Notes on Vases

N. E. Davis ~lagrev-nocfep
Zorp Corp

Abstract

This paper explores the concept of "vases" in the Hoon programming language, an approach to managing type dynamically in a static environment. Vases find applications throughout Nock solid-state systems, including compilation, metaprogramming, and dynamic code evaluation. We will explore how vases are constructed, manipulated, and utilized in the context of Nock and Hoon, providing a foundation for understanding their role in Urbit's programming ecosystem.

Contents

Urbit Systems Technical Journal II:1 (2025): 131–153.

Address author correspondence to ~lagrev-nocfep.

1 Introduction

Nock is homoiconic and only recognizes natural numbers or "atoms" (nonnegative integers, $\mathbb{Z}^*$) and pairs or "cells", as values. Although Nock is untyped, it is desirable to build typed languages targeting 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. In some ways like a macro over Nock, Hoon introduces static typing (thus extending from Nock's exclusive use of natural numbers) and other design patterns convenient to programmers.[1] Most of the time, developers work in "static Hoon", which maintains guard rails and is easier to reason about. However, a more flexible and powerful modality is available. 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

[1] The Jock language does not directly expose vases for utilization, but does utilize them inside the compiler, written in Hoon.

value a "vase".[2] Most commonly used in building code and in manipulating vanes, vases permit the developer to edit the type and the value directly. Essentially, a value can be lifted from a static Hoon context into a dynamic vase context, then reduced back to static Hoon or untyped Nock.

Vase mode, then, describes a design pattern for 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.[3]

2 Hoon Compilation in a Nutshell

Before we can really grasp the utility of vase mode, we require a brief diversion into how Hoon code is actually compiled and built into Nock. That is, given a Hoon expression, how do we get it into a form that can be executed by the Nock virtual machine? A number of convenience functions wrap various parts of the Hoon compiler, which primarily consists of `+vast` and `+ut`.

Text→Nock . To evaluate a string containing Hoon directly, one merely needs to `+make` the `cord` (text atom).[4]

[2] 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.

[3] A vase visualization tool has been produced by `~migrev-dolseg`, available at `https://hawk.computer`.

[4] Note that Nock instructions are marked as constants with `%` for legibility, but the value is simply the associated integer.

```
> (make '~[1 2 3]')
[%1 p=[1 2 3 0]]

> (make '=+(a=5 -)')
[%8 p=[%1 p=5] q=[%0 p=2]]
```

Since `+make` assumes no subject, all information must be provided in the expression. Furthermore, although not present explicitly, `+make` utilizes an internal `$type` of `%noun`.

```
> (make '-')
[%0 p=2]

> .*([1 2] (make '-'))
1
```

Text→AST→Nock . The simplified high-level lifecycle of Hoon code sees it go from text to an abstract syntax tree (AST) and thence to executable Nock.[5]

To convert Hoon text from a `cord` to an AST, use `+ream`. The AST is a tree of Hoon expressions marked by rune and components such as wings (lookup names in the subject) and atoms or cells.

```
> (ream '(add 1 2)')
[%cncl p=[%wing p=~[%add]] q=~[[%sand p=%ud q=1]
                               [%sand p=%ud q=2]]]

> (ream '-:!>(5)')
[%tsgl p=[%cnts p=~[[%.y p=2]] q=~]
                q=[%zpgr p=[%sand p=%ud q=5]]]
```

A Hoon AST is properly a `$hoon`, which is a complicated type union over all of the kinds of things that Hoon knows something to be able to be.[6] Sugar runes can be represented simply or reparsed by `+open:ap` into a more fundamental form. However, at this point no desugaring has taken place; equivalent forms may still have different AST representations as `$hoon`:

[5] In practice, the text is commonly derived from a file via the `%hoon` mark.

