Gerunds and Representations*

Robert Bernecky† **Roger K.W. Hui**‡

July 17, 2006

Abstract

Gerunds, verbal forms that can be used as nouns, are recognized as having utility in the realm of programming languages. We show that gerunds can be viewed as arrays of atomic representations of verbs (functions), in a way which is consistent with the syntax and semantics of APL, and which allows verbs to be first class objects in the language. We define derivations of verbs from gerunds in the J dialect of APL, and show how these derivations provide control structures for sequencing, selection (in the sense of generalized forms of CASE or SWITCH statements and IF/THEN/ELSE), iteration (DO UNTIL), recursion, and parallel computation (MIMD, or Multiple Instruction, Multiple Data) . We conclude with alternative representations of verbs which are useful in other contexts.

1 Introduction

A gerund is a verbal form that can be used as a noun, as "programming" in "programming is an art". The utility of nouns derived from verbs is also recognized in the realm of programming languages: for example, in C, such forms exist as arrays of "pointers to functions"; and in APL, "function arrays" have been proposed by Bernecky [2], Brown [3], Falkoff [5], Landaeta [15, 16], and Piper [17].

We show that gerunds can be viewed as arrays of atomic representations of verbs, in a way consistent with APL syntax and semantics. We then describe derivations of verbs from gerunds, to perform sequencing, selection, iteration, recursion, and parallel computations; and conclude with alternative representations of verbs useful in other contexts. Illustrations are drawn from the J dialect, described in Hui *et al.* [7] and Iverson [12, 13, 14].

2 Atomic Representation

The conjunction denoted by the grave accent () catenates a noun to the atomic representation of a verb, and produces a noun. Given two verbs, it forms the atomic representation of both verbs, and catenates them. Thus if `m` and `n` are nouns, `u` and `v` verbs, and `ru` and `rv` atomic representations of `u` and `v`, then:

```
m ` n   is   m,n
m ` v   is   m,rv
```

*This paper originally appeared in the ACM SIGAPL APL91 Conference Proceedings

†Snake Island Research Inc., 18 Fifth St., Ward's Island, Toronto, Ontario, Canada M5J 2B9; Tel: (416) 203 0854 email: bernecky@acm.org

‡Iverson Software Inc., 33 Major St., Toronto, Ontario, Canada M5S 2K9;(416) 925-6096

```
   u ` n   is   ru,n
   u ` v   is   ru,rv

   ger =. + ` - ` = ` !
   ger
+-+-+-+-+
|+|-|=|!|
+-+-+-+-+

   $ger
4

   +/ ` % ` #
+-------+-+-+
|+-+---+|%|#|
||/|+-+|| | |
||  ||+||| | |
||  |+-+|| | |
|+-+---+| | |
+-------+-+-+
```

ger is a 4-element boxed list of the atomic representation of `+`, `-`, `=`, and `!`. The phrase `+/%#` produces the 3-element boxed list of the atomic representation of `+/` (sum), `%` (divide), and `#` (tally, the length of the leading axis).

The atomic (boxed, canonical) representation of an object uniquely represents it, and is either a boxed identifier (for primitive objects); or a boxed list of two boxes, the identifier and arguments (for derived objects). The identifier for a primitive verb, adverb, or conjunction is simply (and arbitrarily) its 1- or 2-letter spelling; the identifiers for a noun, hook, and fork are `0`, `2`, and `3`, respectively. Arguments are recursively so represented. User-defined conjunctions and adverbs (operators) and verbs (functions) are merely those derived from the definition conjunction (`:`), and are, therefore, covered by this scheme.

```
   t =. '$.=.1+0<y.';'1';'y.*$:y.-1'
   fact =. t : ''
   fact ` ''
+-------------------------+
+--+---------------------+|
||:|+--------------+----+||
||  ||+-+----------+|+-++|||
||  |||0|$.=.1+0<y.|||0|||||
||  ||| |1         ||+-++|||
||  ||| |y.*$:y.-1 ||    |||
||  ||+-+----------+|    |||
||  |+--------------+----+||
|+-+---------------------+|
+-------------------------+
```

An array so constructed, a noun derived from verbs, is a *gerund*. Since gerunds are arrays, they can be manipulated by verbs. In the jargon of computer scientists, verbs in J are thereby "first-class" objects.

