Numerical Computing with the CLR

Jeffrey Sax

Extreme Optimization

What would it take to make .NET a great platform for numerical computing?

What is “numerical computing”?

- Interactive computations
- Guided interactive computations
- Analysis component of a larger application
- Batch processing

What makes a great platform?

- Make it easy to be correct
  - Notation close to familiar mathematical notation
  - Danger: It looks like normal math, but it isn't
- Make it easy to be efficient
  - Scripting languages offload the hard work to libraries
  - So, we need fast libraries
- Minimize impact on general-purpose computing

Floating Point Context

- Since 1984: IEEE 754 – IEC 60559 standard
- Number formats: single, double, extended
- Special values: NaN, +/-inf, -0
- Exception flags
  - Example: complex division
- Rounding modes
  - Example: sensitivity analysis

Complex Division

- Classic formulas

```
d = b.re*b.re + b.im*b.im;
r = (a.re*b.re+a.im*b.im) / d;
i = (a.im*b.re-a.re*b.im) / d;
```

  Work most of the time, but risk of over/underflow

- IronPython does better but
  - Has extra division (slower than necessary)
  - May give inexact results:

```
(27-21j) / (9-7j) - 3 = -4.4e-16
```

Complex Division

With floating-point exception flags:

```
using (FloatingPointContext fp =
  FloatingPointContext.Create())
{
  fp.ClearExceptionFlag(
    FloatingPointExceptionFlag.Underflow);
  double d = z2.re*z2.re + z2.im*z2.im;
  double resultRe = z1.re * z2.re + z1.im * z2.im;
  double resultIm = z1.im * z2.re - z1.re * z2.im;
  if (fp.IsExceptionFlagRaised(
    FloatingPointExceptionFlag.Underflow) {
    // Code for the special cases.
  }
  return new Complex(resultRe / d, resultIm / d);
}
```

Imprecise Faults

- CLR strictly adheres to declared behavior

```
double[] x = new double[1000];
for (int i = 0; i <= 1000; i++)
    x[i] = i;
```

- Relaxing this requirement allows more optimizations:
  - Hoisting runtime checks out of loops
  - Reordering loops
  - Automatic parallellization
- See Annex F to ECMA-335 (CLI Spec)

Lightweight Exceptions

- Don’t create full exceptions if a CLR generated exception is handled inside the same method

```
try {
  checked {
     result = op1 + op2;
  }
}
catch (OverflowException)
{ ... }
```

Other Low Level Features

- Inlining of methods with value type parameters
- Ability to specify data alignment of arrays
  - SIMD instructions require 16 byte alignment
  - Difference: up to 3x
- JIT support for SIMD instructions
- Optimization hints
  - Use an attribute to tell the JIT compiler how much to optimize
- OpenMP support in ParallelFX

Numerical Types

- Currently many different, incompatible implementations
- Complex numbers
- BigInteger, BigRational, BigFloat, etc.
- Tuples, Slices

1D, 2D, N-D Array Abstraction

- Methods for selecting sub-arrays
- Index can be int, IEnumerable<int> or IList<int>
  - Slices: 1:2:9 = {1, 3, 5, 7, 9}
  - Arrays of indexes
  - Boolean arrays
- Allow different internal representations

Pluggable algorithms

- Many solutions exist for the same problem
  - Example: random numbers, optimization
- Appropriateness of algorithm depends on problem characteristics
- Must allow room for different implementations

Pluggable algorithms - examples

- Linear algebra
  - Large sizes: managed code is an order of magnitude slower than native code.
  - Small sizes: managed/unmanaged overhead dominates.
  - New technologies: GPU based computing
- Random number generation
  - Defects may show up as systematic errors in large simulations
  - Trade-off of quality vs. speed

Compound Assignment

- a @= b
- CLI spec: static methods
- C++/CLI: instance methods
- C#, VB.NET: rewrite as a = a @ b
- Problems
  - Performance
  - Sometimes different meaning
    - Example: add a vector to a row in a matrix

Generic arithmetic

- Allow operations to be performed using any numerical type.
- Ideal outcome: generic algorithms run at close to 100% efficiency for primitive types.
- Algebraic structures: group, ring, Euclidean ring (integers, polynomials), field.

Generic Arithmetic: approaches

- Use class to encapsulate operations
  - F# uses this technique (INumeric interfaces)
- Compiled lambda expressions
  - Good performance, but you lose type safety
- Build on the DLR?
  - Keep entire algorithm as an expression tree++

Generic Arithmetic : Operator Class

- Generic interface defines operators on element type

```
interface IGroupArithmetic<T> {
   T Add(T a, T b);
   T Subtract(T a, T b);
   T Negate(T a);
   T Zero { get; }
}
```

- Generic algorithms take two parameters: element type and associated operator class

```
public class Adder<T,U>
  where U : IGroupArithmetic<T> {

  private static U ops = ...;
  public static T Sum(T[] array) {
     T result = ops.Zero;
     foreach (T item in array)
         result = ops.Add(result, item);
     return result;
  }
}
```

