Vase Mode

Abstract

Nockchain and Urbit are platforms which run on a virtual machine layer based on an SKI-derived combinator called Nock. Nock is homoiconic and only recognizes nonnegative integers and pairs as values. Thus Nock's type system is extremely simple. Although it is untyped, we want to build strongly typed computations on top of this instruction set architecture.
The Hoon programming language solves this by using a pair of a type description and a noun, or raw Nock value. The type description, or type of the subject, contains information like variable names, variable types, aliases, and the overall structure of the execution environment. Essentially, it acts as a strictly defined metadata reference set for the associated Nock value and its execution environment. We call such a pair of type and value a “vase”.
Vase mode, then, describes a way of lifting statically-typed values from their normal compile-time world into a dynamically-typed operating environment, which is then used to produced valid and validated Nock ISA code while enforcing the Hoon language type constraints. Vase mode is used in building compilers, interpreters, and code imports. Since code is transmitted as a noun value to be built and executed on the receiver's side, this also affects hotloading code and system upgrades.
A vase algebra describes how to combine the type subject information and the associated value information, as in an import or a REPL. This lets Hoon have human-legible affordances in type and validation, while retaining a simple target ISA for execution, code proving, and optimized execution.
In this talk, we'll explore how vase mode works, how it's used, and how it can result in a tiny clean ISA even across multiple possible targets. We can compare this to how dynamic typing is implemented in Scheme, and “gradual typing”, and the `Dynamic` type in Haskell as well.

Introduction

I will frame my discussion between two approaches, what I will call the library and the garage. The library takes an extremely formal perspective: this is the lambda calculus and the Turing machine. The garage replies, well, that's all fine and good but we have to make chips, you know. There's always been a tension there, Lisp vs. Fortran.
Computers are used to compute, obviously. What we mean by that was the result of centuries of consideration and false starts. I'll gloss over “computation” a bit here, but I will proceed from the following basic standpoint:

1. Computation requires values, operations, and state.
2. Any computation requires a representation.
3. Representations are interchangeable.
4. All components can be represented unambiguously on the machine.
5. Types guide interpretation.

Machine representation on a modern digital computer requires us to write down a value as a bare ordered sequence of bits. Even at the architecture level, the chip interprets such sequences using some kind of type, meaning that we can talk about a sequence of bits as “meaning” one of:

1. Command
2. Number
3. Address

Conventionally, the type in this sense is determined by the position of the sequence. This is the garage type system. It has advantages, such as never needing additional metadata, but shorn of this structure it is impossible to tell “what” a sequence of bits “means”.
The library's response to this is to tell us that we assign mathematical categories to each sequence of bits which allows us to determine how it can be used. In particular, we end up with two practical approaches to managing type in programming languages: static type and dynamic type. In fact, where we are headed with this talk on vase mode is how the bridge is constructed from statically-typed Hoon, to a dynamically-typed runtime space, back to the untyped Nock code which results.
While we could use some part of the bit sequence itself to tell us what the associated value “means”, such as is sometimes done to distinguish kinds of addresses (as pointers), what we commonly do instead is associate a separate type with the value. This metadata and the value it is associated with must be meticulously tracked together; this is very commonly handled by the compiler such that the end developer doesn't need to do anything special to take advantage of it besides declaring the type.
Different interpreters and compilers call this pair different things. For instance, CPython calls this a “boxed value” and keeps the value as a pair of datum and type tag. We are going to call it a vase, which is essentially saying that we have a value and we keep it in a certain “shape” by pouring it into its appropriate vase. So a `vase` is a pair of type and data. Vase mode, therefore, describes working with such pairs rather than first-order typed data (the library approach) or strictly sequences of bits (the garage approach).
Gödel's line: "Trees are the most inspiring structures"

We don't need to know a lot about Hoon or Nock to proceed today, but we will see a little of both incidentally. The remainder of this talk will treat vases at a high level, but using some of the terminology of Nock and Hoon.

