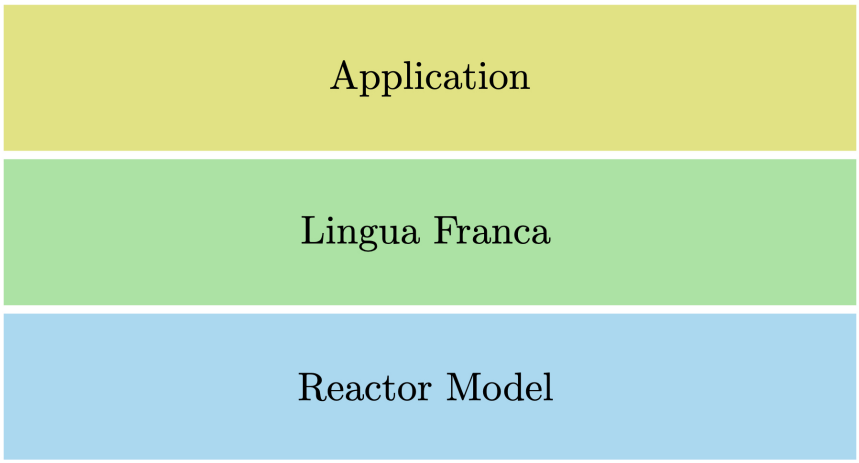
Talk: Provable Determinism in Reactors

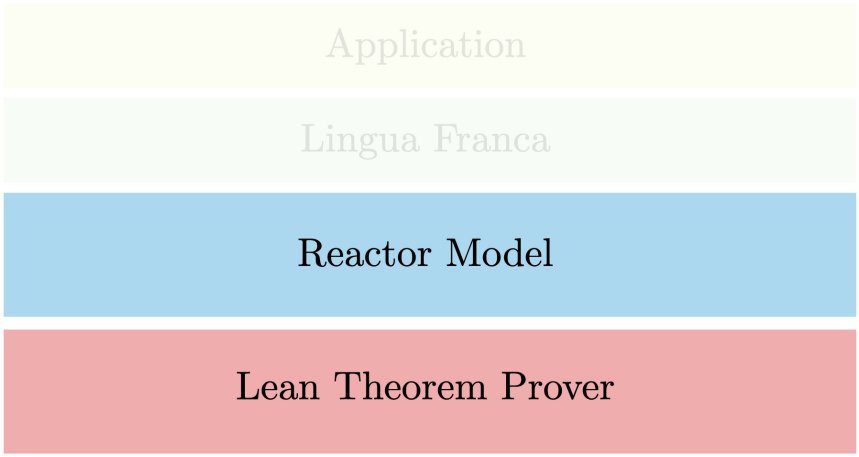
This talk is going to be about my bachelor thesis *Provable Determinism in Reactors*, in which I show how to formalize a small subset of the *Reactor model* using a tool called *Lean*, and prove that execution of such simplified Reactors is deterministic. Just for some context: This presentation is actually part of my defense for this thesis. Since part of it is about motivating Reactors, which shouldn't be necessary for this audience, I'll be skipping some slides. So what I want to talk about *today* is mainly Lean, i.e. the tool for formalizing Reactors and proving properties about them.

To get started, let's briefly consider what this talk is *not* going to be about.



I'm guessing you're all familiar with *Lingua Franca*, which is a software framework upon which we can build (concurrent) applications. What we hope to attain by building applications upon this framework, are certain beneficial properties like loose coupling, testability and determinism. Lingua Franca adheres to the structure defined in the Reactor model, which is a purely mathematical description of reactors, their components and behaviors. One of the reasons we build the software framework on this mathematical foundation is so that we can actually *prove* those beneficial properties that we're trying to attain. So while we're *hoping* that Lingua Franca gives us, for example, determinism, we can actually *prove* this mathematically for the Reactor model.

