# Metacircular Virtualization & Practical Nock Interpretation

N. E. Davis ~lagrev-nocfep
Zorp Corp

**Abstract**

## Contents

Manuscript submitted for review.

Address author correspondence to ~lagrev-nocfep.

## 1 Introduction

The Nock combinator calculus is used as a target instruction set architecture (ISA) for the Hoon and Jock languages. However, Nock itself is a "crash-only" paradigm which has no error handling mechanism within the terms of the ISA itself. Furthermore, sometimes information is necessary to complete a calculation which may not be present in the apparent subject. Practical Nock interpreters handle this by running Nock within Nock, a process of metacircular virtualization. This article explores the design of `++mock`, the basic virtualized Nock interpreter, and its role within a Nock-based kernel and standard library.

## 2 Background

In 1958, John McCarthy began his lifelong work on the Lisp programming language (**McCarthy1996**, **McCarthy1996**). While the form would evolve over the years, the fundamental components included S-expressions, or parenthesized lists of homoiconic code–data; `'` or quote, a deferred expression; and `eval`, a function to evaluate quoted code.[1] Together these components allowed McCarthy and colleague Steve Russell to implement a Lisp interpreter in Lisp itself, the first metacircular interpreter (**McCarthy1978**, **McCarthy1978**). Such was a radical departure from the imperative programming paradigm of the time, exemplified in Fortran (1954–55); Lisp's reliance on recursion and higher-order functions was both a precursor

[1]Paul Graham, in his 2001 short essay "What Made Lisp Different", grouped these concepts into the phrase "The whole language always available" (**Graham2001**, **Graham2001**).

to the functional program paradigm and a rejuvenation of the Church and Curry schools of fundamental computing logic.

Metacircularity was considered sufficiently important that several programming languages implemented it as a first-class feature, including Forth, TeX, and Prolog. They have generally been used either for tightly interpreted features like debuggers and code introspection tools, or for language extensions like macros.

Amin and Rompf (**Amin2017**) analyzed how metacircular interpreters build on top of each other, each adding a new layer of abstraction. While primarily examining a Lisp dialect, they showed practically how interpreted language extensions (even in cascading towers) can be collapsed to bottom-level interpretation, similar to Nock's own `++mink` function. They concisely epitomized a line of research on reflective languages and self-interpreted languages, such as quasiquotation in Lisp (**Bawden1999**, **Bawden1999**) and Template Meta-Haskell (**Sheard2002**, **Sheard2002**). The ultimate point of this enquiry was to show that metacircularity was a powerful and well-grounded tool for building interpreters, and that it could be used to in fact build practical interpreters.

Thus, before Nock was even named, one of its earliest design criteria was the ability to straightforwardly implement a metacircular interpreter (`~sorreg-namtyv`; `~sorreg-namtyv`, 2006; 2010). While the original motivation was to implement a hyper-Turing operator over a referentially transparent bound namespace, Nock's support for virtualization also soon proved useful for managing error traces and crashes. Yarvin wrote in an early document:

> Metacircularity in most languages is a curiosity and corner case. Programmers are routinely warned not to use it, and for good reason. Practical metacircularity is a particular strength of Nock, or any functional assembly language. Dynamic loading and/or virtualization works, at least logically, as in an OS.
>
> We hypothesize that a metacircular functional dissemination protocol can serve as a complete, general-

> purpose system software environment—there is no problem it cannot solve, in theory or in practice. Such a protocol can be described as an "operating function" or *OF*—the functional analog of an imperative operating system. (`~sorreg-namtyv`, 2010)

Metacircularity thus closely ties to the idea of a functional operating system. This is a theme that has been explored in other contexts, such as the Lisp machines and the Haskell GHC runtime system. For Nock, a virtualized interpreter is no mere corner case but a first-class component of the system, both for practical execution and for resource discovery.

## 3 Virtualized Nock

Given an argument for

Some essential components provided by vanilla Hoon for virtualizing Nock include:

- `++mink`, compute bottom-level virtual Nock.
- `++mock`, compute formula on subject with hint. Wraps `++mink`; common entrypoint.
- `++mule`, kick a trap (deferred computation). Wraps `++mock`; common entrypoint.
- `++soft`, wrap a typecast as a unit (i.e., fail with ~ rather than crash).

