requires 0 <= right && right < numbers.Length;
    requires left <= right;
    modifies numbers[*];
{
    //int pivotValue = numbers[right];
    int storeIndex = left;
    for(int i = left; i < right; i++)
        // #2 make the inner Exchange happy
        invariant i >= left;
        invariant storeIndex >= left;
        invariant storeIndex < i + 1;
        // #3 make the outer Exchange happy
        invariant i < numbers.Length;
    {
        if(numbers[i] <= numbers[right]) {
            Exchange(numbers, i, storeIndex);
            storeIndex++;
        }
    }
    Exchange(numbers, storeIndex, right);
    return storeIndex;
}
```

Partition

```
static int Partition(int[]! numbers, int left, int right)
    // #1 pre conditions and modifies clause
    requires 0 <= left && left < numbers.Length;
    requires 0 <= right && right < numbers.Length;
    requires left <= right;
    modifies numbers[*];
    // #4 lower post condition
    ensures forall{int k in (left..result); numbers[k] <= numbers[result]};
{
    //int pivotValue = numbers[right];
    int storeIndex = left;
    for(int i = left; i < right; i++)
        // #2 make the inner Exchange happy
        invariant i >= left;
        invariant storeIndex >= left;
        invariant storeIndex < i + 1;
        // #3 make the outer Exchange happy
        invariant i < numbers.Length;
    {
        if(numbers[i] <= numbers[right]) {
            Exchange(numbers, i, storeIndex);
            storeIndex++;
        }
    }
    Exchange(numbers, storeIndex, right);
    return storeIndex;
}
```

Partition

```
static int Partition(int[]! numbers, int left, int right)
    // #1 pre conditions and modifies clause
    requires 0 <= left && left < numbers.Length;
    requires 0 <= right && right < numbers.Length;
    requires left <= right;
    modifies numbers[*];
    // #4 lower post condition
    ensures forall{int k in (left..result); numbers[k] <= numbers[result]};
{
    //int pivotValue = numbers[right];
    int storeIndex = left;
    for(int i = left; i < right; i++)
        // #2 make the inner Exchange happy
        invariant i >= left;
        invariant storeIndex >= left;
        invariant storeIndex < i + 1;
        // #3 make the outer Exchange happy
        invariant i < numbers.Length;
        // #5a use post condition (#4) as invariant
        invariant forall{int k in (left:storeIndex); numbers[k] <= numbers[right]};
    {
        if(numbers[i] <= numbers[right]) {
            Exchange(numbers, i, storeIndex);
            storeIndex++;
        }
    }
    // #5b make sure we don't mess with the lower part
    assert(storeIndex <= right);
    Exchange(numbers, storeIndex, right);
    return storeIndex;
}
```

Partition

```
static int Partition(int[]! numbers, int left, int right)
    // #1 pre conditions and modifies clause
    requires 0 <= left && left < numbers.Length;
    requires 0 <= right && right < numbers.Length;
    requires left <= right;
    modifies numbers[*];
    // #4 lower post condition
    ensures forall{int k in (left..result); numbers[k] <= numbers[result]};
{
    //int pivotValue = numbers[right];
    int storeIndex = left;
    for(int i = left; i < right; i++)
        // #2 make the inner Exchange happy
        invariant i >= left;
        invariant storeIndex >= left;
        invariant storeIndex < i + 1;
        // #3 make the outer Exchange happy
        invariant i < numbers.Length;
        // #5a use post condition (#4) as invariant
        invariant forall{int k in (left:storeIndex); numbers[k] <= numbers[right]};
        // #6 prove #5b
        invariant storeIndex <= right;
    {
        if(numbers[i] <= numbers[right]) {
            Exchange(numbers, i, storeIndex);
            storeIndex++;
        }
    }
    // #5b make sure we don't mess with the lower part
    assert(storeIndex <= right);
    Exchange(numbers, storeIndex, right);
    return storeIndex;
}
```

Partition

