Progress In NARS2000
October 2013 to September 2015

Bob Smith
Sudley Place Software
Originally Written
14 Sep 2015
Updated
23 Sep 2015

Released Features

Hypercomplex Numbers

By hypercomplex, I mean Complex (ℂ), Quaternion (ℍ), and Octonion (▫)▫(CHO) numbers, which together round out the set of a particular kind of algebraic structures (normed division algebras) starting with the Real (ℝ) numbers. A CHO number has, respectively, 2-, 4-, or 8-coefficients which as a set may all be of one data type, to wit

- Fixed precision (64-bit) integers,
- Fixed precision (64-bit) floating point numbers,
- Multiple precision integer/rational numbers, or
- Multiple precision floating point numbers.

The end result is to add twelve new datatypes to APL bringing the total number of datatypes to 21. Not surprisingly, this was a major effort over three solid months.

At the moment, I'm not quite ready to release this feature, but it is available in alpha version from my website: http://nars2000.org/download/binaries/alpha/. By alpha version, I mean

that (modulo bugs), all still unsupported features should signal a `NONCE ERROR` or a `DOMAIN ERROR` – if it doesn't, that's a bug.

More on Hypercomplex numbers and their implementation in a later talk.

Grammars

Point Notation Grammar

The forms of constants in NARS2000 are many and varied.

- Fixed precision integer (`1234`)
- Fixed precision decimal/exponential floating point (`1.2E¯34`)
- Multiple precision integer/rational (`12x` or `12r34`)
- Multiple precision floating point (`1.2v` or `1.2E¯34v`)
- CHO fixed precision integer (`2i3`)
- CHO fixed precision floating point (`1.2i3.4j5`)
- CHO multiple precision integer/rational (`1r2i3r4j5x`)
- CHO multiple precision floating point (`1.2i3.4j5v`)
- Euler point (`2x3 ↔ 2×(*1)*3`)
- Pi point (`2p3 ↔ 2×(○1)*3`)
- Base point (`16b01FF ↔ 16⊥0 1 15 15`)

The entire grammar is handled by one file which runs through a YACC-like program to produce a C source file which is then compiled into NARS2000. It is a tribute to the designers of LALR compiler-compilers. I would not like to tackle the above notation parsing problem by writing a program from scratch. This problem is ideally suited to a grammar representation.

Function Header Grammar

Another grammar in NARS2000 parses user-defined function/operator

headers. What with the many variations in syntax used for optional and multi-named results/left argument/right argument in both functions and operators as well as left and right operand syntax for operators, this can be a complicated string to parse. For example, the following is a maximum feature function header:

```
{Z1 Z2}←{L1 L2 L3} (LO F[X] RO) (R1 R2 R3 R4);A B ⍝Hi
```

which defines a dyadic operator `F` with an axis value, left and right operands, a four-element right argument, an optional three-element left argument, a non-displayable two-element result, two local names, and a comment.

Recently, spaces were added as an alternative separator in the list of locals as per APL2:

```
∇Z←foo R;a b c d
```

Control Structure Grammar

A third grammar parses control structures on a whole defined function. This grammar is becoming difficult to maintain and could use a re-write.

Syntax Grammar

A fourth grammar was used to parse APL syntax. However this attempt proved to be very difficult both to write and maintain, so I abandoned it in favor of 2by2. The problems with parsing APL syntax in an LALR grammar are several:

- the dual nature of hybrid symbols (`/ ⌿ \ ⍀`)
- parenthesized functions and operators
- bracket indexing applied to variables vs. functions vs. operators

among others. On the other hand, what is a problem for an LALR grammar is easy to handle with a parser based on binding strength such as 2by2.

Syntax Coloring

With the availability of Hypercomplex numbers, making their display more readable is important what with the additional special separators involved. To this end, NARS2000 supports a syntax-coloring category of Point Notation which covers these symbols as used in constants: `beEijJklprvx`. For example, I find it easier to read `2.1i3j4k5` with the separators in a different color.

Implement ⎕T

This niladic system function is a simple idea (suggested by David Rabenhorst) to return the current CPU Tick Count in seconds. This makes for precise timings by differencing such as

```
a←⎕T ⋄ ×○0i1 ⋄ ⎕T-a
```

Implement Reduction Of Singletons

You may remember from Minnowbrook APL Workshop 2010, that I proposed this concept. I've now polished it to the point that I released it in a version of NARS2000 earlier this year. One sticking point was the treatment of `,\1 2 3` which previously (i.e., on your systems), returns `1 (1 2) (1 2 3)`, but on NARS2000 now signals a `DOMAIN ERROR`. I'm now more then ever convinced that this is a good idea. There is a more recent version of the PDF file on my http://sudleyplace.com/APL website – see “Reduction of Singletons”.

Implement Negative Zero (¯0)

This idea stems from implementing infinity, in particular signed infinities. There are two ways to adjoin infinity to a programming language such as APL: unsigned (i.e., one infinity such that `-∞ ↔ ∞`), or signed (i.e. two infinities, positive and negative such that `-∞ ↔ ¯∞` and `-¯∞ ↔ ∞`). Early on, I chose to implement the latter and integrated it into the implementation. This revealed a lot of indeterminate cases which were handled by a new system variable `⎕IC` which allows the user to control

each case individually.