Nock execution environments virtualize Nock for two primary reasons:

1. Crash handling (provisional execution).
2. Language extension (additional opcodes).

## 3.1 Crash handling

While virtualization necessarily incurs some overhead relative to more direct execution, it gives the Nock programmer a way to bail (crash), trace (log with debugging information), and bind (time-limit) a computation. In fact, stack traces are computed through virtualization, and so are guaranteed to be as correct as their implementation.

A Nock virtualization environment needs to distinguish at least three kinds of results:

1. A successful result of computation, along with a valid Nock noun.
2. A block, which indicates that the computation is not yet complete and should be continued.
3. A halt or deterministic error, which indicates that the computation has failed and should not be continued.

These correspond to standard scry results (see "Opcode 12" below) in this order:

1. `[~ ~ noun]`, a successful result.
2. `~`, a result is not currently available (block).
3. `[~ ~]`, a result will never become available (halt).

Because Nock is untyped and we need to distinguish a "zero" from a "block", the structure of each scry result fundamentally differs from the other options.

Nondeterministic errors—crashes in the runtime, such as an out-of-memory error—are not handled by or visible to the Nock interpreter as a solid-state computer.

## 3.2 Language extension

Virtualization enables the extension of Nock using "fake opcodes". At the current time, only opcode 12 has been added;[2]

[2]Compare, however, the approach of **Atman2025** (**Atman2025**), pp. XX–XX in this volume, who presented an explicitly constructive Nock-like paradigm in Ax.

this opcode provides a generalized way to supply the OS context to the scope in a way that appears like a `GOTO` or remote call but actually preserves Nock semantics including strict subject scoping. There is a sense in which the scry operation and the bound (referentially transparent) scry namespace are identical to the metacircularity of the Nock interpreter itself. The operating function itself can be treated as a first-class member of its own ecosystem.

Hoon defines `+$nock` already suited for virtualized Nock execution including fake opcode 12 (Listing 1).

However, while `++mink` is compiled to and evaluated in Nock, it is written in Hoon and accordingly uses Hoon's affordances, particularly vases. Vases are a pair of type and data intended to store relevant type information for correctly evaluating raw untyped Nock nouns.[3] Accordingly, it is Hoon's higher-level introspective tools that facilitate `++mink`'s operation.

The remainder of this section analyzes how each `++mink` implements each Nock opcode, including particularly the fake opcode 12. The following code samples are taken from `/sys/hoon` in the Urbit codebase.

`++mink` produces a head atom marking the kind of result the interpreter has raised: if a `%0`, the result is a valid Nock noun; if a `%1`, a block is indicated (i.e. the halting problem continues); if a `%2`, then a deterministic error has occurred with an error trace.

## 3.3 Opcode 0

Opcode 0 is the identity function, which returns its argument unchanged. This is implemented in `++mink` as follows:

```
::  *[a 0 b]        /[b a]
  [%0 axis=@]
=/  part  (frag axis.formula subject)
?~  part  [%2 trace]
[%0 u.part]
```

[3]See **Davis2025** (**Davis2025**), pp. XX–XX in this volume, for a discussion of vases and their role in practical Nock compilation.

Listing 1: Hoon's +$nock specification

```
::              ::::::  virtual nock
+$  nock  $^    ::  autocons
                [p=nock q=nock]
          $%    ::  constant
                [%1 p=*]
                ::  compose
                [%2 p=nock q=nock]
                ::  cell test
                [%3 p=nock]
                ::  increment
                [%4 p=nock]
                ::  equality test
                [%5 p=nock q=nock]
                ::  if, then, else
                [%6 p=nock q=nock r=nock]
                ::  serial compose
                [%7 p=nock q=nock]
                ::  push onto subject
                [%8 p=nock q=nock]
                ::  select arm and fire
                [%9 p=@ q=nock]
                ::  edit
                [%10 p=[p=@ q=nock] q=nock]
                ::  hint
                [%11 p=$@(@ [p=@ q=nock]) q=nock]
                ::  grab data from sky
                [%12 p=nock q=nock]
                ::  axis select
                [%0 p=@]
          ==
```

In essence, an axis is provisionally extracted from the subject; if this is ~ then an error is raised with the current execution trace, else the resulting value from that axis is produced.

