Resource Usage Contracts for .NET

Jonathan Tapicer, Diego Garbervetsky and Martin Rouaux - {jtapicer, diegog, mrouaux}@dc.uba.ar
Departamento de Computación, FCEyN, Universidad de Buenos Aires, Argentina

What?
- An extension of Code Contracts to support resource usage specifications in .NET programs.
- Tailored for specifying dynamic memory consumption, a resource that is not only allocated but it is also reclaimed during program execution.

How?
- We introduce new set of annotations enabling specification of both memory consumption and lifetime properties in a modular fashion.
- These annotations allow us to compute an upper bound of the real memory allocated using a compositional analysis.

What?
- An extension of Code Contracts to support resource usage specifications in .NET programs.
- Tailored for specifying dynamic memory consumption, a resource that is not only allocated but it is also reclaimed during program execution.

How?
- We introduce new set of annotations enabling specification of both memory consumption and lifetime properties in a modular fashion.
- These annotations allow us to compute an upper bound of the real memory allocated using a compositional analysis.

How are the annotations checked?

- Static Verifier (Clousot)
- Visual Studio
- Assembly instrumentation
- Points-to analysis
- Arithmetic tool (Barvinok)
- Instrumentation + Verification

Future work
- Automatic inference of quantitative and lifetime annotations in order to mitigate annotation burden.
- Upgrade the language in order to enable finer grained lifetime specs. while maintaining information hiding.
- Use SMT solvers (e.g Z3) and integrate them with tools capable of dealing with non-linear expressions.

Sample code:

```csharp
public class IntLinkedlist { 
    private Node Head;

    public void PushFront(Node node) { 
        Contract.Memory.Tmp<Logger>({1, );
        Contract.Memory.DestTmp();
        Logger logger = new Logger();
        node.Next = this.Head;
        this.Head = node;
        logger.Log("PushFront done");
    }

    public void Fill(int n) {
        Contract.Requires(n > 0);
        Contract.Memory.Rsd<Logger>({Contract.Memory.This, n - 1};
        Contract.Memory.Tmp<Logger>({1); 
        for (int i = 1; i <= n; i++) {
            Contract.Memory.DestRsd(Contract.Memory.This);
            Node node = new Node(1);
            this.PushFront(node);
        }
    }

    public void Clear() {
        this.Head = null;
        Contract.Memory.Rsd<Logger>({Contract.Memory.This, 1};
        Contract.Memory.DestRsd(Contract.Memory.This);
        Logger logger = new Logger();
        logger.Log("Clear done");
    }
}
```

- Contract.Memory.Tmp<T>(n); defines a temporary memory of type T of at most n
- Contract.Memory.DestTmp(); indicates that the next object created is temporary
- Contract.Memory.Tmp<T>(n); defines a temporary memory of type T of at most n
- Contract.Memory.DestRsd(t); indicates that the next object created is residual and tagged as t
- Error: the method requires n residual Node objects
- Error: the object logger is temporary but marked as residual
- Contract.Memory.Rsd<T>(t, n); defines a residual memory of type T tagged as t of at most n

Static Verifier (Clousot)
Visual Studio
Assembly instrumentation + Verification
Points-to analysis
Arithmetic tool (Barvinok)
Instrumentation + Verification

The code is instrumented by inserting counters and standard Code Contracts assertions in terms of those counters. Lifetime annotations correctness is verified using the point-to-analysis.

http://lafhis.dc.uba.ar/resourcecontracts