

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18

The Stakes of Scrying: `. ^` dotket and the Seer Proposal

Ted Blackman `~rovvnys-ricfer`
Paul Driver `~fodwyt-ragful`
Urbit Foundation

Nock 12 and the `. ^` dotket rune break the referential transparency of Nock, with consequences for the purity of Urbit’s functional programming model. We consider the implications of this, and propose a move-based replacement for the `. ^` dotket rune that would restore referential transparency to Nock.

Contents

1	Introduction	2
2	How dotket Works	2
3	Recommendations	7
3.1	The Seer Monad	8
3.2	Move-Based Scry Interface	10
4	Conclusion	10

1 Introduction

While Nock formally has twelve opcodes, numbered 0–11, it is convenient to expose in userspace a fake Nock 12 opcode to expose the scry namespace to userspace developers. This opcode is trivially invoked by using the Hoon `.^ dotket` rune. However, Nock 12 breaks the referential transparency of Nock, and this has consequences for the purity of Urbit’s functional programming model.

2 How `.^ dotket` Works

The Nock 12 opcode and the `.^ dotket` rune are used by userspace code to ask the kernel for data in the scry namespace.¹ When Urbit runs userspace code, that code is not raw Nock. It is instead called Mock, which is Nock with two additions: deterministic errors are caught and returned as values to the kernel; and there is an extra Nock opcode, opcode 12. A simple Nock 12 formula looks like `[12 [1 /foo/bar/baz]]`, where `/foo/bar/baz` is a path literal `[%foo %bar %baz ~]`, representing a path in the scry namespace. The result of running this opcode will be the result of performing the scry request, i.e. dereferencing the path.

Catching deterministic errors is uncontroversial, but the Nock 12 opcode poses some difficulties and has been under scrutiny for years now. The most tangible problem with Nock 12 is that it prevents userspace code from persistently memoizing Nock computations; at the moment, only kernelspace code can hold onto Nock memoization caches across multiple Arvo events. The lifetime of a userspace memoization cache is limited to a single metacircular evaluation context, i.e. a single call to `++mink` or an equivalent gate that runs Mock (virtual Nock with the extra opcode).

¹The scry namespace, in brief, is a collection of immutable values “bound” by convention at fixed endpoints. The scry namespace is defined by a unique path for each resource. A representative scry path would have the entries `/host/rift/life/vane/request-type/revision-number/desk/file-path/mark`.

Userspace code cannot memoize Nock persistently because it is not feasible for the runtime to prevent a jet mismatch in the metacircular evaluator gate. A gate like `++mink` takes in a Mock expression (a cell of `[subject formula]`) and a “scry handler gate” (which is described shortly), and returns a value which is one of:

- `[0 computation-result]` for a successful evaluation;
- `[1 block]` in the case of a failed scry request; or
- `[2 error]` in the case of a deterministic error.

While error handling is not relevant to the current question, a brief description is included here for thoroughness. The `error` in `++mink`’s result `[2 error]` is a deterministic error, meaning that every attempt to evaluate that `[subject formula]` pair will always result in that same error value—deterministic errors correspond to the lines in the Nock spec where `*a` evaluates to `*a`, i.e. a trivial infinite loop. Nondeterministic errors, such as running out of usable memory or getting interrupted by the user, result in the function failing to complete, and the Arvo event will be aborted and rolled back to its state before trying to run this new event.

As mentioned, the `[subject formula]` pair is not the only input to `++mink`. There is also an argument of the form `scry=$-(unit (unit))`, called the scry handler gate. When `++mink` encounters a Nock 12 opcode while evaluating the `[subject formula]` expression, it evaluates the sub-formula to obtain a path to look up in the scry namespace. Once it has this path, it calls the scry handler gate on the path, which returns one of:

- `[~ ~ value]` positive scry response;
- `[~ ~]` permanent scry failure; or
- `~ “block”`.

The positive response case means that the scry was performed successfully and yielded a value, in which case `++mink` injects the value as the result of the Nock 12 opcode and continues

to evaluate the Nock. The permanent failure means the scry namespace knows there will never be a value at that path, so ++mink can immediately return a deterministic error.

A scry block means the system refused to answer the question, i.e. the request was “blocked” for some reason—this could be due to lack of permission, or because the request asked about the future, or the request asked for data from another ship that isn’t synchronously available within this Arvo event. If the scry blocks, ++mink returns a [1 block], where block is the scry request path whose value could not be resolved.

