A Strategy for Compiling APL for the .Net Framework

by Adrian Smith (adrian@causeway.co.uk)

Abstract

The APL literature has many papers on APL compilation, mostly dating from before 1982. What was possible then should be more possible now, given the strong vector support and excellent garbage-collection in the .Net *Common Language Runtime* (CLR). This paper outlines the approach being taken by Causeway to ship the *RainPro* and *NewLeaf* engines as pure .Net DLLs. This involves some hard choices on what subset of APL is truly essential. The rationale for these choices is discussed, and various conversions from Dyalog APL to C# are illustrated.

Motivation

Simple survival. If Causeway is to continue to ship 'possibly the best graphics engine in the world' to the next generation of .Net users, it has to be 'pure managed code' and it should be fast, lightweight and ideally a single DLL. This means that we cannot continue to bundle a complete runtime APL interpreter with our distribution. We cannot rewrite in C# – the only specification we have is the existing APL codebase. Our only path forward is to attempt to regenerate this APL treasure-house into C# source and compile that.

We also believe that to move wholesale to C# would freeze innovation and development in its tracks. APL is still the most productive environment for playing with ideas, experimenting with data and prototyping new solutions. We want to keep our development in APL; we just want to ship the results in .Net.

Literature Search

This may be an old-fashioned approach, but I started by tracking down every paper I could find on APL and compilation. There is a goldmine in old conference proceedings from 1978 to around 1982, then along came APL2 and it just ends there. Maybe that tells you something about APL2?

The overwhelming conclusion was that with 'very few restrictions' VS APL is perfectly possible to compile. Most of the restrictions are either plain common sense (clearly you can't use *execute* or *quadFX* to change the meaning of a token

dynamically) or accepted as good habits by the vast majority of modern APLers (don't re-use locals with different data types, don't look up the stack for data localised in your caller, avoid fancy nested structures if a simple array will do the job, ...) so no great issue there. In common with all APLers, I break my own rules fairly frequently, but I can face the pain of tidying up my APL code to comply with strictures like these. It will probably improve as a result.

The other valuable resources were the survey papers (for example from STSC) showing the usage frequency of the primitives, the inner and outer products, and the rank of data most commonly encountered by the simple primitives like plus and times. This last was quite surprising, but very encouraging. Around 99.6% of ALL expressions were found to operate on arrays of rank 0,1,2 (over a 2-week period of typical usage of the STSC timesharing service). Conclusion – if we write an engine that does scalars, vectors and matrices, it should cover 99.6% of the 'typical' APL system. I can live with that.

A similar survey found the average length of a vector to be 16. If this is typical of our code, then we might find that eliminating interpreter overhead might actually give us a considerable speed-up. An APL engine can only compete for speed with compiled code when the average length of a vector gets up into the thousands.

Spike Test – Eratosthenes yet again

The classic prime-number sieve is my favourite testbed for this kind of experiment. It loops a bit, it hits vectors (and hence memory-management) pretty hard, and you don't need many primitives to make it run. Here is the APL code:

```
      ∇ primes←Eratos max;candidates;check
[1]    ⍝ Returns primes up to n by the sieve of Eratosthenes
[2]     primes←1
[3]     candidates←1↓⍳max
[4]
[5]     :While 0<⍴candidates
[6]       primes←primes,check←candidates[1]
[7]       candidates←(0<check|candidates)/candidates
[8]     :End
      ∇

      ⍴Eratos 100000
9593
```

... which runs in around 10sec in Dyalog APL on my 400MHz laptop. This is a fairly hard target to beat, as it loops 9593 times, but must repeatedly compress a reasonably long vector of possible primes each time around the loop.

At the other end of the scale, we have a hand-coded C# version:

```
using System;
using System.Collections;

class MainClass
{
    public static void Main(string[] args)
    {
        int val = 100000;
        if(args.Length > 0) val = Int32.Parse(args[0]);
        Console.WriteLine("Start with "+val+" ...");
        ArrayList e = Eratos(val);
        Console.WriteLine("Done - " + e.Count);
    }

    private static ArrayList Eratos(int max){
        ArrayList primes = new ArrayList();
        primes.Add(1);
        int[]  candidates = new int[max - 1];
        for(int i = 0; i < max - 1; i++) candidates[i] = i + 2;

        while(candidates.Length > 0){
            int check = candidates[0];
            primes.Add(check);
            bool[] bv = new bool[candidates.Length];
            for(int i = 0; i < candidates.Length; i++) bv[i] = (candidates[i] % check) > 0;
            candidates = Compress(candidates, bv);
        }

        return primes;
    }

    private static int[] Compress(int[] from, bool[] by){
        int size = 0;
        foreach(bool b in by) if(b) size++;
        int[] res = new int[size];
        int resp = 0;
        for(int i = 0; i < by.Length; i++) if(by[i]) res[resp++] = from[i];
        return res;
    }
}
```

This runs in just over 8sec on the same hardware, and is a fairly good facsimile of the APL algorithm, written by Richard Smith in about 20 minutes. It proves a number of things:

1. The C# runtime is definitely not a dog. Experiences of Java made us wary of any environment which compiles to a virtual machine. It would appear that the Microsoft guys have not hit us with a speed penalty here.

