Logic as a Second-Order Generalized Algebraic Theory

Notes on my 3-month internship at the Faculty of Informatics of ELTE (Budapest, Hungary)

Samy Avrillon, supervised by

Ambrus Kaposi (ELTE, Budapest, Hungary)

and Thorsten Altenkirsch (University of Notthingam, United Kingdom)

August 28, 2023

Contents

1 Introduction **2**
- 1.1 Introduction to the problem . . . 2
- 1.2 Motivation . . . 2
- 1.3 Structure of this report . . . 2

2 Propositional Logic **2**
- 2.1 Propositional Logic as a SOGAT . . . 2
- 2.2 From a SOGAT to GAT . . . 3
- 2.3 Finding a strict syntax . . . 5
- 2.4 Stating and Proving Completeness . . . 8

3 First-order logic (or Predicate Logic) **9**
- 3.1 Infinitary First-Order Logic . . . 9
- 3.2 (Finitary) Predicate Logic as a SOGAT . . . 10
  - 3.2.1 Making the SOGAT . . . 10
  - 3.2.2 Making a GAT out of it . . . 10
- 3.3 Finding a Syntax for First-Order Logic . . . 12
- 3.4 Transport Hell . . . 18

4 Summary **18**

5 Bibliography **19**

Appendices **20**

1 Introduction

In the first place, I was supposed to do this internship in Nottingham (UK) with Thorsten Altenkirsh. But, because of administrative issues, I was not able to go there, and Thorsten Altenkirsh contacted Ambrus Kaposi in Budapest, whose university agreed to accept me. Therefore, I went to Budapest and I did the internship under the physical supervision of Ambrus Kaposi, and the remote supervision of Thorsten Altenkirsch.

1.1 Introduction to the problem

What we call a "logic" or "logical framework" is a set of definitions containing formulas and a notion of provability of those formulæ, plus a set of operators/equalities to construct or reduce these proofs [**logicalFramework1993**]. I have studied the most common logical frameworks, that is, Propositional Logic, First-Order Logic with an infinitary operator, and Predicate Logic. For each of those logics, one can define a notion of *model*. A model of a certain kind of logic is something that implements all of the logic's definitions, operators, and equalities. From all of those models, one can extract the *initial model*, also called *syntax*. It is the smallest of all models, which means that for any model of that logic, we have a morphism from the syntax to this model.

Then, our goal is for each logic to prove the completeness of a specific class of models. Completeness can be stated as such: "For any formula, if it is provable in all models of the specified class, then it has a proof in the syntax".

1.2 Motivation

Ambrus is currently studying Second-Order Generalized Algebraic Theories (or SOGAT) and he is trying to state a better definition than that first defined in Taichi Uemura's thesis [**UemuraThesis2021**]. He also wants to write a paper with examples to show why they are useful, and adding some examples for different frameworks of logic can help with that. Having a completeness proof in this (categorical) higher order setting [**hoffmann1999**] can help.

1.3 Structure of this report

In this report, we will gradually increase the complexity of the logic we are studying. We will first study the most simple Propositional logic, which will serve as an explanation of the different concepts used in the report, and then we will study Predicate Logic, first in a layout using a trick to avoid most of the difficulties, and then in a more satisfying layout but which is harder to make.

For each logic, we will first give the SOGAT definition of the logic. We will then transform it into a GAT (i.e. something that can be understood by Agda). Then we will construct the syntax of the logic. Finally, we will construct the completeness proof for the studied logic.

Unfortunately, the proof of the initiality of Predicate Logic's syntax, as well as its completeness proof, are not given in this report as I did not have the time to make them at the end of the internship.

2 Propositional Logic

2.1 Propositional Logic as a SOGAT

The first and most simple logical framework we can work with is that of Propositional Logic (referred to as Zero Order Logic or ZOL in the code). We also work in the most simple framework in which we only have one axiom rule for creating formulæ: the ι rule. Propositional Logic is

```
For        :  Set
— ⟹ —      :  For → For → For
ι          :  For

Pf         :  For → Prop⁺
lam        :  (Pf A  →⁺ Pf B)  → Pf (A  ⟹ B)
app        :  Pf (A  ⟹ B)  → (Pf A  → Pf B)
```

Figure 1: ZOL Sogat Presentation

usually done with a fixed set of propositional variables instead of only one that is fixed, but adding them does not make the construction more interesting, only more complicated.

To state all the definitions, functions, and more importantly, all the equalities that a model of Propositional Logic has to verify, we will write down Propositional Logic as a SOGAT. It goes as described in Figure 1.

We can see that in this Sogat, we have two sorts (For and Pf), each of them having two constructors. Although SOGATs can have equations in them, every SOGAT in this report will not have any.

A keen eye may have seen that there should be a problem with the lam constructor. Its type is indeed not strictly positive (i.e. the type Pf appears to the left of an arrow in the arguments of the constructor). That's why we use a $^+$ on the arrow, and we use the same $^+$ on the sort of Pf. This kind of arrow means that Pf should be *locally representable*. It means that we have to use *proof variables* to implement this SOGAT.

2.2 From a SOGAT to GAT

Unfortunately, Agda cannot understand SOGATs directly (yet). So we have to convert this SOGAT into a (First Order) Generalized Algebraic Theory (or GAT). I will describe the process in this section, each line of the SOGAT giving birth to a set of sorts, constructors, and equations in the GAT.

The first thing we need to define any SOGAT-derived GAT is a base category with a terminal object. We will call *contexts* the objects of this category and *substitutions* the morphisms of the category.

You can see that the equations for the category (∘-associativity, identity to the right and the left) have not been written down. This is because substitutions are defined as **Prop** objects and not **Set** objects. The difference between those two kinds of objects is that the former are said to be *proof-irrelevant*. You can understand it as "there is at most one object into a type that is in **Prop**". Therefore, the equalities described above are trivially true, because any substitution from Γ to Δ is equal to any other substitution from Γ to Δ. This trick of using **Prop** instead of **Set** will be used throughout the report, and it will allow us to write down (and prove) a lot fewer equations.

```
--# We first make the base category with its terminal object
Con : Set ℓ¹
Sub : Con → Con → Prop ℓ² -- It makes a posetal category
_∘_ : {Γ Δ Ξ : Con} → Sub Δ Ξ → Sub Γ Δ → Sub Γ Ξ
id : {Γ : Con} → Sub Γ Γ
◇ : Con -- The terminal object of the category
ε : {Γ : Con} → Sub Γ ◇ -- The morphism from any object to the terminal
```

Once we have our base category, we can start to describe the sorts that we have in our SOGAT. Every sort from the SOGAT will be translated into a functor from our base category to **Set** (or **Prop**, depending on the case). Therefore, we have to define an action on objects, an action on morphisms (that we will always write contra-variant, and we will use the _[_] notation), and some equalities that we will call *functorality* or "*Type* respects substitution"

```
--# Functor Con → Set called For
For : Con → Set ℓ³
_[_]f : {Γ Δ : Con} → For Γ → Sub Δ Γ → For Δ -- Action on morphisms
[]f-id : {Γ : Con} → {F : For Γ} → F [ id {Γ} ]f ≡ F
[]f-∘ : {Γ Δ Ξ : Con} → {α : Sub Ξ Δ} → {β : Sub Δ Γ} → {F : For Γ}
  → F [ β ∘ α ]f ≡ (F [ β ]f) [ α ]f
```

```
--# Functor Con × For → Prop called Pf or ⊢
Pf : (Γ : Con) → For Γ → Prop ℓ
-- Action on morphisms
_[_]p : {Γ Δ : Con} → {F : For Γ} → Pf Γ F → (σ : Sub Δ Γ) → Pf Δ (F [ σ ]f)
```

Now, we will translate the $^+$ in the type of proofs. As we said before, this translates into the fact that we have *proof variables*. This means that we should be able to move from one context to another context in which we added one proof variable. We call this *context extension*. For proof-variables, this context extension takes one parameter, as each proof-variables is of one specific "type" (which is the formula that they prove).

With those context extensions, we also have to add a way to extend substitutions (which are the morphisms from the base category). We do that by adding a *comma* operator that takes a substitution and a proof and which extends the goal of the substitution. (If we see a proof context as a list of formulæ and a proof substitution as a list of proofs, the action of the comma operator is simply to add one proof to the list). We also need to be able to extract the proof and the base substitution from a substitution into an extended context (called *weakening* operation). Therefore, we add two π projections that extract the data from the substitution. We also need some equalities, that are saying that $(\pi_p{}^1, \pi_p{}^2)$, and $,_p$ are reciprocal, plus one equality saying that the comma operator also transposes the substitution. But as both Pf and Sub are in **Prop**, those equations are unneeded.

```
--# → Prop⁺
_▹p_ : (Γ : Con) → For Γ → Con
πp¹ : {Γ Δ : Con}{F : For Γ} → Sub Δ (Γ ▹p F) → Sub Δ Γ
πp² : {Γ Δ : Con}{F : For Γ} → (σ : Sub Δ (Γ ▹p F)) → Pf Δ (F [ πp¹ σ ]f)
_,p_ : {Γ Δ : Con}{F : For Γ} → (σ : Sub Δ Γ) → Pf Δ (F [ σ ]f) → Sub Δ (Γ ▹p F)
```

The last step is to add the constructors of our objects into the algebra. We write them directly as they were presented in the SOGAT, except for the new Con parameter that we need to add to everything. We also need to add one equation for each constructor saying that the constructors do respect substitutions. And we don't add any for Pf constructors as they are unneeded as Pf are in **Prop**.

The only one we cannot translate directly is the $\to^+$ from the lam constructor from the SOGAT. To translate this *locally representable application*, we transform the parameter of the arrow into a context extension of the appropriate type. Therefore, a function Pf $A \to^+$ Pf B becomes a proof Pf $(\Gamma \triangleright_p A)$ B (except that we have to transform B that is a For Γ into a For $(\Gamma \triangleright_p A)$, using the *weakening* substitution $\pi_p{}^1$ id : Sub Γ $(\Gamma \triangleright_p X)$ for any Γ-formula X)

```
--# ι formula
ι : {Γ : Con} → For Γ
[]f-ι : {Γ Δ : Con} {σ : Sub Δ Γ}→ ι [ σ ]f ≡ ι

--# Implication
_⇒_ : {Γ : Con} → For Γ → For Γ → For Γ
[]f-⇒ : {Γ Δ : Con} → {F G : For Γ} → {σ : Sub Δ Γ}
  → (F ⇒ G) [ σ ]f ≡ (F [ σ ]f) ⇒ (G [ σ ]f)

--# Lam & App
lam : {Γ : Con}{F G : For Γ} → Pf (Γ ▹ₚ F) (G [ πₚ¹ id ]f) → Pf Γ (F ⇒ G)
app : {Γ : Con}{F G : For Γ} → Pf Γ (F ⇒ G) → Pf Γ F → Pf Γ G
```

We now have completed our definition of a model for Propositional Logic, in the SOGAT setting. In the meantime, we will define (categorical) morphisms between models. They are composed of mappings for every sort of the GAT, plus equations for the categorical objects (terminal object, identity, and composition), for the action of the functors on morphisms ($[]f$ and $[]p$), and finally equations to say that the constructors are also transported by the morphism. A lot of those equations are not written because Pf and Sub are in **Prop**.

```
--#
mCon : (ZOL.Con S) → (ZOL.Con D)
mSub : {Δ : (ZOL.Con S)}{Γ : (ZOL.Con S)} →
  (ZOL.Sub S Δ Γ) → (ZOL.Sub D (mCon Δ) (mCon Γ))
mFor : {Γ : (ZOL.Con S)} → (ZOL.For S Γ) → (ZOL.For D (mCon Γ))
mPf : {Γ : (ZOL.Con S)} {A : ZOL.For S Γ}
  → ZOL.Pf S Γ A → ZOL.Pf D (mCon Γ) (mFor A)

--#
e◇ : mCon (ZOL.◇ S) ≡ ZOL.◇ D
e[]f : {Γ Δ : ZOL.Con S}{A : ZOL.For S Γ}{σ : ZOL.Sub S Δ Γ} →
  mFor (ZOL._[_]f S A σ) ≡ ZOL._[_]f D (mFor A) (mSub σ)
e▹ₚ : {Γ : ZOL.Con S}{A : ZOL.For S Γ} →
  mCon (ZOL._▹ₚ_ S Γ A) ≡ ZOL._▹ₚ_ D (mCon Γ) (mFor A)
eι : {Γ : ZOL.Con S} → mFor (ZOL.ι S {Γ}) ≡ ZOL.ι D {mCon Γ}
e⇒ : {Γ : ZOL.Con S}{A B : ZOL.For S Γ} →
  mFor (ZOL._⇒_ S A B) ≡ ZOL._⇒_ D (mFor A) (mFor B)
-- No equation needed for lam, app, ∀i, ∀e as their output are in prop
```

2.3 Finding a strict syntax

What will be the use of these morphisms? One thing we want to do is to find the *initial* model, or *syntax* for this class of models. It means that we want to find a model from which, for any model M, there exists one and only one morphism from this model to M. We can also call the syntax the *minimal model*, as it is the model that contains just the objects something needs to be a model of Propositional Logic.

Mathematically, this model is really easy to find. We only have to take all the sorts defined in the GAT and apply a quotient by every equation of the GAT. But this is not very interesting, as the defined model would not be *strict*, i.e. all equations will not be definitional. Having only strict definitions for objects allows us to study completeness while removing a lot of transport issues that make the transport hell far bigger in the completeness proof. Transport hell will be explained in subsection 3.4.

So we have three things to do :

1. Create a model I for Propositional Logic that we think is minimal
2. For any other model M, create a morphism $m_I : I \to M$
3. Show that any morphism $I \to M$ is equal to m_I

Let's get started. This framework of Propositional logic has one property that we can make use of to make the construction of the syntax much simpler. (Actually, we don't know if we could have made the syntax with this method if we haven't had this property. We will try to investigate it further in the future). This property is that formulæ do not depend on contexts. One can understand it as "a formula is the same regardless of the proof variables associated with it, as it doesn't use any of them". Without this property, we would have to define mutually every sort of our theory, which would be extremely complicated.

The first step is to define our sorts, no difficulty, we only make one datatype constructor for each constructor in the SOGAT (or in the GAT). For contexts, the only two constructors are as the terminal object of the base category, or as a context extension of another context. That's where we use the fact that formulæ don't depend on contexts: it allows us to define For and Con separately, but also to define Pf and Sub separately (the lam constructor should refer to _[_]f, but formulæ substitution is identity when formulæ do not depend on contexts). There is also another special constructor for proofs, which is that of proof variables, and which corresponds to the "constructor" which is π_p^2

```
--#
data For : Set where
  ι : For
  _⇒_ : For → For → For

data Con : Set where
  ◇ : Con
  _▷p_ : Con → For → Con
```

```
data PfVar : Con → For → Prop where
  pvzero : PfVar (Γ ▷p A) A
  pvnext : PfVar Γ A → PfVar (Γ ▷p B) A
data Pf : Con → For → Prop where
  var : PfVar Γ A → Pf Γ A
  lam : Pf (Γ ▷p A) B → Pf Γ (A ⇒ B)
  app : Pf Γ (A ⇒ B) → Pf Γ A → Pf Γ B
```

The next step is to define substitutions. But we will first define a weaker property than substitution called "renamings". The two definitions are put one next to the other, and we can see that when Sub are lists of complete proofs, Ren are only lists of proof variables (i.e. direct proofs assumed from the contexts). This weaker form of substitution is needed to define the identity substitution, as we cannot directly define it recursively for fully-featured substitutions.

The projections are simply a pattern-matching against the datatype.

```
data Ren : Con → Con → Prop where
  ε : Ren Γ ◇
  _,pR_ : Ren Δ Γ → PfVar Δ A → Ren Δ (Γ ▷p A)
```

```
data Sub : Con → Con → Prop where
  ε : Sub Γ ◇
  _,p_ : Sub Δ Γ → Pf Δ A → Sub Δ (Γ ▷p A)

πp¹ : Sub Δ (Γ ▷p A) → Sub Δ Γ
πp² : Sub Δ (Γ ▷p A) → Pf Δ A
πp¹ (σ ,p pf) = σ
πp² (σ ,p pf) = pf
```

Because we need to refer to id, and more precisely to π_p^1 id in the notion of proof substitution, we first define proof *renamings* so we can define wkSub (which is the same as π_p^1 id) and only after that, we define fully featured substitution.

```
_[_]pvr : PfVar Γ A → Ren Δ Γ → PfVar Δ A
pvzero [ _ ,pR pv ]pvr = pv
pvnext pv [ σ ,pR _ ]pvr = pv [ σ ]pvr
_[_]pr : Pf Γ A → Ren Δ Γ → Pf Δ A
var pv [ σ ]pr = var (pv [ σ ]pvr)
lam pf [ σ ]pr = lam (pf [ (rightR σ) ,pR pvzero ]pr)
app pf pf' [ σ ]pr = app (pf [ σ ]pr) (pf' [ σ ]pr)

wkSub : Sub Δ Γ → Sub (Δ ▷p A) Γ
wkSub ε = ε
wkSub (σ ,p pf) = (wkSub σ) ,p (pf [ rightR idR ]pr)
```

```
_[_]p : Pf Γ A → Sub Δ Γ → Pf Δ A
var pvzero [ _ ,p pf ]p = pf
var (pvnext pv) [ σ ,p _ ]p = var pv [ σ ]p
lam pf [ σ ]p = lam (pf [ wkSub σ ,p var pvzero ]p)
app pf pf' [ σ ]p = app (pf [ σ ]p) (pf' [ σ ]p)
```

The last step is then to define substitution composition, which is done by applying the second part of the substitution to every element of the first part.

```
_∘_ : {Γ Δ Ξ : Con} → Sub Δ Ξ → Sub Γ Δ → Sub Γ Ξ
ε ∘ β = ε
(α ,p pf) ∘ β = (α ∘ β) ,p (pf [ β ]p)
```

And we now have a working model for Propositional Logic. We can implement our previously defined record and all the equations can be directly understood by Agda (refl is enough proof).

I won't dive into the construction of the initial morphism and the proof that it is unique, since it is a bit ugly because of some *transport hell* (see subsection 3.4 for an explanation of the issue).

Still, below is the construction of the Con and For mappings, as they show the basic idea of the construction. The idea is simple, we pattern-match against the datatype definition of our sort in our syntax, and for each constructor, we call the corresponding one in the algebra.

And the uniqueness proof is pretty simple again because you only need to prove that the Con and For mappings are unique, because the other ones are in **Prop**, so they are automatically unique. And to prove the unicity, you split again on the sort's datatype, and for each constructor, you use the fact that a morphism preserves constructors, and you have a nice proof by recursion (except it's ugly because of transport hell).

```
--#
mCon : Con → (ZOL.Con M)
mFor : {Γ : Con} → For → (ZOL.For M (mCon Γ))

mCon ◇ = ZOL.◇ M
mCon (Γ ▷ₚ A) = ZOL._▷ₚ_ M (mCon Γ) (mFor {Γ} A)
mFor {Γ} ι = ZOL.ι M
mFor {Γ} (A ⇒ B) = ZOL._⇒_ M (mFor {Γ} A) (mFor {Γ} B)
```

2.4 Stating and Proving Completeness

Now that we have our syntax, we can try to state and prove completeness in our specific categorical framework.

The general idea of completeness proofs is to take a class of models (here we will take Kripke Models) and to state that for any formlæ that has a proof in every Kripke model, then it has a proof in the syntax. The other direction is called *soundness*, but we already have proven that when we created our initial morphism. This morphism can indeed turn proofs in the syntax into proofs in any model.

This completeness proof is often done by taking some specific model of the class that is said to be *universal*, and by showing that you can go between proofs in the universal model and proofs from the syntax freely. We often try to distinguish two reciprocal functions: quote that turns a proof from the syntax into a proof in the Kripke model, and an unquote function that does the converse.

To state this in our categorical layout, we will use a different mapping going from the initial model to our Kripke model which is called the *Yoneda mapping*. The Yoneda mapping is simply defined with the following equation:

$$y(\Gamma) = \mathrm{Hom}_{\mathcal{U}}(-, \Gamma) = \mathsf{Sub}_{\mathcal{U}}\ -\ \Gamma$$

This works because, in the universal model, a context is a map from contexts of the syntax to **Prop** (because the category is posetal).

$$\begin{tikzcd} \mathcal{I} \arrow[rr, bend left, "⟦\_⟧"{name=U}] \arrow[rr, bend right, "y"'{name=D}] & & \mathcal{U} \arrow[from=U, to=D, Rightarrow, shift right=3, "q"'] \arrow[from=D, to=U, Rightarrow, shift right=3, "u"'] \end{tikzcd}$$

With this Yoneda Mapping, we want to construct two natural transformations u and q so that $u \circ q = id$. These two natural transformations witness a generalized version of completeness as we can prove the "real" completeness with the following construction.

```
completeness : {Γ Δ : I.Con} → ({Ξ : I.Con} → ⟦ Γ ⟧c Ξ → ⟦ Δ ⟧c Ξ ) → I.Sub Γ Δ
completeness {Γ} {Δ} f = q Δ Γ (f {Γ} (u Γ Γ I.id))
```

This construction means "for any list of formulæ Δ which are proven in the universal model in the context Γ (the data of these proofs is f), we have a list of proofs *in the syntax* of the formulæ of Δ in the context Γ". This is exactly completeness!

$$\begin{array}{lcl}
\mathsf{For} & : & \mathbf{Set} \\
- \Longrightarrow - & : & \mathsf{For} \to \mathsf{For} \to \mathsf{For} \\
\mathsf{R} & : & \mathrm{TM} \to \mathrm{TM} \to \mathsf{For} \\
\forall & : & (\mathrm{TM} \to \mathsf{For}) \to \mathsf{For} \\
 & & \\
\mathsf{Pf} & : & \mathsf{For} \to \mathbf{Prop}^{+} \\
\mathsf{lam} & : & (\mathsf{Pf}\ \mathrm{A} \to^{+} \mathsf{Pf}\ \mathrm{B}) \to \mathsf{Pf}\ (\mathrm{A} \Longrightarrow \mathrm{B}) \\
\mathsf{app} & : & \mathsf{Pf}\ (\mathrm{A} \Longrightarrow \mathrm{B}) \to (\mathsf{Pf}\ \mathrm{A} \to \mathsf{Pf}\ \mathrm{B}) \\
\forall i & : & (\mathrm{t} : \mathrm{TM} \to \mathsf{Pf}\ \mathrm{A}\ \mathrm{t}) \to \mathsf{Pf}\ (\forall \mathrm{A}) \\
\forall e & : & \mathsf{Pf}\ (\forall \mathrm{A}) \to (\mathrm{t} : \mathrm{TM}) \to \mathsf{Pf}\ (\mathrm{A}\ \mathrm{t})
\end{array}$$

Figure 2: IFOL Sogat Presentation

You can notice that we have only worked with negative fragments of logic (i.e. without disjunction) because then, Kripke models would not have been sufficient to make the completeness proof. This is another way to extend the work done during this internship.

During the internship, I also studied different forms of normalization proofs (i.e. saying that if we have a proof of A in Γ then we have a *normal* proof of A in Γ) using different wide subcategories of **Con**. One of those normalizations was exactly completeness. We will try to study a formalization for other logical frameworks of this "generalized normalization proof" [**RedFreeNorm1995**].

3 First-order logic (or Predicate Logic)

In this part, we will try to implement the same as we did for Propositional Logic, but this time for Predicate Logic. We again work in a simpler layout where we don't have function symbols and only have one binary relation R. And again, adding other relations and function symbols does not add a lot of new interesting content, but makes everything less readable and more complex to use.

3.1 Infinitary First-Order Logic

A naive version of first-order logic can be done by implementing ∀ as an infinitary "and" operator. I have implemented this solution, and we can look at what it added to the code.

The trick is to use an external set of terms, that is a parameter of the algebra (and therefore of all the models). This external set is called TM.

The SOGAT for this new logic is given in Figure 2. We can see that we added one constructor for formulæ: ∀ that takes as input a function from our external TM to formulæ and makes this function into a formula. We also add the introduction and elimination rule for ∀. We don't need to put a plus on this arrow as the set on which the recurring is external, so we don't have any strict positivity problem.

Therefore, the GAT we deduce from this is not a lot more complex, we simply have to add the operators as they were described in the SOGAT together with their functorality equalities for the types that are not in **Prop**.

```
--# Forall
∀∀ : {Γ : Con} → (TM → For Γ) → For Γ
[]f-∀∀ : {Γ Δ : Con} → {F : TM → For Γ} → {σ : Sub Δ Γ}
  → (∀∀ F) [ σ ]f ≡ (∀∀ (λ t → (F t) [ σ ]f))
```

```
--# ∀i and ∀e
∀i : {Γ : Con}{A : TM → For Γ} → ((t : TM) → Pf Γ (A t)) → Pf Γ (∀∀ A)
∀e : {Γ : Con}{A : TM → For Γ} → Pf Γ (∀∀ A) → (t : TM) → Pf Γ (A t)
```

The same goes to add the constructors from the syntax, this is pretty straightforward. We add a simple constructor taking a function from TM to For, and two proofs constructors which don't break strict positivity as $\forall i$ takes as input a "list of proofs indexed by TM".

