APL# - an APL for Microsoft.Net

Morten Kromberg, Jonathan Manktelow, John Scholes
Dyalog Ltd
Minchens Court, Minchens Lane, Bramley, RG26 5BH, United Kingdom
aplsharp@dyalog.com

State of the art of array processing languages

Abstract

The paper discusses the design of APL# (pronounced "APL Sharp"), a new dialect of APL designed with object oriented / language-agnostic virtual machine platforms in mind, using Microsoft.NET as the initial target.

Microsoft.Net is a software platform which is based on Microsoft's Common Language Runtime (CLR). Together with Sun's Java Virtual Machine (JVM), the CLR is a popular member of a (relatively) new family of "application virtual machines" (AVMs) which are rapidly growing in importance as platforms for hardware-independent software development. These virtual machines provide high level components such as memory management, exception handling and other services which significantly simplify the task of implementing new programming languages. They also facilitate communication between modules written in different languages by providing a single memory manager and type system which is shared by every component which is built upon them. As the name suggests, the CLR was designed in co-operation with a diverse group of language implementers, and the services that it provides are designed to be "language agnostic".

APL# will also run under Microsoft SilverLight [SilverLight], a web browser plug-in which competes with products like Adobe Flash as a platform for "rich client" web applications. In fact, SilverLight is a cut-down version of the Microsoft.Net framework, capable of running many .Net applications. SilverLight can be installed as a plug-in for most popular web browsers, under Microsoft Windows and Apple's Mac OS X. The list of short-term target platforms also includes Mono and MoonLight [Mono], which are open-source versions of the Microsoft.Net Framework and SilverLight, respectively – available under Linux and a growing number of other operating systems. There is reason to expect that SilverLight and Moonlight will achieve almost universal availability over the next few years, and that it will be possible to deploy applications written in APL# – including "rich" user interfaces if required - on virtually any platform, possibly including the majority of mobile devices.

Motivation

Virtual Machines provide two very important benefits: Firstly, applications become (at least theoretically) universally portable, because the VM can be implemented on many different types of hardware. Modern AVMs include large libraries of high-level components, which make the development of cross-platform applications easier than it has ever been before.

Secondly (but perhaps most significantly in the short term), the modern AVMs provide a secure environment, where the access that an application has to both local and remote resources can be strictly controlled. Such applications are said to be *managed*, or *safe*. Managed applications can be safely distributed across the internet, because it is possible to guarantee that they will not read or write data on the client computer onto which they have been downloaded – and that internet access is restricted to communication with the server from which they were downloaded. From a security perspective, the application can be considered to be running on the host computer, although it is making use of local resources for both presentation and data manipulation.

Dyalog APL has had a bridge to the Microsoft.Net framework since 2002, and this interface has allowed APL developers to successfully tap in to many of the tools that the framework provides. However, there are a few glitches and discontinuities where the memory management model and type system of Dyalog APL conflict with the CLR. The design of APL# will resolve these conflicts, producing a language system which is “native” to the CLR.

The immediate goal of the APL# project is to make the benefits of safe/managed computing available to users of APL. With APL#, it will be possible to run the same APL application on a variety of client platforms, simply by directing users to a web page from which the software is retrieved and run – and it will be possible to do this with the blessings of most security departments. No actual “installation” of software is required on the client computer (the application is merely “cached”, in order to make it start faster the next time).

Freedom from Upwards Compatibility

Most APL interpreters request a single, large quantity of memory (the *workspace*) from the operating system, and manage user data and the APL execution stack without interference from the outside world. On an AVM, we will be sharing the program stack and memory manager with other programming languages, and this more or less dictates some significant changes in APL language semantics. In addition, most of the system functions that provide interfaces to Application Program Interfaces (APIs) will be abandoned in favour of tools provided by the AVM. For the first time in the history of the team behind Dyalog APL, we are designing an APL interpreter which does not need to run the same application code as the previous version[1], and we have taken advantage of this opportunity to take a fresh look at the language.

It is our hope that we will be able to take advantage of some of the work that has been done inside the array language community in languages like J, and combine this with the improved understanding of functional and dynamic languages that has emerged in recent decades. Our goal is to create a more competitive dialect of APL, that will be easier to market to a new generation of users – both to “domain experts” who have limited experience with software development, and to those “software engineers” who are looking for more dynamic tools. Although APL# must be a “first class citizen” of the Microsoft.Net framework, we want a language which will be portable to other VMs (like the Java VM), and can also be implemented on “real” computer hardware. In other words, there must not be any elements of the core language which *require* Microsoft.Net.