[6] The relevant code may be found in `/sys/hoon`.

```
> (ream '~[1 2 3]')
[%clsg p=~[[%sand p=%ud q=1] [%sand p=%ud q=2]
  [%sand p=%ud q=3]]]

> (ream '[1 2 3 ~]')
[%cltr p=~[[%sand p=%ud q=1] [%sand p=%ud q=2]
  [%sand p=%ud q=3] [%bust p=%null]]]
```

Hoon supplements "pure" values with metadata to establish context for the values and enforce its notion of type.

Given a reduced Hoon AST, `+mint:ut` function compiles it into Nock code. `+mint:ut` parses from a Hoon AST into a pair of the type and the Nock.[7]

```
> (~(mint ut %noun) %noun (ream '~[1 2 3]'))
[#t/[@ud @ud @ud %~] q=[%1 p=[1 2 3 0]]]
```

While there is much to say about `+mint`, for our purposes we simply note that it operates on each tag in an AST to convert it recursively to `$nock`.[8]

3 Vases

A `$vase` is a pair of `$type` and value, where `$type` defines the Hoon type set and the tail contains the actual data. (Vases could be treated more generally than simply Hoon types, but in practice are used only with Hoon, so the type data structure will become relevant.)

Vases received bare mention in the 2016 Urbit whitepaper (`~sorreg-namtyv` et al., 2016). They were 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.

[7] One simple approach to understand this part of the Hoon compiler backend is to look at the 2013 Hoon compiler, which presents a relatively uncluttered version.

[8] Nock code is not executed simply in either of the primary runtimes, Vere or NockVM (f/k/a Sword, née Ares). Instead it is converted into a bytecode for execution.

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. (`~rovyns-ricfer`, 2019)

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.

Hoon's type information captures both the set of nouns valid for a particular piece of data and the subject, or environment to which the value has access. For instance, a `%face` is a lookup index into the subject for a value with an associated name.

Vases are frequently used to compile and execute Hoon code, but are also used to store typed data. Since Nock is ultimately homoiconic, any arbitrary noun may be either executable code or static data. A vase preserves the type information associated with the noun, making it possible to reconstruct it in regular (static) Hoon. For instance, as a security mechanism, "live code" is never transmitted over the wire between machines. Instead, the type and value are sent, and the recipient compiles the noun into executable code.

What vases permit over raw Nock manipulations therefore includes:

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)

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.

3.1 Hoon Type

The formal definitions of `$vase` and `$type` are as follows:

```
+$  vase  [p=type q=*]
+$  type  $~  %noun
          $@  $?  %noun
                  %void
              ==
          $%  [%atom p=term q=(unit @)]
              [%cell p=type q=type]
              [%core p=type q=coil]
              [%face p=$@(term tune) q=type]
              [%fork p=(set type)]
              [%hint p=(pair type note) q=type]
              [%hold p=type q=hoon]
          ==
```

- `%noun` is the superset of all nouns.
- `%void` is the empty noun, but won't occur in results (i.e. a compiler affordance).
- `%atom` spans the set of all atoms.
- `%cell` contains ordered pairs.

The other types are more complex:

- `%core` is the descriptor type for a core. Besides the `$type`, it uses a `$coil`, which is a tuple of variance information, context, and chapters (limbs).
- `%face` spans the same set as nouns but includes a face.
- `%fork` is a union, or choice over options.
- `%hint` is an annotation for the compiler (Nock opcode `11`).
- `%hold` types are lazily evaluated, such as a recursive type (like the `+list` mold builder).
  - `%hold` types are why the compiler can have trouble with lists at runtime, such as needing to distinguish a `lest` or the TMI problem with `+snag` &c.
  - A `%hold` type is a "finite subtype" of an infinite type. Hoon doesn't actually know about these directly, just in that it can be lazy about evaluating recursions.
  - They result from arms in cores because the `hoon` of the arm is played against the core type as the subject type to get the result. This permits polymorphism, since the core can be modified to have a sample of a different type.

