

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24

Metacircular Virtualization & Practical Nock Interpretation

N. E. Davis ~lagrev-nocfep
Zorp Corp

Abstract

This paper presents a detailed examination of metacircular virtualization in the Nock combinator calculus, focusing on the practical implementation of a virtualized Nock interpreter within the Urbit ecosystem. We explore the design and functionality of key components, including the ++mink interpreter, and their roles in enabling robust Nock execution environments. Extensions of Nock using fake opcodes, particularly opcode 12, are discussed, and a minimal working example of a Nock-virtualized Nock formula is provided.

Contents

1	Introduction	2
2	Background	2
3	Virtualized Nock	4
3.1	Crash handling	5
3.2	Language extension	6
3.3	Opcode 0	6
3.4	Opcode 1	8
3.5	Opcode 2	8

25	3.6	Opcode 3	9
26	3.7	Opcode 4	9
27	3.8	Opcode 5	9
28	3.9	Opcode 6	10
29	3.10	Opcode 7	10
30	3.11	Opcode 8	11
31	3.12	Opcode 9	11
32	3.13	Opcode 10	12
33	3.14	Opcode 11	12
34	3.15	Opcode 12	13
35	3.16	Et cetera	15
36	4	Conclusion	26

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 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:

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

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 metacircularity, we can now examine how Nock is actually virtualized in practice. The Urbit standard library provides a metacircular Nock interpreter, `++mink`, along with several other components for building virtualized Nock environments. 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 endpoint.
- `++mule`, kick a trap (deferred computation). Wraps `++mock`; common endpoint.

- `++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 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:

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

³See **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 knock
+$  knock  $^  :: autocons
          [p=nock q=nock]
          $%  :: constant
5         [%1 p=★]
          :: compose
          [%2 p=nock q=nock]
          :: cell test
          [%3 p=nock]
10        :: increment
          [%4 p=nock]
          :: equality test
          [%5 p=nock q=nock]
          :: if, then, else
15        [%6 p=nock q=nock r=nock]
          :: serial compose
          [%7 p=nock q=nock]
          :: push onto subject
          [%8 p=nock q=nock]
20        :: select arm and fire
          [%9 p=@ q=nock]
          :: edit
          [%10 p=[p=@ q=nock] q=nock]
          :: hint
25        [%11 p=$@(@ [p=@ q=nock]) q=nock]
          :: grab data from sky
          [%12 p=nock q=nock]
          :: axis select
          [%0 p=@]
30        ==
```

```

187
188 :: *[a 0 b]      /[b a]
189   [%0 axis=0]
190 =/ part (frag axis.formula subject)
191 ?~ part [%2 trace]
192 [%0 u.part]
193

```

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

197 3.4 Opcode 1

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

```

201
202 :: *[a 1 b]      b
203   [%1 constant=*]
204 [%0 constant.formula]
205

```

206 3.5 Opcode 2

207 Opcode 2 is the evaluation function, which evaluates the for-
 208 mula on the subject. This is implemented in ++mink as follows:

```

209
210 :: *[a 2 b c]    *[*[a b] *[a c]]
211   [%2 subject=* formula=*]
212 =/ subject $(formula subject.formula)
213 ?. ?=(%0 -.subject) subject
214 =/ formula $(formula formula.formula)
215 ?. ?=(%0 -.formula) formula
216 %= $
217   subject product.subject
218   formula product.formula
219 ==
220

```

221 Errors are passed directly through, whereas the resulting for-
 222 mula is invoked on the resulting subject.

223 3.6 Opcode 3

224 Opcode 3 is the cell test function, which tests whether the sub-
 225 ject is a cell. This is implemented in ++mink as follows:

```
226 :: *[a 3 b]      ?*[a b]
227               [%3 subject=*]
228 =/ argument $(formula argument.formula)
229 ?. ?=(%0 -.argument) argument
230 [%0 .?(product.argument)]
231
```

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

235 3.7 Opcode 4

236 Opcode 4 is the increment function, which increments the sub-
 237 ject. This is implemented in ++mink as follows:

```
238 :: *[a 4 b]      **[a b]
239               [%4 subject=*]
240 =/ argument $(formula argument.formula)
241 ?. ?=(%0 -.argument) argument
242 ?^ product.argument [%2 trace]
243 [%0 .+(product.argument)]
244
```

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

248 3.8 Opcode 5

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

```
252 :: *[a 5 b c]    =[*[a b] *[a c]]
253               [%5 subject=* formula=*]
254 =/ a $(formula a.formula)
255 ?. ?=(%0 -.a) a
256 =/ b $(formula b.formula)
257 ?. ?=(%0 -.b) b
258 [%0 =(product.a product.b)]
259
```

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

264 3.9 Opcode 6

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

```

269 :: *[a 6 b c d] *[a *[[c d] 0 *[[2 3] 0 *[a 4 4 b]]]]
270   [%6 test=* yes=* no=*]
271   /= result $(formula test.formula)
272   ?. ?=(%0 -.result) result
273   ?+ product.result
274   [%2 trace]
275   %& $(formula yes.formula)
276   %| $(formula no.formula)
277   ==
278
279
```

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

282 3.10 Opcode 7

283 Opcode 7 is the composition function, which evaluates the for-
 284 mula on the subject and then evaluates the next formula on the
 285 result. This is implemented in ++mink as follows:

```

286 :: *[a 7 b c]          *[[a b] c]
287   [%7 subject=* next=*]
288   /= subject $(formula subject.formula)
289   ?. ?=(%0 -.subject) subject
290   %= $
291   subject product.subject
292   formula next.formula
293   ==
294
295
```

296 This implementation simply drops the evaluation of subject
 297 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

336 functions at the point of use, and frequently employs a gate-
 337 building gate pattern. This latter resolves to two opcode 9s in
 338 a series.)

339 3.13 Opcode 10

340 Opcode 10 is the edit function, which edits a value in the sub-
 341 ject at the given axis. This is implemented in ++mink as follows:

```

342 :: *[a 10 [b c] d] #[b *[a c] *[a d]]
343   [%10 [axis=@ value=*] target=*]
344   ?= (0 axis.formula) [%2 trace]
345   /= target $(formula target.formula)
346   ?= (%0 -.target) target
347   /= value $(formula value.formula)
348   ?= (%0 -.value) value
349   /= mutant=(unit *)
350   (edit axis.formula product.target product.value)
351   ?~ mutant [%2 trace]
352   [%0 u.mutant]
```

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

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

362 3.14 Opcode 11

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

```

369 :: *[a 11 b c]          *[a c]
370 :: *[a 11 [b c] d]      *[[*[a c] *[a d]] 0 3]
371   [%11 tag=@ next=*]
```

```

373 =/ next $(formula next.formula)
374 ?. ?=(%0 -.next) next
375 :- %0
376 .* subject
377 [11 tag.formula 1 product.next]
378 ::
379 [%11 [tag=@ clue=*] next=*)
380 =/ clue $(formula clue.formula)
381 ?. ?=(%0 -.clue) clue
382 =/ next
383 =? trace
384 ?=(?(%hunk %hand %lose %mean %spot) tag.formula)
385 [[tag.formula product.clue] trace]
386 $(formula next.formula)
387 ?. ?=(%0 -.next) next
388 :- %0
389 .* subject
390 [11 [tag.formula 1 product.clue] 1 product.next]
391
```

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

⁴Here, 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.)

```

405
406   [%12 ref=* path=*]
407   =/ ref $(formula ref.formula)
408   ?. ?=(%0 -.ref) ref
409   =/ path $(formula path.formula)
410   ?. ?=(%0 -.path) path
411   =/ result (scry product.ref product.path)
412   ?~ result
413   [%1 product.path]
414   ?~ u.result
415   [%2 [%hunk product.ref product.path] trace]
416   [%0 u.u.result]
417

```

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

421 In order to interpret opcode 12, `++mink` takes as an
422 argument, in addition to the subject and formula,
423 a gate which will be invoked with the results of
424 the subformulas of 12 as a sample. The result of
425 executing this gate (not as interpreted by `++mink`
426 but by whichever interpreter is running `++mink`)
427 are then treated as the result of the Nock 12 oper-
428 ation. (“4n `++mink`”, Urbit documentation)

429 Much of consequence is entailed by the use of a `scry` gate,
430 which is a Hoon function that retrieves a value from an asso-
431 ciated sky namespace (nominally, the bound `scry` namespace,
432 although this is not necessarily entailed by Urbit in contempo-
433 rary practice). The `scry` function bears the signature