We also need to add cases for proof substitutions but they are trivial cases. The same goes for the initial morphism and the proof of initiality, feel free to look at the code if you want to check it.

```
data For : Set where
  R : TM → TM → For
  _⇒_ : For → For → For
  ∀∀ : (TM → For) → For
```

```
data Pf : Con → For → Prop where
  var : PfVar Γ A → Pf Γ A
  lam : Pf (Γ ▹p A) B → Pf Γ (A ⇒ B)
  app : Pf Γ (A ⇒ B) → Pf Γ A → Pf Γ B
  ∀i : {A : TM → For} → ((t : TM) → Pf Γ (A t)) → Pf Γ (∀∀ A)
  ∀e : {A : TM → For} → Pf Γ (∀∀ A) → (t : TM) → Pf Γ (A t)
```

And Completeness proof is not harder, we just have to add the Kripke interpretation of this external forall, which is very natural, and we also need to add a case for ∀ to the definition of u and q.

We now have something that mimics first-order logic, but because the set of terms is external, the non-formulæ variables (the term variables) do not exist in our language. This logic is then not exactly first-order, and a more correct name might have been "Zero Order Logic with one Infinitary Operator". Still, this construction is much simpler than the real Predicate Logic that is presented in the next section.

3.2 (Finitary) Predicate Logic as a SOGAT

3.2.1 Making the SOGAT

The first step is again to create the SOGAT of Predicate Logic. The SOGAT is given in Figure 3. We can see that *terms* are now part of the theory, and we see that they have to be locally representable (into **Set**$^+$). This is because the proof constructor $\forall i$ now has a strictly positive arrow because terms are not external anymore.

Except for this difference, the two SOGAT for infinitary and finitary first-order logic looks very similar, and this is a great example to show that SOGATs are a good way of describing models.

3.2.2 Making a GAT out of it

So now that we want to encode a GAT from this SOGAT, we can follow the same process as we followed before. One difference is that Sub now has to be in **Set**, and so we have a lot more

```
Tm          :  Set⁺

For         :  Set
_ ⟹ _       :  For → For → For
R           :  Tm → Tm → For
∀           :  (Tm → For) → For

Pf          :  For → Prop⁺
lam         :  (Pf A →⁺ Pf B) → Pf (A ⟹ B)
app         :  Pf (A ⟹ B) → (Pf A → Pf B)
∀i          :  (t : Tm →⁺ Pf A t) → Pf (∀A)
∀e          :  Pf (∀A) → (t : Tm) → Pf (A t)
```

Figure 3: FFOL Sogat Presentation

equations that were trivial before. For example, the base category now has to make explicit all the categorical equations.

```
--# We first make the base category with its terminal object
Con : Set ℓ¹
Sub : Con → Con → Set ℓ -- It makes a category
_∘_ : {Γ Δ Ξ : Con} → Sub Δ Ξ → Sub Γ Δ → Sub Γ Ξ
∘-ass : {Γ Δ Ξ Ψ : Con}{α : Sub Γ Δ}{β : Sub Δ Ξ}{γ : Sub Ξ Ψ}
  → (γ ∘ β) ∘ α ≡ γ ∘ (β ∘ α)
id : {Γ : Con} → Sub Γ Γ
idl : {Γ Δ : Con} {σ : Sub Γ Δ} → (id {Δ}) ∘ σ ≡ σ
idr : {Γ Δ : Con} {σ : Sub Γ Δ} → σ ∘ (id {Γ}) ≡ σ
◇ : Con -- The terminal object of the category
ε : {Γ : Con} → Sub Γ ◇ -- The morphism from any object to the terminal
ε-u : {Γ : Con} → {σ : Sub Γ ◇} → σ ≡ ε {Γ}
```

We also have to add a term extension operator as we did for formula extension. One difference is that this new kind of extension has no parameters. However, it makes it so Sub have to be in **Set**, and so we have to add a lot of equations about the two-term extensions.

Please ignore the substP for now, this notion will be explained in subsection 3.4, you can just consider that this function returns its third argument unchanged and ignores the two firsts.

```
--# Tm : Set⁺
_▹ₜ : Con → Con
πₜ¹ : {Γ Δ : Con} → Sub Δ (Γ ▹ₜ) → Sub Δ Γ
πₜ² : {Γ Δ : Con} → Sub Δ (Γ ▹ₜ) → Tm Δ
_,ₜ_ : {Γ Δ : Con} → Sub Δ Γ → Tm Δ → Sub Δ (Γ ▹ₜ)
πₜ²∘,ₜ : {Γ Δ : Con} → {σ : Sub Δ Γ} → {t : Tm Δ} → πₜ² (σ ,ₜ t) ≡ t
πₜ¹∘,ₜ : {Γ Δ : Con} → {σ : Sub Δ Γ} → {t : Tm Δ} → πₜ¹ (σ ,ₜ t) ≡ σ
,ₜ∘πₜ : {Γ Δ : Con} → {σ : Sub Δ (Γ ▹ₜ)} → (πₜ¹ σ) ,ₜ (πₜ² σ) ≡ σ
,ₜ∘ : {Γ Δ Ξ : Con}{σ : Sub Γ Ξ}{δ : Sub Δ Γ}{t : Tm Γ}
  → (σ ,ₜ t) ∘ δ ≡ (σ ∘ δ) ,ₜ (t [ δ ]t)
```

```
--# → Prop⁺
_▹ₚ_ : (Γ : Con) → For Γ → Con
πₚ¹ : {Γ Δ : Con}{F : For Γ} → Sub Δ (Γ ▹ₚ F) → Sub Δ Γ
πₚ² : {Γ Δ : Con}{F : For Γ} → (σ : Sub Δ (Γ ▹ₚ F)) → Δ ⊢ (F [ πₚ¹ σ ]f)
_,ₚ_ : {Γ Δ : Con}{F : For Γ} → (σ : Sub Δ Γ) → Δ ⊢ (F [ σ ]f) → Sub Δ (Γ ▹ₚ F)
--# And its equalities
,ₚ∘πₚ : {Γ Δ : Con}{F : For Γ}{σ : Sub Δ (Γ ▹ₚ F)} → (πₚ¹ σ) ,ₚ (πₚ² σ) ≡ σ
πₚ¹∘,ₚ : {Γ Δ : Con}{σ : Sub Δ Γ}{F : For Γ}{prf : Δ ⊢ (F [ σ ]f)}
  → πₚ¹ (σ ,ₚ prf) ≡ σ
-- Equality below is useless because Γ ⊢ F is in Prop
-- πₚ²∘,ₚ : {Γ Δ : Con}{σ : Sub Δ Γ}{F : For Γ}{prf : Δ ⊢ (F [ σ ]f)}
-- → πₚ² (σ ,ₚ prf) ≡ prf
,ₚ∘ : {Γ Δ Ξ : Con}{σ : Sub Γ Ξ}{δ : Sub Δ Γ}{F : For Ξ}{prf : Γ ⊢ (F [ σ ]f)}
  → (σ ,ₚ prf) ∘ δ ≡ (σ ∘ δ) ,ₚ (substP (λ F → Δ ⊢ F) (≡sym []f-∘) (prf [ δ ]p))
```

We also have to add a Tm functor, that will have no other constructor than that of π_t^2. We don't need to change the other constructors except for $\forall i$ in which the strictly positive application becomes a proof from a term-extended context, and for $\forall e$ in which we have to “compute” what it means to apply a formula at some specific term.

```
--# Functor Con → Set called Tm
Tm : Con → Set ℓ²
_[_]t : {Γ Δ : Con} → Tm Γ → Sub Δ Γ → Tm Δ -- Action on morphisms
[]t-id : {Γ : Con} → {x : Tm Γ} → x [ id {Γ} ]t ≡ x
[]t-∘ : {Γ Δ Ξ : Con} → {α : Sub Ξ Δ}{β : Sub Δ Γ} → {t : Tm Γ}
  → t [ β ∘ α ]t ≡ (t [ β ]t) [ α ]t
-------------------------------------------------------------------
--# Lam & App
lam : {Γ : Con}{F G : For Γ} → (Γ ▹ₚ F) ⊢ (G [ πₚ¹ id ]f) → Γ ⊢ (F ⇒ G)
app : {Γ : Con}{F G : For Γ} → Γ ⊢ (F ⇒ G) → Γ ⊢ F → Γ ⊢ G
```

We now have a complete GAT for Predicate logic, and we already see that we have a lot more equations to prove.

3.3 Finding a Syntax for First-Order Logic

Even though the new SOGAT does not seem a lot more complex than the one for IFOL, there are three reasons why making the Syntax will be much harder.

1. We cannot make Sub to be in **Prop** anymore, because we have term extensions, and terms are in **Set**, and therefore we have a lot more equations.
2. We now have two parallel ways of extending contexts and that squares the difficulty
3. Formulæ now have to depend on contexts too (because they can contain terms)

However, we can still do something that simplifies greatly the making of the syntax. We can split the **Con** category in two. The first part will be related to term extensions and the second part will be related to proof extensions.

The difficulty is that those two parts are not independent. Indeed, a proof-extension-related context (or *proof context*) will depend on formulæ, which will themselves depend on term-extension-related contexts (or *term contexts*).

So, let's first define the term contexts, which simply describes how many terms they are in the context (therefore, they are isomorphic to Nat).

```
--# Term contexts are isomorphic to Nat
data Cont : Set where
  ◇t : Cont
  _▹t⁰ : Cont → Cont
```

Then, we can define already define terms (which can only be term variables, because π_t^2 is the only constructor in our simplified version), and formulæ. Both types are indexed only by a **term** context, and we will need later to make them **Con** → **Set** rather than **Cont** → **Set**. This follows the same reasoning we did when we said that Formulæ didn't have to depend on Contexts. Formulæ (and terms) never need to access proof variables, and that is the reason why we can do the separation.

```
--# A term variable is a de-bruijn variable, TmVar n ≈ ⟦0,n-1⟧
data TmVar : Cont → Set where
  tvzero : TmVar (Γt ▹t⁰)
  tvnext : TmVar Γt → TmVar (Γt ▹t⁰)

-- For now, we only have term variables (no function symbol)
data Tm : Cont → Set where
  var : TmVar Γt → Tm Γt
```

```
--# Now we can define formulæ
data For : Cont → Set where
  R : Tm Γt → Tm Γt → For Γt
  _⇒_ : For Γt → For Γt → For Γt
  ∀∀ : For (Γt ▹t⁰) → For Γt
```

We now can define **term** substitutions. The algebra gives us two ways of constructing them: with $,_t$ and as a morphism from the initial object, like in Propositional Logic. That's how we make our constructors. The two projections π_t^1 and π_t^2 are then simply obtained by eliminating the $,_t$ constructor and getting the first or second term. The equalities between ${\pi_t}^1$, ${\pi_t}^2$ and $,_t$ can be proven easily in that layout.

We also have to define the action of term substitutions on terms and formulæ. The definitions are natural, as we just go down the formulæ to get to the term variables and transform them into the terms specified in the substitution.

We also define the categorical constructors for the **Cont** category, that are, the id_t and $\circ_t$ operators, and we have to show their three categorical laws. Please note that these are not the equalities we will use in the final algebra, because they are only related to Subt and not to the more general Sub (but we will surely use them later).

We also show the functorality of _[_]t and _[_]f. One can notice we use some helper functions that are called wk_t and lf_t. Those stand for weakening and lifting. I don't show their definitions here, because they are only helper functions, but wk_t means that from something going from a context Γ, we create something going from the context $\Gamma \rhd_t$ that does not use the added variable. lf_t means that we add one variable to the source, we add one variable to the goal, and this variable is mapped to itself.

We do not need renamings this time to define them because we do not have any term constructor aside from term variables.

```
--# Then we define term substitutions
data Subt : Cont → Cont → Set where
  εt : Subt Γt ◇t
  _,t_ : Subt Δt Γt → Tm Δt → Subt Δt (Γt ▷t⁰)
```

```
--# We now define the action of term substitutions on terms
_[_]t : Tm Γt → Subt Δt Γt → Tm Δt
var tvzero [ σ ,t t ]t = t
var (tvnext tv) [ σ ,t t ]t = var tv [ σ ]t
```

```
--# We can now subst on formulæ
_[_]f : For Γt → Subt Δt Γt → For Δt
(R t u) [ σ ]f = R (t [ σ ]t) (u [ σ ]t)
(A ⇒ B) [ σ ]f = (A [ σ ]f) ⇒ (B [ σ ]f)
(∀∀ A) [ σ ]f = ∀∀ (A [ lftσt σ ]f)
```

```
--# We now can define identity and composition of term substitutions
idt : Subt Γt Γt
idt {◇t} = εt
idt {Γt ▷t⁰} = lftσt (idt {Γt})
_∘t_ : Subt Δt Γt → Subt Ξt Δt → Subt Ξt Γt
εt ∘t β = εt
(α ,t x) ∘t β = (α ∘t β) ,t (x [ β ]t)
```

We can now start creating the other category related to proof variables. For that, we can define the base set of the category as we did with term contexts. But this time, our extension constructor has another parameter: The formula the added proof proves. Therefore, we have a dependency of Conp on Cont, because formulæ are defined in Cont.

This new **Conp** category is indeed a functor from **Cont** to **Set**. So we have to define its action on **Cont**'s morphisms. And we also have to prove that this action respects the functorality of **Conp**.

We will finally create another operator on **Conp**: ▷tp. This operator does a term extension of a proof context, which doesn't add any information to the proofs, it only adds one unused variable to them (we substitute all the proofs with wk$_t$ id$_t$). It can be understood as the action of the general ▷$_t$ functor on **Conp**.

```
--# We can now define proof contexts, which are indexed by a term context
-- i.e. we know which terms a proof context can use
data Conp : Cont → Set where
  ◇p : Conp Γt
  _▷p⁰_ : Conp Γt → For Γt → Conp Γt
```

```
--# The actions of Subt's is extended to contexts
_[_]c : Conp Γt → Subt Δt Γt → Conp Δt
◇p [ σt ]c = ◇p
(Γp ▷p⁰ A) [ σt ]c = (Γp [ σt ]c) ▷p⁰ (A [ σt ]f)
```

```
--# We can also add a term that will not be used in the formulæ already present
-- (that's why we use wktσt)
_▷tp : Conp Γt → Conp (Γt ▷t⁰)
Γ ▷tp = Γ [ wktσt idt ]c
```

We now have everything we need to define proofs, and we simply implement all the constructors from the GAT. You can notice that we have separated term and proof contexts in the definitions, but it is only for the sake of readability because operations on proofs will often modify the proof context without changing the term context.

```
--# With those contexts, we have everything to define proofs
data PfVar : (Γt : Cont) → (Γp : Conp Γt) → For Γt → Prop  where
  pvzero : {A : For Γt} → PfVar Γt (Γp ▹p⁰ A) A
  pvnext : {A B : For Γt} → PfVar Γt Γp A → PfVar Γt (Γp ▹p⁰ B) A

data Pf : (Γt : Cont) → (Γp : Conp Γt) → For Γt → Prop  where
  var : {A : For Γt} → PfVar Γt Γp A → Pf Γt Γp A
  app : {A B : For Γt} → Pf Γt Γp (A ⇒ B) → Pf Γt Γp A → Pf Γt Γp B
  lam : {A B : For Γt} → Pf Γt (Γp ▹p⁰ A) B → Pf Γt Γp (A ⇒ B)
  p∀∀e : {A : For (Γt ▹t⁰)} → {t : Tm Γt} → Pf Γt Γp (∀∀ A) → Pf Γt Γp (A [ idₜ ,ₜ t ]f)
  p∀∀i : {A : For (Γt ▹t⁰)} → Pf (Γt ▹t⁰) (Γp ▹tp) A → Pf Γt Γp (∀∀ A)
```

We also define the action of term substitutions on proofs (because Pf is a functor that is indexed by **Cont**). Terms substitutions do nothing to the proofs, they only affect formulæ and terms, which you cannot find inside a proof. So it is only changing the type of a proof from Pf Δ_t Δ_p A to Pf Γ_t $\Delta_p[\sigma_t]$ $A[\sigma_t]$.

```
--# The action on Cont's morphisms of Pf functor
_[_]pvₜ : {A : For Δt}→ PfVar Δt Δp A → (σ : Subt Γt Δt)→ PfVar Γt (Δp [ σ ]c) (A [ σ ]f)
pvzero [ σ ]pvₜ = pvzero
pvnext pv [ σ ]pvₜ = pvnext (pv [ σ ]pvₜ)
_[_]pₜ : {A : For Δt} → Pf Δt Δp A → (σ : Subt Γt Δt) → Pf Γt (Δp [ σ ]c) (A [ σ ]f)
var pv [ σ ]pₜ = var (pv [ σ ]pvₜ)
app pf pf' [ σ ]pₜ = app (pf [ σ ]pₜ) (pf' [ σ ]pₜ)
lam pf [ σ ]pₜ = lam (pf [ σ ]pₜ)
_[_]pₜ {Δp = Δp} {Γt = Γt} (p∀∀e {A = A} {t = t} pf) σ =
  substP (λ F → Pf Γt (Δp [ σ ]c) F) (≡tran² (≡sym []f-∘) (cong (λ σ → A [ σ ]f)
  (cong  _,ₜ_ (≡tran² wkₜ∘ₜ,ₜ idrₜ (≡sym idlₜ)) refl)) ([]f-∘))
  (p∀∀e {t = t [ σ ]t} (pf [ σ ]pₜ))
_[_]pₜ {Γt = Γt} (p∀∀i pf) σ
  = p∀∀i (substP (λ Ξp → Pf (Γt ▹t⁰) (Ξp) _) ▹tp-lfₜ (pf [ lfₜσₜ σ ]pₜ))
```

Again, before we can define proof substitutions, we need to define *renamings* as we did for Propositional Logic. They allow us to define proof-weakening and the identity substitution. Those two functions are needed to define fully-featured substitutions. But please note that those proof substitutions are defined with a fixed term context. So they are weaker than the final substitutions we will have to define for our algebra, which have to be able to be morphisms between contexts that differ on term-contexts and proof context at the same time. But this is not an issue and we can define all substitutions using only those restricted proof-substitutions.

Again, these substitutions are a functor over the category **Cont**, so we have to construct the action on morphisms, which is simply applying the transformations to all the proofs contained in the substitution.

```
--# We now can create Renamings, a subcategory from (Conp,Subp) that
-- A renaming from a context Γₚ to a context Δₚ means when they are seen
-- as lists, that every element of Γₚ is an element of Δₚ
-- In other words, we can prove Γₚ from Δₚ using only proof variables (var)
data Ren : Conp Γₜ → Conp Γₜ → Set where
  zeroRen : Ren ◇p Γₚ
  leftRen : {A : For Δₜ} → PfVar Δₜ Δₚ A → Ren Δₚ' Δₚ → Ren (Δₚ' ▹p⁰ A) Δₚ
```

```
--# But we need something stronger than just renamings
-- introducing: Proof substitutions
-- They are basicly a list of proofs for the formulæ contained in
-- the goal context.
-- It is not defined between all contexts, only those with the same term context
data Subp : {Δₜ : Cont} → Conp Δₜ → Conp Δₜ → Prop where
  εₚ : Subp Δₚ ◇p
  _,ₚ_ : {A : For Δₜ} → (σ : Subp Δₚ Δₚ') → Pf Δₜ Δₚ A → Subp Δₚ (Δₚ' ▹p⁰ A)
```

```
--# The action of Cont's morphisms on Subp
_[_]σₚ : Subp {Δₜ} Δₚ Δₚ' → (σ : Subt Γₜ Δₜ) → Subp {Γₜ} (Δₚ [ σ ]c) (Δₚ' [ σ ]c)
εₚ [ σₜ ]σₚ = εₚ
(σₚ ,ₚ pf) [ σₜ ]σₚ = (σₚ [ σₜ ]σₚ) ,ₚ (pf [ σₜ ]pₜ)
```

With this new kind of substitution, we can substitute on proofs (that's what they're for). In other words, we have to describe the action of the Pf functor on the morphisms of the category **Conp**.

Proof substitution of proofs is simple to define, as we only get down the proof until we find a proof variable, in which case we replace it with the proof in the substitution, as we did for term-substitutions and formulæ.

Again, we have defined wk$_p$ and lf$_p$. The former will add an unused proof variable (an assumption) to a substitution, and the latter adds a formula to the goal which is proven using an assumption.

```
_[_]p : {A : For Δₜ} → Pf Δₜ Δₚ A → (σ : Subp {Δₜ} Δₚ' Δₚ) → Pf Δₜ Δₚ' A
var pvzero [ σ ,ₚ pf ]p = pf
var (pvnext pv) [ σ ,ₚ pf ]p = var pv [ σ ]p
app pf pf [ σ ]p = app (pf [ σ ]p) (pf [ σ ]p)
lam pf [ σ ]p = lam (pf [ wkₚσₚ σ ,ₚ var pvzero ]p)
p∀∀e pf [ σ ]p = p∀∀e (pf [ σ ]p)
p∀∀i pf [ σ ]p = p∀∀i (pf [ wkₜσₚ σ ]p)
```

This definition of proof substitutions allows us to define the categorical operators for proof substitutions.

Of course, we have to show that the categorical laws apply to them, that _[_]σ_p respects them, as we did for the Subt's laws.

The proofs of these laws contain a lot of transports which make them not very readable. This transport issue is explained in subsection 3.4.

```
--# We can now define identity and composition on proof substitutions
idₚ : Subp {Δₜ} Δₚ Δₚ
idₚ {Δₚ = ◇p} = εₚ
idₚ {Δₚ = Δₚ ▷p⁰ x} = lfₚσₚ (idₚ {Δₚ = Δₚ})
_∘ₚ_ : {Γₚ Δₚ Ξₚ : Conp Δₜ} → Subp {Δₜ} Δₚ Ξₚ → Subp {Δₜ} Γₚ Δₚ → Subp {Δₜ} Γₚ Ξ
εₚ ∘ₚ β = εₚ
(α ,ₚ pf) ∘ₚ β = (α ∘ₚ β) ,ₚ (pf [ β ]p)
```

The last step is to merge our two kinds of contexts and substitutions to match the algebra.

For contexts, we simply have a record type containing both a Cont and a Conp that depends on it. But for substitutions, we defined Subp to only describe substitutions between Conps with *the same Cont*. Our solution is to understand global substitutions as a sequence of two substitutions, first on terms, and then on proofs. That's why in the definition below, the "proof" part of the substitution can only be applied to proofs on $\Delta_p[t]$, i.e. proofs that have already been term-substituted by the "term" part of the substitution. The definition of the global proof substitution matching the shape given to the algebra is given after, and you can see that substitutions work exactly as described, first, the proof and everything inside is substituted using the "term" part of the substitution, and only after that the "proof" part of the substitution is applied.

```
--# We can now merge the two notions of contexts, substitutions, and everything
record Con : Set where
  constructor con
  field
    t : Cont
    p : Conp t

record Sub (Γ : Con) (Δ : Con) : Set where
  constructor sub
  field
    t : Subt (Con.t Γ) (Con.t Δ)
    p : Subp {Con.t Γ} (Con.p Γ) ((Con.p Δ) [ t ]c)

pf [ σ ]p = (pf [ Sub.t σ ]pₜ) [ Sub.p σ ]p
```

And we can finally define our categorical operators for the merged version. They are only a concatenation of the previously defined operators, so no hard logic here. Those constructions are followed by the equations needed for the algebra.

```
--# (Con,Sub) is a category with an initial object
id : Sub Γ Γ
id {Γ} = sub idₜ (substP (Subp _) (≡sym []c-id) idₚ)
_∘_ : Sub Δ Ξ → Sub Γ Δ → Sub Γ Ξ
sub αₜ αₚ ∘ sub βₜ βₚ = sub (αₜ ∘ₜ βₜ) (substP (Subp _) (≡sym []c-∘) (αₚ [ βₜ ]σₚ) ∘ₚ βₚ)

--# We have our two context extension operators
_▷t : Con → Con
Γ ▷t = con ((Con.t Γ) ▷t⁰) (Con.p Γ ▷tp)
_▷p_ : (Γ : Con) → For (Con.t Γ) → Con
Γ ▷p A = con (Con.t Γ) (Con.p Γ ▷p⁰ A)
```

And we now have a working syntax for Predicate Logic. Unfortunately, I did not yet prove that it was indeed an initial model, because for the proof i need to work with a lot of transport hell, which is not technically hard to solve but takes a lot of time.

3.4 Transport Hell

For you to understand the code, we will now explain what are transports. I'll do this with the example of the definition of id in the Finitary First Order Logic syntax.

To construct it, we use id_t and id_p, merged with the constructor sub. Here are the types of the different elements in an array.

id	$\mathsf{Sub}\ (\mathsf{con}\ \Gamma_t\ \Gamma_p)\ (\mathsf{con}\ \Gamma_t\ \Gamma_p)$
id_t	$\mathsf{Subt}\ \Gamma_t\ \Gamma_t$
id_p	$\mathsf{Subp}\ \Gamma_p\ \Gamma_p$
sub	$(\sigma : \mathsf{Subt}\ \Gamma_t\ \Delta_t)$ $\to \mathsf{Subp}\ \Gamma_p\ (\Delta_p[\sigma]c)$ $\to \mathsf{Sub}\ (\mathsf{con}\ \Gamma_t\ \Gamma_p)\ (\mathsf{con}\ \Delta_t\ \Delta_p)$

But if we try to construct id, then we will eventually end up with the following goal:

$$\mathsf{id} = \mathsf{sub}\ \mathsf{id}_t\ \mathbf{?} \qquad \textit{with} \qquad \mathbf{?} : \mathsf{Subp}\ \Gamma_p\ (\Gamma_p[\mathsf{id}_t]c)$$

But then Agda will complain if we give it as a goal the expression "id_p". Indeed, it is not trivial for them that $\mathsf{Subp}\ \Gamma_p\ (\Gamma_p[\mathsf{id}_t]c)$ is the same as $\mathsf{Subp}\ \Gamma_p\ \Gamma_p$. Even if we can easily prove the equality between those two elements of **Prop** (the proof of the equality is derived from the fact that Conp is a functor from **Cont** to **Set**, and therefore it has to respect the identity of **Conp**).