This endeavor raises some meta-problems. First, how can show that Lingua Franca actually adheres to the Reactor model? Second, how can we be sure that our mathematical formalization is actually well-defined and our proofs are valid? We won't tackle the first problem here, but instead focus on the second one. And the second problem really isn't unique to the Reactor model, but applies to mathematics as a whole: How can we be sure that mathematical work is correct?



This is where Lean Theorem Prover plays its part. Lean is a tool for formalizing mathematics, such that we can be (almost) certain that it's error-free. So what I'd like to show you in the next couple of minutes are, firstly the foundations of Lean that are necessary for working with this tool and secondly a hands-on example of formalizing a (very small) piece of the Reactor model.

Pen-And-Paper

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

- $I \subseteq \Sigma$ are inputs
- $O \subseteq \Sigma$ are outputs
- $S \subseteq \Sigma$ are state variables
- $A \subseteq \Sigma \times \mathbb{N} \times \mathfrak{D}$ are actions
- $\mathcal{N} \subseteq \mathfrak{R}$ are reactions
- $\mathcal{P} : \mathcal{N} \to \mathbb{N}$ are priorities

Lean

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

I don't know your backgrounds, but I'm guessing that most of you have worked with traditional, pen-and-paper mathematical definitions like the ones shown on the left. For example, here we define what a reactor is, and notably, we use the language of set theory and predicate logic. That's the norm in traditional mathematics: we use ZFC set theory (which is based on predicate logic). On the right side, you can see what definitions look like in Lean. While

this looks a lot like regular programming language code, in part because of the font and colors, this is actually a strictly mathematical definition. So in the following slides, when you see code like this, try to read it as *mathematical notation* not *programming language code*.

Even though this code snippet is a mathematical definition, there's actually a fundamental difference relative to the definition on the left: The mathematical foundations for Lean are not ZFC, but something called the *Calculus of Inductive Constructions* (CiC). CiC is not a theory of *sets*, but a theory of *types*. Hence, creating formalizations in Lean requires us to first (briefly) review these different mathematical foundations. Luckily, this will also provide us with an opportunity to introduce some of Lean's syntax.

Lean's Foundations

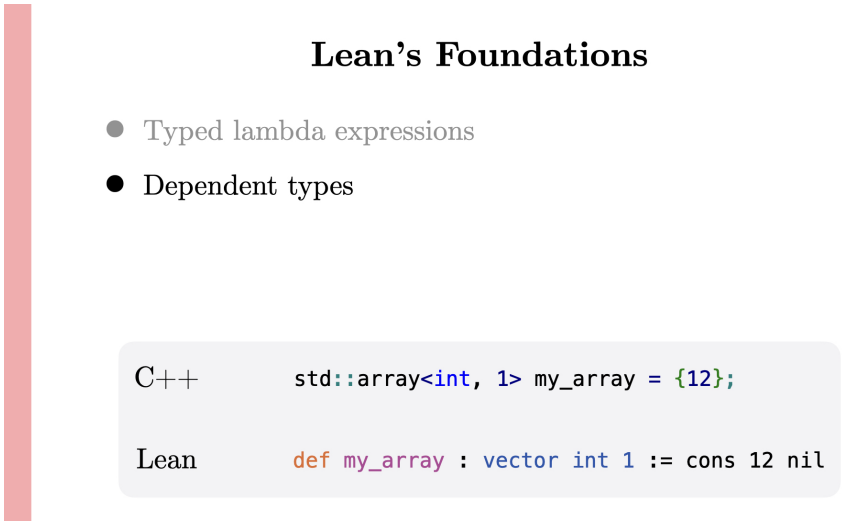
- Typed lambda expressions

```
def double : ℕ → ℕ := λ n, 2 * n

def ten : ℕ := double 5
```

