UNIVERSITY OF BUCHAREST

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Security and Applied Logic

MASTER'S THESIS

Mechanizing Many-Sorted Polyadic Hybrid Logic in the Lean 4 Prover

Student

Andrei-Alexandru Oltean

Advisor

Prof. dr. Ioana Leuștean

Bucharest, 2025

Abstract

This thesis presents a formalization of many-sorted polyadic hybrid modal logic within the Lean 4 proof assistant. Our primary contribution is a machine-checked proof of its soundness theorem and the development of the necessary framework for its Henkin completeness proof. While the soundness proof is fully mechanized, we outline definite directions for future work to complete the formalization of completeness and showcase its applicability to the operational semantics of operational semantics. All code and formalizations are available in the public repository: https://github.com/alexoltean61/msphml-lean

Contents

Chapter 1

Introduction

The central objective of this thesis is to provide a Lean 4 formalization of *many-sorted polyadic hybrid modal logic*. This logic, recently introduced and studied in [12], extends traditional hybrid modal logic with polyadic modalities and many-sorted signatures.

Interest in such a system and its mechanization comes from two interconnected directions. First, this system promises to provide a highly expressive modal framework for defining the syntax and semantics of programming languages, and reasoning about properties of programs. It is therefore vital to ensure that it satisfies basic mathematical requirements, such as soundness, in order to ensure that its applications to program verification are correct and, in a sense, bug-free. With the work already published offering a purely mathematical treatment to such questions, more confidence is gained once the logic is formalized in a computer-based proof assistant and the relevant theorems are successfully checked.

Moreover, once the validity of fundamental mathematical results is machine-checked, the formal artifact can be used as a stepping stone in a broader effort to bring software verification to proof assistants, under the unifying framework of hybrid modal logic. The promise of many-sorted polyadic hybrid logic's expressivity makes it an attractive choice to formalize, opening the door to the application and automation of modal techniques to program verification.

1.1 State of the Art

This thesis is the natural continuation of previous work [15], in which we formalized a simpler, monadic and mono-sorted, system of hybrid modal logic. The present effort significantly generalizes that work, both in scope (by adopting a many-sorted, polyadic setting), and in methodology, by redeveloping the system from scratch rather than extending the prior implementation. This decision, while more demanding, enabled a more modular design that aligned better with our research goals of verifying the soundness and completeness theorems.

Being a formal system that is still the subject of ongoing theoretical research, there is no prior implementation of this particular logic in Lean 4, or some other proof assistant. In fact, to the best of our knowledge, no work has been published concerning the Lean formalization of any system of many-sorted logic. In this sense, the present project is novel.

Nonetheless, our work connects to a broader tradition of using proof assistants to specify and reason about the formal semantics of programming languages. We note, for instance, the Isabelle/HOL textbook *Concrete Semantics* [14] which follows [24] in its treatment of imperative semantics, and the *Software Foundations* series [17] treating many of the same topics in Rocq.

In the Lean ecosystem, several projects have targeted the formalization of specific computational models, including the RISC-V instruction set [16], the Ethereum Virtual Machine (EVM) [13], and the RISC0 zkVM [19]. More recently, since version 4.22.0 released in July 2025, Lean has introduced experimental support for Hoare-logic style verification of Lean programs [22], reflecting growing interest in using Lean for program verification tasks.

We note, however, that all aforementioned projects are tied to concrete programming languages or architectures, and are not aimed at providing a general logical framework for arbitrary semantics. In this respect, our aims are much more closely aligned to those of matching logic [7], the system underlying the K Framework [20]. matching logic promises to be a unifying logical system in which the syntax and semantics of any programming language can be defined and reasoned about. Earlier research in polyadic hybrid logic showed clear parallels with matching logic [11], so the connection is already known. Matching logic, for its part, has seen formalizations in both Rocq [4] and Lean [8].

1.2 Our Contributions

The main contribution of this thesis is the full formalization of many-sorted polyadic hybrid modal logic in Lean 4. The formal artifact we provide can serve as a research tool in its own right, aiding to the study of hybrid logics in general.

In this respect, we claim that the work we present here has proven its merits. In the process of verifying the system's soundness, we identified several axioms and rules which, in their original form, led to unsound derivations. Through the Lean 4 formalization we have implemented, we were able to detect and correct these issues, thereby refining the logical system itself. We then successfully completed the formal proof of the soundness theorem.

With regard to completeness, we provide substantial progress towards a fully mechanized Henkin-style proof. While the complete formalization is ongoing, we establish key intermediate results and outline a concrete path to completing the argument.

All work presented in this thesis is made publicly available at `https://github.com/alexoltean61/msphml-lean`.

1.3 Thesis Structure

In Chapter 2, we establish some general preliminaries regarding the system of logic at hand, the Lean prover in general, and our design decisions for this formalization. Chapter 3 introduces the syntax and semantics of the logic, along with our mechanization thereof. Chapter 4 focuses on the core theoretical results, presenting our complete soundness proof and the framework for completeness. Finally, Chapter 5 discusses applications to operational semantics and outlines directions for future research.

Chapter 2

Preliminaries

Throughout this thesis, we will often abbreviate *many-sorted polyadic hybrid modal logic* by `MSPHML`.

2.1 Many-Sorted Polyadic Hybrid Modal Logic

The language of `MSPHML` extends in several ways the basic modal language. (See [6] for the main reference on modal logics in general.) First, it is *polyadic*, that is, it may contain several modal operators, each of which may have arbitrary arity. Second, it is *many-sorted*; which means formulas are partitioned by sorts, and operators constrain the sorts of arguments they can be applied to (much like data in programming contexts is partitioned by *data types*, and functions expect their parameters to have specific types). Finally, it is *hybrid*, using nominal and variable symbols to reference the Kripke model states. In particular, we will present the syntax of $\mathscr{H}_\Sigma(@, \forall)$, which introduces the @ and $\forall$ binders for nominals and variables, respectively.

Many-sorted languages have been extensively studied from the viewpoint of universal algebra (see [9], [21]), so it should come as little surprise that in our modal setting we inherit the formulation of many-sorted signatures from the former. However, we note that `MSPHML` extends the usual definition of signatures to also include *constant nominals*, for reasons which have to do with the modal semantics of operators.

2.2 A Primer to Lean

Lean is a functional programming language, as well as a proof assistant based on the calculus of inductive constructions. It features dependent typing, inductive types and metaprogramming. Its dependent type system is classified along a non-cumulative hierarchy of universes, also known as sorts. The bottom-most universe is that of *propositions*, `Prop`. By the Curry-Howard isomorphism, a term `stmt : Prop` corresponds to a mathematical statement; and a term `pf : stmt` corresponds to a proof of the respective statement. All other universes apart from the bottom-most contain *data types*. Some data types are familiar to regular programming contexts, such as `Nat` or `Bool`. These types usually live in the second to bottom-most type universe. Other types may not be as easily expressible in standard languages; take, e.g., `(α : Type) → List α`, which corresponds to functions taking a data type and returning a list of elements of that type, as a value.

2.2.1 A Hands-on Example: Propositional Logic

We give a hands-on example of some of the Lean features we will abundantly use: dependent types, inductive definitions, and indexed families of types. Consider the Lean code below:

```
inductive Propositional : Type → Type where
  | prop : α → Propositional α
  | bot  : Propositional α
  | imp  : Propositional α → Propositional α → Propositional α
```

It defines a family of types for propositional logic formulas. If you wish, you may consider the *indices* to this family of types (the `α` parameter) as serving as the terminal symbols of our propositional grammar. Non-terminals are represented by each of the three constructors, `prop`, `bot` and `imp`.

We can therefore construct propositional formulas with terminals taken from the type of characters, or from the type of natural numbers:

```
example : Propositional Char := .imp (.prop 'p') (.prop 'q')
example : Propositional Nat  := .imp (.prop 61) (.prop 11)
```

But also we can use less orthodox types as terminals, such as functions from naturals to naturals:

```
example : Propositional (Nat → Nat)  := .imp (.prop (· + 1)) (.prop (2 * ·))
```

We can also give a simple semantics for this propositional grammar. If we have a function `e : α → Bool` that maps terminals to boolean values, we can extend this to a mapping of full formulas to booleans. The Lean snippet below makes use of pattern matching over the `Propositional α` type to define such a mapping. Atomic formulas are mapped to their value under `e`, the bottom formula is mapped to boolean false, and implications make use of recursive calls to implement boolean "not φ or ψ":

```
def e_plus (e : α → Bool) : Propositional α → Bool
  | .prop p  => e p
  | .bot     => false
  | .imp φ ψ => !(e_plus e φ) || e_plus e ψ
```

2.2.2 Simple Proofs. Sigma and Pi Types

This very simple formalism already allows us to check prove some properties in Lean. The code below shows that $\varphi \to \varphi$ is a tautology. Lean makes no distinction between universal statements and functions to `Prop`, so the proof below is given as a function definition in functional programming style:

```
theorem id_taut {φ : Propositional α} : ∀ e, e_plus e (.imp φ φ) = true :=
  -- Note: this invokes a theorem in the standard library,
  -- Bool.not_or_self : ∀ (x : Bool), (!x || x)
  λ e => (e_plus e φ).not_or_self
```

The same may also be proved using so-called tactics, which are a higher-level interface to writing proof terms:

```
theorem id_taut' {φ : Propositional α} : ∀ e, e_plus e (.imp φ φ) = true :=
  by
  intro e        -- Mathematically, says: "Let e be an evaluation"
  unfold e_plus -- Unfolds the definition of e_plus
  simp -- Invokes Lean's simplifier to look for equalities that match the goal
```

Let us try to prove an existential statement: ”there exists a formula which is a tautology”. An existential is proved by providing a *witness* (in our case, we provide $\bot \to \bot$), and a *proof* that the witness satisfies the required property (in our case, we reference `id_taut` we proved earlier):

```
theorem exists_taut : ∃ φ : Propositional α, (∀ e, e_plus e φ = true) :=
  ⟨.imp .bot .bot, id_taut⟩
```

The proof above is a very good example of dependent typing. Notice that the proof we provided as second element in the tuple above *depends* on the formula we provided as first element. For if we had chosen $\bot \to (\bot \to \bot)$ as our witness instead, the proof given in `id_taut` would no longer have applied.

In fact, this example showcases both dependent products (Σ types, which are almost identical to existential statements) and dependent arrows (Π types, which are the syntactic sugar behind universal statements *and* functions). We can rewrite the previous example in the following form, making all dependent types fully explicit:

```
def exists_taut_constructive :
  Σ' φ : Propositional α, (Π (e : α → Bool), e_plus e φ = true) :=
    ⟨.imp .bot .bot, id_taut⟩
```

In simple terms, Σ types define pairs where the type of the second element depends on the value of the first. Complementarily, Π types define functions whose return types depend on the input value.

2.2.3 Universe Polymorphism. Type Classes

Additionally, we will make use of Lean's support for *universe polymorphism*: definitions over types living in arbitrary universes. At the declarative level, this is enabled by means of *universe-level variables*, e.g., the variable u between curly braces in the example below. The example also makes use of *type-class instance synthesis*, the unnamed parameter of type `Inhabited α` between square brackets. `Inhabited` ensures that a type has at least one element, accessible via `Inhabited.default`. The square brackets indicate to Lean that it should try to automatically search for a proof of inhabitance among the proofs that it knows about, instead of requiring the user to provide it explicitly. So the function below takes an inhabited type in any universe, and returns the list containing only its default element:

```
def default_list.{u} (α : Type u) [Inhabited α] : List α :=
    [Inhabited.default]
```

An example of an inhabited type is `Nat`, the natural numbers. Moreover, `Nat` lives in universe level 1, that of data types. If we check the type of applying `default_list` to `Nat`, we see that it is `List Nat`, as expected:

```
#check Nat
-- Output: Nat : Type
#check default_list Nat
-- Output: default_list Nat : List Nat
```

To understand where universe polymorphism comes into play, let us write an `Inhabited` instance for `Type 2 → Nat`, which lives in a higher universe than `Nat`, and notice that `default_list` still successfully applies to this new type:

```
#check (Type 2 → Nat)
-- Output: Type 2 → ℕ : Type 3
instance : Inhabited (Type 2 → Nat) where
    default := λ _ => 0
#check default_list (Type 2 → Nat)
-- Output: default_list (Type 2 → ℕ) : List (Type 2 → ℕ)
```

2.3 Our Design Principles

In the implementation of the language, we tried to follow a few guiding principles.

