Serializing Nouns

N. E. Davis ~lagrev-nocfep, Brian Klatt ~zod,*
& Sam Parker ~zod†
Zorp Corp, Zorp Corp, & —

Abstract

Noun serialization is commonly used for Nock communication, both between instances like Urbit ships and with the runtime and the Unix host operating system. This article describes and compares the two principal conventions for representing nouns in slightly compressed form as byte arrays, as well as introduces variant encodings for educational purposes.

Contents

*Brian Klatt contributed the description of `++jam`.
†Sam Parker contributed the description of `++bulk`.

Manuscript submitted for review.

1 Introduction

The Nock combinator calculus deals in nouns and does not know about bit encodings or memory layouts. A Nock interpreter (runtime) must, however, deal with the practicalities. In other words, there must be a way of writing down an abstract binary tree (consisting of cells/pairs and atoms) as an actual, physical array of bits in memory. Every possible Nock noun can be represented as a finite sequence of bytes (an atom), and there are multiple ways to do so.[1]

A noun serialization strategy is rather like a Gödel numbering in that it systematically encodes a mathematical object (a noun) as a number (an atom). Unlike Gödel numbering, which classically serially encodes the symbols of mathematical statements, noun serialization encodes a binary tree structure.[2] Because a noun may be any atom—and atoms cannot have leading zeroes—both structure and value need to be unambiguously encoded and cannot be simply delimited (as by a `0` bit or similar). There are two basic strategies to encode a noun as an atom:

1. Run-length serialization, with or without references.

2. Directed graph serialization, depending on a reentrant graph encoding.

Both of these embody a tradeoff between simplicity, size, and speed. Below, we describe both of these strategies and some new variants which offer possible advantages. Reversible noun encoding is essential for Nock communication, both between Nock execution layers such as Vere and NockVM, and between the runtime and the host operating system.

[1] The converse is not true: not every atom represents a valid deserialization or conversion from a graph encoding.

[2] This also echoes the way that S-expressions are encoded in Lisp.

2 Naïve Serialization

A noun may be an atom or a cell, and so we first propose to label nodes by type, with `0` for atom and `1` for cell.

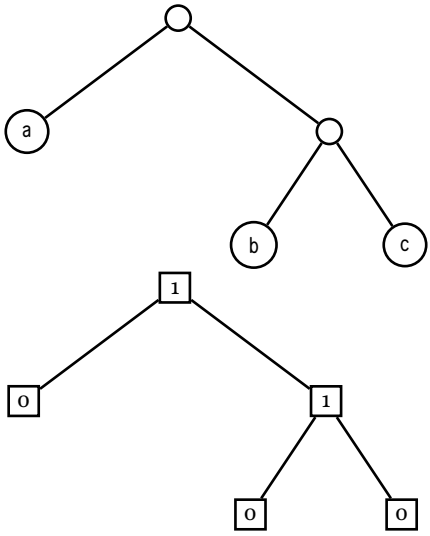

Figure 1: The noun `[a b c]` (the same as `[a [b c]]`) as a binary tree and its metadata labeling in the simplest scheme.

The noun in Figure 1 is represented as a binary tree, where each node is labeled with its type (atom or cell) and its value (if applicable). To serialize this noun, we can perform a pre-order traversal of the tree, recording the type and value of each node. The serialized form would be (in LSB order):

```
c 0 b 0 1 a 0 1
```

The issue, however, is that in decoding this bitstream we do not know where an atom value (such as `a`) ends. After all, the point of serialization is to be able to deserialize the bitstream back into the original noun. Thus we need a way to indicate the length of each atom.

Let's augment this scheme by providing length information for each atom. One basic idea is to include a string of `0` bits as the least-significant bits (LSB) of the atom which indicates the length of the atom in bits. Since the atom could be zero, we also delimit the length with a `1` bit. (See Table 1 for examples.)

$$a = a_{\ell-1}a_{\ell-2}\ldots a_1a_0 \xrightarrow[\text{encode}]{} \bar{a} = a_{\ell-1}a_{\ell-2}\ldots a_1a_0\ \texttt{1}\ \texttt{0}^n,$$

where $\bar{a}$ is the encoded atom a and a has length ℓ bits.[3]

Table 1: Example atom encodings with length information appended as trailing zeros.

Atom	Length (bits)	Encoded Form
`0b0`	1	`0b010`
`0b1`	1	`0b110`
`0b10`	2	`0b10100`
`0b11`	2	`0b11100`
`0b101`	3	`0b1011000`
`0b111`	3	`0b1111000`
`0x70`	7	`0b111000010000000`
`0xff`	8	`0b111111111100000000`

For instance, the atom `0b101010` (decimal `42`) could be encoded as:

```
:: atom 0x2a
> `@ub`(gel 0b101010)
0b101010
:: cell [0x2a 0x3b]
> `@ub`(gel [0b101010 0b111011])
0b1.101010@0.111011
```

The simplest way to encode a noun as a binary tree in an atom (byte array) without compression is to utilize bits to mark atom (`0`) vs. cell (`1`), the length of the atom in unary of `1`s terminated by `0`, and the actual value. (Since the leading zeros

[3] We will use the overbar notation $\bar{a}$ throughout to notionally indicate the encoded form of a, regardless of the method used.

are stripped, they may need to be added back in.) This is in least-significant *byte* (LSB) order, so you should read the written atom starting from the *right* in binary.

