Untying the Gordian Knot: Agreement in J

Martin Neitzel
Maschstr. 30
38114 Braunschweig
Germany
neitzel@miles.gaertner.de

Abstract

The "Dictionary of J" is not written for the faint of heart. Its numerous self-references make it very difficult to tackle. The author thinks that section II.B, "Verbs", is one of the keys for getting a confident grip of the language. This paper basically just repeats this section using a tutorial style, some historical background, and several pictures. Hopefully, it will enable people with analytical minds to find their own way through the Dictionary. It closes with a discussion of possible language changes in the context of II.B.

Keywords J, agreement, cells, rank, execution model, array processing

1 Dabbling in J

The first time I tried to learn J was about four years ago. I was very confused at that time over unexpected results in my experiments, much like others have described [EisPe93, McINT91]. However, the problem was not that my little J expressions kept failing with "rank errors". Rather the contrary was the case—the interpreter accepted and executed more sentences than I expected, and even very short ones did not work as intended. I was able to explain most of the results to occasional bystanders.[1] But to tell the truth, I was quite mystified myself. All the "new stuff" available in J—trains, tacit programming, adverbs and conjunctions—appeared to me to be not so difficult. But basic array processing went wrong again and again.

[1] That is, I was faster at spotting which items had been multplied by which and could do all the appropriate hand-waving.

APL '95, San Antonio, Texas, USA

One day I decided to forget all the tutorial introductions and to start reading the Dictionary very carefully from scratch in order to understand every single word and mutually-recursive definition. On the second reading, I discovered the rule that had hampered my getting along with J initially: "Agreement".

Section II.B of the Dictionary, *Verbs,* defines how verbs are evaluated, and its subsection "Agreement" in particular defines how the two operands in the dyadic case must match. These rules allow you to apply a function not only to scalar values but also to entire arrays. They are at the heart of APL and all its dialects. If you want to point your finger at the one principle in J liberating you from the von Neumann style, it is section II.B.

These rules have been changed a bit in various APL dialects, and I want to describe their development up to what J is doing now. The changes look small, but are important to know. I was not aware of them for a long time, and that was why I kept stumbling over my own feet.

2 APL Classic

Figure 1 shows how a scalar function like "+" in flat-APL turns scalar and array operands into results.

This is so simple to understand and to use that any formalism to describe it looks a bit superfluous. Nevertheless, here is a rule defining how the left and right arguments have to agree in the dyadic case:

Both are scalars:
: a simple addition results in a scalar result.

Both are arrays of the same shape:
: simple additions are executed between "corresponding" elements to form a result of the same array shape.

One is a scalar, the other is an array:
: the scalar is added repeatedly to every element of the array to form a result array.

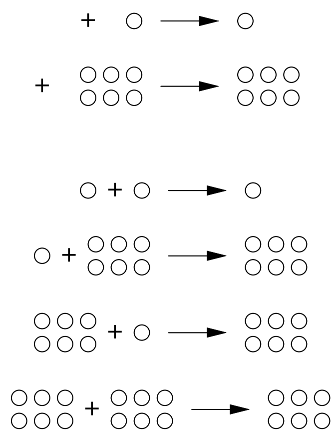

Abbildung 1 Agreement in flat APL

However you put it, any computation will involve either a one-to-one pairing of corresponding elements or a one-to-many distribution of a scalar over an array, but not both at the same time.

3 Agreement in APL2

APL2 has nested arrays, i.e. any element of an array can itself be an array. Functions are said to be *pervasive* since they will reach their arguments no matter how deeply nested they are.

The agreement rule for the dyadic case is a simple, recursive extension of the flat-APL rule [BROWN91, BrCr91].