```

434 $-(^ (unit (unit)))

```

435 The result of this operation is either a noun or a block, and it
436 is returned as a `(unit (unit noun))`. These correspond to
437 standard `scry` results in this order:

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

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

```

441
442
443
444 ++ roof
445 |$ [vase]
446 $- $: lyc=(unit (set ship))      :: leakset
447     pov=path                      :: provenance
448     [vis=view bem=beam]          :: perspective
449     ==                            ::
450 %- unit                          :: ~      unknown
451 %- unit                          :: ~ ~   invalid
452 (pair mark vase)                 :: envased result
453 +$ view $@ (term [way=term car=term]) :: perspective
454

```

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.

```

479
480   [ ^ * ]
481   =/ head $(formula -.formula)
482   ?. ?=(%0 -.head) head
483   =/ tail $(formula +.formula)
484   ?. ?=(%0 -.tail) tail
485   [%0 product.head product.tail]

```

487 There is little to be said of this machinery, but particularly notice
 488 that the error is propagated from the member of the cell
 489 which crashes.

490 ++mink is naturally jettied for efficiency, ultimately by
 491 u3m_soft_run() in Urbit’s Vere (C) and by interpret() in
 492 NockApp’s NockVM (née) Sword (Rust), both the virtualiza-
 493 tion context for their respective runtimes. Some internal af-
 494 fordances for the runtime are in place to keep execution con-
 495 sistent; e.g., scry gates are stored on a stack to prevent re-entry
 496 if it scries.

497 Nock is a crash-only language, which means that when
 498 something goes wrong, the exception is an exception for the
 499 runtime or virtual machine to handle as it will. Typically, for a
 500 runtime this means an injection of the error trace back into the
 501 Nock kernel (cf. section “u3n: nock execution” in u3.md in the
 502 Urbit documentation⁵). (For ++mink, errors are attempted to
 503 be front-run by the interpreter and thus bad Nock should not
 504 be executed as such.)

505 Harkening back to **Amin2017** (**Amin2017**)’s collapsed
 506 towers of interpreters, the affordances of ++mock as a metacir-
 507 cular interpreter permit us to build a tower of interpreters on
 508 top of it without incurring excessive performance overhead.
 509 As the author of u3.md concluded, “we [even] simply treat
 510 Nock proper as a special case of mock.”

511 What would such an interpreter look like, and how would
 512 it relate to actual operation on a runtime in practice? Let us
 513 extend ++mink to implement a logical loobean operator. (Call
 514 the modified gate ++pink.) This operation will be invoked with
 515 a new fake opcode 13. Under the hood, opcode 13 will simply

⁵This section devoted a fair amount of attention to opcode 11 now 12 (it being written for Nock 5K) and hint evaluation.

supply logically negate a value (via an opcode 6), much as how opcode 12 supplies a consistent Nock formula for the scry operation. (Similar to [0 0], we will be able to use [13 1 2] to initiate a crash if desired.)

```

520
521 :: *[a 13 b]      NOT *[a b]
522   [%13 subject=*]
523 =/  argument  $(formula argument.formula)
524 ?= (%0 -. argument)  argument
525 ?^  product.argument [%2 trace]
526 ?= (%1 product.argument) [%0 %.y]
527 ?= (%0 product.argument) [%0 %.n]
528 [%2 trace]
529

```

This is invoked as:

```

530
531
532 > (pink [0 13 1 0] *$-(^ (unit (unit))))
533 [%0 product=0]
534 > (pink [0 13 1 1] *$-(^ (unit (unit))))
535 [%0 product=1]
536 > (pink [0 13 1 2] *$-(^ (unit (unit))))
537 [%2 trace=~]
538

```

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. (Neither of these are particularly complicated, but they entail some basic arithmetic operations as well.) By omitting `++scry` and opcode 12, as well as reducing to a structural match with a `+$tone` instead of an explicit Hoon type match, one could produce a reasonably sized Hoon program compiling to something close to the minimal viable Nock virtualized interpreter. No doubt intrepid code golfers will find it very much an upper bound rather than a lower.

This Nock formula produces a working virtualized Nock interpreter derived from Hoon code with some hand-tuned design. The input function signature is `[subject=* formula=*]` with output of the form `[@ product=*]`; if the

557 head is %0, then the product is a valid Nock noun, while a head
 558 of %bail indicates a crash in the evaluation. Opcode 9 calls
 559 may be addressed into the core as subject, and the formula it-
 560 self is complete with an empty subject.

```

561 [1
562   :: +mul multiplication of two atoms
563   [8 [1 1 1]
564     [1 8 [1 0]
565       8 [1 6 [5 [1 0] 0 60]
566         [0 6]
567           9 2 10 [60 8 [9 686 0 31]
568             9 2 10 [6 0 124] 0 2]
569             10 [6 8 [9 20 0 31]
570               9 2 10 [6 [0 125] 0 14]
571                 0 2] 0 1]
572           9 2 0 1] 0 1]
573   [:: +add addition of two atoms
574   [8 [1 0 0]
575     [1 6 [5 [1 0] 0 12]
576       [0 13]
577         9 2 10
578         [6 [8 [9 686 0 7] 9 2 10 [6 0 28] 0 2]
579           4 0 13]
580           0 1] 0 1]
581   [:: +list type definition
582   [8
583     [8 [1 0]
584       [1 8 [0 6] 8 [5 [0 14] 0 2] 0 6]
585       0 1]
586     [1 8 [1 0]
587       [1 8
588         [6 [3 0 6]
589           [[6 [5 [1 0] 0 12]
590             [1 0]
591             0 0]
592           8 [0 30] 9 2 10 [6 0 29] 0 2]
593           6 [5 [1 0] 0 6]
594           [1 0]
595           0 0]
596         8 [5 [0 14] 0 2] 0 6]
597         0 1] 0 1]
598
```

```

599     [:: +dvr division with remainder
600     8 [1 1 1]
601     [1 6 [5 [1 0] 0 13]
602     [0 0]
603     8 [1 0]
604     8 [1 6 [8 [9 687 0 31]
605     9 2 10 [6 [0 124] 0 125] 0 2]
606     [[0 6] 0 60]
607     9 2 10
608     [60 8 [9 23 0 31]
609     9 2 10 [6 [0 124] 0 125] 0 2]
610     10 [6 4 0 6] 0 1]
611     9 2 0 1] 0 1]
612 [:: +cap:
613 8 [1 0]
614 [1 6 [5 [1 2] 0 6]
615 [1 2]
616 6 [5 [1 3] 0 6]
617 [1 3]
618 6 [6 [5 [1 1] 0 6] [1 0] 5 [1 0] 0 6]
619 [0 0]
620 9 2 10
621 [6 7 [8 [9 170 0 7]
622 9 2 10
623 [6 [0 14] 7 [0 3] 1 2]
624 0 2] 0 2]
625 0 1] 0 1]
626 [:: +dec decrement an atom
627 8 [1 0]
628 [1 6 [5 [1 0] 0 6]
629 [0 0]
630 8 [1 0]
631 8 [1 6 [5 [0 30] 4 0 6]
632 [0 6]
633 9 2 10 [6 4 0 6] 0 1]
634 9 2 0 1] 0 1]
635 :: +lth less-than test
636 8 [1 0 0]
637 [1 6 [6 [5 [0 12] 0 13] [1 1] 1 0]
638 [6 [8
639 [1 6 [5 [1 0] 0 28]
640 [1 0]

```