First, from our earlier experience with formalizing mono-sorted hybrid logic [15] we learned that simplicity in defining the syntax comes at the cost of sacrificing generality. In this previous implementation, the sets which make up the signature of the language were fixed, implemented as structures around the Nat type. While this was adequate for basic purposes, e.g. proving theorems in the deductive system and even proving soundness, this approach proved insufficiently general to support the Henkin-style completeness theorem we needed. This would be unsurprising: in effect the implementation was limited to just *one* hybrid language, while the techniques required in the completeness proof called for *expanding* the language with new symbols.

For this renewed effort at implementing a hybrid language in Lean, we were therefore guided by a simple first slogan:

Syntax should be modular by definition.

It should be relatively easy to define different languages around different signatures; to define language expansions or restrictions; and concrete types should not, wherever possible, be built into the definition. Universe polymorphism is preferred.

The second slogan is:

Well-sorted formula = *Well-typed term*

Lean's rich dependent typing system makes it a very promising choice for mechanizing many-sorted syntax like `MSPHML`'s. We intend to make complete use of its potential, striving to ensure well-sortedness of formulas is guaranteed entirely by Lean's type checker. Formulas should be sorted *by design*, as they are on paper, and no proofs of well-sortedness should ever be required to be separately supplied.

Finally:

Representational distance should be minimized.

As much as possible, the system defined in Lean should not lose its original modal flavour. It may be tempting, for example, to allow partial application of polyadic operators, as in lambda calculus, as such a choice may ease the task of defining the syntax. We strived to resist such temptations.

Chapter 3

Syntax and Semantics

In the sequel, we will be following the presentation in [12], where many-sorted polyadic hybrid logic is introduced, to define its syntax and semantics. After laying down the definitions, we will discuss our design choices for implementing this system in Lean 4.

3.1 Syntax

We begin by providing the definition of *signatures with constant nominals*, as will be used throughout this thesis.

Definition 3.1.1 (Signatures with constant nominals)**.** A **signature with constant nominals** is a triple (S, Σ, N), where:

- S is a non-empty, countable set, called the **sort set**;
- Σ is an $S^* \times S$-indexed family of countable sets; i.e., $\Sigma = (\Sigma_{w,s})_{w \in S^*, s \in S}$. Each element of these sets is called an **operator**;
- N is an S-indexed family of non-empty, countable sets; i.e. $N = (N_s)_{s \in S}$. Each element of these sets is called a **constant nominal**.

In the definition above, by S^* we mean the set of *words* given by S: the finite (possibly empty) sequences of elements in S. We will denote the empty sequence by λ.

Given a signature (S, Σ, N), in order to define a full-fledged hybrid language, we require three more S-sorted sets, consisting of the *propositional*, (non-constant) *nominal*, and *state variable* symbols, respectively: $(PROP)_{s \in S}$, $(NOM)_{s \in S}$ and $(SVAR)_{s \in S}$. We require that $PROP_s$ and NOM_s are countable; and $SVAR_s$ is denumerably infinite.

Having fixed these, we can introduce the grammar of $\mathscr{H}_\Sigma(@, \forall)$, the `MSPHML` language which is our object of study.

Definition 3.1.2 ($\mathscr{H}_\Sigma(@, \forall)$ formulas)**.** The set of formulas of sort $s \in S$ is:

$$\varphi_s := p \mid j \mid y \mid \neg\varphi_s \mid \varphi_s \vee \varphi_s \mid \sigma(\varphi_{s_1}, \dots, \varphi_{s_n}) \mid @^s_k \varphi_t \mid \forall x \varphi_s$$

Where the signature Σ is otherwise clear, we will often silently drop the subscript and speak of "$\mathscr{H}(@, \forall)$ formulas".

Where $p \in PROP_s$, $j \in NOM_s \cup N_s$, $k \in NOM_t \cup N_t$, $y \in SVAR_s$, $x \in SVAR_t$ and $\sigma \in \Sigma_{s_1 \dots s_n, s}$. Derived propositional operators are defined as usual. The dual binder $\exists$ is defined as $\exists x\varphi := \neg\forall x \neg\varphi$. For all operators $\sigma \in \Sigma_{s_1 \dots s_n, s}$, the dual operator $\sigma^\Box$ is defined as $\sigma^\Box(\varphi_1, \dots, \varphi_n) = \neg\sigma(\neg\varphi_1, \dots, \neg\varphi_n)$.

Notice that interaction between sorts happens in several ways. First, the satisfaction operator @ acts also as a "sort-casting" operator, with taking a nominal and a formula living in some sort s, and living in an arbitrary sort t. Next, we have modal operators and FOL-like binders. When applied to arguments with sorts that match their domains, modal operators produce formulas living in the sort given by their range. Furthermore, the bound variables in universal or existential formulas are allowed to take any sort.

With the exception of requiring substitution to be sorted, substitution and substitutability are defined as usual in hybrid logics with binders (see [5], besides the original paper [12]). Similarly, the idea of free and bound variables is standard.

3.2 Syntax Definition in Lean

Let us look at our choices for defining syntax. The crux of this is implemented in the *Language.Signature* and *Language.Form* modules.

We fix an arbitrary type α to serve as the alphabet of our language. All symbols, including ones used for sorts, modal operators, and variables, will be taken from type α. Moreover, we impose no restriction on the universe of type α, making all syntax definitions universe polymorphic.

The definition of (S, Σ, N)-signatures is given below.[1] For brevity, we omit the countability and non-emptiness restrictions on S, Σ and N, but we note that the actual Lean implementation requires them.

```
structure Signature (α : Type u) where
  S     : Set α
  «Σ»   : List S → S → Set α
  N     : S → Set α
```

We use `S` as the set of sorts. For this reason, we also make heavy usage of dependent functions from `S` to some other type to define the S-indexed families we require. Let us focus briefly on the «Σ» field, which we use as our set of modal operators. For each operator domain (the `List S` parameter) and range (the `S` parameter), it defines a set of operator symbols (`Set α`). As an example, consider the following context:

```
variable (sig : Signature String)
variable (s₁ s₂ : sig.S)
variable (σ : sig.«Σ» [s₁, s₂] s₂)
```

It assumes a (S, Σ, N)-signature named `sig`, two sorts s_1, $s_2 \in S$ and an operator $\sigma \in \Sigma_{s_1 s_2, s_2}$. Similarly, a variable typed as below denotes a constant nominal $c \in N_{s_2}$:

```
variable (c : sig.N s₂)
```

We also note that a different approach is possible:

```
structure Signature' (α : Type u) where
  S       : Set α
  «Σ»     : Set α
  arity   : «Σ» → ℕ
```

[1]We are required to use guillemets in «Σ» to distinguish the variable name from the reserved keyword Σ, the type constructor for dependent products.

```
    domain : (σ : «Σ») → (Fin (arity σ)) → S
    range  : «Σ» → S
    -- and so on
```

We decided against this method, primarily because it introduces a lot of definitional overhead. We believe we found a cleaner and more natural representation.

We keep track of the $PROP$, NOM and $SVAR$ sets of symbols in a structure called `Symbols`:

```
structure Symbols (α : Type u) where
  signature  : Signature α
  prop : (s : signature.S) → Set α
  nom  : (s : signature.S) → Set α
  svar : (s : signature.S) → Set α
```

Note the many-sorted signature being itself part of this structure. With a `Symbols` structure on hand, we are almost ready to give the definition of `MSPHML` formulas.

Before doing so, however, we start with a comment on our design choices. In line with the "Well-sorted formula = Well-typed term" slogan, we intend to specify an *indexed family of types*: for each sort, its own type of formulas. But a difficult problem then arises: how do we specify the type of the polyadic application constructor? Let us think of our operator $\sigma \in \Sigma_{s_1 s_2, s}$ from earlier. Intuitively, we apply σ to a list of formulas φ_1, φ_2, with φ_1 of sort s_1 and φ_2 of sort s_2, and obtain $\sigma(\varphi_1, \varphi_2)$ of sort s.

It's unclear how we could specify this argument list, since its elements are supposed to have *different types*, while lists must be type homogenous. Dependent arrows from a finite type would be a possible solution; but earlier we rejected such an approach for introducing definitional overhead.

We settled for the following design. We defined a family of types for *lists of formulas*, indexed by *lists of sorts*. Proper formulas, as defined mathematically in Chapter 3, are *singleton* lists of formulas. Extending on the grammar given in Chapter 3, we introduce an additional `cons` constructor, used to construct non-singleton formula lists. Fulfilling our intuition, an operator $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ will expect to be applied to a formula list of sorts $s_1, ..., s_n$, constructing a formula list of sorts s. Since this is a singleton list, such an operator application is guaranteed to be a proper formula.

We give the Lean definition of formula lists (`FormL`) and of proper formulas (`Form`) below:

```
inductive FormL (symbs : Symbols α) : List symbs.signature.S → Type u
| prop : symbs.prop s → FormL symbs [s]
| nom  : symbs.nominal s → FormL symbs [s]
| svar : symbs.svar s → FormL symbs [s]
| appl : symbs.signature.«Σ» (h :: t) s → FormL symbs (h :: t) → FormL symbs [s]
| or   : FormL symbs [s] → FormL symbs [s] → FormL symbs [s]
| neg  : FormL symbs [s] → FormL symbs [s]
| at   : symbs.nominal t → FormL symbs [t] → FormL symbs [s]
| bind : symbs.svar t → FormL symbs [s] → FormL symbs [s]
| cons : FormL symbs [s₁] → FormL symbs (s₂ :: t) → FormL symbs (s₁ :: s₂ :: t)

abbrev Form (symbs : Symbols α) (s : symbs.signature.S) := FormL symbs [s]
```

From the definition above, note that it is impossible to construct *unsorted* formulas: terms of type `FormL symbs []`. Derived operators are defined as expected. Using notations defined via Lean metaprogramming, formulas were made to resemble as much as possible their usual mathematical representation.

Let us exemplify the grammar and its notations. Building on our previous example, consider the following variable context:

```
variable (symbs : Symbols String)
variable (s₁ s₂ : symbs.signature.S)
variable (σ : symbs.signature.«Σ» [s₁, s₂] s₂)
variable (j : symbs.nominal s₁)
variable (k : symbs.nominal s₂)
variable (p : symbs.prop s₂)
```

We have $\sigma \in \Sigma_{s_1 s_2, s_2}$, $j \in NOM_{s_1} \cup N_{s_1}$, $k \in NOM_{s_2} \cup N_{s_2}$ and $p \in PROP_{s_2}$. We construct the (well-sorted) formula $p \wedge \sigma(j, k)$, and we query Lean for its type:

```
#check p ∧ ℋ⟨σ⟩ (j, k)
```

The Lean infoview will correctly let us know the term has type `FormL symbs [s₂]`. By stripping this term off its notational sugar and exposing the bare constructor applications, we would obtain:

```
#check (FormL.prop p).and $ FormL.appl σ ((FormL.nom j).cons (FormL.nom k))
```

As expected, attempting to check the type of $p \wedge \sigma(j, j)$ will fail:

```
#check p ∧ ℋ⟨σ⟩ (j, j)
```

With Lean complaining, correctly, that the second argument to σ has sort s_1 instead of required sort s_2.

A final note on this sorted syntax. If we have `φ : Form symbs s₁`, `ψ : Form symbs s₂`, and further a proof `pf : s₁ = s₂`, φ and ψ will nonetheless be terms of *different* types. When this happened, we used the ▸ macro (see [2]) to cast them to the same type. Its usage, however, is generally *not* recommended due to difficulty in constructing proofs with casted terms. In the future, a principled solution that avoids using the cast macro should be developed. While reasoning with casts per-se did not prove problematic in the particular instances that we resorted to it, equality between types of formulas did lead to problems when we defined syntax extensions. A more thorough discussion on this topic is provided in Section 4.4.1.

3.3 Semantics

We outline the Kripke-style semantics for the language just introduced.

Definition 3.3.1 (Frames)**.** A (S, Σ, N)-frame is a tuple $\mathcal{F} = (W, (R_\sigma)_{\sigma\in\Sigma}, N^{\mathcal{F}})$, where:

- $W = (W_s)_{s\in S}$ is an S-sorted set of **worlds**, each set assumed non-empty;
- $R_\sigma \subseteq W_s \times W_{s_1} \times \cdots \times W_{s_n}$, for all $\sigma \in \Sigma_{s_1\ldots s_n,s}$, is the **accessibility relation** of operator σ;
- $N^{\mathcal{F}} = (N^{\mathcal{F}}_s)_{s\in S}$ is an S-sorted function, with $N_s : N_s \to \mathcal{P}(W_s)$ for all s, and $N_s(c)$ a *singleton* for all $c \in N_s$. This is the **valuation** of constant nominals.