2. The garbage collection works really well. That *candidates* vector gets re-assigned and discarded 9593 times and there is no visible hit on the memory usage of the machine while this is running. Good.

3. The code required to implement *compress* is pretty straightforward. OK, this just does the vector case, and only for integers, but you can see that the C# array capabilities are not too stupid. Arrays are fully dynamic, and indexing works as you expect. They even know their own length.

4. That *while* structure looks suspiciously similar to the APL version. Maybe (just maybe) the modern APL control structures map across almost unchanged. That would be a major bonus.

Conclusion – this is worth a go. So how do we set about transforming our APL source-code into the C# version automatically? Time to make some more choices.

Route-1 – the easy way

Imagine a library of APL primitives where everything simply takes *object* and returns *object*. Imagine a memory-manager class which can store data for you, and even imitate the APL scoping rules. Well, we did more than imagine it; in the cool of the evening in Vancouver (Causeway did *SVG-Open* last year) we coded enough of it to run *Eratos*. Which it did, like a dog. Like a 2min 4sec dog, in fact. The C# code looks fine, but the snag with this approach is that every primitive has to run through a block of code to find out what kind of thing it was passed. Then it must coerce one or both arguments to the most general type (say *double[]* for most of the arithmetic stuff) before doing whatever it does, before wrapping the result up as *object* ready to return it. We knew it would be slow, but not *this* slow. Back to the drawing board.

Route-2 – the *SmartArrays* (and *Rowan*) way

Both Jim Brown and Richard Smith take the approach of making the Java/C# programmer declare a specific data-type (Jim's is an *smArray*) which is a little structure just like the typical APL array of mainframe days. Effectively this describes the data content for you, saving nearly all the overhead incurred by Route-1. *Rowan* runs *Eratos* in about 18sec, so this is definitely a tenable approach. We chose not to do it, because we wanted to generate C# that looked like the C# a 'real' programmer would write. We also took note of just how much scalar code there is in the average chunk of commercial APL, and felt that on balance our primitives should (for example) be able to add an *int* to an *int[]* without the extra fuss of declaring our own special data-type.

Route-3 – use native C# data-types everywhere

The spike-test for this one took all of a morning to write, and it ran *Eratos* in 11sec. The downside is that you need lots and lots of overloads for all the primitive functions – at the last count *AE.Times* managed 63 variants, working upwards

from *Times(int,int)* and including *Times(int,bool[])* and so on. Unfortunately, C# regards rank-2 and rank-3 arrays as different types, so we took a hard decision and stopped at rank-2, just to keep everyone sane. One or two really oddball functions (like *QuadGets*) do take *object* and try to figure it out at runtime, but we came to the view that in general, the benefit of strong typing was worth the pain.

Of course, most of the pain comes at the APL end – this approach actually makes the function-library much simpler to write (given some suitable template-expansion code) and the C# compiler takes almost all the decisions at compile time. It also eliminates nearly all run-time errors – if you can get your converted APL past the compiler, it will almost certainly run correctly.

The hard part is moved over to the code-convertor, which must be able to deduce the correct data-type for all the local variables *lexically*. I did toy with the idea of infecting the target functions with a suitable analyser, running the application a few times, and then patching the original APL code with declarative comments. It felt like a kludge, but it did at least reassure me that I was fairly reliable in avoiding local-variable re-use. Much better to do it right.

Figuring out Data-types at Parse Time

Well, the C# compiler can do it (how else does it know to call the correct overload?) so surely we can do the same trick? Let's start with something really obvious ...

```
      cc.chk 'a←⍳12'

       a      ←      ⍳      12
MARK   NOUN   ASGN   VERB   NOUN
a = AE.Range(12);  int[]
```

A brief word of explanation. I have a `cc` namespace which does all the hard stuff. The first stage is called **Word Formation** and this is followed by **Parsing**. All `chk` does is run these back to back with ⎕← in between so you can see what the word-formation saw. I copied Roger Hui's parse-table hook, line and sinker from *Vector* Vol.9 No.4 (p.92) so I use the J terminology throughout.

The two-element result from `cc.chk` gives me the converted expression, and the data-type of the final result. It takes an optional left argument which I can use to tell it the types of any previously assigned names.

For example:

```
      ('double[]' 'local') cc.chk 'a←⍳local'
       a      ←      ⍳      local
MARK   NOUN   ASGN   VERB   NOUN
TYPE - double[] not supported by AE.Range
a = AE.Range(local);  ????
```

Not unreasonably, this gets upset about being asked to run ⍳ on a floating-point vector. It has a go at converting the statement, but returns ???? as the resulting datatype, as this is undetermined at this point. If we were to run:

```
      'int' 'local' cc.chk 'a←⍳local'
       a      ←      ⍳      local
MARK   NOUN   ASGN   VERB   NOUN
a = AE.Range(local);  int[]
```

... then everything is fine. So how about:

```
      'int' 'local' cc.chk 'a←12⍴⍳local'
       a      ←      12     ⍴      ⍳      local
MARK   NOUN   ASGN   NOUN   VERB   VERB   NOUN
a = AE.Reshape(12,AE.Range(local));  int[]
```