Thus the atom `0b0` would simply be encoded as:

```
:: 0x0 = 0 for atom; 1 for length (special case);
::       0 for end of length; 0 for value
> `@ub`(gel 0b0)
0b10
```

interpreted as (rightmost, LSB) `0` for atom, run length of `1`, and value `0` (stripped). I.e., `0b010` or

←

0 1 0

Other values include:

```
:: 0x1 = 0 for atom; 1 for length;
::       0 for end of length; 1 for value
> `@ub`(gel 0b1)
0b1010

:: [0x0 0x1] = 1 for cell; zero, then one
> `@ub`(gel [@0b0@ 0b1])
0b1.010@0.010@1

:: [0x1 0x0] = 1 for cell; one, then zero
> `@ub`(gel [@0b0@ 0b1])
0b10@1.010@1

:: 0x2 = 0 for atom; 2 for length;
::       0 for end of length; 2 for value
> `@ub`(gel 0b10)
0b10.0111

:: 0xff
> `@ub`(gel 0xff)
0b11.1111.1101.1111.1110

:: [[0x0 0x1] [0x2 0x3]]
> `@ub`(gel [@0b0@ 0b1])
0b10@1.010@1
```

Our code implementation for `++gel` is as follows:

```
!:  |%
++  gel
  =gel !:  |=  a=*
  ^-  @
  =+  l=0
  =+  b=0
  =<  -
  |-
  ?^  a
    =+  lv=$(a -.a)
    =+  rv=$(a +.a)
    =+  [c l]=(mash rv lv)
    [(con (lsh [0 1] c) 0b1) +(l)]
  ?:  =(0 a)  [0b10 4]
  :: need another mash in here for unary length
  =+  [c l]=(mash [a l] [b +((met 0 b))])
  [(con (lsh [0 1] c) 0b0) +(l)]
:: length of atom in unary
++  len
  |=  a=@
  ^-  @
  (fil 0 (met 0 a) 0b1)
:: mash two atoms together
++  mash
  |=  [a=[p=@ l=@] b=[p=@ l=@]]
  ^-  [c=@ l=@]
  :-  (con (lsh [0 l.b] p.a) p.b)
  (add l.a l.b)
--
```

Note that `l` is not the length of an atom in unary, but the length of the encoded noun in binary.

3 Practical Serialization

Whatever the pedagogical advantages of `++gel`, the algorithm has practical flaws: it is subject to collisions XXX and it is verbose.[4] `++jam` improves the basic strategy by altering the RLE

[4]Note the claim of `~dozreg-toplud`, p. TODO of this issue, that an operational Arvo instance may have up to 1.66×10^{21} nouns, greatly reduced by

algorithm slightly and supporting internal references for noun subtrees that have already been encoded.

`++jam` converts a noun into a buffer and deduplicates repeated subtrees. It walks subtrees and encodes each in a way that allows for efficient storage and retrieval, while also permitting references to previously encoded values.

- get Bryan notes

special-cases o

The new RLE calculation is to post the number plus one in binary rather than unary (e.g., for the length would be `0b010`).

(Since there is a value less significant than the length, the leading zero is not lost but serves as a divider.)

```
++  jam
  ~/  %jam
  |=  a=*
  ^-  @
  =+  b=0
  =+  m=`(map * @)`~
  =<  q
  |-  ^-  [p=@ q=@ r=(map * @)]
  =+  c=(~(get by m) a)
  ?~  c
    =>  .(m (~(put by m) a b))
    ?:  ?=(@ a)
      =+  d=(mat a)
      [(add 1 p.d) (lsh 0 q.d) m]
    =>  .(b (add 2 b))
    =+  d=$(a -.a)
    =+  e=$(a +.a, b (add b p.d), m r.d)
    :+  (add 2 (add p.d p.e))
      (mix 1 (lsh [0 2] (cat 0 q.d q.e)))
    r.e
  ?:  ?&(?=(@ a) (lte (met 0 a) (met 0 u.c)))
    =+  d=(mat a)
    [(add 1 p.d) (lsh 0 q.d) m]
  =+  d=(mat u.c)
  [(add 2 p.d) (mix 3 (lsh [0 2] q.d)) m]
```

A Python example of `++jam` is included in Appendix A.

structural sharing.

3.1 `newt` Encoding

"Newt" encoding is a runtime-oriented extension of `++jam`-based noun serialization which adds a short identifying header in case of future changes to the serialization format. A version number (currently a single bit) precedes a RLE serialization length followed by the `++jam` serialization of the noun. The version number is currently `0b0`.

```
V.LLLL.JJJJ.JJJJ.JJJJ.JJJJ.JJJJ.JJJJ
```

where `V` is the version number, `L` is the total length of the noun in bytes, and `J` is the `++jam` serialization of the noun.