That's where transport comes in. The function name is "subst" but we call it *transport* to avoid confusion with Sub. Its definition is as follows:

$$\mathsf{subst} : \{A : \mathbf{Set}\}(P : A \to \mathbf{Prop})\{a\ a' : A\} \to a \equiv a' \to P\ a \to P\ a'$$

This is exactly the thing that will solve our problem. With the equality between the two elements of **Prop**, we can now convert id_p to something that will correctly match the goal required by Agda.

This section is called "Transport Hell" because those transports between equal sets can get annoying. In a former version of the syntax, the Subp were **Set**s instead of **Prop**s. And that created a lot of equalities related to polymorphic functions (quite all Subp's related functions were then polymorphic, as they would depend on proof contexts). And to solve those equalities, you have to extract what are precisely the polymorphic functions you are working with. You can see those proofs on former versions of the syntax, they are long and not very interesting (as it is mainly trying to tweak our equation so that Agda understands that two types are equal).

4 Summary

During this internship, I created a set of Agda definitions for different frameworks of logic. Even though this has already been done by other people, my definitions are directly derived from the SOGAT definitions. It makes the files and this report a good example of how to use SOGAT, and how to convert them into GAT.

We also created a proof of completeness in this layout, which shows the usefulness of the theory. I did not make the completeness proof of predicate logic, but it is the next thing that will be added to the project.

This project is also a great entry point to study the different ways a SOGAT can be turned into a GAT. We need further work to understand the specificities of the SOGATs used in this report that allowed us to turn them directly into strict syntaxes (it is not the case for any SOGAT, especially for those with equations). Making a formalization of this transformation for a specific class of SOGAT would be a great step forward.

