Provable Determinism in Reactors

Marcus Rossel
Supervised by Andrés Goens

April 1, 2021

Bachelor Thesis at Technical University Dresden
Chair for Compiler Construction

Matriculation Number: 4744651

Fakultät Informatik Institut für Technische Informatik, Professur Compilerbau

Task Description for Bachelor Thesis

for:	**Marcus Rossel**
Major:	Bachelor Informatik, Year 2017
Matriculation Nr.:	4744651
Title:	**Provable Determinism in Reactors**

The model of Reactors [1] is a novel Model of Computation developed in collaboration between multiple institutions, including the Chair for Compiler Construction. In the model, applications are divided into reactors with individual reactions which specify semantics in a distributed fashion. A central aspect of the model are its time semantics with two distinct timelines [2].

While we claim Reactors to be deterministic [1] under certain conditions, we have not formally proved this. In fact, the formulation of [1] has multiple aspects of the formalization that are imprecise.

The goal of this thesis is to fill the existing gap in the formalization and proof of determinism in the Reactors model. We plan to use the Lean theorem prover, a proof assistant based on the Calculus of Constructions and developed by Microsoft Research. Lean has an open, welcoming community and an ambition project, Mathlib, which aims to formalize most of modern mathematics. As such, it is a great environment for formalizing a proof of determinism for Reactors. Since the Reactors model is complex and in constant evolution, this thesis will focus on a core of the model to prove determism.

In particular, this Bachelor Thesis shall include the following tasks. The student shall:

1. Learn and understand the Reactors model. Learn and understand the Lean language and how to use the theorem prover.
2. Formalize a core model of Reactors in Lean, adapting and filling the imprecise aspects of the original formalization.
3. Prove the model is deterministic for acyclic reaction graphs and without physical actions.
4. Optional: extend the core model formalized and/or relax some of the assumptions and prove determinism in this more general setting.
5. Optional: investigate under which conditions determinism holds also for (logical) timings in the model, not only for the values resulting from channels.

[1] Lohstroh, Marten, et al. "Reactors: A deterministic model for composable reactive systems." Cyber Physical Systems. Model-Based Design. Springer, Cham, 2019. 59-85.

[2] Lohstroh, Marten, et al. "A Language for Deterministic Coordination Across Multiple Timelines." 2020 Forum for Specification and Design Languages (FDL). IEEE, 2020.

Declaration

I declare that I have prepared this thesis independently and that I used only the references and auxiliary means indicated in the thesis.

[illegible]

Marcus Rossel

Dresden — April 1, 2021

Abstract

Reactors are a novel model of computation introduced in “Reactors: A Deterministic Model for Composable Reactive Systems” [1]. A core claim of this model is that it behaves predictably under certain conditions, i.e. it is deterministic. Currently, its mathematical model is neither well-defined nor verified. In this thesis we formalize a well-defined subset of the Reactor model and rigorously prove its determinism. To ensure correctness of our mathematical work, we employ a modern approach to formalization, by using a proof assistant: *Lean Theorem Prover*. Along the way we show some of the unique aspects of working with Lean.

Contents

1 Introduction

In 2013, the *Reactive Manifesto* [2] defined the core principles of so-called *Reactive systems*. They are software systems, which aim to be

> more flexible, loosely-coupled and scalable. This makes them easier to develop and amenable to change. They are significantly more tolerant of failure and when failure does occur they meet it with elegance rather than disaster. [2]

The manifesto does not mandate hard and fast rules for reactive systems. Hence, over the years a variety of systems with different trade-offs have been created.[1] Some of these systems are implicitly defined by their software implementations, while others define their behavior upon mathematical foundations. In this thesis, we consider a specific model of computation for reactive systems called *Reactors*.[2] The Reactor model aims at minimizing one of the downsides of reactive systems:

> While scalability, resilience, elasticity, and responsiveness — all tenets of the manifesto — are clearly important, the gains in these dimensions come at the loss of testability due to the admittance of nondeterminism. [...] We argue that the goals of reactive programming can also be achieved without adopting a nondeterministic programming model. [1]

Loosely speaking, a system is considered *deterministic* if multiple executions of the system on the same inputs exhibit the same behavior. While determinism is rather common in closed software systems, it can be hard to maintain upon interaction with real-world components, like sensors, actuators, etc. This is especially unfortunate in that these so-called *cyber-physical systems* can come with a much higher cost of failure than purely digital systems.[3] A vital means for avoiding software failure, especially for larger systems, is traditional testing. While software testability is determined by many factors, it is significantly impaired if the system under test is inherently nondeterministic [4, p. 7]. Hence, the Reactor model's stride towards *determinism* is also a stride towards *testability*.[4]

While the Reactor model aims at being deterministic under certain conditions, its determinism is yet unproven. Proving this claim is also hindered by the fact that the model lacks a well-defined formalization. Thus, the first goal of this thesis is to provide a well-defined subset of the model.

[1] E.g. the cross-platform framework *ReactiveX* and Apple's main UI-framework *SwiftUI*.
[2] An implementation of this model is realized by *Lingua Franca* [3].
[3] For an example, cf. [1] Section 1.1.
[4] For a complete discussion of the benefits of Reactors, cf. [1] and [4].

1.1 Reactor Model

In this section we start by giving a very brief overview of Reactors. Further details are discussed in subsequent sections.

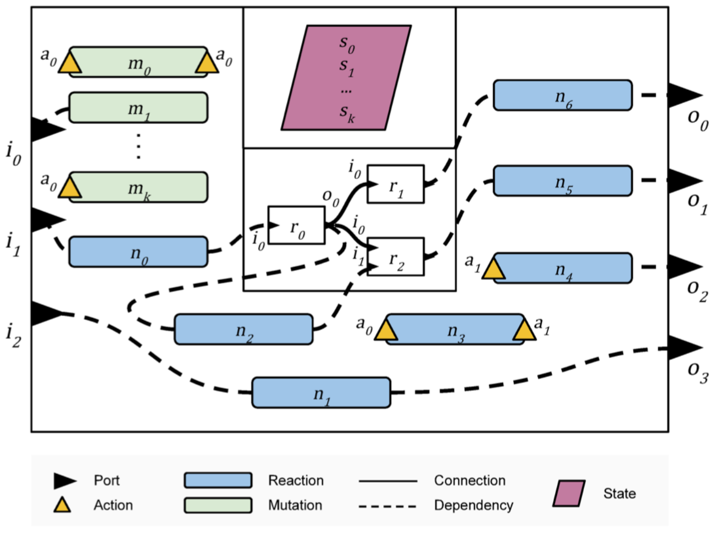

Figure 1: Schematic representation of a reactor [1]

Components: The Reactor model is best understood visually — Figure 1 illustrates most of its components. The entirety of the illustration shows a single *reactor*, which serves as an isolated container of functionality. Communication, i.e. data flow, between different reactors is achieved by their *ports*. Ports can be written to and read from, and therefore carry *values*. The main components that read and write values are called *reactions*. They can be considered the atomic functional units within a reactor. Reactions are *triggered* by incoming values, perform computations on those values and produce a resulting output. Aside from their purely functional aspect though, reactions also have access to mutable *state* that is shared among all reactions in a given reactor.

The full Reactor model has a breadth of additional components and features that allow modeling complex behaviors [4, pp. 18-40]. For the sake of brevity, this thesis will only consider a subset of those features, which we call the *Simple Reactor* model.

Semantic Time: The Simple Reactor model, as described so far, is purely instantaneous — it has no notion of time. A key feature of the Reactor model, is its *semantic notion of time*. It introduces two distinct variants of time: *logical* and *physical*. While physical time represents "real" time, as on a clock, logical time is a system-internal notion. That is, the separation between the digital and the physical worlds that is present in cyber-physical systems, is applied to time as well [5, p. 235]. Separating these two notions of time makes it possible to reason deterministically about events in logical time, while still allowing for interaction with physical time under explicitly defined semantics. While the details of these interactions are not explored in this thesis, we will reap some of the benefits of logical time in later sections. For example, we can easily define an order of execution of events within logical time, such that we need not care about their actual runtimes. Adding a notion of physical time would further allow us to impose physical time limits on their execution.

Actions: Time manifests itself in reactors through *actions*. They provide a means for reactions to send and receive values *over time*. The distinct notions of time described above carry over to distinct notions of actions — there are physical and logical actions. A reaction can *schedule* a logical action, to occur at a specific logical time in the future. When that time is reached, the target of that action will receive a specified value as input. Physical actions on the other hand have external origins — they are scheduled by entities like physical sensors. In the Simple Reactor model, we only consider *logical* time and its actions.

1.2 Simple Reactor Model

In this section we define the Simple Reactor model as a subset of the full Reactor model (with minor changes). The definitions given here may change considerably in the rigorous formalization presented in Section 3, as they contain problematic inaccuracies.[5] Additionally, the following definitions may be partially redundant. This is a result of trying to remove any aspects not relevant to readers unfamiliar with [1], while still showing how this model is obtained by reducing the full Reactor model.

Identifiers & Values: Two of the primitive notions in the Reactor model are *identifiers* and *values*. Identifiers, defined as members of a set Σ, are used to refer to a variety of components within the Reactor model: state variables, actions, ports, etc. A set of values $\mathfrak{V}$ contains opaque values that are the "data" upon which computation takes place. We require that $\mathfrak{V}$ contain a special value ϵ called the *absent value*. The opaque nature of identifiers and values shows that our model does not depend on their structure.[6]

Tags: Logical time progresses in discrete steps. For each of those steps we define a *tag*, as an element of $\mathbb{T} = \mathbb{N} \times \mathbb{N}$. For a tag $t = (v, m)$ we call v the *time value* and m the *microstep index*. Notationally, we reference fields of a tuple by their given names. For example, for a given tag t, its time value can be

[5]Cf. Section 1.2.2.

[6]We will see in Section 5.1 that it *can* be useful to add some well-defined structure to them.

referenced as $v(t)$ and its microstep index as $m(t)$. This representation of time is called *superdense* [5], as it allows us to step through an arbitrary number of tags before reaching a new time value. The order of these tags is given by their lexicographical order[7].

Actions: An action a is defined as $a = (x, d, \mathfrak{o})$, where:

1. $x \in \Sigma$ is the action's *identifier.*
2. $d \in \mathbb{N}$ is the *delay* — an offset from the current logical time that specifies when the action should be scheduled.
3. $\mathfrak{o} \in \mathfrak{O}$ is the *origin* — a descriptor for whether the action is logical or physical. Since the Simple Reactor model only allows for logical actions, the set $\mathfrak{O}$ of *origins* can be described by the unit set $\{logical\}$, and is therefore redundant.

Scheduling an action implies the creation of an event. The value carried by an action is placed in its corresponding event.

Events: An event e is defined as $e = (\mathfrak{t}, \mathfrak{v}, \mathfrak{g})$, where:

1. $\mathfrak{t} \in \Sigma$ is the *event trigger* — the object that scheduled the event.
2. $\mathfrak{v} \in \mathfrak{V}$ is the *trigger value* — the value to be associated with $\mathfrak{t}$ when the event occurs.
3. $\mathfrak{g} \in \mathbb{T}$ is the *event tag* — the logical time at which the event should occur.

Reactors: A reactor *rtr* is defined as a tuple $rtr = (I, O, A, S, N, P)$, where:

- $I \subseteq \Sigma$ is the set of *input ports.*
- $O \subseteq \Sigma$ is the set of *output ports.*
- $S \subseteq \Sigma$ is the set of *state variables.*
- $A \subseteq \Sigma \times \mathbb{N} \times \mathfrak{O}$ is the set of *actions.*
- $\mathcal{N}$ is the set of *reactions.*
- $P : \mathcal{N} \to \mathbb{N}$ is the *priority function* — an injective map from reactions to their priorities. Priorities are used to determine order of execution for reactions. The full Reactor model does not require this function to be injective, thereby allowing multiple reactions to have the same priority. This is a deliberate choice, as it allows non-determinism to be introduced if explicitly chosen. The Simple Reactor model requires reactions to have unique priorities, as this imposes a total order on them, which simplifies the proof of determinism.

[7]For tags t_1 and t_2: $t_1 \leq t_2$ if $v(t_1) < v(t_2)$ or $v(t_1) = v(t_2) \ \wedge \ m(t_1) \leq m(t_2)$.

Reactions: A reaction rcn is defined as $rcn = (D, T, B, D^{\vee}, H)$. Let $\mathcal{C}$ be the reactor that contains rcn, then:

1. $D \subseteq I(\mathcal{C})$ is the set of *dependencies* — ports that can be accessed when the reaction executes.

2. $\mathcal{T} \subseteq D \cup x(A(\mathcal{C}))$ is the set of *triggers* — ports and actions that cause the execution of the reaction's body, when set to a non-absent value.

3. B is the *body* of the reaction, which is opaque executable code.

4. $D^{\vee} \subseteq O(\mathcal{C})$ is the set of *antidependencies* — ports that can be written to when the reaction executes.

5. $H \subseteq x(A(\mathcal{C}))$ is the set of *schedulable actions* — actions for which the reaction can generate events.

Reactor Networks: In the full Reactor model the mechanism for creating a *network* of reactors is given by a reactor's ability to define *nested reactors*. As we do not account for this feature in the Simple Reactor model, we need to define a separate mechanism for creating networks. Hence, we define a reactor network σ as a graph structure $\sigma = (R, E)$, where:

1. R is the set of *reactors*.

2. $E \subseteq \Sigma \times \Sigma$ are the directed *edges* connecting reactors' ports.

1.2.1 Execution Model

The execution model as presented in [1] is defined by a given set of procedures. Here we present the execution model *visually* instead of *algorithmically* for two reasons:

- The procedures in [1] need to handle many features (like physical actions) which we have omitted in the Simple Reactor model. Any attempt at reducing the algorithms would retain no more than a vague correspondence.

- An algorithm that uses the definitions stated above, would diverge greatly from the rigorous algorithms presented in Section 3, and thus have little utility.

Prerequisites: We define computation for reactor *networks*. Our definition is rudimentary in that we define no real starting point. That is, we simply begin computation upon whatever state our given reactor network is in, without any previous setup.[8] To perform computations, we require three additional objects:

[8]This is in contrast to the full Reactor model, which defines initialization actions.

- A global tag — since we only consider logical time, this is our only time keeping mechanism.
- An event queue — a list of events to be processed, ordered by their tags. As actions are scheduled, events are added to this queue. Execution is complete when this queue is empty.
- A precedence graph — a graph that manifests the dependencies between reactions in the network. This is used to determine the order in which reactions should be executed.

Let Figure 2 illustrate our reactor network. We have three reactors, $R1$, $R2$ and $R3$ with a total of five reactions. $R1$ can propagate data from its output ports, to $R2$ and $R3$. $R2$ can propagate data back to itself, as well as to $R3$. The only reactor that uses actions is $R1$.

