Metacircular Virtualization & Practical Nock Interpretation

N. E. Davis ~lagrev-nocfep
Zorp Corp

Abstract

Contents

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. Practical Nock interpreters handle this by running Nock within Nock, a

Manuscript submitted for review.

Address author correspondence to ~lagrev-nocfep.

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**,

[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**).

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 (**Yarvin2006**; **Yarvin2010**, **Yarvin2006**; **Yarvin2010**). 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. (**Yarvin2010a**, **Yarvin2010a**)

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

3.2 Language extension

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

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

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

[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=@]
          ==
```

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:

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

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:

```
  [%1 constant=*]
[%0 constant.formula]
```

> 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.

```
::  +mink: raw virtual nock
::
++  mink  !.
  ~/  %mink
  |=  $:  [subject=* formula=*]
          scry=$-(^ (unit (unit)))
      ==
  =|  trace=(list [@ta *])
  |^  ^-  tone
      ?+  formula  [%2 trace]
          [^ *]
        =/  head  $(formula -.formula)
        ?.  ?=(%0 -.head)  head
        =/  tail  $(formula +.formula)
        ?.  ?=(%0 -.tail)  tail
        [%0 product.head product.tail]
      ::
          [%0 axis=@]
        =/  part  (frag axis.formula subject)
        ?~  part  [%2 trace]
        [%0 u.part]
      ::
          [%1 constant=*]
        [%0 constant.formula]
      ::
          [%2 subject=* formula=*]
        =/  subject  $(formula subject.formula)
        ?.  ?=(%0 -.subject)  subject
        =/  formula  $(formula formula.formula)
        ?.  ?=(%0 -.formula)  formula
        %=  $
          subject  product.subject
          formula  product.formula
        ==
      ::
          [%3 argument=*]
        =/  argument  $(formula argument.formula)
        ?.  ?=(%0 -.argument)  argument
        [%0 .?(product.argument)]
      ::
          [%4 argument=*]
```

```
  =/  argument  $(formula argument.formula)
  ?.  ?=(%0 -.argument)  argument
  ?^  product.argument  [%2 trace]
  [%0 .+(product.argument)]
::
  [%5 a=* b=*]
=/  a  $(formula a.formula)
?.  ?=(%0 -.a)  a
=/  b  $(formula b.formula)
?.  ?=(%0 -.b)  b
[%0 =(product.a product.b)]
::
  [%6 test=* yes=* no=*]
=/  result  $(formula test.formula)
?.  ?=(%0 -.result)  result
?+  product.result
      [%2 trace]
  %&  $(formula yes.formula)
  %|  $(formula no.formula)
==
::
  [%7 subject=* next=*]
=/  subject  $(formula subject.formula)
?.  ?=(%0 -.subject)  subject
%=  $
  subject  product.subject
  formula  next.formula
==
::
  [%8 head=* next=*]
=/  head  $(formula head.formula)
?.  ?=(%0 -.head)  head
%=  $
  subject  [product.head subject]
  formula  next.formula
==
::
  [%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
    ==
  ::
    [%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]
  ::
    [%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]
  ::
    [%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]
        ?~  u.result
          [%2 [%hunk product.ref product.path] trace]
        [%0 u.u.result]
      ==
  ::
```