Microsoft

Generic Math in .NET

Contractual Static Interface Members in C#

Aaron Bockover
Miguel de Icaza

Miguel de Icaza
Partydonk, 2020
Oil on Canvas

Background

- How could .NET participate in the world of Deep Learning?
  - Not just as a consumer (inferencing), but as means to develop AI models (training)
  - .NET can participate, just not very well today, and the solutions are clunky
  - We described our findings in https://github.com/partydonk/partydonk

- Today we will cover a foundational of the piece of the puzzle…

Contractual Static Interface Members

Generic Math: the primary motivator

- Operators `+ - / *` etc.
- Additive identity property **`Zero`**
- Common mathematical functions

```
T Average<T>(params T[] values)
    where T : INumeric<T>
{
    var sum = T.Zero;
    foreach (var value in values)
        sum += value;
    return sum / values.Length;
}
```

```
T Sigmoid<T>(T x)
    where T: IReal<T>
    => 1 / (1 + T.Exp(x))
```

Status

- We have a proof of concept implemented in Roslyn + Mono
  - https://github.com/partydonk/roslyn/tree/dev/abock/asim/asim-playground
  - https://github.com/Partydonk/partydonk/issues/1

- Very modest changes are required to the language and runtime
  - They flow directly from the existing language and runtime design.

- Additional work in corlib required to fully realize numerics dream

Context

- Unbeknownst to us, Carol Eidt and Dave Detlefs described this work in 2010 in the document "Increasing the power of generics through static constraints"
  - They landed on abstract static for contract definition, and constrained calls in CIL as well
  - So, a big thanks to Carol and Dave!
- Swift baked this into the language
  - Then benefits from Chris Lattner moving to Google to advance numerical computing uses
  - Swift type hierarchy and capabilities while not perfectly named, has all the right elements
- Swift has `self` + `associatedtype` that makes this easy
  - We thought we would need this – but
  - Mads Torgersen offered a great way of expressing it…

Expressing Generic Operations in .NET

```
interface IAdditiveArithmetic<TSelf> : IEquatable<TSelf>, IComparable<TSelf>
    where TSelf : IAdditiveArithmetic<TSelf>
{
    abstract static TSelf Zero { get; }
    abstract static TSelf operator +(TSelf left, TSelf right);
    abstract static TSelf operator -(TSelf left, TSelf right);
}

interface IExpressibleByIntegerLiteral<TSelf>
    where TSelf : IExpressibleByIntegerLiteral<TSelf>
{
    abstract static implicit operator TSelf(int value);
}

interface INumeric<TSelf> : IAdditiveArithmetic<TSelf>,
    IExpressibleByIntegerLiteral<TSelf>
    where TSelf : INumeric<TSelf>
{
    TSelf Magnitude { get; }
    abstract static TSelf operator *(TSelf left, TSelf right);
}
```

Modeling the behavior of `self` in Swift, this pattern is pervasive in our .NET Numerics prototype.

How does it work?

- Runtime
  - Methods in IL flagged as `abstract static` are now valid (currently rejected)
  - In Mono, everything else just works; we suspect that will be the case for CoreCLR as well

- Roslyn
  - Language extensions to declare conformance
  - We will describe that next

Static Interface Member Refresher

- New to C# in 8.0 and must have a body. Introduced along with default implementations in interfaces:

```
interface IContract
{
    static void M()
        => WriteLine("Hello from IContract.M");
}
```

- Invocable against the interface type itself – does not participate in the interface contract:

```
IContract.M();
```

Static Interface Member Refresher

- However, this is not new in CIL:

```
.method public static void M()
```

- Likewise, invocable against the interface type itself:

```
call void IContract::M()
```

Contractual Static Interface Members (CSIM)

- `static` members participate as part of an interface's contract.
- We need a keyword to disambiguate "helper" static members now that the C# 8.0 ship has sailed.
- `abstract` is natural and easy to reuse in the compiler.
  - No new lexer work (we explored a `self` keyword, etc).
- We will want `virtual` behavior too…