The Nock 12 opcode is problematic because it renders the result of the computation dependent on external data which are not in the [subject formula] that is being evaluated. The result is no longer a pure function of the subject and formula: depending on what the scry handler gate is, the same [subject formula] could yield different results. Consider [subject formula] of [0 [12 1 /foo/bar]]. This returns the result of looking up the /foo/bar scry request path from the scry gate. Userspace code (which is untrusted) could call (mink [[0 12 1 /foo/bar]] |=(^ [~ ~ 33])), acquire the result 33, then call (mink [[0 12 1 /foo/bar]] |=(^ [~ ~ 44])) to yield 44. The subjects and formulas were identical, but because they were evaluated with different scry gates, they yielded different results.

Given that Urbit is supposed to be a pure-functional system, how can this be acceptable? The answer is that Mock, virtual Nock, is purely functional (mathematically, a “partial function” like Nock itself) as long as it’s only ever run using the real scry namespace, which is referentially transparent. That is to say, all data in the scry namespace can be thought of as something like axiomatic data that everyone agrees on and never changes: all virtual Nock computations will get the same answers to all Nock 12 opcodes that complete, because everyone agrees on which value is bound to each path in the namespace.

Another way to think of the scry namespace is as comprising an extra implicit argument passed to every Mock computation along with the subject and formula. This extra argument is discovered dynamically over time as more ships bind more values. Since any path can only be bound once, it becomes im-

mutable; all Mock expressions that dereference some scry path will either fail to complete or complete with identical results, hence pure functional.

Note that there’s no requirement that all the scry gates be noun-equivalent (intensionally equal), or even contain the same set of namespace paths and values (extensionally equal). The requirement is that no two scry gates ever return different values for any given scry request path.²

The Arvo kernel’s scry handler gate, as described in the original Moron Lab blog post about Urbit (Yarvin ~ sorreg-namtyv, 2010), is a “static functional namespace”, so even though that gate changes frequently, it maintains the invariant that every scry path’s binding to a value is immutable.³

As long as a correct Arvo kernel is supplying the scry handler, Mock retains its purely functional nature. But consider again the case where userspace code runs the same `[subject formula]` twice, but with two different scry gates that give different answers to some scry path. From the perspective of a simple Nock interpreter, it’s still deterministic: `++mink` is just some Nock function that runs some other computation (Mock) on some inputs. Because it’s just some Nock function, it can’t break determinism of the Arvo event; replaying the Arvo event will still yield the same result. The problem only arises when trying to get clever about how the runtime evaluates Nock, in particular by caching the results of Nock computations.

Development versions of Vere support persistent Nock memoization. This means a programmer can emit a Nock hint to the interpreter asking the interpreter to cache the result of the computation, so that if it gets run again, the result will be retrieved from the runtime’s cache instead of rerun step by step. The cache key is a `[subject formula]` pair. When the interpreter sees a `%memo` hint, it looks at the `[subject formula]` that the hint surrounds. If the memoization cache has a value for that key, the runtime uses that value as the result of the

²A scry gate that always blocks is trivially legitimate, since it never gives any answers, much less answers that differ from those of another scry gate.

³Up to some known flaws that are addressed by UIP-0116: Arvo Ticks (Monk ~ wicdev-wisryt and Blackman ~ rovnys-ricfer, 2023a,b; Wilson ~ midden-fabler, 2024).

computation. Otherwise the runtime runs the computation, inserts the result into the cache, and returns the result.

For the moment, only the kernel can place items into the cache. Userspace code runs Mock, so if it were allowed to place items into the cache, there would be three options:

1. Let userspace code place arbitrary values into the cache. This poses a severe security problem: event replay would be nondeterministic, influenced by the contents of the memoization cache, allowing malicious userspace agents to perform a kind of denial of service attack by preventing event replay, or possibly even worse, by injecting cache lines that other applications would use, causing those other applications to malfunction in a way the malicious code wants. The worst case would be causing a system app such as %hood to fail to enforce a security check, allowing the malicious code to escalate its own privileges and permitting it to ask the system app to do something like reinstall the kernel.
2. Place the scry gate into the cache key for the memoization cache, so a cache key would be `[[subject formula] scry-gate]` instead of just `[subject formula]`. This would close the security hole, but since Arvo's scry gate undergoes small changes on between every agent activation, it would make these cache lines useless outside of the immediate execution context within a single `++mink` call. This is exactly the situation userspace code is already in, and thus not helpful at all.
3. Have the interpreter record the set of scry `[path value]` pairs looked up by the computation. This would allow userspace code to perform scry requests safely, but the cost is a nontrivial amount of machinery in the runtime, along with a computational cost of looking up a cache line that grows linearly with the number of scry paths that were requested. Of the three options listed here, this is the most reasonable, but it is still highly nontrivial and the requirement to compare all requested paths could be onerous.