In addition, APL# is expected to have performance characteristics which – at least initially – will be significantly different from those of Dyalog APL. We expect some things will be much faster; others will be slower, as many of the optimization techniques used by current APL interpreters cannot be translated to a strongly typed AVM memory manager, which does not provide reference counts for allocated memory [MicrosoftGCol]. We (or the APL application developers) will need to find new ways to write fast code. As a result, Dyalog will not be recommending that customers move quickly to translate existing applications to APL#. We think of APL# as a companion to traditional APL, aimed at new applications which can benefit from being native to an AVM – or as components of multi-tier applications which use APL# together with other APL systems. We also expect that some of the new language features pioneered in APL# will find their way “back” into Dyalog APL, and that the two languages will grow closer in the decade(s) to come.

[1] Changes have not been introduced lightly; except for missing APIs, we believe that it remains possible to semi-automatically translate the bulk of existing APL algorithms into APL# and intend to build tools to do so.

Challenges

The services provided by the CLR make it relatively easy to implement a new programming language, and the ability to inter-operate with solutions written in other languages makes a wide variety of tools available to the application developer. However, this is a double-edged sword: when the framework is managing a heap ("workspace") containing data which is shared between many programming languages, and also allows the stack to consist of functions written in different programming languages, APL# is not only forced to use the same type system – it should ideally also agree with other languages about what an application stack can contain. Under Microsoft.Net, it will be common for an APL# function to call a function written in a language like C# - which in turn calls another (or the original) APL# function. Unless we want to be faking a lot of things behind the covers, and be a language which has strange discontinuities when debugging applications which use more than one programming language, we need to abandon some of the most central dogma of "classic" APL interpreters:

- The illusion that APL only has two basic data types: numbers and characters: APL# will still treat most numeric types (including Booleans) as a single logical type, but will also support all the data types available on the platform, as elements of an array.
- That arguments are always passed "by value": in APL#, arrays are passed "by reference", as they would be in most other programming languages.
- That user-defined names are global by default, and local variables are visible to all called functions: in APL#, all names *created by* a function are strictly local, unless they are placed within a pre-existing global container.

The challenge is to retain the key advantages of APL, while transforming the language in ways which are dictated by the constraints of an "alien" software platform which is heavily influenced by scalar- and object-oriented compiled languages. Fortunately, the most important language changes, such as the strictly local scope for new names and the availability of arrays which are passed by reference, are not new ideas; they have been proposed by many APL language theorists in the past [Seeds1978]. In fact, variants of these features have become available in Dyalog APL over the past decade or more: local scope is the default for the functional dialect known as D-fns [Scholes1996] – and users of namespaces and other object oriented features of Dyalog APL will be familiar with object references. As a result of our experience with these features, we are reasonably comfortable that their rise to prominence as "mainstream" features of APL# is not going to impair the use of APL# as an effective tool of thought[2].

When designing a programming language for an AVM, there is a strong temptation to adopt characteristics of the languages most closely identified with the VM (Java for the JVM and C# for the CLR). We are not the first team to attempt to design an APL system for Microsoft.Net: in 2003, APL2000 released VisualAPL, an APL system for Microsoft.Net. VisualAPL provides the programmer with the option of using in-line C# syntax as well as APL syntax, and the APL itself has also adopted certain elements of C# syntax and semantics. Although compatibility with C# does have significant benefits, like making it possible to use code samples written for C# directly, the underlying philosophy of C# and APL languages is substantially different, and we feel very uneasy about tackling the design decisions that will arise as a result of "mixing languages together". Although our goal is to *enable* the mixing of APL with other languages, we are determined to do what we can to avoid making it a *requirement* that APL users are familiar with mainstream programming paradigms.

[2] Despite the fact that software engineers would also approve of these changes ☺.