1. Nock is an SKI-derived combinator calculus, sometimes described as a pico-Lisp. It was developed back in the 00s and first applied as the instruction set architecture of the Urbit virtual machine. Later on, it has been employed by Zorp (my employer) to produce the Nockchain zero-knowledge VM. I will be using a slightly simplified Nock today. (Specifically, I will omit compound runes and homoiconicity of the original in favor of something more legible.)

2. Hoon is a macro-like systems language that compiles to Nock. Hoon is rather unique and APL-like in that it uses ASCII digraphs to denote operations. This leads to highly visually structured code, but is often dissuading to developers seeing it for the first time. While it is what we actually write vases in, I will use a schematic version of vases rather than Hoon in my examples today.

(examples of atoms) (example of combining vase of `[@ @]`)
What vase mode permits over raw Nock manipulations is:

1. allowing types to be straightforwardly combined
2. permitting code evaluation (Lisp `eval`)
3. facilitating metacircular interpretation
4. managing persistent code artifacts
5. building and running system upgrades while running live (i.e. hotswapping code)

That is, vases enable the end developer to work at a higher level of abstraction than raw typed code and data would itself make straightforward in a statically typed functional programming environment.
Vases receive bare mention in the 2016 Urbit whitepaper. They are noted merely as having the property of allowing a statically typed compiler to operate on dynamically typed runtime code—and thus, to reload portions of the kernel from source while running.
However, vases are critically important to practical code composition and execution on Urbit and NockApp as functional-as-in-paradigm execution environments:

> 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. (Ted Blackman, “Why Hoon”)

To start us off, I want to work an abstract example of a variable definition. Then I will take it down to a concrete level, and finally show a synthesis of the same operation using vase mode.
This is the essential program we are implementing:

```
let a = 5;
a
```

Raw variable with variable push and reference.

```
[8 [1 5] [0 2]]
```

Vase mode.

```
[[%atom %ud ~] 5]
```

No combination yet, so this is still a bare variable description.

What we see is a type with some qualifying metadata:

- `%atom` means a bare non-negative integer.
- `%ud` means an unsigned decimal.
- `~` in this case means non-constant.

Next, let's combine values.

```
let a:@ = 5;
let b:@ = 10;
(a b)
```

Raw Nock:

```
[8 [1 5] 8 [1 10] [[0 6] [0 2]]]
```

Vase type:

```
[[%cell [%atom %ud ~] [%atom %ud ~]] [5 10]]
```

In order to evaluate code against a given subject as vase, I need to be able to convert a text string into an abstract syntax tree for evaluation. In Hoon, this is `+ream`:

```
> !>(a=42)
[#t/a=@ud q=42]

> (slap !>(a=5) (ream '42'))
[#t/@ud q=42]
```

Because Nock only knows about nouns, that is, atoms and cells, the type information is retained only in the vase. Here are a few other types:

```
 > !>(0b10.1010)
[#t/@ub q=42]

> !>(0xdead.beef)
[#t/@ux q=3.735.928.559]

> !>(~zod)
[#t/@p q=0]
```

We can look into `+ream`'s result and the actual AST, but in general that's out of scope for this. We will just rely on `+ream` directly where necessary.

```
> (ream '42')
[%sand p=%ud q=42]

> (ream 'a=42')
[%ktts p=term=%a q=[%sand p=%ud q=42]]
```

(You can also manually construct an AST. In my normal work on the Jock language, I do this in the compiler, besides hand-rolling Nock expressions.)

- `=compile (cury ~(mint ut %noun) %noun)`

In Nock, we are essentially pinning a new variable at the head of our existing formula, then our tail is the actual access by binary tree address.

```
(~(mint ut %noun) %noun (ream '=/(a 5 =/(b 10 [a b]))'))
[@ @]
[8 [1 5] 8 [1 10] [[0 6] [0 2]]]
```

Now, we may have values which appear identical in their resultant Nock, but which Hoon would enforce different type constraints upon, such as a null-terminated tuple of finite length and a list:

```
:: null-terminated tuple
> (compile (ream '~[1 2 3]'))
[#t/[@ud @ud @ud %~] q=[%1 p=[1 2 3 0]]]


:: formal list type with deferred evaluation
> (~(mint ut -:!>(.)) %noun (ream '(gulf 1 3)'))
[#t/it(@) q=[%8 p=[%9 p=200.245.085 q=[%0 p=1.023]] q=[%9 p=2 q=[%10 p=[p=6 q=[p=[%7 p=[%0 p=3] q=[%1 p=1]]
q=[%7 p=[%0 p=3] q=[%1 p=3]]]] q=[%0 p=2]]]]]
```