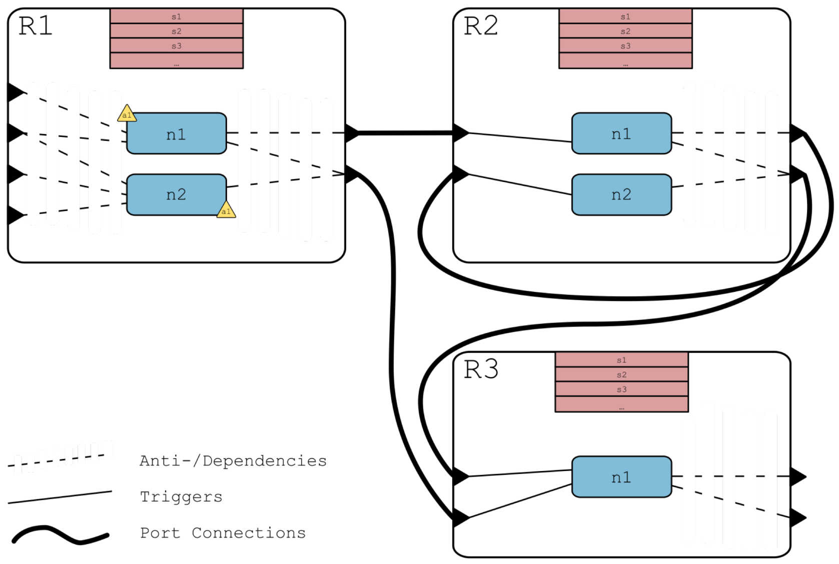

Figure 2: Reactor network

The precedence graph for this network is illustrated in Figure 3.[9] We obtain this graph by placing edges between reactions according to their priorities within a reactor, as well as the connections between their anti/-dependencies. Notably, our precedence graph contains no cycles. This is an important restriction of the Reactor model: we only define computation for reactor networks that have acyclic precedence graphs. Without this restriction, it wouldn't be possible to deterministically define an order of execution, as each reaction would depend on some other reaction before it could execute. With an acyclic precedence

[9]An algorithm for computing precedence graphs is presented in [1].

graph, we define reactions' order of execution as a topological ordering over the precedence graph.[10] Thus, we don't require the precedence graph itself for execution, but rather a topological ordering ($\mathcal{Q}_R$) over it:

$$\mathcal{Q}_R = [R1.n1,\ R1.n2,\ R2.n1,\ R2.n2,\ R3.n1]$$

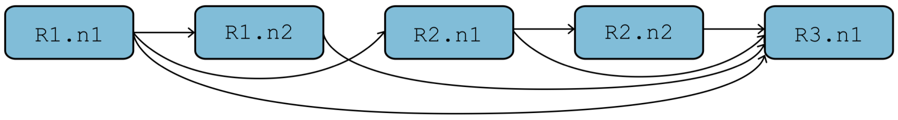

Figure 3: Precedence graph

Execution: Using these prerequisites, we can describe how to execute a reactor network.

1. Increase the global tag to match the next event e in the event queue. If there is none, execution has completed.
2. Run each reaction in the order defined by $\mathcal{Q}_R$:
   (a) Check if the reaction is triggered by its dependencies, i.e. if at least *some* trigger is non-absent.
   (b) If so, run the reaction's body. If not, advance to the next reaction.
   (c) For each action scheduled during the body's execution, add a corresponding event to the event queue. Actions can only be scheduled with a tag *greater* than the current one.[11]
   (d) For each output port written to by the reaction's body, propagate the values to any connected ports.
3. Clear all ports, dequeue all events for the current tag, and go to Step 1.

Example: We can visualize these steps with an example. Let Figure 4 be the initial state of our reactor network. That is, our current tag is $(4, 2)$, we have one event in the event queue and all ports are set to ϵ.

Figure 5: Our first step is to increase the tag to $(5, 0)$, as that's the tag of the next event. We then start executing the reactions with $R1.n1$. We assume here, that $a1$ belongs to its triggers and thus $R1.n1$ triggers. Thus, $R1.n1$'s body executes and aside from its port-dependencies (which are all ϵ), it receives the value V specified for $a1$ at the current tag.

[10] If it were not acyclic, such an ordering would not exist.

[11] An action's event's tag is a result of the *delay* specified by the action. If a delay of 0 is specified, then a microstep delay of 1 is added.

Figure 6: $R1.n1$ cannot schedule any actions (according to the illustration), but we assume here that $R1.n1$'s body produces outputs W and X for its ports respectively. After propagating those values to all connected ports we progress to the execution of reaction $R1.n2$.

Figure 7: $R1.n2$ does not trigger for the given state, so we progress to $R2.n1$ immediately. $R2.n1$'s trigger is non-absent, so we execute it, producing output values Y and Z.

Figure 8: $R2.n2$ triggers, as its trigger is non-absent. Assuming the reaction outputs some value O, this values overrides the current value Z of its output port and is propagated to $R3$.

Figure 9: Lastly, $R3.n1$ triggers, as at least one of its triggers (and in fact both) is non-absent. We assume here that it does produce any output during execution. Thus, we have run all reactions and progress to Step 3 of execution. Here we set all ports to ϵ again and remove all events with tag $(5, 0)$ from the event queue.

The resulting event queue is empty and we have fully executed this network.

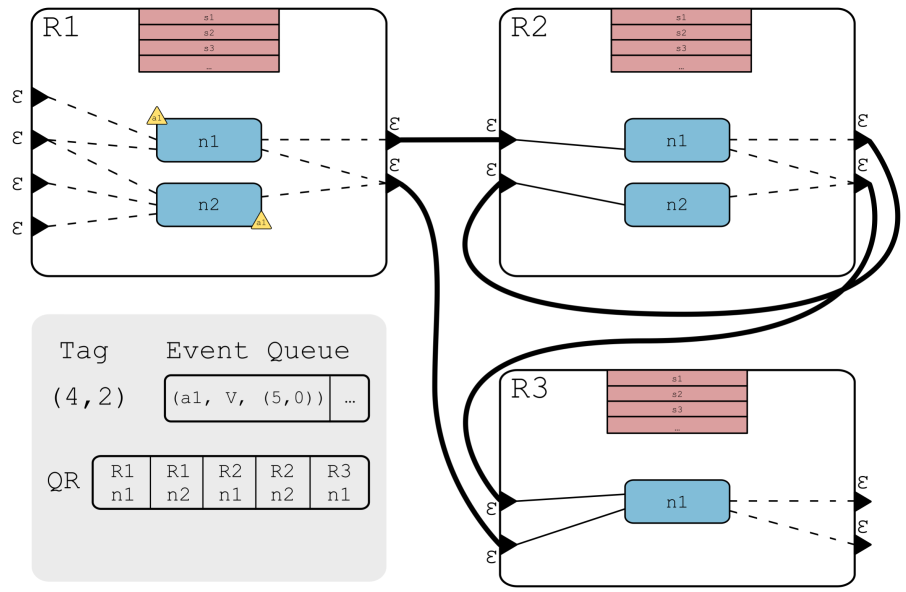

Figure 4: Execution example — initial state

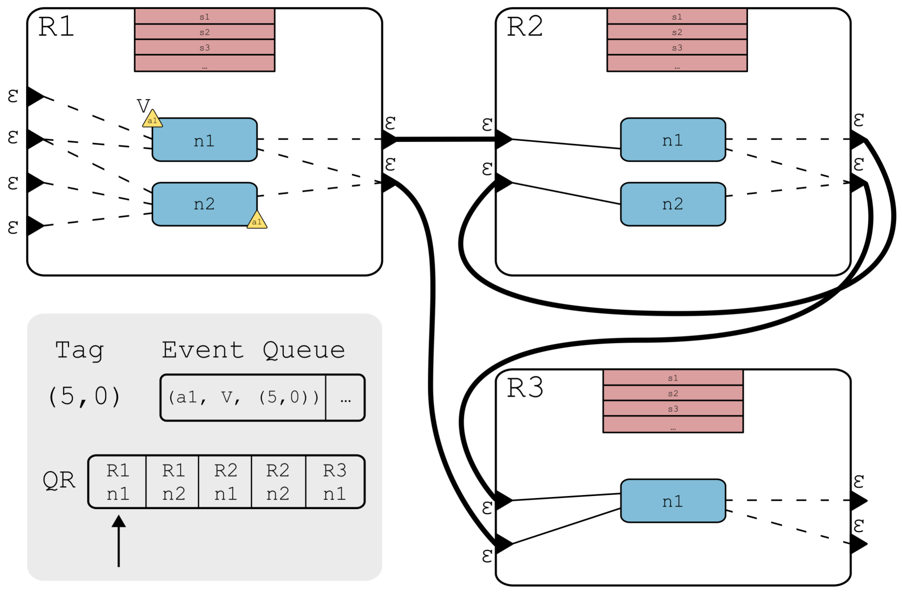

Figure 5: Execution example — right before $R1.n1$'s execution

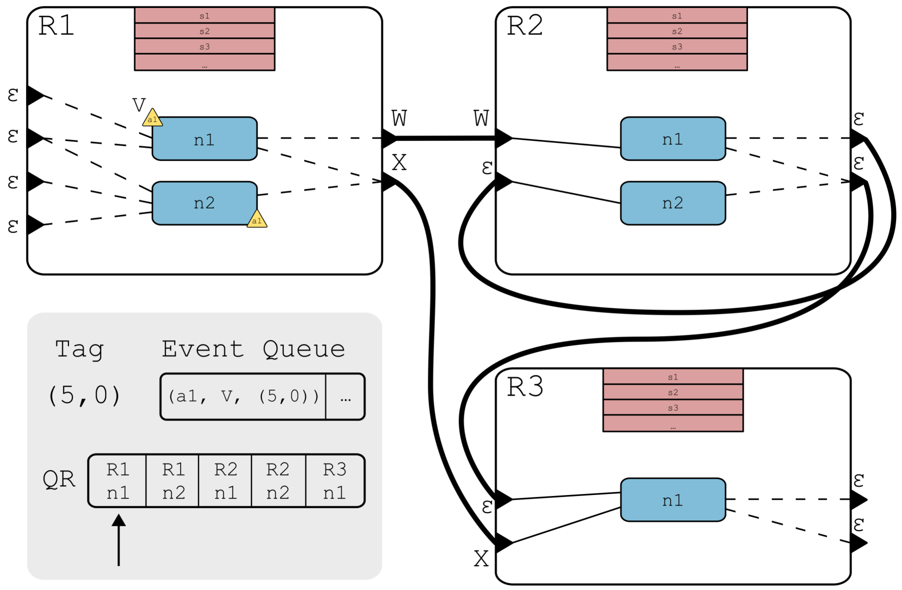

Figure 6: Execution example — after $R1.n1$'s values have been propagated

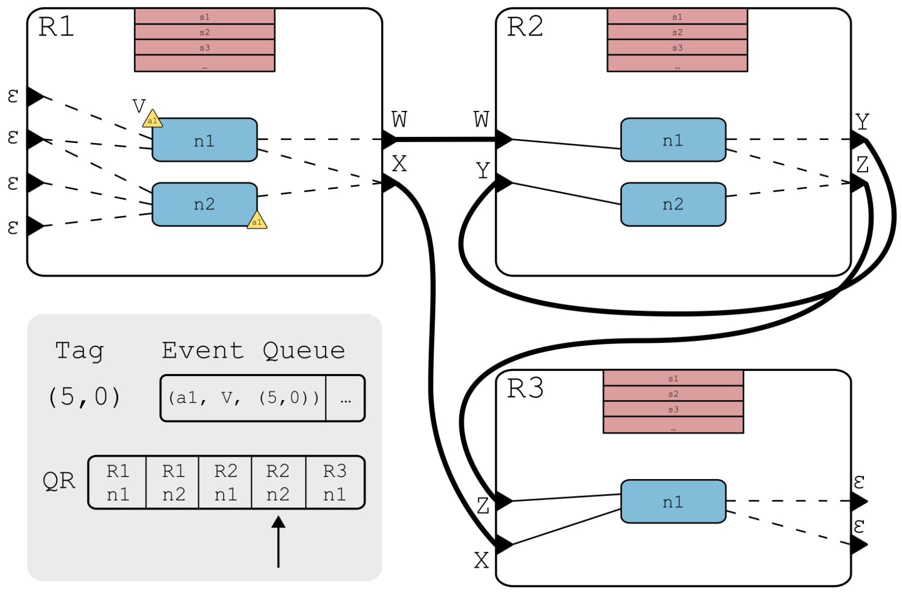

Figure 7: Execution example — right before $R2.n2$'s execution

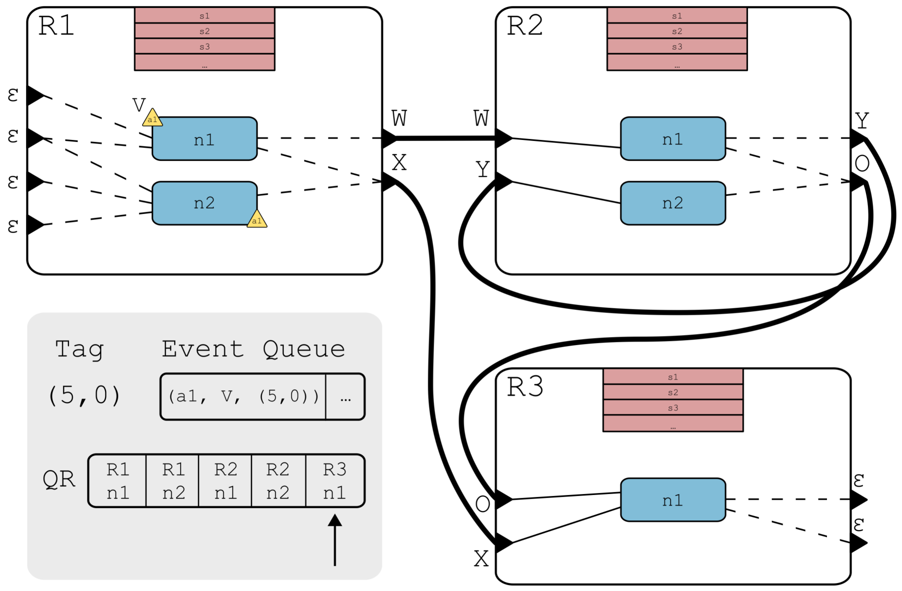

Figure 8: Execution example — right before $R3.n1$'s execution

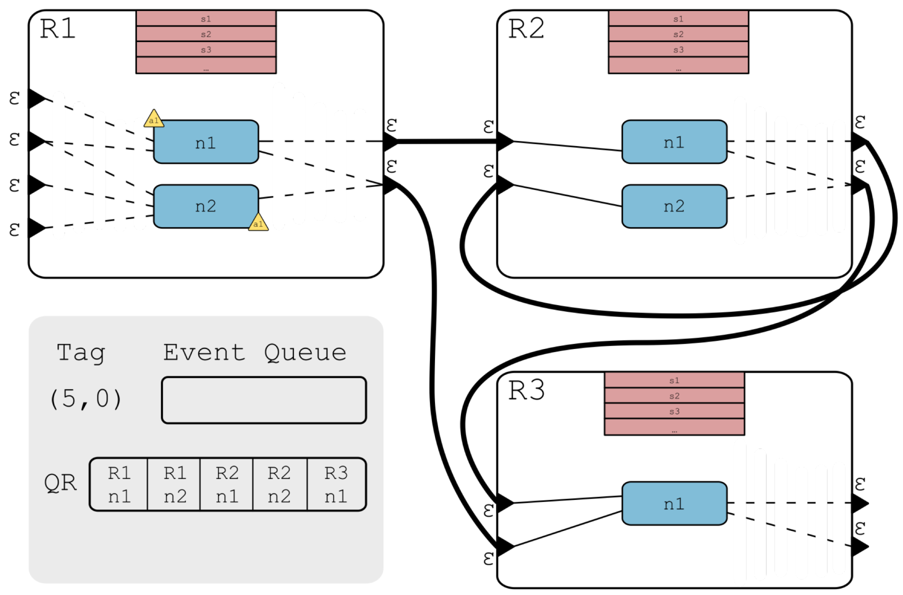

Figure 9: Execution example — end of execution

1.2.2 Inaccuracies

Despite our reduction from the full Reactor model, the formalization presented above has some issues that need to be addressed:

- The definitions of reactors and reactions are circular.[12]
- The concept of *opaque* objects used for identifiers, values and reaction bodies is not well-defined. A rigorous formalization will need to back this up with something concrete.
- The actual "position" of values ($\in \mathfrak{V}$) is undefined. That is, there exists no map from identifiers to values. It is therefore impossible to obtain the values of components like ports or state variables. This issue has been addressed in [4] by defining affected fields, like a reactor's ports, as Cartesian products of identifiers and values.
- The definition of a reaction's body B has the aforementioned problem of being opaque. For *this* object especially, the opaqueness hinders us from creating any rigorous definitions of computation in a reaction — and hence in the Simple Reactor model in general. For example, it is not clear how reactions access the values associated with actions/events.

