The following is what I call a ‘semantic design’ for Nu's scripting system (as well as an unrelated replacement for micro-services called MetaFunctions). The concept of a semantic design is inspired by Conal Elliot’s denotational design - https://www.youtube.com/watch?v=bmKYiUOEo2A. The difference is that semantic design does not connect back to an existing language such as mathematics but is instead built upon an orthogonal set of axiomatic definitions.

Whereas denotational design is a more thorough design treatment that is used in greenfield development to yield high-precision design artifacts, semantic design works well for projects that don’t satisfy any simple denotational design, such as those that are already far into their implementation.

To specify semantic designs generally, I’ve created a meta-language called ADELA (for Axiomatic Design Language). First, we present the definition of ADELA, then the semantic design for Nu and MetaFunctions in terms of ADELA.

Adela Language Definition

Axiom := Axiom**[!]** str

where ! denotes intended effectfulness
and str *is a string literal.*

Product := MyProduct<*A*, ..., *Z*> = **A |** (A **:** **A**, ..., Z **:** **Z**) **| Axiom**

where MyProduct *is the* **Product Identifier**
and A ... Z are **Type Parameters**
and A ... Z *are* **Field Identifiers**
and **A ... Z** *are* **Type Identifiers**

Sum := MySum<*A*, ..., *Z*> = | A of **A** | ... | Z of **Z**

where MySum *is the* **Sum Identifier**
and A ... Z are **Type Parameters**
and A ... Z *are* **Case Identifiers**
and **A ... Z** *are* **Type Identifiers**

Category := category MyCategory<*A*, ..., *Z*> =
| f = *A* -> ... -> *Z*
| g = *A* -> ... -> *Z*
| ...

where MyCategory *is the* **Category Identifier**
and A ... Z are **Type Parameters**
and f, g, ... *are* **Equivilence Identifiers**

Witness := witness MyCategory =
| f (_ : A) ... (_ : Z) : R = E**xpr**
| g (_ : A) ... (_ : Z) : R = E**xpr**
| ...

where MyCategory *is a* **Category Identifier**
and f, g, ... *are* **Equivilence Identifiers**
and A ... Z, R *are* **Type Identifiers**

Type Identifier := Product Identifier | Sum Identifier

Semantic := fn (a : A) ... (z : Z) : R = E**xpr**

where fn *is the* **Semantic Identifier**
and a ... z *are* **Parameter Identifiers**
and A ... Z, R *are* **Type Identifiers**

Applicn := **Example:** f a (g b)

where f *and* g *are a* **Semantic Identifiers**
and a *and* b *are* **Paremeter Identifiers**

E**xpr :=** **Applicn | Axiom**

Constraint :=a => **Category**<a> *where* a *is a* **Type Parameter**

Categorization := Explanation: *iff type* A *has a witness for category* **A***; type* A *is allowable for type parameter constrained to category* **A**

fun *a b ... z -> expr* **:=** *a* (*b* (... *z.expr*))

a **->** *b* **:=** _ = (_ : *a*) : *b*

Adela Language Prelude

```
Unit = Axiom "The empty value."

category Semigroup<A> =
    | append = A -> A -> A
```

Nu Semantic Design

```
Relation = Axiom "Indexes a simulant or event relative to the local simulant."
Address = Axiom "Indexes a global simulant or event."
Name = Axiom "Indexes a property of a simulant."

Stream<a> = Axiom "A stream of values."
eventStream<a> : Address -> Stream<a> = Axiom "Construct a stream of values from event data."
foldStream<a, b> : (b -> a -> b) -> Stream<a> -> b = Axiom "Fold over a stream."
productStream<a, b> : Stream<a> -> Stream<b> -> Stream<a, b> = Axiom "Combines two streams into a single product stream"
sumStream<a, b> : Stream<a> -> Stream<b> -> Stream<(a, b)> = Axiom "Combines two streams into a single sum stream."

get<a> : Name -> Relation -> a = Axiom "Retrieves a property of a simulant indexed by Relation."
getAsStream<a> : Name -> Relation -> Stream<a> = Axiom "Construct a stream of values from a simulant property."

set<a> : Name -> Relation -> a -> a = Axiom! "Updates a property of a simulant indexed by Relation, then returns its
value."
setToStream<a> name relation stream = foldStream (fun _ -> set<a> name relation) stream
```

Semantic Design for MetaFunctions (a replacement for micro-services - unrelated to Nu)

```
Symbol = Axiom "Symbolic type such as the one defined by Prime."
Vsync<a> = Axiom "Potentially asynchronous monad such as the one defined by Prime."

IPAddress = String // a network address
Port = Int // a network port
Endpoint = (IPAddress, Port)
Meaning = String // the intended meaning of a MetaFunction (indexes functionality from a provider)

Container = Meaning -> Symbol -> Vsync<Symbol>
Provider = Endpoint | Container
MetaFunction = Provider -> Meaning -> Symbol -> Vsync<Symbol>

makeContainer (asynchrounous : Bool) (gitUrl : String) (envDeps : Map<String, Any>) : Container = Axiom "Make a
container configured with its Vsync as asyncronous or not, built from source pulled from the givern GIT url, and
provided the given environmental dependencies."

attachDebugger (container : Container) = Axiom! "Attach debugger to code called inside the given container."
call (mfn : MetaFunction) provider meaning args : Vsync<Symbol> = mfn provider meaning args
```