```
static int Partition(int[]! numbers, int left, int right)
    // #1 pre conditions and modifies clause
    requires 0 <= left && left < numbers.Length;
    requires 0 <= right && right < numbers.Length;
    requires left <= right;
    modifies numbers[*];
    // #4 lower post condition
    ensures forall{int k in (left..result); numbers[k] <= numbers[result]};
    // #7 upper post condition
    ensures forall{int k in (result..right); numbers[result] <= numbers[k]};
{
    //int pivotValue = numbers[right];
    int storeIndex = left;
    for(int i = left; i < right; i++)
        // #2 make the inner Exchange happy
        invariant i >= left;
        invariant storeIndex >= left;
        invariant storeIndex < i + 1;
        // #3 make the outer Exchange happy
        invariant i < numbers.Length;
        // #5a use post condition (#4) as invariant
        invariant forall{int k in (left:storeIndex); numbers[k] <= numbers[right]};
        // #6 prove #5b
        invariant storeIndex <= right;
    {
        if(numbers[i] <= numbers[right]) {
            Exchange(numbers, i, storeIndex);
            storeIndex++;
        }
    }
    // #5b make sure we don't mess with the lower part
    assert(storeIndex <= right);
    Exchange(numbers, storeIndex, right);
    return storeIndex;
}
```

Partition

```
static int Partition(int[]! numbers, int left, int right)
    // #1 pre conditions and modifies clause
    requires 0 <= left && left < numbers.Length;
    requires 0 <= right && right < numbers.Length;
    requires left <= right;
    modifies numbers[*];
    // #4 lower post condition
    ensures forall{int k in (left..result); numbers[k] <= numbers[result]};
    // #7 upper post condition
    ensures forall{int k in (result..right); numbers[result] <= numbers[k]};
{
    //int pivotValue = numbers[right];
    int storeIndex = left;
    for(int i = left; i < right; i++)
        // #2 make the inner Exchange happy
        invariant i >= left;
        invariant storeIndex >= left;
        invariant storeIndex < i + 1;
        // #3 make the outer Exchange happy
        invariant i < numbers.Length;
        // #5a use post condition (#4) as invariant
        invariant forall{int k in (left:storeIndex); numbers[k] <= numbers[right]};
        // #6 prove #5b
        invariant storeIndex <= right;
    {
        if(numbers[i] <= numbers[right]) {
            Exchange(numbers, i, storeIndex);
            storeIndex++;
        }
    }
    // #8 Create a better post condition for the loop
    assert(forall{int k in (storeIndex..right); numbers[right] <= numbers[k]});
    // #5b make sure we don't mess with the lower part
    assert(storeIndex <= right);
    Exchange(numbers, storeIndex, right);
    return storeIndex;
}
```

Partition

```
static int Partition(int[]! numbers, int left, int right)
    // #1 pre conditions and modifies clause
    requires 0 <= left && left < numbers.Length;
    requires 0 <= right && right < numbers.Length;
    requires left <= right;
    modifies numbers[*];
    // #4 lower post condition
    ensures forall{int k in (left..result); numbers[k] <= numbers[result]};
    // #7 upper post condition
    ensures forall{int k in (result..right); numbers[result] <= numbers[k]};
{
    //int pivotValue = numbers[right];
    int storeIndex = left;
    for(int i = left; i < right; i++)
        // #2 make the inner Exchange happy
        invariant i >= left;
        invariant storeIndex >= left;
        invariant storeIndex < i + 1;
        // #3 make the outer Exchange happy
        invariant i < numbers.Length;
        // #5a use post condition (#4) as invariant
        invariant forall{int k in (left:storeIndex); numbers[k] <= numbers[right]};
        // #6 prove #5b
        invariant storeIndex <= right;
        // #9 use the "post condition" (#8) as invariant
        invariant forall{int k in (storeIndex:i); numbers[right] <= numbers[k]};
    {
        if(numbers[i] <= numbers[right]) {
            Exchange(numbers, i, storeIndex);
            storeIndex++;
        }
    }
    // #8 Create a better post condition for the loop
    assert(forall{int k in (storeIndex..right); numbers[right] <= numbers[k]});
    // #5b make sure we don't mess with the lower part
    assert(storeIndex <= right);
    Exchange(numbers, storeIndex, right);
    return storeIndex;
}
```

Partition