[12] In [1], the circularity isn't as direct. There, the interdependency of definitions goes reactors $\rightarrow$ reactions $\rightarrow$ container function $\rightarrow$ reactors.

- The definition of a reactor's reactions $\mathcal{N}$ as a *set* disallows multiple identical reactions from living in the same reactor. Though this is not a problem from a mathematical standpoint, we consider it to be undesirable. As an example, consider Figure 10. Let the reactions' names (1, 2, 3) also be their priorities. Reactions 1 and 3 have the same dependencies, antidependencies, triggers and schedulable actions (none). If we also assume their bodies to be equal, the reactions are identical. The only differentiating factor are their priorities. As a reactor's set of reactions does not include the priorities, reactions 1 and 3 would collapse into a single element. Thus, our current model does not allow us to create the depicted reactor.

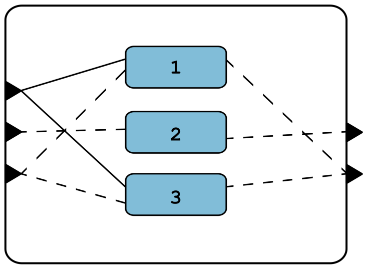

Figure 10: Example of identical reactions in the same reactor

Garbage In, Garbage Out: We would like to emphasize here that the following aspect of the model is *not* an issue: we define no restrictions on the sets of identifiers in a reactor. That is, a reactor's input ports I, output ports O and state variables S can contain *any* identifiers. This could be considered an issue, as a reactor could contain an identifier i with $i \in I$, $i \in O$ *and* $i \in S$. Instead, it is rather representative of an approach to formalization used throughout this thesis. It can be summarized by a principle called "Garbage In, Garbage Out" (GIGO), by which we mean: If one defines nonsensical objects, like a reactor where a single identifier is used for multiple components, then it should not be expected that it behaves sensibly. This approach has the benefit of allowing us to focus an object's definition on what we *want* its properties to be, without having to be precise about the properties it should *not* have. Hence, the definitions above can be viewed as necessary but not sufficient for well-behaved reactors.

2 Lean Theorem Prover

A stated goal of this thesis is to provide a well-defined formalization of the model presented above. In this section we introduce the tool that will help us achieve this goal: *Lean Theorem Prover*.

2.1 History of Proof Systems

In modern pen-and-paper mathematics, there is a common way for how we "do mathematics". We don't construct an entire system of syntax and reasoning ourselves, but rather use predicate logic and the standard axioms declared by Zermelo-Fraenkel set theory. Thus, our basic premise is *everything is a set*. When set theoretic foundations were first formulated by Georg Cantor and Gottlob Frege in the late 19th century, they contained paradoxes[13]. As a result, the early 20th century saw a host of new ideas to solve these problems during the *foundational crisis of mathematics*. Notably, Russel's and Whitehead's *Principia Mathematica* attempted to solve the problems of previous theories by introducing a hierarchy of "types" [6]. Zermelo set theory[14] was being developed around the same time and has become the prevalent theory used today. This did not put an end to research on mathematical foundations though. In particular, the idea of types stuck around. In the 1940s it was incorporated into Church's *Typed Lambda Calculus*[15] and in 1972 Per Martin-Löf published his *Intuitionistic Type Theory* [7] as an alternative foundation of mathematics. Not least the conception of programming languages introduced new applications for types in the form of *type systems*. Thus, types could be used to formalize mathematics *and* help us communicate precisely with computers. It would thereby stand to reason that computers could help humans with mathematics.

Computer Assisted Proofs: Computers could help humans by their raw compute power in what is called a *computer assisted proof*. For example, if a mathematical proof required checking a large (finite) number of cases, a computer could do that comparatively quickly. The canonical first example of using this approach was the proof of the Four Color Theorem in 1976. Arguably this type of computer assistance is the least helpful for coming up with new ideas, as the human already needs to know all the details of a proof.

Proof Assistants: A more potent means of computers assisting humans are *proof assistants* or *interactive theorem provers*. To use a proof assistant, the desired mathematical statements have to be written in a specific machine-parsable syntax. A statement can then be proven by specifying a sequence of axioms, substitutions, proven theorems, etc. For example, the following pseudo-code could specify a proof of the statement $(1+2)+3 = 2+(1+3)$:

[13]The canonical example is Russel's paradox in Frege's *Naive set theory*.
[14]This later became Zermelo-Fraenkel set theory.
[15]Cf. Section 2.2.1.

```
-- apply commutativity of addition to the first term on the
-- left side of the equation
use(left, first) nat_add_comm

-- apply associativity of addition to the left side of the
-- equation
use(left) nat_add_assoc

-- constuct a proof of the statement as a result of equality
-- being reflexive
cons eq_reflexive
```

The proof assistant then fulfills two main functions:

1. It checks whether the given sequence of expressions constitutes a valid proof of the desired statement.
2. It interactively visualizes the state of the current proof. This includes showing which remaining sub-statements have to be proven to complete the proof, the assumptions that have been introduced so far, etc.

Proof assistants still require humans to come up with the entire proof, but they provide certainty about the proof's correctness and aid the thought process. There are several notable examples of proof assistants, reaching as far back as the 1970s. These days, typical examples are *Agda*, *Coq*, *Idris* and *Lean*.[16]

Automated Theorem Provers: The next step in computers helping us create proofs would be for them to write the proofs themselves, i.e. perform *automated* theorem proving. Currently, the term *automated theorem prover* is used somewhat loosely to also describe proof assistants, as "real" automated theorem provers have not yet been developed. Fortunately, this ambiguity may be warranted as proof assistants are becoming smarter.

Proving a theorem in a primitive proof assistant can be laborious, since every step of the proof has to be meticulously specified. As proof assistants are becoming more sophisticated, humans are allowed to gloss over more trivial details. The proof sequence shown above is an example of a meticulously specified proof. Modern proof assistants can handle all of this term rewriting themselves and could accept a statement like `normalize` as a full proof.

One of the specific goals of the proof assistant used in this thesis is to be even better at these kinds of tasks. Lean aims to bridge the gap between interactive and automated theorem proving.

[16] All of these systems use the concept of *dependent types* (cf. Section 2.2.2).

2.2 Lean's Foundations

For this thesis we use Lean version 3. There are three aspects of Lean that are of interest to us. Namely, how to ...

- Formalize mathematical objects
- Formalize algorithms on those objects
- Construct proofs about those objects and algorithms

All of these aspects are covered by only three fundamental notions:

- Π-Types
- Type Universes
- Inductive Types

To see where these notions come from, we have to take a closer look at Lean's mathematical foundations: the *Calculus of Inductive Constructions* [8, 9], or CIC for short. For the most part, Sections 2.2.1 and 2.2.2 are a summary of information presented in Mario Carneiro's MS Thesis presentation on the type theory of Lean [10].

2.2.1 Typed Lambda Calculus

CIC is derived from *Lambda Calculus*. We assume that readers of this paper have at least some familiarity with it and therefore only give an overview of its concepts here.

In lambda calculus a *term* can be one of the following:

- a variable: x
- a lambda abstraction: $(\lambda x, t)$ — where x is a variable and t is a term
- an application: $(t\ s)$ — where t and s are terms

A notion of computation is achieved by performing *reductions* on terms — most notably the *β-reduction*. If we have a term $(\lambda x, t)s$ then the β-reduction allows us to rewrite it as $t[x/s]$ i.e. by replacing every occurrence of x in t with s. If repeated β-reduction yields a term that cannot be further reduced, it has reached *β-normal form*. This normal form can serve as a notion of equality of terms. Importantly though, repeated β-reduction is not assured to terminate, and hence, a term may not have a β-normal form.[17]

A *Typed Lambda Calculus* extends the former with a notion of *types*. We can simply declare an identifier, like ι, to be a type — called a *base type*. Or, if σ

[17] For example, the *Y-combinator*: $\lambda f, (\lambda x, f(x\ x))(\lambda x, f(x\ x))$

and τ are types, then we can construct a *function type* $\sigma \to \tau$. From now on σ and τ are used to denote arbitrary types, and if a term t has type τ, we write $t : \tau$.

The type judgement for terms follows three rules:

- For each base type ι, we can assign a set of variables to have type ι.
- If $t : \sigma \to \tau$ and $s : \sigma$, then $(t\ s) : \tau$.
- If $t : \tau$, then $(\lambda x : \sigma, t) : \sigma \to \tau$.

As the last rule shows, we have extended the syntax of lambda abstraction to require that its variables declare a type (above: $x : \sigma$). This has strong implications on reductions and therefore computation. The β-reduction is now restricted, such that it can only be performed on a term $(t\ s)$, when $t : \sigma \to \tau$ and $s : \sigma$. The result is that *every* term has a β-normal form — or computationally speaking, every computation terminates.

The expressions used so far correspond closely to Lean's syntax.[18] Base types and unbound variables can be declared using the `constant` keyword. For example, we could declare that `α` be a type and `a` be an instance of that type:

```
constant α : Type*
constant a : α
```

Declaring constants is equivalent to declaring axioms, so we won't ever use this feature outside these examples. Functions and applications can be written exactly as above:

```
λ x : α, a -- function that maps every instance x of type α to a
f s        -- application of some function f to some term s
```

To give names to such expressions we use the `def` keyword:

```
def my_func := λ x : α, a
def my_val  := my_func a
```

As all terms are typed, so too are `my_func` and `my_val`. To check their types we can use the `#check` directive:

```
#check my_func -- prints "my_func : α → α"
#check my_val  -- prints "my_val : α"
```

2.2.2 Dependent Types

Dependent Types represent another extension on our current calculus. The goal when using dependent types is to extend the expressivity of functions. More specifically, given a function type $\sigma \to \tau$ we want to make it possible for the

[18] An easy way to try out these examples is to use Lean's web editor: https://leanprover-community.github.io/lean-web-editor/

type τ to depend on the given *term* of type σ. We call this a *dependent function type*, or Π-*type* and write $\Pi x : \sigma, \tau$ to denote the dependent function type from σ to τ where τ can depend on x. Using dependent functions changes the type judgement for lambda abstraction and application:

- If $t : \tau$, then $(\lambda x : \sigma, t) : (\Pi x : \sigma, \tau)$.
- If $t : (\Pi x : \sigma, \tau)$ and $s : \sigma$, then $(t\ s) : \tau[x/s]$ — that is, the type τ can refer to the term x, which is replaced by s upon application $(t\ s)$.

The syntax in Lean is again exactly the same:

```
def my_dep_func : Π x : τ, σ x := λ ...
```

This definition declares `my_dep_func` to be a dependent function from type τ to type `(σ x)`. The ellipses (...) are not a part of Lean's syntax, but are used throughout this thesis to omit redundant code.

As we have not introduced a way for types to refer to terms, i.e. we have no way of *computing* a type from a term, Π-types do not add any functionality yet. We can fix this with the following adjustments.

Types as Terms: Our current calculus differentiates between *terms* and *types*, and defines functions to compute on terms. If we loosen the first restriction, and say that every type is a term, we gain the power of performing computations at the type-level. That is, we keep the current calculus, but allow a type to be used wherever we previously expected a term. For example, we can define the function $\lambda x : \sigma, \tau$ which for any input x returns not a term of type τ, but the type τ itself. This begs the question: what is the type of this function?

Universes: Say we wanted to write a function that returns exactly the type that it received as input — an identity function. Since we require a function's input to be typed, the application $(\lambda x : \sigma, x)\ \tau$ will never succeed. That is, we have no type σ such that $\tau : \sigma$. To solve this, we need to introduce a type of types, which we call a *universe* ($\mathcal{U}$). We can then define the identity function as $\lambda x : \mathcal{U}, x$. To avoid paradoxes that can arise by using just a single universe[19], we define a countable hierarchy of universes. We denote the lowest universe with $\mathcal{U}_0$ and define that for all universes $\mathcal{U}_n : \mathcal{U}_{n+1}$. Hence, $\mathcal{U}_n$ "contains" all types that live in universes $\mathcal{U}_m$ with $m < n$. We call these subscripts the *universe level*. The universe at level 0 has a special role, which will be covered in Section 2.2.4.

In Lean, we can define the aforementioned identity function as:

```
universe u
def identity : Type u → Type u := λ x : Type u, x
```

That is, `Type u` is the universe at universe level `u`. Alternatively, we can make the universe declaration implicit and use a more common function syntax:

[19]For example, Girard's Paradox.

```
def identity (x : Type*) : Type* → Type* := x
```

Or even drop the type annotation:

```
def identity (x : Type*) := x
```

Definitional Equality: A problem that arises with the ability to compute types is, what happens when two types look different but are actually the same. For example, suppose $\tau : \mathcal{U}_1$ and $id := (\lambda\sigma : \mathcal{U}_1, \sigma)$. Then, should τ and $(id\ \tau)$ be considered the same types? The answer given in dependent type theory is, yes — we call them *definitionally equal*. The details of definitional equality are subject to a host of rules, but can loosely be summarized as:

- Two types are definitionally equal if one is β-reducible to the other.
- Definitional equality is an equivalence relation.

2.2.3 Inductive Types

The last concept required for CIC are *inductive types*. These types are rather complex to define mathematically [10, p. 9], so we introduce them only through Lean.

The syntax we've introduced so far is great for creating *functions*, i.e. descriptions of computations. The only mechanism we've seen for creating *new types* was the axiomatic `constant` declaration. The only other way of creating new types in Lean is as inductive types.

> Intuitively, an inductive type is built up from a specified list of constructors. In Lean, the syntax for specifying such a type is as follows:
>
> ```
> inductive foo : Type u
>   | constructor₁ : ... → foo
>   | constructor₂ : ... → foo
>   ...
>   | constructorₙ : ... → foo
> ```
>
> The intuition is that each constructor specifies a way of building new objects of `foo`, possibly from previously constructed values. The type `foo` consists of nothing more than the objects that are constructed in this way. [11]

Constructors: Inductive types are best understood by example, so let's first consider Lean's product and sum types [11, p. 99], `prod` and `sum`:

```
inductive prod (α : Type*) (β : Type*)
  | mk (fst : α) (snd : β) : prod

inductive sum (α : Type*) (β : Type*)
  | inl (a : α) : sum
  | inr (b : β) : sum
```

```
#check prod -- "Type u_1 → Type u_2 → Type (max u_1 u_2)"
#check sum  -- "Type u_1 → Type u_2 → Type (max u_1 u_2)"
```

The `prod` type is a function from two types (α and β) to some new type: their product type. The type defines exactly one way to create instances for it — we say it has exactly one *constructor*. The constructor `mk` is a function that expects an instance of α and β and returns a corresponding instance of `prod α β`. If, for example, we have two natural numbers `x` and `y`, we can create an instance of their product type (a tuple of $\mathbb{N} \times \mathbb{N}$):

```
#check (prod.mk x y) -- "prod ℕ ℕ"
```

The `sum` type on the other hand declares two constructors — one that takes an α and one that takes a β. Thus, we have two ways to create an instance of `sum α β`:

```
#check n -- "ℕ"
#check z -- "ℤ"

#check (sum.inl n) -- "sum ℕ ?M_1"
#check (sum.inr z) -- "sum ?M_1 ℤ"
```

The `?M_1` means that Lean cannot infer what the given type parameter is. We ignore this here.

Recursors: Bundling up data into new instances is of little use if we don't have a way to unpack those instances to access their data. This is what *recursors* are for. For example, if we want to add the two components of an instance of $\mathbb{N} \times \mathbb{N}$, we first need to unpack the tuple and then add the respective values:

```
def my_add (t : prod ℕ ℕ) : ℕ :=
  prod.rec (λ n₁ n₂, n₁ + n₂) t
```

Lean automatically generates recursors for inductive types. The `prod.rec` recursor is a function that expects two parameters. First, a function from the values contained in an instance of the product type (n_1 and n_2) to *some* other value ($n_1 + n_2$). Second, an instance of the product type (`t`) upon which this function should be applied. The recursor of the `sum` type differs from that of `prod`, as `sum` has multiple constructors:

```
def negate_if_left (s : sum ℤ ℤ) : ℤ :=
  sum.rec (λ l, -l) (λ r, r) s
```

The recursor `sum.rec` expects *two* functions as parameters — one for each constructor of the sum type. Hence, we can perform different functions based on which kind of instance of `sum` we get. This is akin to `switch`-statements over enums in C-like programming languages.

