# Chapter 1: Down the Rabbit Hole

I’m trying to figure out why I’m getting 12% worse performance from F# compared to C# on a simple benchmark. Simple, but highly meaningful – the methods at the heart of this can get called millions of times.

First, the benchmark data:

| Method | size | Mean | Error | StdDev | Ratio | RatioSD | Allocated | Alloc Ratio |

|---------------- |-------- |---------:|----------:|----------:|------:|--------:|----------:|------------:|

| CSharpIsNumeric | 1000000 | 1.944 ms | 0.0333 ms | 0.0295 ms | 1.00 | 0.00 | 1 B | 1.00 |

| FSharpIsNumeric | 1000000 | 2.187 ms | 0.0210 ms | 0.0196 ms | 1.12 | 0.02 | 2 B | 2.00 |

Allocations are clearly not an issue. I’m not going to quibble about two nibbles.

The benchmarks themselves are fairly simple. We are comparing the performances of two methods written in F# vs the same two methods written in C# (the originals, from ClojureCLR).

[<Benchmark(Baseline=true)>]

member this.CSharpIsNumeric() =

let mutable cnt : int = 0

for i in 0 .. this.size - 1 do

if TypeDispatch.CSharp.TypeDispatch.IsNumeric (this.items.[i])

then cnt <- cnt + 1

cnt

[<Benchmark>]

member this.FSharpIsNumeric() =

let mutable cnt : int = 0

for i in 0 .. this.size - 1 do

if FsharpTypeDispatch.TypeDispatch.IsNumeric (this.items.[i])

then cnt <- cnt + 1

cnt

items is an array of 1,000,000 integers. IsNumeric in either case will always return true.

Here are IsNumeric and IsNumericType in each language

static member IsNumeric(o: obj) =

match o with

| null -> false

| \_ -> TypeDispatch.IsNumericType(o.GetType())

static member public IsNumericType(t: Type) =

match Type.GetTypeCode(t) with

| TypeCode.SByte

| TypeCode.Byte

| TypeCode.Int16

| TypeCode.Int32

| TypeCode.Int64

| TypeCode.Double

| TypeCode.Single

| TypeCode.UInt16

| TypeCode.UInt32

| TypeCode.UInt64 -> true

| \_ -> System.Type.op\_Equality(t, typeof<BigInteger>)

public static bool IsNumeric(object o)

{

return o != null && IsNumericType(o.GetType());

}

public static bool IsNumericType(Type type)

{

switch (Type.GetTypeCode(type))

{

case TypeCode.SByte:

case TypeCode.Byte:

case TypeCode.Int16:

case TypeCode.Int32:

case TypeCode.Int64:

case TypeCode.Double:

case TypeCode.Single:

case TypeCode.UInt16:

case TypeCode.UInt32:

case TypeCode.UInt64:

return true;

default:

return type == typeof(BigInteger);

The explicit call to op\_Equality was an attempt to avoid some generic comparison hackery that F# performs if I do straight equality test there. Because we are passing integers in all calls, we never hit that code.

The F# code is as close as I could make it to the C# code. In fact, running the IL through ILSpy to disassemble, we have near parity. Decompiling to C#, we see just a little bit of clause rearrangement in IsNumeric. But IsNumericType sees a numeric calculation in the C# and a switch statement in the F#:

public static bool IsNumeric(object o)  
{  
    if (o == null)  
    {  
        return false;  
    }  
    return IsNumericType(o.GetType());

public static bool IsNumeric(object o)  
{  
    return o != null && IsNumericType(o.GetType());  
}

We see a difference here. Does it matter?

// F#

public static bool IsNumericType(Type t)  
{  
    switch (Type.GetTypeCode(t))  
    {  
    case TypeCode.SByte:  
    case TypeCode.Byte:  
    case TypeCode.Int16:  
    case TypeCode.UInt16:  
    case TypeCode.Int32:  
    case TypeCode.UInt32:  
    case TypeCode.Int64:  
    case TypeCode.UInt64:  
    case TypeCode.Single:  
    case TypeCode.Double:  
        return true;  
    default:  
        return t == typeof(BigInteger);  
    }

}

// C#

public static bool IsNumericType(Type type)  
{  
    TypeCode typeCode = Type.GetTypeCode(type);  
    TypeCode typeCode2 = typeCode;  
    if ((uint)(typeCode2 - 5) <= 9u)  
    {  
        return true;  
    }  
    return type == typeof(BigInteger);  
}

Certainly there is a difference in the IL. Using Sharplab.io to generate the IL.