As an example of the difficult decisions that can result from approaching the mainstream closely: VisualAPL has changed the meaning of the symbol `=` to denote assignment *by reference* (as in C# and many other languages). The old APL assignment arrow (`←`) still means assignment *by value* (making a copy). This is an elegant choice, but means that an alternative must be found for the old use of the `=` symbol: VisualAPL uses syntax borrowed from C# (`==`) to denote *exact* comparison, while *tolerant* equality is written using the symbol `≈`, to emphasize the dependence on comparison tolerance. However, there is only one instance of each of the other relational primitives (like `≤` and `≥`), which are written using the normal APL symbols, despite the fact that they *are always* tolerant. The use of two symbols to denote a single function (`==`) also has the side-effect of removing the option of ever assigning a monadic definition to `=`.

The "paradigm conflicts" extend to the underlying implementation: In the first versions of VisualAPL, tolerant equality was defined in terms of an absolute difference between numbers being compared. In APL, comparison tolerance is defined as a relative difference, which has been carefully calibrated to hide the effects of errors introduced by the use of double-precision floating-point arithmetic (a typical default value of `1E¯14` expresses tolerance to accumulated errors in the 14th digit of the 16-17 digits which are available). Recent versions of VisualAPL do implement the relative difference in accordance with the APL standard, but the default value for comparison tolerance has been retained (the Microsoft.Net constant known as `Double.Epsilon` - roughly `4.9E¯324`). This value effectively expresses tolerance to errors beyond the 300th digit of the result, which means that the function called "approximately equals" is "exact" for all intents and purposes when using double-precision floating-point numbers. It is possible to get the result "that one would expect" by using a tolerance similar to that used in other APL systems – but by default, in VisualAPL:

```
                                                                 A←⍳10
         A ≈ (A*0.5)*2 ⍝ Note use of “Approximately Equal” (≈)
1 1 0 0 1 0 0 0 0 1
```

We are also proposing significant changes to APL. In fact, APL# is perhaps more radical than VisualAPL in its departures from certain aspects of traditional APL. Our biggest worry is that we will pick features which "break" (rather than strengthen) the language, as a result of adopting features which are fundamentally at odds with APL. It is easy for a small team to convince itself that each step in a train of choices is elegant, or even "the only logical choice", and still arrive at a result that a majority of users will find unpalatable. We hope that the APL community will provide us with constructive feedback and that we will be able to proceed with confidence after a suitable discussion.

Language Direction

We believe that it is critical to the future growth of the APL language that it has a simple and consistent definition, allowing the user to maintain a small and precise mental model of the language, keeping it a good tool for the description of mathematical identities and processes. We *do also* expect to make (some) software engineers significantly happier with APL# than they were with APL, but - as illustrated in the previous section - we feel that we need to pick our way very carefully in our search for ways in which APL can become a first class citizen of the AVM platforms, without introducing "language-breaking features".

We believe that there is a real opportunity to re-introduce APL as a modern language which comfortably integrates a number of paradigms which are currently gaining respect: Functional, Dynamic and Array-oriented programming – into an Object-Oriented package. We believe that some current users of "functional" and "scripting" languages like F#, Python, Ruby and Perl will find APL to be an attractive alternative for computational and extremely dynamic applications, and that the changes required to attract this market have a relatively low risk of breaking the language.

We have decided to use the Extended APL Standard (ISO/IEC 13751) – henceforth referred to as EAS - as the foundation for the new language – with a few extensions which mostly derive from current Dyalog APL. The syntax for user-defined functions and operators builds upon the work that John Scholes has pioneered at Dyalog, which has resulted in a functional dialect of APL known as D-Fns. D-Fns, with their roots in Ken Iverson's Direct Definition [Iverson1976], provide a notation for functional programming embedded within Dyalog APL. From a humble beginning, D-Fns have grown continuously in popularity among users of Dyalog APL, and have become a significant component of many applications.

Core Language Specification

The language specification is still changing, and this paper is by no means a complete description of the current specification. We are preparing to discuss the specification with users of Dyalog APL, the APL community, and ideally also other programming communities, following the first publication of the specification in conjunction with the APL2010 conference in Berlin. Before the conference begins, we will publish the current language specification (which will still be "work in progress") on the language web page, http://www.aplsharp.com. Although we have started work on an implementation, we are prepared to make significant changes if feedback from the community warrants it.

