Semantic Design

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.

To specify semantic designs generally, I’ve created a meta-language called SEDELA (for Semantic Design Language). First, we present the definition of SEDELA, then the semantic design for Nu’s scripting system as well MetaFunctions in terms of SEDELA. Although I may aim to write a parser and type-checker for SEDELA, there will never be a compiler or intepreter. Thus, SEDELA will have no syntax for **if** expressions and the like. The only Meanings (SEDELA’s nomenclature for functions) defined in the Prelude will be combinators such as id, const, flip, and etc. SEDELA’s primitive types are all defined in terms of Axioms (types without formal definitions) with no available operations.

To understand Sedela, it is useful to talk about its intended capabilities as well as how it contrasts with denotational design.

The first intended capability of Sedela is to allow program designers to encode the abstract structure of their program outside of its implementation language. I believe that getting correct a program’s abstract structure is the most important task of program design. Also important is finding a way to encode the program’s abstract structure in a way that is separate from - and not limited by! - the program’s implementation language.

The second intended capability of Sedela is to allow program designers to specify their program’s intended semantics in a formal (denotative) or informal (textual) way. Sedela’s syntax allows designers to encode the semantics of their program’s abstract operations with a typed lambda calculus where such encoding is deemed beneficial. More importantly, Sedela offers designers a way to elide this level of formality by specifying ‘axioms’ that directly provide an informal (textual) definition for their program’s abstract operations. This less formal approach makes Sedela a usable tool for describing systems whose denotations are too complex to warrant formal specification, in particular, legacy programs.

I currently contrast Sedela’s semantic design with Conal’s denotational design as follows -

1) Denotational design requires no specialized host language whereas semantic design requires something like the small one specified in this document.

2) Denotational design restricts its domain of use to programs / subprograms whose semantics can be specified denotatively (EG, formally and in full). This is an advantage for those working on greenfield projects and projects that demand formal definition anyway (such as with programming languages).

3) Semantic design provides a ‘knob’ for the level of semantic detail at which program designers would like to specify their designs. It has been found to be useful to increase the level of semantic detail for designs by replacing some informal definitions with ‘more’ denotative ones (definitions built upon other defintion, be they formal or informal). Semantic design may in general be a useful bridge from a low-detail design to a design that can (and should be) specified denotatively.

While denotative design seems ideal, I invented semantic design for either one of two reasons – a) I could not apply denotational design to my current needs due to its limited domain of application, or b) I did not understand denotative design well enough to realize its domain of application was big enough to in fact satisfy my needs. Denotational design is admittedly still a mystery to me in some ways, so while I am confident in Sedela's utility, I am not entirely confident that Sedela cannot be entirely subsumed by denotational design. It remains to be seen.

Sedela Language Definition

Axiom :=	Axiom**[!]** "Informal (textual) definition."	*where ! denotes intended effectfulness*
Meaning Type :=	A -> ... -> Z	*where* A ... Z *are* **Type Expressions**
Meaning Defn :=	f (a : A) ... (z : Z) : R = **Expression | Axiom**	*where* f *is the* **Meaning Identifier** *and* a ... z *are* **Parameter Identifiers** *and* A ... Z, R *are* **Type Expressions**
Expression :=	**Example:** f a (g b)	*where* f *and* g *are a* **Meaning Identifiers** *and* a *and* b *are* **Paremeter Identifiers**
Product :=	MyProduct<...> = A **|** (***A*** : A, ..., ***Z*** : Z) **| Axiom**	*where* MyProduct<...> *is the* **Product Identifier** *and* ***A*** ... ***Z*** *are* **Field Identifiers** and A ... Z *are* **Type Expressions**
Sum :=	MySum<...> = | ***A*** of **(**A **| Axiom)** | ... | ***Z*** of **(**Z **| Axiom)**	*where* MySum<...> *is the* **Sum Identifier** *and* ***A*** ... ***Z*** *are* **Case Identifiers** *and* A ... Z *are* **Type Expressions**
Type Identifier :=	**Product Identifier | Sum Identifier**	
Type Expression :=	**Meaning Type | Type Identifier**	
Type Parameters :=	**Type Identifier**< A, ..., Z; ***A***<A, ..., Z>; ...; ***Z***<...>>	*where* A ... Z *are* **Type Expressions** *and* ***A*** ... ***Z*** *are* **Category Identifiers** *used for constraining* A ... Z
Category :=	category MyCat<...> = | f : A | ... | g : Z	*where* MyCat<...> *is the* **Category Identifier** *and* f ... g *are* **Equivilence Identifiers** *and* A ... Z *are* **Types Expressions**
Witness :=	witness ***A*** = | f (a : A) ... (z : Z) : R = **Expression** | ... | g (a : A) ... (z : Z) : R = **Expression**	*where* ***A*** *is a* **Category Identifier** *and* f ... g *are* **Equivilence Identifiers** *and* a ... z *are* **Parameter Identifiers** *and* A ... Z, R *are* **Type Expressions**
Categorization :=	**Rule:** *iff type* A *has a witness for category* ***A,*** A *is allowable for type parameter categorized as* ***A***	