... and so on. It looks tricky, but actually it is remarkably easy for the parser to match up the incoming data-types for each primitive with the (unique) combination of data-types supported by the corresponding primitive in the library, and hence unambiguously establish the return type. We just keep rolling along, adding parentheses as we go:

```
      'int' 'local' cc.chk 'a←2.5×12⍴⍳local'
       a      ←      2.5    ×      12     ⍴      ⍳      local
MARK   NOUN   ASGN   NOUN   VERB   NOUN   VERB   VERB   NOUN
a = AE.Times(2.5,AE.Reshape(12,AE.Range(local)));  double[]
```

... and when we hit an assignment, we store away the incontrovertible fact that a is now (beyond a shadow of a doubt) an array of type *double[]*. If a subsequent line should try to assign something else, we splatter a message all over the session.

Which just leaves:

```
    cc.chk 'a←1 0 1'
      a      ←      1 0 1
MARK   NOUN   ASGN   NOUN
a = new int[] {1,0,1};  int[]
```

... and a few things like it. Is this an array of type *bool[]* or *int[]* or even *double[]*? There is no way we can know, so I am afraid you just have to declare these.

Fortunately, this sort of thing doesn't come up too often in real code and when it does, the convertor can take the hint:

```
      ('double[]' 'a') cc.chk 'a←1 0 1'
         a      ←      1 0 1
MARK   NOUN   ASGN   NOUN
a = new double[] {1,0,1};  double[]

      ('bool[]' 'a') cc.chk 'a←1 0 1'
         a      ←      1 0 1
MARK   NOUN   ASGN   NOUN
a = new bool[] {true,false,true};  bool[]
```

and poor *zilde* gets some serious abuse in this case:

```
      ('int[]' 'a') cc.chk 'a←⍬'
         a      ←      ⍬
MARK   NOUN   ASGN   NOUN
a = new int[0];  int[]

      ('bool[][]' 'a') cc.chk 'a←⍬'
         a      ←      ⍬
MARK   NOUN   ASGN   NOUN
a = new bool[0][];  bool[][]
```

Basically, the default assumption is *int* or *int[]* if there is nothing to go on, otherwise any constant vector will be formatted up to match the pre-set type of the name to the left of the assignment. Which just leaves:

```
      cc.chk 'a←∨/1 0 1'
         a      ←      ∨/     1 0 1
MARK   NOUN   ASGN   VERB   NOUN
TYPE - int[] not supported by AE.OrReduce
```

... where the default assumption is clearly wrong, and the error spurious. Again, this hardly ever happens in real application code, but the way out (until I can think of something better) is to define a pair of globals `true` and `false` in your workspace, and recode the above as:

```
      cc.chk 'a←∨/true false true'
         a      ←      ∨/     ⊣true false true⊢
MARK   NOUN   ASGN   VERB   NOUN
a = AE.OrReduce(new bool[]{true,false,true});  bool
```

... which stops up that hole for us. If (for some very obscure reason) we wanted it treated as *double[]* here (rather than *int[]*) the parser has one more trick up its sleeve.

Consider:

```
      cc.chk 'a←+1 0 1'
         a      ←      +      1 0 1
   MARK  NOUN   ASGN   VERB   NOUN
   a = AE.Promote(new int[] {1,0,1});  double[]
```

Purists may scream now. What we chose to do is to implement the C# version of monadic *plus* as 'leave the value unchanged, but increment the data-type' so an extreme case would be:

```
      cc.chk 'a←+++true false'
         a      ←      +      +      ⊣true false⊢
   MARK  NOUN   ASGN   VERB   VERB   NOUN
   a = AE.Promote(AE.Promote(new bool[]{true,false}));  double[]
```

Of course you can do the usual stuff with `floor` and `1=` to demote to integer and boolean respectively. Note (in passing) that strands are treated as NOUNs here, but get flagged by the left and right tack, just to be sure. There is a reason for this, to do with multiple arguments to functions. We will come to this later.

Stocktaking - what do we have and what do we need?

Well, it looks pretty plausible that we can convert an APL expression to a C# expression which will do the exact same job, as long as the data-types are knowable in advance. There will be a few bear-traps like:

```
'int[]' 'qq' cc.chk 'qq/⍨←msk'
        qq     /      ⍨      ←      msk
 MARK   NOUN   VERB   ADVB   ASGN   NOUN
 qq = AE.Compress(msk,qq);  int[]
```

... but these all prove to be a lot easier than you might think. Once you have decided to use a Hui-style parse-table, then fancy syntax is just adding patterns to the table, and all the Dyalog extensions fall out pretty easily.

So - we can convert an expression. The next challenge is to convert a function, which adds a few new challenges, starting with branching and control-structures. Let's tackle these next.

GoTo Considered (mostly) harmless

I very, very rarely use branches and labels in code I write today, but it seemed stupid to force a rewrite of old code just for the sake of it, when C# supports labels and *Goto* quite happily.