We have to be able to envase `E` and devase `D` a value, which we will write Lisp-style. We also want to be able to combine vases by “slopping” them together ,`S` for slop. Finally, we need to “slap” an expression we wish to evaluate against a vase, `R` for run. Everything else in vase mode can be built out of these primitives.

A vase is a pair of type and data, particularly as used in Nock and related languages. Vase mode, therefore, describes working with such pairs rather than first-order typed data. The vase algebra allows types to be straightforwardly combined, permitting code evaluation, facilitating metacircular interpretation, managing persistent code artifacts, and otherwise working at a higher level of abstraction than raw typed code and data. In this talk, we will examine the role of vases, the vase algebra, and work through several examples of how and why vase mode is useful throughout Nock-based platforms.

We will employ two main tools in Hoon to envase (or make dynamic) and devase (or make static):

- `!>` zapgar envases, or produces a `vase`, a pair of `type` and `noun`.
- `!<` zapgal devases, or produces a statically-typed value from a `vase`. It requires you to know what type you expect so that you can match them.

The second process is less common; while you can trivially ascend from a statically-typed environment to a dynamically-typed one, to enter Hoon's static layer again is more difficult and you have to know exactly what the type of the result is beforehand. (If the value does not actually fit the type it claims, we say that it is an “evil vase“ and cannot process it.) Normally, once you have a vase in the code builder, you will use the vase algebra which we will discuss in a moment to produce static untyped Nock code instead of static Hoon code.

<evil-vase>

(A quick review of the Urbit kernel code base suggests that most instances of `!<` unsafe vase elimination are either unpacking command-line arguments which have in principle been vetted by the runtime first, or handling a received value over the network, presumably from its remote counterpart.)

Urbit's application architecture has a persistent solid-state kernel which runs user applications in a sort of sandbox. They cannot use vase mode directly, but they are built and modified using vase mode. In particular, library imports and the REPL interface both use vase mode extensively. Likewise, NockApp supplies a kernel wrapping specialized I/O drivers for the executable Nock, produced by Hoon or Jock. Vases are generally used for building code sent as nouns over the wire and in the compilers.

Vase Algebra

+slop

Now that we know a bit about vases as artifacts, let's start to work with them.
If I ask the Hoon REPL what the vase of a pair of numbers is, it generates the following description:

```
> !>([100 200])
[#t/[@ud @ud] q=[100 200]]

> !<([@ud @ud] !>([100 200]))
[100 200]
```

Let's suppose that I intend to append a hexadecimal number as a tail. The original vase no longer holds:

```
> !>([[100 200] 0x12c])
[#t/[[@ud @ud] @ux] q=[[100 200] 300]]

> !<([[@ud @ud] @ux] !>([[100 200] 0x12c]))
[[100 200] 0x12c]
```

Combining these manually can be done, but it's very tedious and error-prone. To combine them dynamically I can use the first tool of our vase algebra, `+slop`. It combines two vases, which means combining their types and their values consistently.
Here, I `++slop` to produce the cell of those two nouns:

```
> !>([100 200])
[#t/[@ud @ud] q=[100 200]]

> !>(0x12c)
[#t/@ux q=300]

> (slop !>([100 200]) !>(300))
[#t/[[@ud @ud] @ud] q=[[100 200] 300]]
```

+slap

Okay, so we can manipulate vases. To use them, we need to hearken back to our list of reasons to use vases at all: chief among them was the ability to dynamically evaluate code at runtime. We will use `+slap` to calculate the result of a Hoon expression against a given subject as vase.
With that, we can resolve a variable name (or "face") against a vase:

```
> !>(a=42)
[#t/a=@ud q=42]

> (slap !>(a=42) (ream 'a'))
[#t/@ud q=42]

> (slap !>(a=42) (ream '[a a]'))
[#t/[@ud @ud] q=[42 42]]
```

