Hot Nock: A History of the Nock Combinator Calculus

N. E. Davis `~lagrev-nocfep`
Zorp Corp

Abstract

Contents

1 Introduction

Nock is a combinator calculus which serves as the computational specification layer for the Urbit and Nockchain/NockApp systems. It is a hyper-RISC instruction set architecture (ISA) intended for execution by a virtual machine (but see **Mopfel2025** (**Mopfel2025**), pp. XX–XX herein). Nock's simplicity and unity of expression make it amenable to proof-based reasoning and

Manuscript submitted for review.

Address author correspondence to `~lagrev-nocfep`.

guarantees of correctness. Its Lisp-like nature surfaces the ability to introspect on the code itself, a property which higher-level languages compiling to it can exploit. Nock permits itself a finite number of specification changes, called "decrements" or "kelvins", which allow it to converge on a balance of expressiveness and efficacy. This article outlines the extant historical versions of the Nock combinator calculus and reconstructs the motivation for the changes made at each kelvin decrement. It begins with an exposition of Nock as a tool of computation, outlines the history of Nock's decrements, and speculates on motivations for possible future developments.

2 Nock as a Combinator Calculus

Fundamental computer science research has identified a family of universal computers which may be instantiated in a variety of ways, such as the Turing machine, the lambda calculus, and the combinator calculus. Equivalence theorems such as the Church–Turing thesis show that these systems are equivalent in their computational power, and that they can be used to compute any computable function. The combinator calculus is a family of systems which use a small set of combinators to express computation. The most well-known member of this family is the SKI combinator calculus, which uses only three combinators: S, K, and I. Other members of this family include the BCKW combinator calculus and the H combinator calculus. These systems are all equivalent in their computational power, but they differ in their syntax and semantics. The Nock combinator calculus is an extension of the SKI combinator calculus which adds a few axiomatic rules to navigate and manipulate binary trees, carry out a very primitive arithmetic, and provide for side effects.

Perhaps better put, Nock is a family of combinator calculi that sequentially converge on an "optimal" expressiveness for certain design desiderata. This includes an economy of expression (thus several macro opcodes) and consideration of how a higher-level language would invoke stored procedural expressions. Furthermore an opcode exists which produces and then ignores a computation, intended to signal to a runtime layer that a side effect may be desired by the caller.

Nock bears the following characteristics:

- Turing-complete. Put formally, Turing completeness (and thus the ability to evaluate anything we would call a computation) is exemplified by the -recursive functions (**WikipediaMuRecursive**, **WikipediaMuRecursive**). In practice, these amount to operations for constant, increment, variable access, program concatenation, and looping (**Raphael2012**, **Raphael2012**). Nock supports these directly through its primitive opcodes.

- Functional (as in language). Nock is a pure function of its arguments. In practice, the Urbit operating system provides a simulated global scope for userspace applications, but this virtualized environment reduces to garden-variety Nock. (See **Davis2025b** (**Davis2025b**), pp. XX–XX in this volume, for details of a Nock virtualized interpreter.)

- Subject-oriented. Nock evaluation consists of a formula as a noun to be evaluated against a subject as a noun. Taken together, these constitute the entire set of inputs to a pure function.

  > Everything about a scope, including name bindings, aliases, and docstrings, is stored in the subject's type. This allows Hoon's compilation discipline to be similarly minimal: the compiler is a function from subject type and Hoon source to product type and compiled Nock. Running this Nock against a value of the subject type produces a vase of the result. It's hard to imagine a more streamlined formalism for compilation. (**Blackman2020**, **Blackman2020**)

  Some Nock opcodes alter the subject, such as a variable declaration, by producing a new subject which is utilized for subsequent axis lookups.

- Homoiconic. Nock unifies code and data under a single representation. A Nock atom is a natural number, and a Nock cell is a pair of nouns. Every Nock noun is acyclic, and every Nock expression is a binary tree. For example, Nock expressions intended to be evaluated as code are often pinned as data by the constant opcode until they are retrieved by evaluating the constant opcode at that axis.

