Shrubbery: the greenpaper

Liam Fitzgerald ~hastuc-dibtux
Urbit Foundation

Abstract

XX TODO: write abstract In this groundbreaking paper, we embark on a cosmic journey with the renowned character Ernest P. Worrell as he ventures into the unexplored realm of Martian computing. Drawing inspiration from Ernest's comically ingenious encounters with everyday challenges, we investigate the foundations of what we term "Artificial Stupidity." As Ernest grapples with Martian technology, we delve into the intricacies of programming errors, algorithmic missteps, and the curious phenomena that arise when human-like intelligence meets extraterrestrial computing systems. Our analysis sheds light on the unexpected intersections between humor, artificial intelligence, and the cosmic absurdity of Martian software. Join us in this interplanetary exploration as we unravel the mysteries of Artificial Stupidity through the lens of Ernest's interstellar escapades.

Contents

Chapter 1

Overview

This article uses 'shrubbery' to refer to the general notion of namespace-maximalist application environments, and 'neo' to refer to the specific implementation by the author that lives as an agent.

There has been a wealth of Introduce the scope of your article and investigation.

Chapter 2

Namespace Maximalism

2.1 Background and Literature

The idea of 'namespace maximalism' has precedent in a number of non-urbit contexts.

2.2 Internal Discussion

The original 'Tenets of Namespace Maximalism' (`~rovnys-ricfer`, 2021) screed is the first attempt to categorically define what 'namespace maximalism' means in an Urbit context. Tenets 0-2 are uncontroversial and are an explication of the internal thinking about the namespace. Tenet 3 appears to contradict the entire idea of 'namespace maximalism', wanting to banish compiled code from the namespace. However, at the time of its writing, Urbit engineering culture had never really grappled in-depth with the notion of overlay namespace, and so the entire idea of "virtual" namespaces was not considered. Tenet 4 was a leap at the time, but is no longer a point of contention.

The shrubbery models considered in the old shrubbery group (,), have some interesting characteristics are worth highlighting. Pokes existing in the namespace naturally led to the consensus on shrub implementations being heavily diff-based.

2.3 Theory of the Path

2.3.1 Axioms

Given an arbitrary path *foo* in a namespace, we can derive a set *Ancestors* which is simply the set of paths who are prefixes of *foo*. Putting this to the side for now, we can also derive a set *Children* which is the set of paths for which *foo* is a prefix.

Axiom 0: Constraint A metabinding is code that is designed to constrain the shape of a binding in the namespace. A metabinding constrains both the shape of the noun that the path is bound to, but also the shape of the metabindings permissible underneath it.

Axiom 1: Relocatablity A metabinding has a contract with its children. It looks and understands what is beneath its path, but does not (in general) assert anything other than what it is told about ancestor and cousin nodes. This provides arbitrary relocatiblity for metabindings, i.e. they may be installed at any point in the namespace.

Axiom 2: Transactionality The conclusion of Axioms 0 and 1 is that a metabinding, when installed at a point in the namespace, defines a ”zone of consistency” or a ”transactionality boundary”. Which is to say, the metabinding is responsible for defining an external interface with which it can be programmed.

Axiom 3: Separation We now need to separate the noun from its children. Consider the dataflow graph of the conceptual object bound in the namespace cone. Any mutual recursion should be hoisted to the noun, any variable sized datastructure should be put in the children, and everything else organised in order to minimise the number of dataflow dependencies between siblings or uncles. Anything that possibly needs to be versioned seperately should also be put in the children.

Axiom 4: Identification When creating children, the path at which they live is analogous to a primary key, and should be treated as such. Do not use the namespace to provide sorting if that may cause path collisions.

Axiom 5: Canonicity For any given datum, there are an arbitrary number of ways to query and store it. A well-behaved namespace should store this datum once, in a canonical type, The canonical type is whatever representation minimises transactionality loss

Axiom 6: Virtuality A virtual binding is code that transforms some part of the namespace according to a pure function

$$f(name, namespace)$$

Virtual bindings are used to restore the infinite multiplicity of data representation, dynamically reprojecting the datum as query patterns demand. Virtual bindings can be used to replicate querying facilities like those given in any standard DBMS. For instance, sort keys should be implemented in this way (if they are not also the primary key (path)).

