Unifying T-Fns and D-Fns in APL#

Jonathan Manktelow
Morten Kromberg
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

APL systems provide a definition mechanism so that expressions may be collected into non-primitive or “user-defined” functions and operators.

The traditional APL defined function (T-Fn) , even when extended with control-structures, is procedural in nature, and does nothing to discourage looping, destructive assignment or the use of non-result-returning and niladic “functions”.

In 1996, Dyalog introduced a purer function definition style, now referred to as a “D-Function” (D-Fn), which was designed to fit better with the functional programming paradigm.

This paper details an attempt, in Dyalog's **APL#** project, to combine both the traditional “T-Fn” and direct “D-Fn” definition styles into a unified whole, which supports both the procedural and functional modes of programming.

T-Fns

APL provides a function definition mechanism, the traditional “T-fn” in which:

- The function's name is declared
- Any arguments are named
- Any result is named explicitly
- Any local names are declared
- Control structures determine the order of execution of lines of APL

T-fns encourage a procedural style of programming in which it is easy to commission side-effects such as the mutation of data structures and the maintenance of state.

Here is Euclid's GCD algorithm coded as a T-fn. Notice that local variables m and n are repeatedly updated and that result variable z is assigned in the last line of the function.

```
     ∇ z←m gcd n
         :While n≠0
             (m n)←n,n|m
         :End
         z←|m
     ∇
```

This second function, given a 3-vector of its coefficients, returns a vector of the real roots of a quadratic equation:

```
     ∇ v←roots cfs;a;b;c;d         ⍝ Real roots of quadratic
         (a b c)←cfs               ⍝ coefficients
         d←(b*2)-4×a×c             ⍝ discriminant:
         :If d<0                   ⍝   negative:
             v←⍬                   ⍝     only complex roots
         :ElseIf d=0               ⍝   zero:
             v←-b÷2×a              ⍝     duplicate roots
         :Else                     ⍝   positive:
             v←(-b+¯1 1×d*0.5)÷2×a ⍝     two roots
         :EndIf
     ∇
```

Notice that the header line names: the result; function; argument; and local temporary variables.

A T-fn terminates after its last line or a `:Return` statement is evaluated, at which point, the current referent of the declared result variable is returned as the result of the function.

D-Fns

Dyalog's "D-Function" direct definition style provides an alternative in which:

- The optional naming of the function is separated from its definition
- Arguments are special tokens `⍺` and `⍵`
- The function result is implicit
- Names assigned within the function are local by default
- The guarded expression is the only control

A D-fn terminates with its first unassigned, unguarded expression or when a guard "fires" by evaluating to 1. D-fns may be named in the same way that arrays are named, by using the `←` assignment arrow.

```
      {⍵ ⍵} 0         ⍝ apply unnamed function
0 0
      dup←{⍵ ⍵}       ⍝ name function

      dup 0           ⍝ apply named function
0 0
```

D-fns encourage a purer form of programming in which the result of a function is just a single expression of its arguments. For ease of comprehension, the result expression may make use of local definitions, which precede it. Such an arrangement, where variables don't *vary* is an important characteristic of the "Functional Programming Style" [1].

Here are Euclid's algorithm and the function for the real roots of a quadratic, coded as D-fns:

```
{
    ⍵=0:|⍺
    ⍵ ∇ ⍵|⍺          ⍝ (∇ calls function recursively)
}


{                              ⍝ Real roots of quadratic
    a b c←⍵                    ⍝ coefficients
    d←(b*2)-4×a×c              ⍝ discriminant

    d<0:⍬                      ⍝ -ive: only complex roots
    d=0:-b÷2×a                 ⍝ zero: duplicate roots
    d>0:(-b+¯1 1×d*0.5)÷2×a    ⍝ +ive: two roots
}
```

APL#

“APL Sharp” is a new dialect of APL, which is aimed at the Microsoft.NET and similar “virtual machine” frameworks. Since full integration with the target frameworks means that the new language will not be 100% upwards compatible with current Dyalog APL, we have taken the opportunity to rationalize function definition and various other aspects of APL language design. [2]

