

# Re: Higher Order Instructions

---

*Source:*

<http://www.tech-archive.net/Archive/DotNet/microsoft.public.dotnet.framework.clr/2007-04/msg00044.html>

---

- *From:* "Christopher Diggins" <[cdiggins@xxxxxxxxxx](mailto:cdiggins@xxxxxxxxxx)>
  - *Date:* 12 Apr 2007 12:37:46 -0700
- 

On Apr 11, 2:18 pm, Barry Kelly <[barry.j.ke...@xxxxxxxxxx](mailto:barry.j.ke...@xxxxxxxxxx)> wrote:

Christopher Diggins wrote:

On Apr 8, 3:03 pm, Barry Kelly <[barry.j.ke...@xxxxxxxxxx](mailto:barry.j.ke...@xxxxxxxxxx)> wrote:

Christopher Diggins wrote:

I was wondering if there has been any research into adding higher-order instruction to the CIL? In other words instructions that either push or pop instructions on the evaluation stack.

There are only a few core instructions that would be necessary to build others :

- constantly : pop a value, push an instruction on the stack that returns that value
- compose : pop two instructions, push a new instruction that that evaluates the first, then the second.
- eval : pop an instruction and evaluate

CIL maps linearly to machine code. What you're describing here doesn't.

## Re: Higher Order Instructions

It's pretty close though. Data as instructions is not an uncommon technique in assembly code as far as I understand.

Sure, but the CLR (& JVM) don't model von Neumann machines, as I sent a comment to your blog :)

That doesn't matter though. The point is that if we simply add typed higher order instructions to a stack machine these can be trivially translated to a von neuman machine (i.e. common CPU architecture). For those interested a naive (read unoptimized) scheme for translating from higher order stack instructions to assembly can be found at <http://cdiggins.com/2007/04/11/cat-higher-order-instructions-to-assembly/>

And I take slight exception to suggesting that 'data as instructions' is being not uncommon in assembly, because it's effectively self-modifying code, which has a bad reputation of being hard to understand and debug –

Yes the technique is uncommon in human written code, but there is no reason not to generate it.

which is why we use higher order models that have type systems etc. that can be more formally & rigourously tested etc.

Yes, like the Cat type system (I'll refer readers again to <http://www.cat-language.com/paper.html> )

My point was efficiency though, and even more importantly, intuitiveness of the efficiency of imperative code in the ordinary linear style, with similar arguments to the old "worse is better" story re C/Lisp etc.

I don't understand what you are saying.

You can't deny that your ideas are very similar to a kind of 'Lisp on a stack' :)

I would characterize the ideas as more of "Haskell" on a stack, due to the type safe nature of the operations. Lisp is just too flexible to afford the kinds of optimizations and safety we need from intermediate languages.

## Re: Higher Order Instructions

It's not something I'd rule out on this kind of basis though, don't get me wrong, I love writing compilers and this kind of thing has appeal. I only expand on it because it's the biggest thing I see.

However, if your instructions aren't first class (i.e. can't be passed or returned to / from methods, or stored / loaded from variables), then this scheme amounts to macro expansion since e.g. your compose operation can be statically expanded to its constituents.

What I propose are first-class instructions, but only in the context of the IL itself. You wouldn't be able to do straight macro expansion because of the possibility of conditional composition based on run-time values.

And if your instructions are first class, verification would not be easy for e.g. constrained devices, and performance analysis would not be trivial. It could have similar problems by analogy to e.g. call by name from Algol, where evaluating an argument inside a function might be as simple as a variable read or as complex as a network call.

Because it is restricted to the IL level, the composed IL functions could not come from an untrusted source.

OK, I understand what you're getting at much better now.

I get the impression from reading some of your other stuff that you have a rather different stack in mind to the one that exists in the CLR or JVM, which basically only exists as arguments for other instructions (including CALL etc. instructions). For example, on both CLR and JVM:

\* every method has its own stack

That restriction is still fine. I can combine functions through

## Re: Higher Order Instructions

inlining.

\* by simulating instructions, it should not be possible for the stack to ever have a different height or type composition for any possible execution path – this is the foundation of verification

This is guaranteed by the type system.

\* the stack is not arbitrarily permutable without using external storage like a local etc.

This is also guaranteed by the type system.

Would your proposed changes break these things?

Nope.

In particular, would these composed instructions turn into .NET delegates at any point?

Nope.

Or would they be condemned to live only on the frame of the method that created them?

Well I would expect one to be able to return opcode blocks from methods.

So let me be clear here, what I propose requires the introduction of a new data type, which consists of a list of opcodes. It is like a method, except it isn't ever seen by the user, only compiler writers. They use it to emulate higher order functions.

Also, as it exists, CIL can be trivially interpreted, in a pinch (type info added to stack values or to instructions after single-pass analysis). What your suggesting seems to me to be more like a kind of graph reduction machine, which would (naively, from 30

## Re: Higher Order Instructions

seconds analysis)  
suggest to me continuous dynamic allocation, quite unlike  
CIL.

What do you mean by continuous dynamic allocation?

I mean, when simulating the .NET or JVM stack, one typically just pushes and pops, and maybe writes into an array of locals / parameters or news up an object. With this model, I'm trying to figure out what you'd be pushing on after one of those compose operations.

An opcode block. An array of raw opcodes allocated either on the stack (if small enough) or on the heap.

It seems to me to be a dynamically allocated structure from the GC heap, since it can be grown to arbitrary size with successive compose operations, yet it's not an object.

Yes, we can at the very least make a simple optimization to separate between small opcode blocks (allocated on the stack and "copied") and large opcode blocks, placed on the heap.

Also, it needs to magically turn into machine code or stay as a graph, depending on whether it's going to be called or composed again.

I don't see why you wouldn't just turn it into machine code right off the bat, and use that as your representation.

Turning it into machine code for calling isn't going to be totally trivially cheap, it's going to have to enter a compiler somewhere...

Well yes, just like any byte code.

If it turns into a delegate, won't you want to cache that somewhere, to avoid getting that hit again next time... and doesn't this work seem like not such a big win over just doing it yourself with `DynamicMethod`...?

## Re: Higher Order Instructions

Sorry, you lost me.

It is true that higher order functions would require a graph reduction machine, however this is offset by the fact that far fewer instructions are needed to achieve high-level functionality. A lot of dynamic IL emitting code, which is currently very expensive, could be replaced by higher-order IL code.

I guess, the more I think about it, I'm not sure what you're proposing, because (as I've sketched out above) when I think about how a JIT would "want" to do it, it seems like it isn't a win, and that you'd be better off if the runtime \*interpreted\* your instructions – or at least, that's the only way to get a net win out of it.

You have come to the conclusion that interpreting higher order functions would be faster than compiling? I completely disagree.

What would you say, if I generated assembly code for the following example:

```
int[] a = new array[1000000];
for (int i=0; i < a.length(); ++i) a[i] = i;
Map(a, MapDelegate(int x) { return x % 2 == 0 ? x * 2 : x })
int sum = Fold(a, FoldDelegate(int x, int y) { return x + y; })
```

And it was over 20 times as fast as C#? Would you buy me a pint of guinness?

Cheers,  
Christopher Diggins

.