- Untyped. Nock is untyped, meaning that it does not impose any type system on the expressions it evaluates. Nock "knows" about the natural numbers in two senses: such are used for addressing axes in the binary tree of a noun, and such are manipulated and compared using the increment and equality opcodes.

- Solid-state. A Nock interpreter is a solid-state machine, meaning that it operates from a state to a new state strictly according to inputs as a pure lifecycle function. The Nock interpreter must commit the results of a successful computation as the new state before subsequent computations, or events, can be evaluated. Transient evaluations (uncompleted events) and crashes (invalid evaluations) may be lost without consequence, and the Nock interpreter layer persists the underlying state of the machine.

We have asserted without demonstration that Nock is a combinator calculus. We now show that this is the case, with reference to Nock 4K, the latest specification. The simplest combinator calculus consists of only three combinators (**SomeoneSmart**, **SomeoneSmart**), S, K, and I. These combinators are defined as follows:

1. S substitution. This corresponds to Nock 4K's opcode 2, which substitutes the second argument into the first argument at the third argument's axis. (There are some subtle differences to Nock's expression of S as opcode 2 that we will elide as being fundamentally similar, but perhaps worthy of its own monograph.)

2. K constant. This corresponds to Nock 4K's opcode 1, which yields its argument as a constant noun.

3. I identity. This corresponds to Nock 4K's opcode 0, a generalized axis lookup operator, which can trivially retrieve the current subject or expression as well as any children.

While Nock introduces a few more primitive operations as a practicality, the above identities establish its bona fides as a combinator calculus capable of general computation. Similar to Haskell Curry's BCKW system, which can be written in forms reducible to SKI, Nock provides a set of primitive rules and a set of economic extended rules for convenience in writing a compiler. (See **Galebach2025** (**Galebach2025**), pp. XX–XX in this volume, for exposition on how to evaluate a Nock expression by hand or by interpreter.)

Event Loop

The beating heart of an operational system using Nock is the event loop formula. For each event that is injected, the system runs this formula in the kernel's '+aeon' arm:

"' .*([arvo -.epic] [9 2 10 [6 0 3] 0 2]) "'

Let's talk through what is going on here.

- The subject consists of a pair of 'arvo', the evolution from underlying operating system and standard library until the current epitome state, and the head of 'epic', where 'epic' is the remaining event log to be evaluated. - The formula is a 9 which runs the head of the event log, '[0 3]', against the whole subject, then replaces that into the head of the subject. - The next event is then evaluated, until the list is empty. - When it reaches the end of the event log, the runtime holds it in stasis until the next event is prepared to be injected.

Jets

Jet dispatch does incur some overhead, which is generally worthwhile

Tree-Walking Interpreter

Nock is interpreted in the sense that it is produced to be executed by a virtual machine layer. A human end-user language will be parsed and interpreted into an abstract syntax tree, then converted into raw Nock for evaluation. This Nock is itself interpreted at runtime by a tree-walking interpreter.

This interpreter takes a Nock formula as input, recursively traverses the formula tree, evaluates each operation according to its subject, and returns the resulting noun.

The upside of Nock is that this process is fully general and portable. And since Nock is a both an ISA and a specification, when jetted code is available it can be evaluated in a faster way as much as possible.

So, some of you are going to ask me, what's the alternative to a tree-walking interpreter? To answer that, let's think about the Nock execution pattern. The major downside of Nock as an ISA is that it is a hyper-RISC that does not closely map to von Neumann architecture CPU operations. Modern computer architectures prefer to grab arrays in a hierarchy of caches, but a binary tree implementation does not lend itself well to this arrangement. Furthermore, because Nock is homoiconic, it is not straightforward to tell from a tree-walking interpreter if a given noun is code or data. A newer paradigm, called subject knowledge analysis, proposes to fix this by a system of JIT-like affordances produced by a static analysis at compile time. SKA enables efficient analysis and execution of Nock code by unearthing the computational structure of the code even in an untyped procedure-free environment.