Line Comment :=	**Example:** // comment text
fun *a* *b* ... *z* -> *expr* **:=**	\a (\b (... \z.*expr*))
a **->** *b* **:=**	_ = (_ : *a*) : *b*
() **:=**	**Explanation:** The unit type / value.
f . g **:=**	**Explanation:** Function composition.

Sedela Language Prelude

```
Any = Axiom "The universal base type."
Bool = Axiom "A binary type."
Real = Axiom "A real number type."
Whole = Axiom "A whole number type."
String = Axiom "A textual type."
Maybe<a> = | Some of a | None
Either<a, b> = | Left of a | Right of b
List<a> = | Nil | Link of (a, List<a>)
Map<a, b> = | Leaf of (a, b) | Node of (Map<a, b>, Map<a, b>)

category Semigroup<a> =
  | append : a -> a -> a

category Monoid<m; Semigroup<m>> =
  | empty : m

category Functor<f> =
  | map<a, b> : (a -> b) -> f<a> -> f<b>

category Pointed<p> =
  | pure<a> : a -> p<a>

category FunctorPointed<f; Functor<f>; Pointed<f>>

category Applicative<p; FunctorPointed<p>> =
  | apply<a, b> : p<a -> b> -> p<a> -> p<b>

category Monad<m; Applicative<m>> =
  | bind<a, b> : m<a> -> (a -> m<b>) -> m<b>

category Alternative<l; Applicative<l>> =
  | empty<a> : l<a>
  | choice : l<a> -> l<a> -> l<a>

category Comonad<c; Functor<c>> =
  | extract<a> : c<a> -> a
  | duplicate<a, b> : c<a> -> c<c<a>>
  | extend<a, b> : (c<a> -> b) -> c<a> -> c<b>

category Foldable<f> =
  | fold<a, b> : (b -> a -> b) -> f<a> -> b
```

```
category Traversable<t; Functor<t>; Foldable<t>> =
  | traverse<a, b, p; Applicative<f>> : (a -> p<b>) -> t<a> -> p<t<b>>

category Functor2<g; Functor<g>> =
  | map2<a, b, c> : g<a> -> g<b> -> g<c>

category Producible<p; Functor2<p>> =
  | product<a, b> : p<a> -> p<b> -> p<(a, b)>

category Summable<s; Producible<s>> =
  | sum<a, b> : s<a> -> s<b> -> s<Either<a, b>>

category Foldable2<f; Foldable<f>> =
  | fold2<a, b, c> : (c -> a -> b -> c) -> f<a> -> f<b> -> c

// TODO: define the Arrow categories.

id a = a
const a _ = a
flip f a b = f b a
```