Here is my testbed:

```
      ∇ msg←branches fred
[1]    ⍝ Assorted branch styles
[2]    ⍝:Public
[3]    ⍝: int fred
[4]    ⍝: string msg
[5]     msg←'Hello from branches'
[6]     →(fred<6)↓L1
[7]    L1:→0/⍨fred<4
[8]     →(fred>2)↑Done
[9]     →0
[10]   Done:→Quit
[11]   Quit:msg,←' ... Quit reached'
      ∇
```

It has a couple of special comments, but otherwise looks just like the stuff we all had to write before James Wheeler smuggled control structures into APL+Win.

Let's convert this to C# and see what it looks like:

```
  ↑cc.ccr 'branches'

// ============ branches ============
// Assorted branch styles

public string branches(int fred)
{
  string msg;  // Result
  msg = "Hello from branches";
  if (AE.EQ(0,(fred < 6))) goto L1;
 L1:if (fred < 4) return msg;
  if (fred > 2) goto Done;
  return msg;
 Done:goto Quit;
 Quit: msg = msg + " ... Quit reached";
```

Just for old-times' sake, I even exdent the labels one stop. Basically you can use simple branching instructions, but any attempts to do clever stuff that manipulates vectors of labels:

```
      cc.chk '→expr⌽lab'
       →      expr  ⌽      lab
  MARK  CTRL  NOUN  VERB  NOUN
 Unsupported computed branch AE.Rotate(expr,lab)
```

... are doomed to failure. Fortunately, these are usually very easy to redo.

Control structures R Us

Just how well do the APL structures map over into C#? This really is crucial to maintaining the readability of the generated code, and absolutely essential if we are ever to abandon APL altogether and maintain *RainPro* only in C#. Unlikely for the base graphics engine, but very likely indeed for things like the SVG output conversion. Let's start with the easy ones and build up from there:

```
     ∇ r←ifreturn x
[1]   ⍝ Simple Return example
[2]   ⍝:Public
[3]   ⍝: int x
[4]   ⍝: string r
[5]
[6]    :If x>0
[7]      r←'Out of here'
[8]      :Return
[9]    :Else
[10]     r←'Done'
[11]   :End
     ∇
```

```
// ============ ifreturn ============
// Simple Return example

public string ifreturn(int x)
{
  string r;  // Result
  if (x > 0) {
    r = "Out of here";
    return r;
  } else {
    r = "Done";
  }
  return r;
}
```

At this point, I will have upset about 50% of the C# programmers in the world. Just as we APLers have religious wars about ⎕IO, conventional programmers have wars about code layout conventions. I went for the old K&R style, mainly because it generates a lot less white space, and the lines match quite closely with the lines of APL code they came from. Here is a slightly more complex case:

```
     ∇ r←ifandifandifelse x
[1]   ⍝ Basic IF block example
[2]   ⍝:Public
[3]   ⍝: int r,x
[4]
[5]    :If x>0
[6]    :AndIf x>1
[7]    :AndIf x>2
[8]      r←0
[9]    :Else
[10]     r←12
[11]   :End
     ∇
```

```
public int ifandifandifelse(int x)
{
  int r;  // Result

  if (x > 0
    && (x > 1)
    && (x > 2)) {
    r = 0;
  } else {
    r = 12;
  }

  return r;
}
```

Again, the match is very accurate, and we get the same behaviour that the 'and' clauses are only evaluated if all the prior expressions succeed. Note in passing that the conversion process recognises where a C# native function (like *greaterthan* on two scalars) is adequate to do the job, and doesn't bother to call our APL

library for these. It is slightly depressing just how often this seems to happen in 'realistic' application code. Anyway, here's a *For* loop to see how that goes:

```
 r←forloop n;ct;stuff;nv
⍝ Basic looper
⍝:Public
⍝: int    n,stuff
⍝: string r

 nv←⍳32 ⋄ stuff←0

 :For ct :In ⍳⍴nv
   stuff+←nv[ct]+1000
 :End

 r←'Done ',⍕stuff
```

```
public string forloop(int n)
{
  string r;  // Result
  int   stuff;
  int[] nv;

  nv = AE.Range(32);
  stuff = 0;

  for (int ct = 1; ct <= nv.Length; ct++) {
    stuff = stuff + (nv[ct-1] + 1000);
  }
  r = "Done " + AE.Format(stuff);
  return r;
}
```

The conversion special-cases loops over *iota* as true *for(;;)* structures. Anything else turns into a C# *foreach()* block which is much more general, but less efficient.

A couple of other things to note here:

- we can generally forget monadic *rho* in favour of the *Length* property, which all arrays support automatically.
- indexing by scalars is fine, but C# is origin-0 so we must subtract 1, always.
- character vectors are mapped to C# strings, and the format primitive always returns a string, rather than a vector of characters. This allows us to map most string catenations to the C# + operator which adds hugely to readability, but does have some annoying side effects. Another one to upset the purists, but the pragmatists will understand.

So far, so good. *While* loops fall out easily enough:

```
 whileleave arg
⍝ Check out conversion
 :While true
   ⎕←arg
   :If arg<3
     :Leave
   :End
   arg←arg-1
 :EndWhile
```

```
public void whileleave(int arg)
{
  while (true) {
    AE.Display(arg);
    if (arg < 3) {
      break;
    }
    arg = arg - 1;
  }
}
```

Note in passing that we chose to treat *assignment to quad* as a function (which echos its argument to the console, then returns it untouched).