5 Bibliography

Appendices

A Agda Code

The Agda code has been written to A4 paper and is given in the next pages

1. PropUtil
2. ListUtil
3. ZOL
4. ZOLInitial
5. ZOLCompleteness
6. IFOL
7. IFOLInitial
8. IFOLCompleteness
9. FFOL
10. FFOLInitial

```
{-# OPTIONS --prop --rewriting #-}

module PropUtil where

  open import Agda.Primitive
  variable ℓ ℓ' : Level

  data ⊥s : Set where
  record ⊤s : Set ℓ where
    constructor tts


  -- ⊥ is a data with no constructor
  -- ⊤ is a record with one always-available constructor
  data ⊥ : Prop where
  record ⊤ : Prop ℓ where
    constructor tt


  data _∨_ : Prop → Prop → Prop where
    inj₁ : {P Q : Prop} → P → P ∨ Q
    inj₂ : {P Q : Prop} → Q → P ∨ Q

  record _∧_ (P Q : Prop ℓ) : Prop ℓ where
    constructor ⟨_,_⟩
    field
      p : P
      q : Q

  infixr 10 _∧_
  infixr 11 _∨_

  -- ∧ elimination
  proj₁ : {P Q : Prop ℓ} → P ∧ Q → P
  proj₁ pq = _∧_.p pq
  proj₂ : {P Q : Prop ℓ} → P ∧ Q → Q
  proj₂ pq = _∧_.q pq

  -- ∨ elimination
  dis : {P Q S : Prop} → (P ∨ Q) → (P → S) → (Q → S) → S
  dis (inj₁ p) ps qs = ps p
  dis (inj₂ q) ps qs = qs q

  -- ¬ is a shorthand for « → ⊥ »
  ¬ : Prop → Prop
  ¬ P = P → ⊥

  -- ⊥ elimination
  case⊥ : {P : Prop} → ⊥ → P
  case⊥ ()

  -- ⇔ shorthand
  _⇔_ : Prop → Prop → Prop
  P ⇔ Q = (P → Q) ∧ (Q → P)


  -- Syntactic sugar for writing applications
  infixr 200 _$_
  _$_ : {T U : Prop} → (T → U) → T → U
  h $ t = h t
  postulate _≈_ : ∀{ℓ}{A : Set ℓ}(a : A) → A → Set ℓ
  infix 3 _≡_
  data _≡_ {ℓ}{A : Set ℓ}(a : A) : A → Prop ℓ where
    refl : a ≡ a
  {-# BUILTIN REWRITE _≡_ #-}

  ≡sym : {ℓ : Level} → {A : Set ℓ}→ {a a' : A} → a ≡ a' → a' ≡ a
  ≡sym refl = refl


  ≡tran : {ℓ : Level} {A : Set ℓ} → {a a' a'' : A} → a ≡ a' → a' ≡ a'' → a ≡ a''
  ≡tran² : {ℓ : Level} {A : Set ℓ} → {a₀ a₁ a₂ a₃ : A} → a₀ ≡ a₁ → a₁ ≡ a₂ → a₂ ≡ a₃
→ a₀ ≡ a₃
  ≡tran³ : {ℓ : Level} {A : Set ℓ} → {a₀ a₁ a₂ a₃ a₄ : A} → a₀ ≡ a₁ → a₁ ≡ a₂ → a₂ ≡
a₃ → a₃ ≡ a₄ → a₀ ≡ a₄
  ≡tran⁴ : {ℓ : Level} {A : Set ℓ} → {a₀ a₁ a₂ a₃ a₄ a₅ : A} → a₀ ≡ a₁ → a₁ ≡ a₂ → a₂
≡ a₃ → a₃ ≡ a₄ → a₄ ≡ a₅ → a₀ ≡ a₅
  ≡tran⁵ : {ℓ : Level} {A : Set ℓ} → {a₀ a₁ a₂ a₃ a₄ a₅ a₆ : A} → a₀ ≡ a₁ → a₁ ≡ a₂ →
a₂ ≡ a₃ → a₃ ≡ a₄ → a₄ ≡ a₅ → a₅ ≡ a₆ → a₀ ≡ a₆
  ≡tran⁶ : {ℓ : Level} {A : Set ℓ} → {a₀ a₁ a₂ a₃ a₄ a₅ a₆ a₇ : A} → a₀ ≡ a₁ → a₁ ≡
a₂ → a₂ ≡ a₃ → a₃ ≡ a₄ → a₄ ≡ a₅ → a₅ ≡ a₆ → a₆ ≡ a₇ → a₀ ≡ a₇
  ≡tran⁷ : {ℓ : Level} {A : Set ℓ} → {a₀ a₁ a₂ a₃ a₄ a₅ a₆ a₇ a₈ : A} → a₀ ≡ a₁ → a₁
≡ a₂ → a₂ ≡ a₃ → a₃ ≡ a₄ → a₄ ≡ a₅ → a₅ ≡ a₆ → a₆ ≡ a₇ → a₇ ≡ a₈ → a₀ ≡ a₈
  ≡tran  refl refl = refl
  ≡tran² refl refl refl = refl
  ≡tran³ refl refl refl refl = refl
  ≡tran⁴ refl refl refl refl refl = refl
  ≡tran⁵ refl refl refl refl refl refl = refl
  ≡tran⁶ refl refl refl refl refl refl refl = refl
  ≡tran⁷ refl refl refl refl refl refl refl refl = refl

  cong : {ℓ ℓ' : Level}{A : Set ℓ}{B : Set ℓ'} → (f : A → B) → {a a' : A} → a ≡
a' → f a ≡ f a'
  cong f refl = refl
  cong₂ : {ℓ ℓ' ℓ'' : Level}{A : Set ℓ}{B : Set ℓ'}{C : Set ℓ''} → (f : A → B →
C) → {a a' : A} {b b' : B} → a ≡ a' → b ≡ b' → f a b ≡ f a' b'
  cong₂ f refl refl = refl
  cong₃ : {ℓ ℓ' ℓ'' ℓ''' : Level}{A : Set ℓ}{B : Set ℓ'}{C : Set ℓ''}{D : Set
ℓ'''} → (f : A → B → C → D) → {a a' : A} {b b' : B}{c c' : C} → a ≡ a' → b ≡ b' →
c ≡ c' → f a b c ≡ f a' b' c'
  cong₃ f refl refl refl = refl

  -- We can make a proof-irrelevant substitution
  substP      : ∀{ℓ}{A : Set ℓ}{ℓ'}(P : A → Prop ℓ'){a a' : A} → a ≡ a' → P a → P
a'
  substP P refl h = h
  substPP      : ∀{ℓ}{A B : Set ℓ}{Q : A → Prop ℓ}{ℓ'}(P : {k : A} → Q k → Prop
ℓ'){a a' : A}{x : Q a}
    → (eq : a ≡ a') → P x → P (substP Q eq x)
  substPP P refl h = h
  substP² : ∀{ℓ ℓ' ℓ'' : Level}{A : Set ℓ}{B : Set ℓ'}(P : A → B → Prop ℓ''){a a'
: A}{b b' : B} → a ≡ a' → b ≡ b' → P a b → P a' b'
  substP² P refl refl p = p


  postulate coe : ∀{ℓ}{A B : Set ℓ} → A ≡ B → A → B
  postulate coerefl : ∀{ℓ}{A : Set ℓ}{eq : A ≡ A}{a : A} → coe eq a ≡ a

  postulate ≡fun : {ℓ ℓ' : Level} → {A : Set ℓ} → {B : Set ℓ'} → {f g : A → B} →
((x : A) → (f x ≡ g x)) → f ≡ g
  postulate ≡fun' : {ℓ ℓ' : Level} → {A : Set ℓ} → {B : A → Set ℓ'} → {f g : (a :
```

```
A) → B a} → ((x : A) → (f x ≡ g x)) → f ≡ g

  coeaba : {ℓ : Level}{A B : Set ℓ}{eq1 : A ≡ B}{eq2 : B ≡ A}{a : A} → coe eq2
(coe eq1 a) ≡ a
  coeaba {eq1 = refl} = ≡tran coerefl coerefl

  coefgcong : {ℓ : Level}{A B C D : Set ℓ}{aa : A ≡ A}{dd : D ≡ D}{cb : C ≡ B}{f
: B → A}{g : D → C}{x : D} → f (coe cb (g (coe dd x))) ≡ coe aa (f (coe cb (g
x)))
  coefgcong {cb = refl} {f} {g} = ≡tran (cong f (cong (coe _) (cong g coerefl)))
(≡sym coerefl)

  coecong : {ℓ : Level}{A B : Set ℓ}{eq : B ≡ B}{eq' : A ≡ A}(f : A → B){x : A} →
(f (coe eq' x)) ≡ (coe eq (f x))

  coecong f = ≡tran (cong f coerefl) (≡sym coerefl)

  coecoe-coe : {ℓ : Level}{A B C : Set ℓ}{eq1 : B ≡ A}{eq2 : C ≡ B}{x : C} → coe
eq1 (coe eq2 x) ≡ coe (≡tran eq2 eq1) x
  coecoe-coe {eq1 = refl} {refl} = coerefl

  coe-f : {ℓ : Level}{A B C D : Set ℓ} → (A → B) → A ≡ C → B ≡ D → C → D
  coe-f f ac bd x = coe bd (f (coe (≡sym ac) x))
  coewtf : {ℓ : Level}{A B C D E F G H : Set ℓ}{ab : A ≡ B}{cd : C ≡ D}{ef : E ≡
F}{gh : G ≡ H}{f : F → B}{g : H → E}{x : G} →
             {fd : F ≡ D} → f (coe ef (g (coe gh x))) ≡ coe ab ((coe-f f fd
(≡sym ab)) (coe cd ((coe-f g (≡sym gh) (≡tran² ef fd (≡sym cd))) x)))
  coewtf {ab = refl} {refl} {refl} {refl} {f} {g} {fd = refl} = ≡tran (cong f
(cong (coe _) (≡sym coeaba))) (≡sym coeaba)

  coeshift : {ℓ : Level}{A B : Set ℓ}{x : A} {y : B} {eq : A ≡ B} → coe eq x ≡ y
→ x ≡ coe (≡sym eq) y
  coeshift {eq=refl} refl = ≡sym coeaba

  subst : ∀{ℓ}{A : Set ℓ}{ℓ'}(P : A → Set ℓ'){a a' : A} → a ≡ a' → P a → P a'
  subst P eq p = coe (cong P eq) p
  subst² : ∀{ℓ ℓ' ℓ'' : Level}{A : Set ℓ}{B : Set ℓ'}(P : A → B → Set ℓ''){a a' :
A}{b b' : B} → a ≡ a' → b ≡ b' → P a b → P a' b'
  subst² P eq eq' p = coe (cong₂ P eq eq') p
  subst¹P : ∀{ℓ ℓ' ℓ'' : Level}{A : Set ℓ}{B : Prop ℓ'}(P : A → B → Set ℓ''){a a'
: A}{b : B} → a ≡ a' → P a b → P a' b
  subst¹P P {b = b} eq p = coe (cong (λ x → P x b) eq) p

  --{-# REWRITE transprefl   #-}

  coereflrefl : {ℓ : Level}{A : Set ℓ}{eq eq' : A ≡ A}{a : A} → coe eq (coe eq'
a) ≡ a
  coereflrefl = ≡tran coerefl coerefl

  substrefl   : ∀{ℓ}{A : Set ℓ}{ℓ'}{P : A → Set ℓ'}{a : A}{e : a ≡ a}{p : P a} →
subst P e p ≡ p
  substrefl = coerefl
  --{-# REWRITE substrefl   #-}
  substreflrefl : {ℓ ℓ' : Level}{A : Set ℓ}{P : A → Set ℓ'}{a : A}{e : a ≡ a}{p :
P a} → subst P e (subst P e p) ≡ p
  substreflrefl {P = P} {a} {e} {p} = ≡tran (substrefl {P = P} {a = a} {e = e} {p
= subst P e p}) (substrefl {P = P} {a = a} {e = e} {p = p})

  cong₂' : {ℓ ℓ' ℓ'' : Level}{A : Set ℓ}{B : A → Set ℓ'}{C : Set ℓ''} → (f : (a :
A) → B a → C) → {a a' : A} {b : B a} {b' : B a'} → (eq : a ≡ a') → subst B eq b ≡
b' → f a b ≡ f a' b'
  cong₂' f {a} refl refl = cong (f a) (≡sym coerefl)

  congsubst : {ℓ ℓ' : Level}{A : Set ℓ}{P : A → Set ℓ'}{a a' : A}{e : a ≡ a}{p :
P a}{p' : P a} → p ≡ p' → subst P e p ≡ subst P e p'
```

```
  congsubst {P = P} {e = refl} h = cong (subst P refl) h

  substpoly : {ℓ ℓ' : Level}{A : Set ℓ}{P : A → Set ℓ'}{f : {ξ : A} → P ξ}
    {α β : A}{eq : α ≡ β}
    → subst P eq (f {α}) ≡ f {β}
  substpoly {eq = refl} = coerefl

  substfpoly : {ℓ ℓ' : Level}{A : Set ℓ}{P R : A → Set ℓ'}{α β : A}
    {eq : α ≡ β} {f : {ξ : A} → R ξ → P ξ} {x : R α}
    → coe (cong P eq) (f {α} x) ≡ f (coe (cong R eq) x)
  substfpoly {eq = refl} {f} = ≡tran coerefl (cong f (≡sym coerefl))

  substppoly : {ℓ ℓ' ℓ'' ℓ''' : Level}{A : Set ℓ}{P : A → Set ℓ'}{R : A → Set
ℓ''}{Q : A → Set ℓ'''}{α β : A}
    {eq : α ≡ β}{f : {ξ : A} → R ξ → Q ξ → P ξ} {x : R α} {y : Q α}
    → coe (cong P eq) (f {α} x y) ≡ f {β} (coe (cong R eq) x) (coe (cong Q eq) y)
  substppoly {eq = refl} {f}{x}{y} = ≡tran coerefl (cong₂ f (≡sym coerefl) (≡sym
coerefl))

  substfpoly⁴ : {ℓ ℓ' ℓ'' : Level}{A : Set ℓ}{P R : A → Set ℓ'}{Q : A → Prop ℓ''}
{α β : A}
    {eq : α ≡ β} {f : {ξ : A} → R ξ → Q ξ → P ξ} {x : R α} {y : Q α}
    → coe (cong P eq) (f {α} x y) ≡ f {β} (coe (cong R eq) x) (substP Q eq y)
  substfpoly⁴ {eq = refl} {f}{x}{y} = ≡tran² coerefl (cong (λ x → f x y) (≡sym
coerefl)) refl
  substfgpoly : {ℓ ℓ' : Level}{A B : Set ℓ} {P Q : A → Set ℓ'} {R : B → Set ℓ'}
{F : B → A} {α β : A} {ε φ : B}
       {eq₁ : α ≡ β} {eq₂ : F ε ≡ α} {eq₃ : F φ ≡ β} {eq₄ : ε ≡ φ}
       {g : {a : A} → Q a → P a} {f : {b : B} → R b → Q (F b)} {x : R ε}
    → g {β} (subst Q eq₃ (f {φ} (subst R eq₄ x))) ≡ subst P eq₁ (g {α} (subst Q
eq₂ (f {ε} x)))
  substfgpoly {P = P} {Q} {R} {eq₁ = refl} {refl} {refl} {refl} {g} {f} = ≡tran³
(cong g (substrefl {P = Q} {e = refl})) (cong g (cong f (substrefl {P = R} {e =
refl}))) (cong g (≡sym (substrefl {P = Q} {e = refl}))) (≡sym (substrefl {P = P}
{e = refl}))

  {-# BUILTIN EQUALITY _≡_ #-}

  coep² : {ℓ₁ ℓ₂ ℓ₃ ℓ₄ : Level} {A : Set ℓ₁} {R : A → Set ℓ₂}{T : A → Set ℓ₃}{S : A
→ Set ℓ₄}{α β : A}
    {p : {ξ : A} → R ξ → T ξ → S ξ}{x : R α}{y : T α}{eq : α ≡ β}
    → subst S (≡sym eq) (p {β} (subst R eq x) (subst T eq y)) ≡ p {α} x y
  coep² {S = S}{p = p}{x}{y}{refl} = ≡tran (substrefl {P = S} {e = refl}) (cong₂
p (substrefl {a = x} {e = refl}) (substrefl {a = y} {e = refl}))
  coep²'' : {ℓ ℓ' : Level} {A : Set ℓ} {R S : A → Set ℓ'}{T : A → Prop ℓ'}{α β :
A}
    {p : {ξ : A} → R ξ → T ξ → S ξ}{x : R α}{y : T α}{eq : α ≡ β}
    → subst S (≡sym eq) (p {β} (subst R eq x) (substP T eq y)) ≡ p {α} x y
  coep²'' {S = S}{p = p}{x}{y}{refl} = ≡tran (substrefl {P = S} {e = refl}) (cong
(λ X → p X y) (substrefl {a = x} {e = refl}))
  coep²' : {ℓ ℓ' : Level} {A : Set ℓ} {R T S : A → Set ℓ'}{α β : A}
    {p : {ξ : A} → R ξ → T ξ → S ξ}{x : R β}{y : T α}{eq : α ≡ β}
    → subst S (≡sym eq) (p {β} x (subst T eq y)) ≡ p {α} (subst R (≡sym eq) x) y
  coep²' {S = S}{p = p}{x}{y}{refl} = ≡tran (substrefl {P = S} {e = refl}) (cong₂
p (≡sym (substrefl {a = x} {e = refl})) (substrefl {a = y} {e = refl}))

  coep∘ : {ℓ ℓ' : Level}{A : Set ℓ} {R : A → A → Set ℓ'} {α β γ δ ε φ : A}
        {p : {x y z : A} → R x y → R z x → R z y}{x : R β γ}{y : R α β}
        {eq1 : α ≡ δ} {eq2 : β ≡ ε} {eq3 : γ ≡ φ} →
        coe (cong₂ R (≡sym eq1) (≡sym eq3)) (p (coe (cong₂ R eq2 eq3) x) (coe
(cong₂ R eq1 eq2) y)) ≡ p x y
  coep∘ {p = p}{eq1 = refl}{refl}{refl} = ≡tran coerefl (cong₂ p coerefl coerefl)
  coep∘-helper = λ {ℓ ℓ' ℓ'' : Level}{B : Set ℓ}{A : B → Set ℓ''} {R : (b : B) →
A b → A b → Set ℓ'}
    {b₁ b₂ : B} {α γ : A b₁} {δ φ : A b₂}
```

```
        {eq0 : b₁ ≡ b₂}{eqa : subst A eq0 α ≡ δ}{eqb : subst A eq0 γ ≡ φ}
     → (≡tran² (cong (R b₂ δ) (≡sym eqb)) (cong (λ X → R b₂ X (subst A eq0 γ))
(≡sym eqa)) (≡tran (≡sym (substrefl {P = λ X → Set ℓ'}{a = R b₂ (subst A eq0 α)
(subst A eq0 γ)}{e = refl})) (coep² {p = λ {t} x y → R t x y}{eq = eq0})))
  coep∘-helper-coe : {ℓ ℓ' ℓ'' : Level}{B : Set ℓ}{A : B → Set ℓ''} {R : (b : B)
→ A b → A b → Set ℓ'}
    {b₁ b₂ : B} {α γ : A b₁} {δ φ : A b₂}
        {eq0 : b₁ ≡ b₂}{eqa : subst A eq0 α ≡ δ}{eqb : subst A eq0 γ ≡ φ} → {a : R
b₂ δ φ}{a' : R b₁ α γ} → coe (coep∘-helper {eq0 = eq0} {eqa = eqa} {eqb = eqb}) a
≡ a
  coep∘-helper-coe {eq0 = refl}{refl}{refl} = coerefl


  coefun : {ℓ : Level}{A B C : Set ℓ}{f : B → C}{eq : A ≡ B}
    → f ≡ coe (cong (λ X → X → C) eq) (λ (x : A) → f (coe eq x))
  coefun {f = f} {eq = refl} = ≡tran (≡fun (λ x → cong f (≡sym (coerefl {eq =
refl})))) (≡sym (coerefl {eq = refl}))
  coefun' : {ℓ : Level}{A B C : Set ℓ}{f : A → B}{eq : B ≡ C}
    → (λ (x : A) → coe eq (f x)) ≡ coe (cong (λ X → A → X) eq) (λ (x : A) → f x)
  coefun' {f = f} {eq = refl} = ≡tran (≡fun (λ x → coerefl)) (≡sym coerefl)





  data Nat : Set where
    zero : Nat
    succ : Nat → Nat

  {-# BUILTIN NATURAL Nat #-}

  record _×_ (A : Set ℓ) (B : Set ℓ') : Set (ℓ ⊔ ℓ') where
    constructor _,×_
    field
      a : A
      b : B

  record _×'_ (A : Set ℓ) (B : Prop ℓ') : Set (ℓ ⊔ ℓ') where
    constructor _,×'_
    field
      a : A
      b : B

  record _×p_ (A : Prop ℓ) (B : A → Prop ℓ') : Prop (ℓ ⊔ ℓ') where
    constructor _,×p_
    field
      a : A
      b : B a

  record _×''_ (A : Set ℓ) (B : A → Prop ℓ') : Set (ℓ ⊔ ℓ') where
    constructor _,×''_
    field
      a : A
      b : B a

  record _×ᵈ_ (A : Set ℓ) (B : A → Set ℓ') : Set (ℓ ⊔ ℓ') where
    constructor _,×ᵈ_
    field
      a : A
      b : B a


  proj×₁ : {ℓ ℓ' : Level}{A : Set ℓ}{B : Set ℓ'} → (A × B) → A
  proj×₁ p = _×_.a p
  proj×₂ : {ℓ ℓ' : Level}{A : Set ℓ}{B : Set ℓ'} → (A × B) → B
  proj×₂ p = _×_.b p

  proj×'₁ : {ℓ ℓ' : Level}{A : Set ℓ}{B : Prop ℓ'} → (A ×' B) → A
  proj×'₁ p = _×'_.a p
  proj×'₂ : {ℓ ℓ' : Level}{A : Set ℓ}{B : Prop ℓ'} → (A ×' B) → B
  proj×'₂ p = _×'_.b p

  proj×p₁ : {ℓ ℓ' : Level}{A : Prop ℓ}{B : A → Prop ℓ'} → (A ×p B) → A
  proj×p₁ p = _×p_.a p
  proj×p₂ : {ℓ ℓ' : Level}{A : Prop ℓ}{B : A → Prop ℓ'} → (p : A ×p B) → B
(proj×p₁ p)
  proj×p₂ p = _×p_.b p

  proj×''₁ : {ℓ ℓ' : Level}{A : Set ℓ}{B : A → Prop ℓ'} → (A ×'' B) → A
  proj×''₁ p = _×''_.a p
  proj×''₂ : {ℓ ℓ' : Level}{A : Set ℓ}{B : A → Prop ℓ'} → (p : A ×'' B) → B
(proj×''₁ p)
  proj×''₂ p = _×''_.b p

  proj×ᵈ₁ : {ℓ ℓ' : Level}{A : Set ℓ}{B : A → Set ℓ'} → (A ×ᵈ B) → A
  proj×ᵈ₁ p = _×ᵈ_.a p
  proj×ᵈ₂ : {ℓ ℓ' : Level}{A : Set ℓ}{B : A → Set ℓ'} → (p : A ×ᵈ B) → (B (proj×ᵈ₁
p))
  proj×ᵈ₂ p = _×ᵈ_.b p


  ×≡ : {A : Set ℓ}{B : Set ℓ'}{a a' : A}{b b' : B} →  a ≡ a' → b ≡ b' → a ,× b ≡
a' ,× b'
  ×≡ refl refl = refl

  ×ᵈ≡ : {A : Set ℓ}{B : A → Set ℓ'}{a a' : A}{b : B a}{b' : B a'} →  (eq : a ≡
a') → subst B eq b ≡ b' → a ,×ᵈ b ≡ a' ,×ᵈ b'
  ×ᵈ≡ {B = B} {a = a}{b = b} refl refl = cong₂' _,×ᵈ_ refl refl
```

```
-- The Agda primitives (preloaded).

{-# OPTIONS --cubical-compatible --no-import-sorts #-}

module Agda.Primitive where

------------------------------------------------------------------------
-- Universe levels
------------------------------------------------------------------------

infixl 6 _⊔_

{-# BUILTIN TYPE Set #-}
{-# BUILTIN PROP Prop #-}
{-# BUILTIN SETOMEGA Setω #-}
{-# BUILTIN STRICTSET      SSet  #-}
{-# BUILTIN STRICTSETOMEGA SSetω #-}

-- Level is the first thing we need to define.
-- The other postulates can only be checked if built-in Level is known.

postulate
  Level : Set

-- MAlonzo compiles Level to (). This should be safe, because it is
-- not possible to pattern match on levels.

{-# BUILTIN LEVEL Level #-}

postulate
  lzero : Level
  lsuc  : (ℓ : Level) → Level
  _⊔_   : (ℓ₁ ℓ₂ : Level) → Level

{-# BUILTIN LEVELZERO lzero #-}
{-# BUILTIN LEVELSUC  lsuc  #-}
{-# BUILTIN LEVELMAX  _⊔_   #-}
{-# OPTIONS --prop --rewriting #-}

module ListUtil where


  infixr 5 _∷_
  data List : (T : Set₀) → Set where
    [] : {T : Set₀} → List T
    _∷_ : {T : Set₀} → T → List T → List T

  {-# BUILTIN LIST List #-}

  private
    variable
      T : Set₀
      L : List T
      L' : List T
      L'' : List T
      A : T
      B : T


  -- Definition of list appartenance
  -- The definition uses reflexivity and never any kind of equality
  infix 3 _∈_
  data _∈_ : T → List T → Prop where
    zero∈ : A ∈ A ∷ L
    next∈ : A ∈ L → A ∈ B ∷ L

  {- RELATIONS BETWEEN LISTS -}

  -- We have the following relations
  --         ↗ ⊂⁰ ↘
  -- ⊆ → ⊂ → ⊂⁺ → ∈*
  infix 4 _⊆_ _⊂_ _⊂⁺_ _⊂⁰_ _∈*_
  {- ⊆ : We can remove elements but only from the head of the list -}
  -- Similar definition : {L L' : List T} → L ⊆ L' ++ L
  data _⊆_ : List T → List T → Prop where
    zero⊆ : L ⊆ L
    next⊆ : L ⊆ L' → L ⊆ (A ∷ L')

  -- One useful lemma
  retro⊆ : {L L' : List T} → {A : T} → (A ∷ L) ⊆ L' → L ⊆ L'
  retro⊆ {L' = []} () -- Impossible to have «A∷L ⊆ []»
  retro⊆ {L' = B ∷ L'} zero⊆ = next⊆ zero⊆
  retro⊆ {L' = B ∷ L'} (next⊆ h) = next⊆ (retro⊆ h)

  refl⊆ : L ⊆ L
  refl⊆ = zero⊆

  tran⊆ : L ⊆ L' → L' ⊆ L'' → L ⊆ L''
  tran⊆ zero⊆ h2 = h2
  tran⊆ (next⊆ h1) h2 = tran⊆ h1 (retro⊆ h2)

  {- ⊂ : We can remove elements anywhere on the list, no duplicates, no
reordering -}
  data _⊂_ : List T → List T → Prop where
    zero⊂ : [] ⊂ L
    both⊂ : L ⊂ L' → (A ∷ L) ⊂ (A ∷ L')
    next⊂ : L ⊂ L' → L ⊂ (A ∷ L')

  ⊆→⊂ : L ⊆ L' → L ⊂ L'
  refl⊂ : L ⊂ L
  ⊆→⊂ {L = []} h = zero⊂
  ⊆→⊂ {L = x ∷ L} zero⊆ = both⊂ (refl⊂)
  ⊆→⊂ {L = x ∷ L} (next⊆ h) = next⊂ (⊆→⊂ h)
  refl⊂ = ⊆→⊂ refl⊆

  tran⊂ : L ⊂ L' → L' ⊂ L'' → L ⊂ L''
  tran⊂ zero⊂ h2 = zero⊂
  tran⊂ (both⊂ h1) (both⊂ h2) = both⊂ (tran⊂ h1 h2)