```

641          6 [6 [6 [5 [1 0] 0 29]
642              [1 1]
643              1 0]
644          [6 [9 2 10
645              [14 [8 [9 686 0 15]
646                  9 2 10 [6 0 60]
647                  0 2]
648              8 [9 686 0 15]
649              9 2 10 [6 0 61] 0 2]
650              0 1]
651          [1 0]
652          1 1]
653          1 1]
654          [1 0] 1 1] 9 2 0 1]
655          [1 0] 1 1] 1 1] 0 1]
656  [:: +jink, raw virtual Nock
657      8 [1 0 0]
658      [1 8
659          [1 [:: +frag fragmentary subtree of noun
660              8 [1 0 0]
661                  [1 6 [5 [1 0] 0 12]
662                      [1 0]
663                      8 [1 6 [5 [1 1] 0 28]
664                          [[1 0] 0 29]
665                          6 [6 [3 0 29] [1 1] 1 0]
666                          [1 0]
667                      8 [8 [9 342 0 127] 9 2 10 [6 0 60] 0 2]
668                      9 2 10
669                      [14 [8 [9 87 0 255]
670                          9 2 10 [6 0 124] 0 2]
671                          6 [5 [1 2] 0 2]
672                          [0 122]
673                          6 [5 [1 3] 0 2]
674                          [0 123]
675                          1 1.818.845.538 0]
676                          0 3] 9 2 0 1] 0 1]
677  [:: +edit edit an atom at a given axis
678      8 [1 0 0 0]
679          [1 6 [5 [1 1] 0 12]
680              [[1 0] 0 27]
681              6 [6 [3 0 26] [1 1] 1 0]
682              [1 0]

```

```

683         8 [9 2 10
684           [26
685             8 [8 [9 342 0 63]
686               9 2 10 [6 0 28] 0 2]
687             6 [5 [1 2] 0 2]
688             [0 116]
689             6 [5 [1 3] 0 2]
690             [0 117]
691             0 0]
692           10 [12 8 [9 87 0 127]
693             9 2 10 [6 0 60] 0 2] 0 3]
694         6 [5 [1 0] 0 2]
695         [1 0]
696         6 [5 [1 2] 0 6]
697         [[1 0] [0 5] 0 245]
698         6 [5 [1 3] 0 6]
699         [[1 0] 0 244]
700         1 1.818.845.538 0]
701       0 5] 0 1]
702     6 :: Cell distribution
703     [6 [3 0 29] [3 0 58] 1 1]
704     [8 [9 11 10 [29 0 58] 0 1]
705       6 [5 [1 0] 0 4]
706       [8 [9 11 10 [29 0 123] 0 3]
707         6 [5 [1 0] 0 4]
708         [[1 0] [0 13] 0 5]
709         0 2]
710       0 2]
711     6 :: Opcode 0
712     [6 [3 0 29]
713       [6 [5 [1 0] 0 58]
714         [6 [3 0 59] [1 1] 1 0]
715         1 1]
716       1 1]
717     [8 [8 [9 4 0 1]
718       9 2 10 [6 [0 123] 0 60] 0 2]
719       6 [5 [1 0] 0 2]
720       [1 1.818.845.538 0]
721       [1 0]
722     0 5]
723     6 :: Opcode 1
724     [6 [3 0 29]

```

```

725         [5 [1 1] 0 58] 1 1]
726     [[1 0] 0 59]
727     6 :: Opcode 2
728     [6 [3 0 29]
729         [6 [5 [1 2] 0 58] [3 0 59] 1 1] 1 1]
730     [8 [9 11 10 [29 0 118] 0 1]
731         6 [5 [1 0] 0 4]
732         [8
733             [9 11 10 [29 0 247] 0 3]
734             6
735             [5 [1 0] 0 4]
736             [9 11 10 [14 [0 13] 0 5] 0 7]
737             0 2]
738         0 2]
739     6 :: Opcode 3
740     [6 [3 0 29]
741         [5 [1 3] 0 58] 1 1]
742     [8 [9 11 10 [29 0 59] 0 1]
743         6 [5 [1 0] 0 4]
744         [[1 0] 3 0 5]
745         0 2]
746     6 :: Opcode 4
747     [6 [3 0 29]
748         [5 [1 4] 0 58] 1 1]
749     [8
750         [9 11 10 [29 0 59] 0 1]
751         6
752         [5 [1 0] 0 4]
753         [6 [6 [3 0 5] [1 1] 1 0]
754             [[1 0] 4 0 5]
755             1 1.818.845.538 0]
756         0 2]
757     6 :: Opcode 5
758     [6 [3 0 29]
759         [6 [5 [1 5] 0 58] [3 0 59] 1 1] 1 1]
760     [ 8
761         [9 11 10 [29 0 118] 0 1]
762         6
763         [5 [1 0] 0 4]
764         [ 8
765             [9 11 10 [29 0 247] 0 3]
766             6

```