3 Verbs from Gerunds

Many useful verbs can be defined from gerunds. In this section we give a few examples of such verbs. For the nonce, we use the conjunction `:` to define some such gerundial forms, using an integer scalar right argument to encode the particular form; separate symbols can be assigned to the more common of such forms.

3.1 Abut

Abut, denoted as `g:0`, allows parallel application of an array of verbs to arrays of data. The ranks of abut are the maxima of the ranks of the elements of `g`; the result, the catenation of the results. (If `g` has a single element, i.e., if `g` is the atomic representation of a single verb `v`, then `g:0` is exactly `v`.) For example, the negative (`-`), reciprocal (%) , and exponential (^) of a list of numbers can be found by:

```
   g=. - ` % ` ^ `:0

   g 1 2 3 4
     _1       _2        _3       _4
      1      0.5 0.333333     0.25
2.71828 7.38906  20.0855 54.5982

   5 g 1 2 3 4
4         5   5
3       2.5  25
2 1.66667 125
1      1.25 625

   2 3 g 4 5
_2 0.5  16
_2 0.6 243
```

3.2 Parallel Computation

Flynn [6] developed a taxonomy of computer architectures, based on the number of instruction-issuing units (single or multiple), and the number of data elements (single or multiple) manipulated by a single instruction. Traditional computers are SISD: Single Instruction, Single Data. The Maspar MP-1 and Thinking Machines CM-2 processors, and the ISO APL language are Single Instruction, Multiple Data. A number of MIMD (Multiple Instruction, Multiple Data) machine architectures, such as the Intel iPSC/2 have recently appeared on the computing scene. These machines, typically consisting of thousands of processors connected by communication networks such as meshes, banyan trees, hypercubes, or butterflies, present significant problems to programmers trying to use them effectively.

In 1984, Bernecky [2] proposed introduction of *function arrays* into APL as a way to achieve MIMD capabilities in an otherwise SIMD (Single Instruction, Multiple Data) language. This section presents an implementation of function arrays.

`g` is a gerund `s$ v0 v1 v2 ... vn`, i.e., a gerund composed from verbs $\mathtt{v}_i$ with shape `s`. The derived verb `g:6` has infinite ranks. `g:6 y` splits `y` along its leading axes, according to the shape of `g`, and applies the verbs $\mathtt{v}_i$ to corresponding cells of `y`.

Thus, if `g` consists of 2 3 repetitions of matrix inverse `%.`, then `g:6 y` views `y` as a 2 3 array of cells, and applies `%.` individually to those cells.

```
   g =. 2 3 $ + ` %.

   $y
2 3 5 4

   $ g `: 6 y
2 3 4 5
```

`g:6` derives a MIMD (multiple-instruction, multiple data) computation, with the individual verbs in the gerund providing the "multiple instruction", and cells in the array argument the "multiple data". The computation is similar to alternative 3 in [2], but does the "function array" split on the leading axes. The idea arose at the 1990 Minnowbrook APL Workshop, in a late-night discussion involving Kirk Iverson and the authors.

3.3 Ease of Compilation

ISO APL[9] is resistant to efficient compilation:

- Dynamic scoping either invokes a substantial execution time penalty, or requires meticulous inspection of all possible calling trees. Dynamic scoping is also implicated in the common APL bug of failure to localize a variable, but a discussion of that issue is beyond the scope of this paper.

- The absence of traditional control structures other than branch makes data flow analysis difficult.
- Execute, often used to perform CASE statements, requires substantual compile time analysis, and frequently cannot be compiled successfully.

One of our design goals is to make efficient execution of J easier, without limiting its expressive nature.

The gerundal forms we now introduce differ from compositions and expressions generated with execute or the definition primitives in that they treat verbs as first-class objects. Furthermore, they do not require the presence of a symbol table to determine the meaning of a compositional element at execution time. Rather, such meaning may be determined at compile time, allowing execution to proceed at high speed.