CiC based on *Lambda Calculus*, which most of you probably had to learn during your studies at some point. If not, it suffices to know that this is a model of computation (like Turing machines), which provides us with a concept of *functions* and hence *computation*. In CiC we use a flavor of lambda calculus, which is *typed*. That is, every term has an associated type — just like in typed programming languages. At bottom of this slide you can see examples of this (this is valid Lean syntax). First of all, we define a function called `double`, which has the type $\mathbb{N} \to \mathbb{N}$ and is defined by the term `λ n, 2 * n`, which maps a natural number `n` to the natural number `2 * n`. This is an example of how we can define a function. Secondly, we have an example of applying a function: The symbol `ten` is a natural number, defined by the result of applying the function `double` to the value `5`. These are the basic notions of computation in typed lambda calculus. And if you've ever programmed in a typed programming language, these basic type-related notions should come quite natural to you. But if you tend to make heavy use of types your style of programming, Lean should be all the more exciting for you, because in Lean types and values are

strongly intermingled. This has to do with the second fundamental aspect of Lean, called *dependent types*.



If you've ever written generic code, you might have wished you could add a value as a generic parameter. E.g. you might want to program an array type that has a fixed length, without requiring special compiler support. C++ has *some* support for this, so you can write the code shown at the bottom. But this only works for statically determined values. In Lean values can naturally be a part of types, without any constraints. So we can express precisely the same definition in Lean using the `vector` type. Thus, the type of `my_array` is precisely `vector int 1`, making the value `1` part of the type. And in fact, the value doesn't have to be a static `1`, but could be the result of some function call. The origin of dependent types are *dependent functions*, which we can talk about later if you're interested. So dependent parameters are a neat aspect of types in Lean, but how do we even define a type - e.g. how is `vector` defined? To uncover this, we'll have to consider another key aspect of Lean: *inductive types*. They are the main mechanism by which we define new types, and by extension values, in Lean. As an example, let's consider the definition of the `list` type (from which the `vector` type is derived).

Lean's Foundations

- Typed lambda expressions
- Dependent types
- Inductive types

```
inductive list (α : Type*)
  | nil : list
  | cons (hd : α) (tl : list) : list
```

What you see here is an inductive type definition. This is like a scheme for how to make instances of the `list` type. In this example, it tells us that there are two possible ways to make a list: Firstly, the symbol `nil` is a list, which represents the empty list. That is, `nil` a new kind of value of type `list`, which we can create out of thin air - kind of like when you declare a case in an enum in a C-style programming language. Secondly, if `hd` is an instance of type α and `tl` is a list (over elements of type α), then those two things taken together are an instance of type `list` α. So by calling the `cons` constructor, we basically reinterpret these two items as an instance of `list` α. Now there's one additional part here, which is the declaration of α at the top. The α is a dependent parameter, just like the value `1` was in our vector example. That is, any concrete `list` type depends on a parameter α of type `Type*`. The interesting part here is this `Type*`. `Type*` is the *type of types*. That is, in Lean we can use types as values and their type is `Type*`. So this α behaves exactly like a generic type parameter in regular programming languages, making it possible to define lists over all kinds of elements. And since Lean supports dependent types, this α doesn't *have to* be a type (`: Type*`), but could also be some value like a natural number (`: ℕ`). Technically there's a bit more going on with this `Type*` thing, which we can again talk about at the end if you're interested.

Now we have our notion of functions from the lambda calculus, we can define types using inductive schemas, and we can make those types dependent on other types and values. This is almost all we need for formalization most objects in the Reactor model. We can define lists this way, or natural numbers, reactor ports, state variables, reactor networks, etc. Of course, these objects aren't all that interesting without theorems about them. And if we want to show that the Reactor model behaves deterministically, we need to be able to prove theorems about it. For this, we'll have to look at one important type of object that we haven't considered yet: mathematical propositions. In Lean, propositions live at an interesting intersection between types and values. The reason for this is

the *Curry-Howard isomorphism*.

