Semantic Design for an Observable Property System

```
let Participant = Axiom "Associates with observable properties."
let PropertyChangeHandler<Key> = EventSystem -> EventSystem -> EventSystem
and PropertyChangeUnhandler<Key> = EventSystem -> EventSystem
and EventSystem = Axiom "A publisher-neutral event system."

let participantExists : Participant? -> EventSystem -> bool =
    Axiom "Check that a participant exists."

let getPropertyOpt<A> : String -> Participant? -> EventSystem -> Maybe<A> =
    Axiom "Obtain a participant property if it exists."

let setPropertyOpt<A> : String -> Participant? -> Maybe<A> -> EventSystem -> EventSystem =
    Axiom "Set a participant property if it exists."

let handlePropertyChange : String -> Participant? -> PropertyChangeHandler -> (PropertyChangeUnhandler, EventSystem) =
    Axiom "Invoke the given handler when a participant property is changed."
```

Semantic Design for Nu Game Engine

```
let World = Axiom "The world value."
let Simulant = (SimulantAddress : Address<Simulant>; Participant)
let Game = (GameAddress : Address<Game>; Simulant)
let Screen = (ScreenAddress : Address<Screen>; Simulant)
let Layer = (LayerAddress : Address<Layer>; Simulant)
let Entity = (EntityAddress : Address<Entity>; Simulant)
let Dispatcher = Axiom "Specifies the shape and behavior of a simulant."


let getGame : World -> Game = Axiom "Get the global game handle."
let getScreens : World -> List<Screen> = Axiom "Get all screen handles belonging to the global game."
let getLayers : Screen -> World -> List<Layer> = Axiom "Get all layer handles belonging to the given screen."
let getEntities : Layer -> World -> List<Entity> = Axiom "Get all entity handles belonging to the given layer."

let tryGetParent : Simulant -> World -> Maybe<Simulant> = Axiom "Attempt to get the parent of a simulant."
let getChildren : Simulant -> World -> List<Simulant> = Axtion "Get the children of a simulant."
let getProperty : String -> Simulant -> World -> Any = Axiom "Get the property of a simulant."
let getDispatcher : Simulant -> World -> Dispatcher = Axiom "Get the dispatcher belonging to a simulant."
let getPropertyDefinition : String -> Dispatcher -> World -> PropertyDefinition = Axiom "Get property definition of dispatcher."
let getBehaviors<A, S :> Simulant> : Dispatcher -> World -> List<Behavior<A, S>> = Axiom "..."

let PropertyDefinition =
    (Type : Axiom "A value type.",
     Default : Any)

let Event<A, S :> Simulant> =
    (Data : Any,
     Publisher : Simulant,
     Subscriber : S,
     Address : Address<A>)

let Behavior<A, S :> Subscriber> =
    Event<A, S> -> World -> World
```

Nu Script Semantic Design

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

witness Monoid =
  | append = script "+"
  | empty = script "[empty -t-]"

witness Monoid =
  | append = script "*"
  | empty = script "[identity -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"

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

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

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