### 3.4 Opcode 1

Opcode 1 is the constant function, which returns a constant value extricated from the formula. This is implemented in `++mink` as follows:

```
::  *[a 1 b]      b
  [%1 constant=*]
[%0 constant.formula]
```

### 3.5 Opcode 2

Opcode 2 is the evaluation function, which evaluates the formula on the subject. This is implemented in `++mink` as follows:

```
::  *[a 2 b c]    *[*[a b] *[a c]]
  [%2 subject=* formula=*]
=/  subject  $(formula subject.formula)
?.  ?=(%0 -.subject)  subject
=/  formula  $(formula formula.formula)
?.  ?=(%0 -.formula)  formula
%=  $
  subject  product.subject
  formula  product.formula
==
```

Errors are passed directly through, whereas the resulting formula is invoked on the resulting subject.

### 3.6 Opcode 3

Opcode 3 is the cell test function, which tests whether the subject is a cell. This is implemented in `++mink` as follows:

```
::  *[a 3 b]      ?*[a b]
  [%3 subject=*]
=/  argument  $(formula argument.formula)
```

```
?.  ?=(%0 -.argument)  argument
[%0 .?(product.argument)]
```

This reduces to an evaluation of a base-level opcode 3 after a preliminary check that the formula evaluates correctly.

## 3.7 Opcode 4

Opcode 4 is the increment function, which increments the subject. This is implemented in `++mink` as follows:

```
::  *[a 4 b]      +*[a b]
  [%4 subject=*]
=/  argument  $(formula argument.formula)
?.  ?=(%0 -.argument)  argument
?^  product.argument  [%2 trace]
[%0 .+(product.argument)]
```

Note that like opcode 3, this reduces to an evaluation of a base-level opcode 4.

## 3.8 Opcode 5

Opcode 5 is the equality test function, which tests whether the subject and formula are equal. This is implemented in `++mink` as follows:

```
::  *[a 5 b c]    =[*[a b] *[a c]]
  [%5 subject=* formula=*]
=/  a  $(formula a.formula)
?.  ?=(%0 -.a)  a
=/  b  $(formula b.formula)
?.  ?=(%0 -.b)  b
[%0 =(product.a product.b)]
```

Like opcodes 3 and 4, this reduces to an evaluation of a base-level opcode 5 after checking the left-hand and right-hand formulas.

## 3.9 Opcode 6

Opcode 6 is the if-then-else or conditional branch function, which evaluates the formula on the subject and then evaluates either the first or second formula depending on the result. This is implemented in `++mink` as follows:

```
::  *[a 6 b c d]  *[a *[[c d] 0 *[[2 3] 0 *[a 4 4 b]]]]
  [%6 test=* yes=* no=*]
=/  result  $(formula test.formula)
?.  ?=(%0 -.result)  result
?+  product.result
      [%2 trace]
  %&  $(formula yes.formula)
  %|  $(formula no.formula)
==
```

As with the direct opcode 6, there is no short-circuit evaluation in the conditional statement.

## 3.10 Opcode 7

Opcode 7 is the composition function, which evaluates the formula on the subject and then evaluates the next formula on the result. This is implemented in `++mink` as follows:

```
::  *[a 7 b c]            *[*[a b] c]
  [%7 subject=* next=*]
=/  subject  $(formula subject.formula)
?.  ?=(%0 -.subject)  subject
%=  $
  subject  product.subject
  formula  next.formula
==
```

This implementation simply drops the evaluation of `subject` through as the explicit subject for the next formula.

## 3.11 Opcode 8

Opcode 8 is the variable push function, which pushes a value onto the subject. This is implemented in `++mink` as follows:

```
::  *[a 8 b c]            *[[*[a b] a] c]
  [%8 head=* next=*]
=/  head  $(formula head.formula)
```

```
?.  ?=(%0 -.head)  head
%=  $
  subject  [product.head subject]
  formula  next.formula
==
```

Here the conceptual differences between opcode 7 and opcode 8 can be sharply compared: how 7 requires an evaluation against the current subject, leading to something of the nature of $c(b(a))$ in terms of function composition, versus how 8 simply augments the subject.