  tran⊂ (both⊂ h1) (next⊂ h2) = next⊂ (tran⊂ (both⊂ h1) h2)
  tran⊂ (next⊂ h1) (both⊂ h2) = next⊂ (tran⊂ h1 h2)
  tran⊂ (next⊂ h1) (next⊂ h2) = next⊂ (tran⊂ (next⊂ h1) h2)

  {- ⊂⁰ : We can remove elements and reorder the list, as long as we don't
duplicate the elements -}
  -----> We do not have unicity of derivation ([A,A] ⊂⁰ [A,A] can be prove by
identity or by swapping its two elements
  --> We could do with some counting function, but ... it would not be nice,
would it ?
  data _⊂⁰_ : List T → List T → Prop where
    zero⊂⁰ : _⊂⁰_ {T} [] []
    next⊂⁰ : L ⊂⁰ L' → L ⊂⁰ A ∷ L'
    both⊂⁰ : L ⊂⁰ L' → A ∷ L ⊂⁰ A ∷ L'
    swap⊂⁰ : L ⊂⁰ A ∷ B ∷ L' → L ⊂⁰ B ∷ A ∷ L'
    error  : L ⊂⁰ L'
    -- TODOTODOTODOTODO Fix this definition
  {- ⊂⁺ : We can remove and duplicate elements, as long as we don't change the
order -}
  data _⊂⁺_ : List T → List T → Prop where
    zero⊂⁺ : _⊂⁺_ {T} [] []
    next⊂⁺ : L ⊂⁺ L' → L ⊂⁺ A ∷ L'
    dup⊂⁺ : L ⊂⁺ A ∷ L' → A ∷ L ⊂⁺ A ∷ L'

  ⊂→⊂⁺ : L ⊂ L' → L ⊂⁺ L'
  ⊂→⊂⁺ {L' = []} zero⊂ = zero⊂⁺
  ⊂→⊂⁺ {L' = x ∷ L'} zero⊂ = next⊂⁺ (⊂→⊂⁺ zero⊂)
  ⊂→⊂⁺ (both⊂ h) = dup⊂⁺ (next⊂⁺ (⊂→⊂⁺ h))
  ⊂→⊂⁺ (next⊂ h) = next⊂⁺ (⊂→⊂⁺ h)
  refl⊂⁺ : L ⊂⁺ L
  refl⊂⁺ = ⊂→⊂⁺ refl⊂
  tran⊂⁺ : L ⊂⁺ L' → L' ⊂⁺ L'' → L ⊂⁺ L''
  tran⊂⁺ zero⊂⁺ zero⊂⁺ = zero⊂⁺
  tran⊂⁺ zero⊂⁺ (next⊂⁺ h2) = next⊂⁺ h2
  tran⊂⁺ (next⊂⁺ h1) (next⊂⁺ h2) = next⊂⁺ (tran⊂⁺ (next⊂⁺ h1) h2)
  tran⊂⁺ (next⊂⁺ h1) (dup⊂⁺ h2) = tran⊂⁺ h1 h2
  tran⊂⁺ (dup⊂⁺ h1) (next⊂⁺ h2) = next⊂⁺ (tran⊂⁺ (dup⊂⁺ h1) h2)
  tran⊂⁺ (dup⊂⁺ h1) (dup⊂⁺ h2) = dup⊂⁺ (tran⊂⁺ h1 (dup⊂⁺ h2))

  retro⊂⁺ : A ∷ L ⊂⁺ L' → L ⊂⁺ L'
  retro⊂⁺ (next⊂⁺ h) = next⊂⁺ (retro⊂⁺ h)
  retro⊂⁺ (dup⊂⁺ h) = h

  {- ∈* : We can remove or duplicate elements and we can change their order -}
  -- The weakest of all relations on lists
  -- L ∈* L' if all elements of L exists in L' (no consideration for order nor
duplication)
  data _∈*_ : List T → List T → Prop where
    zero∈* : [] ∈* L
    next∈* : A ∈ L → L' ∈* L → (A ∷ L') ∈* L

  -- Founding principle
  mon∈∈* : A ∈ L → L ∈* L' → A ∈ L'
  mon∈∈* zero∈ (next∈* x hl) = x
  mon∈∈* (next∈ ha) (next∈* x hl) = mon∈∈* ha hl
  -- We show that the relation is reflexive and is implied by ⊆
  refl∈* : L ∈* L
  ⊂⁺→∈* : L ⊂⁺ L' → L ∈* L'
  refl∈* {L = []} = zero∈*
  refl∈* {L = x ∷ L} = next∈* zero∈ (⊂⁺→∈* (next⊂⁺ refl⊂⁺))
  ⊂⁺→∈* zero⊂⁺ = refl∈*
  ⊂⁺→∈* {L = []} (next⊂⁺ h) = zero∈*
  ⊂⁺→∈* {L = x ∷ L} (next⊂⁺ h) = next∈* (next∈ (mon∈∈* zero∈ (⊂⁺→∈* h)))
(⊂⁺→∈* (retro⊂⁺ (next⊂⁺ h)))
  ⊂⁺→∈* (dup⊂⁺ h) = next∈* zero∈ (⊂⁺→∈* h)

  -- Allows to grow ∈* to the right
  right∈* : L ∈* L' → L ∈* (A ∷ L')
  right∈* zero∈* = zero∈*
  right∈* (next∈* x h) = next∈* (next∈ x) (right∈* h)

  both∈* : L ∈* L' → (A ∷ L) ∈* (A ∷ L')
  both∈* zero∈* = next∈* zero∈ zero∈*
  both∈* (next∈* x h) = next∈* zero∈ (next∈* (next∈ x) (right∈* h))

  tran∈* : L ∈* L' → L' ∈* L'' → L ∈* L''
  tran∈* {L = []} = λ x x₁ → zero∈*
  tran∈* {L = x ∷ L} (next∈* x₁ h1) h2 = next∈* (mon∈∈* x₁ h2) (tran∈* h1 h2)

  ⊆→∈* : L ⊆ L' → L ∈* L'
  ⊆→∈* h = ⊂⁺→∈* (⊂→⊂⁺ (⊆→⊂ h))

  open import PropUtil using (Nat; zero; succ)
  open import Agda.Primitive
  variable
    ℓ : Level

  data Array (T : Set ℓ) : Nat → Set ℓ where
    zero : Array T zero
    next : {n : Nat} → T → Array T n → Array T (succ n)

  map : {T U : Set ℓ} → (T → U) → {n : Nat} →  Array T n → Array U n
  map f zero = zero
  map f (next t a) = next (f t) (map f a)
```

```
\begin{code}
{-# OPTIONS --prop --rewriting #-}

open import PropUtil

module ZOL2 where

  open import Agda.Primitive
  open import ListUtil

  variable
    ℓ¹ ℓ² ℓ³ ℓ⁴ : Level
    ℓ¹' ℓ²' ℓ³' ℓ⁴' : Level

  record ZOL : Set (lsuc (ℓ¹ ⊔ ℓ² ⊔ ℓ³ ⊔ ℓ⁴)) where
    infixr 10 _∘_
    field

      --# We first make the base category with its terminal object
      Con : Set ℓ¹
      Sub : Con → Con → Prop ℓ² -- It makes a posetal category
      _∘_ : {Γ Δ Ξ : Con} → Sub Δ Ξ → Sub Γ Δ → Sub Γ Ξ
      id : {Γ : Con} → Sub Γ Γ
      ◇ : Con -- The terminal object of the category
      ε : {Γ : Con} → Sub Γ ◇ -- The morphism from any object to the terminal

      --# Categorical equations don't need to be stated as the category is
*posetal*
      --∘-ass : {Γ Δ Ξ Ψ : Con}{α : Sub Γ Δ}{β : Sub Δ Ξ}{γ : Sub Ξ Ψ}
      --idl : {Γ Δ : Con} {σ : Sub Γ Δ} →  (id {Δ}) ∘ σ ≡ σ
      --idr : {Γ Δ : Con} {σ : Sub Γ Δ} →  σ ∘ (id {Γ}) ≡ σ
      --  → (γ ∘ β) ∘ α ≡ γ ∘ (β ∘ α)
      --ε-u : {Γ : Con} → {σ : Sub Γ ◇} → σ ≡ ε {Γ}

      --# Functor Con → Set called For
      For : Con → Set ℓ³
      _[_]f : {Γ Δ : Con} → For Γ → Sub Δ Γ → For Δ -- Action on morphisms
      []f-id : {Γ : Con} → {F : For Γ} → F [ id {Γ} ]f ≡ F
      []f-∘ : {Γ Δ Ξ : Con} → {α : Sub Ξ Δ} → {β : Sub Δ Γ} → {F : For Γ}
        → F [ β ∘ α ]f ≡ (F [ β ]f) [ α ]f

      --# Functor Con × For → Prop called Pf or ⊢
      Pf : (Γ : Con) → For Γ → Prop ℓ⁴
      -- Action on morphisms
      _[_]p : {Γ Δ : Con} → {F : For Γ} → Pf Γ F → (σ : Sub Δ Γ) → Pf Δ (F [ σ
]f)
      --# Equalities below are useless because Pf Γ F is in prop
      -- []p-id : {Γ : Con} → {F : For Γ} → {prf : Pf Γ F}
      --  → prf [ id {Γ} ]p ≡ prf
      -- []p-∘ : {Γ Δ Ξ : Con}{α : Sub Ξ Δ}{β : Sub Δ Γ}{F : For Γ}{prf : Pf Γ F}
      --  → prf [ α ∘ β ]p ≡ (prf [ β ]p) [ α ]p

      --# → Prop⁺
      _▹p_ : (Γ : Con) → For Γ → Con
      πp¹ : {Γ Δ : Con}{F : For Γ} → Sub Δ (Γ ▹p F) → Sub Δ Γ
      πp² : {Γ Δ : Con}{F : For Γ} → (σ : Sub Δ (Γ ▹p F)) → Pf Δ (F [ πp¹ σ ]f)
      _,p_ : {Γ Δ : Con}{F : For Γ} → (σ : Sub Δ Γ) → Pf Δ (F [ σ ]f) → Sub Δ (Γ
▹p F)
      --# Equality below are useless because Pf and Sub are in Prop
      --,p∘πp : {Γ Δ : Con}{F : For Γ}{σ : Sub Δ (Γ ▹p F)} → (πp¹ σ) ,p (πp² σ) ≡ σ
      --πp¹∘,p : {Γ Δ : Con}{σ : Sub Δ Γ}{F : For Γ}{prf : Pf Δ (F [ σ ]f)}
      --  → πp¹ (σ ,p prf) ≡ σ
      -- πp²∘,p : {Γ Δ : Con}{σ : Sub Δ Γ}{F : For Γ}{prf : Pf Δ (F [ σ ]f)}
      -- → πₚ² (σ ,ₚ prf) ≡ prf
      --,ₚ∘ : {Γ Δ Ξ : Con}{σ : Sub Γ Ξ}{δ : Sub Δ Γ}{F : For Ξ}{prf : Pf Γ (F [
σ ]f)}
      --  → (σ ,ₚ prf) ∘ δ ≡ (σ ∘ δ) ,ₚ (substP (Pf Δ) (≡sym []f-∘) (prf [ δ ]p))

      --#
      {-- FORMULAE CONSTRUCTORS --}

      --# ι formula
      ι : {Γ : Con} → For Γ
      []f-ι : {Γ Δ : Con} {σ : Sub Δ Γ}→ ι [ σ ]f ≡ ι

      --# Implication
      _⇒_ : {Γ : Con} → For Γ → For Γ → For Γ
      []f-⇒ : {Γ Δ : Con} → {F G : For Γ} → {σ : Sub Δ Γ}
        → (F ⇒ G) [ σ ]f ≡ (F [ σ ]f) ⇒ (G [ σ ]f)

      --#
      {-- PROOFS CONSTRUCTORS --}
      -- Again, we don't have to write the _[_]p equalities as Proofs are in Prop

      --# Lam & App
      lam : {Γ : Con}{F G : For Γ} → Pf (Γ ▷ₚ F) (G [ πₚ¹ id ]f) → Pf Γ (F ⇒ G)
      app : {Γ : Con}{F G : For Γ} → Pf Γ (F ⇒ G) → Pf Γ F → Pf Γ G

      --#

  record Mapping (S : ZOL {ℓ¹} {ℓ²} {ℓ³} {ℓ⁴}) (D : ZOL {ℓ¹'} {ℓ²'} {ℓ³'} {ℓ⁴'}) :
Set (lsuc (ℓ¹ ⊔ ℓ² ⊔ ℓ³ ⊔ ℓ⁴ ⊔ ℓ¹' ⊔ ℓ²' ⊔ ℓ³' ⊔ ℓ⁴')) where
    field

      --#
      mCon : (ZOL.Con S) → (ZOL.Con D)
      mSub : {Δ : (ZOL.Con S)}{Γ : (ZOL.Con S)} →
        (ZOL.Sub S Δ Γ) → (ZOL.Sub D (mCon Δ) (mCon Γ))
      mFor : {Γ : (ZOL.Con S)} → (ZOL.For S Γ) → (ZOL.For D (mCon Γ))
      mPf : {Γ : (ZOL.Con S)} {A : ZOL.For S Γ}
        → ZOL.Pf S Γ A → ZOL.Pf D (mCon Γ) (mFor A)
      --#

  record Morphism (S : ZOL {ℓ¹} {ℓ²} {ℓ³} {ℓ⁴}) (D : ZOL {ℓ¹'} {ℓ²'} {ℓ³'} {ℓ⁴'})
: Set (lsuc (ℓ¹ ⊔ ℓ² ⊔ ℓ³ ⊔ ℓ⁴ ⊔ ℓ¹' ⊔ ℓ²' ⊔ ℓ³' ⊔ ℓ⁴')) where
    field m : Mapping S D
    mCon = Mapping.mCon m
    mSub = Mapping.mSub m
    mFor = Mapping.mFor m
    mPf  = Mapping.mPf m
    field
      --#
      e◇ : mCon (ZOL.◇ S) ≡ ZOL.◇ D
      e[]f : {Γ Δ : ZOL.Con S}{A : ZOL.For S Γ}{σ : ZOL.Sub S Δ Γ} →
        mFor (ZOL._[_]f S A σ) ≡ ZOL._[_]f D (mFor A) (mSub σ)
      e▷ₚ : {Γ : ZOL.Con S}{A : ZOL.For S Γ} →
        mCon (ZOL._▷ₚ_ S Γ A) ≡ ZOL._▷ₚ_ D (mCon Γ) (mFor A)
      eι : {Γ : ZOL.Con S} → mFor (ZOL.ι S {Γ}) ≡ ZOL.ι D {mCon Γ}
      e⇒ : {Γ : ZOL.Con S}{A B : ZOL.For S Γ} →
        mFor (ZOL._⇒_ S A B) ≡ ZOL._⇒_ D (mFor A) (mFor B)
      -- No equation needed for lam, app, ∀i, ∀e as their output are in prop
      --#

  record TrNat {S : ZOL {ℓ¹} {ℓ²} {ℓ³} {ℓ⁴}} {D : ZOL {ℓ¹'} {ℓ²'} {ℓ³'} {ℓ⁴'}} (a
: Mapping S D) (b : Mapping S D) : Set (lsuc (ℓ¹ ⊔ ℓ² ⊔ ℓ³ ⊔ ℓ⁴ ⊔ ℓ¹' ⊔ ℓ²' ⊔
ℓ³' ⊔ ℓ⁴')) where
    field
```

```
      f : (Γ : ZOL.Con S) → ZOL.Sub D (Mapping.mCon a Γ) (Mapping.mCon b Γ)
      -- Unneeded because Sub are in prop
      --eq : (Γ Δ : ZOL.Con S)(σ : ZOL.Sub S Γ Δ) → (ZOL._∘_ D (f Δ)
(Mapping.mSub a σ)) ≡ (ZOL._∘_ D (Mapping.mSub b σ) (f Γ))

  _∘TrNat_ : {S : ZOL {ℓ¹} {ℓ²} {ℓ³} {ℓ⁴}}{D : ZOL {ℓ¹'} {ℓ²'} {ℓ³'} {ℓ⁴'}}{a b c
: Mapping S D} → TrNat a b → TrNat b c → TrNat a c
  _∘TrNat_ {D = D} α β = record { f = λ Γ → ZOL._∘_ D (TrNat.f β Γ) (TrNat.f α Γ)
}

  idTrNat : {S : ZOL {ℓ¹} {ℓ²} {ℓ³} {ℓ⁴}}{D : ZOL {ℓ¹'} {ℓ²'} {ℓ³'} {ℓ⁴'}}{a :
Mapping S D} → TrNat a a
  idTrNat {D = D} = record { f = λ Γ → ZOL.id D }

\end{code}
```

```
\begin{code}
{-# OPTIONS --prop --rewriting #-}

open import PropUtil

module ZOLInitial where

  open import ZOL2
  open import Agda.Primitive
  open import ListUtil

  --#
  data For : Set where
    ι : For
    _⇒_ : For → For → For

  data Con : Set where
    ◇ : Con
    _▹p_ : Con → For → Con

  --#
  variable
    Γ Δ Ξ : Con
    A B : For

  --#
  data PfVar : Con → For → Prop where
    pvzero : PfVar (Γ ▹p A) A
    pvnext : PfVar Γ A → PfVar (Γ ▹p B) A
  data Pf : Con → For → Prop where
    var : PfVar Γ A → Pf Γ A
    lam : Pf (Γ ▹p A) B → Pf Γ (A ⇒ B)
    app : Pf Γ (A ⇒ B) → Pf Γ A → Pf Γ B

  --#
  data Ren : Con → Con → Prop where
    ε : Ren Γ ◇
    _,pR_ : Ren Δ Γ → PfVar Δ A → Ren Δ (Γ ▹p A)

  --#
  rightR : Ren Δ Γ → Ren (Δ ▹p A) Γ
  rightR ε = ε
  rightR (σ ,pR pv) = (rightR σ) ,pR (pvnext pv)

  idR : Ren Γ Γ
  idR {◇} = ε
  idR {Γ ▹p A} = (rightR (idR {Γ})) ,pR pvzero

  --#
  data Sub : Con → Con → Prop where
    ε : Sub Γ ◇
    _,p_ : Sub Δ Γ → Pf Δ A → Sub Δ (Γ ▹p A)

  --#
  πp¹ : Sub Δ (Γ ▹p A) → Sub Δ Γ
  πp² : Sub Δ (Γ ▹p A) → Pf Δ A
  πp¹ (σ ,p pf) = σ
  πp² (σ ,p pf) = pf

  --#
  SubR : Ren Γ Δ → Sub Γ Δ
  SubR ε = ε
  SubR (σ ,pR pv) = SubR σ ,p var pv
```

```
id : Sub Γ Γ
id = SubR idR

--#
_[_]pvr : PfVar Γ A → Ren Δ Γ → PfVar Δ A
pvzero [ _ ,ₚR pv ]pvr = pv
pvnext pv [ σ ,ₚR _ ]pvr = pv [ σ ]pvr
_[_]pr : Pf Γ A → Ren Δ Γ → Pf Δ A
var pv [ σ ]pr = var (pv [ σ ]pvr)
lam pf [ σ ]pr = lam (pf [ (rightR σ) ,ₚR pvzero ]pr)
app pf pf' [ σ ]pr = app (pf [ σ ]pr) (pf' [ σ ]pr)

wkSub : Sub Δ Γ → Sub (Δ ▷ₚ A) Γ
wkSub ε = ε
wkSub (σ ,ₚ pf) = (wkSub σ) ,ₚ (pf [ rightR idR ]pr)

--#
_[_]p : Pf Γ A → Sub Δ Γ → Pf Δ A
var pvzero [ _ ,ₚ pf ]p = pf
var (pvnext pv) [ σ ,ₚ _ ]p = var pv [ σ ]p
lam pf [ σ ]p = lam (pf [ wkSub σ ,ₚ var pvzero ]p)
app pf pf' [ σ ]p = app (pf [ σ ]p) (pf' [ σ ]p)

--#
_∘_ : {Γ Δ Ξ : Con} → Sub Δ Ξ → Sub Γ Δ → Sub Γ Ξ
ε ∘ β = ε
(α ,ₚ pf) ∘ β = (α ∘ β) ,ₚ (pf [ β ]p)

--#



zol : ZOL
zol = record
        { Con = Con
        ; Sub = Sub
        ; _∘_ = _∘_
        ; id = id
        ; ◇ = ◇
        ; ε = ε
        ; For = λ Γ → For
        ; _[_]f = λ A σ → A
        ; []f-id = refl
        ; []f-∘ = refl
        ; Pf = Pf
        ; _[_]p = _[_]p
        ; _▷ₚ_ = _▷ₚ_
        ; πₚ¹ = πₚ¹
        ; πₚ² = πₚ²
        ; _,ₚ_ = _,ₚ_
        ; ι = ι
        ; []f-ι = refl
        ; _⇒_ = _⇒_
        ; []f-⇒ = refl
        ; lam = lam
        ; app = app
        }

module InitialMorphism (M : ZOL {ℓ¹} {ℓ²} {ℓ³} {ℓ⁴}) where

    --#
    mCon : Con → (ZOL.Con M)
    mFor : {Γ : Con} → For → (ZOL.For M (mCon Γ))
```

```
      mCon ◇ = ZOL.◇ M
      mCon (Γ ▷p A) = ZOL._▷p_ M (mCon Γ) (mFor {Γ} A)
      mFor {Γ} ι = ZOL.ι M
      mFor {Γ} (A ⇒ B) = ZOL._⇒_ M (mFor {Γ} A) (mFor {Γ} B)

      --#

      mSub : {Δ : Con}{Γ : Con} → Sub Δ Γ → (ZOL.Sub M (mCon Δ) (mCon Γ))
      mPf : {Γ : Con} {A : For} → Pf Γ A → ZOL.Pf M (mCon Γ) (mFor {Γ} A)

      e[]f⁰ : {Γ : Con}{A B : For} → mFor {Γ ▷p B} A ≡ ZOL._[_]f M (mFor {Γ} A)
(ZOL.πp¹ M (ZOL.id M))
      e[]f⁰ {A = ι} = ≡sym (ZOL.[]f-ι M)
      e[]f⁰ {A = A ⇒ B} = ≡sym (≡tran (ZOL.[]f-⇒ M) (cong₂ (ZOL._⇒_ M) (≡sym
(e[]f⁰ {A = A})) (≡sym (e[]f⁰ {A = B}))))

      e[]f : {Γ Δ : Con}{A : For}{σ : Sub Δ Γ} → mFor {Δ} A ≡ ZOL._[_]f M (mFor
{Γ} A) (mSub σ)
      e[]f {A = ι} = ≡sym (ZOL.[]f-ι M)
      e[]f {Γ} {Δ} {A = A ⇒ B} {σ} = ≡sym (≡tran (ZOL.[]f-⇒ M) (cong₂ (ZOL._⇒_
M) (≡sym (e[]f {A = A}{σ})) (≡sym (e[]f {A = B}{σ}))))



      mPf {A = A} (var (pvzero {Γ})) = substP (ZOL.Pf M _) (≡sym (e[]f⁰ {Γ} {A}
{A})) (ZOL.πp² M (ZOL.id M))
      mPf {A = A} (var (pvnext {Γ} {B = B} pv)) = substP (ZOL.Pf M _) (≡sym
(e[]f⁰ {Γ} {A} {B})) (ZOL._[_]p M (mPf (var pv)) (ZOL.πp¹ M (ZOL.id M)))
      mPf {Γ} (lam {A = A} {B} pf) = ZOL.lam M (substP (ZOL.Pf M _) (e[]f⁰ {Γ}
{B} {A}) (mPf pf))
      mPf (app pf pf') = ZOL.app M (mPf pf) (mPf pf')

      mSub ε = ZOL.ε M
      mSub (_,p_ {Δ} {Γ = Γ} {A} σ pf) = ZOL._,p_ M (mSub σ) (substP (ZOL.Pf M _)
(e[]f {Γ} {Δ} {A} {σ}) (mPf pf))

      mor : Morphism zol M
      mor = record {
        m = record {
          mCon = mCon
          ; mSub = mSub
          ; mFor = λ {Γ} A → mFor {Γ} A
          ; mPf = mPf
        }
        ; e◇ = refl
        ; e[]f = λ {Γ}{Δ}{A}{σ} → e[]f {A = A} {σ}
        ; e▷p = refl
        ; eι = refl
        ; e⇒ = refl
        }

  module InitialMorphismUniqueness {M : ZOL {ℓ¹} {ℓ²} {ℓ³} {ℓ⁴}} {m : Morphism
zol M} where

    open InitialMorphism M
    mCon≡ : {Γ : Con} → mCon Γ ≡ (Morphism.mCon m Γ)
    mFor≡ : {Γ : Con} {A : For} → mFor {Γ} A ≡ subst (ZOL.For M) (≡sym mCon≡)
(Morphism.mFor m {Γ} A)

    mCon≡ {◇} = ≡sym (Morphism.e◇ m)
    mCon≡ {Γ ▷p A} = ≡sym (≡tran (Morphism.e▷p m) (cong₂' (ZOL._▷p_ M) (≡sym mCon≡)
(≡sym mFor≡)))
    mFor≡ {Γ} {ι} = ≡sym (≡tran
      (cong (subst (ZOL.For M) (≡sym mCon≡)) (Morphism.eι m))
      (substpoly {f = λ {Ξ} → ZOL.ι M {Ξ}} {eq = ≡sym mCon≡})
      )
    mFor≡ {Γ} {A ⇒ B} = ≡sym (≡tran²
      (cong (subst (ZOL.For M) (≡sym mCon≡)) (Morphism.e⇒ m))
      (substppoly {eq = ≡sym (mCon≡ {Γ})} {f = λ {ξ} X Y → ZOL._⇒_ M {ξ} X Y})
      (cong₂ (ZOL._⇒_ M) (≡sym (mFor≡ {Γ} {A})) (≡sym (mFor≡ {Γ} {B}))))

    -- Those two lines are not needed as those sorts are in Prop
    --mSub≡ : {Δ : Con}{Γ : Con}{σ : Sub Δ Γ} → mSub {Δ} {Γ} σ ≡ Morphism.mSub m
{Δ} {Γ} σ
    --mPf≡ : {Γ : Con} {A : For}{p : Pf Γ A} → mPf {Γ} {A} p ≡ Morphism.mPf m {Γ}
{A} p

\end{code}
```

```
\begin{code}
{-# OPTIONS --prop --rewriting #-}

open import PropUtil

module ZOLCompleteness where

  open import Agda.Primitive
  open import ZOL2
  open import ListUtil

  record Kripke : Set (lsuc (ℓ¹)) where
    field
      World : Set ℓ¹
      _-w->_ : World → World → Prop ℓ¹ -- arrows
      -w->id : {w : World} → w -w-> w -- id arrow
      _∘-w->_ : {w w' w'' : World} → w -w-> w' → w' -w-> w'' → w -w-> w'' -- arrow composition

      I : World → Prop ℓ¹
      I≤ : {w w' : World} → w -w-> w' → I w → I w'

    infixr 10 _∘_

    Con : Set (lsuc ℓ¹)
    Con = (World → Prop ℓ¹) ×'' (λ Γ → {w w' : World} → (w -w-> w')→ Γ w → Γ w')
    Sub : Con → Con → Prop ℓ¹
    Sub Δ Γ = (w : World) → (proj×''₁ Δ) w → (proj×''₁ Γ) w
    _∘_ : {Γ Δ Ξ : Con} → Sub Δ Ξ → Sub Γ Δ → Sub Γ Ξ
    α ∘ β = λ w γ → α w (β w γ)
    id : {Γ : Con} → Sub Γ Γ
    id = λ w γ → γ
    ◇ : Con -- The initial object of the category
    ◇ = (λ w → ⊤) ,×'' (λ _ _ → tt)
    ε : {Γ : Con} → Sub Γ ◇ -- The morphism from the initial to any object
    ε w Γ = tt

    -- Functor Con → Set called For
    For : Set (lsuc ℓ¹)
    For = (World → Prop ℓ¹) ×'' (λ F → {w w' : World} → (w -w-> w')→ F w → F w')

    -- Proofs
    Pf : (Γ : Con) → For → Prop ℓ¹
    Pf Γ F = ∀ w (γ : (proj×''₁ Γ) w) →  (proj×''₁ F) w
    _[_]p : {Γ Δ : Con} → {F : For} → Pf Γ F → (σ : Sub Δ Γ) → Pf Δ F -- The functor's action on morphisms
    prf [ σ ]p = λ w → λ γ → prf w (σ w γ)
    -- Equalities below are useless because Γ ⊢ F is in prop
    -- []p-id : {Γ : Con} → {F : For Γ} → {prf : Γ ⊢ F} → prf [ id {Γ} ]p ≡ prf
    -- []p-∘ : {Γ Δ Ξ : Con} → {α : Sub Ξ Δ} → {β : Sub Δ Γ} → {F : For Γ} → {prf : Γ ⊢ F} → prf [ α ∘ β ]p
≡ (prf [ β ]p) [ α ]p


    -- → Prop⁺
    _▷p_ : (Γ : Con) → For → Con
    Γ ▷p F = (λ w → (proj×''₁ Γ) w ∧ (proj×''₁ F) w) ,×'' λ s γ → ⟨ proj×''₂ Γ s (proj₁ γ) , proj×''₂ F s
(proj₂ γ) ⟩

    πp¹ : {Γ Δ : Con} → {F : For} → Sub Δ (Γ ▷p F) → Sub Δ Γ
    πp¹ σ w δ = proj₁ (σ w δ)
    πp² : {Γ Δ : Con} → {F : For} → (σ : Sub Δ (Γ ▷p F)) → Pf Δ F