192 Instead of implementing any of these options, Urbit has cur-
193 rently simply disabled the ability for userspace code to modify
194 the Nock memoization cache. This is enforced by the jet for
195 `+ + m i n k`; the jet is simply a call back into the normal Nock inter-
196 preter, which is really a Mock interpreter, whose state includes
197 a list of nouns representing the stack of scry handler gates for
198 the current execution context. If this list is null `~`, then we must
199 be in the kernel and the code is allowed to modify the memo-
200 ization cache; otherwise, we are in some kind of Mock context,
201 and the code is not allowed to modify the cache.

202 The concerns with `. ^ dotket` go deeper than memoization
203 however, despite that being the only known major concrete is-
204 sue.⁴ Conceptually, the fact that a Mock computation has im-
205 plicit external dependencies makes the pure-functional nature
206 of the system more fragile—the memoization issue is down-
207 stream of this fragility.

208 Morally and aesthetically, the fact that Nock makes no as-
209 sumptions about any of its contents is part of the magic of
210 Urbit. Nock is a true functional programming language with
211 no implicit external dependencies at all, and no constraints on
212 the nouns it operates on. Anything that pollutes this vision is
213 potentially harmful to the environment Urbit creates for pro-
214 grams and their authors. Mock’s implicit dependency on the
215 namespace makes Nock context-dependent instead of context-
216 independent, as it should be. A related issue is that userspace
217 code does not have access to raw Nock, only Mock. The Hoon
218 compiler willingly outputs Nock 12 instructions when handed
219 a `. ^ dotket` rune, and there is no means for userspace code to
220 signal that it will refrain from using Nock 12.

221 3 Recommendations

222 The removal of Nock 12 is fraught with difficulty for userspace
223 and kernel development. The core development team has con-
224 sidered two proposals which will allow the kernel to gracefully
225 deprecate Nock 12. The two concrete lines of action are:

⁴Some complexity is introduced in the runtime when jetting the metacir-
cular interpreter.

1. Add a new metacircular evaluator gate that does not run Nock 12 expressions (the Seer proposal).

2. Add a move-based interface for scrying to the Gall interface.

This approach will allow core developers and userspace application developers to experiment with code that does not use `.^ dotket`. If that resulting code and associated design patterns work well, we will deprecate the `.^ dotket` rune and the Nock 12 opcode in favor of move-based scrying.

3.1 The Seer Monad

Seer is a monadic scry interface which allows application developers to read from the namespace without using `.^ dotket` or its virtual Nock operator 12. Userspace code run inside of this gate can reestablish the guarantee that it is purely functional, allowing it to modify the memoization cache.⁵

Seer is designed as a replacement for `.^ dotket`, and thus attempts to be as ergonomic as the `.^ dotket` pattern. Consider a Gall agent that scries using the current `.^ dotket` pattern:

```
++ on-poke
| = [m=mark v=vase]
^- (quip card _this)
= / foo-result .^(@da /cx/(scot %da now)/foo)
:: do something with foo-result...
[~ this]
```

A corresponding snippet of a Gall agent that scries using `++seer` in the continuation style would look like this:

```
++ gear      :: gall seer
| * a=mold
| (seer vase a)
++ on-poke
```

⁵Note that it's still acceptable for that gate to be called from Mock code, or even nested Mock code, that had previously performed scry requests that were not referentially transparent: raw Nock is context-free, so no matter how the [subject formula] cell was created, the result of its evaluation is still pure.


```

259 | = [m=mark v=vase]
260 ^- (gear (quip card _this)) :: (seer vase (quip card _this))
261 :+ %scry /cx/(now)/foo | = foo=vase
262 =/ foo-result !<(@ foo)
263 :: do something with foo-result...
264 [%done ~ this]
265

```

++seer values can be composed using a monadic bind:

```

267
268 ++ foo ^- (seer vase @) !! :: /foo
269 ++ bar ^- (seer vase @) !! :: /bar
270 ++ on-poke
271 | = [m=mark v=vase]
272 =* r (quip card _this)
273 ^- (gear r)
274 =/ bind (rapt vase r)
275 ;< foo=@ bind foo
276 ;< bar=@ bind bar
277 [%done ~ this(some-state (add foo bar))]]
278

```

(In this example, use is made of ++rapt. ++rapt is a monadic scry binding gate (Levan ~ mastyr-bottec, 2023a).)

