Semantic Design

The following is what I call a 'semantic design' (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 their program's abstract structure separate from and - as much as possible, prior to! - its implementation. I believe that getting a program's abstract structure correct is the most important task of program design and that doing so up front yields maximal benefits. Also important is encoding the program's abstract structure in a way that is not limited by the implementation language – in particular by its type system, such as with dynamic, weak, or ad-hoc type systems (see lisp, python, or any object-oriented language).

The second intended capability of Sedela is to allow program designers to specify their program's intended semantics in one of two ways – in a formal, denotative way (terms defined entirely in terms of other fully defined terms), or in an informal, textual way (terms defined with just descriptive text). Where a denotative approach is required, Sedela allows designers to encode their program's abstract terms in terms of algebraic data types and a typed lambda calculus (here System F^{ς}). Where a more informal approach is permitted, Sedela offers designers the ability to define their terms with 'axioms'. The less formal definitions enabled by 'axioms' makes Sedela a usable tool for describing systems whose denotations may be too complex to warrant formal specification, in particular, for 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 (IE, formally and in full). This is an advantage for those working on greenfield projects and on projects that otherwise demand formal definition (such as with programming languages).

3) Semantic design provides a 'knob' for the level of semantic detail at which designers would like to specify their programs. It has been found to be useful to increase the level of semantic detail for designs by replacing some informal definitions with more detailed ones (terms defined in terms of other terms) while leaving less detailed other definitions for brevity or temporary convenience. Semantic design may end up being a useful bridge from a low-detail 'on-napkins' design to one that can (and should be) specified denotatively with denotational design.

While denotative design seems ideal, I invented semantic design for either one of two reasons – 1) I could not apply denotational

design to my current needs due to its limited domain of application, or 2) 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. To me, 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 := let 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 := let 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 := let MySum<...> = *where* MySum<...> *is the* **Sum Identifier**
| ***A*** of **(**A **| Axiom)** *and* ***A*** ... ***Z*** *are* **Case Identifiers**
| ... *and* A ... Z *are* **Type Expressions**
| ***Z*** of **(**Z **| Axiom)**

Type Identifier := **Product Identifier | Sum Identifier**

Type Expression := **Meaning Type | Type Identifier**

Type Parameters := **Type Identifier**< *where* A ... Z *are* **Type Expressions**
A, ..., Z; *and* ***A*** ... ***Z*** *are* **Category Identifiers** *used for*
A<A, ..., Z>; ...; ***Z***<...>> *constraining* A ... Z

Category := category MyCat<...> = *where* MyCat<...> *is the* **Category Identifier**
| f : A *and* f ... g *are* **Equivilence Identifiers**
| ... *and* A ... Z *are* **Types Expressions**
| g : Z

Witness := witness ***A*** = *where* ***A*** *is a* **Category Identifier**
| f (a : A) ... (z : Z) : R = **Expression** *and* f ... g *are* **Equivilence Identifiers**
| ... *and* a ... z *are* **Parameter Identifiers**
| g (a : A) ... (z : Z) : R = **Expression** *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* **:=**	let _ = (_ : *a*) : *b*
() **:=**	**Explanation:** The unit type / value.
f . g **:=**	**Explanation:** Function composition.

Sedela Language Prelude

```
let Any = Axiom "The universal base type."
let Bool = Axiom "A binary type."
let Real = Axiom "A real number type."
let Whole = Axiom "A whole number type."
let String = Axiom "A textual type."
let Maybe<a> = | Some of a | None
let Either<a, b> = | Left of a | Right of b
let List<a> = | Nil | Link of (a, List<a>)
let 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 Category<t> =
  | id<a> : t<a, a>
  | compose<a, b, c> : t<b, c> -> t<a, b> -> t<a, c>
```

```
category Arrow<a; Category<a>> =
  | arr<b, c> : (b -> c) -> a<b, c>
  | first<b, c, d> : a<b, c> -> a<(b, d), (c, d)>

category ArrowChoice<a; Arrow<a>> =
  | left<b, c, d> : a<b, c> -> a<Either<b,d>, Either<c, d>>

category ArrowApply ...

category ArrowLoop ...

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

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

Semantic Design for MetaFunctions (a replacement for micro-services)

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

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

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

let 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."

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

let 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)

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

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

let 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>)
```