Sedela – Semantic Design Language

This is a specification for a typed program design language called ‘Sedela’. Sedela aims to be a flexible, type-checkable, and mostly denotational program design description language. Sedela was inspired by Conal Elliot’s talk, “Denotational Design – From Meanings to Programs” - https://www.youtube.com/watch?v=bmKYiUOEo2A. Sedela is a language for creating ‘semantic designs’, which are just like Conal’s denotational designs, but with ‘propositions’. Put simply –

Semantic Design = Denotational Design + Propositions

Although I am currently writing a parser and a type-checker for Sedela, there will never be a compiler or intepreter. A compiler or interpreter would have no meaning for a pure design language like Sedela.

First, I will present the definition of Sedela, then I will use it to specify the semantic design of MetaFunctions (a system architecture that aims to replace microservices).

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

The primary 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 constrained or warped by the complexity and limitations of its implementation language.

The secondary intended capability of Sedela is to allow program designers to specify their program’s structure in one or both of two ways – in a formal, compositional way (in terms of Sedela’s denotationally-defined features), and / or in an informal, textual way (in terms of ‘propositions’). Where a formal approach is required, Sedela allows designers to encode their program’s design in terms of algebraic data types and a typed lambda calculus (here System $F_{<:}$ with Type Families and opt-in Subtyping). Where a more informal approach is permissable, Sedela allows designers to encode their program’s design in terms of natural language via ‘propositions’. By leveraging both the denotationally-defined features alongside propositions, Sedela can be used to describe systems that may be too complex to warrant formal specification, in particular, for legacy systems.

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

Denotational design restricts its domain of application to programs whose structure can be specified completely and denotatively. This is an advantage for those working on greenfield projects whose design demands such formal definition (such as writing a new programming language). In contrast, Sedela’s semantic design allows designs to leave portions of their system informally-specified via ‘propositions’, thus providing a ‘knob’ for the level of detail at which designers can specify their design. For this reason, I think Sedela makes more sense for designing legacy systems than Conal’s approach.

Sedela, being a explicitly-specified language - unlike Conal’s notation – will also provide a parser and type-checker out-of-the-box. Once this tooling is complete, Sedela may become much preferable to Conal’s notation.

Sedela Language Definition

Proposition := Proposition**[!]** "Informal (natural language) definition."
where ! denotes intended effectfulness

Function Type := A -> ... -> Z
where A ... Z *are* **Type Expressions**

Function Defn := let f (a : A) ... (z : Z) : R = **Expression | Proposition**
where f *is the* **Function Identifier**
and a ... z *are* **Parameter Identifiers**
and A ... Z, R *are* **Type Expressions**

Derivation := **Nested Example:** f a (g b)
where f *and* g *are a* **Function Identifiers**
and a *and* b *are* **Binding Identifiers**

Product := type MyProduct<...> = A **|** (***A*** : A, ..., ***Z*** : Z) **| Proposition**
where MyProduct<...> *is the* **Product Identifier**
and ***A*** ... ***Z*** *are* **Field Identifiers**
and A ... Z *are* **Type Expressions**

Sum :=
```
type MySum<...> =
  | A of (A | Proposition)
  | ...
  | Z of (Z | Proposition)
```
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 := **Function 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* **:=** let _ = (_ : *a*) : *b*

() **:=** **Explanation:** The unit type / value.

f . g **:=** **Explanation:** Function composition.

Any? **:=** **Explanation:** The universal subtype. Only types that end with '?' allow for substitution (this preserving free theoroms elsewhere).

Sedela Language Prelude

```
type Bool = True | False
type Maybe<a> = | Some of a | None
type Either<a, b> = | Left of a | Right of b
type List<a> = | Nil | Link of (a, List<a>)
type 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 Pointed<p> =
    pure<a> : a -> p<a>

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

category PointedFunctor<f; Pointed<f>; Functor<f>>

category Applicative<p; PointedFunctor<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 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 ... // TODO: implement!

category ArrowLoop ... // TODO: implement

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

category Category<t> =
    id<a> : t<a, a>
    compose<a, b, c> : t<b, c> -> t<a, b> -> t<a, c>

category Cartesian<k; Category<k>> = // taken from Conal Elliott’s talk “Compiling to Categories”
    type Cross<u, v> // a type alias family member
    exl : k<Cross<a, b>, a>
    exr : k<Cross<a, b>, b>
    fork : k<a, c> -> k<a, d> -> k<a, Cross<c, d>>

category Cocartesian<k; Category<k>> =
    type Plus<u, v>
    inl : k<a, Plus<a, b>>
    inr : k<b, Plus<a, b>>
    join : k<a, c> -> k<a, d> -> k<Cross<c, d>, a>

category CartesianClosed<k; Cartesian<k>> =
    type Yield<a, b>
    apply : k<Cross<Yield<a, b>, a>, b>
    curry : k<Cross<a, b>, c> -> k<a, Yield<b, c>>
    uncurry : k<a, Yield<b, c>> -> k<Cross<a, b>, c>
```

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

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

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

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

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

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

let makeContainer (asynchrounous : Bool) (repositoryUrl : String) (credentials : (String, String)) (envDeps : Map<String, Any>) :
    Container = Proposition "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) = Proposition! "Attach debugger to code called inside the given container."

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

Additional Examples

Please see the Sedela design for the Nu Game Engine here -

https://github.com/bryanedds/Nu/blob/master/Nu/Nu.Documentation/Nu%20Semantic%20Design.pdf