Semantic Design for the Observable-Property System

let Propertied = Axiom "Something with observable properties."

let PropertyChangeHandler = PropertySystem -> PropertySystem -> PropertySystem

and PropertyChangeUnhandler = PropertySystem -> PropertySystem

and PropertySystem = Axiom "An observable-property system."

let getPropertyOpt<a> : String -> Propertied? -> PropertySystem -> Maybe<a> =

Axiom "Obtain a participant property if it exists."

let setPropertyOpt<a> : String -> Propertied? -> Maybe<a> -> PropertySystem -> PropertySystem =

Axiom "Set a participant property if it exists."

let handlePropertyChange : String -> Propertied? -> PropertyChangeHandler -> (PropertyChangeUnhandler, PropertySystem) =

Axiom "Invoke the given handler when a participant property is changed."

// TODO: See what categories the property system yields and make witnesses for them.

Semantic Design for the Publisher-Neutral Event System

let Address<a> = List<String>

let Participant = (\_ : Axiom "A participant in the event system.", Propertied)

let Event<a, s :> Participant> = (Data : a, Publisher : Simulant, Subscriber : s, Address : Address<a>)

let EventSystem = (\_ : Axiom "A publisher-neutral event system."; PropertySystem)

let EventHandler<a, s :> Participant> = Event<a, s> -> EventSystem -> EventSystem

let EventUnhandler = EventSystem -> EventSystem

let Stream<a> = Axiom "A stream of data flowing from events."

let Chain<e, a> = Axiom "A programmable ‘chain’ of events."

let getLiveness : EventWorld -> Bool =

Axiom "Check that the event system is either live or terminated."

let participantExists : Participant? -> EventWorld -> bool =

Axiom "Check that a participant exists."

let publish<a, p :> Participant> : a -> Address<a> -> p -> EventSystem -> EventSystem =

Axiom "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)

Axiom "Subscribe to an event with the given event address with the given subscriber."

let mapStream<a, b> : (a -> b) -> Stream<a> -> Stream<b> =

Axiom "Map over a stream."

let foldStream<a, b> : (b -> a -> b) -> b -> Stream<a> -> b =

Axiom "Fold over a stream."

let map2Stream<a, b, c> : (a -> b -> c) -> Stream<a> -> Stream<b> -> Stream<c> =

Axiom "Map over two stream."

let productStream<a, b> : Stream<a> -> Stream<b> -> Stream<(a, b)> =

Axiom "Make a pairwise product from two streams."

let sumStream<a, b> : Stream<a> -> Stream<b> -> Stream<Either<a, b>> =

Axiom "Make an either sum from two streams."

let pureChain<a> : a -> Chain<e, a> =

Axiom "Construct a chain from a single value."

let mapChain<a, b> : (a -> b) -> a -> Chain<e, b> =

Axiom "Construct a chain from a single value."

let applyChain<a, b> : Chain<e, a -> b> -> Chain<e, a> -> Chain<e, b> =

Axiom "Apply a function in the context of a Chain."

let bindChain<e, a> : Chain<e, a> -> (b -> Chain<e, b>) -> Chain<e, b> =

Axiom "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 = (\_ : Axiom "The world value."; EventSystem)

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 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"