1 Formalization Structure

This section gives a high-level overview of the definitions involved in the formalization of the Reactor model:

1. Components of the Reactor model
2. Means of changing and retrieving objects within a hierarchy of reactors
3. Execution model
4. Determinism and proof thereof

Most code shown in this section is intended purely to aid explanation and will be covered in more detail in subsequent sections.

1.1 Components of the Reactor Model

Formalizing reactors will require us to define the following components. The first two are rather a technicality, and can be glossed over.

1.1.1 IDs

Throughout the Reactor model, we use IDs to reference various kinds of components like ports, reactions, actions, etc. The precise nature of IDs isn't relevant, and should remain "opaque". [1] There are two ways of achieving this:

1. We can make components like reactions and reactors generic over an ID-type (much like list- or set-types are generic over their element-type in many programming languages).
2. We can define the type of IDs as a "constant". In Lean, constants can be considered as "opaque" definitions. That is, the only thing we know about the object we're defining is its type – we can never unfold its definition. Thus, by defining IDs as `constant ID : Type`, we can use `ID` throughout the model as if it were a generic type, without having to explicitly declare it as a type parameter on various components.

We opt for the second approach as it reduces notation overhead and as we don't need the option of specifying the precise type of IDs in the model.

[1] The definition of reactors will implicitly impose some structure on IDs (the type needs to have at least as many distinct members as there are identifiable components in a reactor), but this does not affect their opaque nature.

1.1.2 Values

Values are the objects upon which we perform computations in the Reactor model. They are similar to IDs in that their precise structure is irrelevant – we therefore define them as constants again. One feature we *do* need to impose on values is that they have a special element called the "absent value" (denoted by ⊥). This value is used to represent the absence of a value in ports. We can still formalize this with constants, giving us an opaque type `Value` and opaque element of this type called `Value.absent`:

```
constant _Value : Σ V : Type, V := Sigma.mk Unit Unit.unit
def Value : Type _ := _Value.fst
constant Value.absent : Value := _Value.snd

notation "⊥" => Value.absent
```

1.1.3 Ports

Ports are the interface points of a reactor, through which they can exchange values. There are two kinds of ports: input and output ports. The resulting definition reflects precisely these two aspects:

```
inductive Port.Kind
  | «in»
  | out

structure Port where
  kind : Port.Kind
  val : Value
```

1.1.4 Changes & Reactions

Reactions are the basic computational units in the Reactor model. They take input values and output a set of "changes" to be realized in their reactor.

Changes are analogous to API-calls that can be performed by reactions in a Lingua Franca program. The following can be read as defining the `Change` type as an enumeration where each case has a payload:

```
inductive Change
  | port (target : ID) (value : Value)
  | state (target : ID) (value : Value)
  | action (target : ID) (time : Time) (value : Value)
  | connect (src : ID) (dst : ID)
  | disconnect (src : ID) (dst : ID)
  | create (cl : Reactor.Class)
  | delete (rtr : ID)
```

If a reaction outputs a `Change.port p v` this is analogous to a reaction calling `SET` with port `p` and value `v` in Lingua Franca. As a reaction can perform multiple API-calls in a single execution of its body, we formalize a reaction's body as producing a *list* of changes:

```
structure Reaction where
  body :        Input → List Change
  deps :        Port.Kind → Finset ID
  triggers :    Finset ID
  prio :        Priority
  tsSubInDeps : triggers ⊆ deps «.in»
  ... -- Additional constraints omitted.
```

Here we can also see that reactions have anti-/dependencies (which we also refer to as input and output dependencies), triggers and a priority. The last field `tsSubInDeps` shows a feature of dependently types languages like Lean: we can constrain instances of types to fulfill given propositions. Here we place the constraint that for any instance of `Reaction` it must hold that the set of triggers is a subset of the reaction's input dependencies.

Mutations We introduce a distinction between different types of reactions, based on whether they can produce "mutating" changes. A change is considered mutating if it can change the *structure* of a reactor. This is the case for `connect`, `disconnect`, `create` and `delete`.

If there is any input for which a reaction produces a mutating change, we call that reaction a "mutation". If a reaction never produces mutating changes, we call it a "normal" reaction.

Pure Reactions Another aspect by which we differentiate reactions is whether they are "pure". A reaction is considered pure if its output does not depend on its reactor's state, and it only produces changes to ports and actions.

The purity of a reaction will become relevant when formalizing connections between ports through a special kind of reaction called a "relay reaction", which needs to have a specific priority that is only available to pure reactions.

1.1.5 Reactors

Reactors are the building blocks which combine the previously described components:

```
inductive Raw.Reactor
  | mk
    (ports : ID ⇉ Port)
    (state : ID ⇉ Value)
    (rcns :  ID ⇉ Reaction)
    (nest :  ID → Option Raw.Reactor)
    (acts :  ID ⇉ Time.Tag ⇉ Value)
```

The double-arrow symbol ⇉ denotes a finite map (basically a hashmap or partial function defined on finitely many inputs). Thus, a reactor consists of:

- a set of identified ports
- a set of state variables, which are just identified values
- a set of identified reactions – as previously mentioned, we will use a special kind of reaction to model connections between reactors' ports
- a set of identified nested reactors (we can't define these as a finite map yet, due to technical reasons detailed below)
- a set of identified actions, where each action is a mapping from a time tag to a value

Structuring the components of a reactor such that they are all "hidden" behind IDs will later allow us to easily change and retrieve components in a reactor hierarchy by referring to their IDs.

Raw to Proper The type defined above is called `Raw.Reactor` as it is missing some of the constraints that will need to be present in reactors. For example, we need to constrain each input port to have at most one incoming connection. For technical reasons (which relate to how Lean maps inductive definitions to its underlying mathematical theory), the recursive nature of reactors doesn't allow us to place constraints directly on the `Raw.Reactor` type. Instead, we will need to perform the following workaround:

1. We define a "raw" reactor type without constraints (this is `Raw.Reactor` as shown above).
2. We define the required set of constraints using this raw reactor type. A raw reactor satisfying these constraints is then called "directly well-formed".
3. We define a notion of (complete) "well-formedness", which holds if the raw reactor itself, as well as all of its nested reactors are directly well-formed.
4. We define a "proper" `Reactor` type as the type of well-formed raw reactors.
5. We restate all of the constraints which were previously stated for raw reactors in terms of proper reactors and prove that the `Reactor` type satisfies these constraints (we call this process of going from the raw world to the proper world "lifting").

The crux of this process is to perform the lifting in such a way that proper reactors never leak their raw underpinnings. Thus, any subsequent uses of reactors can completely ignore this workaround.

1.2 Changing and Retrieving Objects