## 3.12 Opcode 9

Opcode 9 is the arm select function, which selects an arm of the subject and evaluates it. This is implemented in `++mink` as follows:

```
:: *[a 9 b c]    *[*[a c] 2 [0 1] 0 b]
  [%9 axis=@ core=*]
=/  core  $(formula core.formula)
?.  ?=(%0 -.core)  core
=/  arm  (frag axis.formula product.core)
?~  arm  [%2 trace]
%=  $
  subject  product.core
  formula  u.arm
==
```

For all the mechanics involved in opcode 9, the implementation is perhaps surprisingly simple. Extract the arm at the axis and evaluate it structurally using an opcode 2. (As an aside, Hoon which compiles to Nock aggressively utilizes lambdas to define functions at the point of use, and frequently employs a gate-building gate pattern. This latter resolves to two opcode 9s in a series.)

## 3.13 Opcode 10

Opcode 10 is the edit function, which edits a value in the subject at the given axis. This is implemented in `++mink` as follows:

```
::  *[a 10 [b c] d]  #[b *[a c] *[a d]]
  [%10 [axis=@ value=*] target=*]
?:  =(0 axis.formula)  [%2 trace]
=/  target  $(formula target.formula)
?.  ?=(%0 -.target)  target
=/  value  $(formula value.formula)
?.  ?=(%0 -.value)  value
=/  mutant=(unit *)
  (edit axis.formula product.target product.value)
?~  mutant  [%2 trace]
[%0 u.mutant]
```

`++edit` traverses the noun to attempt to access a given axis; if it succeeds, it returns a new noun with the value at that axis replaced by the given value. If it fails, it returns an empty `unit`, indicating without a crash that the edit failed.

There's more plumbing here because of the mutation of the noun, but the essential concept of locating and replacing a valid subtree is straightforward.

## 3.14 Opcode 11

Opcode 11 is the hint function, which provides a hint to the interpreter about how to interpret the argument. This is the only way to produce side effects from Nock as a pure function, and `++mink` respects and passes through hints raised by the virtualized Nock formula. This is implemented in `++mink` as follows:

```
::  *[a 11 b c]      *[a c]
::  *[a 11 [b c] d]  *[[*[a c] *[a d]] 0 3]
  [%11 tag=@ next=*]
=/  next  $(formula next.formula)
?.  ?=(%0 -.next)  next
:-  %0
.*  subject
[11 tag.formula 1 product.next]
::
  [%11 [tag=@ clue=*] next=*]
=/  clue  $(formula clue.formula)
?.  ?=(%0 -.clue)  clue
```

```
=/  next
  =?    trace
      ?=(?(%hunk %hand %lose %mean %spot) tag.formula)
    [[tag.formula product.clue] trace]
  $(formula next.formula)
?.  ?=(%0 -.next)  next
:-  %0
.*  subject
[11 [tag.formula 1 product.clue] 1 product.next]
```

Two options match for opcode 11, a static and a dynamic hint. Even though this virtualized code "sandboxes" execution, the Nock formula including the `clue` is in fact evaluated. The runtime listens for a `clue`, which is compared to a whitelisted set of supported hints, as in normal (nonvirtualized) Nock execution.

Note that the argument cannot simply be discarded by the interpreter, because it may result in a crash. Thus, as with direct Nock execution, the hint must be evaluated.[4]

## 3.15 Opcode 12

Opcode 12 is the scry function, which retrieves a value from the `sky` namespace environment. This is implemented in `++mink` as follows:

```
  [%12 ref=* path=*]
=/  ref  $(formula ref.formula)
?.  ?=(%0 -.ref)  ref
=/  path  $(formula path.formula)
?.  ?=(%0 -.path)  path
=/  result  (scry product.ref product.path)
?~  result
  [%1 product.path]
```

[4]Here, this should be read "evaluated safely", with the caveat that some hints may result in a runtime crash. For instance, an ill-formed `%slog` hint (`printf`) will crash the interpreter if the `clue` is not structured correctly. This is a consequence of the fact that Nock is untyped and so the interpreter cannot know whether the hint is well-formed or not. The `++mink` interpreter does not attempt to validate hints, but merely passes them through to the runtime for evaluation. (One could imagine a Nock virtualizer which itself handled certain hints as if they were "side effects" to the evaluation.)

```
?~  u.result
  [%2 [%hunk product.ref product.path] trace]
[%0 u.u.result]
```

Opcode 12 does not have a direct Nock expression since it "steps outside" the subject–formula pair via the expedient of the `scry` gate.