Recursive Types: The `prod` and `sum` types don't show the full capabilities of inductive types. They are "flat", i.e. they only combine other types into new ones. The real power of inductive types becomes evident when we make them recursive. For example, consider the type of natural numbers `nat`:

```
inductive nat
  | zero : nat
  | succ (n : nat) : nat
```

This type shows two new kinds of constructors:

- `zero` is a constructor that doesn't expect any parameters. Hence `nat.zero` is a truly new value by its own right.
- `succ` is a *recursive* constructor, i.e. it tells us how we can construct a new `nat` from a given one. In this example it tells us that for a given `n : nat` we can create its successor `nat.succ n`.

The result is that we get a formalization of the natural numbers that is analogous to that of the *Peano axioms*. The combination of atomic and recursive constructors allows us to create an infinite number of new values — in this case all natural numbers.

2.2.4 Curry-Howard Isomorphism

The main motivator for using Lean in this thesis is as a tool for theorem proving. So far we have not discussed how to state a proposition or a proof. In this section we explore the unique properties of propositions in CIC and show how to express them in Lean.

In Section 2.2.2 we referred to the "special role" of the universe at level 0. This lowest universe is special in that it contains only propositions and all propositions. Thus, if we have a proposition p, we know $p : \mathcal{U}_0$ — in Lean's syntax `p : Prop`. This has an important implication: *Propositions are types*. This property and especially its consequences (covered below) are known as the *Curry-Howard Isomorphism*. The benefit of propositions being types is that it allows us to model proofs as instances of their respective propositions. Say we have the proposition $(p \wedge q) : \mathcal{U}_0$. By the propositions-as-types paradigm, we prove this proposition by constructing an instance for it. And just as with other types we've seen so far, this can be done by using one of the type's constructors:

```
#check p -- "p : Prop"
#check q -- "q : Prop"
#check pᵢ -- "pᵢ : p"
#check qᵢ -- "qᵢ : q"

def my_proof : and p q := and.intro pᵢ qᵢ
```

Here `p` and `q` are propositions, and $\texttt{p}_i$ and $\texttt{q}_i$ are instances (proofs) of those propositions. To create a proof of `and p q` we create an instance of this type using `and`'s constructor `intro`, which expects a proof of its respective sub-propositions. Thus, in Lean, proofs can be used as data, just like numbers and lists.[20] Propositions are also defined like any other type:

```
inductive and (a b : Prop) : Prop
  | intro (left : a) (right : b) : and

inductive or (a b : Prop) : Prop
  | inl (h : a) : or
  | inr (h : b) : or
```

In fact, if we compare these definitions with those of `prod` and `sum` we notice that they are almost exactly the same — the only difference being that these propositions' parameters (`a` and `b`) are themselves propositions instead of *any* type. This observation is one of the key aspects of the Curry-Howard isomorphism, as it allows us to draw the following correspondence:

Logic	Types
$\wedge$	product type
$\vee$	sum type
$\implies$	function type
$\forall$	Π-type
$\exists$	Σ-type
$\top$	unit type
$\bot$	empty type

Figure 11: Correspondences in the Curry-Howard isomorphism

A unit type is a type that has exactly one instance, whereas an empty type has no instances. A Σ-type is a dependent product type. These types are exactly how `true`, `false` and `Exists` are implemented in Lean:

```
inductive true : Prop
  | intro : true

inductive false : Prop

inductive Exists {α : Type*} (p : α → Prop) : Prop
  | intro (w : α) (h : p w) : Exists
```

We won't review the other correspondences here, as they should become evident during the course of this thesis.

Proof Irrelevance: Another unique aspect of propositions in CIC is their *proof irrelevance* [10, pp. 6-7]: All proofs of a proposition, and therefore all instances of it, are considered equal. This is achieved by defining any two instances of the same proposition to be definitionally equal. As a result, we can never depend on the specific objects used to construct a proof, but rather use the proof as a certificate.

[20] There is the caveat of proof irrelevance, as covered below.

For example, we can prove the proposition (`∃ n : ℕ, n = n`), by providing the instance `Exists.intro 5 (refl 5)`, where `refl 5` is a proof that `5 = 5`. If we use an instance of this proposition though (say, in a proof), we cannot rely on `n` being `5`, since we might as well have received a definitionally equal proof: `Exists.intro 4 (refl 4)`.

2.3 Dependent Type Hell

Now that we've covered the formal aspects of Lean, we also need to consider an informal yet important facet of Lean called *Dependent Type Hell*. As Kevin Buzzard[21] puts it:

> It seems that in dependent type theory one has [to] think about [formalization] in a certain kind of way in order to avoid pain. [12]

To demonstrate what exactly is meant by "pain" in this context, we show an example of dependent type hell and how to circumvent it. For this purpose, we formalize one of the basic objects required for the Simple Reactor model: a digraph. We place two requirements on our formalization, which are not present in conventional digraphs:

- It should be possible for the same vertex value to occur multiple times in the graph. For example, if our digraph connects elements of ℕ it should be possible for 45 to occur multiple times. We achieve this by using *identifiers* as vertices and define a mapping from IDs to values.
- Aside from being directed, edges should also be able to carry arbitrary data. For example, if we form a graph of reactors, it should be possible for the edges to specify which ports they connect. We achieve this by making digraphs dependent on an (almost) arbitrary edge type.

Thus, we will not formalize a simple digraph, but rather a labeled multidigraph, which we here call "L-graph". We use this opportunity to introduce more of Lean's concepts and syntax:

```
structure lgraph (ι δ ε : Type*) :=
  (ids : finset ι)
  (data : ι → δ)
  (edges : finset ε)
```

The `structure` keyword can be used when defining an inductive type with a single constructor, so that `ids`, `data` and `edges` can be thought of as the "fields". The `finset` type represents finite sets over a given type. The parameters ι, δ, ε are the L-graph's generic ID, data, and edge types. This definition of `lgraph` is not complete in that the edge type ε is completely unrestrained. A *directed* edge should have a concept of *source* and *destination*.

[21] Active member of the Lean community and professor of pure mathematics at Imperial College London.

Type Classes: To enforce this restriction, we use *type classes* — a concept first introduced in the Haskell programming language. Type classes are a concept akin to what is known as interfaces, traits or protocols in mainstream programming languages — though the analogy breaks quickly:

```
class edge (ε ι : Type*) :=
  (src : ε → ι)
  (dst : ε → ι)
```

Type classes are declared with the `class` keyword, which is syntactic sugar for an inductive type declaration with a certain attribute. A type can become part of a type class, by providing an instance for it. For the sake of brevity, we won't explain instances further.

Thus, for any type ε to conform to the type class `edge`, it needs to provide a corresponding instance, which allows us to extract a source (`src`) and destination (`dst`). We can force our L-graph's edge type ε into conforming to `edge` with the following notation:

```
structure lgraph (ι δ ε : Type*) [edge ε ι] := ...
```

Going to Hell: To demonstrate dependent type hell, we review an extended version of `lgraph`. The aim of this extension is to constrain `lgraph`, such that:

1. The `data` map does not have to define corresponding data for *all* possible IDs (instances of ι), but only those which are part of the graph (`∈ ids`).
2. The edge type ε should be able to depend on the graph's `ids` and `data`. This information *could* be required by the edge type to make sure that no invalid edges can be constructed.

We can achieve the first goal by restricting the domain of the `data` map using a *subtype*. A subtype is a mechanism for restricting a given type, such that all of its instances have to satisfy a given predicate.[22] The subtype of all ι that are an element of `ids` can be written as `{i : ι // i ∈ ids}`.

The second goal can be realized by extending the type of edges ε. Thus, ε should not be a `Type*`, but rather a dependent function (Line 2) that computes the edge type as a function of a set of IDs and a map of IDs to data. An L-graph's `edges` (Line 8) are then a set over the edge type that we get by passing the graph's own `ids` and `data` to the edge type function. That is, `(ε ids data)` is the type of edges in an L-graph.

The `variable(s)` keyword allows us to factor out the parameters of a type. Thus, ι, δ, ε and the type class constraint are still all part of `lgraph`'s definition:

```
1  variables ι δ : Type*
2  variable ε : Π ids : finset ι, ({i // i ∈ ids} → δ) → Type*
3  variable [Π i d, edge (ε i d) ι]
4
```

[22]This is similar to the principle of restricted comprehension in set theory.

```
structure lgraph :=
  (ids : finset ι)
  (data : {i // i ∈ ids} → δ)
  (edges : finset (ε ids data))
```

While these constraints on L-graph aren't problematic themselves, they create problems when working with L-graphs. As an example, consider what happens when we try to remove a vertex from the graph:

1. We remove the corresponding ID from the graph's `ids`.
2. Since the `ids` have changed, the *type* `{i // i ∈ ids}` has also changed. Hence, we need to construct a new `data` map to match the new type.
3. Since the `ids` and `data` have changed, the *type* `(ε ids data)` has also changed. Rectifying this issue is non-trivial, since it means that we have to type-cast every edge in `edges` to the new edge-type.

Thus, not only is it non-trivial to perform a relatively simple operation, it also changes the *type* of the resulting L-graph's edges. This is a strong indicator that we have entered dependent type hell.

Escaping Hell: We've explained previously that in Lean, proofs can be used as first-class data. One of the applications of this is to ensure properties about values ad hoc. For example, if we have some instance `n : ℕ`, and also have a proof that `n = 0 * n`, then we've obtained a constraint on `n` without requiring the type of `n` to be constrained (for example, as `{n : ℕ // n = 0 * n}`). Thus, we can navigate our way out of dependent type hell, by adding the following field to our original definition of L-graphs:

```
structure lgraph (ι δ ε : Type*) [edge ε ι] :=
  ...
  (well_formed:
    ∀ e ∈ edges, (edge.src e) ∈ ids ∧ (edge.dst e) ∈ ids)
```

The type of `well_formed` is the proposition that all edges have a source and destination that belongs to the set of `ids`. Adding this field implies that any instance of an L-graph must come with an instance of this proposition, and hence a proof that its edges are formed over its IDs. Hence, it is only ever possible to instantiate "valid" L-graphs — all without constraining its edge type.

This section has shown how Lean and the underlying Calculus of Inductive Constructions provide powerful mathematical foundations, which allow us to formalize mathematical objects, functions on those objects, and construct proofs about those objects and functions. Thus, in the following sections we will use the basic notions of universes, Π-types and inductive types to formalize the Simple Reactor model. Yet, as we have just seen, we need to be careful in how we wield these tools.

3 Formalizing Instantaneous Reactors

Despite the Simple Reactor model being relatively small, formalizing it and especially proving properties about it, is an extensive undertaking. To achieve a clean separation of concerns, we split the implementation into two parts:

- In this section we implement the instantaneous components of the model, i.e. everything that does not involve time. We call this subset the *Instantaneous Reactor* model. In Section 4 we prove determinism only for the Instantaneous Reactor model.
- In Section 5 we add the time-based aspects of the Simple Reactor model as a generalization of the Instantaneous Reactor model. That is, we add them *on top* and do not make any changes to the Instantaneous Reactor model in the process.

3.1 Primitives

One issue in the Simple Reactor model is circular definitions. To avoid this in our formalization, it is important to be clear about which primitives are used by other objects in their definitions. For example, if ports are a component of reactors, and reactions also reference ports in their definition, then having reactions as a part of reactors makes the definitions circular. Hence, we first define some primitives that are used by other objects.

Values ($\in \mathfrak{V}$) are supposed to be opaque, but have one requirement: they should have a notion of equality. To achieve this, we don't define a type of values explicitly, but instead make all other definitions dependent on some type parameter υ. That way, it is impossible for any definitions that use υ to depend on *concrete* instances of υ — i.e. the values are opaque. We ensure that this type is equatable, by constraining it to the type class `decidable_eq`:

```
variables (υ : Type*) [decidable_eq υ]
```

We can model a reactor's ports and state variables as lists over such values. Using names like `reactor.ports` is part of Lean's namespace syntax. Here we declare that the definitions of `ports` and `state_vars` are part of the namespace `reactor`:

```
def reactor.ports := list (option υ)
def reactor.state_vars := list υ
```

We account for the absent value ϵ, by using optional values. Wrapping any type in the `option` type makes the values optional, by adding a `none` value, which is generally used to represent the absence of a value. We assume here that ϵ is only intended for modeling absence of values in *ports*, and hence omit the optionality for state variables. The term *port assignment* is used throughout this thesis to mean *an instance of* `reactor.ports` υ. As an example of a port assignment, consider the following instance:

```
def p : reactor.ports υ := [some 1, none, some 2, some 1]
```

If `p` are the input ports of some reactor, then that reactor has exactly four input ports, where the value in the second port is absent.

Using lists for these components solves some inaccuracies of the Simple Reactor model. Firstly, we now know where their values live: in the lists. Secondly, we get the type of their identifiers for free, as we can just use the lists' indices ($\in \mathbb{N}$) as IDs. Since a reactor will hold two instances of `reactor.ports`, one for its inputs and one for its outputs, these IDs will in fact be ambiguous within a reactor. That is, an arbitrary port-ID `n` could refer either to the `n`th input port or the `n`th output port. This ambiguity will always be resolved by context.

It should be noted here that the length of these lists is not fixed, so we could change the number of a reactor's ports, by changing the number of elements in one of these lists. This can be avoided, by instead using a fixed-length vector type. The downside is that this leads to dependent type hell, so we opt for the GIGO-approach here by declaring that port assignments' length should not be changed during execution.

Many of the definitions of reactor components in this thesis are accompanied by auxiliary definitions. For example, we explicitly define fully absent `ports` and a related proposition:

```
def ports.empty (n : ℕ) : ports υ := list.repeat none n
def is_empty (p : ports υ) : Prop := (p = ports.empty υ p.length)
```

Definitions like these improve the comprehensibility of later definitions greatly. We will sometimes use such definitions without showing them, and instead resort to describing their functionality in prose.

3.2 Reactors & Reactions

Using `ports` and `state_vars` makes the definition of reactors straightforward:

```
structure reactor :=
  (input : ports υ)
  (output : ports υ)
  (state : state_vars υ)
  (priorities : finset ℕ)
  (reactions : ℕ → reaction υ)
```

The notable fields are `priorities` and `reactions`. One of the issues with the definition of reactors in the Simple Reactor model is that since the reactions ($\mathcal{N}$) are in a set, they all have to be unique. Additionally, to ensure that no two reactions have the same priority, the priority function (P) has to be injective. We can circumvent both of these problems, by switching the positions of priorities and reactions. That is, what was the set of *reactions* becomes the set of *priorities*, and the priority function becomes a reaction function, which maps the priorities to concrete reactions. Thus, we use a reaction's priority as an identifier for it. To avoid dependent type hell, we *don't* restrict the domain of the `reactions` function to be *exactly* the reactor's set of priorities. Instead, we require a function from *all* $\mathbb{N}$, but only ever employ the map for `n : ℕ` which are in the set of priorities.

Alternatively, we could use a partial function ℕ → `option (reaction` υ`)` and add a constraint that all `n` ∈ `priorities` must map to a non-`none` value. This causes inconvenience at the call-site though, as we have to unwrap the optional values.

