Semantic Design for the Observable-Property System

```
let Propertied = Proposition "Something with observable properties."
let PropertyChangeHandler = PropertySystem -> PropertySystem -> PropertySystem
and PropertyChangeUnhandler = PropertySystem -> PropertySystem
and PropertySystem = Proposition "An observable-property system."

let fail<a> : String -> a =
    Proposition "Terminate evaluation with an error with the given message."

let getPropertyOpt<a> : String -> Propertied? -> PropertySystem -> Maybe<a> =
    Proposition "Obtain a property if it exists."

let setPropertyOpt<a> : String -> Propertied? -> Maybe<a> -> PropertySystem -> PropertySystem =
    Proposition "Set a property, creating it if Some and it doesn’t yet exist, destroying it if None and it does exist."

let handlePropertyChange : String -> Propertied? -> PropertyChangeHandler -> (PropertyChangeUnhandler, PropertySystem) =
    Proposition "Invoke the given handler when a property is changed."

witness Comonad<PropertySystem> =
    extract n p s =
        match getPropertyOpt n p s with
        | Some a -> a
        | None -> fail "Cannot find property."
    extend n p f s =
        match getPropertyOpt n p with
        | Some a -> setPropertyOpt n p (Some (f a)) s
        | None -> fail "Cannot find property."
    duplicate n p s =
        setPropertyOpt n p (Some s) s
```

Semantic Design for the Publisher-Neutral Event System

```
let Address<a> = List<String>
let Participant = (_ : Proposition "A participant in the event system.", Propertied)
let Event<a, s :> Participant> = (Data : a, Publisher : Simulant, Subscriber : s, Address : Address<a>)
let EventSystem = (_ : Proposition "A publisher-neutral event system."; PropertySystem)
let EventHandler<a, s :> Participant> = Event<a, s> -> EventSystem -> EventSystem
let EventUnhandler = EventSystem -> EventSystem
let Stream<a> = Proposition "A stream of data flowing from events."
let Chain<e, a> = Proposition "A programmable ‘chain’ of events."

let getLiveness : EventWorld -> Bool =
    Proposition "Check that the event system is either live or terminated."

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

let publish<a, p :> Participant> : a -> Address<a> -> p -> EventSystem -> EventSystem =
    Proposition "Publish an event with the given data with the given event address for the given participant."

let subscribe<a, s :> Participant> : Address<a> -> s -> EventSystem -> EventHandler<a, s> -> (EventUnhandler, EventSystem)
    Proposition "Subscribe to an event with the given event address with the given subscriber."

let mapStream<a, b> : (a -> b) -> Stream<a> -> Stream<b> =
    Proposition "Map over a stream."

let foldStream<a, b> : (b -> a -> b) -> b -> Stream<a> -> b =
    Proposition "Fold over a stream."

let map2Stream<a, b, c> : (a -> b -> c) -> Stream<a> -> Stream<b> -> Stream<c> =
    Proposition "Map over two stream."

let productStream<a, b> : Stream<a> -> Stream<b> -> Stream<(a, b)> =
    Proposition "Make a pairwise product from two streams."

let sumStream<a, b> : Stream<a> -> Stream<b> -> Stream<Either<a, b>> =
    Proposition "Make an either sum from two streams."

let pureChain<a> : a -> Chain<e, a> =
    Proposition "Construct a chain from a single value."

let mapChain<a, b> : (a -> b) -> a -> Chain<e, b> =
    Proposition "Construct a chain from a single value."
```

```
let applyChain<a, b> : Chain<e, a -> b> -> Chain<e, a> -> Chain<e, b> =
    Proposition "Apply a function in the context of a Chain."

let bindChain<e, a> : Chain<e, a> -> (b -> Chain<e, b>) -> Chain<e, b> =
    Proposition "A monadic bind over a chain"

witness Functor =
    map = mapStream

witness Foldable =
    fold = foldStream

witness Functor2 =
    map2 = map2Stream

witness Summable =
    product = productStream
    sum = sumStream

witness Monad =
    pure = pureChain
    map = mapChain
    apply = applyChain
    bind = bindChain
```

Semantic Design for Nu Game Engine

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

let getGame : World -> Game = Proposition "Get the global game handle."
let getScreens : World -> List<Screen> = Proposition "Get all screen handles belonging to the global game."
let getGroups : Screen -> World -> List<Group> = Proposition "Get all group handles belonging to the given screen."
let getEntities : Group -> World -> List<Entity> = Proposition "Get all entity handles belonging to the given group."

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

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

let Behavior<a, s :> Subscriber> =
    Event<a, s> -> World -> World
```

Nu Script Semantic Design

```
let script (str : String) = Proposition "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 = Proposition "A property of a simulant."
let Relation = Proposition "Indexes a simulant or event relative to the local simulant."

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

let Stream<a> = Proposition "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"
```