Iterator Proof Rules for C# V2.0

Bart Jacobs,
Frank Piessens,
KU Leuven

Wolfram Schulte
Microsoft Research

Outline

- Iterators in C#
- How to specify and verify iterators and foreach loops?
- How to prevent interference between iterators and foreach loops?

The Iterator pattern in C# 2.0

```
public interface IEnumerator<T> {
    T Current { get; }
    bool MoveNext();
}

public interface IEnumerable<T> {
    IEnumerator<T> GetEnumerator();
}
```

Foreach Loops

foreach (*T* **x** *in* **C**) S

is implemented as

```
IEnumerable<T> c = C;
IEnumerator<T> e = c.GetEnumerator();
while (e.MoveNext())
{ T x = e.Current; S }
```

C# 2.0 Iterator Methods

```
IEnumerable<int> FromTo(int a, int b) {
    for (int x = a; x < b; x++)
        yield return x;
}
```

is implemented as

```
IEnumerable<int> FromTo(int a, int b)
{ return new FromTo_Enumerable(a, b); }
```

Compiler-generated class

C# 2.0 Iterator Methods

```
class FromTo_Enumerator :  IEnumerator<int> {
   int a; int b; int pc; int x; int current;
   public FromTo_Enumerator(int _a, int _b) { a = _a;
    b = _b; }
   public int Current { get { return current; } }
   public bool MoveNext() {
      switch (pc) {
         case 0: x = a; goto case 1;
         case 1: if (!(x < b)) goto case 4;
         case 2: current = x; pc = 3; return true;
         case 3: x++; goto case 1;
         case 4: pc = 4; return false;
      }}}
```

How to specify and verify iterators?

```
static IEnumerable<int> FromTo(int a, int b)
    requires a <= b;
    invariant forall{int i in (0; b – a); values[i] == a + i};
    invariant values.Count <= b – a;
    ensures values.Count == b – a;
{
    for (int x = a; x < b; x++)
        invariant values.Count == x – a;
    { yield return x; }
}
```

Enumeration invariant must be proved at start of iterator method…

… and after each yield return statement.

Ensures clause must be proved at end of method (and at yield break statements)

How to specify and verify foreach loops?

```
int sum = 0;
Seq<int> values = new Seq<int>();
 while (*)
    invariant sum == Math.Sum(values);
    free invariant forall{int i in (0:values.Count); values[i]==1+i};
    free invariant values.Count <= 3 – 1;
{
    int x; havoc x; values.Add(x);
    assume forall{int i in (0:values.Count); values[i]==1+i};
    assume values.Count <= 3 – 1;
    sum += x;
}
assume values.Count == 3 – 1;
assert sum == 6;
```

Interference

```
List<int> xs = new List<int>();
xs.Add(1); xs.Add(2);
    xs.Add(3);
int sum = 0;
foreach (int x in xs)
{ sum += x; xs.Remove(0); }
//assert sum == 6;
```

```
class List<T> : IEnumerable<T> {
  IEnumerator<T>
   GetEnumerator() {
    int n = Count;
    for (int i = 0; i < n; i++)
      yield return this[i];
  }}
```

ArgumentOutOfRangeException !

Parties execute in an interleaved fashion

But we wish to verify them as if they executed in isolation

Proposed solution:
Prevent either party from seeing the other party's effects

Proposed Solutio

Error: unsatisfied
requires this.readCount == 0;

Enforced using an extension of the Boogie methodology

```
List<int> xs = new List<int>();
xs.Add(1); xs.Add(2);
    xs.Add(3);
int sum = 0;
foreach (int x in xs)
{ sum += x; xs.Remove(0); }
//assert sum == 6;
```

```
class List<T> : IEnumerable<T> {
  ...
  IEnumerator<T>
   GetEnumerator()
    reads this; {
    int n = Count;
    for (int i = 0; i < n; i++)
      yield return this[i];
  }}
```

reads clause declares the set of pre-existing objects the iterator method wishes to read

The iterator method may not read or write any other pre-existing objects

And the foreach loop body may not write the objects in the reads clause

The Boogie methodology

- Enforces object invariants
- Uses a dynamic ownership system
- Each object gets two extra fields:
  - **bool inv;**
  - **bool writable;**
- **o.f := x;** requires **o.writable && !o.inv**
- ***unpack o;*** requires **o.writable && o.inv**
  - Sets **o.inv := *false;***
  - Makes owned objects writable
- ***pack o;*** reverses the effect of ***unpack o;***

Adding read-only objects to the Boogie methodology

- Each object gets three special fields:
  - `bool inv;`
  - `bool writable;`
  - `int readCount; // never negative`
- `o.f = x;` requires

  `o.writable && o.readCount == 0 && !o.inv`
- `x = o.f;` requires

  `o.writable || 0 < o.readCount`

Read-o

```
partial class List<T> {
   [Owned] T[] elems;
   T this[int index] {
      get
         requires
            inv &&
            (writable ||
            0 < readCount);
      { read (this)
         { return elems[index]; }
      }
   }}
```

```
read (o) S

   means

assert o.writable || 0 < o.readCount;
assert o.inv;
o.readCount++;
foreach ([Owned] field f of o)
   o.f.readCount++;
S
foreach ([Owned] field f of o)
   o.f.readCount–;
o.readCount–;
```

```
      for (int i = 0; i < n; i++)
         yield return this[i];
   }}
```

How is this call verified?

Thank you