Rational and Variable-precision Floating Point Numbers

Bob Smith
Sudley Place Software
18 Jun 2016

Introduction

Rational and Variable-precision Floating Point (VFP) numbers are two new datatypes in NARS2000. **Rational numbers** complement the existing 64-bit integer datatype to provide infinite precision (or `WS FULL`) at the cost of some performance. Similarly, **VFP numbers** complement the existing 64-bit floating point (IEEE-754) datatype to provide more precision (as much as the user cares to specify) again with some cost in performance.

There is no separate datatype for infinite or extended precision integers. Instead, they are represented as a special case of rational numbers. In the discussion below, the phrase **rational integer** means a rational number whose denominator is one.

► Throughout this discussion the similarity between integer and rational numbers as well as floating point and VFP numbers will become apparent.

Rationale

Simply put: precision. If the 53 bits of precision in the floating point result of, say, `2÷3` is not enough, you now have two more choices: one as an exact number, and one as a binary floating point number with as much precision as you care to specify.

For example,

```
      ⎕PP←60 ⋄ ⎕FPC←512
      2÷3
0.666666666666666
      2÷3x
2r3
      2÷3v
0.666666666666666666666666666666666666666666666666666666666667
      2*200
1.606938044258990E60
```

```
      2*200x
1606938044258990275541962092341162602522202993782792835301376
      ○1
3.141592653589793
      ○1v
3.141592653589793238462643383279502884197169399375105820974944
```

Constants

Rational constants may be entered by suffixing an integer constant with an x (for a rational integer) or separating the numerator and denominator with an r (for a rational number) as follows:

- `123x` for the constant `123` (the suffix `x` is but a shorthand for `r1`)
- `123r4567` for the constant `123÷4567`

VFP constants may be entered by suffixing the integer or floating point constant with a v as follows:

- `123v` for the constant `123`
- `123.4567v` for the constant `123.4567`
- `123.4567E3v` for the constant `123.4567E3`

The above constants (except for the suffix `x`) may be used in other constants such as `1r4p2` to generate a shorter and more accurate value for `π²/4` than, say, `((○1)*2)÷4`, without requiring any computation.

Precision

Rational numbers have **infinite precision**. They are stored with a separate numerator and denominator, both of which are exact numbers in the sense that their size (and hence precision) grows limited only by the available workspace.

VFP numbers have **finite** and **user-controlled variable precision**. The default precision at startup is controlled by the value of the system variable `⎕FPC`. The system default of this value is `128` in units of bits of precision of the mantissa (the digits) of the number, not counting the exponent (which is of fixed size). The current precision may be changed as needed by assigning a new value to the system variable. All newly created VFP numbers will have the new precision – the precision of VFP numbers already present in the workspace does not change.

Generally, precision is set once for a particular application and unchanged thereafter. Although not recommended, it is possible to mix VFP numbers of different precisions in a single array – presumably you **really** know what you are doing. The system function `0 ⎕DR` may be used to display an array's precision(s).

Datatype Propagation

Generally, the datatype of rational and VFP constants propagates through a calculation. That is, if you start with a rational number and don't calculate with irrational or transcendental functions, you'll end up with a rational result, and if you start with a VFP number, you'll end up with a VFP result.

An example from the programming problems site ProjectEuler.net illustrates this point. Problem #48 asks what are the low-order ten digits of the sum of the first thousand instances of N^N?

The obvious expression `¯10↑⍕+/*⍨⍳1000` at first sight seems to solve the problem until you realize that it quickly runs afoul of the limited precision of 64-bit integer and floating point numbers. Clearly, this is a problem for the infinite precision of rational integers.

As `⍳1000` generates the first thousand integers as an integer datatype (actually an Arithmetic Progression Array), `⍳1000x` generates the same values as rational integers. Next, `*⍨⍳1000x` generates the first thousand instances of N^N as exact rational integers, and unlike its integer counterpart, there is no overflow to floating point, just an increase in precision (as well as space used in the workspace). Then, `+/*⍨⍳1000x` sums them into a single 3001-digit rational integer, and finally `¯10↑⍕+/*⍨⍳1000x` converts the large integer to characters and extracts the low-order ten digits – `9110846700` – all in a small number of milliseconds.

Note how we started with an obvious expression that failed because of its limited precision, and made a single change to suffix the constant `1000` with an `x` to convert it to a rational integer which then propagates through the calculation with infinite precision to yield the correct result.

