A comparison of the compilation of

(defn f4 [n]

(let [n (int n)]

(loop [i (int 0)

x (int 0)]

(if (= i n)

x

(recur (inc i) (+ x i))))))

JVM HERE

CLR HERE

COMMENT

0: aload\_1

1: invokestatic #72; //Method clojure/lang/RT.intCast:(Ljava/lang/Object;)I

4: istore\_2

IL\_0000: ldarg.1

IL\_0001: call int32 [Clojure]clojure.lang.RT::intCast(object)

IL\_0006: stloc.0

(let [ n (int n) ] …

JVM: n is in local2,

CLR: n is in local0

5: getstatic #47; //Field const\_\_3:Ljava/lang/Object;

8: invokestatic #72; //Method clojure/lang/RT.intCast:(Ljava/lang/Object;)I

11: istore\_3

IL\_0007: ldsfld object 'dm/testcast$fn\_\_59$f4\_\_61\_base'::const\_\_3

IL\_000c: unbox.any [mscorlib]System.Int32

IL\_0011: call int32 [Clojure]clojure.lang.RT::intCast(int32)

IL\_0016: stloc.1

(loop [i (int 0) …

JVM/CLR – const3 is 0. I might be able to type the field better and avoid the unbox.any. Does it matter?

JVM: I is local.3

CLR: I is local.1

12: getstatic #49; //Field const\_\_4:Ljava/lang/Object;

15: invokestatic #72; //Method clojure/lang/RT.intCast:(Ljava/lang/Object;)I

18: istore 4

IL\_0017: ldsfld object 'dm/testcast$fn\_\_59$f4\_\_61\_base'::const\_\_4

IL\_001c: unbox.any [mscorlib]System.Int32

IL\_0021: call int32 [Clojure]clojure.lang.RT::intCast(int32)

IL\_0026: stloc.2

(loop [ … x (int 0) ]

JVM/CLR – const4 is 0. I might be able to type the field better and avoid the unbox.any. Does it matter?

JVM: x is local.4

CLR: x is local.2

20: iload\_3

21: invokestatic #45; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;

24: iload\_2

25: invokestatic #45; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;

28: invokestatic #78; //Method clojure/lang/Util.equiv:(Ljava/lang/Object;Ljava/lang/Object;)Z

31: ifeq 43

IL\_0027: ldloc.1

IL\_0028: call object […::Int32ToObject(int32)

IL\_002d: ldloc.0

IL\_002e: call object […::Int32ToObject(int32)

IL\_0033: call bool clojure.lang.Util::equiv(object,object)

IL\_0038: brfalse IL\_0048

(if (= I n)

BOTH: testing is a call to c.l.Util.equiv which takes objects. So we have to box. Each has its own approach.

BOTH: Equivalent to if Util.Equiv( (object)I, (object) n )

Test is true, get ready to return

34: iload 4

36: invokestatic #45; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;

39: goto 59

42: pop

IL\_003d: ldloc.2

IL\_003e: box [mscorlib]System.Int32

IL\_0043: br IL\_0077

Return (object)x;

Test is false, prepare to loop

43: iload\_3

44: invokestatic #83; //Method clojure/lang/Numbers.inc:(I)I

47: iload 4

49: iload\_3

50: invokestatic #87; //Method clojure/lang/Numbers.add:(II)I

53: istore 4

55: istore\_3

56: goto 20

IL\_0048: ldloc.1

IL\_0049: call int32 [Clojure]clojure.lang.Numbers::inc(int32)

IL\_004e: box [mscorlib]System.Int32

IL\_0053: unbox.any [mscorlib]System.Int32

IL\_0058: stloc.3

IL\_0059: ldloc.2

IL\_005a: ldloc.1

IL\_005b: call int32 [Clojure]clojure.lang.Numbers::'add'(int32,

int32)

IL\_0060: box [mscorlib]System.Int32

IL\_0065: unbox.any [mscorlib]System.Int32

IL\_006a: stloc.s V\_4

IL\_006c: ldloc.3

IL\_006d: stloc.1

IL\_006e: ldloc.s V\_4

IL\_0070: stloc.2

IL\_0071: br IL\_0027

Essentially computing , I = I + 1, x = x + I, but in parallel.

The JVM version uses a more efficient trick: Compute I+1, leave it on the stack, compute x+I, leave it on the stack

Then assign to x and assign to I the values on the stack.

I can’t do that easily in ExpressionTrees, so I compute into temporaries. I might be able to speed this up.

The CLR version does the equivalent of i1 = I + 1; x1 = x+I; I = i1; x = x1.

59: areturn

IL\_0076: ldnull

IL\_0077: ret

The ldnull is unreachable.

At worst, you might expect a factor of 2 slower performance?

Actual timing (using a JVM timing method versus a CLR timing method): 20X slower.

Please explain.