Definition 3.3.2 (Models)**.** A (S, Σ, N)-model based on a frame $\mathcal{F} = (W, (R_\sigma)_{\sigma\in\Sigma}, N^{\mathcal{F}})$ is a tuple $\mathcal{M} = (\mathcal{F}, V)$, where $V = (V_s)_{s\in S}$ is an S-sorted function (the **valuation** function), with $V_s : PROP_s \cup NOM_s \to \mathcal{P}(W_s)$. For all $k \in NOM_s$, $V_s(k)$ is required to be a *singleton*.

We will often specify the frame together with the model, writing $\mathcal{M} = (W, (R_\sigma)_{\sigma\in\Sigma}, N, V)$.

While the valuation function interprets nominals and propositional variables, in the case of state variables we define a Tarski-style assignment function, as customary in hybrid settings:

Definition 3.3.3 (Assignment function)**.** Given a (S, Σ, N)-frame $\mathcal{F} = (W, (R_\sigma)_{\sigma\in\Sigma}, N^{\mathcal{F}})$, an assignment is an S-sorted function $(g_s)_{s\in S}$, $g_s : SVAR_s \to W_s$.

Definition 3.3.4 (Satisfaction relation)**.** Given a model $\mathcal{M} = (W, (R_\sigma)_{\sigma\in\Sigma}, N, V)$, an assignment function g and $w \in W_s$:

- $\mathcal{M}, g, w \models^s a$ iff $w \in V_s(a)$, for $a \in PROP_s \cup NOM_s$;
- $\mathcal{M}, g, w \models^s c$ iff $w \in N_s^{\mathcal{F}}(c)$, for $c \in N_s$;
- $\mathcal{M}, g, w \models^s x$ iff $w = g_s(x)$, for $x \in SVAR_s$;
- $\mathcal{M}, g, w \models^s \neg\varphi$ iff $\mathcal{M}, g, w \not\models^s \varphi$;
- $\mathcal{M}, g, w \models^s \varphi \vee \psi$ iff $\mathcal{M}, g, w \models^s \varphi$ or $\mathcal{M}, g, w \models^s \psi$;
- $\mathcal{M}, g, w \models^s \sigma(\varphi_{s_1}, \dots, \varphi_{s_n})$ iff there exist $w_1, \dots, w_n \in W_{s_1} \times \dots \times W_{s_n}$ such that $(w, w_1, \dots, w_n) \in R_\sigma$ and $\mathcal{M}, g, w_i \models^{s_i} \varphi_i$ for all $1 \le i \le n$;
- $\mathcal{M}, g, w \models^s @_k^s\varphi$ iff $\mathcal{M}, g, u \models^t \varphi$, where k and φ have sort t and $V_t^N(k) = \{u\}$;
- $\mathcal{M}, g, w \models^s \forall x\varphi$ iff $\mathcal{M}, g', w \models^s \varphi$ for all $g' \rightsquigarrow^x g$.

Remark 1. The satisfaction of derived propositional operators follows immediately from this definition and is standard. Similarly, it readily follows that:

- $\mathcal{M}, g, w \models^s \exists x\varphi$ iff there exists $g' \rightsquigarrow^x g$ such that $\mathcal{M}, g', w \models^s \varphi$

Remark 2. It is important to pay attention to the satisfaction of dual operators. While σ can be seen as a generalization of *diamond* in the basic modal language, $\sigma^{\Box}$ does not generalize $\Box$ in quite the same way:

- $\mathcal{M}, g, w \vDash^s \sigma^{\Box}(\varphi_{s_1}, \dots, \varphi_{s_n})$ iff for all $w_1, \dots, w_n \in W_{s_1} \times \dots \times W_{s_n}$ such that $(w, w_1, \dots, w_n) \in R_\sigma$, there exists $1 \leq i \leq n$ such that $\mathcal{M}, g, w_i \vDash^{s_i} \varphi_i$.

Proof. Unfold the definition of the $\sigma^{\Box}$ and use the satisfaction clauses for negations and operator applications. $\Box$

In the sequel, we will frequently treat constant and non-constant nominals unitarily, speaking of "nominals" as $NOM \cup N^{\mathcal{F}}$. As well, to avoid notational kludge, we will talk of a single valuation function $V : PROP \cup NOM \cup N$.

Remark 3. At this point, we should be able to see the reason for introducing the constant nominals N into the usual many-sorted signatures (S, Σ). Modally, we interpret constant operators $\sigma \in \Sigma_{\lambda,s}$ into *sets* of worlds. Yet from an algebraic point of view, we expect constant to denote a *single* object. This is what hybridization and nominals guarantee: nominals are constrained to be true at a *single* world. Furthermore, the interpretation of constant nominals does not vary across models, being fixed in the frame.

Definition 3.3.5 (Validity)**.** Let $\mathbb{C}$ be a class of models. We write $\vDash^s_{\mathbb{C}} \varphi$ and say "φ is *valid in class* $\mathbb{C}$", if, for any model $\mathcal{M} \in \mathbb{C}$, and any $w \in W$ and assignment g, we have $\mathcal{M}, g, w \vDash^s \varphi$.

We say φ is *valid in a model* $\mathcal{M}$ if, for all $w \in W$ and assignments g, we have $\mathcal{M}, g, w \vDash^s \varphi$.

We say φ is *valid in a a frame* $\mathcal{F}$ if, for all models $\mathcal{M}$ based on $\mathcal{F}$, we have that $\mathcal{M} \vDash^s \varphi$.

Definition 3.3.6 (Local entailment)**.** If Γ is a set of formulas of sort $s \in S$, we say that "Γ *entails* φ in class $\mathbb{C}$", if, for all models $\mathcal{M} \in \mathbb{C}$ and all $w \in W$ and assignments g, whenever $\mathcal{M}, g, w \vDash^s \psi$ for all $\psi \in \Gamma$, we have $\mathcal{M}, g, w \vDash^s \varphi$. This is the *local entailment* relation.

Note that entailment may also be defined globally thus: For all $\mathcal{M} \in \mathbb{C}$, whenever $\mathcal{M} \vDash^s \psi$ for all $\psi \in \Gamma$, we have $\mathcal{M} \vDash^s \varphi$. This is a different relation, which will not be the focus of our work.

Definition 3.3.7. Let Λ be an S-sorted set of $\mathcal{H}(@, \forall)$ formulas. By $Mod(\Lambda)$ we mean the class of models in which every formula in Λ is valid. By $Fr(\Lambda)$ we mean the class of models based on frames where every formula in Λ is valid. Clearly, $Fr(\Lambda) \subseteq Mod(\Lambda)$.

3.4 Semantics Definition in Lean

Let us turn to the definition of frames, models and the satisfaction relation. The first aspect we will pay attention to lies in the many-sorted accessibility relation. For an operator $\sigma \in \Sigma_{s_1 \dots s_n, s}$, we introduced R_σ earlier as a subset of $W_s \times W_{s_1} \times \dots \times W_{s_n}$. Formalizing this product required a bit of machinery. We defined the type `WProd` as below, that takes a sorted family of types (the `W` argument) and a list of sorts, maps the list with `W` and computes the *product type* of the resulting list of types:

```
abbrev WProd {signature : Signature α} (W : signature.S → Type u) :
  List (signature.S) → Type u
  | []         => PEmpty
  | [s]        => W s
  | s :: sorts  => W s × WProd W sorts
```

In effect, given W and a non-empty list $[s_1, \dots, s_n]$ `: List (signature.S)`, applying the list to `WProd W` will compute exactly the type $W_{s_1} \times \dots \times W_{s_n}$. If the list is empty, the computed product will be the empty type. `WProd` is heavily used in our semantics as the *product of sorted worlds* type (hence its name); in essence all it does is a list fold.

With this out of the way, we give the definition of frames below.

```
structure Frame (signature : Signature α) where
  W  : signature.S → Type u
  R  : signature.«Σ» dom range → Set (WProd W (range :: dom))
  Nm : {s : signature.S} → signature.N s → W s
```

Notice `W` is a sorted family of types *in universe level* u, which is in fact the same universe level as `α`, the alphabet of our language. Later on (see Chapter 4), we will define worlds using a quotient of the set of nominals. Due to this reason, it was necessary to ensure the levels of `W` and `α` coincide.

Models and assignment functions are defined straightforwardly:

```
structure Model (symbs : Symbols α) where
  Fr   : Frame symbs.signature
  Vp   : symbs.prop s → Set (Fr.W s)
  Vn   : symbs.nom s → Fr.W s
abbrev Assignment (M : Model symbs) :=
  {s: symbs.signature.S} → symbs.svar s → M.Fr.W s
```

Our notion of satisfaction is more general than the one presented mathematically earlier, since we chose to take *formula lists* as our basic type of formulas. Earlier, we gave a mathematical formulation for the satisfaction of a formula φ of sort s at a world $w \in W_s$. Instead, the mechanized definition specifies the satisfaction of a formula list $\varphi_1, ..., \varphi_n$ of sorts $s_1, ..., s_n$, at a product of worlds $(w_1, ..., w_n) \in W_{s_1} \times ... \times W_{s_n}$. This makes our task of specifying the satisfaction of polyadic operators much easier.

To see why, consider that the mechanized relation added the following clauses to the mathematical definition, with everything else unchanged:

- $\mathcal{M}, g, (w_1, ..., w_n) \vDash^{s_1,...,s_n} (\varphi_{s_1}, ..., \varphi_{s_n})$ iff $\mathcal{M}, g, w_1 \vDash^{s_1} \varphi_{s_1}$ and $\mathcal{M}, g, (w_2, ..., w_n) \vDash^{s_2,...,s_n} (\varphi_{s_2}, ..., \varphi_{s_n})$
- $\mathcal{M}, g, w \vDash^{s} \sigma(\varphi_{s_1}, ..., \varphi_{s_n})$ iff there exist $w_1, ..., w_n \in W_{s_1} \times ... \times W_{s_n}$ such that $(w, w_1, ..., w_n) \in R_\sigma$ and $\mathcal{M}, g, (w_1, ..., w_n) \vDash^{s_1,...,s_n} (\varphi_{s_1}, ..., \varphi_{s_n})$

We give the definition of the new clauses of `Sat` is below:

```
def Sat (M : Model symbs) (g : Assignment M) (w : WProd M.Fr.W sorts) :
  FormL symbs sorts → Prop
| .cons ψ ψs      => Sat M g w.1 ψ ∧ Sat M g w.2 ψs
| .appl σ arg     => ∃ w', Sat M g w' arg ∧ ⟨w, w'⟩ ∈ M.Fr.R σ
```

3.5 A Hilbert Proof System

Below we give the axiomatization of $\mathscr{H}(@, \forall)$.[2] [3] [4]

The logic of $\mathscr{H}(@, \forall)$	
Axioms:	
Prop	All propositional tautologies
K	$\vdash^s \sigma^\Box(\dots, \varphi \to \psi, \dots) \to \sigma^\Box(\dots, \varphi, \dots) \to \sigma^\Box(\dots, \psi, \dots)$
K@	$\vdash^s @^s_j(\varphi_t \to \psi_t) \to (@^s_j\varphi \to @^s_j\psi)$
Agree	$\vdash^s @^s_k @^{s'}_j \varphi_t \leftrightarrow @^s_j \varphi_t$
SelfDual	$\vdash^s @^s_j\varphi_t \leftrightarrow \neg @^s_j \neg\varphi$
Intro	$\vdash^s j \to (\varphi_s \leftrightarrow @^s_j\varphi_s)$
Back	$\vdash^s \sigma(\dots, @^{s_i}_j \psi_t, \dots)_s \to @^s_j\psi$
Ref	$\vdash^s @^s_j j$
Q1	$\vdash^s \forall x(\varphi \to \psi) \to (\varphi \to \forall x\psi)$, if x does not occur free in φ
Q2	$\vdash^s \forall x\varphi \to \varphi[y/x]$, if y is substitutable for x in φ
Name	$\vdash^s \exists x x$
Barcan	$\vdash^s \forall x\sigma^\Box(\varphi_1, \dots, \varphi_n) \to \sigma^\Box(\varphi_1, \dots, \forall x\varphi_i, \dots, \varphi_n)$, if x does not occur free in φ_j for $j \neq i$
Barcan@	$\vdash^s \forall x @_j\varphi \to @_j\forall x\varphi$
Nom	$\vdash^s @_k x \wedge @_j x \to @_k j$
Rules:	
MP	If $\vdash^s \varphi \to \psi$ and $\vdash^s \varphi$, then $\vdash^s \psi$
UG	If $\vdash^{s_i} \varphi_i$, then $\vdash^s \sigma^\Box(\varphi_1, \dots, \varphi_i, \dots, \varphi_n)$
BroadcastS	If $\vdash^s @^s_j\varphi_t$, then $\vdash^{s'} @^{s'}_j \varphi_t$
Gen@	If $\vdash^s \varphi$, then $\vdash^{s'} @^{s'}_j \varphi$
Name@	If $\vdash^s @^s_j\varphi$, then $\vdash^{s'} \varphi$, if $j \notin N$ and j does not occur in φ
Paste	$\vdash^s @_j\sigma(\dots, k, \dots) \wedge @_k\varphi \to \psi$, then $\vdash^s @_j\sigma(\dots, \varphi, \dots) \to \psi$, for $k \neq j$, $k \notin N$, and k does not occur in φ, ψ, or the ... formulas
Gen	If $\vdash^s \varphi$, then $\vdash^s \forall x\varphi$

If Λ is an S-sorted set of $\mathscr{H}(@, \forall)$ formulas, then by $\mathscr{H}(@, \forall) + \Lambda$ we mean the proof system obtained by extending $\mathscr{H}(@, \forall)$ with Λ as additional axioms, closed under the usual deduction rules. We write $\vdash^s_\Lambda \varphi$ to denote that φ is a theorem in the system

[2]In Barcan, the restriction on x was omitted in the original [12], where MSPHML was introduced. Using Lean, we found that earlier version was unsound and added the appropriate restriction.

