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. 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. Although I may yet write a parser and type-checked for ADELA, there will never be a compiler or intepreter. Thus, ADELA will have no syntax for if expressions or the like. The only meanings that will be defined in its Prelude will be combinators such as id, const, flip, and etc. Its data 'primitives' are all defined as axioms and have no available operations. This is important because it enforces the appropriate barrier between a program’s design in ADELA and its implementation in a compilable language.

Adela Language Definition

**Axiom :=** Axiom**[!]** "Informal 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 | Axiom
  | ...
  | g (a : A) ... (z : Z) : R = Expression | Axiom
```
*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 constrained to* ***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. |

Adela Language Prelude

```
Bool = Axiom "A binary type."
Whole = Axiom "A whole number type."
Real = Axiom "A real number type."
String = Axiom "A textual type."
DateTime = Axiom "A date time type."
Maybe<a> = | Some of a | None
Either<a, b> = | Left of a | Right of b

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

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


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


category MonadPlus<m; Monoid<m>; Monad<m>>


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

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>

id a = a

const a _ = a

flip f a b = f b a
```

Nu Semantic Design

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

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> = Axiom "Construct a stream of values from a simulant property."
setToStream<a> property relation stream = foldStream (fun _ -> set<a> property relation) stream
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<Either<a, b>> = Axiom "Combines two streams into a single sum stream."
mapStream<a, b> mapper stream = foldStream (fun _ -> mapper a) stream


witness Comonad =
  | map = mapStream
  | extract = fun f a -> f a
  | duplicate = fun f -> f f
  | extend f = map f . duplicate
```

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

```
Any = Axiom "The base type of all types."
List<a> = Axiom "The functional list type such as the one defined by F#."
Map<a, b> = Axiom "The functional map type such as the one defined by F#."
Vsync<a> = Axiom "The potentially asynchronous monad such as the one defined by Prime."
Symbol = Axiom "Symbolic type such as the one defined by Prime."

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

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 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 intent args : Vsync<Symbol> = mfn provider intent args
```