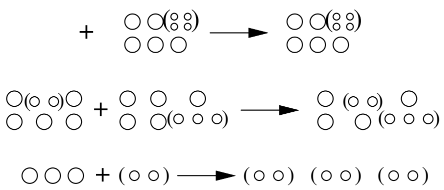

Abbildung 2 Agreement with nested arrays

As before, scalars will pair with scalars, two arrays must have the same shape at depth 0 and will pair their corresponding elements. In addition, wherever a scalar on one side corresponds to an array on the other, it will be distributed over it. Nested arrays are treated as scalars and will be extended (replicated) when confronted with an array.

Notice that one-to-one correspondences and one-to-many distributions can now occur at the same time at different places.

4 From Scalars to Cells

In flat-APL only so-called "scalar functions" behave according to the simple rules of array processing. The other functions do not follow them but use their own, special rules.

The Dictionary of APL [IVERS87] introduces the concepts of *cell*, *frame*, and *verb rank* to solve this problem. Matrix divide, for example, is defined to operate on rank-2 cells. The complete set of flat-APL agreement rules and the corresponding diagrams from Figure 1 now become applicable if each blob is not a scalar but, say, a 5 by 5 matrix. An argument of shape 2 3 5 5 falls into a sixpack arrangement (the 2 3 frame) of 5 by 5 cells.

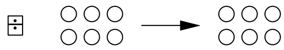

Abbildung 3 Inverting an array of shape 2 3 5 5

Any verb (functions are called that in the Dictionary) has a set of three ranks associated with it, defining the ranks of the expected arguments. For example, "+" has the associated ranks 0, 0, and 0 for the monadic, left, and right arguments respectively, indicating that it basically operates on scalars. ρ has rank (monadic: unbounded; left: 1; right: unbounded), so you can deduce that re-shape will take the elements from the right argument "as a whole" and rearrange them according to a vector on the left. In the following, compact notations of the form (m l r), (l r), and (m) are used to indicate the ranks of a verb.

The old, flat-APL agreement rules are used with only one crucial change. We just have to substitute "cells" for "scalars" in our argument descriptions. Argument arrays can now be considered as two-level arrays. The shape of the entire array can be broken down into (a) the shape pertaining to the cells (with the rank requested by the verb at hand) and (b) the "frame", which is the arrangement of cells to make up the entire argument. Such a division is just an interpretation on behalf of a verb, not an intrinsic property of an array itself.

The pairing of corresponding cells is exactly as it was for scalars in flat-APL. A single cell (in an empty frame) is distributed over an array of cells on the side (in a non-empty frame). If there are arrays of cells on both sides, the frames must be identical and corresponding cells are paired. For diagrams, just look at the "APL Classic" picture and squint a little. Excellent and thorough introductions of function rank have been written in the context of Dictionary APL and are still useful for J programmers [BERNE88, BENKA84].

5 How Will We Three Meet Again?

Dictionary APL also has to tackle a new problem. Since all verbs fall under the evaluation rules, it is now possible to get partial results that are non-trivial to combine. It is obvious that all the cell pairings yield a result *cell* each, not only a scalar as could be assumed in flat-APL. It is furthermore obvious that all partial result cells should be embedded in a common result according to the frame at hand. But their shapes might differ and, since only orthogonal arrays are valid objects, there may be a need to make these compatible first.

Consider the following three-element vector made up from (scalar) boxes (J-ers please substitute ";" for the *link* ⊃):

```
      someboxes ← 1 ⊃ 2 3  ⊃ 4 5 6
┌─┐ ┌───┐ ┌─────┐
│1│ │2 3│ │4 5 6│
└─┘ └───┘ └─────┘
      > someboxes
1 0 0
2 3 0
4 5 6
```

When *someboxes* is opened again, the *open* applies to each single box individually (because it has rank (0)), producing as intermediate results a scalar, a two-element vector, and a three-element vector. These have to be *conformed,* as the standard for Extended APL puts it [EKLOF93]. That means to elevate the scalar "1" into a one-element vector and then to pad the shorter vectors to the common length 3. After this, the three vectors can finally be put into a single common result array.