> One can "evaluate" a hold by asking the compiler to "play" the hoon against the subject type, meaning to infer what type of value would result from running that hoon against a value of the subject type. For a recursive type, this result type refers to the same hold, usually in one or more of the cases of a `%fork`. (`~rovnys-ricfer`, personal communication)

For instance, let's evaluate a value of the type `(list @ud)` using Hoon:

```
> =a `(list @ud)`~[1 2 3]

> !>(a)
[#t/it(@ud) q=[1 2 3 0]]

> -<:!>(a)
%hold

> -<:!>(?~(a 0 a))
%fork

> ->-:!>(?~(a 0 a))
#t/@ud
[%atom p=%ud q=~]

> ->+<:!>(?~(a 0 a))
l=[#t/[i=@ud t=it(@ud)] l=~ r=~]
```

(As an aside, while `+vase` is a reasonable name for a thing and the thing that shapes it, the original term was `+vise`, which also works in this capacity.)

3.2 Producing Vases

While vases can be constructed manually, Hoon provides several affordances for producing vases that are less error-prone and more convenient.

The most common way to produce a vase is to use the `!>` zapmic rune. This takes a noun and produces a vase of the raw value and its type.

```
> !>(~[1 2 3])
[#t/[@ud @ud @ud %~] q=[1 2 3 0]]
```

Note that the type is very literal: in this case, it is a three-tuple rather than a list. Hoon's type inference is quite literalistic, and this commonly leads to the "too much information" (TMI) problem, in which the type is too specific to be useful.

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 -:!>(*(list))) %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 and devase a value, which we will write Lisp-style. We also want to be able to combine vases by "slopping" them together. Finally, we need to "slap" an expression we wish to evaluate against a vase. 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 employ two main tools in Hoon to envase (or make dynamic) and devase (or make static):

- `!>` zapgar envases, producing a vase from a noun.
- `!<` zapgal devases, producing a noun from a vase.

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]
```

3.3 Eliminating Vases

`!<` zapgal devases, which requires you to know what type you expect so that you can match them. This 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 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.

(A quick review of the Urbit kernel code base suggests that most instances of `!<` zapgal 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.)

4 Vase Algebra[9]

The essential operators on vases are `+slop` and `+slap`. Essentially all other operations can be expressed in terms of these two, but many supplemental operators are supplied with the Hoon compiler for convenience.

[9] This section benefitted from discussions with `~rovnys-ricfer`, to whom I am indebted.

4.1 `+slop`

If one queries 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 one intends 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 rather tedious and error-prone in general. To combine them dynamically one can use the first tool of our vase algebra, `+slop`. `+slop` combines two vases, which means combining their types and their values consistently.

Here, `+slop` produces 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]]
```

`+slop` simply combines heads and tails using a `%cell` head tag:

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

4.2 `+slap`

Chief among our list of reasons to use vases was the ability to dynamically evaluate code at runtime. How do we actually write code to `+slap` a given Hoon expression against a subject as vase? The `+slap` operator is the second part of the vase algebra. It takes a Hoon expression and a vase, and produces a new vase of the computational result.

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

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, we will henceforth `+slop` in the Hoon subject including its standard library as `.` dot. 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 to the head of the subject because of the binary tree search order.)

4.3 Supplemental Operators

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.

1. `+slam` is an affordance for easy execution of a Hoon gate or function. Given a gate 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]]
```

2. `+slym` is an untyped version of `+slam`. The type of the sample (argument) is ignored, and the type of the resulting vase is determined by the gate alone. This is particularly useful with the "wet" gate system, which is a paradigm of gate-building gates used frequently in Hoon. 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.

   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.)

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.)

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

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

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

4. `+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]"
```

5. `+slew` retrieves the axis of a name in a vase.

6. `+swat` doubly defers a `+slap` operation for certain cases.