\_+Tests.IsNumeric(System.Object)

L0000: sub rsp, 0x28

L0004: test rcx, rcx

L0007: jne short L0010

L0009: xor eax, eax

L000b: add rsp, 0x28

L000f: ret

L0010: call 0x00007ffe83abab70

L0015: mov rcx, rax

L0018: add rsp, 0x28

L001c: jmp 0x00007ffe39810040

TypeDispatch.CSharp.TypeDispatch.IsNumeric(System.Object)

L0000: push rbx

L0001: sub rsp, 0x20

L0005: test rcx, rcx

L0008: je short L0043

L000a: call 0x00007ffe83abab70

L000f: mov rbx, rax

L0012: mov rcx, rbx

L0015: call qword ptr [0x7ffe23eda6e0]

L001b: add eax, 0xfffffffb

L001e: cmp eax, 9

L0021: ja short L002a

L0023: mov eax, 1

L0028: jmp short L003d

L002a: mov rax, 0x201e07d5b88

L0034: cmp rbx, rax

L0037: sete al

L003a: movzx eax, al

L003d: add rsp, 0x20

L0041: pop rbx

L0042: ret

L0043: xor eax, eax

L0045: add rsp, 0x20

L0049: pop rbx

L004a: ret

I’m not sure why there is such a massive difference here. Let’s look at IsNumericType:

\_+Tests.IsNumericType(System.Type)

L0000: push rbx

L0001: sub rsp, 0x20

L0005: mov rbx, rcx

L0008: test rbx, rbx

L000b: je short L0051

L000d: mov rcx, 0x7ffe23eda318

L0017: cmp [rbx], rcx

L001a: jne short L0055

L001c: mov rcx, rbx

L001f: call qword ptr [0x7ffe23eda6e0]

L0025: add eax, 0xfffffffb

L0028: cmp eax, 9

L002b: ja short L0038

L002d: mov eax, 1

L0032: add rsp, 0x20

L0036: pop rbx

L0037: ret

L0038: mov rax, 0x201e07d5b88

L0042: cmp rbx, rax

L0045: sete al

L0048: movzx eax, al

L004b: add rsp, 0x20

L004f: pop rbx

L0050: ret

L0051: xor eax, eax

L0053: jmp short L0025

L0055: mov rcx, rbx

L0058: mov rax, [rbx]

L005b: mov rax, [rax+0x98]

L0062: call qword ptr [rax+0x10]

L0065: jmp short L0025

TypeDispatch.CSharp.TypeDispatch.IsNumericType(System.Type)

L0000: push rbx

L0001: sub rsp, 0x20

L0005: mov rbx, rcx

L0008: test rbx, rbx

L000b: je short L0051

L000d: mov rcx, 0x7ffe23eda318

L0017: cmp [rbx], rcx

L001a: jne short L0055

L001c: mov rcx, rbx

L001f: call qword ptr [0x7ffe23eda6e0]

L0025: add eax, 0xfffffffb

L0028: cmp eax, 9

L002b: ja short L0038

L002d: mov eax, 1

L0032: add rsp, 0x20

L0036: pop rbx

L0037: ret

L0038: mov rax, 0x201e07d5b88

L0042: cmp rbx, rax

L0045: sete al

L0048: movzx eax, al

L004b: add rsp, 0x20

L004f: pop rbx

L0050: ret

L0051: xor eax, eax

L0053: jmp short L0025

L0055: mov rcx, rbx

L0058: mov rax, [rbx]

L005b: mov rax, [rax+0x98]

L0062: call qword ptr [rax+0x10]

L0065: jmp short L0025

Yes, they are identical!

Can all this performance difference come down to just the first call?