Another example, this time dyadically, is:

```
twoshapes ← 2 2 ρ 2 8  4 5
twoshapes ρ 'abc'
```

In terms of cells, we have the following arrangement here:

○○ ρ ○ ——► ○○

The individual result cells as produced by the second ρ are a 2 by 8 and a 4 by 5 matrix respectively. Space-padding will be used to create two 4 by 8 matrices, and these can be combined into the overall result of shape 2 4 8.

The problem of conformance does not arise in the APL2 model because APL2 has no problem mixing sub-arrays of different shapes. On the other hand, APL2 still distinguishes scalar functions from others.

Dictionary APL also introduces the concept of the *rank conjunction*. However, things are already complicated enough without it in the context of J, so we will defer its discussion until later.

6 The Three-Step Model

When I programmed in flat-APL, I just imagined scalar functions to be smart enough to operate on entire arrays. With Dictionary APL or J, it pays to use a more detailed model for function evaluation.

Each dashed box in Figure 4 represents an activation of *u* with cells as input and ellipses as intermediate results. There are three steps involved:

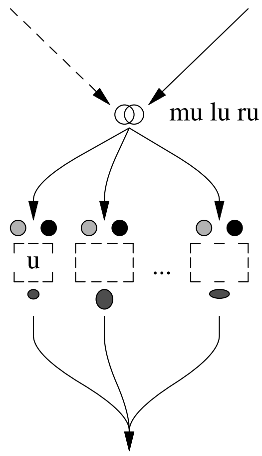

Abbildung 4 The 3-step model for verb evaluation

1. Argument dissection and cell pairing

   The actual arguments are sliced into cells as requested by the verb's ranks. While the monadic case is trivial, the dyadic case requires the establishment of all pairings between cells from the left with cells from the right.

2. Evaluation

   For each cell or cell pair, an instance of the verb computes an intermediate result. If the verb is not one of the traditional scalar functions, these results may differ in their shape. (Randomly shaped ellipses are drawn to remind the reader of this possibility.)

3. Recombination

   All intermediate results are forced into a common shape if necessary and combined into an overall result.

Newcomers to J may benefit from the following points:

- Dissection and recombination are considered to be outside the scope of any specific verb. They will happen anyway, as part of the fundamental J mechanics. The Dictionary defines each verb only by describing what happens in step 2. In J, the verb "+" has dyadic rank (0 0) and will just add exactly one scalar to another one, as in most other conventional languages. But whenever you confront a verb with more than one cell (or cell pair), steps 1 and 3 take care of that, so "arrays still work".
- In many situations, dissection and recombination become simple no-ops. For verbs with unbounded rank, the actual argument can be used directly as cell, and the single result need not be combined with anything else. Similarly, there are certain contexts which ascertain that a verb acts on exactly one cell (or cell pair).
- Whenever two verbs are composed, the dissection and recombination steps will interfere with each other and may give rise to surprising results. It is surprising, because all seems to work fine as long as you do your experiments at the cell level, but *woe!* if you start throwing higher-ranked arrays at your new function.

- When discussing programs with other people, please try to avoid using the unadorned word ‘‘argument’’ because it occurs in two contexts. It is better to refer distinctively to ‘‘the actual argument’’ as before dissection on the one hand, and on the other hand to cells passed to the verb in step 2 (which are always of the requested rank).
- Several verbs impose further structure on their cell arguments. When this is the case, it happens in addition to the dissection into cells that has already occurred by that time. For example, *copy* (dyadic #) copies rows if the right argument is a matrix.

7 Agreement in J

We have already seen that the recombination step can get a bit complicated in Dictionary APL. In addition, J provides a pepped-up cell pairing strategy for step 1. The agreement rule for dyadic expressions has been changed. It now reads:

> ... the arguments of *v* must *agree* in the sense that their frames (relative to the ranks of *v*) must either match, or one must be a prefix of the other... [IVERS90]

This is admittedly very abstract. Here is an example demonstrating the meaning of the involved shapes, frames and prefixes.