Contractual Static Interface Members (CSIM)

- `static` members participate as part of an interface's contract:

```
interface IContract
{
    abstract static void M();
}
```

```
class A : IContract
{
    public static void M()
        => WriteLine("Hello from A.M");
}
```

- Invocable against constrained generic type arguments:

```
void InvokeM<T>()
    where T : IContract
    => T.M();
```

```
InvokeM<A>();
```

Contractual Static Interface Members (CSIM)

- Reuse CIL metadata; `constrained. call` to invoke:

```
.class interface IContract
{
    .method public abstract virtual static void M() { }
}

.class A implements IContract
{
    .method public virtual final static void M() { }
}

.method static void InvokeM<(IContract) T>()
{
    constrained. !!T
    call void IContract::M()
}
```

Runtime Support Needed

Currently prototyped against `dotnet/runtime` Mono.

CSIM Roslyn Prototype

- `abstract` on `static` interface members
- Allow for `static` member accesses against generic type arguments
- Emit `constrained. call` against generic type
- Special case support for operators – lots of existing diagnostics around operators are relaxed in the CSIM context
- Diagnostic to enforce impl of `abstract static` members
- Feature flag needed to ensure target runtime support

- https://github.com/partydonk/roslyn/tree/dev/abock/asim/asim-playground
- https://github.com/Partydonk/partydonk/issues/1

CSIM Roslyn Prototype

- Lots of small changes sprinkled across Roslyn

```
private static bool IsInterfaceMemberImplementation(Symbol candidateMember, Symbol interfaceMember, …)
{
-   if (candidateMember.DeclaredAccessibility != Accessibility.Public || candidateMember.IsStatic)
+   if (candidateMember.DeclaredAccessibility != Accessibility.Public)
```

- Most of these checks relax restrictions that yield “you cannot do this” diagnostics. This relaxation needs to be behind the feature flag:

  - `System.Runtime.CompilerServices.RuntimeFeature.ContractualStaticInterfaceMembers`