Display

Rational integers are displayed as an integer with no special adornment; rational non-integers are displayed as a numerator and denominator separated by an `r` as in `34r9`. As with the integer datatype, the numerator and denominator of a rational number are displayed exactly, unaffected by the current setting for Printing Precision (`⎕PP`).

```
      !40
8.159152832478977E47
      !40x
815915283247897734345611269596115894272000000000
      +\÷⍳10x
1 3r2 11r6 25r12 137r60 49r20 363r140 761r280 7129r2520 7381r2520
```

VFP numbers are displayed as decimal numbers to the precision inherent in the number or `⎕PP`, whichever is smaller, just as floating point numbers are displayed. For example,

```
      ⎕PP←100
      ⎕FPC←64
      ○1
3.141592653589793
      ○1v
3.1415926535897932385
      ⎕FPC←128
      ○1v
3.1415926535897932384626433832795028842
```

where both of the above displays were limited by the precision of the number, not ⎕PP.

However, the first of the following displays **is** limited by ⎕PP:

```
      ⎕FPC←128
      ⎕PP←20
      !40v
8.1591528324789773435E47
      ⎕PP←80
      !40v
8.15915283247897734345611269596115894272E47
```

Formatted Display

The system function ⎕FMT has been enhanced to allow formatting of rational numbers via the (new) R-format specifier. For example,

```
      'R4.2' ⎕FMT ∘.÷⍨⍳6x
1    1r2 1r3 1r4 1r5 1r6
2    1   2r3 1r2 2r5 1r3
3    3r2 1   3r4 3r5 1r2
4    2   4r3 1   4r5 2r3
5    5r2 5r3 5r4 1   5r6
6    3   2   3r2 6r5 1
```

Moreover, the Symbol Substitution (S<…>) feature of ⎕FMT allows you to substitute a different symbol for the default r used to separate the numerator and denominator of a rational number, as in

```
      'S<r/>R4.2' ⎕FMT ∘.÷⍨⍳6x
1    1/2 1/3 1/4 1/5 1/6
2    1   2/3 1/2 2/5 1/3
3    3/2 1   3/4 3/5 1/2
4    2   4/3 1   4/5 2/3
5    5/2 5/3 5/4 1   5/6
6    3   2   3/2 6/5 1
```

Datatype Promotion

For the most part, rational numbers beget rational numbers and VFP numbers beget VFP numbers. However, when irrational, transcendental, and certain other functions are used, rational numbers beget VFP numbers. For example,

```
      ⎕pp←40
      *1
2.718281828459045
      *1x
2.718281828459045235360287471352662497757
```

where the datatype of the two results are floating point and VFP, respectively. That is, in a manner similar to how some primitive functions with integer arguments may return floating point results, when a rational number is used as an argument to a primitive function that can't return a result with infinite precision, it returns a VFP number.

► The reason irrational, transcendental, and certain other functions on rational numbers do not return rational numbers is that, by definition, the result of such a function is, in general, not representable as a rational number; instead, VFP numbers are better suited to represent irrational results where the end user may control exactly how much precision is desired in an obviously inexact number.

Two special functions are the prime decomposition (`πR`)/number theoretic (`LπR`) functions. In these cases, fractional or VFP right arguments are converted to integers or rational integers, respectively, which is the datatype of the result except for `0πR` (Primality Test) which always returns a Boolean result regardless of the type of `R`.

Ignoring purely structural functions, the list of functions that produce VFP numbers given rational numbers is as follows:

- Power: `*R` and `L*R` (except when `R` is a 32-bit integer, in which case the result is a rational number)
- Logarithm: `⍟R` and `L⍟R`
- Pi Times and Circle functions: `○R` and `L○R`
- Root: `√R` and `L√R`
- Factorial and Binomial: `!R` and `L!R` (except when the arguments are rational integers, in which case the result is a rational integer)

Beyond the ones mentioned above, the list of functions that **don't** produce a rational or VFP result given those argument(s) is as follows:

- Depth: `≡` (Integer)
- Dyadic Comparison: `= ≠ < ≤ ≥ > ≡ ≢` (Boolean)
- Nand and Nor: `⍲ ⍱` (Boolean)
- Grade Up/Down: `⍋ ⍒` (Integer)
- Index Of: `⍳` (Integer or Nested Integer Vectors)
- Member Of: `∊` (Boolean)
- Find: `⍷` (Boolean)

- Subset and Superset: ⊆ ⊇ (Boolean)
- Format: ⍕ (Character)
- Indices: ⍸ (Integer)
- Array Lookup: ⍸ (Integer or Nested Integer Vectors)
- Factor and Number-theoretic: π (Boolean, Integer, or Rational)

Otherwise, rational argument(s) produce rational result(s) and VFP argument(s) produce VFP result(s).