That leaves *Select/Case* as the only slight oddball.

```
  r←caselist item;opt
⍝ Test caselist block
⍝:Public
⍝: string r,item,opt

  r←'(not set)'
  opt←'One'

  :Select item
  :Case 'fat'
    r←'fatter'
  :CaseList 'cat' 'moggy'
    :Select opt
    :Case 'One'
      r←'Hello cat'
    :Case 'Two'
      r←'Hello moggy'
    :Else
      r←'not my cat'
    :EndSelect
  :CaseList 'bonzo' 'snapper'
    r←'not my dog'
  :Case 'sat'
    r←'Sitting'
  :End
```

```
public string caselist(string item)
{
  string r;  // Result
  string opt;
  r = "(not set)";
  opt = "One";
  switch (item) {
    case "fat":
      r = "fatter";
      break;
    case "cat": case "moggy":
      switch (opt) {
        case "One":
          r = "Hello cat";
          break;
        case "Two":
          r = "Hello moggy";
          break;
        default:
          r = "not my cat";
          break;
      }
      break;
    case "bonzo": case "snapper":
      r = "not my dog";
      break;
    case "sat":
      r = "Sitting";
      break;
  }
  return r;
}
```

This one (as you can see) was a touch tricky to get right. You have to add in all those *break;* statements, as well as chopping up the *CaseList* clause into a list of *cases*. However (major bonus) in C# we can use **strings** in a case block which is a huge step forward from Java where you can only *switch/case* on integers.

Of course, control structures are not the only block element supported by APL. There is *each* to cope with, and in Dyalog APL we might encounter *composition* and *dynamic functions*, probably used in combination with operators like *each* or *reduction*. How do we get around these?

Various Operators and Inline Dynamic Functions

In sketching the initial design for the C# generator, we had decided to follow A+ and simply 'can' all the common reductions, scans and inner/outer products as primitives. This remains a good approach (for example, expressions like `+\vec` can run much faster when special-cased), but most modern APLs make life a little

harder by permitting user-defined functions to the left of operators like reduction and scan, for example:

```
      plus/⍳3
  6
```

... where `plus` has the obvious (dyadic) definition. The way interpreters handle this is to create a temporary 'derived function' *plus-reduction* and then proceed to apply this monadically to the right argument. The only solution I can see is to copy this approach exactly, for example:

```
  cc.chk'plus/⍳3'
```

```
int plusReduce(int[] myrarg)  // Monadic Reduce
{
 int res = myrarg[0];
 int elem = myrarg.Length;
 if( elem > 1 ) {
   res = plus(myrarg[elem-2],myrarg[elem-1]);
   for (int i=elem-3;i>=0;i--)
     res = plus(myrarg[i],res);
 }
 return res;
}

 plusReduce(AE.Range(3));  int
```

The target function is type-checked, and assuming it is dyadic, and supports matching left and right argument types (both must be of rank one less than the argument to the generated temporary function) then we just make a block of C# code to do what reduction does (walk back down the array from right to left) and return as the parsed expression a simple monadic call to it.

Clearly, the same approach will work for *Scan*, but *Each* is a lot nastier, as the operator itself can be monadic or dyadic, which affects the acceptable valence of its function argument(s). Here is a simple example of the dyadic case:

```
  cc.chk '3 take ¨(1 2)(3 4 5)'
```

```
int[][] takeEach(int mylarg,int[][] myrarg)  // Dyadic Each
{
 int[][] res = new int[myrarg.Length][];
 for (int i=0;i<myrarg.Length;i++)
   res[i] = take(mylarg,myrarg[i]);
 return res;
}

 takeEach(3,new int[][]{new int[]{1,2},new int[]{3,4,5}});  int[][]
```

... and when we start allowing dynamic functions, you really have to cross your fingers and hope it works:

```
  cc.chk '3 {⍺↑⍵} ¨(1 2)(3 4 5)'

int[] noname_d1(int alpha, int[] omega) { return AE.Take(alpha,omega); }  // Dfn dyad

int[][] noname_d1Each(int mylarg,int[][] myrarg)  // Dyadic Each
{
 int[][] res = new int[myrarg.Length][];
 for (int i=0;i<myrarg.Length;i++)
   res[i] = noname_d1(mylarg,myrarg[i]);
 return res;
}

 noname_d1Each(3,new int[][]{new int[]{1,2},new int[]{3,4,5}});  int[][]
```

The principle is the same, but we must make a unique name for the generated Dfn, as C# (unfortunately) does not permit local functions. In real code, the Dfn would very likely be local, and would be prefixed with the name of its caller.

Oh, I nearly forgot. Now we have dynamic functions, I just cheat with *composition* and pretend it was a dfn all along:

```
       cc.chk '3∘↑¨(1 2)(3 4 5)'
        3      ∘      ↑      ¨      ⊣(1 2) (3 4 5)⊢
  MARK  NOUN   CONJ   VERB   ADVB   NOUN

  int[] noname_d1(int[] omega) { return AE.Take(3,omega); }
```

... and then everything carries on as before. Notice that I have followed the J rules here, and am treating *compose* as CONJ rather than ADVB – I think it probably is something 'special' in the syntax, rather than just another operator.

