Migration From APL2 to APL/W

by Rex Swain

Introduction

I've recently spent a fair amount of time migrating workspaces from IBM's APL2 to Dyadic System's APL/W. This article, a review of various features that will require conversion, is an effort to help anyone else who is either contemplating or actually charged with a similar migration task.

Of course, all us techies know that what *really* should be done is a complete re-design and re-coding of the entire application! Segregate the logic, user interface, and database stuff, create a proper event-driven GUI, and so on. But sometimes project funding realities require compromise, typically spelled p-o-r-t.

APL is remarkably portable in general. But if you are considering an APL2 to APL/W migration, there are many issues that will slow down what you may have thought would be a quick job. Hopefully the advice here will enable you to anticipate the full scope of a migration effort, save you a bit of work, and help you react calmly to errors in code that used to run just fine.

I should point out that Dyalog APL doesn't purport to be identical to APL2, and considering the circumstances, it is surprisingly similar. Dyalog was based on STSC's NARS version of the language. APL2 (even the pre-release IUP version) wasn't released until they had fixed the language specification and coded most of the product. It is Dyadic's intention to develop closer compatibility with APL2, and they have already made good progress.

My specific experience involved moving from APL2 version 2.2 under VM/CMS (sometimes called "APL2/370") to Dyalog APL/W version 7.1 under Windows for Workgroups, but I believe that the bulk of these issues would be relevant with other releases.

Types of Differences

There are several classes of differences that will require changes:

1. Many differences are easy to find with a simple workspace searching tool. For example, `⎕TF` is not supported in APL/W, so scan all functions for any

occurrence of `⎕TF`, and replace with an emulation function. In this case, you can find *every* instance without ever actually executing the application code.

2. Some differences require a more sophisticated search that may not find 100% of the cases. E.g., optional left arguments must be enclosed in braces in APL/W function headers. So search all dyadic functions for `⎕NC` of the left argument, and where found, wrap the argument in braces. This will *almost* always suffice, but one can easily imagine tricky cases where the argument is never referenced in any obvious way.

3. Some differences are very difficult to find mechanically. E.g., `1 0 1/¨⍳⍴⊂⍳3` works in APL2 but causes a *LENGTH ERROR* in APL/W; you need to enclose the left argument to get this working.

The good news about these first three classes is that they blow up with errors when executed, so even if you can't find them all programatically, at least if you exercise your application long enough they will announce themselves. The bad news, of course, is that this assumes that your application is straightforward enough that you *can* test all possible execution paths and data cases. (More probably, your end-users will eventually become conversion beta testers.)

Unfortunately, there are two more classes...

4. Some differences are a nightmare! E.g., the spacing of results from monadic format may be slightly different in APL/W, depending on the depth and structure of its argument. In the workspaces I worked on, there seemed to be 1E6 uses of monadic format, and of course the structure of the argument is not always immediately obvious. These are the worst kind of differences, because the code executes just fine and the result is only subtly different, and the difference may or may not be significant in the context of your application.

5. Finally, there is all the operating system dependent stuff. Some of this can be *extremely* difficult (or impossible) to emulate, and may require you to implement a totally different (but more appropriate) solution.

Following is your migration task list...

Getting Started

- **Transfer your workspaces** from APL2 to APL/W. Dyadic supplies a workspace `WDYALOG\WS\APL2IN` that helps to read APL2 transfer files created by `)OUT`. I wrote an enhanced version which Dyadic distributes in workspace `WDYALOG\OUTPRODS\TOOLS\APL2IN2`.