A much more interesting case arises when we introduce functions to the mix. Since we always need to know about a function to evaluate it, I will always `+slop` in the Hoon subject including its standard library as .. Furthermore, if we don't have variable names in this example, we can refer to the values by their binary tree addresses, in this case the head of the tail and the tail of the tail.

```
> (slap (slop !>([a=1 b=2]) !>(.)) (ream '(add a b)'))
[#t/@ q=3]

> (slap (slop !>([1 2]) !>(.)) (ream '(add +6 +7)'))
[#t/@ q=3]
```

(It is conventional to +slop new information on the left, because of the binary tree search order.)
What else would we like to be able to do, given a subject?
The received wisdom from core developers has been that one can build whichever higher-level operations one needs from +slap and +slop. This seems to be more or less true, but the mechanics of it start to depend very much on your high-level language and its relation to the instruction set architecture it targets. In practice, Hoon provides some supplemental functions for vase mode because we need to handle the stack trace for errors as well as supplying a context for certain out-of-subject references in userspace code.

+slab

+slab tells us, given a subject, whether a name is present.

```
> (slab %read %a -:!>(a=5))
%.y

> (slab %read %a -:!>(b=5))
%.n
```

(There is a %rite as well but I don't know how it's used.)

+slam

Given a function and its arguments as vases, produce a vase containing the result.

```
> (slam !>(|=([a=@ud b=@ud] [b a])) !>([1 2]))
[#t/[@ud @ud] q=[2 1]]
```

+text

Vases are also used for the prettyprinter, since the conversion from raw value to standardized text output requires type information to resolve.

```
> (text !>(~[1 2 3]))
"[1 2 3 ~]"

> (text !>((gulf 1 3)))
"~[1 2 3]"
```

Aside from head/tail references, I have avoided relying on Nock's nature as a binary tree when evaluating code. However, I can also access and manipulate code via tree axes (e.g. +slew).
There is also a system for deferring a +slap (+swat) so that we can doubly-defer code in certain cases.

+mock

Nock is a crash-only language; it has no error handling when run raw. To produce a manageable system, Nock is almost always run in a virtualized mode called +mock. While the main operator of this engine works only on raw untyped Nock, some of the adjuncts do preserve type for

subsequent use of the outputs as statically-typed values to be raised back into vases at various points in the compiler.
(tie back in library/garage stuff)
https://x.com/i/grok?conversation=1909688391047737438

- urbit/urbit PR #1446, original proposal for `!<` devase
- urbit/urbit PR #6858, proposed additions to vase algebra

$type

What is $type in Hoon, the head of our vase? So far we've only seen atoms and cells, but there are a few more.

```
+$  type  $+  type
      $~  %noun                           ::
      $@  $?  %noun                       ::  any nouns
              %void                       ::  no noun
          ==                              ::
      $%  [%atom p=term q=(unit @)]       ::  atom / constant
          [%cell p=type q=type]           ::  ordered pair
          [%core p=type q=coil]           ::  object
          [%face p=$@(term tune) q=type]  ::  namespace
          [%fork p=(set type)]            ::  union
          [%hint p=(pair type note) q=type]  ::  annotation
          [%hold p=type q=hoon]           ::  lazy evaluation
      ==                                  ::
```

What does this mean? Let's restructure it a bit:

- %noun, undiscriminated noun
- %void, no noun (compiler stage)
- [%atom term (unit @)], atom or constant. A constant must match the value; text constants are very commonly associated with type discrimination.
- [%cell type type], cell.
- [%core type coil], object. A core has a collection of methods, types, and polymorphic behavior.
- [%face term type], namespace or label. An indexed collection of other types.
- [%fork (set type)], type union. A type which has not yet been resolved to a particular member.
- [%hint [type note] type], runtime hint/annotation.
- [%hold type hoon], deferred (lazy) evaluation. An interesting example of this arises with the vase of a list: !>((gulf 1 10)).

You could imagine other type systems, but this is Hoon's.