    πp² σ w δ = proj₂ (σ w δ)
    _,p_ : {Γ Δ : Con} → {F : For} → (σ : Sub Δ Γ) → Pf Δ F → Sub Δ (Γ ▷p F)
    (σ ,p pf) w δ = ⟨ (σ w δ) , pf w δ ⟩


    -- Base formula
    ι : For
    ι = (λ w → I w) ,×'' λ s f → I≤ s f

    -- Implication
    _⇒_ : For → For → For
    (F ⇒ G) = (λ w → {w' : World} → (s : w -w-> w') → ((proj×''₁ F) w') → ((proj×''₁ G) w')) ,×'' λ s f s'
f' → f (s ∘-w-> s') f'

    -- Lam & App
    lam : {Γ : Con} → {F : For} → {G : For} → Pf (Γ ▷p F) G → Pf Γ (F ⇒ G)
    lam {Γ} pf w γ {w'} s x = pf w' ⟨ proj×''₂ Γ s γ , x ⟩
    --lam prf = λ w γ w' s h → prf w (γ , h)
    app : {Γ : Con} → {F G : For} → Pf Γ (F ⇒ G) → Pf Γ F → Pf Γ G
    app pf pf' w γ = pf w γ -w->id (pf' w γ)
    -- Again, we don't write the _[_]p equalities as everything is in Prop

    zol : ZOL
    zol = record
            { Con = Con
            ; Sub = Sub
            ; _∘_ = λ {Γ}{Δ}{Ξ} σ δ → _∘_ {Γ}{Δ}{Ξ} σ δ
            ; id =  λ {Γ} → id {Γ}
            ; ◇ = ◇
            ; ε = λ {Γ} → ε {Γ}
            ; For = λ Γ → For
            ; _[_]f = λ A σ → A
            ; []f-id = refl
            ; []f-∘ = refl
            ; Pf = Pf
            ; _[_]p = λ {Γ}{Δ}{F} pf σ → _[_]p {Γ}{Δ}{F} pf σ
            ; _▹p_  = _▹p_
            ; πp¹  = λ {Γ}{Δ}{F}σ → πp¹  {Γ}{Δ}{F}  σ
            ; πp²  = λ {Γ}{Δ}{F}σ → πp²  {Γ}{Δ}{F}  σ
            ; _,p_  = λ {Γ}{Δ}{F} σ pf → _,p_ {Γ}{Δ}{F}σ pf
            ; ι = ι
            ; []f-ι = refl
            ; _⇒_  = _⇒_
            ; []f-⇒ = refl
            ; lam = λ {Γ}{F}{G} pf → lam {Γ}{F}{G} pf
            ; app = λ {Γ}{F}{G} pf pf' → app {Γ}{F}{G} pf pf'
            }

  module U where

    import ZOLInitial as I

    U : Kripke
    U = record
        { World = I.Con
        ; _-w->_ = λ Γ Δ → I.Sub Δ Γ
        ; -w->id = I.id
        ; _∘-w->_  = λ σ σ' → σ I.∘ σ'
        ; I = λ Γ → I.Pf Γ I.ι
        ; I≤ = λ s pf → pf I.[ s ]p
        }

    open Kripke U

    y : Mapping I.zol zol
    y = record
      { mCon = λ Γ → (λ Δ → I.Sub Δ Γ) ,×'' λ σ δ → δ I.∘ σ
      ; mSub = λ σ Ξ δ → σ I.∘ δ
      ; mFor = λ A → (λ Ξ → I.Pf Ξ A) ,×'' λ σ pf → pf I.[ σ ]p
      ; mPf = λ pf Ξ σ → pf I.[ σ ]p
      }
    m : Morphism I.zol zol
    m = I.InitialMorphism.mor zol

    q : (Γ : I.Con) → Sub (Morphism.mCon m Γ) (Mapping.mCon y Γ)
    u : (Γ : I.Con) → Sub (Mapping.mCon y Γ) (Morphism.mCon m Γ)

    ⟦_⟧c = Morphism.mCon m
    ⟦_,_⟧f = λ A Γ →  Morphism.mFor m {Γ} A

    q⁰ : {F : I.For} → {Γ Γ₀ : I.Con} → proj×''₁ ⟦ F , Γ₀ ⟧f Γ → I.Pf Γ F
    u⁰ : {F : I.For} → {Γ Γ₀ : I.Con} → I.Pf Γ F → proj×''₁ ⟦ F , Γ₀ ⟧f Γ

    q⁰ {I.ι} {Γ} h = h
    q⁰ {A I.⇒ B} {Γ} h = I.lam (q⁰ {B} (h (I.πp¹ I.id) (u⁰ {A} (I.var I.pvzero))))
    u⁰ {I.ι} {Γ} pf = pf
    u⁰ {A I.⇒ B} {Γ} pf iq hF  = u⁰ {B} (I.app (pf I.[ iq ]p) (q⁰ hF) )


    q I.◇ w γ = I.ε
    q (Γ I.▹p A) w γ = (q Γ w (proj₁ γ) I.,p q⁰ (proj₂ γ))
    u I.◇ w σ = tt
    u (Γ I.▹p A) w σ = ⟨ (u Γ w (I.πp¹ σ)) , u⁰ (I.πp² σ) ⟩

    ηq : TrNat (Morphism.m m) y
    ηq = record { f = q }
    ηu : TrNat y (Morphism.m m)
    ηu = record { f = u }

    eq : ηu ∘TrNat ηq ≡ idTrNat
    eq = refl

    realCompleteness : {Γ Δ : I.Con} → ({Ξ : I.Con} → (proj×''₁ ⟦ Γ ⟧c) Ξ → (proj×''₁ ⟦ Δ ⟧c) Ξ )  → I.Sub Γ
Δ
    realCompleteness {Γ} {Δ} f = q Δ Γ (f {Γ} (u Γ Γ I.id))

\end{code}
```

```
\begin{code}
{-# OPTIONS --prop --rewriting #-}

open import PropUtil

module IFOL2 where

  open import Agda.Primitive
  open import ListUtil

  variable
    ℓ¹ ℓ² ℓ³ ℓ⁴ : Level
    ℓ¹' ℓ²' ℓ³' ℓ⁴' : Level

  record IFOL (TM : Set) : Set (lsuc (ℓ¹ ⊔ ℓ² ⊔ ℓ³ ⊔ ℓ⁴)) where
    infixr 10 _∘_
    field

      -- We first make the base category with its terminal object
      Con : Set ℓ¹
      Sub : Con → Con → Prop ℓ² -- It makes a category
      _∘_ : {Γ Δ Ξ : Con} → Sub Δ Ξ → Sub Γ Δ → Sub Γ Ξ
      id : {Γ : Con} → Sub Γ Γ
      ◇ : Con -- The terminal object of the category
      ε : {Γ : Con} → Sub Γ ◇ -- The morphism from any object to the terminal

      -- Functor Con → Set called For
      For : Con → Set ℓ³
      _[_]f : {Γ Δ : Con} → For Γ → Sub Δ Γ → For Δ -- Action on morphisms
      []f-id : {Γ : Con} → {F : For Γ} → F [ id {Γ} ]f ≡ F
      []f-∘ : {Γ Δ Ξ : Con} → {α : Sub Ξ Δ} → {β : Sub Δ Γ} → {F : For Γ}
        → F [ β ∘ α ]f ≡ (F [ β ]f) [ α ]f

      -- Functor Con × For → Prop called Pf or ⊢
      Pf : (Γ : Con) → For Γ → Prop ℓ⁴
      -- Action on morphisms
      _[_]p : {Γ Δ : Con} → {F : For Γ} → Pf Γ F → (σ : Sub Δ Γ) → Pf Δ (F [ σ
]f)
      -- Equalities below are useless because Γ ⊢ F is in prop
      -- []p-id : {Γ : Con} → {F : For Γ} → {prf : Γ ⊢ F}
      --   → prf [ id {Γ} ]p ≡ prf
      -- []p-∘ : {Γ Δ Ξ : Con}{α : Sub Ξ Δ}{β : Sub Δ Γ}{F : For Γ}{prf : Γ ⊢ F}
      --   → prf [ α ∘ β ]p ≡ (prf [ β ]p) [ α ]p

      -- → Prop⁺
      _▹p_ : (Γ : Con) → For Γ → Con
      πp¹ : {Γ Δ : Con}{F : For Γ} → Sub Δ (Γ ▹p F) → Sub Δ Γ
      πp² : {Γ Δ : Con}{F : For Γ} → (σ : Sub Δ (Γ ▹p F)) → Pf Δ (F [ πp¹ σ ]f)
      _,p_ : {Γ Δ : Con}{F : For Γ} → (σ : Sub Δ Γ) → Pf Δ (F [ σ ]f) → Sub Δ (Γ
▹p F)
      -- All equalities are useless because Sub and Pf are in prop


      {-- FORMULAE CONSTRUCTORS --}
      -- Formulas with relation on terms
      R : {Γ : Con} → (t u : TM) → For Γ
      R[] : {Γ Δ : Con} → {σ : Sub Δ Γ} → {t u : TM}
        → (R t u) [ σ ]f ≡ R t u

      -- Implication
      _⇒_ : {Γ : Con} → For Γ → For Γ → For Γ
      []f-⇒ : {Γ Δ : Con} → {F G : For Γ} → {σ : Sub Δ Γ}
        → (F ⇒ G) [ σ ]f ≡ (F [ σ ]f) ⇒ (G [ σ ]f)
      --# Forall
      ∀∀ : {Γ : Con} → (TM → For Γ) → For Γ
      []f-∀∀ : {Γ Δ : Con} → {F : TM → For Γ} → {σ : Sub Δ Γ}
        → (∀∀ F) [ σ ]f ≡ (∀∀ (λ t → (F t) [ σ ]f))

      --#
      {-- PROOFS CONSTRUCTORS --}
      -- Again, we don't have to write the _[_]p equalities as Proofs are in Prop

      -- Lam & App
      lam : {Γ : Con}{F G : For Γ} → Pf (Γ ▷p F) (G [ πp¹ id ]f) → Pf Γ (F ⇒ G)
      app : {Γ : Con}{F G : For Γ} → Pf Γ (F ⇒ G) → Pf Γ F → Pf Γ G

      --# ∀i and ∀e
      ∀i : {Γ : Con}{A : TM → For Γ} → ((t : TM) → Pf Γ (A t)) → Pf Γ (∀∀ A)
      ∀e : {Γ : Con}{A : TM → For Γ} → Pf Γ (∀∀ A) → (t : TM) → Pf Γ (A t)
      --#

  record Mapping (TM : Set) (S : IFOL {ℓ¹} {ℓ²} {ℓ³} {ℓ⁴} TM) (D : IFOL {ℓ¹'}
{ℓ²'} {ℓ³'} {ℓ⁴'} TM) : Set (lsuc (ℓ¹ ⊔ ℓ² ⊔ ℓ³ ⊔ ℓ⁴ ⊔ ℓ¹' ⊔ ℓ²' ⊔ ℓ³' ⊔ ℓ⁴'))
where
    field

      -- We first make the base category with its final object
      mCon : (IFOL.Con S) → (IFOL.Con D)
      mSub : {Δ : (IFOL.Con S)}{Γ : (IFOL.Con S)} → (IFOL.Sub S Δ Γ) → (IFOL.Sub
D (mCon Δ) (mCon Γ))
      mFor : {Γ : (IFOL.Con S)} → (IFOL.For S Γ) → (IFOL.For D (mCon Γ))
      mPf : {Γ : (IFOL.Con S)} {A : IFOL.For S Γ} → IFOL.Pf S Γ A → IFOL.Pf D
(mCon Γ) (mFor A)


  record Morphism (TM : Set)(S : IFOL {ℓ¹} {ℓ²} {ℓ³} {ℓ⁴} TM) (D : IFOL {ℓ¹'}
{ℓ²'} {ℓ³'} {ℓ⁴'} TM) : Set (lsuc (ℓ¹ ⊔ ℓ² ⊔ ℓ³ ⊔ ℓ⁴ ⊔ ℓ¹' ⊔ ℓ²' ⊔ ℓ³' ⊔ ℓ⁴'))
where
    field m : Mapping TM S D
    mCon = Mapping.mCon m
    mSub = Mapping.mSub m
    mFor = Mapping.mFor m
    mPf   = Mapping.mPf m
    field
      e◇ : mCon (IFOL.◇ S) ≡ IFOL.◇ D
      e[]f : {Γ Δ : IFOL.Con S}{A : IFOL.For S Γ}{σ : IFOL.Sub S Δ Γ} → mFor
(IFOL._[_]f S A σ) ≡ IFOL._[_]f D (mFor A) (mSub σ)
      -- Proofs are in prop, so some equations are not needed
      --[]p : {Γ Δ : IFOL.Con S}{A : IFOL.For S Γ}{pf : IFOL._⊢_ S Γ A}{σ :
IFOL.Sub S Δ Γ} → mPf (IFOL._[_]p S pf σ) ≡ IFOL._[_]p D (mPf pf) (mSub σ)
      e▷p : {Γ : IFOL.Con S}{A : IFOL.For S Γ} → mCon (IFOL._▷p_ S Γ A) ≡
IFOL._▷p_ D (mCon Γ) (mFor A)
      --πp² : {Γ Δ : IFOL.Con S}{A : IFOL.For S Γ}{σ : IFOL.Sub S Δ (IFOL._▷p_ S Γ
A)} → mPf (IFOL.πp² S σ) ≡ IFOL.πp¹ D (subst (IFOL.Sub D (mCon Δ)) ▷p (mSub σ))
      eR : {Γ : IFOL.Con S}{t u : TM} → mFor {Γ} (IFOL.R S t u) ≡ IFOL.R D t u
      e⇒ : {Γ : IFOL.Con S}{A B : IFOL.For S Γ} → mFor (IFOL._⇒_ S A B) ≡
IFOL._⇒_ D (mFor A) (mFor B)
      e∀∀ : {Γ : IFOL.Con S}{A : TM → IFOL.For S Γ} → mFor (IFOL.∀∀ S A) ≡
IFOL.∀∀ D (λ t → (mFor (A t)))
      -- No equation needed for lam, app, ∀i, ∀e as their output are in prop

  record TrNat {TM : Set}{S : IFOL {ℓ¹} {ℓ²} {ℓ³} {ℓ⁴} TM} {D : IFOL {ℓ¹'} {ℓ²'}
{ℓ³'} {ℓ⁴'} TM} (a : Mapping TM S D) (b : Mapping TM S D) : Set (lsuc (ℓ¹ ⊔ ℓ² ⊔
ℓ³ ⊔ ℓ⁴ ⊔ ℓ¹' ⊔ ℓ²' ⊔ ℓ³' ⊔ ℓ⁴')) where
    field
      f : (Γ : IFOL.Con S) → IFOL.Sub D (Mapping.mCon a Γ) (Mapping.mCon b Γ)
      -- Unneeded because Sub are in prop
      --eq : (Γ Δ : IFOL.Con S)(σ : IFOL.Sub S Γ Δ) → (IFOL._∘_ D (f Δ)
(Mapping.mSub a σ)) ≡ (IFOL._∘_ D (Mapping.mSub b σ) (f Γ))

  _∘TrNat_ : {TM : Set}{S : IFOL {ℓ¹} {ℓ²} {ℓ³} {ℓ⁴} TM}{D : IFOL {ℓ¹'} {ℓ²'}
{ℓ³'} {ℓ⁴'} TM}{a b c : Mapping TM S D} → TrNat a b → TrNat b c → TrNat a c
  _∘TrNat_ {D = D} α β = record { f = λ Γ → IFOL._∘_ D (TrNat.f β Γ) (TrNat.f α
Γ) }

  idTrNat : {TM : Set}{S : IFOL {ℓ¹} {ℓ²} {ℓ³} {ℓ⁴} TM}{D : IFOL {ℓ¹'} {ℓ²'}
{ℓ³'} {ℓ⁴'} TM}{a : Mapping TM S D} → TrNat a a
  idTrNat {D = D} = record { f = λ Γ → IFOL.id D }
\end{code}
```

```
\begin{code}
{-# OPTIONS --prop --rewriting #-}

open import PropUtil

module IFOLInitial (TM : Set) where

  open import IFOL2
  open import Agda.Primitive
  open import ListUtil

  --#
  data For : Set where
    R : TM → TM → For
    _⇒_ : For → For → For
    ∀∀ : (TM → For) → For
  --#

  data Con : Set where
    ◇ : Con
    _▷ₚ_ : Con → For → Con

  variable
    Γ Δ Ξ : Con
    A B : For

  data PfVar : Con → For → Prop where
    pvzero : PfVar (Γ ▷ₚ A) A
    pvnext : PfVar Γ A → PfVar (Γ ▷ₚ B) A
  --#
  data Pf : Con → For → Prop where
    var : PfVar Γ A → Pf Γ A
    lam : Pf (Γ ▷ₚ A) B → Pf Γ (A ⇒ B)
    app : Pf Γ (A ⇒ B) → Pf Γ A → Pf Γ B
    ∀i : {A : TM → For} → ((t : TM) → Pf Γ (A t)) → Pf Γ (∀∀ A)
    ∀e : {A : TM → For} → Pf Γ (∀∀ A) → (t : TM) → Pf Γ (A t)
  --#


  data Ren : Con → Con → Prop where
    ε : Ren Γ ◇
    _,ₚR_ : Ren Δ Γ → PfVar Δ A → Ren Δ (Γ ▷ₚ A)

  rightR : Ren Δ Γ → Ren (Δ ▷ₚ A) Γ
  rightR ε = ε
  rightR (σ ,ₚR pv) = (rightR σ) ,ₚR (pvnext pv)

  idR : Ren Γ Γ
  idR {◇} = ε
  idR {Γ ▷ₚ A} = (rightR (idR {Γ})) ,ₚR pvzero

  data Sub : Con → Con → Prop where
    ε : Sub Γ ◇
    _,ₚ_ : Sub Δ Γ → Pf Δ A → Sub Δ (Γ ▷ₚ A)

  πₚ¹ : Sub Δ (Γ ▷ₚ A) → Sub Δ Γ
  πₚ² : Sub Δ (Γ ▷ₚ A) → Pf Δ A
  πₚ¹ (σ ,ₚ pf) = σ
  πₚ² (σ ,ₚ pf) = pf

  SubR : Ren Γ Δ → Sub Γ Δ
  SubR ε = ε
  SubR (σ ,ₚR pv) = SubR σ ,ₚ var pv
```

```
id : Sub Γ Γ
id = SubR idR

_[_]pvr : PfVar Γ A → Ren Δ Γ → PfVar Δ A
pvzero [ _ ,ₚR pv ]pvr = pv
pvnext pv [ σ ,ₚR _ ]pvr = pv [ σ ]pvr
_[_]pr : Pf Γ A → Ren Δ Γ → Pf Δ A
var pv [ σ ]pr = var (pv [ σ ]pvr)
lam pf [ σ ]pr = lam (pf [ (rightR σ) ,ₚR pvzero ]pr)
app pf pf' [ σ ]pr = app (pf [ σ ]pr) (pf' [ σ ]pr)
∀i fpf [ σ ]pr = ∀i (λ t → (fpf t) [ σ ]pr)
∀e pf t [ σ ]pr = ∀e (pf [ σ ]pr) t

wkSub : Sub Δ Γ → Sub (Δ ▷ₚ A) Γ
wkSub ε = ε
wkSub (σ ,ₚ pf) = (wkSub σ) ,ₚ (pf [ rightR idR ]pr)

_[_]p : Pf Γ A → Sub Δ Γ → Pf Δ A
var pvzero [ _ ,ₚ pf ]p = pf
var (pvnext pv) [ σ ,ₚ _ ]p = var pv [ σ ]p
lam pf [ σ ]p = lam (pf [ wkSub σ ,ₚ var pvzero ]p)
app pf pf' [ σ ]p = app (pf [ σ ]p) (pf' [ σ ]p)
∀i fpf [ σ ]p = ∀i (λ t → (fpf t) [ σ ]p)
∀e pf t [ σ ]p = ∀e (pf [ σ ]p) t

_∘_ : {Γ Δ Ξ : Con} → Sub Δ Ξ → Sub Γ Δ → Sub Γ Ξ
ε ∘ β = ε
(α ,ₚ pf) ∘ β = (α ∘ β) ,ₚ (pf [ β ]p)


ifol : IFOL TM
ifol = record
        { Con = Con
        ; Sub = Sub
        ; _∘_ = _∘_
        ; id = id
        ; ◇ = ◇
        ; ε = ε
        ; For = λ Γ → For
        ; _[_]f = λ A σ → A
        ; []f-id = refl
        ; []f-∘ = refl
        ; Pf = Pf
        ; _[_]p = _[_]p
        ; _▷ₚ_ = _▷ₚ_
        ; πₚ¹ = πₚ¹
        ; πₚ² = πₚ²
        ; _,ₚ_ = _,ₚ_
        ; R = R
        ; _⇒_ = _⇒_
        ; ∀∀ = ∀∀
        ; R[] = refl
        ; []f-⇒ = refl
        ; []f-∀∀ = refl
        ; lam = lam
        ; app = app
        ; ∀i = ∀i
        ; ∀e = ∀e
        }

module InitialMorphism (M : IFOL {ℓ¹}{ℓ²}{ℓ³}{ℓ⁴} TM) where

    mCon : Con → (IFOL.Con M)
```

```
      mFor : {Γ : Con} → For → (IFOL.For M (mCon Γ))

      mCon ◇ = IFOL.◇ M
      mCon (Γ ▷ₚ A) = IFOL._▷ₚ_ M (mCon Γ) (mFor {Γ} A)
      mFor {Γ} (R t u) = IFOL.R M t u
      mFor {Γ} (A ⇒ B) = IFOL._⇒_ M (mFor {Γ} A) (mFor {Γ} B)
      mFor {Γ} (∀∀ A) = IFOL.∀∀ M (λ t → mFor {Γ} (A t))


      mSub : {Δ : Con}{Γ : Con} → Sub Δ Γ → (IFOL.Sub M (mCon Δ) (mCon Γ))
      mPf : {Γ : Con} {A : For} → Pf Γ A → IFOL.Pf M (mCon Γ) (mFor {Γ} A)

      e[]f⁰ : {Γ : Con}{A B : For} → mFor {Γ ▷ₚ B} A ≡ IFOL._[_]f M (mFor {Γ} A)
(IFOL.πₚ¹ M (IFOL.id M))
      e[]f⁰ {A = R t u} = ≡sym (IFOL.R[] M)
      e[]f⁰ {A = A ⇒ B} = ≡sym (≡tran ( IFOL.[]f-⇒ M) (cong₂ (IFOL._⇒_ M) (≡sym
(e[]f⁰ {A = A})) (≡sym (e[]f⁰ {A = B}))))
      e[]f⁰ {Γ}{A = ∀∀ A} = ≡sym (≡tran (IFOL.[]f-∀∀ M {F = λ t → mFor {Γ} (A
t)}) (cong (IFOL.∀∀ M) (≡fun (λ t → ≡sym (e[]f⁰ {Γ} {A = A t})))))

      e[]f : {Γ Δ : Con}{A : For}{σ : Sub Δ Γ} → mFor {Δ} A ≡ IFOL._[_]f M (mFor
{Γ} A) (mSub σ)
      e[]f {A = R t u} {σ} = ≡sym (IFOL.R[] M)
      e[]f {A = A ⇒ B} {σ} = ≡sym (≡tran ( IFOL.[]f-⇒ M) (cong₂ (IFOL._⇒_ M)
(≡sym (e[]f {A = A} {σ})) (≡sym (e[]f {A = B} {σ}))))
      e[]f {Γ}{A = ∀∀ A} {σ} = ≡sym (≡tran (IFOL.[]f-∀∀ M {F = λ t → mFor {Γ} (A
t)}) (cong (IFOL.∀∀ M) (≡fun (λ t → ≡sym (e[]f {Γ} {A = A t} {σ})))))

      mPf {A = A} (var (pvzero {Γ})) = substP (IFOL.Pf M _) (≡sym (e[]f⁰ {Γ} {A}
{A})) (IFOL.πₚ² M (IFOL.id M))
      mPf {A = A} (var (pvnext {Γ} {B = B} pv)) = substP (IFOL.Pf M _) (≡sym
(e[]f⁰ {Γ} {A} {B})) (IFOL._[_]p M (mPf (var pv)) (IFOL.πₚ¹ M (IFOL.id M)))
      mPf {Γ} (lam {A = A} {B} pf) = IFOL.lam M (substP (IFOL.Pf M _) (e[]f⁰ {Γ}
{B} {A}) (mPf pf))
      mPf (app pf pf') = IFOL.app M (mPf pf) (mPf pf')
      mPf (∀i pf) = IFOL.∀i M (λ t → mPf (pf t))
      mPf (∀e pf t) = IFOL.∀e M (mPf pf) t

      mSub ε = IFOL.ε M
      mSub (_,ₚ_ {Δ} {Γ = Γ} {A} σ pf) = IFOL._,ₚ_ M (mSub σ) (substP (IFOL.Pf M
_) (e[]f {Γ} {Δ} {A} {σ}) (mPf pf))

      mor : Morphism TM ifol M
      mor = record {
        m = record {
          mCon = mCon
          ; mSub = mSub
          ; mFor = λ {Γ} A → mFor {Γ} A
          ; mPf = mPf
        }
        ; e◇ = refl
        ; e▷ₚ = refl
        ; e[]f = λ {Γ}{Δ}{A}{σ} → e[]f {Γ}{Δ}{A}{σ}
        ; eR = refl
        ; e∀∀ = refl
        ; e⇒ = refl
        }

  module InitialMorphismUniqueness {M : IFOL {lzero} {lzero} {lzero} {lzero} TM}
{m : Morphism TM ifol M} where

    open InitialMorphism M
    mCon≡ : {Γ : Con} → mCon Γ ≡ (Morphism.mCon m Γ)
    mFor≡ : {Γ : Con} {A : For} → mFor {Γ} A ≡ subst (IFOL.For M) (≡sym mCon≡)
```

```
(Morphism.mFor m {Γ} A)

    mCon≡ {◇} = ≡sym (Morphism.e◇ m)
    mCon≡ {Γ ▷p A} = ≡sym (≡tran (Morphism.e▷p m) (cong₂' (IFOL._▷p_ M) (≡sym
mCon≡) (≡sym mFor≡)))
    mFor≡ {Γ} {A ⇒ B} = ≡sym (≡tran²
      (cong (subst (IFOL.For M) (≡sym mCon≡)) (Morphism.e⇒ m))
      (substppoly {eq = ≡sym (mCon≡ {Γ})} {f = λ {ξ} X Y → IFOL._⇒_ M {ξ} X Y})
      (cong₂ (IFOL._⇒_ M) (≡sym (mFor≡ {Γ} {A})) (≡sym (mFor≡ {Γ} {B}))))
    mFor≡ {Γ} {R t u} = ≡sym (≡tran (cong (subst (IFOL.For M) (≡sym mCon≡))
(Morphism.eR m)) (substpoly {f = λ {ξ} → IFOL.R M {ξ} t u} {eq = ≡sym mCon≡}))
    mFor≡ {Γ} {∀∀ A} = ≡sym (≡tran³
      (cong (subst (IFOL.For M) (≡sym mCon≡)) (Morphism.e∀∀ m))
      (substfpoly {eq = ≡sym mCon≡}{f = λ {ξ} X → IFOL.∀∀ M {ξ} X}{x = λ t →
Mapping.mFor (Morphism.m m) {Γ} (A t)})
      (cong (IFOL.∀∀ M) (≡sym coefun'))
      (cong (IFOL.∀∀ M) (≡fun λ t → ≡sym (mFor≡ {Γ} {A t}))))


    -- Those two lines are not needed as those sorts are in Prop
    --mSub≡ : {Δ : Con}{Γ : Con}{σ : Sub Δ Γ} → mSub {Δ} {Γ} σ ≡ Morphism.mSub m
{Δ} {Γ} σ
    --mPf≡ : {Γ : Con} {A : For}{p : Pf Γ A} → mPf {Γ} {A} p ≡ Morphism.mPf m {Γ}
{A} p

\end{code}
```

```
\begin{code}
{-# OPTIONS --prop --rewriting #-}

open import PropUtil

module IFOLCompleteness (TM : Set) where

  open import Agda.Primitive
  open import IFOL2
  open import ListUtil

  record Kripke : Set (lsuc (ℓ¹)) where
    field
      World : Set ℓ¹
      _-w->_ : World → World → Prop ℓ¹ -- arrows
      -w->id : {w : World} → w -w-> w -- id arrow
      _∘-w->_ : {w w' w'' : World} → w -w-> w' → w' -w-> w'' → w -w-> w'' -- arrow composition

      REL : World → TM → TM → Prop ℓ¹
      REL≤ : {t u : TM} {w w' : World} → w -w-> w' → REL w t u → REL w' t u

    infixr 10 _∘_

    Con : Set (lsuc ℓ¹)
    Con = (World → Prop ℓ¹) ×'' (λ Γ → {w w' : World} → (w -w-> w')→ Γ w → Γ w')
    Sub : Con → Con → Prop ℓ¹