```

767         [5 [1 0] 0 4]
768         [[1 0] 5 [0 13] 0 5]
769         0 2]
770     0 2]
771 2746 6 :: Opcode 6
772     [6 [3 0 29]
773       [6 [5 [1 6] 0 58]
774         [6 [3 0 59] [3 0 119] 1 1]
775         1 1] 1 1]
776 2746 [8 [9 11 10 [29 0 118] 0 1]
777     6
778     [5 [1 0] 0 4]
779     [6 [5 [1 0] 0 5]
780       [9 11 10 [29 0 494] 0 3]
781 2814 6 [5 [1 1] 0 5]
782     [9 11 10 [29 0 495] 0 3]
783     1 1.818.845.538 0]
784 :: 11
785 :: [1.936.945.012 1 0]
786 2816 :: 1 1.818.845.538 0
787     0 2]
788 6 :: Opcode 7
789     [6 [3 0 29]
790       [6 [5 [1 7] 0 58] [3 0 59] 1 1] 1 1]
791 2816 [8 [9 11 10 [29 0 118] 0 1]
792     6 [5 [1 0] 0 4]
793       [9 11 10 [14 [0 5] 0 247] 0 3]
794       0 2]
795 6 :: Opcode 8
796 2816 [6 [3 0 29]
797     [6 [5 [1 8] 0 58] [3 0 59] 1 1] 1 1]
798     [8 [9 11 10 [29 0 118] 0 1]
799     6 [5 [1 0] 0 4]
800       [9 11 10 [14 [[0 5] 0 60] 0 247] 0 3]
801 2816 0 2]
802 6 :: Opcode 9
803     [6 [3 0 29]
804     [6 [5 [1 9] 0 58]
805       [6 [3 0 59]
806         [6 [3 0 118] [1 1] 1 0]
807         1 1]
808       1 1]

```

```

809         1 1]
810     [8 [9 11 10 [29 0 119] 0 1]
811         6 [5 [1 0] 0 4]
812         [8 [8 [9 4 0 3]
813             9 2 10 [6 [0 502] 0 13] 0 2]
814         6 [5 [1 0] 0 2]
815         [1 1.818.845.538 0]
816     9 11 10 [14 [0 13] 0 5]
817     0 7]
818     0 2]
819     6 :: Opcode 10
820     [6 [3 0 29]
821         [6 [5 [1 10] 0 58]
822             [6 [3 0 59]
823                 [6 [3 0 118]
824                     [6 [3 0 236] [1 1] 1 0]
825                         1 1]
826                             1 1]
827                                 1 1]
828                                     1 1]
829     [6 [5 [1 0] 0 236]
830         [1 1.818.845.538 0]
831     8 [9 11 10 [29 0 119] 0 1]
832     6 [5 [1 0] 0 4]
833         [8 [9 11 10 [29 0 493] 0 3]
834         6 [5 [1 0] 0 4]
835         [8 [8 [9 10 0 7]
836             9 2 10 [6 [0 2.028] [0 29] 0 13]
837                 0 2]
838                 6 [5 [1 0] 0 2]
839                 [1 1.818.845.538 0]
840                 [1 0]
841                 0 5]
842                 0 2]
843                 0 2]
844     6 :: Opcode 11
845     [6 [3 0 29]
846         [6 [5 [1 11] 0 58]
847             [6 [3 0 59]
848                 [6 [3 0 118]
849                     [1 1]
850                         1 0]

```