As an aside, while vase is a reasonably good name for a thing and the thing that shapes it, the original term was vise, which also works in this capacity.

Vase Algebra Redux

+slop

To write a `+slop` which, as you'll recall, conses two vases together, is straightforward:

```
++  slop
  |=  [hed=vase tal=vase]
  ^-  vase
  [[%cell p.hed p.tal] [q.hed q.tal]]
```

+slap

How do we actually write code to `+slap` a given Hoon expression against a subject as vase?

```
++  slap
  |=  [vax=vase gen=hoon]  ^-  vase
  =+  gun=(~(mint ut p.vax) %noun gen)
  [p.gun .*(q.vax q.gun)]
```

Gates

Default Sample

Now, at the Nock level, a gate is a structure that follows a convention putting the arms in the battery, the sample at +6, and the context at +7. The sample contains simply the bunt, or default information, of

Dry and Wet Gates

Coming from a higher-level typed language, there is more than one way to build a Nock gate. Hoon, for instance, uses what are called "dry" and "wet" approaches. (In Jock, all gates are dry.) Put briefly, a dry gate is essentially a regularly typed gate: function argument types are checked and then inserted into the AST expression, which is built via vases in the conventional way.
<example>
In contrast, a wet gate is more like a C-style macro, in which the types are accepted as tentatively passing the necessary checks, then as long as they work the original types are passed back out as appropriate. It's a way of asking if the Nock formula itself will just work—if it will, then build it and return the associated type we expect. (This is how typed lists are built, for instance.)
<example>
You see wet gates used frequently as gate builders—that is, if you need a gate operating for a particular type, the easiest way to do it is to produce a gate specialized by type. (You can see the dynamism enabled by vases shining through the cracks of this model.)

Asides

Typed Quote

There is also a "typed quote" `!;` zapmic rune, which wraps the product of its second child as the type of the example given as the first child. This is not commonly used.

```
> !;  *type  [1 0x0]
[#t/[@ud @ux] 1 0x0]
```

Beyond the Vase Algebra

Hoon used to directly expose many of the mechanics of this when file handles could be produced using a sophisticated but complicated build system.

It still uses vase mode explicitly to import libraries, in particular, but that's the only place that a userspace developer would typically encounter it. Vases are used in transmitting certain kinds of information, but the vase algebra has been restricted to the compiler level.

The language I am working on for Zorp, Jock, uses vase mode somewhat in its compiler, but actually manages its type in a similar system that leans on Hoon's implicit management instead of explicitly slapping and slopping. Because Jock is more dynamically typed from the beginning, it doesn't need to think of elevating static values into a dynamic context to operate with them—the necessary information is already available in the compiler.

Metavases

A meta-vase is a `vase` of a `vase`, that is, an untyped `vase`. These can be used to check structural data nesting without enforcing type, for instance. (Urbit types used to be total; that is, on a mismatch they would return their default rather than crashing.)

The vane interface is normally strictly typed, but using a metavase it can punch a hole through the `type` system. (This used to be the only way to do vase reduction before `!<` zapgal was introduced: a vane had to pass to the Arvo kernel to get into double vase mode, which Arvo would collapse into one vase mode and hand back.)

Since Urbit is built to handle hot-swapped kernel updating, one interesting complication with vase handling lies in upgrades to the type system itself:

> Some subtleties regarding types arise when handling OTA updates, since they can potentially alter the type system. Put more concretely, the type of type may be updated. In that case, the update is an untyped Nock formula from the perspective of the old kernel, but ordinary typed Hoon code from the perspective of the new kernel. Besides this one detail, the only functionality of the Arvo kernel proper that is untyped are its interactions with the Unix runtime.

Urbit caches vases aggressively and I expect that NockApp will as well as it develops.

Beyond Nock

Type and data pairs as in vase mode is a general and generalizable approach. Another interesting question is to think about targeting other ISAs. No part of the vase algebra is specific to Nock except its instantiation as nouns.

During my preparation for this talk, I ran across "A Dependently Typed Assembly Language", a fascinating concept which shares some commonalities with vase mode via the expedient of a certifying compiler designed to guarantee type safety.