    Sub Δ Γ = (w : World) → (proj×''₁ Δ) w → (proj×''₁ Γ) w
    _∘_ : {Γ Δ Ξ : Con} → Sub Δ Ξ → Sub Γ Δ → Sub Γ Ξ
    α ∘ β = λ w γ → α w (β w γ)
    id : {Γ : Con} → Sub Γ Γ
    id = λ w γ → γ
    ◇ : Con -- The initial object of the category
    ◇ = (λ w → ⊤) ,×'' (λ _ _ → tt)
    ε : {Γ : Con} → Sub Γ ◇ -- The morphism from the initial to any object
    ε w Γ = tt

    -- Functor Con → Set called For
    For : Set (lsuc ℓ¹)
    For = (World → Prop ℓ¹) ×'' (λ F → {w w' : World} → (w -w-> w')→ F w → F w')

    -- Proofs
    Pf : (Γ : Con) → For → Prop ℓ¹
    Pf Γ F = ∀ w (γ : (proj×''₁ Γ) w) →  (proj×''₁ F) w
    _[_]p : {Γ Δ : Con} → {F : For} → Pf Γ F → (σ : Sub Δ Γ) → Pf Δ F -- The functor's action on morphisms
    prf [ σ ]p = λ w → λ γ → prf w (σ w γ)
    -- Equalities below are useless because Γ ⊢ F is in prop
    -- []p-id : {Γ : Con} → {F : For Γ} → {prf : Γ ⊢ F} → prf [ id {Γ} ]p ≡ prf
    -- []p-∘ : {Γ Δ Ξ : Con} → {α : Sub Ξ Δ} → {β : Sub Δ Γ} → {F : For Γ} → {prf : Γ ⊢ F} → prf [ α ∘ β ]p
≡ (prf [ β ]p) [ α ]p


    -- → Prop⁺
    _▷p_ : (Γ : Con) → For → Con
    Γ ▷p F = (λ w → (proj×''₁ Γ) w ∧ (proj×''₁ F) w) ,×'' λ s γ → ⟨ proj×''₂ Γ s (proj₁ γ) , proj×''₂ F s
(proj₂ γ) ⟩

    πp¹ : {Γ Δ : Con} → {F : For} → Sub Δ (Γ ▷p F) → Sub Δ Γ
    πp¹ σ w δ = proj₁ (σ w δ)
    πp² : {Γ Δ : Con} → {F : For} → (σ : Sub Δ (Γ ▷p F)) → Pf Δ F
    πp² σ w δ = proj₂ (σ w δ)
    _,p_ : {Γ Δ : Con} → {F : For} → (σ : Sub Δ Γ) → Pf Δ F → Sub Δ (Γ ▷p F)
    (σ ,p pf) w δ = ⟨ (σ w δ) , pf w δ ⟩


    -- Base relation formula
    R : TM → TM → For
    R t u = (λ w → REL w t u) ,×'' REL≤

    -- Implication
    _⇒_ : For → For → For
    (F ⇒ G) = (λ w → {w' : World} → (s : w -w-> w') → ((proj×''₁ F) w') → ((proj×''₁ G) w')) ,×'' λ s f s'
f' → f (s ∘-w-> s') f'

    -- Forall
    ∀∀ : (TM → For) → For
    (∀∀ F) = (λ w → {t : TM} → proj×''₁ (F t) w) ,×'' λ s h {t} → proj×''₂ (F t) s (h {t})

    -- Lam & App
    lam : {Γ : Con} → {F : For} → {G : For} → Pf (Γ ▷p F) G → Pf Γ (F ⇒ G)
    lam {Γ} pf w γ {w'} s x = pf w' ⟨ proj×''₂ Γ s γ , x ⟩
    --lam prf = λ w γ w' s h → prf w (γ , h)
    app : {Γ : Con} → {F G : For} → Pf Γ (F ⇒ G) → Pf Γ F → Pf Γ G
    app pf pf' w γ = pf w γ -w->id (pf' w γ)
    -- ∀i and ∀e
    ∀i : {Γ : Con} {A : TM → For} → ((t : TM) → Pf Γ (A t)) → Pf Γ (∀∀ A)
    ∀i pf w γ {t} = pf t w γ
    ∀e : {Γ : Con} {A : TM → For} → Pf Γ (∀∀ A) → (t : TM) → Pf Γ (A t)
    ∀e pf t w γ = pf w γ
    -- Again, we don't write the _[_]p equalities as everything is in Prop

    ifol : IFOL TM
    ifol = record
            { Con = Con
            ; Sub = Sub
            ; _∘_ = λ {Γ}{Δ}{Ξ} σ δ → _∘_ {Γ}{Δ}{Ξ} σ δ
            ; id =  λ {Γ} → id {Γ}
            ; ◇ = ◇
            ; ε = λ {Γ} → ε {Γ}
            ; For = λ Γ → For
            ; _[_]f = λ A σ → A
            ; []f-id = refl
            ; []f-∘ = refl
            ; Pf = Pf
            ; _[_]p = λ {Γ}{Δ}{F} pf σ → _[_]p {Γ}{Δ}{F} pf σ
            ; _▷p_  = _▷p_
            ; πp¹  = λ {Γ}{Δ}{F}σ → πp¹  {Γ}{Δ}{F}  σ
            ; πp²  = λ {Γ}{Δ}{F}σ → πp²  {Γ}{Δ}{F}  σ
            ; _,p_  = λ {Γ}{Δ}{F} σ pf → _,p_ {Γ}{Δ}{F}σ pf
            ; R = R
            ; R[] = refl
            ; _⇒_  = _⇒_
            ; []f-⇒ = refl
            ; ∀∀ = ∀∀
            ; []f-∀∀ = refl
            ; lam = λ {Γ}{F}{G} pf → lam {Γ}{F}{G} pf
            ; app = λ {Γ}{F}{G} pf pf' → app {Γ}{F}{G} pf pf'
            ; ∀i = λ {Γ}{A} F → ∀i {Γ}{A} F
            ; ∀e = λ {Γ}{A} F t → ∀e {Γ}{A} F t
            }
  module U where

    import IFOLInitial TM as I

    U : Kripke
    U = record
        { World = I.Con
        ; _-w->_ = λ Γ Δ → I.Sub Δ Γ
        ; -w->id = I.id
        ; _∘-w->_ = λ σ σ' → σ I.∘ σ'
        ; REL = λ Γ t u → I.Pf Γ (I.R t u)
        ; REL≤ = λ σ pf → pf I.[ σ ]p
        }

    open Kripke U

    y : Mapping TM I.ifol ifol
    y = record
      { mCon = λ Γ → (λ Δ → I.Sub Δ Γ) ,×'' λ σ δ → δ I.∘ σ
      ; mSub = λ σ Ξ δ → σ I.∘ δ
      ; mFor = λ A → (λ Ξ → I.Pf Ξ A) ,×'' λ σ pf → pf I.[ σ ]p
      ; mPf = λ pf Ξ σ → pf I.[ σ ]p
      }
    m : Morphism TM I.ifol ifol
    m = I.InitialMorphism.mor ifol

    q : (Γ : I.Con) → Sub (Morphism.mCon m Γ) (Mapping.mCon y Γ)
    u : (Γ : I.Con) → Sub (Mapping.mCon y Γ) (Morphism.mCon m Γ)

    ⟦_⟧c = Morphism.mCon m
    ⟦_,_⟧f = λ A Γ →  Morphism.mFor m {Γ} A

    q⁰ : {F : I.For} → {Γ Γ₀ : I.Con} → proj×''₁ ⟦ F , Γ₀ ⟧f Γ → I.Pf Γ F
    u⁰ : {F : I.For} → {Γ Γ₀ : I.Con} → I.Pf Γ F → proj×''₁ ⟦ F , Γ₀ ⟧f Γ

    q⁰ {I.R t v} {Γ} h = h
    q⁰ {I.∀∀ A}  {Γ} h = I.∀i (λ t → q⁰ {A t} h)
    q⁰ {A I.⇒ B} {Γ} h = I.lam (q⁰ {B} (h (I.πp¹ I.id) (u⁰ {A} (I.var I.pvzero))))
    u⁰ {I.R t v} {Γ} pf = pf
    u⁰ {I.∀∀ A}  {Γ} pf {t} = u⁰ {A t} (I.∀e pf t)
    u⁰ {A I.⇒ B} {Γ} pf iq hF  = u⁰ {B} (I.app (pf I.[ iq ]p) (q⁰ hF) )


    q I.◇ w γ = I.ε
    q (Γ I.▷p A) w γ = (q Γ w (proj₁ γ) I.,p q⁰ (proj₂ γ))
    u I.◇ w σ = tt
    u (Γ I.▷p A) w σ = ⟨ (u Γ w (I.πp¹ σ)) , u⁰ (I.πp² σ) ⟩

    ηq : TrNat (Morphism.m m) y
```

```
    ηq = record { f = q }
    ηu : TrNat y (Morphism.m m)
    ηu = record { f = u }

    eq : ηu ∘TrNat ηq ≡ idTrNat
    eq = refl

\end{code}
```

```
\begin{code}
{-# OPTIONS --prop --rewriting #-}

open import PropUtil

module FFOL where

  open import Agda.Primitive
  open import ListUtil

  variable
    ℓ¹ ℓ² ℓ³ ℓ⁴ ℓ⁵ : Level

  record FFOL : Set (lsuc (ℓ¹ ⊔ ℓ² ⊔ ℓ³ ⊔ ℓ⁴ ⊔ ℓ⁵)) where
    infixr 10 _∘_
    infixr 5 _⊢_
    field

      --# We first make the base category with its terminal object
      Con : Set ℓ¹
      Sub : Con → Con → Set ℓ⁵ -- It makes a category
      _∘_ : {Γ Δ Ξ : Con} → Sub Δ Ξ → Sub Γ Δ → Sub Γ Ξ
      ∘-ass : {Γ Δ Ξ Ψ : Con}{α : Sub Γ Δ}{β : Sub Δ Ξ}{γ : Sub Ξ Ψ}
        → (γ ∘ β) ∘ α ≡ γ ∘ (β ∘ α)
      id : {Γ : Con} → Sub Γ Γ
      idl : {Γ Δ : Con} {σ : Sub Γ Δ} →  (id {Δ}) ∘ σ ≡ σ
      idr : {Γ Δ : Con} {σ : Sub Γ Δ} →  σ ∘ (id {Γ}) ≡ σ
      ◇ : Con -- The terminal object of the category
      ε : {Γ : Con} → Sub Γ ◇ -- The morphism from any object to the terminal
      ε-u : {Γ : Con} → {σ : Sub Γ ◇} → σ ≡ ε {Γ}

      --# Functor Con → Set called Tm
      Tm : Con → Set ℓ²
      _[_]t : {Γ Δ : Con} → Tm Γ → Sub Δ Γ → Tm Δ -- Action on morphisms
      []t-id : {Γ : Con} → {x : Tm Γ} → x [ id {Γ} ]t ≡ x
      []t-∘ : {Γ Δ Ξ : Con} → {α : Sub Ξ Δ}{β : Sub Δ Γ} → {t : Tm Γ}
        → t [ β ∘ α ]t ≡ (t [ β ]t) [ α ]t

      --# Tm : Set⁺
      _▹ₜ : Con → Con
      πₜ¹ : {Γ Δ : Con} → Sub Δ (Γ ▹ₜ) → Sub Δ Γ
      πₜ² : {Γ Δ : Con} → Sub Δ (Γ ▹ₜ) → Tm Δ
      _,ₜ_ : {Γ Δ : Con} → Sub Δ Γ → Tm Δ → Sub Δ (Γ ▹ₜ)
      πₜ²∘,ₜ : {Γ Δ : Con} → {σ : Sub Δ Γ} → {t : Tm Δ} → πₜ² (σ ,ₜ t) ≡ t
      πₜ¹∘,ₜ : {Γ Δ : Con} → {σ : Sub Δ Γ} → {t : Tm Δ} → πₜ¹ (σ ,ₜ t) ≡ σ
      ,ₜ∘πₜ : {Γ Δ : Con} → {σ : Sub Δ (Γ ▹ₜ)} → (πₜ¹ σ) ,ₜ (πₜ² σ) ≡ σ
      ,ₜ∘ : {Γ Δ Ξ : Con}{σ : Sub Γ Ξ}{δ : Sub Δ Γ}{t : Tm Γ}
        → (σ ,ₜ t) ∘ δ ≡ (σ ∘ δ) ,ₜ (t [ δ ]t)

      --# Functor Con → Set called For
      For : Con → Set ℓ³
      _[_]f : {Γ Δ : Con} → For Γ → Sub Δ Γ → For Δ -- Action on morphisms
      []f-id : {Γ : Con} → {F : For Γ} → F [ id {Γ} ]f ≡ F
      []f-∘ : {Γ Δ Ξ : Con} → {α : Sub Ξ Δ} → {β : Sub Δ Γ} → {F : For Γ}
        → F [ β ∘ α ]f ≡ (F [ β ]f) [ α ]f

      --# Functor Con × For → Prop called Pf or ⊢
      _⊢_ : (Γ : Con) → For Γ → Prop ℓ⁴
      -- Action on morphisms
      _[_]p : {Γ Δ : Con} → {F : For Γ} → Γ ⊢ F → (σ : Sub Δ Γ) → Δ ⊢ (F [ σ ]f)
      --# Equalities below are useless because Γ ⊢ F is in prop
      -- []p-id : {Γ : Con} → {F : For Γ} → {prf : Γ ⊢ F}
      --  → prf [ id {Γ} ]p ≡ prf
      -- []p-∘ : {Γ Δ Ξ : Con}{α : Sub Ξ Δ}{β : Sub Δ Γ}{F : For Γ}{prf : Γ ⊢ F}
      --  → prf [ α ∘ β ]p ≡ (prf [ β ]p) [ α ]p

      --# → Prop⁺
      _▹ₚ_ : (Γ : Con) → For Γ → Con
      πₚ¹ : {Γ Δ : Con}{F : For Γ} → Sub Δ (Γ ▹ₚ F) → Sub Δ Γ
      πₚ² : {Γ Δ : Con}{F : For Γ} → (σ : Sub Δ (Γ ▹ₚ F)) → Δ ⊢ (F [ πₚ¹ σ ]f)
      _,ₚ_ : {Γ Δ : Con}{F : For Γ} → (σ : Sub Δ Γ) → Δ ⊢ (F [ σ ]f) → Sub Δ (Γ ▹ₚ F)
      --# And its equalities
      ,ₚ∘πₚ : {Γ Δ : Con}{F : For Γ}{σ : Sub Δ (Γ ▹ₚ F)} → (πₚ¹ σ) ,ₚ (πₚ² σ) ≡ σ
      πₚ¹∘,ₚ : {Γ Δ : Con}{σ : Sub Δ Γ}{F : For Γ}{prf : Δ ⊢ (F [ σ ]f)}
        → πₚ¹ (σ ,ₚ prf) ≡ σ
      -- Equality below is useless because Γ ⊢ F is in Prop
      -- πₚ²∘,ₚ : {Γ Δ : Con}{σ : Sub Δ Γ}{F : For Γ}{prf : Δ ⊢ (F [ σ ]f)}
      -- → πₚ² (σ ,ₚ prf) ≡ prf
      ,ₚ∘ : {Γ Δ Ξ : Con}{σ : Sub Γ Ξ}{δ : Sub Δ Γ}{F : For Ξ}{prf : Γ ⊢ (F [ σ ]f)}
        → (σ ,ₚ prf) ∘ δ ≡ (σ ∘ δ) ,ₚ (substP (λ F → Δ ⊢ F) (≡sym []f-∘) (prf [ δ ]p))
      --#


      {-- FORMULAE CONSTRUCTORS --}
      --# Formulas with relation on terms
      R : {Γ : Con} → (t u : Tm Γ) → For Γ
      R[] : {Γ Δ : Con} → {σ : Sub Δ Γ} → {t u : Tm Γ}
        → (R t u) [ σ ]f ≡ R (t [ σ ]t) (u [ σ ]t)

      --# Implication
      _⇒_ : {Γ : Con} → For Γ → For Γ → For Γ
      []f-⇒ : {Γ Δ : Con} → {F G : For Γ} → {σ : Sub Δ Γ}
        → (F ⇒ G) [ σ ]f ≡ (F [ σ ]f) ⇒ (G [ σ ]f)

      --# Forall
      ∀∀ : {Γ : Con} → For (Γ ▹ₜ) → For Γ
      []f-∀∀ : {Γ Δ : Con} → {F : For (Γ ▹ₜ)} → {σ : Sub Δ Γ}
        → (∀∀ F) [ σ ]f ≡ (∀∀ (F [ (σ ∘ πₜ¹ id) ,ₜ πₜ² id ]f))

      --#
      {-- PROOFS CONSTRUCTORS --}
      -- Again, we don't have to write the _[_]p equalities as Proofs are in Prop

      --# Lam & App
      lam : {Γ : Con}{F G : For Γ} → (Γ ▹ₚ F) ⊢ (G [ πₚ¹ id ]f) → Γ ⊢ (F ⇒ G)
      app : {Γ : Con}{F G : For Γ} → Γ ⊢ (F ⇒ G) → Γ ⊢ F → Γ ⊢ G

      --# ∀i and ∀e
      ∀i : {Γ : Con}{F : For (Γ ▹ₜ)} → (Γ ▹ₜ) ⊢ F → Γ ⊢ (∀∀ F)
      ∀e : {Γ : Con}{F : For (Γ ▹ₜ)} → Γ ⊢ (∀∀ F) → {t : Tm Γ} → Γ ⊢ ( F [(id {Γ}) ,ₜ t ]f)


    --# Examples
    -- Proof utils
    forall-in : {Γ Δ : Con} {σ : Sub Γ Δ} {A : For (Δ ▹ₜ)} → Γ ⊢ ∀∀ (A [ (σ ∘ πₜ¹ id) ,ₜ πₜ² id ]f) → Γ ⊢ (∀∀ A [ σ ]f)
    forall-in {Γ = Γ} f = substP (λ F → Γ ⊢ F) (≡sym ([]f-∀∀)) f
    wkₜ : {Γ : Con} → Sub (Γ ▹ₜ) Γ
    wkₜ = πₜ¹ id
    0ₜ : {Γ : Con} → Tm (Γ ▹ₜ)
    0ₜ = πₜ² id
    1ₜ : {Γ : Con} → Tm (Γ ▹ₜ ▹ₜ)
    1ₜ = πₜ² (πₜ¹ id)
    wkₚ : {Γ : Con} {A : For Γ} → Sub (Γ ▹ₚ A) Γ
    wkₚ = πₚ¹ id
    0ₚ : {Γ : Con} {A : For Γ} → Γ ▹ₚ A ⊢ A [ πₚ¹ id ]f
    0ₚ = πₚ² id

    --  Examples
    ex0 : {A :  For ◇} → ◇ ⊢ (A ⇒ A)
    ex0 {A = A} = lam 0ₚ
    {-
    ex1 : {A : For (◇ ▹ₜ)} → ◇ ⊢ ((∀∀ A) ⇒ (∀∀ A))
    -- πₚ¹ id is adding an unused variable (syntax's llift)
    ex1 {A = A} = lam (forall-in (∀i (substP (λ σ → ((◇ ▹ₚ ∀∀ A) ▹ₜ) ⊢ (A [ σ ]f)) {!!} {!!})))
    -- (∀ x ∀ y . A(y,x)) ⇒ ∀ x ∀ y . A(x,y)
    -- translation is (∀ ∀ A(0,1)) => (∀ ∀ A(1,0))
    ex1' : {A : For (◇ ▹ₜ ▹ₜ)} → ◇ ⊢ ((∀∀ (∀∀ A)) ⇒ ∀∀ (∀∀ ( A [ (ε ,ₜ 0ₜ) ,ₜ 1ₜ ]f)))
    ex1' = {!!}
    -- (A ⇒ ∀ x . B(x)) ⇒ ∀ x . A ⇒ B(x)
    ex2 : {A : For ◇} → {B : For (◇ ▹ₜ)} → ◇ ⊢ ((A ⇒ (∀∀ B)) ⇒ (∀∀ ((A [ wkₜ ]f) ⇒ B)))
    ex2 = {!!}
    -- ∀ x y . A(x,y) ⇒ ∀ x . A(x,x)
    -- For simplicity, I swiched positions of parameters of A (somehow...)
    ex3 : {A : For (◇ ▹ₜ ▹ₜ)} → ◇ ⊢ ((∀∀ (∀∀ A)) ⇒ (∀∀ (A [ id ,ₜ 0ₜ ]f)))
    ex3 = {!!}
    -- ∀ x . A (x) ⇒ ∀ x y . A(x)
    ex4 : {A : For (◇ ▹ₜ)} → ◇ ⊢ ((∀∀ A) ⇒ (∀∀ (∀∀ (A [ ε ,ₜ 1ₜ ]f))))
    ex4 = {!!}
    -- (((∀ x . A (x)) ⇒ B)⇒ B) ⇒ ∀ x . ((A (x) ⇒ B) ⇒ B)
    ex5 : {A : For (◇ ▹ₜ)} → {B : For ◇} → ◇ ⊢ ((((∀∀ A) ⇒ B) ⇒ B) ⇒ (∀∀ ((A ⇒ (B [ wkₜ ]f)) ⇒ (B [ wkₜ ]f))))
    ex5 = {!!}
    -}

  record Mapping (S : FFOL {ℓ¹} {ℓ²} {ℓ³} {ℓ⁴} {ℓ⁵}) (D : FFOL {ℓ¹} {ℓ²} {ℓ³} {ℓ⁴} {ℓ⁵}) : Set (lsuc (ℓ¹ ⊔ ℓ² ⊔ ℓ³ ⊔ ℓ⁴ ⊔
ℓ⁵)) where
    field

      -- We first make the base category with its final object
      mCon : (FFOL.Con S) → (FFOL.Con D)
      mSub : {Δ : (FFOL.Con S)}{Γ : (FFOL.Con S)} → (FFOL.Sub S Δ Γ) → (FFOL.Sub D (mCon Δ) (mCon Γ))
      mTm : {Γ : (FFOL.Con S)} → (FFOL.Tm S Γ) → (FFOL.Tm D (mCon Γ))
      mFor : {Γ : (FFOL.Con S)} → (FFOL.For S Γ) → (FFOL.For D (mCon Γ))
      m⊢ : {Γ : (FFOL.Con S)} {A : FFOL.For S Γ} → FFOL._⊢_ S Γ A → FFOL._⊢_ D (mCon Γ) (mFor A)


  record Morphism (S : FFOL {ℓ¹} {ℓ²} {ℓ³} {ℓ⁴} {ℓ⁵}) (D : FFOL {ℓ¹} {ℓ²} {ℓ³} {ℓ⁴} {ℓ⁵}) : Set (lsuc (ℓ¹ ⊔ ℓ² ⊔ ℓ³ ⊔ ℓ⁴ ⊔
ℓ⁵)) where
    field m : Mapping S D
    mCon = Mapping.mCon m
    mSub = Mapping.mSub m
    mTm  = Mapping.mTm  m
    mFor = Mapping.mFor m
    m⊢   = Mapping.m⊢ m
    field
      e∘ : {Γ Δ Ξ : FFOL.Con S}{δ : FFOL.Sub S Δ Ξ}{σ : FFOL.Sub S Γ Δ} → mSub (FFOL._∘_ S δ σ) ≡ FFOL._∘_ D (mSub δ) (mSub
σ)
      eid : {Γ : FFOL.Con S} → mSub (FFOL.id S {Γ}) ≡ FFOL.id D {mCon Γ}
      e◇ : mCon (FFOL.◇ S) ≡ FFOL.◇ D
      eε : {Γ : FFOL.Con S} → mSub (FFOL.ε S {Γ}) ≡ subst (FFOL.Sub D (mCon Γ)) (≡sym e◇) (FFOL.ε D {mCon Γ})
      e[]t : {Γ Δ : FFOL.Con S}{t : FFOL.Tm S Γ}{σ : FFOL.Sub S Δ Γ} → mTm (FFOL._[_]t S t σ) ≡ FFOL._[_]t D (mTm t) (mSub
σ)
      e▹ₜ : {Γ : FFOL.Con S} → mCon (FFOL._▹ₜ S Γ) ≡ FFOL._▹ₜ D (mCon Γ)
      eπₜ¹ : {Γ Δ : FFOL.Con S}{σ : FFOL.Sub S Δ (FFOL._▹ₜ S Γ)} → mSub (FFOL.πₜ¹ S σ) ≡ FFOL.πₜ¹ D (subst (FFOL.Sub D (mCon
Δ)) e▹ₜ (mSub σ))
      eπₜ² : {Γ Δ : FFOL.Con S}{σ : FFOL.Sub S Δ (FFOL._▹ₜ S Γ)} → mTm (FFOL.πₜ² S σ) ≡ FFOL.πₜ² D (subst (FFOL.Sub D (mCon
Δ)) e▹ₜ (mSub σ))
      e,ₜ : {Γ Δ : FFOL.Con S}{σ : FFOL.Sub S Δ Γ}{t : FFOL.Tm S Δ} → mSub (FFOL._,ₜ_ S σ t) ≡ subst (FFOL.Sub D (mCon Δ))
(≡sym e▹ₜ) (FFOL._,ₜ_ D (mSub σ) (mTm t))
      e[]f : {Γ Δ : FFOL.Con S}{A : FFOL.For S Γ}{σ : FFOL.Sub S Δ Γ} → mFor (FFOL._[_]f S A σ) ≡ FFOL._[_]f D (mFor A)
(mSub σ)
```

```
      -- Proofs are in prop, so no equation needed
      --[]p : {Γ Δ : FFOL.Con S}{A : FFOL.For S Γ}{pf : FFOL._⊢_ S Γ A}{σ : FFOL.Sub S Δ Γ} → m⊢ (FFOL._[_]p S pf σ) ≡
FFOL._[_]p D (m⊢ pf) (mSub σ)
      e▷ₚ : {Γ : FFOL.Con S}{A : FFOL.For S Γ} → mCon (FFOL._▷ₚ_ S Γ A) ≡ FFOL._▷ₚ_ D (mCon Γ) (mFor A)
      eπₚ¹ : {Γ Δ : FFOL.Con S}{A : FFOL.For S Γ}{σ : FFOL.Sub S Δ (FFOL._▷ₚ_ S Γ A)} → mSub (FFOL.πₚ¹ S σ) ≡ FFOL.πₚ¹ D
(subst (FFOL.Sub D (mCon Δ)) e▷ₚ (mSub σ))
      --πₚ² : {Γ Δ : FFOL.Con S}{A : FFOL.For S Γ}{σ : FFOL.Sub S Δ (FFOL._▷ₚ_ S Γ A)} → m⊢ (FFOL.πₚ² S σ) ≡ FFOL.πₚ¹ D
(subst (FFOL.Sub D (mCon Δ)) ▷ₚ (mSub σ))
      e,ₚ : {Γ Δ : FFOL.Con S}{A : FFOL.For S Γ}{σ : FFOL.Sub S Δ Γ}{pf : FFOL._⊢_ S Δ (FFOL._[_]f S A σ)}
        → mSub (FFOL._,ₚ_ S σ pf) ≡ subst (FFOL.Sub D (mCon Δ)) (≡sym e▷ₚ) (FFOL._,ₚ_ D (mSub σ) (substP (FFOL._⊢_ D (mCon
Δ)) e[]f (m⊢ pf)))
      eR : {Γ : FFOL.Con S}{t u : FFOL.Tm S Γ} → mFor (FFOL.R S t u) ≡ FFOL.R D (mTm t) (mTm u)
      e⇒ : {Γ : FFOL.Con S}{A B : FFOL.For S Γ} → mFor (FFOL._⇒_ S A B) ≡ FFOL._⇒_ D (mFor A) (mFor B)
      e∀∀ : {Γ : FFOL.Con S}{A : FFOL.For S (FFOL._▷ₜ S Γ)} → mFor (FFOL.∀∀ S A) ≡ FFOL.∀∀ D (subst (FFOL.For D) e▷ₜ (mFor
A))
      -- No equation needed for lam, app, ∀i, ∀e as their output are in prop

  record Tarski : Set₁ where
    field
      TM : Set
      REL : TM → TM → Prop
    infixr 10 _∘_
    Con = Set
    Sub : Con → Con → Set
    Sub Γ Δ = (Γ → Δ) -- It makes a posetal category
    _∘_ : {Γ Δ Ξ : Con} → Sub Δ Ξ → Sub Γ Δ → Sub Γ Ξ
    f ∘ g = λ x → f (g x)
    id : {Γ : Con} → Sub Γ Γ
    id = λ x → x
    ε : {Γ : Con} → Sub Γ ⊤ₛ -- The morphism from the initial to any object
    ε Γ = ttₛ
    ε-u : {Γ : Con} → {σ : Sub Γ ⊤ₛ} → σ ≡ ε {Γ}
    ε-u = refl

    -- Functor Con → Set called Tm
    Tm : Con → Set
    Tm Γ = Γ → TM
    _[_]t : {Γ Δ : Con} → Tm Γ → Sub Δ Γ → Tm Δ -- The functor's action on morphisms
    t [ σ ]t = λ γ → t (σ γ)
    []t-id : {Γ : Con} → {x : Tm Γ} → x [ id {Γ} ]t ≡ x
    []t-id = refl
    []t-∘ : {Γ Δ Ξ : Con} → {α : Sub Ξ Δ} → {β : Sub Δ Γ} → {t : Tm Γ} → t [ β ∘ α ]t ≡ (t [ β ]t) [ α ]t
    []t-∘ {α = α} {β} {t} = refl {_} {_} {λ z → t (β (α z))}

    _[_]tz : {Γ Δ : Con} → {n : Nat} → Array (Tm Γ) n → Sub Δ Γ → Array (Tm Δ) n
    tz [ σ ]tz = map (λ s → s [ σ ]t) tz
    []tz-∘  : {Γ Δ Ξ : Con} → {α : Sub Ξ Δ} → {β : Sub Δ Γ} → {n : Nat} → {tz : Array (Tm Γ) n} → tz [ β ∘ α ]tz ≡ tz [ β
]tz [ α ]tz
    []tz-∘ {tz = zero} = refl
    []tz-∘ {α = α} {β = β} {tz = next x tz} = substP (λ tz' → (next ((x [ β ]t) [ α ]t) tz') ≡ (((next x tz) [ β ]tz) [ α
]tz)) (≡sym ([]tz-∘ {α = α} {β = β} {tz = tz})) refl
    []tz-id : {Γ : Con} → {n : Nat} → {tz : Array (Tm Γ) n} → tz [ id ]tz ≡ tz
    []tz-id {tz = zero} = refl
    []tz-id {tz = next x tz} = substP (λ tz' → next x tz' ≡ next x tz) (≡sym ([]tz-id {tz = tz})) refl
    thm : {Γ Δ : Con} → {n : Nat} → {tz : Array (Tm Γ) n} → {σ : Sub Δ Γ} → {δ : Δ} → map (λ t → t δ) (tz [ σ ]tz) ≡ map (λ
t → t (σ δ)) tz