Multadic Functions

Every so often, you paint yourself well into a corner with a series of perfectly reasonable decisions, all of which seem fine at the time. Remembering that C# is strictly scoped, we have a cast-iron rule that a function cannot see local variables in its caller. Fine, we just need to be sure to pass in everything as arguments, which is good APL practice. The snag is that APL has only one way to do this, which is to construct a nested array from the argument set, pass this in as a single item, and then unpick it inside the called function.

"Oh bother", said Pooh. We just took the decision to support arrays of arrays which are homogeneous all the way down (C# types like int[][]) and to avoid the overhead of the general nested structure, which C# represents as object[] and which implies a lot of runtime expense in recovering (and casting) the contents.

When in doubt, cheat! Here is the dodge we use to get around this particular fix:

```
      ∇ msg←multadic arg;first;second;third
[1]    ⍝ Sample multi-argument function
[2]    ⍝:Args first,second,third
[3]    ⍝: double first
[4]    ⍝: double second
[5]    ⍝: string third,msg
[6]
[7]     (first second third)←arg ⋄ first←1⌈first
[8]
[9]     msg←'Third argument was ',third
      ∇
```

... which is converted to C# as:

```
string multadic(double first,double second,string third)
{
  string msg;  // Result
  first = AE.Max(1,first);
  msg = "Third argument was " + third;
  return msg;
}
```

... and called appropriately:

```
       cc.chk 'multadic 12.3 23.4 ''Hello'' '
        multadic  ⊣12.3 23.4 'Hello'⊢
  MARK  VERB      NOUN
  multadic(12.3,23.4,"Hello");  string
```

... which is why the word-formation stage is very concerned to flag up strands. If the parser finds itself formatting a strand as part of a function argument, it wraps it in parentheses and separates the elements with commas. When a function with an *Args* comment is processed, the header is built from this comment, and the (hopefully obvious) strand assignment is eliminated from the generated code. Note the difference from:

```
       cc.chk 'qqq←12.3 23.4 ''Hello'' '
        qqq   ←     ⊣12.3 23.4 'Hello'⊢
  MARK  NOUN  ASGN  NOUN
  qqq = new object[] {12.3,23.4,"Hello"};  object[]
```

So, you can have nilads, monads, dyads and multiads in this strange new world. What you cannot have is a function with a complex left argument as well as a complex right argument. I am sure I could cope with these if I had to, but we have to draw the line somewhere.

Structures and Simple Classes

From this point, the way the code-conversion works is quite specific to Dyalog version 10. Everything else could run in Dyalog, APL+Win or APLX with very little modification. What we have discovered along the way is that there are quite a few places where the ability to instance namespaces (and pass references as function arguments) is extremely helpful in migrating towards a pure C# coding style. A good example is the *StringBuilder* class, which comes free with C# and is extremely useful when building up longish strings (like the SVG representation of a towerchart) efficiently. To use it, you simply create a new instance, run its *Append* method as often as you want, and eventually collect the result by running its *ToString* method.

Something very similar in APL could look like:

```
      ∇ str←sbt;fluff;nl;word
[1]    ⍝ Give the StringBuilder a whirl
[2]    ⍝:Public
[3]    ⍝: string str,word
[4]     nl←⎕AV[4]
[5]     fluff←new #.StringBuilder
[6]     fluff.Append'Hello, world?',nl
[7]     fluff.Append'Wow!',nl
[8]     fluff.Append'Farewell, cruel world.',nl
[9]    ⍝ Check string substitution
[10]    fluff.Replace'world' 'Universe'
[11]   ⍝ Return completed string
[12]    str←fluff.ToString
      ∇
```

Of course, we could use the Dyalog-10 capability of talking to .Net to get straight to the real thing, but somehow this feels like overkill. Why not just make a `StringBuilder` namespace with a couple of utterly trivial functions called `Append` and `ToString` that do the same job? Then we need a mildly hairy function called `new` to get a true instance of the namespace. This does some pretty horrid stuff with `⎕NS` to copy all the functions and variables, and then it runs the constructor, if present. The constructor is defined to be any function with the same name as the class. In C#, we have exactly this behaviour already available via the *new* keyword, so when I run the code-generator on this function:

```
public string sbt()
{
  string str;  // Result
  char         nl;
  string       word;
  StringBuilder fluff;
```

```
  nl = '\n';
  fluff = new StringBuilder();
  fluff.Append("Hello, world?" + nl);
  fluff.Append("Wow!" + nl);
  fluff.Append("Farewell, cruel world." + nl);
 // Check string substitution
  fluff.Replace("world","Universe");
 // Return completed string
  str = fluff.ToString();
  return str;
}
```

... it looks remarkably similar, and behaves in exactly the same way. Of course you can use the same trick to define little structures (the legend definition in *RainPro* is a good example) as namespaces, and include these in the generated DLL. It seems moderately efficient in the source APL, and certainly moves one some way towards an object-oriented approach. Definitely worth doing.