3.4 Gerundal Forms

Consider `g` to be a gerund formed from the verbs v_i, as follows:

```
g =.  v0 v1 v2 ...  vn
```

We now define the following verbs derived from `g`:

- dynamic compose,
- hook/fork, and
- generalized insert.

3.4.1 Dynamic Composition

Dynamic Compose (`g:57`) composes the elements of `g`, as:

```
v0 & v1 & v2 & ...  & vn
```

3.4.2 Hook and Fork

Hook/Fork (`g:59`) is the hook/fork [11] phrasal form, defined as:

`v0 v1 ...  vn`.

3.4.3 Generalized Insert

Reduction, first introduced by APL, and recently by Fortran 90, provides a way to insert a single verb between subarrays of an argument and evaluate the resulting expression. [1] Reduction is called *insert* in J, after the adverb's definition, rather than after the shape of the result of its application.

We define a generalization of insert to accept a gerund as the left argument, allowing more than one verb to be inserted: *Insert*, denoted as `g:3 y`, or `g/y`, inserts elements of the gerund `g` between items (major cells) of `y`, cyclically repeating the elements of `g`, or stopping after exhausting `y`.

`g/y` is:

```
(0[y)v0(1[y)v1(2[y)v2(3[y)...
```

As an example of the utility of generalized insert, consider the use of Horner's Rule for evaluating a polynomial `y` at `x`:

```
   y=. 12 7 _2 3
   x=. 4
   12 + 4 * 7 + 4 * _2 + 4 * 3
200
   t=. 12 4 7 4 _2 4 3
```

[1] Fortran 90 provides only a few reductions, as library functions.

```
   + ` */t
200
```

3.5 Control Structures

Traditional programming languages support a variety of control structures such as IF/THEN/ELSE, and CASE or SWITCH. These structures are intended to improve the maintainability of programs and simplify control flow analysis (important for proving correctness as well as for generation of optimized code). In the past, APL dialects have by and large ignored control structures, because the power of APL's array-oriented verbs and adverbs reduce the number of cases where explicit control is required.

However, there are cases where iteration or control flow changes are required, and these have, in the past, required clumsy expressions using explicit branching or execute statements. Gerunds allow introduction of control structures into J in a concise and elegant fashion. Moreover, because the control structures are array-oriented, they offer significantly more capability than their cognates in scalar-oriented languages such as C or Fortran.

3.5.1 IF and CASE Statements

If `g` is a gerund, `g:5` derives a CASE verb, whose left rank is 0, and whose right rank is the maximum of the monadic ranks of elements of `g`. In `x g:5 y`, the result for corresponding cells `xc` and `yc` is `(xc[g):0 yc`. (The phrase `xc[g` selects from `g` using index `xc`.)

`(k[g):0 y` and `k g:5 y` both implement CASE. The former defines a verb from a selected element of the gerund; the latter defines a verb from the entire gerund with the selection occurring at the time the verb is used, and is therefore more usefully composeable with other verbs. The hook phrasal form `g:5~ f`, where `(g:5~ f) y` is `(f y) g:5 y`, is expected to be useful.

Consider the calculation of bank interest on a savings account, according to the following rules:

- If the balance is less than \$1000, there is a service charge of \$20.
- If the balance is between \$1000 and \$20000, the interest rate is 8% on the entire balance.
- If the balance exceeds \$20000, the interest rate is 9% on the balance over \$20000 and 8% on the first \$20000.

A scalar-oriented traditional language would express this as two nested control structures, a loop over the array of bank balances `y`, and an IF-THEN-ELSE statement:

```
for (i=0; i<length; i++)
  if (1000>y[i])
     z[i] = -20;
  else
     if(1000<=y[i]&&y[i]<=20000)
       z[i] = 0.08*y[i];
     else
       z[i] = 0.08*20000 +
              0.09*(y[i]-20000);
```

Using gerunds and CASE, we might proceed as follows:

```
   v0    =. _20&[
   v1    =. 0.08 & *
   v2    =. (0.08*20000)&+ @
            (0.09&*) @ (-&20000)
   int   =. v0 ` v1 ` v2 ` :5"0
   range =. +/@(1000 20000&<:)"0
```

`int` forms a composite interest verb from `v0`, `v1`, and `v2`, applying to scalars (rank 0 cells); `range` computes the index of the applicable interest calculation. The interest on `y` may then be calculated by `(int~ range) y` or equivalently, `(range y) int y`.