This definition of `reactor` is only valid, if we've already defined `reaction`. That is, our definition of `reaction` cannot depend on `reactor`:

```
structure reaction :=
  (dᵢ : finset ℕ)
  (dₒ : finset ℕ)
  (triggers : finset {i // i ∈ dᵢ})
  (body : ports υ → state_vars υ → ports υ × state_vars υ)
  (in_con : ∀ {i i′} s, (i =dᵢ= i′) → body i s = body i′ s)
  (out_con : ∀ i s {o}, o ∉ dₒ → (body i s).1.nth o = none)
```

The sets $\mathtt{d}_i$ and $\mathtt{d}_o$ are the reaction's dependencies and antidependencies (D and $D^\vee$). Throughout this thesis we also refer to them as *input-* and *output-dependencies*. They contain IDs of their enclosing reactor's ports, i.e. indices into the reactor's `input` and `output` lists. The `triggers` are a subset of $\mathtt{d}_i$, defined as a set that contains instances of the subtype `{i // i ∈ dᵢ}`.

The first substantial change relative to the Simple Reactor model is made to a reaction's body (B). Our original definition declared it to be opaque code, which we have determined to be problematic. Here we define it as a *function* that maps from an input-port assignment and state variables to an output-port assignment and new state. We define the function in curried form, as this makes the call-site cleaner and easily allows partial application.[23] The symbol × in the target type is syntactic sugar for the `prod` type.

When we run a reaction as part of a reactor[24], its body receives the reactor's input-ports and state variables as inputs, and the body's outputs are incorporated into the reactor by a dedicated function. The `body` function has the option of not writing to an output-port, by returning the absent value for the specific port. That is, the returned instance of `ports` υ holds a value of `none` at the corresponding index.

To ensure that this definition of `body` behaves in a way that is analogous to that of the Simple Reactor model, we need to enforce some restrictions:

1. A `body`'s outputs can only depend on *those* ports which are part of its input-dependencies $\mathtt{d}_i$. Thus, we achieve the same behavior as if we were to restrict the domain of the function to be exactly $\mathtt{d}_i$. This restriction is enforced by the property `in_con`: Given two input-port assignments `i` and `i′` that have the same values for all ports in $\mathtt{d}_i$ (written as `i =dᵢ= i′`), the output of `body` has to be the same. For the purposes of this thesis, parameters that are enclosed by braces (here `{i i′}`) can be considered to be the same as parameters enclosed by parentheses.

[23]It is canonical to define functions in this form in Lean.
[24]Cf. Section 3.5.

2. A reaction should only be able to write to *those* ports which are part of its output-dependencies $\mathtt{d}_o$. Hence, the output-port assignment returned by `body` should be `none` for all ports that are not in $\mathtt{d}_o$. This is enforced by `out_con`: Given any inputs (`i` and `s`) and an output-port ID `o` which is not in $\mathtt{d}_o$, running the body must yield a value of `none` for `o`. The expression `(body i s).1.nth o = none` expresses precisely this last proposition.

3.3 Reactor Networks

As mentioned in Section 1.2, the Simple Reactor model has no notion of nested reactors. Instead, we build reactor networks using a global graph structure. We have previously laid the foundations for this by defining `lgraph`.[25] To construct a reactor network from an L-graph, we need to define which types the graph's *identifiers* and *edges* should have.

Identifiers: In `reactor` and `reaction` we used instances of ℕ for identifying reactions and ports. This works great as long as we only have *one* reactor. In a reactor *network*, we need to extend these IDs to also specify the reactor they live in:

```
def reactor.id := ℕ

structure reaction.id :=
  (rtr : reactor.id)
  (rcn : ℕ)

structure port.id :=
  (rtr : reactor.id)
  (prt : ℕ)
```

That is, every reactor in a network has a unique number ($\in \mathbb{N}$) associated with it. Reactions and ports can then be globally identified by a combination of their reactor and local ID.

Edges: While the vertices in a reactor network are reactors, the components that should be connected by edges are their *ports*. Hence, we need the edges in a reactor network to have a concept of which ports they are connecting:

```
structure inst.network.graph.edge :=
  (src : port.id)
  (dst : port.id)
```

We don't distinguish between types for input-port and output-port IDs, since we know that the `src` always refers to an output-port and the `dst` to an input-port.

Graph: An `inst.network.graph` is now just an `lgraph`, that uses `reactor.id` as its ID-type, `reactor` υ as its data-type and `inst.network.graph.edge` as its edge-type:

[25] Cf. Section 2.3.

```
def inst.network.graph : Type* :=
  lgraph reactor.id (reactor υ) inst.network.graph.edge
```

For convenience, we define some accessors on this structure:

```
def rtr (η : inst.network.graph υ) (i : reactor.id) : reactor υ
  := η.data i

def rcn (η : inst.network.graph υ) (i : reaction.id) : reaction υ
  := (η.rtr i.rtr).reactions i.rcn
```

Network: While this concrete graph *structure* is well-suited for our purposes, it is not sufficiently restrictive. There are two issues that need to be addressed:

1. It is currently possible to have multiple edges ending in the same port. This is explicitly forbidden in the Reactor model. As long as we don't define a global order on reactors (as we have with reactions in a reactor), this restriction is necessary for maintaining determinism during value propagation.

2. Ports can be connected in such a way that associated reactions can have cyclic precedence. In Section 1.2.1 we stated that execution is only defined for networks with acyclic precedence graphs. Here we disallow precedence-cyclic networks entirely.

Hence, our previous definition only defined a network *graph*. The definition of a complete instantaneous reactor network wraps such an `inst.network.graph`, and requires additional proofs about it:

```
structure inst.network :=
  (η : inst.network.graph υ)
  (unique_ins : η.has_unique_port_ins) -- addresses Issue 1
  (prec_acyclic : η.is_prec_acyclic)   -- addresses Issue 2
```

The first restriction is easy to implement. We require that all (non-equal) edges in the network have different destinations:

```
def has_unique_port_ins (η : inst.network.graph υ) : Prop :=
  ∀ (e ∈ η.edges) (e' ∈ η.edges), e ≠ e' → e.dst ≠ e'.dst
```

To implement the second restriction, we first need to formalize a notion of *precedence* in a reactor network.

3.4 Precedence Graphs

The intuition behind precedence in a reactor network is rather simple. A reaction r can take precedence over another reaction s if it satisfies at least one of the following conditions:

1. r and s live in the same reactor, and r has a higher priority than s.
2. A dependency of s is connected to an antidependency of r.[26]
3. r is transitively determined to take precedence over s.

While the first condition can be analyzed locally, the latter require a global view of dependencies. Hence, we define a *precendence graph*. A precedence graph connects *reactions*, such that the edges represent their immediate (non-transitive) precedences:

```
structure prec.graph.edge :=
  (src : reaction.id)
  (dst : reaction.id)

def prec.graph : Type* :=
  lgraph reaction.id (reaction υ) prec.graph.edge
```

Again, while this concrete graph *structure* is well-suited for our purposes, it is not sufficiently restrictive. We need a way of ensuring that its vertices and edges actually reflect the precedences for a given network graph. If this is the case, we call the precedence graph *well-formed* over a fixed instantaneous network graph:

```
1  def is_well_formed_over
2    (ρ : prec.graph υ) (η : inst.network.graph υ) : Prop :=
3      ρ.edges_are_well_formed_over η ∧
4      ρ.ids_are_well_formed_over   η ∧
5      ρ.data_is_well_formed_over   η
```

The three expression (Lines 3-5) are propositions about the three components of `lgraph`, and therefore `prec.graph`: edges, IDs and data.

Edges: The first property (Line 3) is the most interesting. It makes sure that the precedence graph contains edges between exactly those reactions that satisfy precedence conditions 1 or 2:

```
1  def edges_are_well_formed_over
2    (ρ : prec.graph υ) (η : inst,network.graph υ) : Prop :=
3      ∀ e : edge, e ∈ ρ.edges ↔
4        internally_dependent e.src e.dst ∨
5        externally_dependent e.src e.dst η
```

Lines 4 and 5 correspond to precedence conditions 1 and 2 respectively, and are not further explored here.

[26] r and s may or may not live in the same reactor.

Identifiers: The second property in `is_well_formed_over` (Line 4), expresses the restriction that a precedence graph must contain exactly those reactions (represented by their IDs) which are present in the network graph:

```
1  def ids_are_well_formed_over
2    (ρ : prec.graph υ) (η : inst.network.graph υ) : Prop :=
3      ∀ i : reaction.id, i ∈ ρ.ids ↔
4        i.rtr ∈ η.ids ∧
5        i.rcn ∈ (η.rtr i.rtr).priorities
```

Recall here that a reactor identifies its reactions by their priorities. Hence, Line 5 states that `i.rcn` should be a member of the reactor identified by `i.rtr`.

Data: We consider a precedence graph's *data* to be well-formed over a given instantaneous network graph η, if it corresponds exactly to the reactions in η. That is, the reaction-IDs in ρ have to map to the same reactions as in η. Hence, we explicitly ignore the *values* in the network graph, since they do not affect precedence.

```
def data_is_well_formed_over
  (ρ : prec.graph υ) (η : inst.network.graph υ) : Prop :=
    ∀ i : reaction.id, ρ.rcn i = η.rcn i
```

Thus, a precedence graph that is well-formed over a network graph η contains precisely all reactions present in η, while connecting exactly those reactions by edges that fulfill precedence conditions 1 or 2.

Transitivity: To find *transitive* precedences between reactions (precedence condition 3), we need to further analyze the precedence graph. Namely, if the precedence graph has a *path* from one reaction to another, then the former takes precedence over the latter. The notion of paths is defined directly on `lgraph`. If `r` and `s` are IDs and ρ is an L-graph, then the notation `r~ρ~>s` is used for the proposition that there is a path from `r` to `s` in ρ. By using paths to express transitive precedence, we can define the `is_prec_acyclic` property, required for `inst.network`:

```
def lgraph.is_acyclic (g : lgraph ι δ ε) : Prop :=
  ∀ i, ¬(i~g~>i)

def inst.network.graph.is_prec_acyclic
  (η : inst.network.graph υ) : Prop :=
    ∀ ρ : prec.graph υ, (ρ ⋈ η) → ρ.is_acyclic
```

The notation `(ρ ⋈ η)` stands for `(ρ.is_well_formed_over η)`. Thus, we require all well-formed precedence graphs over η to be acyclic.

3.5 Execution Model

Now that we have a formal notion of instantaneous reactor networks, we can define what it means to *run* them. For this we use a hierarchy of small algorithms, composed into a singular `run` function.

3.5.1 Outline

Two of the algorithms required for the `run` function are special, in that we do not define them explicitly. Instead, we define their behavior by constraints.

```
1 structure prec_func :=
2   (func : inst.network υ → prec.graph υ)
3   (well_formed : ∀ σ, (func σ) ⋈ σ.η)
4
5 structure topo_func :=
6   (func : prec.graph υ → list reaction.id)
7   (is_topo : ...)
```

An instance of `prec_func` is a function that generates the precedence graph for a given network (Line 2), while being subject to the constraint that it must produce only well-formed precedence graphs (Line 3). We don't define this function explicitly, as it is of no interest to our model *how* the function works. All we care about is *what* it returns. The same holds for instances of `topo_func`. All we need to know about such a function is that it can compute a topological ordering for a precedence graph.[27] It is irrelevant *how* it does it.

This aspect of the execution model is not simply a convenience, but rather a feature of the Reactor model. The execution of reactor networks behaves the same, independently of the concrete precedence and topological-ordering functions. Thus, formalizing `prec_func` and `topo_func` in this way, ensures that we retain this property of independence.[28]

The *explicitly* defined algorithms required to define instantaneous reactor network execution align quite closely with the following steps. They can be thought of as a more rigorous version of the steps presented in Section 1.2.1, while omitting time-based aspects like the global tag, event queue and actions.

1. Generate the reaction queue as a topological ordering over the network's precedence graph. This is covered by `prec_func` and `topo_func`.

2. For each reaction in the reaction queue, execute it.

   (a) Compute the reactor that we get by running the reaction, isolated to its reactor (without any value propagation).

   (b) Apply that modified reactor to the reactor network.

      i. Swap the "old" reactor for the new one, i.e. point the relevant reactor-ID to the new reactor created in step (a).

      ii. Propagate the values of all of the output-ports that were written to by the executed reaction.

Figure 12 shows that some of these steps are themselves subdivided in the implementation. This is especially true for value propagation.

[27] The `is_topo` property (Line 7) constrains the output of `func` to be a topological ordering. The exact implementation is omitted here to hide unnecessary detail.

[28] A proof of independence will be covered in Section 4.4.

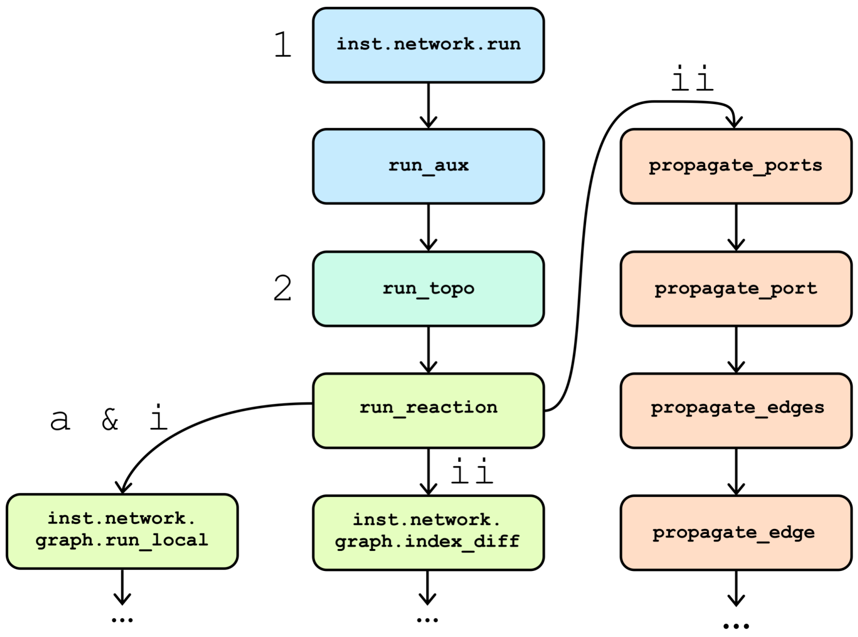

Figure 12: Explicit algorithms for instantaneous reactor network execution

3.5.2 Implementation

Algorithms in Lean are fundamentally functional (as opposed to imperative). That is, they cannot contain *sequences* of statements. Instead, an algorithm consists of one, possibly deeply nested, expression which computes the desired result. For the Instantaneous Reactor model, the `run` function is the root expression which computes the result of running a given reactor network:

```
def run
  (σ : inst.network υ) (f_p : prec_func υ) (f_t : topo_func υ) :
  inst.network υ :=
    run_aux σ (f_t (f_p σ))
```

This definition exemplifies the structure that most of the following algorithms have. They compute one small partial result and pass it on to the next algorithm (according to Figure 12). In the case of `run`, we only compute a topological ordering ($\texttt{f}_t$ $(\texttt{f}_p$ `n`$)$) for the given network, and pass it to `run_aux`:

```
def run_aux (σ : inst.network υ) (t : list reaction.id) :
  inst.network υ :=
    { inst.network . η := σ.η.run_topo t,
      unique_ins := ..., prec_acyclic := ... }
```

The syntax `{ inst.network . η := ..., ... }` is a constructor for structures. In Lean, appending `_aux` to a function's name is common practice, when a function needs to be divided into multiple parts that are tightly coupled. In the case of `run_aux`, all we do is forward the reaction queue from `run` to `run_topo`, which computes the result of executing that reaction queue on the network's graph. The main function of `run_aux` is to then reassemble that result back into an instance of `inst.network`, by providing proofs that it still fulfills the `unique_ins` and `prec_acyclic` properties.[29]

The `run_topo` function iterates over the given reaction queue, executes each reaction and combines the outputs into a single result. For this it uses a *fold*, as is typical in functional languages:

```
def run_topo (η : inst.network.graph υ) (t : list reaction.id) :
  inst.network.graph υ :=
    t.foldl run_reaction η
```

The way in which this function executes each reaction, is by calling `run_reaction`:

```
def run_reaction (η : inst.network.graph υ) (i : reaction.id) :
  inst.network.graph υ :=
    (η.run_local i).propagate_ports
    ((η.run_local i).index_diff η i.rtr role.output).val.to_list
```

This function is central to our implementation. It performs three steps:

1. It runs the given reaction in its reactor and swaps the "old" reactor for the executed one in the network graph: `(η.run_local i)`. Running a reaction isolated in its reactor has its own set of steps, which we omit here for brevity.