Runtime communications vanes like `%khan` and `%lick` utilize this encoding locally. It is exclusively used as a host OS runtime affordance at the current time.

4 Directed Graph Encoding

A directed graph encoding has been independently proposed twice, once by Tlon in the original Hoon codebase as a `+$silo` encoding and once by Sam Parker as the `++bulk` encoding.

5 Aligned Serialization

One of the advantages of `++jam` is its compactness. However, this comes at the cost of speed, since bit-level operations are required to `++cue` the noun back from its serialized form. If a slightly larger size is acceptable, a byte-aligned serialization could facilitate certain kinds of external inspection without requiring deserialization. (For instance, a byte-aligned head tag could be read for a rapid decision without needing to `++cue` the entire noun.)

We propose a strategy to modify `++jam` to align to bytes by padding the length of entries to the nearest byte boundary and marking the distance with a clever binary scheme rather than simply unary. This approach, called `++honey`, aims to balance compactness and speed for certain use cases while retain-

ing a large degree of conceptual backwards compatibility. (The change in byte alignment of course breaks strict compatibility.)

There are two fundamental issues for byte alignment: atoms and lengths. Atoms can be padded with leading zeros to the nearest byte boundary without changing their value. Lengths, however, require a new encoding scheme to compensate for the adjustment in expected bit widths.

6 Benchmarks

7 Conclusion

Appendix A: Python `++jam`/`++cue`

The following is a simple Python implementation of `++jam` serialization drawn from Urbit's auxiliary `pynoun` library.

```
from bitstring import BitArray
noun = int | Cell
# The Cell class represents an ordered pair of two nouns.

def jam_to_stream(n: noun, out: BitArray):
    """jam but put the bits into a stream

    >>> s = BitArray()
    >>> jam_to_stream(Cell(0,0), s)
    >>> s
    BitArray('0b100101')
    """

    cur = 0
    refs = {}

    def bit(b: bool):
        nonlocal cur
        out.append([b])
        cur += 1

    def zero():
        bit(False)
```

```
    def one():
        bit(True)

    def bits(num: int, count: int):
        nonlocal cur
        for i in range(0, count):
            out.append([(num & (1 << i)) != 0])
        cur += count

    def save(a: noun):
        refs[a] = cur

    def mat(i: int):
        if 0 == i:
            one()
        else:
            a = i.bit_length()
            b = a.bit_length()
            above = b + 1
            below = b - 1
            bits(1 << b, above)
            bits(a & ((1 << below) - 1), below)
            bits(i, a)

    def back(ref: int):
        one()
        one()
        mat(ref)

    def r(a: noun):
        dupe = refs.get(a)
        if deep(a):
            if dupe:
                back(dupe)
            else:
                save(a)
                one()
                zero()
                r(a.head)
                r(a.tail)
        elif dupe:
```

```
            isize = a.bit_length()
            dsize = dupe.bit_length()
            if isize < dsize:
                zero()
                mat(a)
            else:
                back(dupe)
        else:
            save(a)
            zero()
            mat(a)
    r(n)

def jam(n: noun):
    """urbit serialization: * -> @

    >>> jam(0)
    2
    >>> jam(Cell(0,0))
    41
    >>> jam(Cell(Cell(1234567890987654321, \\
    ...               1234567890987654321), \\
    ...          Cell(1234567890987654321, \\
    ...               1234567890987654321)))
    228400950958068928742573895 73
    """

    out = BitArray()
    jam_to_stream(n, out)
    return read_int(len(out), out)

def cue_from_stream(s: BitArray):
    """cue but read the bits from a stream

    >>> s = BitArray('0b01')
    >>> cue_from_stream(s)
    0
    """

    refs = {}
    cur = 0
    position = 0
```

```
        def bits(n: int):
110         nonlocal cur, position
            cur += n
            result = 0
            for i in range(n):
                result |= (1 if s[position] else 0) << i
115             position += 1
            return result

        def one():
            nonlocal cur, position
120         cur += 1
            bit = s[position]
            position += 1
            return bit

125     def rub():
            z = 0
            while not one():
                z += 1
            if 0 == z:
130             return 0
            below = z - 1
            lbits = bits(below)
            bex = 1 << below
            return bits(bex ^ lbits)
135
        def r(start: int):
            ret = None
            if one():
                if one():
140                 ret = refs[rub()]
                else:
                    hed = r(cur)
                    tal = r(cur)
                    ret = Cell(hed, tal)
145         else:
                ret = rub()
            refs[start] = ret
            return ret
        return r(cur)
```

```
150
    def cue(i: int):
        """urbit deserialization: @ -> *

        >>> str(cue(228400950958068928742573895 73))
        '[[1234567890987654321 1234567890987654321]
155       1234567890987654321 1234567890987654321]'
        """

        bits = BitArray()
        while i > 0:
160         bits.append([i & 1 == 1])
            i >>= 1
        return cue_from_stream(bits)
```