[3]In Name@,the requirement of j not being a constant nominal was omitted from [12]. Using this Lean formalization, we determined it is crucial for soundness.

[4]In rule Paste, the following restrictions were previously missed: i. The requirement of k not being a constant nominal; ii. The requirement for k to not occur in the dotted formulas. Using this Lean formalization, we found that their omission makes the system unsound.

$\mathcal{H}(@, \forall) + \Lambda$.

Definition 3.5.1 (Local syntactic consequence)**.** Let Γ be a set of formulas of sort $s \in S$. The *local syntactic consequence* relation $\Gamma \vdash^s_\Lambda \varphi$ is defined as: there exist formulas $\varphi_1 \dots \varphi_n \in \Gamma$, such that $\vdash^s_\Lambda (\varphi_1 \wedge \dots \wedge \varphi_n) \rightarrow \varphi$.

Definition 3.5.2 (Consistency)**.** We say that Γ is Λ-*inconsistent* if $\Gamma \vdash^s_\Lambda \bot$. We say that Γ is Λ-*consistent* if it is not Λ-*inconsistent*.

Finally, we define soundness and completeness, the statements which link the semantics and the proof system together. In Chapter 4, we will devote our attention to proving them with respect to pure extensions, which will be introduced then.

Definition 3.5.3 (Frame-soundness and frame-completeness)**.** Let Λ be an S-sorted set of formulas, and let Γ be a set of formulas of sort $s \in S$.

We say $\mathcal{H}(@, \forall) + \Lambda$ is *sound* if the following implication holds, for all φ of sort s:

$$\Gamma \vdash^s_\Lambda \varphi \text{ implies } \Gamma \vDash^s_{Fr(\Lambda)} \varphi.$$

Conversely, we say $\mathcal{H}(@, \forall) + \Lambda$ is *complete* if the following implication holds, for all φ of sort s:

$$\Gamma \vDash^s_{Fr(\Lambda)} \varphi \text{ implies } \Gamma \vdash^s_\Lambda \varphi.$$

3.6 Proof System Definition in Lean. Contexts

Implementing the proof system stems little difficulty. We defined an inductive family for Proof objects, with a term of type `Proof Λ s ψ` corresponding to a formal derivation of φ within $\mathcal{H}(@, \forall) + \Lambda$.

However, some care is necessary to formalize axioms that require substituting one of the arguments to a modal operator. To enumerate them all, these are the axioms Barcan, Back and K, and rules UG and Paste. For simplicity, in this section it suffices to devote our attention only to axiom K:

$$\sigma(\dots, \varphi \to \psi, \dots) \to \sigma(\dots, \varphi, \dots) \to \sigma(\dots, \psi, \dots)$$

With our formalization of syntax, it is not immediately clear how this could be expressed, as it picks out an arbitrary occurrence of $\varphi \to \psi$ within σ's arguments, and substitutes it with its two sides. This is a non-trivial operation that would be tedious to implement naively in our custom-tailored `FormL` lists.

We note, however, its strong resemblance to the Framing rule of Matching μ-Logic, as in fact noted in [7], where Matching μ-Logic is introduced. The Framing rule has the advantage of being compactly expressible, by its use of so-called ”contexts”. We reproduce the definition of contexts and the Framing rule from [7].

Definition 3.6.1 (Matching logic: single symbol context)**.** A *context* C is a pattern with a distinguished placeholder variable $\square$. We write $C[\varphi]$ to mean the result of replacing $\square$ with φ without any α-renaming.

A *single symbol context* has the form $C_\sigma \equiv \sigma(\varphi_1, \dots, \varphi_{i-1}, \square, \varphi_{i+1}, \dots, \varphi_n)$ where $\sigma \in \Sigma_{s_1 \dots s_n, s}$ and $\varphi_1, \dots, \varphi_{i-1}, \varphi_{i+1}, \dots, \varphi_n$ are patterns of appropriate sorts.

Definition 3.6.2 (Matching logic: Framing rule)**.** Given a proof of $\varphi_1 \to \varphi_2$, a proof of $C_\sigma[\phi_1] \to C_\sigma[\phi_2]$ can be inferred.

Even though contexts have no direct equivalent in modal logic, we used the matching logic intuition to define a similar notion in our own setting.

From a purely engineering point of view, we found it unnecessary to deal with a placeholder variable $\square$ as is done in Matching logic. In this formalization, a term of type `Context φ ψ` can be seen as a *pointer* to a single occurrence of φ within the formula list ψ. It is a way to express the *fact* that φ occurs within ψ, as well as also constructing the precise *indication* to where it occurs. We have a very simple inductive definition for contexts, which we give below:

```
inductive Context (φ : Form symbs s) : FormL symbs sorts → Type u
  | refl : Context φ φ
  | head : Context φ (.cons φ ψ)
  | tail : Context φ ψ → Context φ (.cons χ ψ)
```

Let us show an example.

```
variable (symbs : Symbols α)
variable (s : symbs.signature.S)
variable (φ ψ χ : Form symbs s)
-- φ → ψ is identical to itself; it occurs within itself by .refl:
example : (φ → ψ).Context (φ → ψ) := Context.refl
-- φ → ψ is the head of formula list (φ → ψ, χ):
example : (φ → ψ).Context (φ → ψ, χ) := Context.head
-- More involved examples;
-- This points to the first occurrence of φ → ψ in (χ, φ → ψ, φ → ψ):
example : (φ → ψ).Context (χ, φ → ψ, φ → ψ) :=
  Context.tail Context.head
-- This points to the second occurrence of φ → ψ in (χ, φ → ψ, φ → ψ):
example : (φ → ψ).Context (χ, φ → ψ, φ → ψ) :=
  Context.tail $ Context.tail Context.refl
```

Contexts are useful as they allow us to define the substitution of a new formula for the one that it points to, obtaining a new `FormL`. We used notation directly inspired from Matching Logic: if `C : φ.Context ψ`, then `C[χ]` denotes the formula obtained by substituting φ with χ in ψ.

We can therefore obtain a neat expression of the K rule, and all others that require substitution of a specific argument to a modal operator. We indicate the way we formalized the K axiom, and note all other cases enumerated in the beginning of this section can be found in the `Proof.Hilbert` module:

```
inductive Proof (Λ : AxiomSet symbs) :
  (s : symbs.signature.S) → Form symbs s → Type u
  | k φ ψ χ
      (σ : symbs.signature.«Σ» _ s)
      (C : (φ → ψ).Context χ):
            Proof Λ s (ℋ⟨σ⟩□ χ → (ℋ⟨σ⟩□ C[φ] → ℋ⟨σ⟩□ C[ψ]))
```

We finish this section recalling the definition of the local consequence relation 3.5.1, $\Gamma \vdash^s_\Lambda \varphi$: there exist a choice of formulas $\varphi_1 \dots \varphi_n \in \Gamma$, such that $\vdash^s_\Lambda (\varphi_1 \wedge \dots \wedge \varphi_n) \to \varphi$.

In Lean, we call sets of formulas of the same sort `PremiseSets`. The choice of elements in Γ is formalized as a list of terms subtyped to the set Γ, and a proof that the list has no duplicates:

```
abbrev PremiseSet (symbs : Symbols α) (s: symbs.signature.S): Type u :=
  Set (Form symbs s)
abbrev FiniteChoice  (Γ : PremiseSet symbs s) := (L : List Γ) ×' L.Nodup
```

The conjunction of this list is then easily computable:

```
def List.conjunction {Γ: PremiseSet symbs s}: List Γ → Form symbs s
  | []       => ℋ⊤
  | ψ :: ψs => ψs.conjunction ∧ ψ
def FiniteChoice.conjunction {Γ: PremiseSet symbs s} :
  FiniteChoice Γ → Form symbs s :=
  λ ⟨ch, _⟩ => ch.conjunction
```

With all this machinery in place, syntactic consequence is naturally defined: there exists a finite choice of elements in Γ such that their conjunction implies φ:

```
def SyntacticConsequence {symbs : Symbols α} {s : symbs.signature.S}
  (Γ : PremiseSet symbs s) (Λ : AxiomSet symbs) (ψ : Form symbs s): Prop :=
  ∃ ch : FiniteChoice Γ, ⊢(Λ, s) (ch.conjunction → ψ)
```

We now have a complete formalization in Lean 4 of the syntax of MSPHML, its semantics and its proof system, as were defined in Chapter 3. We are now ready to put this formalization to work. In the next chapter, we will discuss our formalization of the soundness theorem, and progress we made towards the completeness theorem.

Chapter 4

Soundness and Completeness

The effort of rigorously verifying soundness and completeness in Lean led us to discover a few missing conditions in the original presentation given in [12]. Fortunately, none of these slight slips had a great effect on the overall standing of the system. We believe, nonetheless, that it is to the system's gain that such slight errors have been now spotted. In the following, we will report only the places where our proofs differ from the original. Some of the other proofs can be found in the technical appendix (Chapter A).

4.1 Soundness

In this section, all statements we give have been machine-checked.

As is standard, soundness is proved by structural induction on formulas. Before that, we will require some lemmas.

Lemma 4.1.1 (Agreement)**.** *Let $\mathcal{M} = (W, R, N, V)$ be a model, φ a formula of sort $s \in S$, $w \in W_s$ a world, and g, g' assignment functions. If g and g' agree on all variables occurring freely in φ, then:*

$$\mathcal{M}, g, w \models^s \varphi \textit{ iff } \mathcal{M}, g', w \models^s \varphi$$

Proof. By induction on the structure of φ. □

Lemma 4.1.2 (Valuation Agreement). *Let φ be a formula of sort $s \in S$ and $j \in NOM_t \cup N_t$ a nominal of sort $t \in S$. Assume j does not occur in φ. If $\mathcal{M}$ and $\mathcal{M}'$ are models such that $V(j) \neq V'(j)$ but otherwise agree, then for all $w \in W_s$, g:*

$$\mathcal{M}, g, w \models^s \varphi \text{ iff } \mathcal{M}', g, w \models^s \varphi$$

Proof. By induction on the structure of φ. □

Lemma 4.1.3 (Substitution). *Let $\mathcal{M} = (W, R, N, V)$ be a model, φ a formula of sort $s \in S$, $w \in W_s$ a world, and g, g' assignment functions. Let $x, y \in SVAR_t$ such that y is substitutable for x in φ. If $g \rightsquigarrow^x g'$, then:*

- *$M, g, w \models^s \varphi$ iff $M, g', w \models^s \varphi[y/x]$, where $g(x) = g'(y)$*
- *$M, g, w \models^s \varphi$ iff $M, g', w \models^s \varphi[j/x]$, where $g(x) = g'(j)$*

Proof. By induction on the structure of φ. □

Theorem 4.1.1 (Soundness). *Let Λ be an S-sorted set of axioms. The following propositions are true:*

1. *If $\vdash^s_\Lambda \varphi$, then $\models^s_{Fr(\Lambda)} \varphi$;*
2. *If $\Gamma \vdash^s_\Lambda \varphi$, then $\Gamma \models^s_{Fr(\Lambda)} \varphi$.*

Proof. 2. is an immediate consequence of 1. For 1., we only show the following cases. All others can be found in Chapter A. Let $\mathcal{M} \in Fr(\Lambda)$, g be an assignment function, w be a world of the right sort.

(Barcan): Assume $\mathcal{M}, g, w \models^s \forall x \sigma^\Box(\varphi_1, \dots, \varphi_i, \dots \varphi_n)$. After swapping the (implicit) universal quantifiers, this is equivalent to: for all $w_1, \dots, w_n \in W_{s_1} \times \dots \times W_{s_n}$ such that $(w, w_1, \dots, w_n) \in R_\sigma$, and for all $g' \rightsquigarrow^x g$, there exists an index j such that $\mathcal{M}, g', w_j$ satisfies the j-th formula in $\varphi_1, \dots, \varphi_i, \dots, \varphi_n$.