The designers of **APL#** felt that both of Dyalog's definition styles had merit but that, rather than implement both, it would be desirable to try to unify them into a consistent whole. In particular APL# should provide:

- The cleanness of the D-function style
- A vehicle for both procedural and functional programming
- Both named and unnamed functions (and operators)
- Optional naming of arguments
- Both traditional control structures and D-fns' guard
- The ability to set both local and global state

An **APL#** defined function is of the form

```
    { <rarg> → <body> }           ⍝ monadic function
```
or
```
    {<larg> <rarg> → <body>}      ⍝ dyadic function
```

where `<larg>` and `<rarg>` are argument names and where `<body>` is a diamond- or newline-separated list of lines of APL code. For example:

```
    dup←{a→a a}         ⍝ duplicate

    sqrt←{n→n*0.5}      ⍝ square root

    root←{m n→n*÷m}     ⍝ m'th root

    gcd←{m n →          ⍝ Euclid's GCD
        n=0: |m
        n ∇ n|m
    }
```

To ease the transfer of D-functions from Dyalog, special tokens `⍺` and `⍵` will continue to refer to the function's left and right argument respectively and a missing argument specification, or "signature" is assumed to represent: `⍺ ⍵ →`. This means that many D-fns will port with only minor amendments to **APL#**. For example:

```
    gcd←{               ⍝ APL# function identical to D-function.
        ⍵=0: |⍺         ⍝ NB: space required to right of guard:.
        ⍵ ∇ ⍵|⍺

    }
```

Ambivalent Functions

In **APL#** all functions will be ambivalent; no special syntax will be needed to indicate whether a left argument is required or may be omitted.

The writer of an ambivalent function must decide what to do when a left argument is not given. Often, this amounts to supplying a default value as in this **T-fn** for the m-th (default square-) root of its numeric argument:

```
        ∇ z←m root n
    [1]   ⍎(0=⎕nc'm')/'m←2'
    [2]   z←n*÷m
        ∇
```

This assignment of a default value for a missing left argument was considered common enough that D-fns provided a special syntax `⍺←`:

```
root←{
    ⍺←2
    ⍺*÷⍵
}
```

Occasionally, the code in the body of a function needs to know *explicitly* whether a left argument has been supplied or not. In this case both T-fns and D-fns normally resort to interrogating the name-class (0 or 2) of the left argument, using `⎕NC`.

A third, and it turns out, equally common, task is to propagate the dynamic valence of the calling function to any called functions. In other words, a function might pass its *optional* left argument to a sub-function. In the early days of D-fns, Phil Last invented a technique for this:

```
⍺←{⍵}      ⍝ left argument defaults to identity fn.
⍺ sub ⍵    ⍝ sub called with/out left argument.
```

In **APL#**, `{⍵}` would be replaced with primitive function “right” `⊢`.

For **APL#**, Phil has suggested an improved mechanism, which incorporates all three cases: The header name corresponding to a missing left argument would be assigned to primitive *function* “`⊢`” on entry to the function. Then the code for the respective cases becomes:

```
larg←⍺⊣99           ⍝ left argument defaults to 99
monadic←↑0 ⍺ 1      ⍝ left argument missing?
⍺ sub ⍵             ⍝ sub called with optional left argument
```

Function termination

Unlike D-functions, where the *first* (and, under normal circumstances, only) unassigned, unguarded expression terminates, in **APL#**, in the absence of an explicit `:Return` statement or of a firing guard, the *last* executed expression of the function terminates and supplies the result.