Let *x* be of shape 2 3 2 5 2, *y* of shape 2 3 2 2, and *v* have ranks (3 0). To find the correct pairings of cells, the first thing to do is to drop the trailing, cell-related axes from the shapes according to the ranks of *v*. This leaves us with the the frames 2 3 for *x* and 2 3 2 2 for *y*. Scalar cells have no axis, so the shape of the argument is the frame. At this point, the common frames and a single trailing frame component become visible:

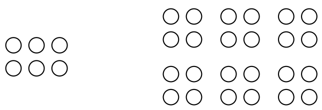

Abbildung 5 Agreement in J; shapes of left and right cells hidden

A common leading frame is used to parcel the arguments into "sections". At least on one side, ‘‘section’’ will be equivalent to ‘‘cell’’. We get the same shape of sections on both sides of the verb, and corresponding sections will be paired one-to-one.

If present, a single trailing frame for one argument means that its sections are (true) arrays of cells, while the corresponding section of the other argument is just a single cell. Distribution takes place for each cell-section pair.

As we can see, these agreement rules work in two distinct stages. The first one allows for one-to-one correspondences between cells, and the second is responsible for one-to-many distributions. Each stage is optional but both can be combined.

You can compare this a bit with APL2. J allows for a simple form of nested-array agreement. The sections are just like subarrays, with the strong restrictions that they must have the same shape and that there is only one level of nesting.

8 Exercises on Agreement

The following experiments are designed to demonstrate what is already possible *without* using the rank conjunction. You know that the left and right cells for + have rank 0. But what are the common and trailing frames in each case? In other words: where are pairings or distributions made?

```
   2 + i. 3 2 4
 2  3  4  5
 6  7  8  9

10 11 12 13
14 15 16 17

18 19 20 21
22 23 24 25

   2 100 + i. 3 2 4
|length error
|   2 100   +i.3 2 4

   2 100 1000 + i. 3 2 4
   2    3    4    5
   6    7    8    9

 108  109  110  111
 112  113  114  115

1016 1017 1018 1019
1020 1021 1022 1023

   (i. 3 2) + i. 3 2 4
 0  1  2  3
 5  6  7  8

10 11 12 13
15 16 17 18

20 21 22 23
25 26 27 28

   (i. 2 4) + i. 3 2 4
|length error
|   (i.2 4)   +i.3 2 4
```

9 Ranks Revisited

Once it is really understood how argument cells are paired or distributed or both, it should be much less forbidding to tackle the rank conjunction.

In practically all texts I have read, the topics “verb rank” and “rank conjunction” are introduced together, almost as if the former would not make any sense without the latter. So I got the wrong impression that the conjunction would somehow change a verb’s “default rank” to dissect its arguments in a way different from normal. The literature certainly invited me to think so. The reference manual for Sharp APL, Dictionary APL’s most faithful implementation, uses this model explicitly [IVERS93]. One of the rank introductions uses the following wording:

> It is often useful to change how a function is applied. [...] The *Rank Operator* [...] provides a convenient way to do this, by allowing specification of the cells to which [the function] applies. [BERNE88]

Turning to J, we find similar explanations. The implementation’s description still talks about “the intrinsic (default) ranks of a verb” [HUI92] . Up to version 7, the rank definition in J read:

> The verb *u"n* is equal to *u* except that its ranks are determined by *n* [IVERS90]

This formulation gave me serious trouble back then. (The wording has been revised for Release 2 and is clearer now.) It took me a long time to remember the principle that a conjunction constructs a *new* verb and to see clearly that the original verb and its rank remain unmodified. It is just embedded in a new context.

In particular, *u* still relies on properly ranked cells when it is invoked during step 2. The rank conjunction cannot simply slice the arguments into differently ranked cells in step 1 and and pass them on to step 2 for evaluation.

Instead, the rank conjunction provides a way to request a two-level dissection/pairing/recombination process. The newly created verb forms the outer level, and its sole role is to deliver (pairs of) cells as specified by the rank conjunction. Each of these new argument pairs is passed on to the inner level, the original verb. There, a second dissection/pairing run is executed to establish the cells as requested by the verb’s original ranks. On return, all intermediate results are recombined in two stages.