Putting it all Together - some Benchmarks

OK, you can convert a line of code, so you can convert a function, so you can convert a whole namespace full of functions, and call it a class. Alternatively you can just take a few functions at random, write these to your source file and call these a class for testing purposes. Here is the way I am doing this at the moment:

```
'd:\tools\aplc\bench' Make csΔbench    ⍝ The Eratos benchmark
Bench - done header ...
(#.Eratos)(#.Looper)
All Done.
1303 bytes written to d:\tools\aplc\bench.cs
```

... where `csΔbench` is mostly just raw C# stuff with a few 'special' lines:

```
using System;
using AE=Causeway.SharpArrays;

public class MainClass
{
  static public void Main()
  {
    Bench test = new Bench();
    Console.WriteLine("Off we go");
    Console.WriteLine((test.Eratos(24000)).Length);   // 1 sec
    Console.WriteLine(test.Looper(160000000));        // 1 sec
  }
}

!Bench=Eratos,Looper   ⍝ Benchmarks, assorted
```

The last line creates a C# class called *Bench* out of the functions listed after the '=' sign. Here is the complete C# source which it produces:

```
using System;
using AE=Causeway.SharpArrays;

public class MainClass
{
  static public void Main()
  {
    Bench test = new Bench();
    Console.WriteLine("Off we go");
    Console.WriteLine((test.Eratos(24000)).Length);  // approx 1 sec
    Console.WriteLine(test.Looper(160000000));       // approx 1 sec
  }
}

public class Bench
{
 // ============ Eratos ============
 // Returns primes up to n by the sieve of Eratosthenes

  public int[] Eratos(int max)
  {
    int[] primes;  // Result
    int   check;
    int[] candidates;

    primes = AE.Ravel(1);
    candidates = AE.Drop(1,AE.Range(max));
    while (0 < candidates.Length) {
      primes = AE.Join(primes,check = candidates[0]);
      candidates = AE.Compress(AE.LT(0,AE.Residue(check,candidates)),candidates);
    }
    return primes;
  }

 // ============ Looper ============
 // Simple looper to test speed when we do very little!

  public string Looper(int n)
  {
    string r;  // Result
    int ct;
    ct = 0;
    while (ct < n) {
      ct = ct + 1;
    }
    r = "Done looping";
    return r;
  }

}  // End of class Bench
```

The *Looper* example copies exactly the 'daft benchmark' which the STSC guys used back in 1982 to put an upper bound on the possible speedup of APL code. They got a factor of 350 (compiling to 370 Assembler) which I was hoping to get close

to. In fact we do rather better - to get this to run for 1sec in C# I have to loop 160,000,000 times and (yes, I did try it) this takes a little over 17min in Dyalog APL, which gives me a factor of around 950. The *Eratos* example is very comparable with the Dyalog version, so your 'typical' commercial APL will be somewhere in between. I think we are getting about ×12 on the chart viewers (which basically convert PostScript macros into XML strings) and maybe ×3 on the graph-generation (the `ch` namespace) which is playing more to APL's strengths. A simple simulation (like the *Monty Hall* problem in the next *Vector*) would probably get you a factor of around 100 as it is mostly interpreter overhead with very little exploitation of the array stuff.

Wrap Up

Given a lightweight library (currently 504K, and unlikely to grow much beyond that) of APL-like functions, converting an APL namespace to a C# class turns out to be rather easier than expected. So far, our experience has been that the older (and hence simpler) the APL code, the easier the job becomes.

A major exception to this rule would be an application written in pure **D**fns, which could be handled very cleanly, and rather more completely than an application written in traditional APL style. For example, **D** has just one control-structure (the guard) and one method of error-trapping (the error-guard) both of which convert very simply to C#. **D** has strictly lexical scope, which is much closer to the C# model, and **D** programmers take a pride in avoiding re-use of local variables. Ambivalent functions are easy to spot syntactically, and should be easy to create overloads for. Of course a **D** compiler should be written in **D**, and then (naturally) used to compile itself. Maybe someone with time on their hands would like to give it a try.

Background Reading

[1] Kenneth E. Iverson, *Formalism in Programming Languages*, in *A Sourcebook of APL*, APL Press 1981

[2] Clark Wiedmann, *Steps Toward an APL Compiler*, APL79 Conference Proceedings (APL Quote Quad Vol.9 No.4 June 1979)

[3] Philip R Chastney, *The Hunting of the Snark*, APL82 Conference Proceedings (APL QQ Vol.11 No.1 Sept 1982).

[4] Clark Wiedmann, *A Performance Comparison between an APL Interpreter and Compiler*, APL83 Conference Proceedings (APL QQ Vol.13 No.3 March 1983).

[5] Timothy A Budd, *An APL Compiler for the UNIX Timesharing System*, APL83 Conference Proceedings (APL QQ Vol.13 No.3 March 1983).

[6] J Philip Benkard, *Valence and Precedence in APL Extensions*, APL83 Conference Proceedings (APL QQ Vol.13 No.3 March 1983).