In any case, if the thought of working with a combinator directly to build programs intrigues you, I will be running a short workshop tomorrow afternoon

Comparisons

Lisp quote

In Lisp, ' is used to defer evaluation, or to treat an expression as data which can be introspected and evaluated at will.

```
(+ 1 2)        ; => 3
'(+ 1 2)       ; => (+ 1 2)
```

Without ', (+ 1 2) evaluates to 3. With ', it's a list: (list '+ '1 '2). This is very similar to the way that Hoon lets you directly construct and manipulate an AST, as with +ream.

```
(define prog '(+ 1 2))
(car prog)      ; => +
(cdr prog)      ; => (1 2)
(eval prog)     ; => 3


(eval '(+ 1 2))  ; => 3
```

This is basically equivalent to Hoon's +ream, altho the Hoon AST is somewhat more regularized and opaque. (I can actually do this without a text string, but the syntax is a bit convoluted and I don't want to muddy the waters any more in this short talk.)

```
> (ream '(add 1 2)')
[%cncl p=[%wing p=~[%add]] q=[i=[%sand p=%ud q=1] t=[i=[%sand p=%ud q=2] t=~]]]


> !,(*hoon (add 1 2))
[%cncl p=[%wing p=~[%add]] q=[i=[%sand p=%ud q=1] t=[i=[%sand p=%ud q=2] t=~]]]
```

A vase is a typed value that an AST, like ' or +ream produces, can be evaluated against. Lisp uses a global or explicit environment, whereas Hoon always lets you supply a subject of your choice, including the "current" subject (with .).

Vase mode is thus a "typed dynamic" system, unlike Lisp's raw dynamism.

- Why not just use eval? Scheme is untyped and error-prone here. Vases add type safety, which in Hoon's case is used to delimit possible kinds of operations.

Haskell Dynamic

Haskellers probably immediately thought of Dynamic. It's true: Nock's vases are similar to Haskell's Dynamic, used for injecting values into a dynamically typed value. Vases carry a subject, or a closure-like scope, not just an expression as data.

- In Haskell, Dynamic lets you pack values of different types into a uniform container and defer type checking until runtime. Like Hoon's vase mode, Dynamic lets you escape static typing temporarily and deferring type resolution.
- In Hoon, a type–value mismatch is an "evil vase" and results in a runtime crash. (Since we are commonly running in an emulated mode, this doesn't take down the whole kernel.) Dynamic will result in Nothing if the types don't match.
- Hoon's vases are much "closer to the metal", since they can trivially result in Nock code. In the end, Hoon's vase mode feels more like a dynamic interpreter or a VM operating against a dynamic subject, while Dynamic is more focused on typed payloads for data evaluation.
- Dynamic is data-centric, safe but limited. Vases carry scope and enable eval but risk evil vase crashes, a trade-off for power.

Conclusion

A vase algebra describes how to combine the type subject information and the associated value information, as in an import or a REPL. This lets Hoon have human-legible affordances in metaprogramming and validation, while retaining a simple target ISA for exeecution, code proving, and optimized execution.
We can think of vase mode as a gradual typing pipeline: we proceed from a static Hoon context, to a dynamic vase mode, to untyped Nock.

Appendix: Old Ford $horns

Hoon used to directly expose many of the mechanics of this when file handles could be produced using a sophisticated but complicated build system.
The old `+$horn` build system in Ford:

- 65a84b:docs/arvo/internals/ford.md
- 65a84b:docs/arvo/internals/ford/runes.md
- 65a84b:docs/arvo/internals/ford/commentary.md

![[../../Pasted image 20250401100446.png]] ![[../../Pasted image 20250401104827.png]]

Resources

- ~rovnys-ricfer, "Vases"
- Vase guide documentation
- Vase mode documentation (Core Academy)
- `++slap` and `++slop` documentation
- ~migrev-dolseg, "Vase Visualizer"
- Ted Blackman makes a tiny userspace handler using vase mode
- Scott J. Maddox, "Foundations of Dawn: The Untyped Concatenative Calculus"
- Maxime Chevalier-Boisvert, "Typed vs Untyped Virtual Machines"