2. It computes which output-ports have been written to as a result of the reaction's execution (`index_diff`).

3. It propagates the values of the affected ports according to the network graph's connections (`propagate_ports`).

Output Propagation: Output propagation has an entire hierarchy of substeps of its own. For the sake of brevity, we show them all at once. The following definitions live in the `inst.network` namespace. Thus, any type whose name starts with "`graph`" actually starts with "`inst.network.graph`".

```
def propagate_ports (η : graph υ) (ps : list port.id) :
  graph υ :=
    ps.foldl propagate_port η

def propagate_port (η : graph υ) (p : port.id) :
  graph υ :=
    η.propagate_edges (η.e_o p).val.to_list

def propagate_edges (η : graph υ) (es : list graph.edge) :
  graph υ :=
    es.foldl propagate_edge η
```

[29]Cf. Section 4.3.

```
12
13 def propagate_edge (η : graph υ) (e : graph.edge) :
14   graph υ :=
15     η.update_port role.input e.dst (η.port role.output e.src)
```

Value propagation is highly repetitive. These functions specify that propagation of a list of ports requires repeated propagation of a single port (Line 3). Propagation of a single port requires propagation of all of its connected edges (Line 7), and propagation of a list of edges requires repeated propagation of a single edge (Line 11). The non-trivial expressions contained in this are:

- Line 7: The expression (η`.e`$_o$ `p`) returns the set of `graph.edge`s out of η that start in port `p`. The suffix `.val.to_list` turns that set into a list.
- Line 15: The expression (η`.update_port` ...) writes a given value to a given port, identified by its ID and its "role" (input or output). The expression (η`.port` ...) returns the value of a given port.

3.6 Constructivism

The `run` function and `prec_func` have something in common: They both define how to map one object to another. The `prec_func` does this by stating *what* the result should be, by virtue of the propositions defined by `is_well_formed_over`. The `run` function defines what an executed reactor network should look like, by providing us with a procedure for how to construct it. The difference between these approaches reflects the difference between *classical* and *constructive* mathematics.

> Constructive mathematics is distinguished from its traditional counterpart, classical mathematics, by the strict interpretation of the phrase "there exists" as "we can construct". In order to work constructively, we need to re-interpret not only the existential quantifier but all the logical connectives and quantifiers as instructions on how to construct a proof of the statement involving these logical expressions. [13]

Thus, in a constructivist world-view, our definition of `prec_func` is of little value, unless we can specify a procedure for how to construct a well-formed precedence graph. It is explicitly not enough to show that an instance of `prec_func` must exist, by indirect inference.

While the distinction between constructive and classical mathematics is not clear-cut, it often involves the decision of whether to use the *axiom of choice* and the *law of excluded middle* (LEM): $\forall p : p \vee \neg p$. Classical mathematics uses a definition of predicate logic, which implies LEM:. Notable consequences are:

- Double negation: $\forall p : \neg\neg p \equiv p$
- Proof by contradiction: $\forall p : p \equiv \neg p \to \bot$

Lean allows us to use classical *or* constructive mathematics. If we want to make our definitions explicitly classical, we can use an additional axiom called `classical.choice`. The namespace `classical` then contains useful theorems, like LEM. In order to be clear about when classical reasoning is used, Lean forces any definitions that rely on `classical.choice` to be marked as `noncomputable`. In fact, all of our explicit algorithms shown above are `noncomputable`, as they rely on non-computable equatability of certain objects — we've simply omitted the keyword here to avoid confusion. More generally, this thesis does not aim to be constructivist, and uses non-computable constructs throughout. It must be noted though, that the differences between a constructive and classical approach, exemplified by `run` and `prec_func`, come with their respective upsides and downsides.

- Defining an object constructively allows us to gloss over many of the details that are implicit in the construction. For example, it would be significantly more complicated to define the `run` function by propositions, which describe what its result should look like. This kind of definition would also be highly non-intuitive, as conceptually the result of `run` should be "the product of executing a given sequence of steps".

- The downside of a constructive approach is that `run` tells us nothing about the properties of the objects it produces. We will see below that in consequence it is much harder to prove properties about `run` than about the propositionally defined `prec_func`.

- In the context of this thesis, there is an additional drawback that comes with a constructive approach. As we are formalizing a *model*, not an *implementation* for reactors, definitions should be descriptive. Formalizing aspects of the model by concrete implementation, like the execution model above, is not descriptive. Thus, any actual implementations of the model (cf. Lingua Franca [3]) must ensure that the implemented behavior matches the implementations of the model. This makes the model less useful, in that it becomes more of a reference implementation than a description. Despite these drawbacks, we've chosen this approach for this thesis, as it allows us to narrow the scope of this initial formalization.

While the first aspect has already manifested itself in this section, the second one will become relevant in the following section.

4 Proving Determinism

In Section 1 we introduced *determinism* as the property of a system, which states that equal inputs lead to equal behavior. With respect to the `run` function, this means that we expect equal inputs to produce equal outputs. As `run` is a function in the mathematical sense, this property is trivially fulfilled. In this section we prove a stronger result. Determinism, as used here, means that for a given instantaneous reactor network, the `run` function produces the same output, *regardless of the given* `prec_func` *and* `topo_func`:

```
theorem determinism
  (σ : inst.network υ) (p p′ : prec_func υ) (t t′ : topo_func υ) :
    σ.run p t = σ.run p′ t′ := ...
```

4.1 Lean's Proof Syntax

The syntax used to state the theorem above, while new, should look familiar. In Lean, proofs of theorems and lemmas are defined using the `theorem` and `lemma` keywords. Their types are propositions, that is, types from *within* `Prop`. As logical implication corresponds to function types in the Curry-Howard isomorphism, theorems can still take parameters just like functions. This extends to *dependent* functions by the correspondence between Π-types and ∀. For example, the type of the theorem `determinism` is:

```
∀ (σ : network υ) (p p′ : prec_func υ) (t t′ : topo_func υ),
  σ.run p t = σ.run p′ t′
```

To prove `determinism`, we need to provide an instance of this type. One way of going about this is to manually construct such an instance, by composing instances of the respective logical connectives. For example: [30]

```
lemma and_add_q (h : p ∧ q) : q ∧ p ∧ q :=
  and.intro h.right (and.intro h.left h.right)
```

While this approach works great for small proofs, it can become hard to manage for larger ones, as it does align with a human way of thinking about proofs: as a sequence of steps towards a proof goal. This is why Lean provides a more human-friendly way of constructing proofs, called *tactics*.

4.1.1 Tactics

> [T]actics are commands, or instructions, that describe how to build [...] a proof. Informally, we might begin a mathematical proof by saying "to prove the forward direction, unfold the definition, apply the previous lemma, and simplify." Just as these are instructions that tell the reader how to find the relevant proof, tactics are instructions that tell Lean how to construct a proof term. [11]

[30]For the definition of `and`, cf. Section 2.2.4.

Therefore, while a proof still has to be a concrete instance of a proposition, we don't have to build this instance ourselves. In this section we briefly introduce Lean's tactic mode by proving a lemma about reactors:

```
1  lemma run_eq_input (rtr : reactor υ) (rcn : ℕ) :
2    (rtr.run rcn).input = rtr.input := ...
3
4  def reactor.run (rtr : reactor υ) (rcn : ℕ) : reactor υ :=
5    if (rtr.reactions rcn).triggers_on rtr.input
6    then rtr.merge
7      ((rtr.reactions rcn).body rtr.input rtr.state)
8    else rtr
```

The `reactor.run` function defines how to run a reaction locally within a reactor. For this, we first check whether the reactor's input ports cause the reaction to trigger (Line 5). If so, we run the reaction's body on the reactor's input and state (Line 7) and merge the result back into the reactor (Line 6). The `merge` function integrates the output of the reaction's body into the reactor by, first, overriding the reactor's state with the new state and, second, overriding exactly those output ports for which the reaction produced a non-absent value. If the reaction does not trigger, we simply return the reactor as is (Line 8).

The lemma `run_eq_input` states that running a reaction does not change the reactor's input ports. Intuitively, this lemma should hold, as running a reaction only changes output ports and state variables. We can rigorously prove this, with the following steps:

```
1  lemma run_eq_input (rtr : reactor υ) (rcn : ℕ) :
2    (rtr.run rcn).input = rtr.input :=
3    begin
4      unfold reactor.run,
5      split_ifs,
6        { unfold reactor.merge },
7        { refl }
8    end
```

We enter tactic mode with `begin` and leave it with `end`. All of the statements between these delimiters will tell Lean how to construct the proof term for this lemma. Using tactic mode allows us to reap the benefits of interactive proof assistants as described in Section 2.1. That is, we get an overview of the *tactic state*, which on Line 4 looks as follows:

```
rtr : reactor υ
rcn : ℕ
⊢ (rtr.run rcn).input = rtr.input
```

The first two lines show our current assumptions: We have instances `rtr : reactor υ` and `rcn : ℕ`. The term after the symbol ⊢ is the current *goal*, i.e. the remaining subproof that needs to be shown to complete the proof of the entire lemma. Our aim is to transform the goal in such a way that we can then prove it by using existing axioms and theorems. In the example above, we start by unfolding a definition: `unfold reactor.run`. On Line 5 our goal state therefore changes to:

```
rtr : reactor υ
rcn : ℕ
⊢ (ite ((rtr.reactions rcn).triggers_on rtr.input) (rtr.merge
    ⇑((rtr.reactions rcn) rtr.input rtr.state)) rtr).input =
    rtr.input
```

It is not uncommon to have goals that are hard to parse when unfolding functions. The problem here is that we have a large if-then-else expression (`ite`) in the goal. To simplify our goal, we can split it into two cases: one where the `ite` resolves to the positive case and one where it resolves to the negative case. We do this by using the `split_ifs` tactic. The result is that we have two distinct goals:

```
rtr : reactor υ
rcn : ℕ
h : (rtr.reactions rcn).triggers_on rtr.input
⊢ (rtr.merge ⇑((rtr.reactions rcn) rtr.input rtr.state)).input =
    rtr.input

rtr : reactor υ
rcn : ℕ
h : ¬(rtr.reactions rcn).triggers_on rtr.input
⊢ rtr.input = rtr.input
```

For the first goal, it suffices to unfold the definition of `merge` (Line 6) for Lean to be able to prove that this goal is true. The second goal can be proven by the reflexivity of equality. Hence, we use the `refl` tactic to close this goal (Line 7). Thus, all goals have been closed, and we have proven the lemma.

There are a variety of other tactics that allow us to change our goal, introduce new assumptions, split our goal into multiple subgoals, apply tactics repeatedly, etc.[31] As this thesis shows virtually no proof code, we will refrain from showing them here. We instead focus on showing the *claims* made by lemmas and theorems, as well as how they compose. The reason we can afford to ignore the proofs themselves is that their correctness is already checked by Lean.

The following proof of `determinism` consists of two parts. We first show that the choice of `prec_func` does not affect the `run` function, as there exists at most one `prec_func`. Secondly, we prove that `run` behaves the same, independently of the reaction queue generated by `topo_func`. Hence, the choice of `topo_func` does not affect the output of `run`.

[31] Cf. https://leanprover.github.io/reference/tactics.html

4.2 Equality of Precedence Graphs

The first step in proving `determinism` is to show that all precedence functions are equal, i.e. there exists at most one unique precedence function:

```
1 theorem prec_func.unique (p p′ : prec_func υ) : p = p′ :=
2   begin
3     rewrite prec_func.ext p p′,
4     funext σ,
5     exact prec.wf_prec_graphs_are_eq
6       (p.well_formed σ) (p′.well_formed σ)
7   end
```

The main step for proving this theorem is to show that all well-formed precedence graphs over a fixed network are equal (used in Line 5):

```
theorem prec.wf_prec_graphs_are_eq
  {η : inst.network.graph υ} {ρ ρ′ : prec.graph υ}
  (hw : ρ ⋈ η) (hw′ : ρ′ ⋈ η) :
  ρ = ρ′ := ...
```

This theorem can be proven relatively directly from the properties of well-formedness of precedence graphs. Using this result we can begin our proof of `determinism`:

```
theorem determinism ... : σ.run p t = σ.run p′ t′ :=
  begin
    rewrite prec_func.unique p′ p,
    ...
```

This leaves us with the goal: `⊢ σ.run p t = σ.run p t′`.

4.3 Reactor Equivalence

Despite omitting many details, we will cover one concept that has been of great utility in proving properties about reactor networks. Many of the properties of reactor networks are a result of their *structure*, rather than the specific data they hold at any given time. It is therefore useful to have a way of expressing that two networks are equal in structure, while ignoring the data they contain. We call such networks *equivalent*. The notion of equivalence starts at the level of reactors:

```
instance equiv : has_equiv (reactor υ) :=
⟨λ r r′, r.priorities = r′.priorities ∧ r.reactions = r′.reactions⟩
```

While some concepts used here[32] have not been explained, it suffices to know that this code adds a notion of equivalence to reactors, such that two reactors are equivalent if their `priorities` and `reactions` are equal. That is, we explicitly ignore their `input`, `output` and `state`. If two reactors `r` and `r′` are equivalent, we can write `r ≈ r′`. This notion of equivalence is lifted to `inst.network.graph` and `inst.network`:

[32] In particular, type class instances and anonymous constructors.

```
instance equiv : has_equiv (inst.network.graph υ) := ⟨λ η η',
  η.edges = η'.edges ∧ η.ids = η'.ids ∧
  ∀ i, (η.rtr i) ≈ (η'.rtr i)
⟩

instance equiv : has_equiv (inst.network υ) :=
  ⟨λ n n', n.η ≈ n'.η⟩
```

Networks are equivalent if they have the same edges and IDs, while only containing equivalent reactors. Hence, again, only structure matters. Using equivalence allows us to prove some very useful results:

- Equivalence of reactors preserves precedence-acyclicity:

  ```
  theorem equiv_prec_acyc_inv
    {η η' : inst.network.graph υ}
    (hq : η ≈ η') (hₚ : η.is_prec_acyclic) :
      η'.is_prec_acyclic
  ```

- Equivalence of reactors preserves uniqueness of connections to input-ports. We show a slightly stronger result here, by only requiring edges to be equal, which is a trivial consequence of equivalence:

  ```
  theorem eq_edges_unique_port_ins
    {η η' : inst.network.graph υ}
    (hₑ : η.edges = η'.edges) (hᵤ : η.has_unique_port_ins) :
      η'.has_unique_port_ins
  ```

Both of these results are integral in defining the `run` function. Since `run` returns not just an `inst.network.graph`, but an `inst.network`, we need to prove that the result of `run_topo` (which is an `inst.network.graph`) preserves precedence-acyclicity and input-uniqueness. This is done (inside of `run_aux`) by using the theorems listed above, in combination with the proof that `run_topo` produces an output that is equivalent to its input:

```
lemma run_topo_equiv
  (η : inst.network.graph υ) (t : list reaction.id) :
    run_topo η t ≈ η
```

4.4 Independence of Topological Ordering

The uniqueness of the precedence function allows us to keep the first step of our proof of determinism self-contained. We don't even consider the `run` function. The remaining proof of determinism requires us to dive into the details of `run`.