2.3.2 Commentary

Chapter 3

Types

3.1 Background and Literature

3.2 Internal Thinking

3.3 Neo type model

Neo is a series of bindings from names to noun and rules for having those bindings evolve. However, we do not want to have to rewrite code for each kind of noun that we wish to store. Hence we introduce the notion of a protocol. A protocol is a kind of noun that constrains a noun. It is simply a type that exists in the namespace. We give these types special names, that map to a regular name in the namespace. The outputs of the build system must be special cased, in order to allow for bootstrapping.

```
$
|%
::   $stud: Identifier for a build system output
+$   stud
  $@   term
   (pair mark (pair ship desk))
::   $tack: Type of code being distributed
```

```
::
::      %con: CONverter of protocols
::      %imp: IMPlemenation of shrub
::      %pro: PROtocol (type)
::
+$   tack
  ?(%con %imp %pro %rol)
::   $post: Name of code being distributed
::
+$   post   (pair tack stud)
--
```

3.3.1 Conversions

For any given protocol, there may be other protocols that are semantically equivalent. This is a 'conversion', a stateless transform from one protocol to another. This is the %con case of $tack. Data loss is acceptable, but if so, the reverse of the transform should not be defined i.e. if you have to leave a field blank, then transform is not valid.

Polymorphism Implementing these conversions allows for a limited kind of polymorphism, akin to rust's traits. Consider, for instance, a $message protocol that contained the barest possible information (pair ship time). This is not particularly useful, because it does not have any contents field, but it does provide the necessary items that a chat needs from it's message children. So, if a chat declares it's children to be of protocol %message, then we allow polymorphism over the children, such that the actual protocol can be anything that has a conversion defined to %message. When constructing the context for a shrub, we include the children, but if they are polymorphised, then we run them through the conversion to monomorphise them before the shrub receives it. NB: we only polymorphise over state, not pokes.

3.3.2 Roles

A $role is just another piece of "code", but this time in English. It corresponds to the %rol case of $tack. It contains an English language description of the semantic quality of a datatype. For instance, some easy roles to think about are %sender and %receiver. We now allow a conversion to decorate itself with an optional $role. If a role is present in a conversion, then we no longer require that the conversion is semantically equivalent. The purpose of the role system is to allow querying by semantic quality, regardless of the underlying datatype, so long as the shrubs being queried understand the role.

```
:: A conversion of role %currency-receiver is valid if
:: A conversion of role %currency-receiver should prod
```

The role system allows for projecting types in a way that combines the best of aspect-oriented programming with EAV-style databases. For instance, the query 'show me every transaction where I received money' can be trivially constructed, even if there are a mixture of cryptocurrency and fiat wallets inside the namespace.

Chapter 4

Application Model

4.1 Background and Literature

Application development environments can be split into two different kinds. The application environments that must be compiled into the C ABI, and those that don't. The majority of application enviroments in use are of the former type, the only exception to this is Javascript. As Urbit is, amongst other things, an attempt to break from the restrictive constraints of the C ABI, we are most interested in the latter. The most well known examples of such are:

1. Haiku
2. Smalltalk
3. The various Lisp Machine OSs
4. Emacs
5. Hypercard

Interestingly, all of these examples converged on using objects as the total organising primitive of the environment, even for those with traditionally more functional heritage (the lisps).

4.1.1 They're just names, Leon

> I'm sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea. The big idea is "messaging".
>
> Alan Kay

Regardless of the actual implementation, all OO systems provide two things. Names, and passing messages to those names. Implicit in the idea of names is the classic OOP notion of 'encapsulation'. Note that under this conception of object-orientation, Unix and Plan 9's concept of 'everything is a file' begins to look suspiciously like object-orientation. Once

The reason that these environments coalesce around the idea of an 'object'

4.2 Internal Discussion

Chapter 5

Conclusion

Exposite the relevant background and literature.

Bibliography

~rovnys-ricfer, Theodore Blackman (2021) “Tenets of Namespace Maximalism”. URL: https://github.com/urbit/UIPs/blob/main/UIPS/UIP-0118.md (visited on ~2024.1.25).

“Shrubbery Urbit Group” (~nodate..). URL: https://github.com/urbit/ares.