Since we want to prove $\mathcal{M}, g, w \models^s \sigma^\Box(\varphi_1, \dots, \varphi_i, \dots, \varphi_n)$, let $w_1, \dots, w_n$ be worlds of the right sorts accessible from w. By assumption, there is some j such that the j-th formula in $\varphi_1, \dots, \varphi_i, \dots, \varphi_n$ is satisfied by assignment g'. Now, we further know that x does not occur free in φ_l for all $l \neq i$. So we do case analysis on j.

If $j = i$, then since $\mathcal{M}, g', w_j \vDash \varphi_i$, it follows that $\mathcal{M}, g, w_j \vDash \forall x \varphi_i$, QED.

If $j \neq i$, then since x is not free in φ_j, and g, g' otherwise agree; we apply the Agreement Lemma (4.1.1) to $\mathcal{M}, g', w_j \vDash \varphi_j$ and obtain $\mathcal{M}, g, w_j \vDash \forall x \varphi_j$, QED.

(Name@): Assume $\vDash^s_{Fr(\Lambda)} @_j \varphi$. We want to show $\mathcal{M}, g, w \vDash^{s'} \varphi$, for any $\mathcal{M} \in Fr(\Lambda)$, g, w. Let us refer to the valuation function of $\mathcal{M}$ as V, and let us consider a model M' identical to $\mathcal{M}$, except for its own valuation V', which we take to be $V'(j) = w$ and $V'(s) = V(s)$ for any $s \neq j$. By hypothesis, j is not a constant nominal: $j \notin N_{s'}$. Therefore, its valuation is not part of the frame; and since $\mathcal{M} \in Fr(\Lambda)$, we are then guaranteed to have $\mathcal{M}' \in Fr(\Lambda)$ as well. Thus, by the initial assumption, $\mathcal{M}', g, w \vDash @_j \varphi$, iff $\mathcal{M}', g, V'(j) \vDash \varphi$, iff $\mathcal{M}', g, w \vDash \varphi$. We apply Valuation Agreement (Lemma 4.1.2) and obtain $\mathcal{M}, g, w \vDash \varphi$.

(Paste): Assume $\vDash^s_{Fr(\Lambda)} @_j \sigma(\dots, k, \dots) \wedge @_k \varphi \to \psi$, let $\mathcal{M} \in Fr(\Lambda)$, g, w, and assume further $\mathcal{M}, g, w \vDash^s @_j \sigma(\dots, \varphi, \dots)$. We want to show $\mathcal{M}, g, w \vDash^s \psi$.

By the second assumption, there exist $w_1, \dots, w_n \in W_{s_1} \times \dots \times W_{s_n}$ such that $(V(j), w_1, \dots, w_n) \in R_\sigma$; and for all l, we know that $\mathcal{M}, g, w_l$ satisfy the l-th formula among $\varphi_1, \dots, \varphi, \dots, \varphi_n$. We will denote the position that φ occurs among these formulas by m, so the corresponding world where it is satisfied will be w_m.

Let us refer to the valuation function of $\mathcal{M}$ as V, and let us consider a model M' identical to $\mathcal{M}$, except for its own valuation V', which we take to be $V'(k) = w_m$ and $V'(s) = V(s)$ for any $s \neq k$. The argument for $\mathcal{M}' \in Fr(\Lambda)$ is identical to the one given in the (Name@) case; since we have a similar hypothesis that k is not a constant nominal. Next, by the first assumption, we obtain proposition *(P)*: $\mathcal{M}', g, w \vDash^s @_j \sigma(\dots, k, \dots) \wedge @_k \varphi \to \psi$. We will show the left-hand side of this implication is satisfied, allowing us to obtain the satisfaction of the right-hand side.

We show $\mathcal{M}', g, w \vDash^s @_k \varphi$ first. This is equivalent to $\mathcal{M}', g, V'(k) \vDash^{s'} \varphi$, which by our choice of V' means $\mathcal{M}', g, w_m \vDash^{s'} \varphi$. By a hypothesis to Paste rule, we know that k does not occur in φ, so we are allowed to apply Valuation Variant (Lemma 4.1.2). This means our goal is equivalent to $\mathcal{M}, g, w_m \vDash^{s'} \varphi$, which we already know to be true.

For the other case, we show $\mathcal{M}', g, w \vDash^{s} @_j\sigma(\dots, k, \dots)$. This is the same as proving $\mathcal{M}', g, V'(j) \vDash^{s} \sigma(\dots, k, \dots)$. Further, we know by hypothesis that $j \neq k$, and so $V'(j) = V(j)$, which means we will prove $\mathcal{M}', g, V(j) \vDash^{s} \sigma(\dots, k, \dots)$. Take $w_1, \dots, w_n$ from earlier, for which we know $(V(j), w_1, \dots, w_n) \in R_\sigma$. Each of these worlds must satisfy the corresponding argument to σ. First, consider w_m: we must show that $\mathcal{M}', g, w_m \vDash^{s'} k$, which is immediately true by the way we took $V'(k)$. Next, consider w_l, for $l \neq k$. We know already that $M, g, w_l \vDash^{s_l} \varphi_l$. By a Paste rule hypothesis, we know k does not occur in φ_l. So by Valuation Variant (Lemma 4.1.2), we get $M', g, w_l \vDash^{s_l} \varphi_l$, as expected.

So the antecedent to proposition *(P)* is proved. Therefore, we obtain $\mathcal{M}', g, w \vDash^{s} \psi$. Once again, a Paste rule hypothesis guarantees that k does not occur in ψ. So we apply Valuation Variant one more time and get $\mathcal{M}, g, w \vDash^{s} \psi$, QED. □

4.2 Proving Soundness in Lean

The three lemmas are proved in the module `Soundness.Lemmas`. The formal statements of our soundness theorems are:

```
theorem Soundness {Λ : AxiomSet symbs} : ⊢(Λ, s) ψ → ⊨Fr(Λ) ψ
theorem StrongSoundness {Λ : AxiomSet symbs} : Γ ⊢(Λ) ψ → Γ ⊨Fr(Λ) ψ
```

Our mathematical presentation given above was written following the verified Lean implementation, while leaving aside some of the technical, implementation-specific work. Consider the proof for Name@:

```
| @nameAt s₁ s₂ j ψ noccψ _ ih =>
    intro M g w
    let M' := ⟨M.1.v_variant j w, Set.Elem.v_variant_modelclass_inv Λ M j w⟩
    let g' : Assignment M'.1 := g.v_variant j w
    let w' := (M'.1.Fr.WNonEmpty s₁).default
    specialize ih M' g' w'
```

```
simp only [Sat.at, M', v_variant_valuation] at ih
rw [v_variant_agreement noccψ w]
exact ih
```

It proceeds by defining the M' model with the valuation variant, while also introducing g' and w' definitions (identical to g and w, but made to match M''s set of worlds; Lean cannot tell that M and M''s set of worlds have not changed). It then specializes the inductive hypothesis (namely, $\vDash^s_{Fr(\Lambda)} @_j\varphi$) to M', g', and w', and after simplification with the satisfaction clauses of the @ operator, obtains $M', g', V'(j) \vDash^s \varphi$. Finally, using the variant agreement lemma, it rewrites this to $M, g, w \vDash^s \varphi$, which is exactly the statement we intended to prove.

Of the technical work that was left aside, we mention that singling out formulas occurring at a specific index within a list was somewhat complex in Lean, due to our handling of `Contexts`. We were forced to define a notion of *context isomorphism*, and prove various lemmas that amounted to the fact that isomorphic contexts behave identically (e.g., substituting a formula in two isomorphic contexts produces the result).

To get an idea of what we mean by ”technical statement”, consider the one below:

```
def subst_not_iso {φ : Form symbs s} {ψ : Form symbs s'}
    {τ : FormL symbs sorts} {C₁ : φ.Context τ}
    (C₂ : ψ.Context τ) (h : ¬C₁.iso C₂) :
  (δ : Form symbs s) → Σ' C₃ : ψ.Context C₁[δ], C₂.iso C₃
```

It says that if φ occurs somewhere among a list $\tau_1, \dots, \varphi, \dots, \tau_n$, and ψ occurs somewhere *else* among the same list (their contexts are not isomorphic), then there exist a formula δ of the same sort as φ and an occurrence of δ within $\tau_1, \dots, \delta, \dots, \tau_n$, such that δ occurs at the same index as φ (their contexts are isomorphic). Moreover, we proved this constructively by actually *producing* a `Context` for δ within said list, not just claiming its existence.

This kind of complexity is not very welcome, as it goes against our third slogan, ”Representational distance should be minimized”. However, for the time being, it allowed us to and formally reason about these axioms, and prove soundness of the system. The net result was that we currently have a complete, `sorry`-free proof of the statements above.

Crucially, the effort to formalize soundness in Lean uncovered several issues in the original axiomatization that led to unsoundness. We have resolved these issues via the sideconditions to axiom Barcan, and to rules Name@ and Paste that have been mentioned in a footnote in Chapter 4, where the proof system is introduced. Thanks to machine-checked theorem proving, we have managed to correct the logic of `MSPHML`.

4.3 Completeness

The outline of the proof is identical to the one in [12], but in certain places we reformulate the argument.

Definition 4.3.1 (Named models)**.** A model $\mathcal{M} = (W, (R_\sigma)_{\sigma\in\Sigma}, N, V)$ is *named* if, for all $s \in S$ and $w \in W_s$, there exists some $k \in NOM_s \cup N_s$ such that $w = V(k)$.

In other words, in a named model, all worlds are guaranteed to be named by a nominal (constant or non-constant).

Definition 4.3.2 (Pure formulas)**.** A formula is *pure* if it does not contain propositional variables. A *pure instance* of a formula is obtained by substituting nominals with nominals of the same sort. A formula is $\forall\exists$-*pure* if it is pure, or it has the form $\forall x_1 \dots \forall x_n \exists y_1 \dots \exists x_n \psi$, where ψ is pure and $\{x \in SVAR \mid x \text{ occurs in } \psi\} \subseteq \{x_1, \dots, x_n, y_1, \dots, y_n\}$.

Definition 4.3.3 (Named, pasted and @-witnessed sets)**.** Let $s \in S$ and Γ be a set of formulas of sort s. We say that:

- Γ is *named* if $j \in \Gamma$ for some $j \in NOM_s \cup N_s$;
- Γ is *pasted* if, whenever $@_k\sigma(\dots, \varphi, \dots) \in \Gamma$, there exists $j \in NOM_{s_i}$ such that $@_k\sigma(\dots, j, \dots) \in \Gamma$ and $@_j^s\varphi \in \Gamma$; for all $t \in S$, $\sigma \in \Sigma_{s_1 \dots s_n, t}$, $k \in NOM_t \cup N_t$, and φ of sort s_i;

- Γ is *@-witnessed* if, cumulatively:
  - If $@^s_k\exists\varphi \in \Gamma$, then there exists $j \in NOM_t$ such that $@^s_k\phi[j/x] \in \Gamma$; for all $s', t \in S$, $x \in SVAR_t$, $k \in NOM_{s'} \cup N_{s'}$ and φ of sort s':
  - For all $t \in S$, $x \in SVAR_t$, $@^s_{j_x}x \in \Gamma$ for some $j_x \in NOM_t$.

Definition 4.3.4 (Maximal consistent sets)**.** Let Λ be a set of axioms, $s \in S$ and Γ a set of formulas of sort s.

We say that Γ is Λ-*maximal consistent* if it is Λ-consistent, and any Γ' such that $\Gamma \subsetneq \Gamma'$ is not Λ-consistent.