- Set APL/W's **migration level** with `⎕ML←3`. This will immediately solve several compatibility issues:

`Z←∈R`	Enlist (not Type)
`Z←↑R`	First (not Mix)
`Z←⊃R`	Mix (not First)
`Z←≡R`	Absolute value of depth
`Z←L⊃R`	Partitioned enclose
`Z←⎕TC`	Order of terminal control characters

- Check for dependence on any **APL2 invocation options**, especially:

`DATEFORM`	Format for timestamps
`DEBUG`	Suppress `⎕LX`, etc.
`INPUT`	Queue input strings
`QUIET`	Suppress output
`RUN`	Auto-invoke function via `⎕NA`
`TERMCODE(-1)`	Controlled invocation

Character Sets

- Beware of the **atomic vector** (any reference to `⎕AV`)! The order of characters is *very* different.

- Remember that the **EBCDIC and ANSI** character sets are quite different. Even characters that seem straightforward may not be — single quotes, vertical bars, and exclamation points should be examined carefully. Watch out for national language characters and currency symbols.

  Also remember that characters read from PC files into APL/W may pass through Dyalog's `APLT=WIN.DOT` translation mechanism. So you really have at least three character sets to worry about: EBCDIC, ANSI and `⎕AV`.

- Watch out for four **overstruck APL2 symbols** not present in APL/W's `⎕AV`:

  ⍥ ⍁ ⍤ ⍨

  Luckily, they don't actually do anything in APL2, so they probably won't matter much.

Different Syntax

- Wrap braces around **optional left arguments**. For example, if in APL2 you had:

```
      ∇ R←A FOO B
[1]     →(0≠⎕NC 'A')⍴L1   ⍝ Left arg supplied?
[2]     A←⍳0              ⍝ Default left arg
[3]    L1:
```

In APL/W you need to change to header to:

```
      ∇ R←{A}FOO B
```

I was able to automatically convert 99% of these cases by writing a workspace searching tool that inspects all dyadic functions for `⎕NC'leftarg'` and makes the appropriate change in the header.

- You cannot **assign system variables** in APL/W. So if your APL2 functions use them as a sink (e.g., `⎕WA←`) you will have to convert. I never liked this technique anyway; I always define a "no-op" function called `SINK`:

```
      ∇ SINK A
[1]    ⍝ Throw away argument
      ∇
```

Then globally change all `'⎕WA←'` and `'⎕TS←'` and etc. to `'SINK '`.

Or, you may be able to exploit APL/W's "shy" explicit result feature — if you find `⎕WA←FOO X`, you can change `FOO`'s header to `∇{R}←FOO A` and then just execute `FOO X`.

- **Trace and stop** controls are different. Use APL/W's `⎕TRACE'FOO'` and `⎕STOP'FOO'` rather than APL2's `T∆FOO←` and `S∆FOO←`. (Note that APL/W also has a much fancier interactive trace facility; see Trace on the Action menu.)

Same Syntax, Limited Capability

- Some complex forms of **selective assignment** are not permitted in APL/W. For example, with a 3 by 4 matrix `M`, something like:

```
(1 0 1/M[;4])←0
```

works in APL2 but generates a `DOMAIN ERROR` in APL/W. You will have to re-code this as:

```
Q←M[;4] ⋄ (1 0 1/Q)←0 ⋄ M[;4]←Q
```

Similarly, `(A⊃B)[I]←X` does not work in APL/W.

- Some APL/W primitive functions generate errors when **reduction is applied to an empty array**. For example, in the case of `,/⍳0` APL2 returns `⊂⍳0` whereas APL/W signals a *DOMAIN ERROR*.

- Does your code use *⎕DL* to **delay**? APL/W does not permit fractional arguments, so something like `⎕DL 1.5` will generate a *DOMAIN ERROR*.

Same Syntax, But Works Differently

- Watch out for the **rank of the result of some system functions**. In APL2, *⎕NC* `'A'` returns a scalar, but APL/W returns a one-element vector. This can escalate into a depth problem if you execute *⎕NC¨A*. Monadic *⎕SVO* also exhibits this behavior.