5 Applications of Vase Mode

5.1 Type Manipulation

Types can be manually constructed as necessary. For example, to manually reconstruct a `list` from a `lest` (or non-null `list`), we can use the `+slop` operator to combine a vase of the null terminator or empty list with the `lest`. (This is, of course, equivalent to upcasting back to a `list`, but the type system

doesn't quite notice this directly as `it(@)`.) Because the collapse from dynamic vase mode to static Hoon code necessarily destroys some type information via the mold in `!<` zapgal, variants of this code work.

```
> =/  p  (gulf 1 5)
  !>(p)
[#t/it(@) q=[1 2 3 4 5 0]]

> =/  p  (gulf 1 5)
  ?~  p  !!
  !>(p)
[#t/[i=@ t=it(@)] q=[1 2 3 4 5 0]]

> =/  p  (gulf 1 5)
  ?~  p  !!
  (slop !>(~) !>(p))
[#t/[%~ i=@ t=it(@)] q=[0 1 2 3 4 5 0]]

> =/  p  (gulf 1 5)
  ?~  p  !!
  !<((lest @) (slop !>(~) !>(p)))
[i=0 t=~[1 2 3 4 5]]

> =/  p  (gulf 1 5)
  ?~  p  !!
  !<((list @) (slop !>(~) !>(p)))
~[0 1 2 3 4 5]
```

5.2 Code Evaluation

As a practical matter and a safety measure, executable Hoon code is not directly transmitted over the wire. Instead, the type and value are sent, and the recipient compiles the noun into executable code. This is a security measure to prevent arbitrary code execution.

Thus vases are used in the kernel and in userspace for certain kinds of transmitted code and data operations, including pokes between user agents. In fact, `~rovnys-ricfer` has produced a minimal working example of a userspace handler using

vase mode.[10]

5.3 Metacircular Interpretation

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. By providing a `$roof` or reference overlay to the fake Nock 12 opcode, for instance, a Nock runtime can supply a referentially transparent expedient even when the perceptual subject is much more constrained.

5.4 Persistent Code Artifacts

Urbit's Arvo kernel and Vere runtime, as well as the NockApp framework with the Sword runtime, aggressively cache vases to expedite code execution. Vases are stored in a map from their hash to the vase itself. This is a performance optimization, since the type information is expensive to compute and the value is expensive to reduce. For instance, Arvo's `$worm` cache maps Hoon vases to pairs of type and Nock nouns.

5.5 System Upgrades

A solid-state Nock runtime such as Arvo consists of an event loop over a list of input events to apply to a current state. System upgrades such as kernel updates are supplied and built using vases.

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.)

[10] See his video of ~2022.5.10; at the time of press, a copy is available at `https://drive.google.com/file/d/10SaE5doCfdeqc2j945t8GvGvexKIBBZq/view`.

The vane interface is normally strictly typed, but using a metavase it can punch a hole through the `type` system. (Before `!<` zapgal was introduced, vase reduction was mediated by a vane passing 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.

A Nock runtime can cache known vases aggressively for performance improvement.

6 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.[11] While Hoon still uses vase mode explicitly to import libraries and deliver user agent pokes, these are the only places that a userspace developer would typically encounter them. The vase algebra has largely been restricted to the compiler level in contemporary practice.

6.1 Comparisons

Pairing type and data is a general and generalizable approach. Languages and compilers must, after all, track typed values in some way. How do other languages and systems handle this?

[11] To wit, the history of the Ford vane, ca. 2016–2019.

6.1.1 Python/CPython

For instance, in the CPython reference implementation all values are "boxed", or `PyObject` structures. This pairing of metadata and value is similar to the way that Hoon vases work, but it is not directly exposed as a first-class element of the Python user environment, nor is there an explicit vase algebra (or subject).