Nock Bytecode

There are currently two practical Nock interpreters in production, Urbit's Vere and Zorp's Sword. They both currently use a tree-walking interpreter, but a more sophisticated protocol is in active development. We will take a brief look at Vere's bytecode implementation.

Vere walks the Nock tree to produce a linear bytecode array for rapid evaluation. The bytecode interpreter is a stack machine built on top of an allocator. It uses a threaded code pattern or a computed 'GOTO'. While it is highly optimized on its own terms, the gains are often clawed back by other parts of the system.

Vere can reveal the bytecode to us if we query it with an '

For instance

We can see these by passing a Nock 11 '

"‘ [lils 65535] halt > > 65.535 [libl i:o] halt > > 65.536 "‘

As with other bytecodes, these can be rather opaque. (In this case, we are sometimes left with labels and pointers to data in other locations rather than the complete data set.) Vere’s bytecode interpreter is, I have been told, rather simple and primitive.

"‘hoon > != +(41) [4 1 41]

> > [lilb 41] bump halt

> != (dec 43) [8 [9 2.398 0 1.023] 9 2 10 [6 7 [0 3] 1 43] 0 2]

> > [fask 1023] [kicb 1] snol head swap tail [lilb 43] musm [ticb 0] halt "‘

3 Nock's Decrements

The earliest extant Nock is `U`, a proto-Nock posted to the *Lambda the Ultimate* blog in 2006 (**Yarvin2006**, **Yarvin2006**).[1] The draft is versioned 0.15, but it is unclear whether this relates to kelvin versioning or not. The document is produced in full in Listing 1.

Yarvin had likely been influenced by X. (He later cited Y.)

Listing 1: U: Definition. The earliest extant Nock.

```
U: Definition

1 Purpose

    This document defines the U function and its data
    model.

2 License

    U is in the public domain.

3 Status
```

[1]Curtis Yarvin was consulted for elements of this history. Unfortunately many elements of the original prehistory of Nock appear to be lost to the sands of time on unrecoverable hard drives.

This text is a DRAFT (version 0.15).

4 Data

A value in U is called a "term." There are three kinds of term: "number," "pair," and "foo."

A number is any natural number (ie, nonnegative integer).

A pair is an ordered pair of any two terms.

There is only one foo.

5 Syntax

U is a computational model, not a programming language.

But a trivial ASCII syntax for terms is useful.

5.1 Trivial syntax: briefly

Numbers are in decimal. Pairs are in parentheses that nest to the right. Foo is "~".

Whitespace is space or newline. Line comments use "#".

5.2 Trivial syntax: exactly

```
term    : number
        | 40 ?white pair ?white 41
        | foo

number  : 48
        | [49-57] *[48-57]

pair    : term white term
        | term white pair

foo     : 126
```

```
    white   : *(32 | 10 | (35 *[32-126] 10))

6 Semantics

    U is a pure function from term to term.

    This document completely defines U.  There is no
    compatible way to extend or revise U.

6.1 Rules

    [name]   [pattern]                [definition]

    (a)      ($a 0 $b)                $b
    (b)      ($a 1 $b $c)             1
    (c)      ($a 1 $b)                0
    (d)      ($a 2 0 $b $c)           $b
    (e)      ($a 2 %n $b $c)          $c
    (f)      ($a 3 $b $c)             =($b $c)
    (g)      ($a 4 %n)                +%n

    (h)      ($a 5 (~ ~ $b) $c)       $b
    (i)      ($a 5 (~ $b $c) $d)      *($a $b $c $d)
    (j)      ($a 5 (~ ~) $b)          ~
    (k)      ($a 5 (~ $b) $c)         *($a $b $c)
    (l)      ($a 5 ($b $c) $d)
                            (*($a $b $d) *($a $c $d))
    (m)      ($a 5 $b $c)             $b

    (n)      ($a 6 $b $c)   *($a *($a 5 $b $c))
    (o)      ($a 7 $b)                *($a 5 $a $a $b)
    (p)      ($a 8 $b $c $d)          >($b $c $d)

    (q)      ($a $b $c)     *($a 5 *($a 7 $b) $c)
    (r)      ($a $b)                  *($a $b)
    (s)      $a                       *$a

    The rule notation is a pseudocode, only used in
    this file. Its definition follows.

6.2 Rule pseudocode: briefly
```

Each line is a pattern match. "%" means "number." Match in order. See operators below.