- **Compress-each** is interpreted differently. For example, in APL2:

```
      1 0 1/¨V←4⍴⊂⍳3
1 3  1 3  1 3  1 3
```

  With APL/W, you will need to enclose the compression vector:

```
      1 0 1/¨V
LENGTH ERROR
      (⊂1 0 1)/¨V
1 3  1 3  1 3  1 3
```

- **Strand indexing** is interpreted differently. For instance:

```
X Y Z[I]
```

  In APL2, the *I* indexes just *Z*, but in APL/W, the *I* indexes the three-item vector `(X Y Z)`. So for APL/W you will have to convert this to:

```
X Y (Z[I])
```

- **Strand assignment** of an enclosed scalar is treated differently:

```
(X Y)←⊂'ABC'
```

  is treated as:

```
X←Y←'ABC'    ⍝ APL2
X←Y←⊂'ABC'   ⍝ APL/W
```

- APL/W's **name class** does not like system functions and variables. In APL2, `⎕NC '⎕AV'` reports `2` and `⎕NC '⎕CR'` reports `3`, whereas APL/W reports `¯1` in both cases.

  Also, in APL/W it is possible for `⎕NC` to return a `9`. So watch out if you have tools that do something like `Z←'IULVFO'[¯1 0 1 2 3 4⍳⎕NC A]`.

- Beware of **applying a user-defined function to an empty array with each**. In APL2, if you execute `Z←FOO¨⍳0`, `FOO` is never actually executed, and the result `Z` is based on the prototype of the `FOO`'s argument. In APL/W, `FOO` is executed once with an argument based on the prototype of `FOO`'s argument, and the result `Z` is based on the prototype of `FOO`'s result `R`.

```
      ∇ R←FOO A
[1]     ⎕←'Arg is:' A
[2]     R←2 4 6
      ∇

       DISPLAY Z←FOO¨⍳0   ⍝ APL/W
 Arg is:  0
.⊖--------.
| .→----. |
| |0 0 0| |
| '~----' |
'∈--------'

       DISPLAY Z←FOO¨⍳0   ⍝ APL2
.⊖.
|0|
'~'
```

  This is a nasty one to detect in advance. Your function may not be prepared to handle a zero or empty argument.

- When **monadic format** is applied to a nested array, the spacing of the result is sometimes different. I first noticed this in mainframe code that composed short messages with format, such as (using ∘ to represent blanks):

```
      ⎕←⍕'FOUND' 9 'DOCUMENTS'
∘FOUND∘∘9∘∘DOCUMENTS∘     ⍝ APL/W
∘FOUND∘9∘DOCUMENTS∘       ⍝ APL2
```

  There are also subtle differences when some more complex nested arrays are formatted. So if your code relies on a certain number of blanks, beware.

Emulation (or Re-Coding) Required

- **Format-by-example** is not available in APL/W. So every time you see something like `'550.03333%'⍕A` you have some work to do! (And don't forget that in APL2, `⎕FC[1]` might be sneaky and change the decimal point to some other character!)

- The **index function** ⌷ (sometimes called "squish-quad") is not supported in APL/W. Dyalog APL does have plenty of ways to do indexing, so conversion should not be too much of a problem.

- APL/W does not support **n-wise reduction** (as in `2+/A`). The 2-line user-defined operator `NWISE` that Dyadic supplies in workspace `WDYALOG\WS\OPS` can help to emulate this feature. I wrote a more complete version that is also somewhat less prone to `WS FULL`.[1]

- **Scalar functions over an axis** are not supported in APL/W, so if your APL2 code does things like `MATRIX+[2]VECTOR` you will have to change them. I use a user-defined operator called `AXIS`.[1]

- Write emulation functions for missing APL2 **system functions**, as required:

`⎕AF`	Atomic Function
`⎕AT`	Attributes[2]
`⎕EA`	Execute Alternate
`⎕EC`	Execute Controlled
`⎕ES`	Event Simulation
`⎕FX`	Fix (dyadic: with execution properties)
`⎕TF`	Transfer Format
`⎕UCS`	Universal Character Set

- Write emulation functions for missing APL2 **system variables**, as required:

`⎕EM`	Event Message
`⎕ET`	Event Type
`⎕FC`	Format Control
`⎕L`	Left argument
`⎕NLT`	National Language Translation
`⎕PR`	Prompt Replacement
`⎕R`	Right argument
`⎕TZ`	Time Zone
`⎕UL`	User Load

- **Name associate** is very different. Write emulation functions for anything involving ⎕*NA*, including:

`10 ⎕NA`	REXX
`11 ⎕NA`	Access external routines
`12 ⎕NA`	Files as arrays

- Write emulation functions for APL2 **external supplied routines**, including:

`∆FM, ∆FV, ∆F`	File read/write/query
`∆EXEC`	Execute REXX
`ATR, CTN, CAN, DAN`	Data conversion routines
`EXP, PACKAGE, QNS`	Namespaces and name scopes[3]
`IN, OUT`	Like)IN and)OUT
`EDITOR2, EDITORX`	Interface to editors

Development Environment

- The APL/W **session manager** is totally different. Inspect your workspaces for use of APL2's AP120.

- The APL/W **function editor** is totally different, so it's time for you to learn yet another editor. (At least it supports cut and paste!)

System Commands

- Note that APL/W does not support some APL2 **system commands**:

`)CHECK`	`)EDITOR`	`)HOST`	`)IN`
`)MCOPY`	`)MORE`	`)NMS`	`)OUT`
`)PBS`	`)PIN`	`)QUOTA`	`)SIC`
`)SIS`	`)SYMBOLS`		

- Also note that some APL2 **system command arguments** are not supported in APL/W:

  `)FNS`, `)VARS`, and `)OPS` do not accept a range of names
  `)RESET` does not allow an argument

- Are you using APL2 utilities that **execute system commands via the stack** and capture their "results"? If so, you'll need a different technique. APL/W generally makes this easy — it has many more matching system functions that return explicit results (like ⎕*WSID*, which is inexplicably missing in APL2).

External Communication

- Anything involving **shared variables** will probably need conversion.
- Anything involving **auxiliary processors** will probably need conversion.

The Operating System

- Don't forget all **operating system dependent non-APL facilities**, such as:

  The CMS stack
  System command language (REXX)
  System editor (XEDIT)
  Document composition (DCF/Script)
  CMS Pipelines
  System file I/O (flat files, etc.)
  Other data access methods (VSAM, etc.)
  Database (SQL)
  Graphics (GDDM)
  VM Backup/Archive
  VM Schedule

Network Communications

- Review your application's use of your **mainframe communications network**. Will remote users be able to access the new system through the same network? Can you dial in from home? Are you sending jobs to an MVS system?

User Interface

- And, last but not least, there's the **user interface**! The procedural vs. event-driven issue is a huge topic in its own right. Suffice to say that unless you want to do extensive re-coding, you will probably be forced to make some compromises here.

Conclusion

This list is, of course, not necessarily complete — these are just the problems I have run into so far, and I'm sure there are more lurking.

It would be a mistake to come away from reading this with the impression that Dyalog APL/W is missing a lot of APL features — in fact, it's just missing some *APL2* features, and it has many compensating and additional features that APL2

does not have. I mean for all this to be free of value judgments — I'll leave it to you to decide whether one system does things "better" than the other.

18 March 1996

Rex Swain
Independent Consultant
8 South Street
Washington, CT 06793
USA

Tel: (+1) 860-868-0131
Fax: (+1) 860-868-9970
Email: rex@rexswain.com
WWW: http://www.rexswain.com

---

[1] The user-defined operators *NWISE* and *AXIS* may be downloaded from the "Dyalog APL/W Tools and Utilities" section of my WWW home page.

[2] Dyadic is adding □*AT* to APL/W versions 7.2 and 8, so function timestamps should be available by the time you read this.

[3] For more details, see my paper "Namespaces: APL/W vs. APL2" in the APL95 conference proceedings.