APL# will adopt Dyalog's definitions of primitive functions and operators, except where these differ from EAS. In other words, APL# will conform to EAS, extended with a number of features that appear in Dyalog APL but not in EAS. This means that the definition of all of the primitive language constructs will be very close to those in Dyalog APL at "migration level 3" (APL# does not have the migration level switch, it always behaves as if `⎕ml←3`). APL# can also be described as a variant of Dyalog APL with more emphasis on functional programming, with a core language which has moved several steps closer to both the EAS *and* to the APL2 language specification.

APL# includes a number of language elements which are either not defined in the EAS, or implemented differently from the EAS. The important differences and extensions are:

- **User-defined functions and operators, control structures and "guard expressions" are specified using a syntax which derives from Dyalog D-Fns.** The most dramatic departure from traditional APL is unquestionably that, within these functions and operators, new names are all *strictly local* unless they are located within containers known as *Spaces*. The details of the new function syntax, which extends D-Fns to encompass "procedural" functions, are the subject of a separate paper at APL2010 [Manktelow2010].

- ***All* arrays are *references*.** The assignment of one name to another name (`B←A`) does not imply the creation of a separate copy of the variable, as it would in traditional APL. If either A or B is modified (changed without being overwritten entirely), the other variable also changes. Arrays are always passed by reference when used as arguments to functions. An explicit Copy function («) must be used to take a separate copy of an array.

- **The vast majority of *system* variables, functions and operators have been eliminated.** In particular, it is proposed that index origin be fixed at 0, and comparison tolerance at `1E¯14`. It is proposed that the Variant operator be used to select variants of primitives. For example, exact comparison can be written as (`=⍠0`), and (`1 2 3≡(⍳⍠1)3`).

- **Additional Primitive operators:** Composition (`∘`) and Power (`⍣`) from Dyalog APL, Rank (`⍤`) from SHARP APL and J, plus the Variant operator (`⍠`), proposed by Ken Iverson in 1978 [Iverson1978] – but using a slightly different symbol. As in Dyalog APL and NARS2000, the symbol / is ambivalent, it is the operator Reduce when a function appears on the left, and the function Replicate when an array is immediately to the left.

- **Parallelism built-in to primitive operators:** The operators each (¨), outer product (∘.ω) and rank (⍤) – and the execution of multiple expressions in using the dot to the right of an array of spaces - will have no pre-defined order of execution, allowing the system to execute them in parallel if it so chooses. Parallel control structures will possibly also be considered.

- **Event handling,** otherwise known as error trapping, **will be different from Dyalog APL** in order to properly support the event handling of the AVM**.**

- **Additional Primitive functions:** Split (monadic ↓), monadic and dyadic Index (⌷) from Dya-

  log APL. New (¤), String ($) and Copy («).

- **Selective specification** is defined **as per Dyalog**, extended with enlist and functions derived from each, as in (∊¨V)←0.

- **Binding strengths as per Dyalog APL.** For example vector (strand) binding will continue to be stronger than operator-operand binding

In addition to the above, APL# is an object oriented language, with the following key features which are not addressed in the EAS:

- **APL# supports the definition of dynamic classes**, which are a relatively new concept in the .Net framework, but have behavior which is very similar to Dyalog APL *namespaces*. As the name suggests, a dynamic object is one into which variables and functions can be injected at runtime. Support for statically typed classes will probably be added after the first release.

- In APL#, the system function **⎕NEW is replaced by the primitive function New (¤)**, which takes a type on the left and constructor arguments on the right. The right (constructor) argument to monadic ¤ is a script which is used to initialize a new Space.

- As in Dyalog APL, structural functions will be able to work with [arrays of] objects supported by the native support for all types supported by the underlying AVM.

  - **Primitive functions will support all relevant numeric .Net types**, including Complex numbers (as Dyalog APL, but covering a few more types).

  - **Double-quotes are used to delimit Strings, which are a new scalar data type.** Strings can be converted to and from character vectors using the String function ($). In other words, (0 = ⍴⍴"Hello") and ('Hello' ≡ $"Hello").

  - **APL# will endeavour to treat any object which is *indexable* as if it were an array.** Under Microsoft.Net, this applies to instances of System.Array and anything which implements the interfaces IList or the “generic” IList<T> (this list may be extended). In current Dyalog APL, it is currently necessary first to apply the monadic Index function (⌷) to turn enumerable or indexable objects into APL arrays.