Datatype Demotion

It is common in APL implementations to demote datatypes where appropriate. For example, the constant `1.0` might actually be represented as an integer or even Boolean datatype. The idea is there is no loss of precision and the storage is typically smaller which might lead to a more efficient algorithm when next used, so why not?

► With rational and VFP numbers those reasons no longer apply. While the constant `1x` might have the same precision as the constant `1.0`, the difference in latent precision between the two is vast. In fact, in order for datatype propagation of rational and VFP numbers to work at all, we must be careful **not** to demote them automatically to a smaller datatype. Otherwise, it would require an intolerable degree of analysis on the part of the programmer to ensure that the desired datatype (rational or VFP) remains in effect throughout a calculation.

Conversions

To convert manually from one datatype to another:

- Integer to rational, use `0x+`
- Integer, floating point, or rational number to VFP, use `0v+`
- Rational integer to 64-bit integer (possibly losing some precision), use `⍎⍕`
- Rational non-integer to floating point number (possibly losing some precision), use `⍎⍕0v+`
- VFP number to floating point number (possibly losing some precision), use `⍎⍕`
- VFP number to rational – at the moment, no APL method exists although the underlying libraries do have such a function

Comparisons

Comparisons between a floating point, rational, or VFP number and any other number is sensitive to the current setting of Comparison Tolerance (`⎕CT`) because, unlike pairs of integers, such pairs of numbers can be arbitrarily close to each other. Only comparisons between two integer forms (Boolean, integer, APA, and HCxI) are not sensitive to `⎕CT`.

Integer Tolerance

Both rational and VFP numbers may be used where the system ordinarily requires an integer (such axis coordinates, indexing, left argument to structural primitives, etc.) just as the system tolerates floating point numbers in those contexts if they are sufficiently near an integer. In all cases, the system attempts to convert the non-integer to an integer using the fixed system comparison tolerance (at the moment, `3E¯15`).

Infinities

Support for `±∞` has been extended to rational and VFP numbers in the same manner as it applies to 64-bit integers and 64-bit floats. That is, the same cases covered by the system variable `⎕IC` (Indeterminate Control) also apply to infinite rational and VFP numbers. Moreover, infinite numeric constants may be entered, for example, as

- `∞x`
- `∞r1`
- `∞v`

Also constants such as `2r∞` resolve to `0x`.

New And/Or Different Behavior

- Both roll (`?R`) and deal (`L?R`) on rational integers use a built-in random number generator so as to use the entire range of rational integers – this algorithm uses its own internal seeds that are much more complicated than the simple integer seed that is `⎕RL` (Random Link). Thus `⎕RL` is unchanged by these functions on rationals.

  For example, if you need **really** large random numbers

  ```
        ?10*60x
  370857192605742854709703007683731949504799559659692534573173
  ```

- Matrix inverse (`⌹R`) and matrix division (`L⌹R`) on rational or VFP arguments each have two limitations above and beyond that of normal conformability:

  - for a square right argument that it be non-singular, and
  - for an overdetermined (`>/⍴R`) right argument that the symmetric matrix `(⍉R)+.×R` be non-singular.

  These limitations are due to the algorithm (Gauss-Jordan Elimination) used to implement matrix inverse/divide on rational and VFP numbers.

  Integer and floating point arguments are not subject to these limitations because they use a more general algorithm (Singular Value Decomposition) that produces a unique result even for singular arguments (e.g., `⌹5 3⍴0`).

Conclusions

The new datatypes offer several benefits:

- They extend the precision of existing integer and floating point datatypes to a much greater level.

- As integer blows up to floating point, rational blows up to VFP, providing a natural parallel progression for irrational and transcendental primitive functions.
- There is a close similarity between integer and rational numbers as well as floating point and VFP numbers.
- Datatype propagation without demotion allows one to code an algorithm in either of the new types easily and without the need for detailed analysis of the datatype in intermediate results.
- All primitives extend naturally to encompass the new types as numbers.
- The notation for constants builds on existing point notation formats.

Acknowledgments

The designers of J are thanked for having the foresight to include rational numbers as a separate datatype.

The following LGPL libraries have been used to provide support for these datatypes:

- MPIR (Multiple Precision Integers and Rationals) at mpir.org.
- MPFR (Multiple Precision Floating-Point Reliable Library) at mpfr.org.

References

The most up-to-date version of this paper may be found at http://wiki.nars2000.org/index.php/Rational_and_VFP_Numbers.