In the following, we will abbreviate "Λ-maximal consistent set" by "Λ-MCS". Λ-MCS's have the following property we will make use of: for any Λ-MCS Γ, $@^s_jk \in \Gamma$ is an equivalence relation on nominals:

Proposition 4.3.1. *Let $j, k, u \in NOM_t \cup N_t$, and Γ a set of formulas of sort s which is a Λ-MCS. The following properties hold:*

1. $@_jj \in \Gamma$*;*
2. *If* $@_jk \in \Gamma$*, then* $@_kj \in \Gamma$*;*
3. *If* $@_jk \in \Gamma$ *and* $@_ku \in \Gamma$*, then* $@_ju \in \Gamma$*.*

Proof. Immediate using theorems (Nom), (Bridge) and (Sym) in [12], Lemma 1. □

Definition 4.3.5 (Lindenbaum extension)**.** Let Γ be a Λ-consistent set. We define a sequence of sets $(\Gamma_n)_{n\in\mathbb{N}}$ in a language extended with denumerably many new nominals. Let $\varphi_1, \ldots$ be an enumeration of all formulas of sort s in the extended language. We have:

- $\Gamma^0 = \Gamma \cup \{k_s\} \cup \{@^s_{j_x}x \mid x \in SVAR_t, t \in S\}$, where k_s is a new nominal of sort s, and $j_x \neq k_s$ is a distinct new nominal of sort t for each variable x;
- If $\Gamma^n \cup \{\varphi_{n+1}\}$ is inconsistent, $\Gamma^{n+1} = \Gamma^n$;

- Otherwise, $$\Gamma^{n+1} = \begin{cases} \Gamma^n \cup \{\varphi_{n+1}\} \cup \{@_j\sigma(\dots,\psi_{i-1},k_i,\psi_{i+1},\dots) \wedge @_k\psi_i \mid i \leq n\}, \\ \qquad \text{if } \varphi \text{ has the form } @_j\sigma(\psi_1,\dots,\psi_n) \text{ and} \\ \qquad k_i \text{ is a distinct new nominal for each } \psi_i; \\ \Gamma^n \cup \{\varphi_{n+1}\} \cup \{@_j\psi[k/x]\}, \\ \quad \text{if } \varphi \text{ has the form } @_j\exists x\psi \text{ and} \\ \quad k \text{ is a new nominal;} \\ \Gamma^n \cup \{\varphi_{n+1}\}, \text{otherwise.} \end{cases}$$

Note that the sets above are well-defined, since there will always be enough new nominals in the extended language.

Furthermore, we let $\Gamma^+ = \bigcup\limits_{n\in\mathbb{N}} \Gamma^n$. We call Γ^+ the *Lindenbaum extension* of Γ.

Lemma 4.3.1 (Extended Lindenbaum Lemma)**.** *Let* Γ *be a* Λ*-consistent set of formulas of sort* $s \in S$*. There exists a named, pasted, @-witnessed* Λ*-MCS* Γ^+ *in an extended language such that* $\Gamma \subseteq \Gamma^+$*.*

Proof. See [12], Lemma 2. □

Definition 4.3.6 (Henkin model)**.** Let Γ be a Λ-MCS of formulas of sort $s \in S$. For all $t \in S$, let $j, k \in NOM_t \cup N_t$, and let $\sim$ denote the relation $@^s_j k \in \Gamma$. We showed in Lemma 4.3.1 that $\sim$ is an equivalence relation.

The *Henkin model* $\mathcal{M}^\Gamma = (W^\Gamma, (R^\Gamma_\sigma)_{\sigma\in\Sigma}, N^\Gamma, V^\Gamma)$ is defined thus:

- $W^\Gamma_t = (NOM_t \cup N_t)/{\sim}$, for all $t \in S$;

- $(|j|, |j_1|, \dots, |j_n|) \in R^\Gamma_\sigma$ iff $@^s_j\sigma(j_1,\dots,j_n) \in \Gamma$, for all $\sigma \in \Sigma_{t_1\dots t_n,t}$;

- $N^\Gamma_t = \{|c|\}$, for all $c \in N_t$ and $t \in S$;

- $V^\Gamma_t(j) = \{|j|\}$, for all $j \in NOM_t$ and $t \in S$;

- $V^\Gamma_t(p) = \{|j| \mid @^s_j p \in \Gamma_s\}$, for all $p \in PROP_t$ and $t \in S$.

Additionally, assuming Γ is @-witnessed, the *Henkin assignment function* $g^\Gamma : SVAR \to W^\Gamma$ is defined as:

- $g^\Gamma(x) = |j|$, where $@^s_j x \in \Gamma$, for all $x \in SVAR_t, j \in NOM_t, t \in S$.

By the definition of @-witnessed sets, this function is guaranteed to be well-defined.

Lemma 4.3.2 (Truth lemma)**.** *Given a named, pasted, @-witnessed Λ-MCS Γ of formulas of sort $s \in S$, the following equivalence holds for any φ of sort $t \in S$ and $j \in NOM_t \cup N_t$:*

$$\mathcal{M}^\Gamma, g^\Gamma, |j| \vDash^t \varphi \quad \textit{iff} \quad @^s_j \varphi \in \Gamma$$

We need one more very important piece. We refer the reader to the original paper [12], Proposition 1, for its proof.

Proposition 4.3.2 (Pure formulas in $\mathcal{H}(@, \forall)$)**.** *Let $\mathcal{M} = (\mathcal{F}, V)$ be a named model and let φ be a $\forall\exists$-pure formula of sort $s \in S$. Then $\mathcal{M} \vDash^s \varphi$ iff $\mathcal{F} \vDash^s \varphi$.*

With all these preliminary lemmas in place, we are able to prove:

Theorem 4.3.1 (Strong frame-completeness for pure extensions)**.** *Let Λ be an S-sorted set of pure formulas, which we take as axioms, let Γ be a set of formulas of sort $s \in S$, and let φ be a formula of sort s. The following implication holds:*

$$\Gamma \vDash^s_{Fr(\Lambda)} \varphi \textit{ implies } \Gamma \vdash^s_\Lambda \varphi$$

4.4 Proving Completeness in Lean

Mechanizing the completeness theorem is a complex task. To the best of our knowledge, the only complete Lean implementation of a Henkin-style completeness theorem for a language with first-order capabilities (in fact, FOL itself) was in [10]. We took a top-down approach, focusing on first on the high-level flow of the proof and ensuring each step has a formal statement; before reaching down to the low-level details of actual `MSPHML` and proofs therein. Due to this reason, it should be noted that various results proved in [12] that pertain to the `MSPHML` proof system itself, are at this moment still *unverified* (for example, the theorems required in the proof of Proposition 4.3.1).

Let us break down this task into smaller subtasks, and discuss our approach and progress on each of them.

4.4.1 Language Extensions

A crucial part in the completeness proof outlined above lies in the existence of extended languages, particularly languages with denumerably many new nominals. In order to tackle this problem, we took the following approach. In module `Lindenbaum.Expansion.Def`, we defined a notion of *symbols morphism*. This is a set of functions that map the symbols of one language (its signature (S, Σ, N), along with the sets $PROP$, NOM and $SVAR$) to the symbols of another:

```
structure Symbols.morphism (S₁ : Symbols α) (S₂ : Symbols β) where
  morph_sort : S₁.signature.S → S₂.signature.S
  morph_op {dom rng} : S₁.signature.«Σ» dom rng →
        S₂.signature.«Σ» (dom.map morph_sort) (morph_sort rng)
  morph_N    {st} : S₁.signature.N st → S₂.signature.N (morph_sort st)
  morph_prop {st} : S₁.prop st → S₂.prop (morph_sort st)
  morph_nom  {st} : S₁.nom st → S₂.nom (morph_sort st)
  morph_svar {st} : S₁.svar st → S₂.svar (morph_sort st)
```

In particular, we called a *symbols embedding* a morphism such that each of the functions above are injective:

```
structure Symbols.embedding (S₁ : Symbols α) (S₂ : Symbols β) where
  m : S₁.morphism S₂
  sort_inj : m.morph_sort.Injective
  op_inj   : ∀ dom rng, (@m.morph_op dom rng).Injective
  N_inj    : ∀ st, (@m.morph_N st).Injective
  prop_inj : ∀ st, (@m.morph_prop st).Injective
  nom_inj  : ∀ st, (@m.morph_nom st).Injective
  svar_inj : ∀ st, (@m.morph_svar st).Injective
```

A symbols embedding formalizes our notion of "extending the language". In module `Lindenbaum.Expansion.Helpers`, we defined functions that fold a larger structure (e.g., a formula, or a frame, or a model) using a morphism. For brevity we give the signatures of three such functions:

```
def morph_formula (φ : FormL S₁ sorts) (m : S₁.morphism S₂) :
    FormL S₂ (sorts.map m.morph_sort)
def morph_frame (F : Frame S₁.signature) (m : S₁.morphism S₂) :
    Frame S₂.signature
def morph_model (M : Model S₁) (m : S₁.morphism S₂) :
    Model S₂
```

In particular, if `m` is an embedding of S_1 into S_2, then the Lean term `morph_formula φ m` denotes φ *viewed as a formula in the extended language given by* S_2. Similarly for frames and models.

These definitions leave us with three main objectives:

1. Showing that for each set of symbols S_1, there exists an embedding of S_1 into some S_2, such that S_2 contains denumerably many new nominals;

We have a proof of this fact, that used a little bit of constructive creativity. We show the statement below. The `S₁ ↪ S₂` shown is our notation for `S₁.embedding S₂`:

```
instance Symbols.new_nominals (S₁ : Symbols α) :
    Σ S₂ : Symbols (ℕ ⊕ α), Inhabited (S₁ ↪ S₂) :=
```

The trick is simple. If S_1's alphabet is based on type α, we let S_2 be based on type $\mathbb{N} \oplus \alpha$ (the disjoint sum type of $\mathbb{N}$ with α). By construction, $\mathbb{N} \oplus \alpha$ contains *denumerably many more terms than* α. Defining the embedding is then an easy task. We embed all of S_1's symbols into S_2 by `Sum.inr`. The set of nominals NOM of S_2 is the union of S_1's nominals, and `Sum.inl n` for every `n : ℕ`. The actual definition is:

```
nom := λ st => (S₁.nom st.preimage).embed ∪ { Sum.inl n | n : ℕ },
```

Which evidently provides us with denumerably many new nominals.

Remember that in the beginning of Chapter 2.3, we stated one of our slogans as: "Syntax should be modular by definition". This result showcases precisely the sort of modularity we sought to enable.

Let us resume to the other two objectives regarding language extensions.

2. Showing that satisfaction of a formula is preserved by considering the formula and model in an expanded language.

There is less progress on this side. The formal statement of the above is:

```
lemma SatFormLift {m : S₁ ↪ S₂} : (⟨M, g, w⟩ ⊨ ψ) ↔ ⟨m+ M, m+ g, m+ w⟩ ⊨ (m+ ψ)
```

While encouraging that we can formally state this property, more work is needed to prove it. In particular with the definitions of the `morph_frame`, `morph_world` and `morph_model` functions we referenced earlier. These are currently very hard to reason about, as currently their *definitions* rely on non-trivial proofs of heterogenous equality.

Finally,

3. Showing that satisfaction of a formula is preserved by considering the formula and model in an expanded language.

Similarly, *little progress has been made in this direction*, due to the very same reasons as above. The formal statement is:

```
lemma ProvableLift {m : S₁ ↪ S₂} : ⊢(m+ Λ, m+ s) (m+ ψ) ↔ ⊢(Λ, s) ψ
```

4.4.2 Countability of the Language

In order to prove Lindenbaum's Lemma, we additionally need to enumerate all formulas in the extended language. The fact that formulas are denumerable is currently just an unimplemented stub, which lives in module `Lindenbaum.Enumeration.Def`. But we are confident that this is within the realm of possibility. Our previous Lean work for mono-sorted hybrid logic [15] gave a proof of denumerability. We know also that this was done in Lean 3 for propositional modal logic [3]. We therefore assigned low priority to this particular issue.

4.4.3 Lindenbaum's Lemma

Stating and proving Lindenbaum's lemma was another area that required careful design. In the mathematical presentation of Definition 4.3.5, we are building a chain of *sets* and taking its least uppper bound. Yet beyond defining plain sets, it is very helpful to observe a simple invariant of each Lindenbaum iteration: the set defined at each step is moreover *guaranteed to* not *contain infinitely many nominals* in the language. In Lean, we defined a structure that corresponds to sets with this property. We called it `ExtendiblePremiseSet`:

```
structure ExtendiblePremiseSet (S : Symbols α) (s: S.signature.S)
  (Λ : AxiomSet S) where
  set : PremiseSet S s
  unused_nominals : { n : S.nominal t | ¬set.occurs n } ≃ ℕ
```

Therefore, our mechanized Lindenbaum iterations define not just sets, but `ExtendiblePremiseSets`. The definition thus comes with a *proof* that the required invariant is satisfied by the defined set.

Moreover, all our Lindenbaum work is done in the context of a `nominal_extension` variable:

```
    variable (ext : @S.nominal_extension α β β_deq β)
```

This encapsulates a symbols embedding to a signature that has countably many new nominals. We already *constructed* such an embedding earlier, but it is cleaner to work with a general structure that has the required property, instead of a particular construction.