Here, a ++seer is the type of programs that scry:

```

282
283 ++ seer
284 |* [r=mold a=mold]
285 $~ [%done *a]
286 $% [%done p=a]
287 [%scry p=path k=$(r (seer r a))]
288 ==
289

```

The ++seer scry system is intentionally incompatible with the current .^ dotket scry system. We believe that ++seer degrades ergonomics only slightly. In the initial introduction, we shall provide a wrapper library which accepts an agent written in the ++seer style and produces an agent that scries in the .^ dotket style.

The current reference implementation of Seer is available at Levan ~ mastyr-bottec (2023b). A more complete example is available at Levan ~ mastyr-bottec (2023c).

3.2 Move-Based Scry Interface

A Gall agent will be able to emit a new kind of move, in addition to all the moves it can emit now, that asks Gall to perform a scry request on the agent's behalf. When Gall processes this move, it will perform the scry request and convert the result of the scry into a response move that will be delivered back into the agent in a later activation of the Gall vane, in accordance with the normal move processing order that Gall and Arvo use. Thus an agent could emit a list of moves such as `[poke-1 scry-A poke-2 scry-B poke-3 ~]` and Gall will handle all of those moves in the same order as if they were all pokes, i.e. the kernel will not introduce any exceptions to its normal move processing order for scry request moves.

This proposal is less developed than the Seer proposal, and thus no code examples can be provided, and no reference implementation exists at the time of writing.

4 Conclusion

There is a general consensus among Urbit core developers that a new move-based scry interface is better suited as the lowest layer of the scry system than the Seer monadic interface. The fact that Seer or `.^ dotket`⁶ could be implemented on top of moves (but not necessarily the other way around) militates in favor of moves at the lowest layer. Another benefit of this arrangement is that by making all scrys asynchronous, the same interface can be used for both local and remote scrys (since remote scrys are always asynchronous). Such an abstraction over local vs. remote scry calls removes what would otherwise be an unavoidable branch point in userspace I/O code. It also matches the rest of Gall's interprocess communication model, which does not distinguish between local and remote. A strong argument can be made that this is also the most "grug-brained" option, which is not to be undervalued.

⁶Or at least an opt-in form of Mock, rather than it being the global default as it is now.

There remain concerns about application development patterns if `. ^ dotket` is removed. Losing synchronous read access to other parts of the system potentially reduces the ergonomic wins of a naturally growing transaction across applications. This can be implemented in a way that is almost synchronous for local scrys, and in particular it is still possible to perform multiple scry requests at the same system snapshot, meaning the requesting agent will see a consistent view of the system without any tearing.

Gall will also need to run the `++on-peek` arm of a Gall agent in the Seer monad, i.e. that arm will return either `[%done scry-result]` to indicate it's done or `[%scry scry-request-path callback]` to ask Gall to perform a scry request, in which case Gall performs the scry, injects the result into the callback gate, and repeats, until a `%done` is returned. This should not be a direct problem, but betokens complexity creeping into the Gall interface.

Because of these concerns with the proposed move-based replacement for `. ^ dotket`, we will first add it as a new feature alongside `. ^ dotket` before committing to removing `. ^ dotket`. In this way, if the application patterns end up suffering inordinately, `. ^ dotket` may be retained. That being said, the current primary plan is to deprecate `. ^ dotket` once move-based scrying has been verified as providing a sufficiently good application model. This is a conservative, incremental approach that will not break any existing code and lets us try out the new style in production applications before committing to the removal of Nock 12.

References

- Levan ~ mastyr-bottec, Matthew (2023a). `/lib/seer`. GitHub repository.
- (2023b). *Seer: A Monadic Scry Interface*. Pull request.
- (2023c). *Seer: A Monadic Scry Interface Examples*. Gist.
- Monk ~ wicdev-wisryt, Philip C. and Ted Blackman ~ rovnys-ricfer (2023a). *UIP-0115: Breadth-First Move Processing in the Arvo Kernel*. Urbit Improvement Proposal.

367 Monk ~ wicdev-wisryt, Philip C. and Ted Blackman ~ rovnys-
368 ricfer (2023b). *UIP-0116: Arvo Ticks*. Urbit Improvement
369 Proposal.
370 Wilson ~ midden-fabler, Scott (2024). *Arvo: Breadth-First Move*
371 *Order #6775*. Pull request.
372 Yarvin ~ sorreg-namtyv, C. Guy (2010). *Urbit: Functional Pro-*
373 *gramming from Scratch*. Blog post. Moron Lab.