Generic Arithmetic : Challenges

- Trade-off between performance and Usability
  - Matrix<T**,N**> **where N : INumeric<T>**
  - Operator class accessed through interface

```
IGroupArithmetic<T> ops =
  Operators.Lookup(typeof(T));
```

- Optimizations for primitive types
  - Also calling into native libraries
- Operations may depend on algebraic structure of type parameter(s)
  - Division for Complex<int> is not well defined

Associated Types

- Many relationships between types aren't captured in the type system
- Making these relationships explicit opens many possibilities
  - Generic arithmetic
  - "Specializations"
  - Designers, visualizers
- Declare named association through attribute

```
[TypeAssociation("Arithmetic", typeof(BigIntOps))
class BigInteger { ... }
```

- Can be used in generic type definition, including generic type constraints

Associated Types : Generic Arithmetic

```
public class Adder<T>
  where T.Arithmetic : IGroupArithmetic<T>
{
  private static T.Arithmetic ops =
    new T.Arithmetic();
  public static T Sum(T[] array) {
     T result = ops.Zero;
     foreach (T item in array)
         result = ops.Add(result, item);
     return result;
  }
}
```

Associated types: Impact

- Requires compiler support to have true generic arithmetic
  - If type has associated Arithmetic type, then translate operator expressions to calls to corresponding methods on the Arithmetic type
- If associated types are used in the type definition, then each instantiation that uses a reference type needs its own copy of the code

Specialization

- Implementation of a generic type specific for a set of type parameters
- Why specialize?
  - Calling into native libraries
  - Elementary operations may have different cost
  - Better extensibility
- Can be done using associated types

Specialization

```
class TypeDefinition<T> {
  [TypeAssociation(typeof(object), "Specialization")]
  public class Specialization {
    virtual public void Specialized (T instance, int a)
    { ... }
  }

  static T.Specialization specialization;
  public void Specialized(int a) {
    specialization.Specialized(this, a); }

[TypeAssociation(typeof(double), “Specialization“)]
sealed class DoubleSpecialization :
  TypeDefinition.Specialization<double>
{
  override public void Specialized(double instance, int a)
  { ... }
}
```

Type Constraint Extensions

- What to do when capabilities of generic type depend on the capabilities of the parameter type?
- Allow type constraints to propagate through the type system
- Allow type constraints on methods

```
interface ICollection<T> {
  void Sort() where T : IComparable<T>
  }
```

- Partial prototype @ MSR Cambridge
  - No full constraint inheritance
  - Implemented on CLR 2.0

Symbolic Manipulation

- Examples:
  - Automatic differentiation
  - Solving equations
  - Smart numerical integration
  - Simplification and optimization
  - Using DSL's inside another language

Symbolic Manipulation

- Traditionally the domain of packages like Mathematica, Maple
- Expression trees provide the core infrastructure

Symbolic manipulation: Rules

- Term rewriting system based on expression trees.
- Rules can be stored as resources in compiled assembly.
  - Automatically loaded when assembly is referenced.
- Example:

```
rule derivative(J1(x),x) = J0(x) - J1(x) / x;
```

Units and Measures

- Common approach: Strongly typed quantity types.
  - Explosion of types.
  - Heavy load on type system.
  - Clumsy or unnatural syntax.
  - Usually a performance penalty.
  - Units have a group structure that the type system can’t handle.

Units and Measures : Annotations

```
Dim speed As Double In km/h
```

- Units in public API are declared with Attributes
- Unit of each variable is determined statically.
- Consistency enforced by the compiler.
- Conversions performed as needed.

```
oneHalf = Math.Sin(30 degrees);
```

- Unit conversions : `speed in mph;`
- User-defined units:

```
unit 1 AU = 149597870 km;
```

- Osprey (Jiang & Su), F#! (Andrew Kennedy @ MSR Cambridge)

First Class Functions

- Class based approach gives more options
- Parameterized functions

```
function Logistic(x; A0, A1, x0, p) =
        A1 + (A0 - A1) / (1 + (x/x0)^p);
f = Logistic(0,1,10^5,3);
f(0) = A0;
f.A1 = 10;
df = gradient(f, [A0,A1,x0,p]);
```

Precision

- 32/64 bit integers and single/double precision aren’t always good enough.
- Make it easy to select required precision.
- Provide exact (bigint, ubigint, bigrational) and approximate (bigfloat, bigdecimal) types
- Precision operator : `

```
Console.WriteLine(pi`30)
  3.14159265358979323846264338328
```

- Precision directed evaluation

```
tan(10^100)`10 = ???
```

Conclusion

What would make .NET into a truly great platform for numerical computing?

- Language teams get together on standard numerical types
- Some JIT/CLR improvements
- Extend generics with associated types
- Compiler support for extended type constraints and generic arithmetic
- .NET languages for numerical computing

Questions?

E-mail: jeffrey@extremeoptimization.com

Web: www.extremeoptimization.com