- **Like Dyalog APL, APL# supports the notion that Arrays are a higher level of organization than Objects [Kromberg2007].** In particular, if `ArrayOfObjs` is an array of objects, then `ArrayOfObjs.PropName` refers to the named property of *each* element of the array, rather than a property of the container itself. The escaped dot (`` `. ``) is used to evaluate names in the context of a container. For example, the number of elements in the container might be found as ``ArrayOfObjs`.Count``.

Several of the above departures are discussed in more detail in the following sections.

User-Defined Functions and Operators

APL# functions and operators are defined using a new syntax which combines the features of D-Fns and T-Fns (traditional APL functions) into a unified whole. The new definition attempts to provide "the best of both worlds", providing a vehicle for both procedural and functional programming, based upon the cleanness of the D-Function style. Both named and unnamed functions (and operators) are supported and, within the bodies, the naming of arguments is also optional. The syntax allows the use of both traditional control structures and D-Fn guards – which can also be used as value-returning expressions.

As an illustration, an example of an APL# function to find the real roots of a quadratic equation is listed below. The header `(a b c)`, which is separated from the body of the function by the → symbol (read "maps to"), specifies that the function takes a three-element vector on the right:

```
roots←{ (a b c) →   ⍝ real roots of quadratic
     d←(b*2)-4×a×c  ⍝ Discriminant
     (:If d<0
         ⍬
      :ElseIf d=0
         -b
      :Else
         -b+¯1 1×d*0.5
      :End)÷2×a     ⍝ Result of if stmt divided by 2×a
}
```

Note the use of a parenthesised if-then-else structure to compute a value which is subsequently used as a left argument to division.

The following is an example of a monadic defined operator which can be identified by the curly braces surrounding the left operand in the header. This operator applies its operand to the leaves of an array.

```
leaf←{ {f} r →  ⍝ Monadic operator, monadic derived function
      0=≡r: f r ⍝ If simple, apply f
      ∇¨r       ⍝ Else recursive application of derived fn
}
```

For more on the "unified" function and operator notation, see [Manktelow2010].

All Names are Strictly Local

One of aspects of the new function and operator syntax which is worthy of separate mention is that names defined within a function (like the variable `d` in the `roots` function) are *strictly* local to the function[3] – not only are they not visible to the calling environment, they are not visible to any functions which are called *by* the function which defines them. Global names must be stored either in the “application root” (`#.TheAnswer←42`) or the “current space” (`⎕this.x←99`)[4] – or indeed within spaces which can be reached from one of those two spaces (`#.TrigConstants.pi←22÷7`). Spaces containing data can of course also be passed as arguments to a function, allowing the creation of new “global” variables within a space:

```
      data←¤'(a b)←1 2' ⍝ or (data←¤'').(a b)←1 2
      {ω.c ← ω.a + ω.b} data
      data.c
3
```

The conversion of traditional APL systems using “semi-global” variables will require the generation of a set of suitable container spaces corresponding to each significant semi-global context, to hold the data (we believe that such a conversion process can be semi-automated based on an analysis of the calling tree of the application – but will undoubtedly require manual intervention).

All Arrays are References

Many of the components that an APL# application would wish to share data with expect arrays to be passed by reference and subsequently shared (the .Net type `System.Array` is a so-called *reference type*). For example, in order to populate a data grid in a user interface with the contents of an array, one would typically pass a reference to the array to a data grid object. Subsequently, changes made by the application to the shared array will be reflected in the data grid, and modifications made by the end user will “automatically” be reflected in the array.

There is no question that APL# needs to support *reference arrays* in order to function as a Microsoft.Net language. This is a significant and possibly “language-breaking” change, and we expect (and welcome) significant debate with members of the APL community. At present, we feel that having two types of function argument and/or two types of APL array would be a significant complication to the mental model of APL#, and thus an impediment to its use as a tool of thought. Given that APL# users *will* need to understand *reference* arrays, the current proposal is therefore that ***all*** arrays be reference arrays - and that a new Copy function (`«`) should be used to explicitly clone data when this is required. Note that `«` is *not* an assignment arrow, it is simply a function which makes a shallow copy of the right argument. It allows assignment by value to be written as the “idiom” `←«`, as in the following examples:

```
                                                         A←'Hallo'
                                                            C←«B←A
                                                  B[1]←'e'
                                                       A         C
Hello Hallo
```

Arrays passed to functions would also (always) be passed by reference; if the function mutates an argument, the change will be propagated to the argument array:

```
      vstar←{v → ((v∊'aeiou')/v)←'*' ◇ v} ⍝ Lower vowels => *
      vstar A
```

[3] They are also visible to any “nested” functions which are defined within the function body.