Since the agreement and recombination rules play a key role at both levels, it is important to be aware of their finer points. I was not when I tried to learn J for the first time and hence attributed effects to the wrong causes repeatedly (and probably inconsistently).

After I had understood these rules, I was able to take a new stab at reading the Dictionary. Many definitions became astonishingly simple. For example, if you look at the *table* adverb “/”, you will see that it is nothing other than a comfortable notation for a common rank request (or, more appropriately, a pairing request).

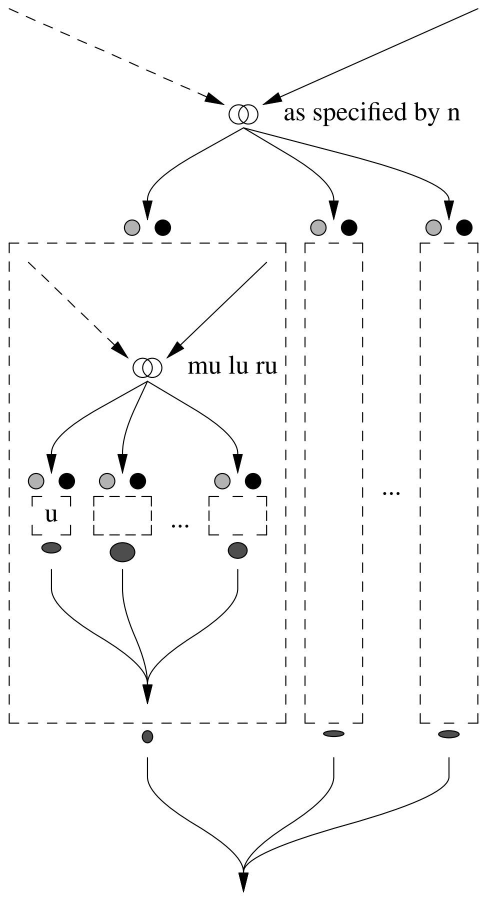

Abbildung 6 Nested evaluation of *u"n*

10 Exercises for Rank

A programmer can set up all kinds of loops and replications just by triggering step 1 where it is appropriate. I have taken the two erroneous cases from above and adorned them with rank conjunctions:

```
   2 100  +"(1 2) i.3 2 4
  2   3   4   5
104 105 106 107

 10  11  12  13
112 113 114 115

 18  19  20  21
120 121 122 123
```

```
    (i. 2 4) +"(2 2) i. 3 2 4
  0  2  4  6
  8 10 12 14

  8 10 12 14
 16 18 20 22

 16 18 20 22
 24 26 28 30
```

One might believe that the rank adverb would be useful only to limit an unbounded (or highly ranked) verb like *box* or *ravel* down to a specific rank (and to split up one big argument accordingly). But as can be seen here, dyadic + has rank (0 0), and we are using a higher rank conjunction just to achieve a two-level argument splitting, with the effect "first distribute, then pair" at the right place. This order is, by the way, impossible to accomplish with any single "verb of modified rank".

The *+"(2 2)* requests matrices from the left and the right. The frames of the actual arguments respective to 2-cells are "empty" and "3". This means there is an empty leading common prefix, and therefore no 1:1 pairings are established. The single trailing frame then causes a 1:3 distribution between the matrices. At the inner level, the verb + then will break up the received matrices into 0 0 cell pairs. The frames respective to the 0-cells are this time 2 4 on both sides. That means: 2 4 is the common frame prefix, all 0-cells will be paired, and no trailing frame causes any additional distribution. Voilà.

You might be interested in finding the answers to the following questions on your own.

Is *(? 3 2 $ 10) +"1 2 (i. 3 2 4)* a legal expression? What would be the answer if a plain + is used instead?

The *reshape* verb *$* has dyadic ranks (1 _). It is no problem to evaluate *3 2 4 $ 'abc'* , but try to find some *invalid* rank specification *n* in *3 2 4 $"n 'abc'* !