We give the full definition of Lindenbaum sets below.

```
noncomputable def PremiseSet.Lindenbaum (Λ : AxiomSet S) (Γ : PremiseSet S s) :
    ℕ → ExtendiblePremiseSet ext.target (ext.m+ s) (ext.m+ Λ)
| 0        =>
    let Γ' : ExtendiblePremiseSet _ _ (ext.m+ Λ) :=
        ⟨(Γ.embed ext Λ).set ∪ (Γ.embed ext Λ).at_witness_vars, enough_nominals_at⟩
    ⟨Γ'.set ∪ { ℋNom (Γ'.prod_even_nominal _ 0 0) }, enough_nominals_singleton⟩
| .succ i =>
```

```
    let ψ_i : Form ext.target (ext.m+ s) := ofNat _ i
    let Γ' : ExtendiblePremiseSet ext.target (ext.m+ s) (ext.m+ Λ) :=
        ⟨(PremiseSet.Lindenbaum Λ Γ i).set ∪ { ψ_i }, enough_nominals_singleton⟩
    if Γ'.set.consistent (ext.m+ Λ) then
      match ψ_i with
      | ℋ@ j (ℋ∃ x ψ) =>
        ⟨Γ'.set ∪ { ℋ@ j ψ[(Γ'.prod_even_nominal _ i 0) // x]},
            enough_nominals_singleton⟩
      | ℋ@ j (ℋ⟨σ⟩ χ) =>
        ⟨Γ'.set ∪ Γ'.paste_args j σ χ i, enough_nominals_paste⟩
      | _ => Γ'
    else PremiseSet.Lindenbaum Λ Γ i

def PremiseSet.LindenbaumExtension (Λ : AxiomSet S) (Γ : PremiseSet S s) :
    PremiseSet ext.target (ext.m+ s) :=
  { ψ | ∃ i : ℕ, ψ ∈ (Γ.Lindenbaum ext Λ i).set }
```

In module `Completeness.Lindenbaum.Lemma` we have a proof of Lindenbaum's lemma using this definition. Its statement is:

```
lemma Lindenbaum (h : Γ.consistent Λ) :
    ∃ Γ' : NamedPastedWitnessedMCS ext.target
        (ext.m+ s) (ext.m+ Λ), (ext.m+ Γ) ⊆ Γ'.set
```

The only missing pieces of the argument are the invariants in the Lindenbaum definition, as well as proofs that the new nominals are indeed fresh. Furthermore, additional work is needed to formalize certain `MSPHML` proof system properties, such as properties of maximal consistent sets, and the generalization on nominals metatheorem.

4.4.4 Henkin Model. Truth Lemma

The natural design choice for the Henkin model was to use Quotient types. Given an equivalence relation on a type (which in Lean is called a setoid), Lean provides a way to weaken *equality* to *equivalence*, in effect factoring the original type by the equivalence relation. We therefore defined a `Setoid` instance on nominals, given a named, pasted, @-witnessed MCS Γ:

```
abbrev NamedPastedWitnessedMCS.nominal_eq (Γ : NamedPastedWitnessedMCS symbs t Λ)
    (s : symbs.signature.S) (i j : symbs.nominal s) : Prop := ℋ@ i j ∈ Γ.set

instance NamedPastedWitnessedMCS.nominalSetoid (Γ : NamedPastedWitnessedMCS symbs t Λ)
    (s : symbs.signature.S) : Setoid (symbs.nominal s) where
    r := Γ.nominal_eq s
    iseqv := nominal_eq.eqv
```

Using this quotient, the definition of the Henkin model can be cleanly formalized, keeping it close to its mathematical formulation:

```
def NamedPastedWitnessedMCS.HenkinModel (Γ : NamedPastedWitnessedMCS symbs s Λ) :
    Model symbs where
  «Fr» := {
    W  := λ s => Quotient (Γ.nominalSetoid s),
    R  := λ {dom _} σ =>
        match dom with
        | [ ]     => { }
        | _ :: _ => { ⟨q, qs⟩ |
            ∃ j, ∃ (js : WProd symbs.nominal _),
                ℋ@ j (ℋ⟨σ⟩ js.to_args) ∈ Γ.set ∧
                    (⟦j⟧ = q ∧ js.to_quotient Γ = qs) },
    Nm := λ n => ⟦.inl n⟧,
    WNonEmpty := λ s => ⟨⟦(symbs.signature.nNonEmpty s).default⟧⟩
  }
  Vp   := λ {t} p => { q | ∃ j : symbs.nominal t, ℋ@ j (ℋProp p) ∈ Γ.set ∧ ⟦j⟧ = q }
  Vn   := λ j => ⟦.inr j⟧
```

The Truth lemma is currently the most recent piece of work that we started. Not all of the inductive cases are currently completed. Particularly, the applicative case proves to be labour intensive, due once again to our reliance on `Contexts` for picking out formulas in a list. We provide the statement of the Truth lemma below:

```
lemma Truth : (⟨Γ.HenkinModel, Γ.HenkinAsgn, ⟦j⟧⟩ ⊨ φ) ↔ (ℋ@j φ) ∈ Γ.set
```

4.4.5 Completeness

Proof gaps earlier described notwithstanding, we give the proof of completeness in a full below:

```
theorem Completeness {Λ : AxiomSet symbs} : Γ ⊨Mod(Λ) φ → Γ ⊢(Λ) φ := by
  rw [ModelExistence]
  intro h_cons
  rw [←SatLift]
  . have ⟨Γ', incl⟩ := Lindenbaum nominal_extension.default h_cons
    apply SatSubset
    case Γ =>
      exact Γ'.set
    . exists ⟨Γ'.HenkinModel, HenkinInΛ⟩
      exists Γ'.HenkinAsgn
      have ⟨j, j_in_Γ'⟩ := Γ'.named
      exists ⟦j⟧
      intro φ
      rw [Truth]
      apply gen_at_closed Γ'.mcs
      exact j_in_Γ'
    . apply incl
```

As you can see, many of the implementation details are still work-in-progress. What we have definitely achieved, though, was laying a strong foundation that ensures all current unsolved tasks have a definite, narrow scope. Moreover, our prior experience of [15] strongly suggests that some of these tasks are feasible within a short timestamp, as we have already implemented something similar previously (an example would be proving the language countable). For all other unimplemented details, the path forward is discussed in the final chapter.

Chapter 5

Conclusions and Further Work

This thesis highlights several promising directions for extending the present work. The most immediate task is to complete the mechanized proof of completeness outlined in the previous chapter. Since all relevant statements have already been formalized, the remaining effort consists of providing the corresponding proof terms. While this represents a substantial amount of work, we believe that the most challenging conceptual difficulties have already been addressed.

5.1 Further Work: Operational Semantics

Once the completeness of the system is fully verified, the formalization of `MSPHML` opens significant opportunities for applications in the semantics of programming languages. In [12], the system is employed to define the syntax and semantics of the $\langle S, M, C\rangle$ machine introduced by Plotkin [18], and to reason about program executions and invariants. Given the expressive power of Lean, particularly in its metaprogramming facilities, a formalization of `MSPHML` within Lean may provide a modular and practical framework for such applications.

We illustrate this potential with two preliminary examples. To demonstrate feasibility, in the module `SMC.Signature` we manually defined the many-sorted signature of the $\langle S, M, C\rangle$ machine as a genuine *hybrid logic signature*, that is, a `Symbols String` object. Furthermore, we introduced notations for formulas over this signature that preserve the intuitive style of structural operational semantics while abstracting away the syntactic complexity intrinsic to MSPHML. As a simple illustration, consider the following structural rule for compound statements:

```
def CStmtAx (s1 s2 : SMCForm SortStmt)
  : SMCForm SortCtrlStack := c(s1 ; s2) ↔ c(s1) ; c(s2)
```

This expression defines an actual many-sorted hybrid formula, hidden under multiple layers of syntactic sugar. As shown in [12], such formulas amount to pure extensions of the underlying proof system, thus automatically gaining soundness and completeness guarantees when used to reason about program properties. Demonstrating that program verification in Lean via MSPHML can be both practical and lightweight, thanks to such syntactic sugar, would represent a significant step forward.

The applicability of this approach is by no means limited to the $\langle S, M, C\rangle$ machine, and neither should one be forced to define the signature manually as we have done. Using Lean's metaprogramming capabilities, it is possible to design a domain-specific language (DSL) for specifying arbitrary BNF-style grammars, which could then be *automatically* elaborated into MSPHML signatures. Initial experiments in this direction were carried out in the module DSL.BNF. Consider the following sketch of defining the $\langle S, M, C\rangle$ machine's signature in our proof-of-concept DSL:

~~~
```hybrid_def SMC
    sort Nat  ::= builtin Nat
    sort Bool ::= builtin Bool | "_==_"(Nat, Nat) | "_<=_"(Nat, Nat)

    sort Var  ::= builtin String
    sort AExp ::= subsort Nat | subsort Var
    sort AEXp ::= "_+_"(AExp, AExp) | "++"(Var)
    sort BExp ::= "_<=_"(AExp, AExp)
    sort Stmt ::= skip
                | "_:=_"(Var, AExp)
                | "if_then_else_"(BExp, Stmt, Stmt)
                | "while_do_"(BExp, Stmt)
                | "_;_"(Stmt, Stmt)
~~~

~~~~
    sort Val ::= subsort Nat | subsort Bool
    sort ValStack ::= nil
                | "_._"(Val, ValStack)
    sort Mem ::= empty | "set"(Mem, Var, Nat)
    sort CtrlStack ::= "c"(AExp)
                | "c"(BExp)
                | "c"(Stmt)
                | "asgn"(Var)
                | plus | leq
                | "_?"(Val)
                | "_;_"(CtrlStack, CtrlStack)
    sort Config ::= "<_,_>"(ValStack, Mem)
```
~~~~

Although this code is currently valid Lean syntax, it is currently *meaningless*: no elaboration mechanism has yet been implemented to translate it into a `Symbols String` object (i.e., a hybrid logic signature). Developing such a compilation pipeline is an engineering direction we have not yet tackled, and will likely require significant implementation effort in its own self.

Furthermore, there is also the immediate promise of studying computability questions of MSPHML and its fragments, following the extensive literature that exists on this topic in regular hybrid settings. Having the system formalized in Lean allows decidability results to be tied to executable artifacts, that can be used in tactics to automatically close proof goals.

5.2 Conclusions

We conclude by briefly noting our main achievements and findings. We were successful in formalizing a very general kind of hybrid modal logic, involving polyadic modal operators and many-sorted signatures. This proves once more that proof assistants, and Lean in particular, have become mature enough to aid the verification of results in published research papers, with [12] serving as our blueprint for this enterprise.

Naturally, the more complex the formalism, the more prone to mistakes the human researcher is. Our work clarified and expanded upon the original formalization, which helped in identifying and resolving certain inconsistencies. Specifically, the Barcan axiom, along with the Name@ and Paste rules, were found to be unsound unless certain restrictions were additionally imposed.

The introduction of computer assistance in mathematical proof construction is widely recognized to have begun with Appel and Haken's solution to the four-color problem [1]. However, as Tymoczko [23] notably observed, their proof exemplified what he termed a "non-surveyable proof": one whose entirety cannot be comprehended by any individual human mind, thus challenging traditional notions of mathematical proof and knowledge.

This observation merits serious consideration. Yet we propose that a complementary perspective deserves equal attention: when the theory under study becomes so complex that even the most attentive researcher cannot simultaneously maintain awareness of all relevant details and their interactions, how else could they gain confidence in the validity of their work if not by means of computer assistance? In such circumstances, formal verification may represent not merely a convenience, but a methodological necessity for advancing knowledge with appropriate rigor.

Bibliography

[1] Kenneth I. Appel and Wolfgang Haken. *Every Planar Map is Four Colorable*. American Mathematical Soc., 1989. 760 pp.

[2] Jeremy Avigad, Leonardo de Moura, Kong Soonho, and Ulrich Sebastian. “Quantifiers and Equality”. In: *Theorem Proving in Lean 4*. URL: https://leanprover.github.io/theorem_proving_in_lean4/ (visited on 08/27/2025).

[3] Bruno Bentzen. “A Henkin-Style Completeness Proof for the Modal Logic S5”. In: *Logic and Argumentation*. Ed. by Pietro Baroni, Christoph Benzmüller, and Y N. Wáng. Cham: Springer International Publishing, 2021, pp. 459–467. DOI: 10.1007/978-3-030-89391-0_25.

[4] Péter Bereczky, Xiaohong Chen, Dániel Horpácsi, Lucas Peña, and Jan Tušil. *Mechanizing Matching Logic In Coq*. Sept. 19, 2022. DOI: 10.4204/EPTCS.369.2. arXiv: 2201.05716[cs]. URL: http://arxiv.org/abs/2201.05716 (visited on 09/02/2025).