[4] We are hoping to come up with a suitable glyph to replace `⎕this`.

```
H*ll*
      A
H*ll*
      vstar «C ⍝ Pass copy of C to avoid modification
H*ll*
```

The above is controversial, but it feels preferable to inventing syntax to declare (and detect) two types of arrays and two types of argument passing. It is possible that the function notation should be extended to allow declarations that selected arguments should be copied (for example, by prefixing those names with a « in the header), but we are keen to keep the notation as lean as possible.

During the review process of this paper, we have moved a few steps closer to adding a separate symbol for "mutating assignment" (an assignment that will "overwrite" a complete, causing all references to an existing name to point to a new value), and that we also need to add a merge operator (as in J), in order to provide an indexed assignment mechanism which does not modify the target array, as indexed assignment does.

Index Origin Zero

APL users almost unanimously agree that variable index origin (`⎕IO`) should be abolished. Unfortunately, the agreement does not extend to the choice of *which* value to use for fixed origin. This choice may well be the most controversial issue in the design of the new language. A significant proportion of users seem to agree that "counting starts at one", and that an index origin of 1 is therefore more intuitive. As our goal is to provide an effective tool of thought, this argument does weigh heavily. Unfortunately, computer hardware addressing starts at zero and this has managed to permeate virtually all modern computer languages, even those that claim to be "high level". Starting at zero does lead to slightly simpler algorithms in many cases.

It seems impossible to arrive at the best choice through objective debate. However, the fact that virtually all other languages have chosen 0, and that APL# will share objects with these languages, seems decisive – whether we consider it to be right or wrong. In APL#, an indexing expression like `X[I]` will often not be executed by the APL interpreter[5]: indexing is generally implemented as a *function call* to the component which implements the object `X`, and it will apply *its own* interpretation to the index `I`. This means that the index origin in APL# will not be applied, unless an origin-1 APL# system tries to pre-adjust the index origin in order to "compensate". However, that would require knowledge of the target system index origin, an item of information which is not available. Objects are sometimes indexed by a key rather than a position - if the index is a phone number, then adjusting it would be catastrophic. Thus, in an APL system with an index origin of 1, arrays which happen to have been created by the current APL session could possibly honour the index origin, but other arrays and other types of objects would not – and there would be no reliable way to determine this. Some more research is required into this issue, but at this time, fixing index origin at zero currently seems to be the only way to make indexing a generally predictable operation.

This might seem to be an insurmountable problem when migrating existing algorithms from traditional APL to APL#. However, it may be possible to automate conversions of origin-1 code by translating all instances of `⍳` to `(⍳⍠1)`, and prefixing all indexing expressions by `¯1+`.

[5] In fact, this problem already exists in Dyalog APL: When indexing COM or .Net objects, `⎕IO` has no effect.

Objects

The function New (¤) takes a *type* on the left and a set of constructor arguments on the right. For example:

```
      today←DateTime ¤ 2010 7 30 ⍝ Instance of System.DateTime
      today.DayOfWeek            ⍝ Reference DayOfWeek property
Friday
```

A monadic call to ¤ is the same as a dyadic call with the `AplSpace` type on the left, and creates a “APL Space”, or *Space* for short. A Space is a dynamic object, a container into which any object can be inserted – almost identical to a Dyalog APL *Namespace*. Dynamic objects are a recent addition to the Microsoft.Net framework, added to support dynamic languages like IronPython, and now also supported by recent versions of C# and other statically typed languages.

```
sp1←¤''               ⍝ Create an empty Space
sp1.var←9             ⍝ Define a variable in the space
sp1.(double←{2×⍵})    ⍝ … and a function …
```

When creating a new Space, the constructor argument is a script, which is used to initialize the space. An alternative to the above would be:

```
      sp1←¤'var←9' 'double←{2×⍵}'
      sp1.(double var)
18
```