11 Verb Combinations and Their Ranks

In this section I would like to show how verbs combine, giving special attention to steps 1 and 3. My examples concentrate on the simplest form of verb combinations, functional composition.

The phrases *x u v y* and *u v y* are used to define the effects of both *atop u@v* and *at u@:v*. The commonly used model for both constructs is in Figure 7. It is compact but hides (a) what the cells are, (b) where dissections and recombinations are done, and (c) how many instances of *u* and *v* will be activated.

Figures 8 and 9 are two detailed diagrams for both conjunctions. As in previous diagrams, the "kernel verbs" in step 2 are recognizable by their dashed boxes and surrounded by steps 1 and 3. In both cases, there are *three*

```
   y          x y
   |          \/
   v           v
   |           |
   u           u
   ↓           ↓
 result      result
```

Abbildung 7 At and Atop at cell level

verbs involved (*u*, *v*, and the verb resulting from the conjunction).

Figure 8 shows *u@v* which has the ranks of *v*. Because of that, the outer level will already dissect and pair cells according to *v*. Therefore, *v*'s steps 1 and 3 become trivial. *v* can directly push its results into *u*. In Figure 9, *u@:v* has unbounded rank. Here, the outer level's steps 1 and 3 become trivial. *v* cares for its own, isolated dissection, pairing and recombination, so that *u* acts on one single intermediate result.

It is easy to forget but important that in section III, *Definitions,* all rules and identities are discussed at the cell level (or, if you prefer, for step 2 in the three-step model). "*u@v y* is *u v y*" is not guaranteed for an argument *y* ranked higher than *v*.

Likewise, one can observe that "*(f@g)@h* usually, but not always, produces the same results as *f@(g@h)*" [McINT91]. It is a good exercise to draw the three-step model pictures for such expressions and to identify the circumstances where the parentheses matter or where @ and @*:* are interchangeable. The Dictionary is very careful to define the ranks of every derived verb. This holds not only for adverbs and conjunctions, but also for hooks and forks. The ranks of any unknown verb *v* can be inquired with *v b. 0*.

12 Unbounded Scalar Functions

If you look into a current Dictionary of J, you'll find that + has ranks (_ 0 0) but not (0 0 0) as in Dictionary APL. This might appear a bit strange.

Up to version 2.9, J had only the conjunctions *u@v* and *u&v* with ranks according to *v*, but not yet their unbounded counterparts *@:* and *&:*. Traditionally scalar monads like *square* *: were given ranks (_ 0 0) in version 3 so that expressions like +/ @ *: 1 2 3 worked without the need to wrap the square into an artificially unbounded verb (+/ @ (*:"_) 1 2 3) [McINT91].

This solution was possible because those monadic scalar functions are defined in terms of scalar dyadic counterparts. For example, *:y is defined as y*y. This is nothing other than deferring the dissection into scalar cells by one level, just what a *:"_ would do, were *: scalar.

Not much later, however, @*:* and &*:* were introduced. I wonder why the unnatural (_ 0 0) definitions were not retracted at that time. +/ @: *: 1 2 3 would since then do