> In order to interpret opcode 12, `++mink` takes as an argument, in addition to the subject and formula, a gate which will be invoked with the results of the subformulas of 12 as a sample. The result of executing this gate (not as interpreted by `++mink` but by whichever interpreter is running `++mink`) are then treated as the result of the Nock 12 operation. ("4n `++mink`", Urbit documentation)

Much of consequence is entailed by the use of a `scry` gate, which is a Hoon function that retrieves a value from an associated `sky` namespace (nominally, the bound scry namespace, although this is not necessarily entailed by Urbit in contemporary practice). The `scry` function bears the signature `$-(^ (unit (unit)))`. The result of this operation is either a noun or a block, and it is returned as a `(unit (unit noun))`. These correspond to standard scry results in this order:

1. `[~ ~ noun]`, a successful result.
2. `~`, a result is not currently available (block).
3. `[~ ~]`, a result will never become available (halt).

For instance, the reference `roof` function for Arvo, the Urbit kernel, is schematically defined as:

```
++  roof
  |$  [vase]
  $-  $:  lyc=(unit (set ship))       ::  leakset
          pov=path                    ::  provenance
          [vis=view bem=beam]         ::  perspective
      ==                              ::
  %-  unit                            ::  ~: unknown
  %-  unit                             ::  ~ ~: invalid
  (pair mark vase)                     ::  envased result
+$  view  $@(term [way=term car=term]) ::  perspective
```

At this point, an excursus into the Arvo core is warranted. The Arvo core is the Nock-based kernel of Urbit, and it is responsible for managing the state of the system and providing a consistent interface for agents to interact with that state. The Arvo core is built on top of the Nock interpreter, and it uses the `++mink` function to interpret Nock formulas. Each vane in the Arvo core receives a `+$roof` in its function call, which it can call when it needs to scry with opcode 12. This gate provides access to the vane's state, and it is used to retrieve values from the vane's state in a way that is consistent with the Nock semantics.

`++look` is a gate that is used to convert the `++mink`-compatible version of the scry handler into a version that can be used by the Arvo core. This gate acts as a bridge between the `.^` dotket and the scry handler (`++look`) with access to the roof that came from the vane. This pattern imposes constraints on interpreters, but provides a way to mediate access to the state of the system in a way that is consistent with Nock semantics. Thus `.^` dotket gives mediated access into the `sky` without exposing the state of the entire system in the subject (untyped permissions-free access).

## 3.16 Et cetera

`++mink` also features some machinery to correctly implement the cell distribution rule.

`++mink` is naturally jetted for efficiency, ultimately by `u3m_soft_m()` in Urbit's Vere (C) and by `interpret()` in NockApp's Sword (Rust), both the virtualization context for their respective runtimes. Some internal affordances for the runtime are in place to keep execution consistent; e.g., scry gates are stored on a stack to prevent re-entry if it scries.

TODO

Finally, it is instructive to consider what the minimal required amount of code machinery would be for one to successfully implement a Nock metacircular interpreter in Nock itself. An examination of `++mink` shows modest reliance on Hoon's higher-level features; tree math, for instance, in `++cap` for distinguishing head vs. tail and `++mas` for finding the relative address of an axis. (None of these are particularly complicated, but it seems much more straightforward to start from a compiled Hoon implementation and cut that down.)

TODO

## 4 Conclusion

Metacircular virtualization is a powerful tool for building interpreters, and Nock is no exception. The design of `++mock` and its underlying `++mink` function provides a practical way to interpret Nock in a way that is both efficient and extensible. One could imagine, for instance, a Nock interpreter cousin to `++mock` that is instrumented for partial evaluation of formulas (as in a debugger or syntax highlighter). Nock supports a rich capacity to build metacircular interpreters as a first-class feature, a potential which remains as yet relatively unexplored in the Nock ecosystem.

## References

`~sorreg-namtyv`, Curtis Yarvin (2006) "U, a small model". URL: `http://urbit.sourceforge.net/u.txt` (visited on ~2024.2.20).

— (2010a) "Urbit: functional programming from scratch". URL: `http://moronlab.blogspot.com/2010/01/urbit-functional-programming-from.html` (visited on ~2024.1.25).

— (2010b) "Watt: A Self-Sustaining Functional Language (preprint)". URL: `https://github.com/cgyarvin/urbit/blob/master/Spec/watt/sss10.tex` (visited on ~2025.5.19).