Nu Script Semantic Design

```
script (str : String) = Axiom "Denotes script code in str."

witness Monoid =
  | append = script "+"
  | empty = script "toEmpty -t-"

witness Monad =
  | pure = script "[fun [a] [pure -t- a]]"
  | map = script "map"
  | apply = script "apply"
  | bind = script "bind"

witness Foldable =
  | fold = script "fold"

witness Functor2 =
  | map2 = script "map2"

witness Summable =
  | product = script "product"
  | sum = script "sum"

Property = Axiom "A property of a simulant."
Relation = Axiom "Indexes a simulant or event relative to the local simulant."

get<a> : Property -> Relation -> a = Axiom "Retrieve a property of a simulant indexed by Relation."
set<a> : Property -> Relation -> a -> a = Axiom! "Update a property of a simulant indexed by Relation, then returns its value."

Stream<a> = Axiom "A stream of simulant property or event values."
getAsStream<a> : Property -> Relation -> Stream<a> = script "getAsStream"
setAsStream<a> : Property -> Relation -> Stream<a> = script "setAsStream"
makeStream<a> : Relation -> Stream<a> = script "makeStream"
mapStream<a, b> (a -> b) -> Stream<a> -> Stream<b> = script "map"
foldStream<a, b> : (b -> a -> b) -> b -> Stream<a> -> b = script "fold"
map2Stream<a, b, c> : (a -> b -> c) -> Stream<a> -> Stream<b> -> Stream<c> = script "map2"
productStream<a, b> : Stream<a> -> Stream<b> -> Stream<(a, b)> = script "product"
sumStream<a, b> : Stream<a> -> Stream<b> -> Stream<Either<a, b>> = script "sum"
```

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

```
Symbol =
  | Atom of String
  | Number of String
  | String of String
  | Quote of Symbol
  | Symbols of List<Symbol>
symbolToString (symbol : Symbol) : String = Axiom "Convert a symbol to string."
symbolFromString (str : String) : Symbol = Axiom "Convert a string to a symbol."

Vsync<a> =
  Axiom "The potentially asynchronous monad such as the one defined by Prime."
vsyncReturn<a> (a : a) : Vsync<a> =
  Axiom "Create a potentially asynchronous operation that returns the result 'a'."
vsyncMap<a, b> (f : a -> b) (vsync : Vsync<a>) : Vsync<b> =
  Axiom "Create a potentially asynchronous operation that runs 'f' over computation of 'a'."
vsyncApply<a, b> (f : Vsync<a> -> Vsync<b>) (vsync : Vsync<a>) : Vsync<b> =
  Axiom "Apply a potentially asynchronous operation to a potentially asynchronous value"
vsyncBind<a, b> (vsync : Vsync<a>) (f : a -> Vsync<b>) : Vsync<b> =
  Axiom "Create a potentially asynchronous operation."

witness Monad =
  | pure = vsyncReturn
  | map = vsyncMap
  | apply = vsyncApply
  | bind = vsyncBind

IPAddress = String
NetworkPort = Whole
Endpoint = (IPAddress, NetworkPort)
Intent = String // the intended meaning of a MetaFunction (indexes a MetaFunction from a Provider – see below)c
Container = Intent -> Symbol -> Vsync<Symbol>
Provider = | Endpoint | Container
MetaFunction = Provider -> Intent -> Symbol -> Vsync<Symbol>

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

attachDebugger (container : Container) = Axiom! "Attach debugger to code called inside the given container."


call (mfn : MetaFunction) provider intent args : Vsync<Symbol> = mfn provider intent args
```

Semantic Design for Unengine (a library for game programming without a monolithic game engine)

```
Time = Axiom "The current simulation time."
Input = Axiom "The current state of HID input."
Address = Axiom "A series of names denoting a simulant in the hierarchy."
Listener = Axiom "A generalized reference to a Listener<_>."
Simulant = Axiom "A generalized reference to a Simulant<_, _, _, _, _>."
RenderMsg = Axiom "A sum type representing the different render requests a simulant can send."
IOMsg<MyUpdateMsg> =
  | CreateSimulant of ... | DestroySimulant of ...
  | CreateListener of ... | DestroyListener of ...
  | CreatePhysicsBody of (Address, PhysicsBodyUpdateMsg -> MyUpdateMsg, ...)
  | DestroyPhysicsBody of ...
  | PlaySound of ...
  | ...

Listener<TheirUpdateMsg, MyUpdateMsg> =
  (Address : Address,
   Import : TheirUpdateMsg -> MyUpdateMsg)

Simulant<Config, State, UpdateMsg> =
  (Name : String,
   Persistent : Bool,
   Init : Config -> Time -> (State, List<UpdateMsg>),
   Sense : State -> Time -> Input -> (State, List<UpdateMsg>),
   Update : State -> Time -> UpdateMsg -> (State, List<UpdateMsg>, List<IOMsg<UpdateMsg>>),
   Draw : State -> Time -> List<RenderMsg>,
   Listeners : List<Listener>,
   Children : List<Simulant>)
```