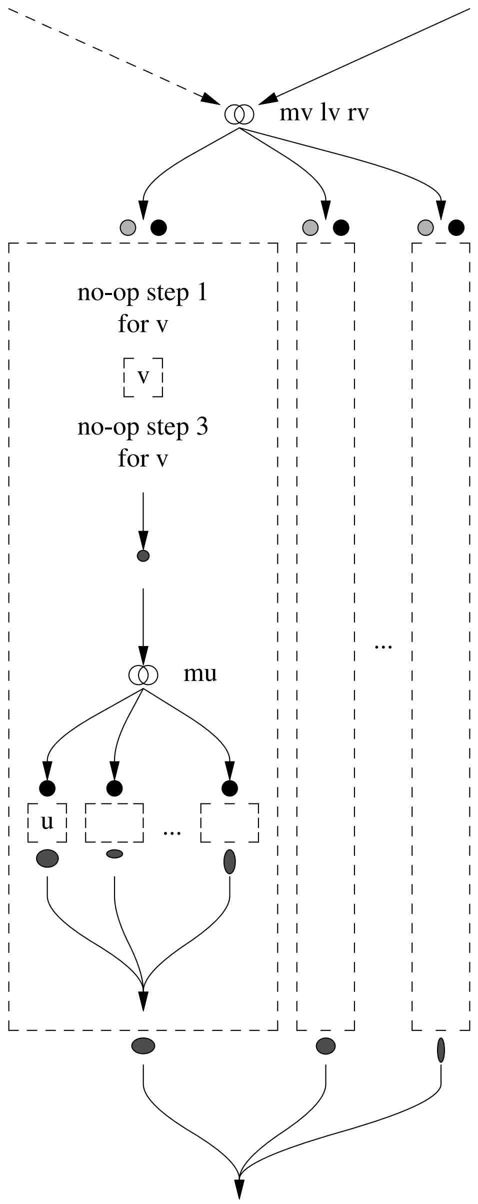

Abbildung 8 u@v

the job with *:'s ranks at (0 0 0), the vector's dissection/recombination happening inside of the squaring stage. Furthermore, this expression works for the dyadic case as well.

13 Controlled Recombinations

The section on Dictionary APL contained the example

```
     > (1 ; 2 3 ; 4 5 6)
1 0 0
2 3 0
4 5 6
```

In certain circumstances it might be useful to pad the vectors with other values than 0. One might think that the *fit*

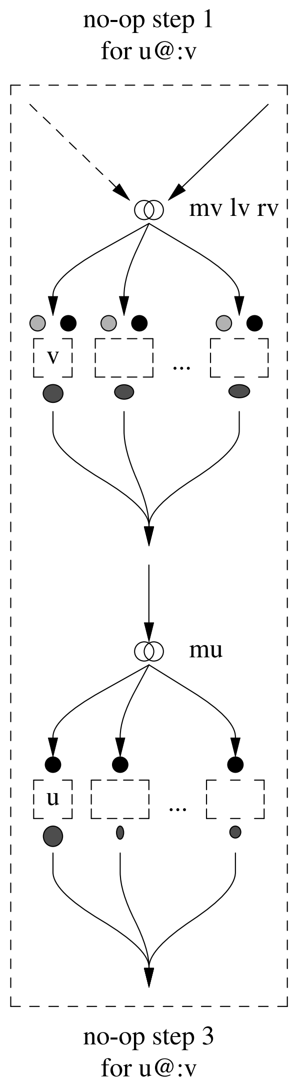

Abbildung 9 u@:v

conjunction *!.* could be used to customize the fill elements for *open*, just as it can be done for overtakes:

```
     >!.9 (1 ; 2 3 ; 4 5 6)
1 9 9
2 3 9
4 5 6
```

This might seem attractive at first but is not a viable way to go. As already explained, the zeroes slip in during step 3, the recombination of the partial results. During step 2, the *open* of an individual cell has no way to "know" what the common result shape will be. So it is not possible to get flexible padding by making *open* smarter.

The general solution would be to attach the control over fillings not to individual verbs, but to the shape assimilation process in step 3.

What I would consider to be a step in the right direction is having a conjunction at the level of the rank ″. Just like ″ gives control over argument dissection and cell pairings prior to verb evaluation, it could allow for control of the fill value for shape assimilation during the reassembly of the intermediate results. Without compromising existing behavior, the rank conjunction could accept a boxed pair containing rank and fill specifications. For example:

```
      > "(0; 9) 1 ; 2 3 ; 4 5 6
1 9 9
2 3 9
4 5 6
```

Details like the order of that boxed pair and the defaulting of specifications are an issue left to the language lawyers and designers. The important point is that it should really be *rank* itself that should be extended. A separate, new *fill conjunction* defined to control step 3 could not do the job in concert with *rank*: applying both conjunctions to a verb would result in nested verbs so that steps 1 and 3 would lie on different levels. No matter what the order of the nesting were, there would never be anything to fill.