Lean's Foundations

- Typed lambda expressions
- Dependent types
- Inductive types
- Curry-Howard isomorphism

```
inductive or (p q : Prop) : Prop
  | inl (l : p) : or
  | inr (r : q) : or
```

```
inductive true : Prop
  | intro : true
```

In the context of Lean, the Curry-Howard isomorphism is the answer the question: How do we express proofs in type theory? There's really not a *correct* answer to this question, but one neat approach was discovered by Haskell Curry: We model propositions as types and proofs as instances of those types. There might be some questions that might come with this, which are hopefully clarified by the examples at the bottom. Firstly, how do we express proposition as types - aren't they terms over values? In short, with types that are dependent over values. For example, consider how we formalize a disjunction with the `or` type: If `p` and `q` are already propositions, then the proposition $p \vee q$ is the type `or p q`. The way we declare `p` and `q` to be propositions, is by declaring their type to be `Prop`. Recall that we model propositions as types. So `p` and `q` are types, and hence, `Prop` is again a type of types, similar to `Type*`. By the Curry-Howard isomorphism we also defined proofs to be instances of the propositions we want to prove. So the definition of `or` also shows us how we can create proofs of a disjunction: Either by providing a proof of `p` using the `inl` constructor, or by providing a proof of `q` using the `inr` constructor. That is, `l` and `r` are proofs of `p` and `q` respectively. As another example of a proposition, consider `true`? What this definition tells us, is that we can simply construct an instance of `true` using the `intro` constructor. We don't require proofs of any other propositions for this. Hence, we can always obtain a proof of the proposition `true` out of thin air. Of course, that's exactly how `true` should behave. We can define all fundamental propositional connectives in Lean using this approach. And in fact, one of the observations of the Curry-Howard isomorphism was the correspondence of these logical connectives to "normal, data-related types".

Lean's Foundations

- Typed lambda expressions
- Dependent types
- Inductive types
- Curry-Howard isomorphism

```
inductive or (p q : Prop) : Prop
  | inl (l : p) : or
  | inr (r : q) : or
```

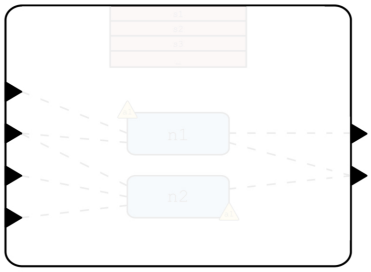
Logic	Types
$\wedge$	product type
$\vee$	sum type
$\Longrightarrow$	function type
$\forall$	Π-type
$\exists$	Σ-type
$\top$	unit type
$\bot$	empty type

For example, the `or` type corresponds to what is known as a *sum type*, i.e. a type that can hold a value of one type or another (this is also called a *discriminated union* in traditional programming languages). The `true` type corresponds to a unit type, that is a type that has only one possible value (this corresponds to the `void` type in C). One of my favorite correspondences is between implication and function types. That is, a proof of an implication $A \Rightarrow B$ corresponds to a function that can map a proof of A to a proof of B. Hence, in Lean the implication arrow is in fact exactly the same construct as the function arrow. While the *propositions as types* paradigm *technically* provides us with a way to construct proofs, it's really not a fun way of proving things. As humans, we tend to write proofs as sequences of expressions that lead us toward a proof goal bit by bit. This approach isn't really possible when directly constructing proofs as instances of types.

To solve this problem, Lean provides us with a way of proving propositions using a sequential approach, called *tactic mode*. To demonstrate tactics, I would like to prove something about reactors with you, which is actually part of the formalization of the Reactor model in my thesis. We're going to formalize and prove an (admittedly trivial) theorem about ports.

A Theorem about Reactors

```
def ports (υ : Type*) : Type* := list (option υ)
```



We define ports as lists over optional values. That is, first of all `ports` is a *type*, indicated by the type annotation `: Type*`. On the right-hand side, we declare what exact type `ports` is: a list over instances of `option` v. The v is the type of *values*. Recall that the Reactor model defines values as *opaque* objects, which are passed around between reactors and reactions. Hence, we model them like a generic type by defining them using a dependent parameter. Lastly, the `option` is just responsible for equipping the values with an *absent value* (as defined in the Reactor model). This isn't really important to us now. Thus, `ports` is a type that models lists of values. We can refer to each individual port by indexing into these lists. In the process of formalizing the Reactor model, at some point I needed to be able to check whether two instances of `ports` have the same values for given indices — for example, whether the second and the fifth port of two instances hold the same value. To express what it would mean for two instances of `ports` to have this correspondence, we can define a proposition: `eq_at`.