As in Dyalog APL, the dot notation extends to arrays of objects by evaluating the expression to the right of the dot within the context of each element of the array on the left (for an extensive discussion of the treatment of arrays of objects, see [Kromberg2007]):

```
      ymd←(2010 7 30)(2010 7 31)(2010 8 1)
      ⍴days←DateTime ¤¨ ymd
3
      days.DayOfWeek
Friday Saturday Sunday
      ymd ≡ days.(Year Month Day)
1
```

In APL#, *every array is also an object*, and we need a notation to access properties of the container itself, rather than the properties of the elements of the array. The current proposal is to use an “escaped dot”, as follows (`Count` is a property that will be exposed by APL arrays, for the benefit of other languages that might be presented with data in this form):

```
                                                  days`.Count
3
```

The treatment of containers as described above will extend to objects of all the types that APL# is able to recognize as indexable arrays.

Integration with Microsoft.Net

As previously mentioned, the APL# language specification may not contain any language elements which are directly linked to Microsoft.Net, as it is a requirement that APL# can be ported to a different AVM or to a native implementation. This restriction does not prevent APL# from integrating fully with the .Net framework; APL# will be a fully managed component which can share data with components written in other .Net languages, and can “provide” components which can be used by both dynamic and statically typed languages on the framework. It is not yet clear whether it will be necessary to provide support for defining statically typed classes *in* APL#, or whether the platforms’ growing support for dynamic languages will make this unnecessary.

Like most other APL systems, APL# will not have any reserved words. However, as with Dyalog APL and other .Net languages[6] , it will provide a mechanism for declaring a list of .Net *namespaces* that a program will be using – which can make it appear as if sections of the framework – or indeed selected third party components – have become reserved words in the language. If an expression contains a name which has no current definition, APL# will search the list of “used” namespaces. For example, a statement which refers to the name `DateTime` in an empty space, will find the `System.DateTime` class, because the Microsoft.Net `System` namespace is on the default set of used namespaces. This may create the impression that `DateTime` is a reserved word in APL#, but this is not the case – the name is available for use by the APL# developer. If necessary, the prefix ``` ` ``` can be used to bypass the current namespace and search the list of used spaces immediately. In other words, ``` `DateTime ``` would ignore any local use of the name. To avoid any potential for name conflicts, the use of the ``` ` ``` can be recommended.

Exception handling is an area where name conflicts may be particularly problematic, as the framework specifies exceptions as a hierarchy of types, such as `DivideByZeroException`, which is a subtype of `ArithmeticException` (both to be found in the `System` namespace). Application code will possibly become bound to a particular AVM if these names are used in control structures for event handling, in place of the event numbers used in traditional APL (in current Dyalog APL, event number 90 will catch *all* .Net exceptions). The use of names does seem preferable to numbers, but an open design issue remains whether APL# should contain a set of exception classes which organise .Net exceptions into platform-independent groups, as an alternative to the .Net names.

Development Tools

At this point, our main focus is on the core language design. However, some ideas about the tools that will surround the new language are forming.

- **Development Environment:** Dyalog is developing a new *Remote Integrated Development Environment* (RIDE), which will be a SilverLight/MoonLight application, able to run on just about any desktop or mobile platform. The RIDE is intended to provide all the important functionality of the current Dyalog session, and will be able to connect to and provide a graphical development environment for Dyalog APL and APL# running on any platform.

  There will also be a front-end to the interpreter which is a simple interactive prompt, and at least one which is able to “pipe” input from standard input and output streams. In the longer term, we expect to provide Visual Studio and eventually also front-ends Eclipse and perhaps also editors like EMACS (the separation of the engine from the IDE will allow third parties to develop their own front-ends).

- **Source Code in Unicode Text Files:** The recommended way to store APL# (and Dyalog APL) source code and scripts is to use Unicode Text Files. A packaging utility

[6] Although Dyalog APL is not *managed*, it provides a high degree of interoperability with the Microsoft.Net framework and is considered to be a “.Net Language” by Microsoft.

which will turn a set of scripts into distributable applications in the form of a .Net *exe* file or a SilverLight *xap* file (Visual Studio support will integrate this tool with VS project management).

In the longer term, we also expect to provide ways to serialize the contents of an active APL# session - at least all the APL# arrays and code contained in the session – to provide the equivalent of a traditional "saved workspace". However, it is unlikely that it will become possible to save a workspace which has a stack, since the stack may contain components not under the control of APL#.