6.1.2 Assembly Language

Typed assembly languages such as TAL (Morrisett et al., 1999) were explored in the late 1990s and early 2000s, with the goal of providing a type-safe assembly language that could be used to write low-level code while still preserving type safety. These researchers recognized that traditional compilation procedures often destroyed type information when it was still useful for verification and optimization. Even dependently typed assembly languages have been explored (Xi and Harper, 2001), conceived as a way of improving bytecode or assembly level performance by permitting the compiler more optimizations than it would naively know enough to implement. A set of rules was defined in order to produce well-formed "type states", or relationships of type index expressions to actual register contents. This approach offers the expedient of a certifying compiler designed to guarantee type safety. While not constituting a vase algebra, the process of retaining type to produce better target ISA code shares a goal with Hoon's vase mode.

6.1.3 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 or S-expression:

```
(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`, although the Hoon AST is somewhat more regularized and opaque.

```
> (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.)

6.1.4 Haskell `Dynamic`

Haskellers probably immediately think of `Dynamic`, which is used for injecting values into a dynamically typed value. `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. Whereas in Hoon, a type–value mismatch is an "evil vase" and results in a runtime crash,[12] `Dynamic` will result in `Nothing` if the types don't match.

[12]Since we are commonly running in an emulated mode, this doesn't take down the whole kernel.

Hoon's vases are in a sense much closer to the metal since they can trivially result in Nock code. Vase mode feels more like a dynamic interpreter or a VM, while `Dynamic` is more focused on typed payloads for data evaluation.

6.1.5 Untyped Virtual Machines

There may even be reason to eschew vases altogether in some cases; Jock does not currently use vases except when interacting with Hoon via a foreign function interface. Chevalier-Boisvert (2022) argued that untyped virtual machines are more resistant to bit rot than typed virtual machines:

> An untyped VM design ... has to enforce a small set of hard constraints, such as making sure that pointer dereferences respect valid address bounds so that the running program can't crash the host VM, but it doesn't really need to care about the types of values or enforcing typing constraints. For performance, it can implement a very barebones JIT compiler based on dynamic binary translation, but it doesn't have to care about optimizing type checks, and it can even allow the running program to manage its own memory and implement its own garbage collector.
>
> ... [T]here's a strong case to be made that an untyped VM design can easily be much smaller and more minimalistic than a typed VM design. A simpler untyped VM has two major strengths. The first is that it doesn't place as many restrictions on running programs. Programs can implement their own control flow structures, their own GC, or even their own JIT. The second is that a smaller, simpler VM is much easier to port, reimplement and maintain. (Chevalier-Boisvert, 2022)

Nock runtimes do indeed hew to this parsimony, but vases are used particularly in interprocess communication in today's Urbit kernel and userspace. It is unclear how they could profitably be excised from the system given their global utility.

A final interesting question is to think about targeting other ISAs. While Hoon itself is closely tied to Nock, no part of the vase algebra (in the abstract) is specific to Nock except its instantiation as nouns.

7 Conclusion

Vase mode provides something like a gradual (un)typing pipeline: one proceeds from a static Hoon context, to a dynamic vase mode, to untyped Nock. 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 execution, code proving, and optimized execution.

References

Chevalier-Boisvert, Maxime (2022) "Typed vs. Untyped Virtual Machines". URL: https://pointersgonewild.com/2022/06/08/typed-vs-untyped-virtual-machines/ (visited on ~2024.7.4).

Morrisett, Greg et al. (1999). "TALx86: A Realistic Typed Assembly Language." In: *Proceedings of the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software*. ACM, pp. 25–35. URL: https://www.cs.cornell.edu/talc/papers/talx86-wcsss.pdf (visited on ~2024.7.4).

~rovyns-ricfer, Ted Blackman (2019) "Why Hoon?" URL: https://urbit.org/blog/why-hoon (visited on ~2025.8.13).

~sorreg-namtyv, Curtis Yarvin et al. (2016). *Urbit: A Solid-State Interpreter*. Whitepaper. Tlon Corporation. URL: https://media.urbit.org/whitepaper.pdf (visited on ~2024.1.25).

Xi, Hongwei and Robert Harper (2001). "A Dependently Typed Assembly Language." In: *Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*. ACM, pp. 169–180.