Furthermore, Dyalog's concept of a “shy result” (where a function, such as `⎕EX`, returned a result *only* if the context required one) has been abandoned. Instead, in a slight departure from tradition, unassigned output from function lines is discarded. To force output to the session, `⎕←` or `⍞←` must be used. Note, however, that the result of an expression typed directly into the session is displayed as before:

```
    {
        ⎕←'display'    ⍝ value displayed
        'discard'      ⍝ value discarded
        'return'       ⍝ value returned
    }0
display
return
```

Tuples

As well as being single names, `<larg>` and `<rarg>` may also be parenthesised vectors of names, which correspond to the items of vector arguments, so:

```
roots←{ (a b c) →                 ⍝ Real roots of quadratic.
    d←(b*2)-4×a×c                 ⍝ discriminant
    d<0: ⍬                        ⍝ only complex roots
    d=0: -b÷2×a                   ⍝ duplicate roots
    d>0: (-b+¯1 1×d*0.5)÷2×a      ⍝ two roots
}

{(a b)→b a}               ⍝ reverse of 2-item vector

rgt←{((a b)c)→a(b c)}     ⍝ binary tree rotation
                          ⍝ www.dyalog.com/dfnsdws/n_BST.htm

rgt⍣2 ⊢('ab' 'c')'d'      ⍝ ((a b)c)d → a(b(c d))
a b cd
```

NB: Notice the difference between monadic `{(a b)→a+b}` and dyadic `{a b→a+b}`.

A more substantial example of tuple-naming might be this partial implementation of a brainfuck [3] machine, in which both the left and right arguments are pairs of lists:

```
bf←{ ((ss s)(t tt)) ((mm m)(n nn)) →    ⍝ ⍺:tokens ⍵:memory
     '+'=t: ((ss s)t)tt ∇ ⍵+0(1 0)      ⍝ increment
     '<'=t: ((ss s)t)tt ∇ (mm m)n ∇ nn  ⍝ shift left
     ...
}                        ⍝ http://en.wikipedia.org/wiki/Brainfuck

(↑{⍺ ⍵}/'∘',src) bf 0     ⍝ dyadic call on bf
```

Local names

In common with D-fns, assignments within the function body will create names, which are local to the function.

Making names local by default is a significant change for anyone converting an application based on T-fns to **APL#**. However, like Dyalog APL and other dynamic object oriented languages, **APL#** supports the creation of “spaces”, which can act as containers corresponding to global or semi-global contexts. Global names can reside simply in the “application root” `#.TheAnswer←42`; or the “current space” `⎕this.x←99`; or in suitably named sub-spaces `#.TrigConstants.pi←22÷7`.

For the moment, the *only* way to assign a global name in the current space is to preface it with “`⎕this.`”, though a neater syntax may emerge. For more on **APL#** “spaces”, see papers elsewhere in these proceedings [2].

Control and Guard Expressions

In **APL#**, control structures are *expressions* and so may return results. Just like a function, the result of a control structure is the result of its *last* executed expression:

```
          1 + (:If 2>3
               4
          :ElseIf 5<6
               7
          :Else
               8
          :EndIf) + 9
     17
          ⎕←(:For i :In 2 3 4
               x←i i
          :End)
     4 4
```

Similarly, a guard expression `(cond:true ⋄ false)` can return a result:

```
          1 + (
               2>3: 4
               5<6: 7
               8
          )+9
     17
```

NB: Name assignments within a guard expression will be local to any enclosing function, not just to the enclosing parentheses.

Operators

Both T- and D- styles allow the definition of operators (more precisely, derived functions), which take function or array operands in addition to array arguments. Each of the following operators applies its function operand to each depth-0 leaf item of the array right argument.

```
         ∇ z←(f leaf)r
     [1]   :If 0=≡r
     [2]       z←f r
     [3]   :Else
     [4]       z←f leaf¨r
     [5]   :EndIf
         ∇

          {
              0=≡⍵:⍺⍺ ⍵
              ∇¨⍵
          }
```

One shortcoming of the D-operator is that it is distinguished from a function only by the presence of ⍺⍺ or ⍵⍵ *somewhere* in the body of its code. For small examples, such as those above, this is fine but when the code extends over many lines, it is not easy to spot whether we're dealing with a function or an operator. **APL#** solves this problem by requiring an explicit signature for an operator.