Therefore, `g:5` provides facilities similar to the IF-THEN-ELSE and CASE constructs of traditional programming languages, but in synergy with other adverbs and conjunctions affords additional flexibility and power.

3.5.2 Recursion

Commonly, a recursive computation has four parts, *Proposition*, *Basis*, *Combine*, and *Reduce*:

- *Proposition* is used to decide whether to continue the recursion. For example, when computing a factorial, the Proposition can be `*` (signum).
- *Basis* computes the result when the proposition is false. In the factorial example, the Basis is `1:` (the constant function 1). When the proposition is true, the result is computed using Combine and Reduce.
- *Combine* applies to the argument and the result of one recursive step, to produce the overall result. In the factorial example, the combine part is `*` (multiply).
- *Reduce* alters the argument for the next recursion step. In the factorial example, the Reduce is `<:` (decrement).

Accordingly, `g:4` derives a monadic recursive verb from a 4-element gerund `g`:

```
g =. p ` b ` c ` r
f =. g ` :4
```

If `p y` is all zeros then `f y` is `b y`; otherwise, it is `y c f r y`. For example:

```
   fact =. * ` 1: ` * ` <: `: 4
   fact 5
120

   cb  =. ] , +/@(_2&{.)@]
   fib =. >&1 ` (i.@>:) ` cb ` <: `: 4
   fib 7
0 1 1 2 3 5 8 13
```

In the Fibonacci example `fib`, the Proposition is `>&1` (exceeding 1); the Basis is `i.@>:` (integers from 0 to n inclusive); the Combine is `cb`, a fork which appends to the right argument the sum of its last two elements (and which ignores the left argument); and the Reduce is `<:` (decrement).

3.5.3 Iteration

DO UNTIL is a common iterative control structure in many programming languages. The key parts of this structure are *Until*, *Gather*, *Main*, and *Select*:

- *Until* is a proposition; the iteration continues until this proposition becomes true. For example, when computing the exponential function using a power series, the iteration continues until the ratio of the current sum to the current term is greater than a fixed number.
- Each iteration contributes to the result. *Gather* applies to the current result and the current term, to obtain a new result. In the exponential example, Gather is simply `+`, adding the current term to the current sum.

- *Main* applies to the current term and the result of Select, to obtain a new current term. In the exponential example, Main is simply `*` (times).

- *Select* applies to the argument and the iteration index `i`. In the exponential example, Select is the hook `%>:` (`x` divided by 1+i). Thus, Select need not be a "selection function" at all.

In much the same manner as recursion, `g:1` derives a *DO UNTIL* verb:

```
   g =. u ` ga ` m ` s
   f =. g `: 1
   x f n;<b
```

`x f n;<b` begins with `r=.c=.b` and `i=._1`, and does `r=.r ga c=.c m x s i=.i+1` at most `n` times, until `r u c` becomes true, returning `r` as the final result. Thus, `u`, `ga`, `m`, and `s` play the roles of Until, Gather, Main, and Select; and `r`, `c`, `i` are the result, current term, and the iteration index. For example:

```
   f =. 1e5&<@% ` + ` * ` (% >:) `: 1

   2 f 20 1
7.38905
```

`f` computes the exponential of 2 using a power series, continuing until the sum `r` is more than one hundred thousand times greater than the current term `c`, obtained from the the previous term by `c*x%>:i`. The computation stops after at most 20 iterations, and the sum is initialized to 1.

4 Other Representations

In principle, gerunds could have been defined in terms of alternative representations. The following are two other representations of verbs.