[5] P Blackburn and M Tzakova. “Hybrid completeness”. In: *Logic Journal of the IGPL* 6.4 (July 1, 1998), pp. 625–650. DOI: 10.1093/jigpal/6.4.625. (Visited on 09/01/2025).

[6] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*. Cambridge Tracts in Theoretical Computer Science. Cambridge: Cambridge University Press, 2001. DOI: 10.1017/CBO9781107050884. (Visited on 08/25/2025).

[7] Xiaohong Chen and Grigore Roşu. “Matching mu-logic”. In: *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS ’19. Vancouver, Canada: IEEE Press, June 8, 2021, pp. 1–13. (Visited on 08/27/2025).

[8] Horațiu Cheval and Bogdan Macovei. *Matching Logic in Lean: Techincal Report*. Oct. 4, 2022. URL: https://gitlab.com/ilds/aml-lean/MatchingLogic (visited on 09/02/2025).

[9] Joseph A. Goguen. *Theorem Proving and Algebra*. Jan. 16, 2021. DOI: 10.48550/arXiv.2101.02690. URL: http://arxiv.org/abs/2101.02690 (visited on 08/25/2025).

[10] Jesse Michael Han and Floris van Doorn. “A formal proof of the independence of the continuum hypothesis”. In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2020. New York, NY, USA: Association for Computing Machinery, Jan. 22, 2020, pp. 353–366. DOI: 10.1145/3372885.3373826. (Visited on 08/29/2025).

[11] Ioana Leuştean, Natalia Moangă, and Traian Florin Șerbănuță. *From Hybrid Modal Logic to Matching Logic and Back*. Sept. 2, 2019. DOI: 10.4204/EPTCS.303.2. arXiv: 1907.05029[cs]. URL: http://arxiv.org/abs/1907.05029 (visited on 09/02/2025).

[12] Ioana Leuștean, Natalia Moangă, and Traian Florin Șerbănuță. “Operational Semantics and Program Verification Using Many-Sorted Hybrid Modal Logic”. In: *Automated Reasoning with Analytic Tableaux and Related Methods*. Ed. by Serenella Cerrito and Andrei Popescu. Cham: Springer International Publishing, 2019, pp. 446–476. DOI: 10.1007/978-3-030-29026-9_25.

[13] *NethermindEth/EVMYulLean*. Aug. 21, 2025. URL: https://github.com/NethermindEth/EVMYulLean (visited on 09/02/2025).

[14] Tobias Nipkow and Gerwin Klein. *Concrete Semantics*. Cham: Springer International Publishing, 2014. DOI: 10.1007/978-3-319-10542-0. (Visited on 09/02/2025).

[15] Andrei-Alexandru Oltean. “A Formalization of Hybrid Logic in Lean”. BSc thesis. University of Bucharest, 2023.

[16] *opencompl/sail-riscv-lean*. URL: https://github.com/opencompl/sail-riscv-lean (visited on 09/03/2025).

[17] Benjamin Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hrițcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. *Programming Language Foundations*. URL: https://softwarefoundations.cis.upenn.edu/plf-current/index.html (visited on 09/02/2025).

[18] Gordon D. Plotkin. *A Structural Approach to Operational Semantics*. Aarhus University, 1981. (Visited on 09/02/2025).

[19] *risc0/risc0-lean4: A model of the RISC Zero zkVM and ecosystem in the Lean 4 Theorem Prover*. URL: https://github.com/risc0/risc0-lean4 (visited on 09/02/2025).

[20] Grigore Roșu and Traian Florin Șerbănuță. “An overview of the K semantic framework”. In: *The Journal of Logic and Algebraic Programming*. Membrane computing and programming 79.6 (Aug. 1, 2010), pp. 397–434. DOI: 10.1016/j.jlap.2010.03.012. (Visited on 09/02/2025).

[21] Donald Sannella and Andrzej Tarlecki. “Universal algebra”. In: *Foundations of Algebraic Specification and Formal Software Development*. Ed. by Donald Sannella and Andrzej Tarlecki. Berlin, Heidelberg: Springer, 2012, pp. 15–39. DOI: 10.1007/978-3-642-17336-3_1. (Visited on 08/25/2025).

[22] *The Lean Language Reference: Lean 4.22.0 (2025-08-14)*. URL: https://lean-lang.org/doc/reference/latest/releases/v4.22.0/ (visited on 08/26/2025).

[23] Thomas Tymoczko. “The Four-Color Problem and Its Philosophical Significance”. In: *The Journal of Philosophy* 76.2 (1979), pp. 57–83. DOI: 10.2307/2025976. (Visited on 09/02/2025).

[24] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction.* Foundations of Computing. Cambridge, MA, USA: MIT Press, Feb. 5, 1993. 384 pp.

Appendix A

Technical Appendix

Reamining cases for Soundness (Theorem 4.1.1).

(Λ): By definition, if $\phi \in \Lambda$, then $\vDash^s_{Fr(\Lambda)} \varphi$. (Prop): All propositional tautologies are clearly true in any model $\mathcal{M}$, with any assignment g and at any world w.

(K): Assume $\mathcal{M}, g, w \vDash^s \sigma^{\Box}(\varphi_1, \dots, (\varphi \to \psi)_i, \dots, \varphi_n)$ and $\mathcal{M}, g, w \vDash^s \sigma^{\Box}(\varphi_1, \dots, \varphi_i, \dots, \varphi_n)$. Let $w_1, \dots w_n \in W_{s_1} \times \dots \times W_{s_n}$ such that $(w, w_1, \dots, w_n) \in R_\sigma$.

By assumption, we know there exist indices j and k such that: (a) $\mathcal{M}, g, w_j$ satisfies the j-th formula in $\varphi_1, \dots, (\varphi \to \psi)_i, \dots, \varphi_n$; and (b) $\mathcal{M}, g, w_k$ satisfies the k-th formula in $\varphi_1, \dots, \varphi_i, \dots, \varphi_n$. If $j = k = i$, then $\mathcal{M}, g, w_j \vDash^{s_i} \varphi \to \psi$ and $\mathcal{M}, g, w_j \vDash^{s_i} \varphi$, so $\mathcal{M}, g, w_j \vDash^{s_i} \psi$, QED. Otherwise, we know φ_j occurs in $\varphi_1, \dots, (\varphi \to \psi)_i, \dots, \varphi_n$, and $\mathcal{M}, g, w_j \vDash^{s_j} \varphi_j$, QED.

(K@): Assume $\mathcal{M}, g, w \vDash^s @^s_j(\varphi \to \psi)$, iff $\mathcal{M}, g, V(j) \vDash^t \varphi \to \psi$. Assume also $\mathcal{M}, g, w \vDash^s @^s_j\varphi$, iff $\mathcal{M}, g, V(j) \vDash^t \varphi$. Thus $\mathcal{M}, g, V(j) \vDash^t \psi$, iff $\mathcal{M}, g, w \vDash^s @^s_j\varphi$.

(Agree): $\mathcal{M}, g, w \vDash^s @^s_k @^{s'}_j \varphi$ iff $\mathcal{M}, g, V(k) \vDash^{s'} @^{s'}_j \varphi$ iff $\mathcal{M}, g, V(j) \vDash^t \varphi$ iff $\mathcal{M}, g, w \vDash^t @^s_j\varphi$.

(SelfDual): $\mathcal{M}, g, w \vDash^s @^s_j\varphi$ iff $\mathcal{M}, g, V(j) \vDash^t \varphi$ iff $\mathcal{M}, g, V(j) \nvDash^t \neg\varphi$ iff $\mathcal{M}, g, w \nvDash^s @^s_j\neg\varphi$ iff $\mathcal{M}, g, w \vDash^s \neg @^s_j \neg\varphi$.

(Intro): Assume $\mathcal{M}, g, w \vDash^s j$, so $V(j) = w$. Then clearly $\mathcal{M}, g, w \vDash^s \varphi$ iff $\mathcal{M}, g, w \vDash^s @_j\varphi$.

(Back): Assume $\mathcal{M}, g, w \vDash^s \sigma(\dots, @^{s_i}_j\psi, \dots)$. Thus, there exist $w_1, \dots, w_n \in W_{s_1} \times \dots \times W_{s_n}$ so that $\mathcal{M}, g, w_j$ satisfies the l-th argument to σ, for all $1 \leq l \leq n$. In particular, $\mathcal{M}, g, w_i \vDash^s @^{s_i}_j\psi$, which is equivalent to $\mathcal{M}, g, V(j) \vDash^s \psi$. So $\mathcal{M}, g, w_i \vDash^s @^{s_i}_j\psi$.

(Ref): Trivial: we have $\mathcal{M}, g, w \vDash^s @_j j$ iff $\mathcal{M}, g, V(j) \vDash^s j$, which is simply the definition of the satisfaction relation.

(Q1): Assume $\mathcal{M}, g, w \vDash^s \forall x(\varphi \to \psi)$ and $\mathcal{M}, g, w \vDash^s \varphi$, for x not occurring freely in φ. Let $g' \rightsquigarrow^x g$. By the first assumption we have $\mathcal{M}, g', w \vDash^s \varphi \to \psi$. Since x does not occur free in φ, and g, g' otherwise agree, we apply the Agreement Lemma (4.1.1) to the second assumption, obtaining $\mathcal{M}, g', w \vDash^s \varphi$. Immediately, then, $\mathcal{M}, g', w \vDash^s \psi$, QED.

(Q2): Assume $\mathcal{M}, g, w \vDash^s \forall x\varphi$, and let y be substitutable for x in φ. Let g' be the x-variant of g such that $g'(x) = g(y)$. Then we have $\mathcal{M}, g', w \vDash^s \varphi$, which, by Substitution Lemma (4.1.3), is equivalent to $\mathcal{M}, g, w \vDash^s \varphi[y/x]$, QED. For the case of substituting with a nominal, we pick $g' = V(j)$.v

(Name): Let $\mathcal{M}, g, w$, and take g' as the x-variant of g such that $g'(x) = w$. Then $\mathcal{M}, g', w \vDash^s x$, so we have $\mathcal{M}, g, w \vDash^s \exists x x$.

(Barcan@): Assume $\mathcal{M}, g, w \vDash^s \forall x @_j \varphi$, iff for all $g' \rightsquigarrow^x g$, $\mathcal{M}, g', w \vDash^s @_j \varphi$, iff for all $g' \rightsquigarrow^x g$, $\mathcal{M}, g', V(j) \vDash^s \varphi$, iff $\mathcal{M}, g', V(j) \vDash^s \forall x\varphi$, iff $\mathcal{M}, g, w \vDash^s @_j \forall x\varphi$.

(Nom): Assume $\mathcal{M}, g, w \vDash^s @_k x$ and $\mathcal{M}, g, w \vDash^s @_j x$. Thus, $g(x) = V(j) = V(k)$. Therefore $\mathcal{M}, g, w \vDash^s @_k j$.

(MP): Trivial by basic reasoning.

(UG): Assume $\vDash^{s_i}_{Fr(\Lambda)} \varphi_i$. We want to show $\vDash^{s}_{Fr(\Lambda)} \sigma^{\Box}(\varphi_1, \dots, \varphi_i, \dots, \varphi_n)$. Therefore, let $\mathcal{M} \in Fr(\Lambda), g, w$, and let $w_1, \dots, w_n \in W_{s_1} \times \dots \times W_{s_n}$ be such that $(w, w_1, \dots, w_n) \in R_\sigma$. By assumption we have $\mathcal{M}, g, w_i \vDash^{s_i} \varphi_i$, QED.

(BroadcastS): Assume $\vDash^{s_i}_{Fr(\Lambda)} @^s_j \varphi$. We want to show $\vDash^{s}_{Fr(\Lambda)} @^{s'}_j \varphi$. Let $\mathcal{M} \in Fr(\Lambda), g$, and let $w' \in W_{s'}$. We have $\mathcal{M}, g, w' \vDash^{s'} @^{s'}_j \varphi$ iff $\mathcal{M}, g, V(j) \vDash^t \varphi$.

By definition, W_s is non-empty, so let $w \in W_s$. By assumption, then, $\mathcal{M}, g, w \vDash^s @^s_j \varphi$, equivalent to $\mathcal{M}, g, V(j) \vDash^t \varphi$, QED.

(Gen@): Assume $\vDash^{s}_{Fr(\Lambda)} \varphi$. We want to show $\mathcal{M}, g, V(j) \vDash^t \varphi$, for any $\mathcal{M} \in Fr(\Lambda), g$. This follows immediately by assumption.

(Gen): Assume $\vDash^{s}_{Fr(\Lambda)} \varphi$. We want to show $\mathcal{M}, g', w \vDash^s \varphi$, for any $\mathcal{M} \in Fr(\Lambda), g, w$ and $g' \rightsquigarrow^x g$. This follows immediately by assumption. □