A Theorem about Reactors

```
def ports (υ : Type*) : Type* := list (option υ)

def eq_at {υ : Type*} (i : finset ℕ) (p p' : ports υ) : Prop :=
  ∀ x ∈ i, p.nth x = p'.nth x
```

That is, `eq_at` *is* a proposition (indicated by the `: Prop`). Namely, the proposition that for each index `x` from the set of indices `i`, the instances `p` and `p'` have corresponding values at `x` (the function `nth` gets the value at the given index).

What I would like to prove now (with your help) is that `eq_at` is reflexive.

A Theorem about Reactors

```
def ports (υ : Type*) : Type* := list (option υ)

def eq_at {υ : Type*} (i : finset ℕ) (p p' : ports υ) : Prop :=
  ∀ x ∈ i, p.nth x = p'.nth x

theorem eq_at_refl (υ : Type*) (i : finset ℕ) (p : ports υ) :
 eq_at i p p :=
 …
```

That is, given any set of indices `i` and any instance of `ports p`, it will always hold that `eq_at i p p`. I hope you see why this is trivially true. What isn't trivial, is how to prove this theorem. First of all, let's consider its definition. We declare theorems in Lean using the `theorem` keyword. The type of a theorem is a proposition: the one we intend to prove. This works, because (if you recall) propositions are types and hence `eq_at i p p` (since it is a proposition) is a

type. Since a proof of a proposition is just an instance of it, proving a theorem means construction an instance of its type. In this case, an instance of type `eq_at i p p` looks like this.

A Theorem about Reactors

```
def ports (υ : Type*) : Type* := list (option υ)

def eq_at {υ : Type*} (i : finset ℕ) (p p' : ports υ) : Prop :=
  ∀ x ∈ i, p.nth x = p'.nth x

theorem eq_at_refl (υ : Type*) (i : finset ℕ) (p : ports υ) :
 eq_at i p p :=
 λ (υ : Type*) (i : finset ℕ) (p : ports υ),
  id (λ (x : ℕ) (h : x ∈ i), eq.refl (list.nth p x))
```

This is, of course, of no help to our human brains. So let's try proving this theorem using tactic mode instead.

A Theorem about Reactors

```
theorem eq_at_refl
  (υ : Type*) (i : finset ℕ) (p : ports υ) :
  eq_at i p p :=
  begin




  end
```

```
υ : Type*
i : finset ℕ
p : ports υ


⊢ eq_at i p p
```

We enter tactic mode using these `begin` and `end` delimiters. Once we're in tactic mode, we can see why Lean is also called a *proof assistant*: It's going to help us keep track of our proof by visualizing what is called the *goal state* on the right. Here we get information about the hypotheses/assumptions we've made so far, as well as the remaining goal (after the ⊢) we need to prove in order to

complete the proof of the theorem. In the beginning, our goal is precisely the proposition of the theorem. Our aim is now to simplify this goal, to the point where we can prove it by using other theorems or axioms. So the first step we'll take here, is to unfold the definition of `eq_at`.

A Theorem about Reactors

```
theorem eq_at_refl
  (υ : Type*) (i : finset ℕ) (p : ports υ) :
  eq_at i p p :=
  begin
    unfold eq_at,



  end
```

```
υ : Type*
i : finset ℕ
p : ports υ


⊢ ∀ (x : ℕ), x ∈ i →
   p.nth x = p.nth x
```

Unfolding the definition changed our goal. Hence, we now need to prove a specific ∀-proposition. Now, this is where I would like your feedback as intelligent human proof assistants: If you're trying to prove a proposition *for all elements* of some set (or *type* in this case), how do you go about that? Exactly, you introduce an arbitrary but fixed element from that set and show that the theorem holds for that arbitrary element. We can do exactly this with the `intro` tactic.