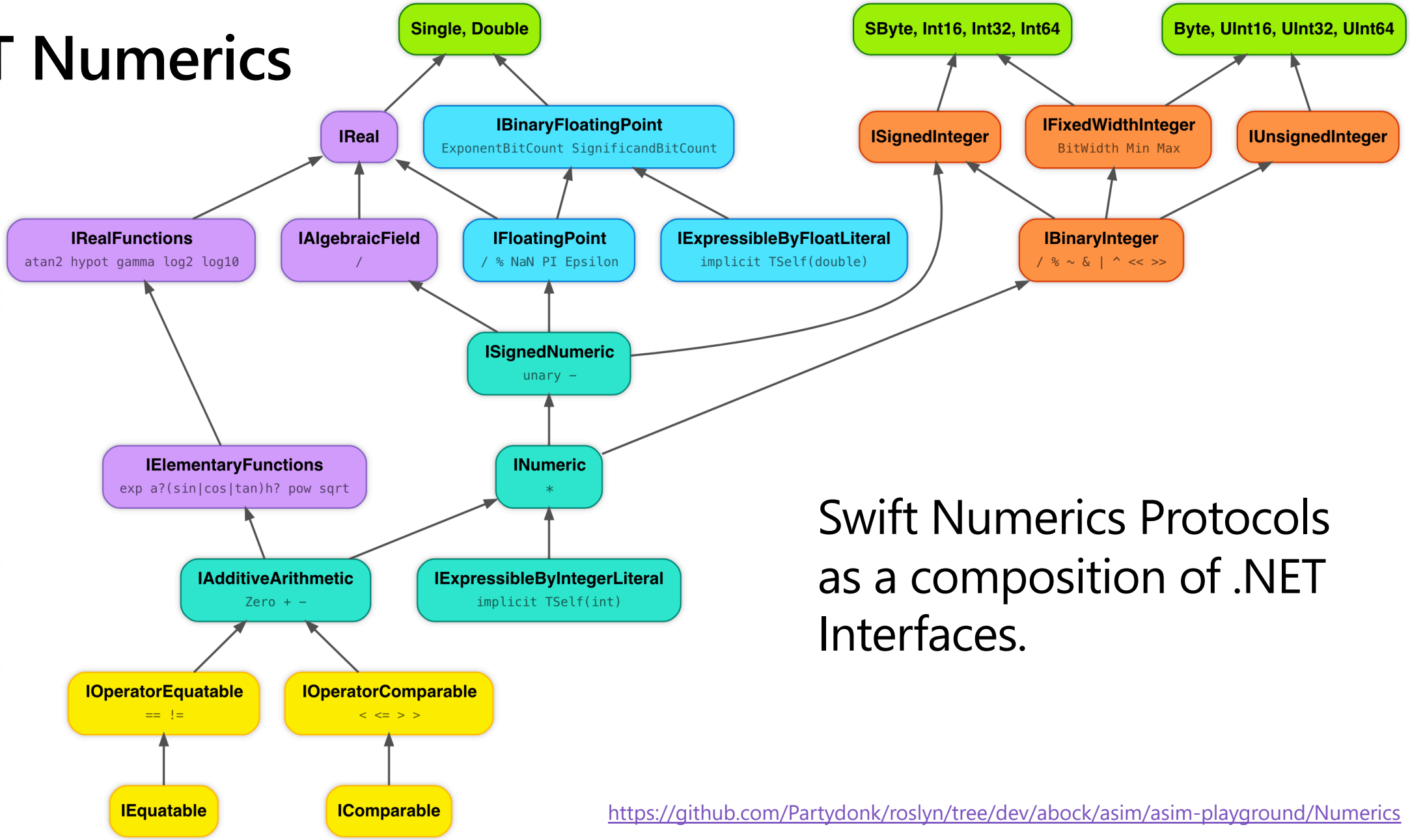
Microsoft

Next Steps

What follows CSIM? .NET Numerics

- With Contractual Static Interface Members in place, we can then expand .NET to leverage the capability
  - Introduce new System.Numerics interfaces to represent various numerical capabilities.
  - Augment existing numeric types to implement these interfaces.
  - Take heavy inspiration from Swift and Swift Numerics!
    - https://swift.org/blog/numerics/
    - https://github.com/apple/swift-numerics

.NET Numerics



Swift Numerics Protocols as a composition of .NET Interfaces.

https://github.com/Partydonk/roslyn/tree/dev/abock/asim/asim-playground/Numerics

For Further Consideration

Generic `SinSquared`:

```
T SinSquared(T x) where T : IReal<T> {
    return Math.Sin(x) * Math.Sin(x); // Error – cannot express today
}
```

We need to surface the math operations on the types directly. Options include:

```
T SinSquared(T x) where T : IReal<T> {
    return T.Sin(x) * T.Sin(x); // as a static method
    return x.Sin() * x.Sin();   // as an instance method
}
```

Or both could be valid if `Sin` could be an extension method…

CSIM: Virtual?

- We have only prototyped `abstract`, but `virtual` is equally valid
- Flows naturally from default interface member support
- Simplifies contract implementation. Ex:
  - Binary subtraction op could be virtual which invokes unary negation and binary addition ops
  - Extend `IEquatable` to light up operator support

```
public interface IEquatable<T>
{
    bool Equals(T other); // Today

    // Future: implement IEquatable<T>.Equals, and
    // receive free support for equality operators.
    virtual static bool operator ==(T l, T r) => l.Equals(r);
    virtual static bool operator !=(T l, T r) => !l.Equals(r);
}
```

CSIM: Override?

- Should we require `override` on the implementation?
  - Technically unnecessary since they're all static members.
  - However, syntax *may* be desired for ceremony/consistency with abstract class members.

```
interface IContract
{
    abstract static void M();
}
```

```
class A : IContract
{
    public static override void M()
        => WriteLine("Hello from A.M");
}
```