6.3 Rule pseudocode: exactly

Both pattern and definition use the same evaluation language, an extension of the trivial syntax.

An evaluation is a tree in which each node is a term, a term-valued variable, or a unary operation.

Variables are symbols marked with a constraint. A variable "$name" matches any term. "%name" matches any number.

There are four unary prefix operators, each of which is a pure function from term to term: "=", "+", "*", and ">". Their semantics follow.

6.4 Evaluation semantics

For any term $term, to compute U($term):

- find the first pattern, in order, that matches $term.

- substitute its variable matches into its definition.

- compute the substituted definition.

Iff this sequence of steps terminates, U($term) "completes." Otherwise it "chokes."

Evaluation is strict: incorrect completion is a bug. Choking is U's only error or exception mechanism.

6.5 Simple operators: equal, increment, evaluate

=($a $b) is 0 if $a and $b are equal; 1 if they are not.

+%n is %n plus 1.

*$a is U($a).

6.6 The follow operator

>($a $b $c) is always 0. But it does not always complete.

We say "$c follows $b in $a" iff, for every $term:

```
    if *($a 5 $b $term) chokes:
        *($a 5 $c $term) chokes.

    if *($a 5 $b $term) completes:
        either:
            *($a 5 $c $term) completes, and
            *($a 5 $c $term) equals *($a 5 $b
$term)
        or:
            *($a 5 $c $term) chokes.
```

If $c follows $b in $a, >($a $b $c) is 0.

If this statement cannot be shown (ie, if there exists any $term that falsifies it, generates an infinitely recursive series of follow tests, or is inversely self-dependent, ie, exhibits Russell's paradox), >($a $b $c) chokes.

7 Implementation issues

This section is not normative.

7.1 The follow operator

Of course, no algorithm can completely implement the follow operator. So no program can completely implement U.

But this does not stop us from stating the correctness of a partial implementation - for example, one that assumes a hardcoded set of follow cases, and fails when it would otherwise have to compute a follow case outside this set.

U calls this a "trust failure." One way to standardize trust failures would be to standardize a fixed set of follow cases as part of the definition of U. However, this is equivalent to standardizing a fixed trusted code base. The problems with this approach are well-known.

A better design for U implementations is to depend on a voluntary, unstandardized failure mechanism. Because all computers have bounded memory, and it is impractical to standardize a fixed memory size and allocation strategy, every real computing environment has such a mechanism.

For example, packet loss in an unreliable packet protocol, such as UDP, is a voluntary failure mechanism.

If the packet transfer function of a stateful UDP server is defined in terms of U, failure to compute means dropping a packet. If the server has no other I/O, its semantics are completely defined by its initial state and packet function.

7.2 Other unstandardized implementation details

A practical implementation of U will detect and log common cases of choking. It will also need a timeout or some other unspecified mechanism to abort undetected infinite loops.

(Although trust failure, allocation failure or timeout, and choke detection all depend on what is presumably a single voluntary failure mechanism, they are orthogonal and should not be confused.)

Also, because U is so abstract, differences in implementation strategy can result in performance disparities which are almost arbitrarily extreme. The difficulty of standardizing performance is well-known.

No magic bullet can stop these unstandardized issues from becoming practical causes of lock-in and incompatibility. Systems which depend on U must manage them at every layer.

4 The Future of Nock

SKEW PLAN

The most likely motivation for a future decrement seems to be the likelihood that some current operation is underspecified, and an implicit choice has been made by every contemporary implementation but should be surfaced.

5 Conclusion

Nock 1K should be the final form of the Nock ISA, leaving one decrement for Gnon.