I am at a loss. Oh, it gets worse. Suppose I define the F# version of IsNumeric like this:

static member IsNumeric2(o: obj) =

o <> null && TypeDispatch.IsNumericType(o.GetType())

instead of

static member IsNumeric(o: obj) =

match o with

| null -> false

| \_ -> TypeDispatch.IsNumericType(o.GetType())

Decompiled to C# it looks:

public static bool IsNumeric2(object o)

{

if (!LanguagePrimitives.HashCompare.GenericEqualityIntrinsic(o, null))

{

return IsNumericType(o.GetType());

}

return false;

}

In IL this looks like:

.method public static

bool IsNumeric2 (

object o

) cil managed

{

// Method begins at RVA 0x20c8

// Code size 23 (0x17)

.maxstack 8

IL\_0000: ldarg.0

IL\_0001: ldnull

IL\_0002: call bool [FSharp.Core]Microsoft.FSharp.Core.LanguagePrimitives/HashCompare::GenericEqualityIntrinsic<object>(!!0, !!0)

IL\_0007: brtrue.s IL\_0015

IL\_0009: ldarg.0

IL\_000a: callvirt instance class [System.Runtime]System.Type [System.Runtime]System.Object::GetType()

IL\_000f: call bool \_/Tests::IsNumericType(class [System.Runtime]System.Type)

IL\_0014: ret

IL\_0015: ldc.i4.0

IL\_0016: ret

} // end of method Tests::IsNumeric2

In JIT ASM: it looks surprisingly like what is generated for the original C# version:

\_+Tests.IsNumeric2(System.Object)

L0000: push rbx

L0001: sub rsp, 0x20

L0005: mov rbx, rcx

L0008: mov rdx, rbx

L000b: mov rcx, 0x7ffe3991cf80

L0015: xor r8d, r8d

L0018: call qword ptr [0x7ffe3143f8e8]

L001e: test eax, eax

L0020: jne short L0037

L0022: mov rcx, rbx

L0025: call 0x00007ffe83abab70

L002a: mov rcx, rax

L002d: add rsp, 0x20

L0031: pop rbx

L0032: jmp 0x00007ffe399e0040

L0037: xor eax, eax

L0039: add rsp, 0x20

L003d: pop rbx

L003e: ret

However, the performance goes into the toilet:

| Method | size | Mean | Error | StdDev | Ratio | RatioSD | Allocated | Alloc Ratio |

|----------------- |-------- |----------:|----------:|----------:|------:|--------:|----------:|------------:|

| CSharpIsNumeric | 1000000 | 1.939 ms | 0.0237 ms | 0.0221 ms | 1.00 | 0.00 | 2 B | 1.00 |

| FSharpIsNumeric | 1000000 | 2.261 ms | 0.0421 ms | 0.0393 ms | 1.17 | 0.03 | 2 B | 1.00 |

| FSharpIsNumeric2 | 1000000 | 11.808 ms | 0.1364 ms | 0.1276 ms | 6.09 | 0.10 | 9 B | 4.50 |

What is going on here?

# Chapter 2 The pool of tears

Not wanting to let it rest, I decided to find out how the coding on that little IsNumeric method could have such a big effect.

So I did it again. Two versions of a simple method in F#, just on e in C#. And the helper they call designed to be identical.

I wrote the helper to be just big enough not to be inlined.

Here is what I have

static member IsEvenRandom(o: obj) : bool =

(System.Random.Shared.Next()

+ System.Random.Shared.Next()

+ System.Random.Shared.Next()) % 2 > 0

static member IsRandom2(o: obj) =

o <> null && TypeDispatch.IsEvenRandom(o)

static member IsRandom(o: obj) =

match o with

| null -> false

| \_ -> TypeDispatch.IsEvenRandom(o)

public static bool IsEvenRandom(object o)

{

return (Random.Shared.Next()

+ Random.Shared.Next()

+ Random.Shared.Next()) % 2 == 0;

}

public static bool IsRandom(object o)

{

return o != null && IsEvenRandom(o);

}

Now we have parity between the C# IsRandom and the F# is Random, but not the F# IsRandom2;

| Method | size | Mean | Error | StdDev | Ratio | RatioSD | Allocated | Alloc Ratio |

|------------------ |-------- |----------:|----------:|----------:|------:|--------:|----------:|------------:|

| CSharpEvenRandom | 1000000 | 9.860 ms | 0.0647 ms | 0.0574 ms | 1.00 | 0.00 | 6 B | 1.00 |

| FSharpEvenRandom | 1000000 | 9.855 ms | 0.1527 ms | 0.1354 ms | 1.00 | 0.01 | 6 B | 1.00 |

| FSharpEvenRandom2 | 1000000 | 20.710 ms | 0.1339 ms | 0.1118 ms | 2.10 | 0.02 | 12 B | 2.00 |

So, what it is not about the difference between IsNumeric in F# (the good one) vs the C# version. But the ancillary method IsNumericType appears to be identical in the generated JIT ASM.