As any given precedence graph may have multiple topological orderings, it is possible that there exist multiple different `topo_func`s. Hence, we need to show that the result of `run` is independent of which `topo_func` is used. By extension, the result of `run_topo` needs to be proven to be independent of the given topological ordering. This is captured by the proof that `run_topo` is commutative:

```
1 theorem run_topo_comm
2   {η : inst.network.graph υ} (hᵤ : η.has_unique_port_ins)
3   {ρ : prec.graph υ} (hw : ρ ⋈ η) :
4     ∀ {t t' : list reaction.id} (hₚ : t ∼ t')
5       (hₜ : t.is_topo_over ρ) (hₜ' : t'.is_topo_over ρ) ,
6       run_topo η t = run_topo η t' := ...
```

The terrible legibility of this theorem is a result of there being many preconditions for the commutativity of `run_topo`. The term "commutative" as used here[33], is meant to express that we require the given topological orderings to be permutations of each other (Line 4 : `t ∼ t'`).[34] We prove the theorem by induction over one of the topological orderings, using the following intermediate results. The notation $[..., ...]$ is used for lists, $r :: l$ denotes the list where element r is prepended to list l, and $l - r$ denotes the list l with all occurrences of r removed:

1. `topo.indep_head`: If $t = [r_1, r_2, ..., r_n]$ is a topological ordering, then r_1 is not dependent on any other element in the list.

2. (1) + `topo.erase_is_topo`: If $t = [r_1, r_2, ..., r_n]$ is a topological ordering and t' is a permutation of t, then $r_1 :: (t' - r_1)$ is also a topological ordering. That is, we can pull independent elements to the front of the list.

3. If t is a topological ordering and r is an independent element in t, then `run_topo` produces the same output for t and $r :: (t - r)$.

While results 1 and 2 are general lemmas about topological orderings, result 3 is specific to `run_topo` and shown by `run_topo_swap`:

```
lemma run_topo_swap
  {η : inst.network.graph υ} (hᵤ : η.has_unique_port_ins)
  {ρ : prec.graph υ} (hw : ρ ⋈ η)
  {t : list reaction.id} (hₜ : t.is_topo_over ρ)
  {i : reaction.id} (hₘ : i ∈ t) (hᵢ : topo.indep i t ρ) :
    run_topo η t = run_topo η (i :: (t.erase i)) := ...
```

We again prove this result by induction over `t`. As a result, it suffices to show that the order in which two independent reactions are executed is irrelevant. That is, running independent reactions is commutative:

```
lemma run_reaction_comm
  {η : inst.network.graph υ} (hᵤ : η.has_unique_port_ins)
  {ρ : prec.graph υ} (hw : ρ ⋈ η)
  {i i' : reaction.id} (hᵢ : ρ.indep i i') :
    (η.run_reaction i).run_reaction i' =
    (η.run_reaction i').run_reaction i := ...
```

[33]While this usage of "commutative" is uncommon in the broader mathematical community, it is canonical within Lean's mathematical library.

[34]The lemma `topo.complete_perm` proves that this must hold for all topological orderings returned by a `topo_func`.

We can prove this lemma with the following observations. If two reactions are independent of each other, they must (1) live in different reactors *and* (2) cannot write to each other's dependencies. Hence, they cannot affect each other's input-ports. Their outputs also cannot affect each other, as we've restrained input-ports to have at most one incoming connection (Line 2: $\mathtt{h}_u$). Therefore, independent reactions neither affect each other's inputs, nor conflict in their outputs. Thus, their order of execution can be swapped. While the idea behind this proof is rather simple, the intermediate results required for this proof are far-reaching and laborious. Hence, we omit any further details here.

Using the theorem `run_topo_comm` we can complete our proof of `determinism` (Figure 13). Thus, determinism of instantaneous reactor networks is governed by two main factors:

1. Equality of all well-formed precedence graphs over a fixed network graph.
2. Independence of `run_topo` from a specific topological ordering.

It is also important to be clear that determinism, as we have shown it, depends on two assumptions: [35]

1. There exists an instance of `prec_func`. This could be proven either constructively by providing an algorithm, or non-constructively.
2. There exists an instance of `topo_func`. Proving this is of less importance, as it is a well-known result that all directed acyclic graphs have a topological ordering.

```
theorem determinism
  (σ : inst.network υ) (p p′ : prec_func υ) (t t′ : topo_func υ) :
  σ.run p t = σ.run p′ t′ :=
  begin
    rewrite prec_func.unique p′ p,
    unfold run run_aux,
    suffices h :
      σ.η.run_topo (t (p′ σ)) = σ.η.run_topo (t′ (p′ σ)),
      by simp only [h],
    have hw, from p′.well_formed σ,
    have hₜ, from t.is_topo hw,
    have hₜ′, from t′.is_topo hw,
    have hₚ, from topo.complete_perm hₜ hₜ′,
    exact graph.run_topo_comm σ.unique_ins hw hₜ.left hₜ′.left hₚ
  end
```

Figure 13: Proof of determinism of the Instantaneous Reactor model

[35] Further assumptions are implicit in the model.

5 Adding Time

One of our goals in formalizing the Simple Reactor model is to separate the instantaneous from the time-based aspects. In previous sections we've built a rigorous model of instantaneous reactors. In this section, we add the time-based aspects *on top*, i.e. without changing the instantaneous model. This allows us to model timed reactor networks as a generalization of instantaneous networks, while retaining a separation of concerns.

5.1 Primitives

Just as instantaneous reactors defined primitives (ports, state variables, etc.), timed reactor networks are built upon some additional basic notions.

Tags: In the Reactor model, we use the term *tag* to refer to a logical timestamp. Hence, if we want to add a notion of time to our model, we need to formalize tags:

```
def tag := lex ℕ ℕ
```

The type `lex` is the Cartesian product, equipped with the lexicographic order. Thus, the definition above states that `tag` is equivalent to $\mathbb{N} \times \mathbb{N}$, and the relation $\leq$ for `tag`s defines a lexicographic order. This corresponds to the superdense representation of time [5] mentioned in Section 1.2.

5.1.1 Actions as Ports

In the Instantaneous Reactor model, the entities responsible for carrying values are ports. In a time-based model, we additionally want to be able to propagate values through *actions*. Hence, we generalize the notion of ports and formalize actions as special ports, called *action ports* — a concept also employed in [4]. We will see that this also allows us to drop the formalization of *events* entirely. In this section we give a rough overview of action ports. Their details will be covered in Sections 5.2 and 5.3.

While regular ports have the ability to carry values *across reactors*, this propagation is limited to occur at a fixed logical time.[36] With action ports, we want to achieve the inverse goal. We want values to be propagated between reactions of a single reactor, but *across time*. As a result, reactions need to be able to specify *when* values should be propagated. To achieve this, we can exploit the fact that the formalization of instantaneous reactors introduces a generic *value* type v, on which virtually all components of the model are dependent. That is, instantaneous reactors work the same no matter which specific type v is. Thus, in time-based reactor networks, we give v a structure that allows reactions to add tags to their outputs. We call this structure a *timed port assignment* (TPA).

[36] All ports are cleared after each logical time step (cf. Section 1.2.1).

TPAs: A TPA consists of a finite number of tag-value pairs:

```
variables (υ : Type*) [decidable_eq υ]
def tpa := finset (tag × option υ)
```

As in Section 3, we define the underlying data *values* by means of an equatable dependent type parameter υ. The tags in a TPA specify *when* actions, which carry given values, should be scheduled. Since reactions can schedule multiple actions per execution, TPAs are not just a single tag-value pair, but a collection of them.

While the definition of TPAs is well-suited for action ports, we must consider their interaction with regular ports. Since we make no distinction in the kinds of values that are passed to different kinds of ports[37], action ports and regular ports both carry TPAs. We may therefore need to define what it means for a regular port (and by extension a reaction) to receive a TPA. Fortunately the GIGO principle can resolve this issue: We declare that all regular ports and reaction bodies must only receive TPAs that contain exactly one tag-value pair where the tag is the current logical time. Otherwise, no guarantees about a network's behavior are made. The "instantaneous value" of such a TPA is then the value contained in the TPA's only tag-value pair.

5.2 Timed Networks

To understand the behavior of TPAs more clearly, we first need to define action ports more rigorously. The only difference between regular ports and action ports is in how we treat them in timed reactor networks:

```
1 structure timed.network :=
2   (σ : inst.network (tpa υ))
3   (time : tag)
4   (event_queue : list tag)
5   (actions : finset action_edge)
6   (well_formed : actions.are_well_formed_for σ)
```

A `timed.network` is an extension on an `inst.network`. We define the underlying network to use `tpas` for its values (Line 2). That is, the data values in the instantaneous world are of type `(tpa υ)`, while the data values in the time-based world are of type υ. Lines 3 and 4 add properties that will be explained in Section 5.3. The properties concerning action ports are on Lines 5 and 6.

5.2.1 Action Edges

Reactors come with *output* and *input* ports, which receive values from reactions and propagate values into reactions respectively. We generalize this notion to action ports and talk about *output action ports* (OAP) and *input action ports* (IAP). OAPs receive the TPAs produced by reactions and IAPs propagate values back into reactions. We can connect OAPs and IAPs using special edges:

[37] We *can't*, without changing the formalization of instantaneous reactors.

```
structure timed.network.action_edge :=
  (oap : port.id)
  (iap : port.id)
```

Thus, everything that is an *action* in the Simple Reactor model is now represented by an OAP, an IAP, and an edge between them.[38] If a reaction wants to be able to schedule a specific action, it needs to declare a corresponding OAP as its antidependency. If a reaction wants to be able to receive values from an action, it needs to declare that action's IAP as a dependency.

While this system works well at first glance, we need to consider the following scenario: What if multiple reactions want to schedule the same action? Since output ports may have multiple inputs, i.e. be an antidependency for multiple reactions, we can connect multiple reactions to the same OAP. This leads to the problem that, if two reactions schedule the same action during the same round of execution, they both write to the same OAP. Thus, one of their TPAs will override the other. What we would want is for their TPAs to be merged into a single TPA. But to achieve this we would have to adjust the implementation of instantaneous reactors. Hence, we take a different approach — we allow each action to have multiple OAPs. Each reaction that wants to schedule an action can then connect to its own unique OAP, so no overriding occurs. Figure 14 shows an example of a reactor that contains an action with multiple connected reactions (and therefore multiple OAPs), as well as a reaction that schedules multiple different actions.

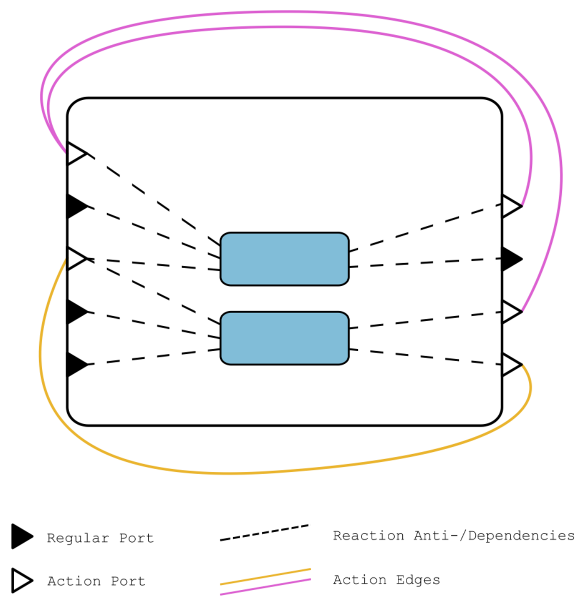

Figure 14: Reactor with two actions

[38]We will see below that we sometimes need multiple OAPs per action.

5.2.2 Well-Formedness

Modeling actions as ports and edges only works subject to some constraints. These constraints are enforced by `timed.network.well_formed` (Line 6 above):

```
def finset.are_well_formed_for
  (es : finset action_edge) (σ : inst.network (tpa υ)) :=
    es.are_many_to_one ∧
    es.are_local ∧
    es.have_unique_source_in σ ∧
    es.are_functionally_unique_in σ ∧
    es.are_separate_from σ
```

We go through these constraints in order:

1. Each action has exactly one IAP, and OAPs can belong to only one action. Thus, OAPs cannot be connected to multiple IAPs:

   ```
   def finset.are_many_to_one (es : finset action_edge) :=
     ∀ e e′ : action_edge, e ∈ es → e′ ∈ es →
       e.oap = e′.oap → e = e′
   ```

2. Actions are local to a single reactor. Hence, action edges must begin and end in the same reactor:

   ```
   def finset.are_local (es : finset action_edge) :=
     ∀ e : action_edge, e ∈ es →
       e.oap.rtr = e.iap.rtr
   ```

3. To avoid overriding of TPAs (as previously explained), OAPs may be connected to at most one reaction. The expression (`σ.η.deps r role.output`) returns the output-dependencies of a reaction `r`:

   ```
   def finset.have_unique_source_in
     (es : finset action_edge) (σ : inst.network (tpa υ)) :=
       ∀ (e : action_edge) (r r′ : reaction.id), e ∈ es →
         (e.oap ∈ σ.η.deps r role.output) →
         (e.oap ∈ σ.η.deps r′ role.output) →
        r = r′
   ```

4. A single reaction cannot use multiple OAPs for scheduling the same action. That is, if a reaction connects to two distinct OAPs, they cannot connect to the same IAP. We make this restriction, because the execution of a timed reactor network will require us to merge TPAs. The order in which we perform merges is based on the priorities of OAPs. An OAP's priority is inherited from the reaction it is connected to. Hence, this restriction is necessary to keep OAPs' priorities unique, thereby allowing us to impose a total order on them, which allows us to retain determinism when merging TPAs:

```
def finset.are_functionally_unique_in
  (es : finset action_edge) (σ : inst.network (tpa υ)) :=
    ∀ (e e′ : action_edge) (r : reaction.id),
      e ∈ es → e′ ∈ es →
      (e.oap ∈ σ.η.deps r role.output) →
      (e′.oap ∈ σ.η.deps r role.output) →
      e.iap = e′.iap →
      e.oap = e′.oap
```

5. While action ports are *just ports*, they should not also take on the role of regular untimed ports. Hence, it must be ensured that no "regular network edges" are attached to them:

```
def finset.are_separate_from
  (es : finset action_edge) (σ : inst.network (tpa υ)) :=
    ∀ (ae : action_edge) (ne : inst.network.graph.edge),
      ae ∈ es → ne ∈ σ.η.edges →
      ae.iap ≠ ne.dst ∧ ae.oap ≠ ne.src
```

Giving action ports their own separate edges and imposing these five restrictions, allows us to model actions as a generalization ports.

5.3 Execution Model

Just as the definition of timed reactor networks builds heavily on that of instantaneous reactor networks, so does the execution model. Time-based execution can be modeled as a sequence of instantaneous executions. To achieve this, we require two things: a global logical time and an event queue. We've already added these components in our definition of `timed.network` above:

- `time` is the tag for the current logical time.
- `event_queue` is an ordered list of tags, that indicates at which logical times the next instantaneous executions need to take place. In our Simple Reactor model, a tag can only be added to this queue as the result of scheduling an action for that tag. Note that, while the event queue in the Simple Reactor model has explicit *event* objects in its event queue, this formalization only queues *tags*. Explicit event objects are not used at all in this formalization.

Execution of a timed reactor network progresses as follows. As long as there are events in the event queue, we execute an instantaneous version of the network at the tag of the next event. The IAPs in this network must contain the values specified by the actions scheduled for this tag. Upon completion, integrate the actions scheduled during the instantaneous execution into the timed network and advance the current logical time.

We formalize this with the following implementation.[39] For every tag T in a timed reactor network's event queue:

1. If the event queue is empty, complete execution. Otherwise:
2. Merge all TPAs from all OAPs into their respective IAPs.
3. Construct an instantaneous reactor network for the current tag T, by removing all tag-value pairs with a tag $\neq T$ from all IAPs' TPAs.
4. Run the instantaneous network as formalized in Section 3.5, but clear all regular ports upon completion.
5. Build a timed network from the result of Step 4:
    (a) Restore the TPAs in the IAPs to the state from before Step 3.
    (b) Set the network's `time` to T.
    (c) For every tag that appears in the executed network's OAPs' TPAs, add it to the existing event queue. Then sort the event queue in increasing order (earlier tags preceding later tags).
6. Repeat this procedure from Step 1, using the timed network from Step 5 for further execution.