- **Utility Libraries for APL:** Although the Microsoft.Net framework provides classes or libraries which we expect will replace most interfaces which are currently implemented as system functions in traditional APL systems, we do anticipate that some interfaces will still need to be "wrapped" in order to be convenient to use from APL. For example, the system function `⎕XML`, which converts XML into arrays, may be sufficiently useful to warrant inclusion as an alternative to the very object oriented interfaces that the framework provides. Wherever possible, such interfaces will be provided as standard .Net libraries, which could be used from other languages than APL.

- **Compiled Code:** The first version of APL# will be interpreted, but in the longer term we expect to be able to compile certain types of applications to "IL".

Interoperability with Dyalog APL

Since APL# is intended as a companion to - rather than a replacement for - Dyalog APL, emphasis will be placed on easy interoperability between the two languages:

- APL# and Dyalog APL will easily be able to call each other in a variety of different ways, since both are Microsoft.Net languages.

- As mentioned in the previous section, the languages will share a common development environment, the RIDE. We expect that it will be common to write applications which use both products, and will endeavour to make the debugging of hybrid Dyalog APL + APL# applications as seamless as possible.

- The component file system and our TCP libraries will be able to exchange APL Arrays between APL# and Dyalog APL – and any other .Net language: we will provide .Net classes to interface to these components.

- Some of the new features of APL# will probably be "back ported" to Dyalog APL in the years to come, with a goal of making it possible to define a subset of the language will be able to move between the platforms with minimal or no conversion.

Conclusion

After more than a year of internal discussion and consultation with a few external advisors, our ideas for a new APL dialect have condensed to the point where we feel that we need to present it to a wider audience to get some feedback. Some of the language features described in this paper may seem alien to current users of APL, but we hope that the reader will bear in mind that the paper is focusing on the differences, and not the similarities, with current APL systems.

The goal is that, despite taking object-orientation and other features of the new virtual machine platforms like the Microsoft.Net framework on board, we will have a language which remains a "pure" APL system; an executable mathematical notation and a Tool of Thought for the exploration of data and algorithms, which enables the dynamic construction of software systems embedding a high degree of domain knowledge. The language must satisfy the requirements of software and engineers with respect to constructing portable, secure applications – but not *require* all APL users to become "experts in IT".

This is perhaps a tall order. However, although they may seem alien at first sight, it is our belief that the new frameworks like Microsoft.Net are fundamentally APL-friendly, and will make it much easier to achieve modes of APL system development which are more similar in flavour to the "open systems" of APLs first "golden age" in the 1970s and 1980s, than the systems built on the difficult APIs from the early GUI years.

As APL approaches its 50th anniversary, our goal is to design an APL system for the next 50 years, and we look forward to a lively debate!

Acknowlegements

Thanks to Joe Blaze of APL2000 for helping us get the facts straight with respect to features of VisualAPL, and to Roger Hui for educating us about the Variant operator and reminding us of many other ideas, his own and others, new and old.

References

[Iverson1976] Direct Definition: Elementary Analysis, APL Press, 1976, Chapter 10, pp. 140-158. http://www.jsoftware.com/papers/DirectDef.htm

[Iverson1978] IBM Research Report RC 7091 (#30399), 1978-04-26, http://www.jsoftware.com/papers/opfns.htm

[Kromberg2007] Arrays of objects. Proceedings of the 2007 symposium on Dynamic Languages.

[Manktelow2010] "Unifying D-Fns and T-Fns in APL#", Manktelow, Kromberg & Scholes, Proceedings APL 2010 LPA - Berlin, 2010.

[MicrosoftGCol] Microsoft.Net Framework 4 Garbage Collection: http://msdn.microsoft.com/en-us/library/0xy59wtx.aspx

[Mono] Mono and Moonlight home page at http://www.mono-project.com

[Scholes1996] "Dynamic Functions in Dyalog APL" www.dyalog.com/download/dfns.pdf

[Seeds1978] Seeds, G.M., A. Arpin, and M. LaBarre, "Name Scope Control in APL Defined Functions", APL Quote-Quad, Volume 8, Number 4, 1978-06, pp. 15-19.

[Silverlight] Home page at http://www.silverlight.net