Appendix: Current Restrictions and Known Issues

Inlined dfns must not span multiple lines of the calling function.

Arrays are simple and homogeneous or genuinely nested, not heterogeneous.

Primitives are limited to depth 2 (we only allow string[][] and int[][] and double[][]). Deeper arrays are allowed, but must be handled with each or loops. The only primitives supported on arbitrary arrays are Pick, Index, Reshape at present. Probably some others like Compress, Drop, Mix, Split will be added as required. Scalar primitives do not pervade deep arrays. Probably you will always need 'each' here.

You need to be careful about rank. Singletons are not the same as scalars! Use `⍴mat` not `1⍴mat` in general!

Text constants are treated as strings, unless they only have one character. If you want to get 'inside' a string, you must explode it into a char[] which is done with ravel (`,`)

```
mycharvec←,'Hello, world'
```

To turn it back into a string, you must format it! In general, strings are treated as scalar 'atoms' so `3⍴'Hello'` is an array of 3 strings "Hello" "Hello" "Hello" and so on. The only exception is indexing, which looks inside and returns the selected characters as a char[] rather than as a string. This adds a few annoyances, but generates much nicer looking C# code which probably runs faster too.

Indexed assignment into arrays can cause strange side-effects. Arrays are passed by reference in normal C# fashion, so calling function `foo` and passing `myarray` ...

```
foo myarray
myarray[2] ← 12
```

will overwrite the 2nd element of `myarray` IN THE CALLING FUNCTION. Be careful!! If you really want to do this, make sure to clone the array first, for example

```
newarray ← array[]
```

Then mess with your new array. Similarly ...

```
vec2 ← vec1
vec1[1] +← 1
```

will increment `vec2[1]` as `vec2` is really only a pointer. Use `vec2 ← vec1[]` to be sure you make a copy.

C# keywords (like while, new, int, static) should be avoided in variable/function names.

The only Goto forms are:

Goto Lab or GoTo 0
Goto(expression){compress | take | drop}Lab
Goto Lab <slash><commute> expr

No execute (but you can use ⎕FI to get numbers from text chars)

No ⎕ELX or ⎕ALX

Only simple `:Trap 0 ... :else ... :EndTrap` is supported.

Dyalog ⎕SIGNAL ignores the number, so behaves like ⎕ERROR in APL+Win

No ⎕FMT (use ⍙PF or write your own, or provide C# alternative expression)

No ⎕FX or ⎕DEF

No ⎕VFI - use ⍙FI and ⍙VI in Dyalog APL to mimic ⎕VI and ⎕FI

No dyadic thorn. Use ⍙PF to pass a C# 'picture' on the left instead.

Residue - implemented with the C# definition so is the same as APL for +ve arguments but will give different results for -ve arguments. Be careful!

For now, all system variables are read-only:

⎕TCNL	Newline
⎕AV[4]	ditto
⎕TCLF	Linefeed
⎕IO	1 always
⎕ML	3 always (matching APL2 and APL+Win)
⎕PP	PrintPrecision - machine default
⎕PW	PrintWidth - irrelevant
⎕CT	ComparisonTolerance is machine precision, always
⎕RL	RandomLink is meaningless (real random numbers only)
⎕AV	AtomicVector - not supported yet - should be avoided

so ⎕IO is fixed at 1 here. It could easily be 0 but will never be switchable.

Ambivalent functions are not supported, as ⎕NC makes no sense here.

No ⎕SHADOW (localisation must be determined at the function level)

No user-defined operators.

Arrays are limited to 0 1 2 dimensions (so we do scalars, vectors, matrices) only. Dyadic transpose is therefore not supported at all.

Reshape takes a SCALAR left arg to create a vector result. Passing a length-1 vector will always be an error, as it will attempt to make a matrix of rank-1. For symmetry the shape of a vector is a scalar. You cannot take the shape of a scalar. Rank = `⍴⍴` is special-cased as a primitive, and works as you would expect.

Take and Drop only have singleton left arguments for now. To work with matrices you must use `2↑[1]mat` here. Accidental enhancement is `2↑mat` which works on the first axis by default in true J,A+ style.

There are limited options to the left of assignment viz

```
⎕←stuff                 ⍝ The only form of console output
var←stuff
arr[this;that]←stuff
arr[this;]←stuff
(msk/arr)←stuff
((expr)⊃arr)←stuff
(one two three)←stuff   ⍝ Parentheses are required here
```

Strand assignment *is* allowed – but you can declare a multi-argument function which eliminates most of these. The resulting strand (usually line-1) is simply eliminated from the generated code. Strands of names and constants seem OK, but avoid if possible as there may be all sorts of odd edge effects when tokenising these. Session echo must be explicit with assignment to ⎕, even at the left-hand end of a line.

If and *AndIf* must be on adjacent lines. No support for

```
:If summat
   do this
:AndIf summatelse
   do this as well
:Else
   do that
:End
```

which works in Dyalog (but not +Win, I think). No support for double-testing:

```
:While (cond1)
  stuff
:Until (cond2)
```

all other structures are OK, but note that Select/Case is limited to integers and strings and the *case* clauses must all be constant expressions.