```
static int Partition(int[]! numbers, int left, int right)
    // #1 pre conditions and modifies clause
    requires 0 <= left && left < numbers.Length;
    requires 0 <= right && right < numbers.Length;
    requires left <= right;
    modifies numbers[*];
    // #4 lower post condition
    ensures forall{int k in (left..result); numbers[k] <= numbers[result]};
    // #7 upper post condition
    ensures forall{int k in (result..right); numbers[result] <= numbers[k]};
    // #10 extra post conditions to help QuickSort
    ensures 0 <= result && result < numbers.Length;
    //ensures forall{int k in (0:left); numbers[k] == old(numbers[k])};
    //ensures forall{int k in (right+1:numbers.Length); numbers[k] == old(numbers[k])};
{
    //int pivotValue = numbers[right];
    int storeIndex = left;
    for(int i = left; i < right; i++)
        // #2 make the inner Exchange happy
        invariant i >= left;
        invariant storeIndex >= left;
        invariant storeIndex < i + 1;
        // #3 make the outer Exchange happy
        invariant i < numbers.Length;
        // #5a use post condition (#4) as invariant
        invariant forall{int k in (left:storeIndex); numbers[k] <= numbers[right]};
        // #6 prove #5b
        invariant storeIndex <= right;
        // #9 use the "post condition" (#8) as invariant
        invariant forall{int k in (storeIndex:i); numbers[right] <= numbers[k]};
    {
        if(numbers[i] <= numbers[right]) {
            Exchange(numbers, i, storeIndex);
            storeIndex++;
        }
    }
    // #8 Create a better post condition for the loop
    assert(forall{int k in (storeIndex..right); numbers[right] <= numbers[k]});
    // #5b make sure we don't mess with the lower part
    assert(storeIndex <= right);
    Exchange(numbers, storeIndex, right);
    return storeIndex;
}
```

QuickSort

```
static void QuickSort(int[]! numbers, int left, int right)
    // #1 pre conditions and modifies clause
    requires left >= 0;
    requires right < numbers.Length;
    modifies numbers[*];
    // #2 lower post condition
    ensures forall{int k in (left..right), int l in (left..k);
        numbers[l] <= numbers[k]};
    // #3 upper post condition
    ensures forall{int k in (left..right), int l in (k..right);
        numbers[k] <= numbers[l]};
    // #4 restrictions on modify
    ensures forall{int k in (0:left); numbers[k] == old(numbers[k])};
    ensures forall{int k in (right+1:numbers.Length); numbers[k] == old(numbers[k])};
{
    if(right <= left) return;

    int pivot = Partition(numbers, left, right);
    //assert(forall{int k in (left:pivot); numbers[k] <= numbers[pivot]});
    QuickSort(numbers, left, pivot - 1);
    //assert(forall{int k in (left:pivot); numbers[k] <= numbers[pivot]});
    //assert(forall{int k in (left..pivot), int l in (k..pivot);
    //  numbers[k] <= numbers[l]});
    QuickSort(numbers, pivot + 1, right);
    //assert(forall{int k in (left..right), int l in (k..right);
    //  numbers[k] <= numbers[l]});
}
```

Ownership

```
        List<int> l = new List<int>();
        l.Add(42);
        Bin b = new Bin(l);
        l.Add(5);
        The call to System.Collections.Generic.List<int>.Add(int item) requires target object to be peer consistent
    }
}

public class Bin {
    [Rep] List<int>! elements;

    public Bin([Captured] List<int>! _elements) {
        elements = _elements;
    }
}
```

Ownership

```
public class Bill {
    [Rep] private int[]! items;
    [Rep] private string[]! texts;
    private int nextFree;
    invariant items.Length == texts.Length;
    invariant 0 <= nextFree && nextFree < items.Length;
    invariant forall{int i in (0:nextFree); items[i] > 0};
    invariant forall{int i in (0:nextFree); texts[i] != null};
    public int total;
    invariant total == sum{int i in (0:nextFree); items[i]};

    public Bill() {
        items = new int[64];
        texts = new string[64];
        total = 0;
    }

    public void Add(string! text, int amount)
        requires amount > 0;
    {
        if(nextFree == items.Length - 1) {
            string[]! tempA = new string[texts.Length * 2];
            int[]! tempB = new int[items.Length * 2];
            expose(this) {
                texts.CopyTo(tempA, 0);
                items.CopyTo(tempB, 0);
                texts = tempA;
                items = tempB;
            }
        }
        expose(this) {
            texts[nextFree] = text;
            items[nextFree] = amount;
            nextFree++;
            total += amount;
        }
    }
```

Ownership

```
public class Car {
    [Peer] bool[] locks;
    [Rep] bool[] moreLocks;

    [Captured]
    public Car(bool[]! initialLocks) {
        Owner.AssignSame(this, initialLocks);
        locks = initialLocks;
    }

    public Car(int a, [Captured] bool[]! initialLocks) {
        moreLocks = initialLocks;
    }
}
```