Why should one be content with mere fill *values*? A programmer could provide a fill *function* that should be triggered in the context of II.B's shape assimilation. The function would be called with the short partial result and the requested shape. A programmer could designed it to have any desired effect, as long as results of a common shape are generated. Common choices are:

padding with a constant
padding by repeating the last item
replicating the entire value
padding on the right instead of the left

Notice that, while these fill functions are easily programmed per se, the context-dependent target shape is not known a priori. For this reason, it cannot be done within the current language. (The overtake is an example of the case where the programmer explicitly requests the target shape.)

A fill function could be specified in two ways: as a gerund in place of the fill value in the rank extension proposed above, or as an intrinsic characteristic of a verb, associated with a verb just like *obverses* and *identity functions*.

14 Conclusions

Currently available introductions to the language based on examples and small experiments often fail to point out the special circumstances they rely on. As a result, newcomers can easily get trapped in over-generalizations and operate for a long time under wrong assumptions. Both can prevent the construction of a consistent model for the language and lead to the abandonment of all efforts in frustration.

To gain a solid understanding it was necessary for me to shift my conceptual model of APL's inner workings. This may be due to the personal circumstance that I jumped from flat-APL to J without going the route over one of the second generation dialects.

A strict separation between a verb's effects on the one hand and effects due to the common evalution rules on the other is necessary to interpret experiments correctly. This holds especially true when one advances to composed and derived verbs. A model which explicitly shows all possible effects and interactions has been suggested.

15 Acknowledgements

I am very thankful to many people. Prof. D. McIntyre for encouraging me to sit down and start writing; B. Benzel, J. Evans, J. J. Münter, C. Kämmerer, J. Schumacher, and U. Thürmann for willingly adding columns to rows and rows to matrices; all draft reviewers for their comments. My biggest thanks go to Prof. A. Frank. She did not hesitate a second and sacrificed a great deal of her precious time when I asked her for some advice to get this paper into a decent form. You really wouldn't like reading many parts like they had originally been :-)

References

BENKA84.
J. Philip Benkard, "Rank vs Depth for Array Partitioning," *APL Quote Quad,* 14, 4, pp. 33-39 (1984).

BERNE88.
Robert Bernecky, "An Introduction to Function Rank," *APL Quote Quad,* 18, 2, pp. 39-43 (1988).

BROWN91.
James A. Brown, "TryAPL2 for free," IBM Corporation (August 1991). Introduction accompanying the demonstration product.

BrCr91.
James A. Brown and Harlan P. Crowder, "APL2: Getting Started," *IBM Systems Journal,* 30, 4, pp. 433-445 (1991).

EisPe93.
Murray Eisenberg and Howard A. Peelle, "Confessions of Two APL Educators Learning J," *APL Quote Quad,* 24, 1, pp. 78-87, ACM Press, Toronto (August 1993).

EKLOF93.
Mark D. Eklof and Eugene McDonnell, "Programming Language APL, Extended," p. 127, section 9.3.3, International Standards Organisation (Jan 6, 1993). Committee Draft 1.

HUI92.
Roger Hui, "An Implementation of J," p. 20, Iverson Software Inc., Toronto (1992).

IVERS87.
Kenneth E. Iverson, "A Dictionary of APL," *APL Quote Quad,* 18, 1 (Sept. 87).

IVERS90.
Kenneth E. Iverson, ''Dictionary of J,'' Iverson Software Inc., Toronto (1990 and continuing).

IVERS93.
Kenneth E. Iverson, ''SHARP APL Reference Manual,'' Iverson Software Inc., Toronto (1993).

MCINT91.
Donald B. McIntyre, ''Mastering J,'' *APL Quote Quad,* 21, 4, pp. 264-273 (August 1991). APL91 Conference Proceedings.