```

851             1 1]
852             1 1]
853             1 1]
854         [8 [9 11 10 [29 0 119] 0 1]
855             6 [5 [1 0] 0 4]
856             [[1 0]
857             2 [0 60] [1 11] [0 246] [1 1] 0 5]
858             0 2]
859 ::         11
860 ::         [1.936.945.012 1 0]
861             1 1.818.845.538 0]
862             9 11 0 1]
863             0 1]
864 :: +mas
865 8 [1 0]
866     [1 6 [6 [5 [1 3] 0 6] [1 0] 5 [1 2] 0 6]
867         [1 1]
868         6 [6 [5 [1 1] 0 6] [1 0] 5 [1 0] 0 6]
869         [0 0]
870         8 [9 20 0 7]
871         9 2 10
872         [6 [7 [0 3] 7 [8 [9 170 0 7]
873             9 2 10
874             [6 [0 14] 7 [0 3] 1 2]
875             0 2] 0 3]
876             7 [0 3] 8 [9 4 0 7]
877             9 2 10 [6
878             [7 [0 3]
879             9 2 10
880             [6 7 [8 [9 170 0 7]
881             9 2 10 [6 [0 14] 7 [0 3] 1 2]
882             0 2] 0 2]
883             0 1]
884             7 [0 3] 1 2]
885             0 2]
886             0 2]
887             0 1]
888 [:: +unit, type definition of unit/Maybe
889 8
890     [8 [1 0] [1 8 [0 6] 8 [5 [0 14] 0 2] 0 6] 0 1]
891     [1 8 [1 0]
892         [1 8 [6 [3 0 6]

```

```

893             [[8 [0 30] 9 2 10 [6 0 28] 0 2]
894              8 [7 [0 7] 8 [9 22 0 7]
895               9 2 10 [6 0 14] 0 2]
896              9 2 10 [6 0 29] 0 2]
897          6 [5 [1 0] 0 6]
898          [1 0]
899          0 0]
900      8 [5 [0 14] 0 2]
901      0 6]
902      0 1]
903      0 1]
904      :: +sub subtraction of two atoms
905      8 [1 0 0]
906      [1 6 [5 [1 0] 0 13]
907         [0 12]
908         9 2 10
909         [6 [8 [9 686 0 7] 9 2 10 [6 0 28] 0 2]
910          8 [9 686 0 7] 9 2 10 [6 0 29] 0 2]
911      0 1] 0 1]

```

913 TODO there's a missing open bracket I need to chase down

914 4 Conclusion

915 Metacircular virtualization is a powerful tool for building in-
916 terpreters, and Nock is not only no exception, it exemplifies
917 the practice. The design of ++mock and its underlying ++mink
918 function provides a practical way to interpret Nock in a way
919 that is both efficient and extensible. One could imagine, for
920 instance, a Nock interpreter cousin to ++mock that is instru-
921 mented for partial evaluation of formulas (as in a debugger or
922 syntax highlighter robust to syntax errors). Nock supports a
923 rich capacity to build metacircular interpreters as a first-class
924 feature, a potential which remains as yet relatively unexplored
925 in the Nock ecosystem.

References

926

- 927 ~sorreg-namtyv, Curtis Yarvin (2006) “U, a small model”. URL:
928 <http://urbit.sourceforge.net/u.txt> (visited on
929 ~2024.2.20).
- 930 — (2010a) “Urbit: functional programming from scratch”.
931 URL: [http://moronlab.blogspot.com/2010/01/urbit-](http://moronlab.blogspot.com/2010/01/urbit-functional-programming-from.html)
932 [functional-programming-from.html](http://moronlab.blogspot.com/2010/01/urbit-functional-programming-from.html) (visited on
933 ~2024.1.25).
- 934 — (2010b) “Watt: A Self-Sustaining Functional Language
935 (preprint)”. URL: [https://github.com/cgyarvin/urbit/](https://github.com/cgyarvin/urbit/blob/master/Spec/watt/sss10.tex)
936 [blob/master/Spec/watt/sss10.tex](https://github.com/cgyarvin/urbit/blob/master/Spec/watt/sss10.tex) (visited on
937 ~2025.5.19).