4.1 Display Representation

In APL, a noun result of a sentence whose last action is not assignment, is automatically displayed. Similarly in J, a noun, verb, adverb, or conjunction result of a sentence (in immediate execution) whose last action is not assignment, is automatically displayed. Furthermore, the display of any named object is available (as an array) via the derived verb `5!:2`. Just as the display of nouns gives insight into their value, the display of verbs, adverbs, and conjunctions gives insight into their value and meaning.

```
   +
+-+
|+|
+-+
   3&+
+-+-+-+
|3|&|+|
+-+-+-+
   mean =. +/ % #
   mean
+-----+-+-+
|+-+-+|%|#|
||+|/|| | |
|+-+-+| | |
+-----+-+-+
   d =. 5!:2 <'mean'
   $d
3
   '#' = >2{d
1
```

4.2 String Representation

Cartwright [4], Bernecky *et al.* [1], and IBM [8] specified the Workspace Interchange Convention representation for APL objects as a string. This string representation facilitates exchange of data and programs between disparate computers and APL dialects. The string representation uniquely represents an object, and representations can be catenated together into a single string.

In J, the string representation of a named object is computed by the derived verb `5!:3`. This representation is a conforming extension of the WSIS convention: boxed arrays are recursively represented, per [1]; and the string representations of verbs, adverbs, and conjunctions are those of their atomic representations.

Briefly, the string representation has the following parts:

- length (excluding the length itself)
- type: character(`c`), numeric (`n`), boxed (`xb`), verb (`xv`), adverb (`xa`), conjunction (`xc`).
- name (or `-` if anonymous)
- rank
- shape
- elements in ravel order, which for boxed arrays are themselves string representations.

For example:

```
   sr  =. 5!:3
   str =. 'Cogito, ergo sum.'
   ces =. (<'Cogito'),(<,','),
          (<'ergo'),<'sum.'
   sum =. +/
   sr <'str'
27cstr 1 17 Cogito, ergo sum.
   ces
+------+-+----+----+
|Cogito|,|ergo|sum.|
+------+-+----+----+
   $ ces
4
   sr <'ces'
60xbces 1 4 13c- 1 6 Cogito
     8c- 1 1 ,11c- 1 4 ergo
     11c- 1 4 sum.
   $sr <'ces'
62
   sr <'sum'
38xvsum 1 2 8c- 1 1 /17xb- 1 1
     8c- 1 1 +
```

The adverb `5!:0` defines an object from its atomic, display, or string representations.

```
   fx =. 5!:0
   (sr<'sum') fx 3 1 4 1 5 9
23
```

The choice of string representation was made with an eye toward compatibility with past work in this area. Because string representation is count-controlled, reconstruction of objects from strings is an iterative process. It would be interesting to contemplate a representation which allowed parallel reconstruction.

5 Acknowledgements

K.E. Iverson originated the key idea of atomic representations, and contributed to the development of gerunds and their uses over many conversations. E.E. McDonnell suggested the analogy between gerunds in natural languages and array representations of verbs. He also shepherded the paper through several drafts that much improved its presentation.

References

[1] Robert Bernecky, Greg Bezoff, and Michael Symes, *APL Workspace Transfer, Sharp APL Technical Note 22*, Revision 4, (1981-07-15).

[2] Robert Bernecky, "Function Arrays", *APL84 Conference Proceedings, APL Quote Quad*, Vol. 14, No. 4, (1984).

[3] James A. Brown, "Function Assignment and Arrays of Functions", *APL84 Conference Proceedings, APL Quote Quad*, Vol. 14, No. 4, (1984).

[4] Dana E. Cartwright, "Workspace Interchange Convention", *APL Quote-Quad*, (December 1977).

[5] A. D. Falkoff, "Algorithms for Parallel Search Memories", *Journal of the ACM*, Vol. 9, No. 4 p. 493, (October 1962).

[6] M. J. Flynn, "Very high-speed computers", *Proceedings of the IEEE*, Vol. 54, pp. 1901-1909, (December 1966).

[7] Roger K. W. Hui, K. E. Iverson, E. E. McDonnell, and Arthur T. Whitney, "APL\?", *APL90 Conference Proceedings, APL Quote Quad*, Vol. 20, No. 4, (July 1990).