APL# distinguishes operators by placing the operand name(s), in braces, to the left of the right argument:

```
{   {f}   n → …}    ⍝ monadic operator / monadic derived fn
{ m {f}   n → …}    ⍝ monadic    ..    / dyadic     ..
{   {f g} n → …}    ⍝ dyadic     ..    / monadic    ..
{ m {f g} n → …}    ⍝ dyadic     ..    / dyadic     ..
```

The above leaf example becomes:

```
leaf←{ {f} a →      ⍝ apply at leaves
     0=≡a: f a
     ∇¨a
}
```

By analogy with functions, special tokens ⍺⍺, ⍵⍵ and ∇∇ may be used within the body of the operator to refer to its left and right operand and to itself, respectively. In common with D-fns, ∇ within the body of an operator refers to the derived function (the operator bound with its operands). See [1] for details. Remember, however, that unlike functions, the operator signature may not be omitted.

Finally, any of the special tokens ⍺, ⍵, ⍺⍺ and ⍵⍵ may appear instead of names, in the signature, providing that each appears in its "proper" position. The coding for leaf could become:

```
leaf←{ {⍺⍺} ⍵ →          ⍝ apply at leaves
     0=≡⍵: ⍺⍺ ⍵
     ∇¨⍵
}
```

Summary

APL# defined functions and operators:

- Begin with an optional (for functions) signature, which defines their class and valence.
- Terminate after the last line, or a guard fires, or an explicit `:Return` is executed.
- Do not display unassigned results in the session.
- Allow a mixture of control structures and guarded expressions.

We have attempted to meld together two quite different modes of function definition. The result is inevitably a compromise and, during the design, there were understandably tensions between the supporters of the opposing styles. In particular, the D-functionista initially saw this as a diluting of his pure world (to the extent that he had to be appeased with the promise of a “golden light” that would appear in the edit window as long as the code under review remained pure and free from side-effects).

Although we have not been bound by maintaining upwards compatibility, the ability to systematically translate existing applications implemented using T-fns or D-fns has been kept in mind during the design, and we believe that semi-automatic conversion of very large parts of existing applications will be feasible.

In short, we believe that we have produced a definition, which will be attractive to both camps and which will turn out to be greater than the sum of its constituent parts. Time will tell.

The APL# project is a "work in progress"; we look forward to feedback from the community before proceeding with the final implementation.

Summary of Syntax

```
fnop       ::= { [signature] body }        ⍝ function / operator defn
signature  ::= [argt] [operands] argt →    ⍝ optional fn/op signature
argt       ::= namelist                    ⍝ function argument
operands   ::= { namelist [namelist] }     ⍝ operator operand(s)
namelist   ::= name | ( name ... )         ⍝ single or multiple names
body       ::= line | line separator body  ⍝ function / operator body
separator  ::= <newline> | ⋄               ⍝ nl- or ⋄- separated lines
```

References

[1] “Dynamic Functions in Dyalog APL” www.dyalog.com/download/dfns.pdf

[2] “An APL for the Microsoft.Net Framework”, Kromberg, M., Proceedings APL 2010 LPA - Berlin, 2010.

[3] An implementation of the brainfuck language in D-fns: www.dyalog.com/dfnsdws/n_bf.htm

Acknowledgement

Thanks to Phil Last for the suggestion regarding the design of ambivalent functions.

Appendix: Further Examples

```
gcd←{m n →               ⍝ looping version of gcd
     |↑ (:While n≠0
          (m n)←n,n|m
        :End)
}

leaf←{ {f} r →           ⍝ leaf with ctrl struct
     :If 0=≡r
          f r
     :Else
          ∇¨r
     :EndIf
}

leaf←{ {f} r →           ⍝ leaf with fn-returning guard
     (0=≡r: f ⋄ ∇¨) r
}

root←{⍵*÷⍺⊣2}            ⍝ ⍺'th root (default sqrt)

roots←{ (a b c) →        ⍝ roots with ctrl struct expression
     d←(b*2)-4×a×c
     (:If d<0
          ⍬
     :ElseIf d=0
          -b
     :Else
          -b+¯1 1×d*0.5
     :End)÷2×a
}

roots←{ (a b c) →        ⍝ roots with guard expression
     d←(b*2)-4×a×c
     (
          -b+(
               d<0: ⍬
               d=0: 0
               d>0: ¯1 1×d*0.5
          )
     )÷2×a
}
```