There are again a variety of small steps required to realize such an algorithm:

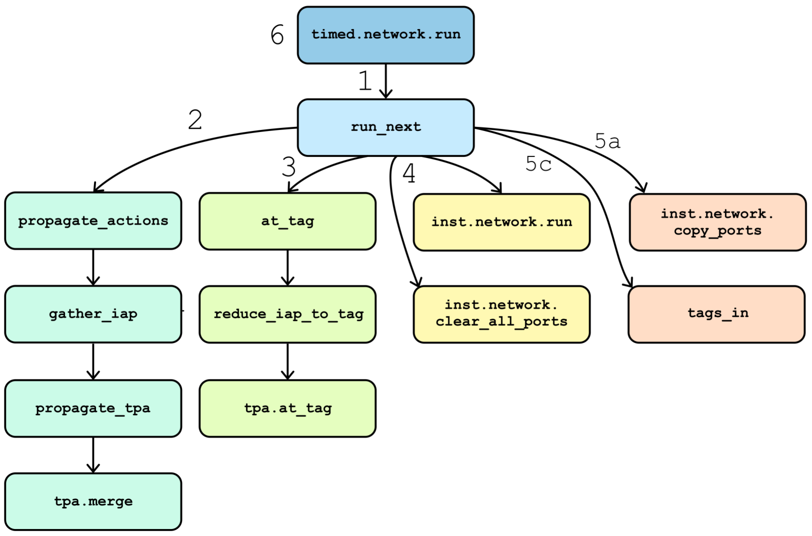

Figure 15: Functions in the execution of timed reactor networks

[39]For a discussion of why we formalize by implementation, cf. Section 3.6.

As we show no proofs about timed networks in this thesis, we omit any details about these functions. Instead, we only consider one important detail about `timed.network.run`, namely that it doesn't exist. We can see why, by examining the core function in the implementation of timed network execution: `run_next`. This function gets the next tag in the event queue and runs the network *only for that tag*:

```
def run_next (τ : timed.network υ)
  (f_p : prec_func (tpa υ)) (t_p : topo_func (tpa υ)) :
  timed.network :=
    match τ.event_queue with
      | [] := τ
      | hd :: tl := ... -- run `hd`
    end
```

The responsibility of `timed.network.run` would then be to implement Step 6 of the execution model. That is, take whatever `run_next` returned and feed it back into `run_next`. As it turns out, directly implementing this step in Lean is impossible. We cover the underlying issue in the following section.

5.3.1 Infinite Recursion

The reason why we cannot implement Step 6 is that it causes potentially infinite recursion. By "repeating the procedure from Step 1" and beginning each iteration *with a new timed network*, we can mutate the event queue on every iteration. In consequence, we can repopulate it each time, thus causing an infinite number of iterations. As a minimal example, consider the following timed reactor network:

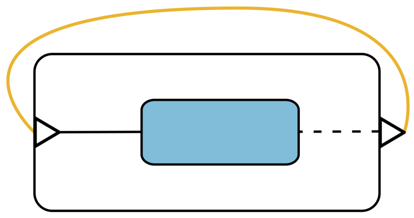

Figure 16: Indefinitely executing timed reactor network

If we assume that the reaction schedules an action every time it executes, it will keep triggering itself, and hence sustain execution indefinitely.

Recursion in Lean: While infinite recursion can be trivially implemented in most programming languages, Lean explicitly forbids it. In fact, recursion is already ruled out by the typed lambda calculus, where the type restrictions for valid terms make it impossible to define recursive functions. Thus, in Lean, any recursive function definition is rewritten into a valid term by the compiler. This is only possible, when the recursively defined function provably terminates.[40] In certain cases, Lean can prove this fact for us, so we don't need to make it explicit. Take as an example this inductively defined function over lists:

```
def list_sum : list ℕ → ℕ
  | [] := 0
  | (hd :: tl) := hd + list_sum tl
```

For this function, Lean can prove to itself that:

1. Every recursive call takes as argument a list that is smaller than the one received by the enclosing function context.
2. There's a smallest list (the empty list) whose definition for this function is non-recursive.

Hence, any call to this function can only recurse finitely many times. When Lean cannot infer that a recursive call is "decreasing", we have to implement the recursion manually by means of the well-founded recursion theorem.[41] This mechanism allows us to explicitly prove the properties that Lean could infer for the function above. Since the execution of timed reactor networks can be inherently non-terminating, we cannot prove such properties and are forced to find a different approach for modeling indefinite sequential computation. We mention a possible solution to this problem in Section 6.2.

[40] There technically is a way to define infinite recursion in Lean — via `meta` functions. These aren't properly checked by Lean and hence allow us to prove `false`.

[41] For this, Lean provides definitions like `well_founded.recursion` and `well_founded.fix`.

6 Conclusion

The formalization of the Simple Reactor model as shown in this thesis represents a first step towards a rigorous formalization of the Reactor model. It allowed us to explore the notions required to express the most primitive aspects of the Reactor model. Sticking to a small model enabled us to incrementally build up a repertoire of lemmas and theorems, which finally lead to a rigorous proof of the model's determinism. Additionally, the separation of the instantaneous and time-based aspects, allowed us to formalize the latter as a generalization of the former. In all of this, the correctness enforced by Lean played an integral role in avoiding mathematical inaccuracies contained in the full Reactor model. It also helped us surface some of the assumptions required for determinism — namely the existence of a precedence and topological-ordering function.

6.1 Working with Lean

While Lean has been of great utility for this project, it is a double-edged sword.

Upsides: The upsides of using Lean as a proof assistant are very compelling. It provides (almost) absolute certainty about the correctness of mathematical work, which greatly reduces the effort required to review theorems and their proofs. For this thesis, we wrote over 2000 lines of Lean code, including over 130 theorems and lemmas, which for the most part can be ignored without further consideration, thanks to Lean.[42]

Further, the certainty provided by Lean can benefit the *user* of the system. It forces amateur or non-mathematicians into being mathematically correct, which is particularly useful for learning how to create correct proofs.

> I don't think I *really* got [proofs] until encountering the Lean theorem prover [...]. With Lean you have your hypotheses, you have your proof goal (the thing you're trying to prove), and you have a set of moves you can make. It's very much like a game [...]. Before I learned Lean, writing proofs was like playing chess without knowing that bishops existed or how knights moved and thinking pawns could teleport around the board. Just knowing the rules, knowing what constitutes a valid move and knowing the space (or at least more of the space) of valid moves, was so powerful for my understanding of how to write a proof. Writing proofs became like navigating an endlessly fascinating maze, using theorems to hop from place to place until finding the conclusion! [14]

Hence, as a non-mathematician it is possible to start proving a theorem without a solid grasp on the steps involved, and stumble upon them along the way.

Lastly, formalization in Lean may have even bigger benefits with the advancement of AI. As formalizations in Lean are growing into an ever larger

[42] It is of course still necessary to check that the theorems state the desired propositions.

corpus [15], it is becoming more plausible to train software to generate proofs in Lean. At *Lean Together 2021*, Jason Rute and Jesse Han presented an implementation of a proof tactic based on machine learning[43], with remarkable results. Efforts like this advance Lean into becoming more of an automated theorem prover than a mere proof assistant.[44]

Downsides: The mathematical precision required by Lean can also be one of its worst aspects. For example, we had to define (labeled multi-)digraphs and some of their properties ourselves, even though they are of virtually no interest to our model. In traditional mathematics we would generally gloss over such an object, and take its lemmas for granted. Additionally, for Lean ...

> [t]he learning curve is steep. It is *very hard* to learn to use Lean proficiently. [...] Are you a student at Imperial being guided by Kevin Buzzard? If not, Lean might not be for you. [16]

While the gamified proof process is certainly useful for proving simple theorems, it can get in the way when trying to prove larger ones. As every detail needs to be correct, it can be easy to lose the overarching argument of a proof to technicalities. Hence, even if you have a proof idea, you may not be able to easily state it in Lean. And once you do manage to start proving a theorem, there are many roadblocks that can stop you dead in your tracks.

Community: When getting stuck, the only remedy is usually to turn to the Lean community. Fortunately, Lean has a very active community with constant discourse on a public forum.[45] It is hard to express the value it provides. Aside from constant helpful answers to even the most niche problems, the community actively develops and maintains the *Mathlib* library. Mathlib is an ever-growing, coordinated effort to collect formalizations of "mainstream" mathematics in a central library. In fact, many of the lemmas and structures used in this thesis are actually part of Mathlib, rather than Lean itself.

6.2 Future Directions

As this thesis explicitly models a subset of the full Reactor model, there are a variety of next steps that can be taken. Most pressing is probably the formalization of non-terminating execution as described in Section 5.3.1. A possible approach for this is to formalize the execution model of timed reactors non-constructively, i.e. by propositions, like `prec_func` and `topo_func`. This approach would also make the formalization of the execution model more descriptive. [46] Formalizing a function in this manner could be done by returning a

[43]*LeanStep* at `https://leanprover-community.github.io/lt2021/schedule.html`
[44]Cf. Section 2.1.
[45]`https://leanprover.zulipchat.com`
[46]Cf. Section 3.6.

(potentially infinite) sequence[47] of instantaneous networks [17], such that adjacent elements in the sequence must be valid predecessors/successors according to the execution model. Determinism could be proven by building upon the determinism of instantaneous networks.

Another desirable addition would be the introduction of physical time. As one of the key aspects of the Reactor model are distinct notions of time, key insights into the model are only possible once time is fully formalized.

To flesh out the model, mutations and nested reactors could be added. Since we've already defined a notion of reactor networks, the addition of nested reactors should be unintrusive. Mutations might require some larger adjustments, as they constitute an additional step in the execution model and therefore have a larger impact on the proof of determinism.

Not least, it might be valuable to prove the existence of a `prec_func` and `topo_func` by providing explicit algorithms for them in Lean. Thus, both of the assumptions upon with determinism of instantaneous reactor networks currently rests, would be proven.

[47] `https://leanprover-community.github.io/mathlib_docs/data/seq/seq.html#seq`

A Project Overview

The code shown in this thesis is only a tiny excerpt of the current formalization of the Simple Reactor model. This section aims to provide a high-level overview of the project structure, so that it may be more easily navigated. Generally, definitions and corresponding lemmas/theorems are placed in the same file, with the theorems appearing immediately after the definition.

The root folder contains formalizations, which are not specific to reactors:

- `lgraph.lean` defines L-graphs, including the definitions of paths and acyclicity.
- `topo.lean` defines (complete) topological orderings, and proves important lemmas about them.
- `mathlib.lean` contains lemmas about structures from Mathlib, which are not (yet) part of Mathlib. These lemmas were all proven by Yakov Pechersky.

The `timed` folder contains definitions about timed reactor networks.

- `basic.lean` defines tags, TPAs, and timed networks.
- `exec.lean` defines the timed execution model, i.e. `run_next` and all of its steps.

The `inst` folder contains definitions about instantaneous reactors.

- `primitives.lean` defines state variables, ports, and many other definitions/lemmas about ports which were not discussed in this thesis, such as *port-roles* and *inhabited indices*.
- `reaction.lean` defines reactions and their triggering condition.
- `reactor.lean` defines reactors, operations for mutating them, a procedure for executing a reaction in them, reactor equivalence, and *relative equality* (another concept omitted in this thesis).

The `inst/network` folder defines notions about instantaneous reactor *networks*.

- `ids.lean` defines reactor-, reaction- and port-IDs.
- `graph.lean` defines instantaneous reactor network graphs, operations for mutating them, a procedure for executing a reaction locally (without output propagation), and network graph equivalence.
- `basic.lean` expands on `graph.lean` by defining full instantaneous networks, as well as lifting some notions from network graphs to networks.
- `prec.lean` defines precedence graphs, their property of well-formedness, the network property `is_prec_acyclic`, and proves the equality of well-formed precedence graphs.

The `inst/exec` folder defines the execution model for instantaneous networks.

- `run.lean` defines the `run` function, as well as the proof of determinism.
- `topo.lean` defines `run_topo` and `run_reaction`, as well as the corresponding proofs `run_topo_comm`, `run_topo_swap` and `run_reaction_comm`.
- `propagate.lean` defines all of the propagation functions.
- `algorithms.lean` defines the "implicit" algorithms, i.e. `prec_func` and `topo_func`, as well as the proof that all `prec_func`s are equal.

References

[1] Marten Lohstroh, Íñigo Íncer Romeo, Andrés Goens, Patricia Derler, Jeronimo Castrillon, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. Reactors: A Deterministic Model for Composable Reactive Systems. In *Cyber Physical Systems. Model-Based Design*, pages 59–85. Springer, 2019.

[2] Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson. The Reactive Manifesto. *Online:* *https://www.reactivemanifesto.org*, 2014.

[3] Marten Lohstroh, Christian Menard, Alexander Schulz-Rosengarten, Matthew Weber, Jeronimo Castrillon, and Edward A Lee. A Language for Deterministic Coordination Across Multiple Timelines. In *2020 Forum for Specification and Design Languages (FDL)*, pages 1–8. IEEE, 2020.

[4] Marten Lohstroh. *Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems*. PhD thesis, 12 2020.

[5] Claudius Ptolemaeus. System Design, Modeling, and Simulation using Ptolemy II, Ptolemy.org, 2014. *Online:* *http://ptolemy.org/books/Systems*, pages 20–23, 2013.

[6] Bernard Linsky and Andrew David Irvine. Principia Mathematica. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2020 edition, 2020.

[7] Peter Dybjer and Erik Palmgren. Intuitionistic Type Theory. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, summer 2020 edition, 2020.

[8] Thierry Coquand and Gérard Huet. The Calculus of Constructions. 1986.

[9] Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. 2015.

[10] Mario Carneiro. The Type Theory of Lean. Master's thesis, 2019. *Online:* *https://github.com/digama0/lean-type-theory/releases/tag/v1.0* *and* *https://youtu.be/3sKrSNhSxik*.

[11] Jeremy Avigad, Leonardo de Moura, and Soonho Kong. Theorem Proving in Lean. *Online:* *https://leanprover.github.io/theorem_proving_in_lean/theorem_proving_in_lean.pdf*, 2021.

[12] Kevin Buzzard. *Online:* *https://leanprover.zulipchat.com/#narrow/stream/113489-new-members/topic/Casting.20Functions/near/221805516*, 2021.

[13] Douglas Bridges and Erik Palmgren. Constructive Mathematics. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, summer 2018 edition, 2018.

[14] Andrew Helwer. Doing a math assignment with the Lean theorem prover. *Online:* https://ahelwer.ca/post/2020-04-05-lean-assignment/, 2020.

[15] Mathlib Statistics. *Online:* https://leanprover-community.github.io/mathlib_stats.html.

[16] Thomas Hales. A Review of the Lean Theorem Prover. *Online:* https://jiggerwit.wordpress.com/2018/09/18/a-review-of-the-lean-theorem-prover/, 2018.

[17] Edward A. Lee and Eleftherios Matsikoudis. The Semantics of Dataflow with Firing. *G. Huet, G. Plotkin, J.-J. Lévy, and Y. Bertot, editors, From Semantics to Computer Science: Essays in Honour of Gilles Kahn*, pages 71–94, 2009.

Acknowledgements

This thesis has grown to become a bigger beast than I would have expected. I would like to thank my advisor Andrés for suggesting such a great topic, providing me with challenging yet helpful material when necessary, and especially for helping me avoid (or sometimes climb out of) the pits of misdirected effort. Though not officially, there has definitely been a second advisor to this thesis: the Lean community. For helping me work through countless roadblocks I would especially like to thank Adam Topaz, Alex J. Best, Bryan Gin-ge Chen, Eric Wieser, Floris van Doorn, Johan Commelin, Kevin Buzzard, Mario Carneiro, Patrick Massot, Reid Barton, Rob Lewis, and Yakov Perchersky.