[8] *APL2 Programming: Language Reference*, IBM Corporation, pp. 427-429, SH20-9227, (1987).

[9] *International Standard for Programming Language APL*, ISO 8485, (1984).

[10] K. E. Iverson, "A Dictionary of APL", *APL Quote Quad*, Vol. 18, No. 1, (September 1987).

[11] K. E. Iverson and E. E. McDonnell, "Phrasal Forms", *APL89 Conference Proceedings, APL Quote Quad*, Vol. 19, No. 4, (August 1990).

[12] K. E. Iverson, "J", *Vector*, Volume 7, Number 1, (August 1990).

[13] K. E. Iverson, *Tangible Math and the ISI Dictionary of J*, Iverson Software Inc., Toronto, (1990-10-06).

[14] K. E. Iverson, "The Dictionary of J", *Vector*, Volume 7, Number 2, (October 1990).

[15] David A. Landaeta, "A Notation for Manipulating Arrays of Operations", *APL86 Conference Proceedings, APL Quote Quad*, Vol. 16, No. 4, (1986).

[16] David A. Landaeta, "The Theory of Function Arrays", *APL88 Conference Proceedings, APL Quote Quad*, Vol. 18, No. 2, (1988).

[17] David B. Piper, "Semantic Class and Arrays of Functions", *Vector*, Volume 6, Number 1, (July 1989).

APPENDIX: Notation Summary

J Version 3.1, released May 1991, implements:

`g‘:0`	Abut
`g‘:1`	DO-UNTIL
`g‘:2` or `g/`	Insert
`g‘:4`	Recursion
`g‘:5`	IF-THEN-ELSE and CASE

J is described in Hui *et al.* [7] and Iverson [13, 14]. This summary includes only those symbols used in the paper, and only where they differ from ISO APL[9]. Primitive words are spelled with one or two letters, with the second letter a period or colon. In what follows, `f` and `g` are verbs, and `m`, `n`, `x`, and `y` are nouns.

Phrase	Monadic Definition	Dyadic Definition
`=.`	Copula (Assignment)	
`<`	Box	Less than
`<:`	Decrement	Less than or equal to
`>`	Open	Greater than
`>:`	Increment	Greater than or equal to
`_`	Negative sign (`_3 = -3`)	
`*`	Signum	Multiply
`%`	Reciprocal	Divide
`%.`	Matrix Inverse	Matrix Divide
`^`	Exponential	Power
`$`	Shape of	Reshape
`$.`	Suite, sequence control (in `:`)	
`$:`	Self Reference (in `:`)	
`f~`	Commute: `f~ y` is `y f y`	`x f~ y` is `y f x`
`:`	Explicit definition	
`;`	Ravel Items	Link
`#`	Tally (number of items) `0[($y),1`	Copy (replicate)
`f@g`	Atop: `f@g y` is `f g y`	`x f@g y` is `f x g y`
`[`	Same: `[y` is `y`	Left: `x[y` is `x`
`]`	Same: `]y` is `y`	Right: `x]y` is `y`
`[`	Catalog	From (indexing)
`[.`	First	Take
`‘`	Gerund from Verbs	
`‘:`	Verb from Gerund	
`m"n`	Constant verb `m` with rank `n`	
`f"n`	Verb `f` with rank `n`	
`m&g`	With: `m&g y` is `m g y`. `0.5&*` is halve.	
`f&n`	With: `y f n`. `^&0.5` is square root.	
`f&g`	Compose: `f&g y` is `f g y`	`x f&g y` is `(g x) f (g y)`
`(f g)`	Hook: `y f g y`	Hook: `x f g y`
`(f g h)`	Fork: `(f y) g (h y)`	Fork: `(x f y) g (x h y)`
`i.`	Integers	Index Of
`o.`	Pi times	Circle
`x.`	Left argument (in `:`)	
`y.`	Right argument (in `:`)	
`m!:n`	External	
`1:`	Constant Verb 1	

A Keywords

- J
- Parallel Computation
- APL
- control structures
- MIMD
- first-classness
- gerund
- function arrays
- representations
- CASE statement
- IF Statement
- DO statement
- Recursion