A Theorem about Reactors

```
theorem eq_at_refl
  (υ : Type*) (i : finset ℕ) (p : ports υ) :
  eq_at i p p :=
  begin
    unfold eq_at,
    intro x,



  end
```

```
υ : Type*
i : finset ℕ
p : ports υ
x : ℕ


⊢ x ∈ i → p.nth x = p.nth x
```

As you can see, not only does this change our goal state, but we've also introduced a new assumption. So now we need to prove an implication. Again I would like to get some feedback from you: If you're trying to prove an implication, how do you go about that? Exactly, you assume the premise to be true and then show that the consequence must hold. For this, we can again use the `intro` tactic.

A Theorem about Reactors

```
theorem eq_at_refl
  (υ : Type*) (i : finset ℕ) (p : ports υ) :
  eq_at i p p :=
  begin
    unfold eq_at,
    intro x,
    intro h,


  end
```

```
υ : Type*
i : finset ℕ
p : ports υ
x : ℕ
h : x ∈ i

⊢ p.nth x = p.nth x
```

Now all that's left to show is that `p.nth x` is equal to itself. There's a really simple way to do this, but I'll take the longer route here, just to demonstrate some more tactics. First of all, there's a really nice theorem in Lean called `congr_arg`, which states that if a function is called with equal arguments, then the result must be the same. We can apply this theorem to out goal using the `apply` tactic.

A Theorem about Reactors

```
theorem eq_at_refl
  (υ : Type*) (i : finset ℕ) (p : ports υ) :
  eq_at i p p :=
  begin
    unfold eq_at,
    intro x,
    intro h,
    apply congr_arg,

  end
```

```
υ : Type*
i : finset ℕ
p : ports υ
x : ℕ
h : x ∈ i

⊢ x = x
```

This is the point where you get to experience Lean in its full glory, because we aren't done yet. We now need to prove that the arguments to the function are in fact equal, i.e. that `x = x`. How do you prove that a thing is equal to itself? — by the definition of equality. In Lean, the equality-proposition is a defined just like the `or` and `true` propositions: as an inductive type. It has a constructor that we can call, where we pass in some object and in return get a proof that that object is equal to itself. We can tell Lean that this instance is precisely our remaining goal by using the `exact` tactic.

A Theorem about Reactors

```
theorem eq_at_refl
  (υ : Type*) (i : finset ℕ) (p : ports υ) :
  eq_at i p p :=
  begin
    unfold eq_at,
    intro x,
    intro h,
    apply congr_arg,
    exact eq.refl x
  end
```

goals accomplished 🎉

Here you can see that the constructor for the equality type `eq` is called `refl`. What you can also see is what can be a great source of relief when proving complicated theorems: the message that we've proven the theorem. As you get more skilled in Lean, you get better at using tactic mode. For example, this theorem can actually be proven in a single line. But when it comes down to it, tactics are always just a mechanism for creating proof instance. Hence, proving determinism for the Reactor model works exactly the same way. First, we create a `theorem`, where the type is precisely the statement we're trying to prove.

Determinism in Reactors

```
theorem determinism
 (σ : inst.network υ) (p p' : prec_func υ) (t t' : topo_func υ) :
 σ.run p t = σ.run p' t' := ...
```

In the case of `determinism`, what we're proving is that running a certain type of reactor network (`inst.network`) always produces the same outputs, independently of certain externalities (`prec_func` and `topo_func`). And second, we prove the statement using tactic mode. I won't show the proof of `determinism` here, because formalizing the small subset of the Reactor model and proving `determinism` actually took about 2000 lines of code — but the approach was exactly what we've seen in the previous slides. I'm guessing there might be some questions about this, but for now we've reached the end of the slides. So for the main part of this talk, we can consider all goals accomplished.

Determinism in Reactors

```
theorem determinism
 (σ : inst.network υ) (p p' : prec_func υ) (t t' : topo_func υ) :
 σ.run p t = σ.run p' t' := ...
```

goals accomplished 🎉