I’m still confused.

# Chapter 3 A Caucus-Race and a long Tale

So let’s try one last time to see where if we can identify where the problem is.

Going back to the original C# code (IsNumeric, IsNumericType). I decided to pit the F# IsNumericType against the C# IsNumericType.

So I wrote two version of IsNumeric in F#. One calls the F# IsNumericType and the other calls the C# IsNumericType:

static member IsNumeric(o: obj) =

match o with

| null -> false

| \_ -> TypeDispatch.IsNumericType(o.GetType())

static member IsNumeric2(o: obj) =

match o with

| null -> false

| \_ -> TypeDispatch.CSharp.TypeDispatch.IsNumericType(o.GetType())

And now we see the difference.

| Method | size | Mean | Error | StdDev | Ratio | RatioSD | Allocated | Alloc Ratio |

|----------------- |-------- |---------:|----------:|----------:|------:|--------:|----------:|------------:|

| CSharpIsNumeric | 1000000 | 1.966 ms | 0.0388 ms | 0.0398 ms | 1.00 | 0.00 | 1 B | 1.00 |

| FSharpIsNumeric | 1000000 | 2.169 ms | 0.0158 ms | 0.0140 ms | 1.10 | 0.02 | 2 B | 2.00 |

| FSharpIsNumeric2 | 1000000 | 1.821 ms | 0.0122 ms | 0.0108 ms | 0.92 | 0.02 | 2 B | 2.00 |

F# calling the C# IsNumericType is the fastest one!!!

# Chapter 4: The Rabbit sends in a little Bill

Directly calling F# IsNumericTYpe and C# IsNumericType – cuttong out the middleman:

| Method | size | Mean | Error | StdDev | Ratio | RatioSD | Allocated | Alloc Ratio |

|---------------------- |-------- |---------:|----------:|----------:|------:|--------:|----------:|------------:|

| CSharpDirectIsNumeric | 1000000 | 1.850 ms | 0.0361 ms | 0.0443 ms | 1.00 | 0.00 | 2 B | 1.00 |

| FSharpDirectIsNumeric | 1000000 | 2.130 ms | 0.0390 ms | 0.0326 ms | 1.14 | 0.03 | 2 B | 1.00 |

So despite sharplab.io saying they JIT to the same ASM, something clearly is different. In real life, the difference in IL ends up causing something different to be generated in the ASM. And that difference makes a 12-14% decline in performance.

With = on the last type

| Method | size | Mean | Error | StdDev | Ratio | RatioSD | Allocated | Alloc Ratio |

|-------------------------- |-------- |---------:|----------:|----------:|------:|--------:|----------:|------------:|

| CSharpDirectIsNumericType | 1000000 | 1.873 ms | 0.0295 ms | 0.0247 ms | 1.00 | 0.00 | 1 B | 1.00 |

| FSharpDirectIsNumericType | 1000000 | 3.007 ms | 0.0182 ms | 0.0162 ms | 1.61 | 0.02 | 2 B | 2.00 |

With op\_Equal on the last type:

| Method | size | Mean | Error | StdDev | Ratio | RatioSD | Allocated | Alloc Ratio |

|-------------------------- |-------- |---------:|----------:|----------:|------:|--------:|----------:|------------:|

| CSharpDirectIsNumericType | 1000000 | 1.818 ms | 0.0176 ms | 0.0147 ms | 1.00 | 0.00 | 1 B | 1.00 |

| FSharpDirectIsNumericType | 1000000 | 2.128 ms | 0.0278 ms | 0.0246 ms | 1.17 | 0.02 | 1 B | 1.00 |