Long after that, I realized there was a failing identity, ÷÷X ↔ X, which should be true on all numeric X (ignoring precision loss), even on zeros and infinities. The missing case was ÷÷¯∞ where without support for negative zero (¯0) incorrectly returns ∞, not ¯∞.

This excellent idea has been present in IEE-754 compliant hardware since the days of the Intel 8087 chip some three decades ago. Very unfortunately it falls short in this context because it relies on hardware support in integer arithmetic not present in modern systems, and not likely to be in any future system. Floating point numbers, both fixed precision in hardware (IEEE-754 and following) and multiple precision in software (MPFR), each support negative zero, and identically so. The problem is that there is no support for negative zero in either integer numbers in hardware nor multiple precision integer and rational numbers in software (MPIR). This means that every time the implementation, say, negates an array of integers, if only one value is zero, the whole array blows up to floating point. This was not a problem when I introduced signed infinities as it often was replacing an error, so blowing up to fixed precision floating point wasn't seen as a downside.

Frankly, the problem for me was not so much the precision and performance hit, but the extraordinary programming overhead in the implementation needed to handle all of the exceptions which could occur from something as simple as multiplying two numbers. This proved to be an exceptional burden when implementing Hypercomplex numbers, as I use many internal functions that take a CHO number or two and return the appropriate CHO number, passing the values as structures rather than pointers. This allows me to write much more natural code in C for CHO fixed precision floating point algorithms, but not if I have to worry about type promotion at every turn. It also meant that I had trouble testing the Complex (Gaussian) integer code as the result could be promoted to Complex floating point at any time. Unbeknownst to me, I was no longer testing Complex integer code defeating the whole purpose of Gaussian integers in the first place.

Eventually, I abandoned that effort and disabled all of the code to handle negative zero, guarding it with an invisible user preference. If you really want it, it's there, but not by default. Maybe I'll enable it again as a local feature, say, `□FEATURE[2]`, where `□FEATURE[1]` controls whether or not to allow negative indexing, another idea that didn't withstand the test of usage.

If I can find the time, I might back off from two signed infinities to one unsigned infinity which doesn't require negative zero. For those implementors of you who were considering implementing signed infinities, once again procrastination is the winning strategy.

Convert Strands As A Whole

One tricky situation with numeric strands and multiple precision numbers is that treating the elements individually instead of as a whole can lead to undesirable results. For example, one way to process the strand `1111111111111111111111111111111111111111 1r2` is to convert the string of `1`s to a floating point number (which is what you would do if it were alone on the line), then merge (catenate) it with `1r2`, first promoting each to multiple precision floating point. A more intelligent way is to scan the string of `1`s noting that it can be represented as a multiple precision integer/rational number, convert it floating point, and then when the `1r2` comes along re-convert the string of `1`s to a multiple precision integer/rational number and merge with `1r2` to yield a multiple precision integer/rational strand.

A different example is `0.1 5v` which is preferably converted to a multiple precision floating point number with `0.1` represented as a multiple precision floating point number natively, rather than converted from a 53-bit precision floating point number to a multiple precision floating point number.

As you can see, the goal is to avoid losing precision by unnecessarily promoting to fixed or multiple precision floating point. Again, this issue is of concern only if you support multiple precision integer/rational numbers.

Release 2by2 Parser

This parser technique comes from the paper by Bunda and Gerth[1]. It is easy to modify to implement different syntax rules. I've copied the rules of APL2 with extensions to cover Trains, Anonymous Functions/Operators, Hybrid symbols (/ ≠ ⍀ \), special handling of niladic functions, etc. At the same time, I revamped the way I handled reference counts, which was quite difficult until I got my head around how to handle derived functions.

Google Code

At the last meeting I mentioned that I was delighted to move my source code from hosting my own SVN repository to GoogleCode. You may already be aware that GoogleCode is folding, so once again I moved my source code this time to SourceForge. Then a couple of months ago, SF took a hit and went down for a week. What a pain!

I also mentioned then that I had moved the discussion forum from hosting it myself to a commercial service. Now, that service has stopped answering tech support questions, so once again I'm in the market for another discussion forum host. I'm open to suggestions.

Switch To New Compiler

Compilers get better over time, however the work to switch to a new compiler can be non-trivial, even daunting. Eventually I decided to bite the bullet and switch from Visual Studio 2008 to VS 2013, and I'm glad I did. While tehre were many changes to the build process, it did improve compile times as well as make available additional features of the C language.

Current & Future Work

Hypercomplex Numbers

While more work remains to be done, I've already prepared to bundle the CHO code together and release it as a LGPL library so that anyone can take advantage of these algebraic structures in manner similar to how MPIR and MPFR are used.

Access to External Processes

What with Hypercomplex numbers available, there is a greater need to show them off, such as plotting them rotating a multi-dimensional figure. This is a job for Shared Variables and Name Association and possibly a system function or two.

References

1. "APL two by two-syntax analysis by pairwise reduction", Bunda, J.D. and Gerth, J.A., APL '84 Proceedings of the International Conference on APL