    thm {tz = zero} = refl
    thm {tz = next x tz} {σ} {δ} = substP (λ tz' → (next (x (σ δ)) (map (λ t → t δ) (map (λ s γ → s (σ γ)) tz))) ≡ (next (x
(σ δ)) tz')) (thm {tz = tz}) refl

    -- Tm⁺
    _▷ₜ : Con → Con
    Γ ▷ₜ = Γ × TM
    πₜ¹ : {Γ Δ : Con} → Sub Δ (Γ ▷ₜ) → Sub Δ Γ
    πₜ¹ σ = λ x → proj×₁ (σ x)
    πₜ² : {Γ Δ : Con} → Sub Δ (Γ ▷ₜ) → Tm Δ
    πₜ² σ = λ x → proj×₂ (σ x)
    _,ₜ_ : {Γ Δ : Con} → Sub Δ Γ → Tm Δ → Sub Δ (Γ ▷ₜ)
    σ ,ₜ t = λ x → (σ x) ,× (t x)
    πₜ²∘,ₜ : {Γ Δ : Con} → {σ : Sub Δ Γ} → {t : Tm Δ} → πₜ² (σ ,ₜ t) ≡ t
    πₜ²∘,ₜ {σ = σ} {t} = refl {a = t}
    πₜ¹∘,ₜ : {Γ Δ : Con} → {σ : Sub Δ Γ} → {t : Tm Δ} → πₜ¹ (σ ,ₜ t) ≡ σ
    πₜ¹∘,ₜ = refl
    ,ₜ∘πₜ : {Γ Δ : Con} → {σ : Sub Δ (Γ ▷ₜ)} → (πₜ¹ σ) ,ₜ (πₜ² σ) ≡ σ
    ,ₜ∘πₜ = refl
    ,ₜ∘ : {Γ Δ Ξ : Con}{σ : Sub Γ Ξ}{δ : Sub Δ Γ}{t : Tm Γ} → (σ ,ₜ t) ∘ δ ≡ (σ ∘ δ) ,ₜ (t [ δ ]t)
    ,ₜ∘ = refl

    -- Functor Con → Set called For
    For : Con → Set₁
    For Γ = Γ → Prop
    _[_]f : {Γ Δ : Con} → For Γ → Sub Δ Γ → For Δ
    F [ σ ]f = λ x → F (σ x)
    []f-id : {Γ : Con} → {F : For Γ} → F [ id {Γ} ]f ≡ F
    []f-id = refl
    []f-∘ : {Γ Δ Ξ : Con} → {α : Sub Ξ Δ} → {β : Sub Δ Γ} → {F : For Γ} → F [ β ∘ α ]f ≡ (F [ β ]f) [ α ]f
    []f-∘ = refl

    R : {Γ : Con} → Tm Γ → Tm Γ → For Γ
    R t u = λ γ → REL (t γ) (u γ)
    R[] : {Γ Δ : Con} → {σ : Sub Δ Γ} → {t u : Tm Γ} → (R t u) [ σ ]f ≡ R (t [ σ ]t) (u [ σ ]t)
    R[] {σ = σ} = cong₂ R refl refl

    -- Proofs
    _⊢_ : (Γ : Con) → For Γ → Prop
    Γ ⊢ F = ∀ (γ : Γ) → F γ
```

```
_[_]p : {Γ Δ : Con} → {F : For Γ} → Γ ⊢ F → (σ : Sub Δ Γ) → Δ ⊢ (F [ σ ]f)
prf [ σ ]p = λ γ → prf (σ γ)
-- Two rules are irrelevent beccause Γ ⊢ F is in Prop

-- → Prop⁺
_▹p_ : (Γ : Con) → For Γ → Con
Γ ▹p F = Γ ×'' F
πp¹ : {Γ Δ : Con} → {F : For Γ} → Sub Δ (Γ ▹p F) → Sub Δ Γ
πp¹ σ δ = proj×''₁ (σ δ)
πp² : {Γ Δ : Con} → {F : For Γ} → (σ : Sub Δ (Γ ▹p F)) → Δ ⊢ (F [ πp¹ σ ]f)
πp² σ δ = proj×''₂ (σ δ)
_,p_ : {Γ Δ : Con} → {F : For Γ} → (σ : Sub Δ Γ) → Δ ⊢ (F [ σ ]f) → Sub Δ (Γ ▹p F)
_,p_ {F = F} σ pf δ = (σ δ) ,×'' pf δ
,p∘πp : {Γ Δ : Con} → {F : For Γ} → {σ : Sub Δ (Γ ▹p F)} → (πp¹ σ) ,p (πp² σ) ≡ σ
,p∘πp = refl
πp¹∘,p : {Γ Δ : Con} → {σ : Sub Δ Γ} → {F : For Γ} → {prf : Δ ⊢ (F [ σ ]f)} → πp¹ {Γ} {Δ} {F} (σ ,p prf) ≡ σ
πp¹∘,p = refl
,p∘ : {Γ Δ Ξ : Con}{σ : Sub Γ Ξ}{δ : Sub Δ Γ}{F : For Ξ}{prf : Γ ⊢ (F [ σ ]f)} →
  (_,p_  {F = F} σ prf) ∘ δ ≡ (σ ∘ δ) ,p (substP (λ F → Δ ⊢ F) (≡sym ([]f-∘ {α = δ} {β = σ} {F = F})) (prf [ δ ]p))
,p∘ {Γ} {Δ} {Ξ} {σ} {δ} {F} {prf} = refl

-- Implication
_⇒_ : {Γ : Con} → For Γ → For Γ → For Γ
F ⇒ G = λ γ → (F γ) → (G γ)
[]f-⇒ : {Γ Δ : Con} → {F G : For Γ} → {σ : Sub Δ Γ} → (F ⇒ G) [ σ ]f ≡ (F [ σ ]f) ⇒ (G [ σ ]f)
[]f-⇒ = refl

-- Forall
∀∀ : {Γ : Con} → For (Γ ▹t) → For Γ
∀∀ {Γ} F = λ (γ : Γ) → (∀ (t : TM) → F (γ ,× t))
[]f-∀∀ : {Γ Δ : Con} → {F : For (Γ ▹t)} → {σ : Sub Δ Γ} → (∀∀ F) [ σ ]f ≡ (∀∀ (F [ (σ ∘ πt¹ id) ,t πt² id ]f))
[]f-∀∀ {Γ} {Δ} {F} {σ} = refl

-- Lam & App
lam : {Γ : Con} → {F : For Γ} → {G : For Γ} → (Γ ▹p F) ⊢ (G [ πp¹ id ]f) → Γ ⊢ (F ⇒ G)
lam pf = λ γ x → pf (γ ,×'' x)
app : {Γ : Con} → {F G : For Γ} → Γ ⊢ (F ⇒ G) → Γ ⊢ F → Γ ⊢ G
app pf pf' = λ γ → pf γ (pf' γ)
-- Again, we don't write the _[_]p equalities as everything is in Prop

-- ∀i and ∀e
∀i : {Γ : Con} → {F : For (Γ ▹t)} → (Γ ▹t) ⊢ F → Γ ⊢ (∀∀ F)
∀i p γ = λ t → p (γ ,× t)
∀e : {Γ : Con} → {F : For (Γ ▹t)} → Γ ⊢ (∀∀ F) → {t : Tm Γ} → Γ ⊢ ( F [(id {Γ}) ,t t ]f)
∀e p {t} γ = p γ (t γ)

tod : FFOL
tod = record
          { Con = Con
          ; Sub = Sub
          ; _∘_ = _∘_
          ; ∘-ass = refl
          ; id = id
          ; idl = refl
          ; idr = refl
          ; ◇ = ⊤ₛ
          ; ε = ε
          ; ε-u = refl
          ; Tm = Tm
          ; _[_]t = _[_]t
          ; []t-id = []t-id
          ; []t-∘ = λ {Γ} {Δ} {Ξ} {α} {β} {t} → []t-∘ {Γ} {Δ} {Ξ} {α} {β} {t}
          ; _▹t = _▹t
          ; πt¹ = πt¹
          ; πt² = πt²
          ; _,t_ = _,t_
          ; πt²∘,t = λ {Γ} {Δ} {σ} → πt²∘,t {Γ} {Δ} {σ}
          ; πt¹∘,t = λ {Γ} {Δ} {σ} {t} → πt¹∘,t {Γ} {Δ} {σ} {t}
          ; ,t∘πt = ,t∘πt
          ; ,t∘ = λ {Γ} {Δ} {Ξ} {σ} {δ} {t} → ,t∘ {Γ} {Δ} {Ξ} {σ} {δ} {t}
          ; For = For
          ; _[_]f = _[_]f
          ; []f-id = []f-id
          ; []f-∘ = λ {Γ} {Δ} {Ξ} {α} {β} {F} → []f-∘ {Γ} {Δ} {Ξ} {α} {β} {F}
          ; _⊢_ = _⊢_
          ; _[_]p = _[_]p
          ; _▹p_ = _▹p_
          ; πp¹ = πp¹
          ; πp² = πp²
          ; _,p_ = _,p_
          ; ,p∘πp = ,p∘πp
          ; πp¹∘,p = λ {Γ} {Δ} {F} {σ} {p} → πp¹∘,p {Γ} {Δ} {F} {σ} {p}
          ; ,p∘ = λ {Γ} {Δ} {Ξ} {σ} {δ} {F} {prf} → ,p∘ {Γ} {Δ} {Ξ} {σ} {δ} {F} {prf}
          ; _⇒_ = _⇒_
          ; []f-⇒ = λ {Γ} {F} {G} {σ} → []f-⇒ {Γ} {F} {G} {σ}
          ; ∀∀ = ∀∀
          ; []f-∀∀ = λ {Γ} {Δ} {F} {σ} → []f-∀∀ {Γ} {Δ} {F} {σ}
          ; lam = lam
          ; app = app
          ; ∀i = ∀i
          ; ∀e = ∀e
          ; R = R
          ; R[] = λ {Γ} {Δ} {σ} {t} {u} →  R[] {Γ} {Δ} {σ} {t} {u}
          }
```

```
    -- (∀ x ∀ y . A(x,y)) ⇒ ∀ y ∀ x . A(y,x)
    -- both sides are ∀ ∀ A (0,1)
    ex1 : {A : For (Tₛ ▹ₜ ▹ₜ)} → Tₛ ⊢ ((∀∀ (∀∀ A)) ⇒ (∀∀ (∀∀ A)))
    ex1 _ hyp = hyp
    -- (A ⇒ ∀ x . B(x)) ⇒ ∀ x . A ⇒ B(x)
    ex2 : {A : For Tₛ} → {B : For (Tₛ ▹ₜ)} → Tₛ ⊢ ((A ⇒ (∀∀ B)) ⇒ (∀∀ ((A [ πₜ¹ id ]f) ⇒ B)))
    ex2 _ h t h' = h h' t
    -- ∀ x y . A(x,y) ⇒ ∀ x . A(x,x)
    -- For simplicity, I swiched positions of parameters of A (somehow...)
    ex3 : {A : For (Tₛ ▹ₜ ▹ₜ)} → Tₛ ⊢ ((∀∀ (∀∀ A)) ⇒ (∀∀ (A [ id ,ₜ (πₜ² id) ]f)))
    ex3 _ h t = h t t
    -- ∀ x . A (x) ⇒ ∀ x y . A(x)
    ex4 : {A : For (Tₛ ▹ₜ)} → Tₛ ⊢ ((∀∀ A) ⇒ (∀∀ (∀∀ (A [ ε ,ₜ (πₜ² (πₜ¹ id)) ]f))))
    ex4 {A} ◇◇ x t t' = x t
    -- (((∀ x . A (x)) ⇒ B)⇒ B) ⇒ ∀ x . ((A (x) ⇒ B) ⇒ B)
    ex5 : {A : For (Tₛ ▹ₜ)} → {B : For Tₛ} → Tₛ ⊢ ((((∀∀ A) ⇒ B) ⇒ B) ⇒ (∀∀ ((A ⇒ (B [ πₜ¹ id ]f)) ⇒ (B [ πₜ¹ id ]f))))
    ex5 ◇◇ h t h' = h (λ h'' → h' (h'' t))
\end{code}
```

```
git s\begin{code}
{-# OPTIONS --prop --rewriting #-}

open import PropUtil

module FFOLInitial where

  open import FFOL
  open import Agda.Primitive
  open import ListUtil

  {-- TERM CONTEXTS - TERMS - FORMULAE - TERM SUBSTITUTIONS --}

  --# Term contexts are isomorphic to Nat
  data Cont : Set₁ where
    ◇t : Cont
    _▹t⁰ : Cont → Cont

  --#
  variable
    Γₜ Δₜ Ξₜ : Cont

  --# A term variable is a de-bruijn variable, TmVar n ≈ ⟦0,n-1⟧
  data TmVar : Cont → Set₁ where
    tvzero : TmVar (Γₜ ▹t⁰)
    tvnext : TmVar Γₜ → TmVar (Γₜ ▹t⁰)

  -- For now, we only have term variables (no function symbol)
  data Tm : Cont → Set₁ where
    var : TmVar Γₜ → Tm Γₜ

  --# Now we can define formulæ
  data For : Cont → Set₁ where
    R : Tm Γₜ → Tm Γₜ → For Γₜ
    _⇒_ : For Γₜ → For Γₜ → For Γₜ
    ∀∀  : For (Γₜ ▹t⁰) → For Γₜ




  --# Then we define term substitutions
  data Subt : Cont → Cont → Set₁ where
    εₜ : Subt Γₜ ◇t
    _,ₜ_ : Subt Δₜ Γₜ → Tm Δₜ → Subt Δₜ (Γₜ ▹t⁰)

  --# We write down the access functions from the algebra, in restricted versions
  πₜ¹ : Subt Δₜ (Γₜ ▹t⁰) → Subt Δₜ Γₜ
  πₜ¹ (σₜ ,ₜ t) = σₜ
  πₜ² : Subt Δₜ (Γₜ ▹t⁰) → Tm Δₜ
  πₜ² (σₜ ,ₜ t) = t
  -- And their equalities (the fact that there are reciprocical)
  πₜ²∘,ₜ : {σₜ : Subt Δₜ Γₜ} → {t : Tm Δₜ} → πₜ² (σₜ ,ₜ t) ≡ t
  πₜ²∘,ₜ = refl
  πₜ¹∘,ₜ : {σₜ : Subt Δₜ Γₜ} → {t : Tm Δₜ} → πₜ¹ (σₜ ,ₜ t) ≡ σₜ
  πₜ¹∘,ₜ = refl
  ,ₜ∘πₜ : {σₜ : Subt Δₜ (Γₜ ▹t⁰)} → (πₜ¹ σₜ) ,ₜ (πₜ² σₜ) ≡ σₜ
  ,ₜ∘πₜ {σₜ = σₜ ,ₜ t} = refl


  --# We now define the action of term substitutions on terms
  _[_]t : Tm Γₜ → Subt Δₜ Γₜ → Tm Δₜ
  var tvzero [ σ ,ₜ t ]t = t
  var (tvnext tv) [ σ ,ₜ t ]t = var tv [ σ ]t
  --# We define weakenings of the term-context for terms
  -- «A term of n variables can be seen as a term of n+1 variables»
  wkₜt : Tm Γₜ → Tm (Γₜ ▹t⁰)
  wkₜt (var tv) = var (tvnext tv)

  -- From a substitution into n variables, we get a substitution into n+1 variables
which don't use the last one
  wkₜσₜ : Subt Δₜ Γₜ → Subt (Δₜ ▹t⁰) Γₜ
  wkₜσₜ εₜ = εₜ
  wkₜσₜ (σ ,ₜ t) = (wkₜσₜ σ) ,ₜ (wkₜt t)

  -- From a substitution into n variables, we construct a substitution from n+1
variables to n+1 variables which maps it to itself
  -- i.e. 0 -> 0 and for all i ->(old) σ(i) we get i+1 -> σ(i)+1
  lftₜσₜ : Subt Δₜ Γₜ → Subt (Δₜ ▹t⁰) (Γₜ ▹t⁰)
  lftₜσₜ σ = (wkₜσₜ σ) ,ₜ (var tvzero)

  -- We show how wkₜt and interacts with [_]t
  wkₜ[]t : {α : Subt Δₜ Γₜ} → {t : Tm Γₜ} →  wkₜt (t [ α ]t) ≡ (wkₜt t [ lftₜσₜ α ]t)
  wkₜ[]t {α = α ,ₜ t} {var tvzero} = refl
  wkₜ[]t {α = α ,ₜ t} {var (tvnext tv)} = wkₜ[]t {t = var tv}

  --# We can now subst on formulæ
  _[_]f : For Γₜ → Subt Δₜ Γₜ → For Δₜ
  (R t u) [ σ ]f = R (t [ σ ]t) (u [ σ ]t)
  (A ⇒ B) [ σ ]f = (A [ σ ]f) ⇒ (B [ σ ]f)
  (∀∀ A) [ σ ]f = ∀∀ (A [ lftₜσₜ σ ]f)




  --# We now can define identity and composition of term substitutions
  idₜ : Subt Γₜ Γₜ
  idₜ {◇t} = εₜ
  idₜ {Γₜ ▹t⁰} = lftₜσₜ (idₜ {Γₜ})
  _∘ₜ_ : Subt Δₜ Γₜ → Subt Ξₜ Δₜ → Subt Ξₜ Γₜ
  εₜ ∘ₜ β = εₜ
  (α ,ₜ x) ∘ₜ β = (α ∘ₜ β) ,ₜ (x [ β ]t)

  --# We now have to show all their equalities (idₜ and ∘ₜ respect []t, []f, wkₜ,
lftₜ, categorical rules

  -- Substitution for terms
  []t-id : {t : Tm Γₜ} → t [ idₜ {Γₜ} ]t ≡ t
  []t-id {Γₜ ▹t⁰} {var tvzero} = refl
  []t-id {Γₜ ▹t⁰} {var (tvnext tv)} = substP (λ t → t ≡ var (tvnext tv)) (wkₜ[]t {t
= var tv}) (substP (λ t → wkₜt t ≡ var (tvnext tv)) (≡sym ([]t-id {t = var tv}))
refl)
  []t-∘ : {α : Subt Ξₜ Δₜ} → {β : Subt Δₜ Γₜ} → {t : Tm Γₜ} → t [ β ∘ₜ α ]t ≡ (t [ β
]t) [ α ]t
  []t-∘ {α = α} {β = β ,ₜ t} {t = var tvzero} = refl
  []t-∘ {α = α} {β = β ,ₜ t} {t = var (tvnext tv)} = []t-∘ {t = var tv}

  -- Weakenings and liftings of substitutions
  wkₜσₜ-∘ₜl : {α : Subt Ξₜ Δₜ} → {β : Subt Δₜ Γₜ} → wkₜσₜ (β ∘ₜ α) ≡ (wkₜσₜ β ∘ₜ lftₜσₜ α)
  wkₜσₜ-∘ₜl {β = εₜ} = refl
  wkₜσₜ-∘ₜl {β = β ,ₜ t} = cong₂ _,ₜ_ wkₜσₜ-∘ₜl (wkₜ[]t {t = t})
  wkₜσₜ-∘ₜr : {α : Subt Γₜ Δₜ} → {β : Subt Ξₜ Γₜ} → α ∘ₜ (wkₜσₜ β) ≡ wkₜσₜ (α ∘ₜ β)
  wkₜσₜ-∘ₜr {α = εₜ} = refl
  wkₜσₜ-∘ₜr {α = α ,ₜ var tv} = cong₂ _,ₜ_ (wkₜσₜ-∘ₜr {α = α}) (≡sym (wkₜ[]t {t = var
tv}))
  lftₜσₜ-∘ : {α : Subt Ξₜ Δₜ} → {β : Subt Δₜ Γₜ} → lftₜσₜ (β ∘ₜ α) ≡ (lftₜσₜ β) ∘ₜ (lftₜσₜ
α)
  lftσₜ-∘ {α = α} {β = εₜ} = refl
  lftσₜ-∘ {α = α} {β = β ,ₜ t} = cong₂ _,ₜ_ (cong₂ _,ₜ_ wkₜσₜ-∘ₜl (wkₜ[]t {t = t}))
refl

  -- Cancelling a weakening with a ,ₜ
  wkₜ[,]t : {t : Tm Γₜ}{u : Tm Δₜ}{β : Subt Δₜ Γₜ} → (wkₜt t) [ β ,ₜ u ]t ≡ t [ β ]t
  wkₜ[,]t {t = var tvzero} = refl
  wkₜ[,]t {t = var (tvnext tv)} = refl
  wkₜ∘ₜ,ₜ : {α : Subt Γₜ Δₜ}{β : Subt Ξₜ Γₜ}{t : Tm Ξₜ} → (wkₜσₜ α) ∘ₜ (β ,ₜ t) ≡ (α ∘ₜ
β)
  wkₜ∘ₜ,ₜ {α = εₜ} = refl
  wkₜ∘ₜ,ₜ {α = α ,ₜ t} {β = β} = cong₂ _,ₜ_ (wkₜ∘ₜ,ₜ {α = α}) (wkₜ[,]t {t = t} {β =
β})

  -- Categorical rules are respected by idₜ and ∘ₜ
  idlₜ : {α : Subt Δₜ Γₜ} → idₜ ∘ₜ α ≡ α
  idlₜ {α = εₜ} = refl
  idlₜ {α = α ,ₜ x} = cong₂ _,ₜ_ (≡tran wkₜ∘ₜ,ₜ idlₜ) refl
  idrₜ : {α : Subt Δₜ Γₜ} → α ∘ₜ idₜ ≡ α
  idrₜ {α = εₜ} = refl
  idrₜ {α = α ,ₜ x} = cong₂ _,ₜ_ idrₜ []t-id
  ∘ₜ-ass : {Γₜ Δₜ Ξₜ Ψₜ : Cont}{α : Subt Γₜ Δₜ}{β : Subt Δₜ Ξₜ}{γ : Subt Ξₜ Ψₜ} → (γ ∘ₜ
β) ∘ₜ α ≡ γ ∘ₜ (β ∘ₜ α)
  ∘ₜ-ass {α = α} {β} {εₜ} = refl
  ∘ₜ-ass {α = α} {β} {γ ,ₜ x} = cong₂ _,ₜ_ ∘ₜ-ass (≡sym ([]t-∘ {t = x}))

  -- Unicity of the terminal morphism
  εₜ-u : {σₜ : Subt Γₜ ◇t} → σₜ ≡ εₜ
  εₜ-u {σₜ = εₜ} = refl

  -- Substitution for formulæ
  []f-id : {F : For Γₜ} → F [ idₜ {Γₜ} ]f ≡ F
  []f-id {F = R t u} = cong₂ R []t-id []t-id
  []f-id {F = F ⇒ G} = cong₂ _⇒_ []f-id []f-id
  []f-id {F = ∀∀ F} = cong ∀∀ []f-id
  []f-∘ : {α : Subt Ξₜ Δₜ} → {β : Subt Δₜ Γₜ} → {F : For Γₜ} → F [ β ∘ₜ α ]f ≡ (F [
β ]f) [ α ]f
  []f-∘ {α = α} {β = β} {F = R t u} = cong₂ R ([]t-∘ {α = α} {β = β} {t = t})
([]t-∘ {α = α} {β = β} {t = u})
  []f-∘ {F = F ⇒ G} = cong₂ _⇒_ []f-∘ []f-∘
  []f-∘ {F = ∀∀ F} = cong ∀∀ (≡tran (cong (λ σ → F [ σ ]f) lftσₜ-∘) []f-∘)

  -- Substitution for formulæ constructors
  -- we omit []f-R and []f-⇒ as they are directly refl
  []f-∀∀ : {A : For (Γₜ ▹t⁰)} → {σₜ : Subt Δₜ Γₜ} → (∀∀ A) [ σₜ ]f ≡ (∀∀ (A [ (σₜ ∘ₜ
πₜ¹ idₜ) ,ₜ πₜ² idₜ ]f))
  []f-∀∀ {A = A} = cong ∀∀ (cong (_[_]f A) (cong₂ _,ₜ_ (≡tran (cong wkₜσₜ (≡sym
idrₜ)) (≡sym wkₜσₜ-∘ₜr)) refl))




  --# We can now define proof contexts, which are indexed by a term context
  -- i.e. we know which terms a proof context can use
  data Conp : Cont → Set₁ where
    ◇p : Conp Γₜ
    _▹p⁰_ : Conp Γₜ → For Γₜ → Conp Γₜ
  --#
  variable
    Γₚ Γₚ' : Conp Γₜ
    Δₚ Δₚ' : Conp Δₜ
    Ξₚ Ξₚ' : Conp Ξₜ

  --# The actions of Subt's is extended to contexts
  _[_]c : Conp Γₜ → Subt Δₜ Γₜ → Conp Δₜ
  ◇ₚ [ σₜ ]c = ◇ₚ
  (Γₚ ⊳p⁰ A) [ σₜ ]c = (Γₚ [ σₜ ]c) ⊳p⁰ (A [ σₜ ]f)
  --# This Conp is indeed a functor
  []c-id : Γₚ [ idₜ ]c ≡ Γₚ
  []c-id {Γₚ = ◇ₚ} = refl
  []c-id {Γₚ = Γₚ ⊳p⁰ x} = cong₂ _⊳p⁰_ []c-id []f-id
  []c-∘ : {α : Subt Δₜ Ξₜ} {β : Subt Γₜ Δₜ} {Ξₚ : Conp Ξₜ} → Ξₚ [ α ∘ₜ β ]c ≡ (Ξₚ [ α
]c) [ β ]c
  []c-∘ {α = α} {β = β} {◇ₚ} = refl
  []c-∘ {α = α} {β = β} {Ξₚ ⊳p⁰ A} = cong₂ _⊳p⁰_ []c-∘ []f-∘


  --# We can also add a term that will not be used in the formulæ already present
  -- (that's why we use wkₜσₜ)
  _⊳tp : Conp Γₜ → Conp (Γₜ ⊳t⁰)
  Γ ⊳tp = Γ [ wkₜσₜ idₜ ]c
  --# We show how it interacts with ,ₜ and lftₜσₜ
  ⊳tp,ₜ : {σₜ : Subt Γₜ Δₜ}{t : Tm Γₜ} → (Γₚ ⊳tp) [ σₜ ,ₜ t ]c ≡  Γₚ [ σₜ ]c
  ⊳tp,ₜ {Γₚ = Γₚ} = ≡tran (≡sym []c-∘) (cong (λ ξ → Γₚ [ ξ ]c) (≡tran wkₜ∘ₜ,ₜ idlₜ))
  ⊳tp-lftₜ : {σ : Subt Δₜ Γₜ} → ((Δₚ ⊳tp) [ lftₜσₜ σ ]c) ≡ ((Δₚ [ σ ]c) ⊳tp)
  ⊳tp-lftₜ {Δₚ = Δₚ} = ≡tran² (≡sym []c-∘) (cong (λ ξ → Δₚ [ ξ ]c) (≡tran² (≡sym
wkₜσₜ-∘ₜl) (cong wkₜσₜ (≡tran idlₜ (≡sym idrₜ))) (≡sym wkₜσₜ-∘ₜr))) []c-∘



  --# With those contexts, we have everything to define proofs
  data PfVar : (Γₜ : Cont) → (Γₚ : Conp Γₜ) → For Γₜ → Prop₁ where
    pvzero : {A : For Γₜ} → PfVar Γₜ (Γₚ ⊳p⁰ A) A
    pvnext : {A B : For Γₜ} → PfVar Γₜ Γₚ A → PfVar Γₜ (Γₚ ⊳p⁰ B) A

  data Pf : (Γₜ : Cont) → (Γₚ : Conp Γₜ) → For Γₜ → Prop₁ where
    var : {A : For Γₜ} → PfVar Γₜ Γₚ A → Pf Γₜ Γₚ A
    app : {A B : For Γₜ} → Pf Γₜ Γₚ (A ⇒ B) → Pf Γₜ Γₚ A → Pf Γₜ Γₚ B
    lam : {A B : For Γₜ} → Pf Γₜ (Γₚ ⊳p⁰ A) B → Pf Γₜ Γₚ (A ⇒ B)
    p∀∀e : {A : For (Γₜ ⊳t⁰)} → {t : Tm Γₜ} → Pf Γₜ Γₚ (∀∀ A) → Pf Γₜ Γₚ (A [ idₜ ,ₜ
t ]f)
    p∀∀i : {A : For (Γₜ ⊳t⁰)} → Pf (Γₜ ⊳t⁰) (Γₚ ⊳tp) A → Pf Γₜ Γₚ (∀∀ A)


  --# The action on Cont's morphisms of Pf functor
  _[_]pvₜ : {A : For Δₜ}→ PfVar Δₜ Δₚ A → (σ : Subt Γₜ Δₜ)→ PfVar Γₜ (Δₚ [ σ ]c) (A [
σ ]f)
  pvzero [ σ ]pvₜ = pvzero
  pvnext pv [ σ ]pvₜ = pvnext (pv [ σ ]pvₜ)
  _[_]pₜ : {A : For Δₜ} → Pf Δₜ Δₚ A → (σ : Subt Γₜ Δₜ) → Pf Γₜ (Δₚ [ σ ]c) (A [ σ
]f)
  var pv [ σ ]pₜ = var (pv [ σ ]pvₜ)
  app pf pf' [ σ ]pₜ = app (pf [ σ ]pₜ) (pf' [ σ ]pₜ)
  lam pf [ σ ]pₜ = lam (pf [ σ ]pₜ)
  _[_]pₜ {Δₚ = Δₚ} {Γₜ = Γₜ} (p∀∀e {A = A} {t = t} pf) σ =
    substP (λ F → Pf Γₜ (Δₚ [ σ ]c) F) (≡tran² (≡sym []f-∘) (cong (λ σ → A [ σ ]f)
    (cong₂ _,ₜ_ (≡tran² wkₜ∘ₜ,ₜ idrₜ (≡sym idlₜ)) refl)) ([]f-∘))
    (p∀∀e {t = t [ σ ]t} (pf [ σ ]pₜ))
  _[_]pₜ {Γₜ = Γₜ} (p∀∀i pf) σ
    = p∀∀i (substP (λ Ξₚ → Pf (Γₜ ⊳t⁰) (Ξₚ) _) ⊳tp-lftₜ (pf [ lftₜσₜ σ ]pₜ))
  --# We now can create Renamings, a subcategory from (Conp,Subp) that
  -- A renaming from a context Γₚ to a context Δₚ means when they are seen
  -- as lists, that every element of Γₚ is an element of Δₚ
  -- In other words, we can prove Γₚ from Δₚ using only proof variables (var)
  data Ren : Conp Γₜ → Conp Γₜ → Set₁ where
    zeroRen : Ren ◇p Γₚ
    leftRen : {A : For Δₜ} → PfVar Δₜ Δₚ A → Ren Δₚ' Δₚ → Ren (Δₚ' ▹p⁰ A) Δₚ

  --# We now show how we can extend renamings
  rightRen :{A : For Δₜ} → Ren Γₚ Δₚ → Ren Γₚ (Δₚ ▹p⁰ A)
  rightRen zeroRen = zeroRen
  rightRen (leftRen x h) = leftRen (pvnext x) (rightRen h)
  bothRen : {A : For Γₜ} → Ren Γₚ Δₚ → Ren (Γₚ ▹p⁰ A) (Δₚ ▹p⁰ A)
  bothRen zeroRen = leftRen pvzero zeroRen
  bothRen (leftRen x h) = leftRen pvzero (leftRen (pvnext x) (rightRen h))
  reflRen : Ren Γₚ Γₚ
  reflRen {Γₚ = ◇p} = zeroRen
  reflRen {Γₚ = Γₚ ▹p⁰ x} = bothRen reflRen

  -- We can extend renamings with term variables
  PfVar▹tp : {A : For Δₜ} → PfVar Δₜ Δₚ A → PfVar (Δₜ ▹t⁰) (Δₚ ▹tp) (A [ wkₜσₜ idₜ ]f)
  PfVar▹tp pvzero = pvzero
  PfVar▹tp (pvnext x) = pvnext (PfVar▹tp x)
  Ren▹tp : Ren Γₚ Δₚ → Ren (Γₚ ▹tp) (Δₚ ▹tp)
  Ren▹tp zeroRen = zeroRen
  Ren▹tp (leftRen x s) = leftRen (PfVar▹tp x) (Ren▹tp s)

  -- Renamings can be used to (strongly) weaken proofs
  wkrpv : {A : For Δₜ} → Ren Δₚ' Δₚ → PfVar Δₜ Δₚ' A → PfVar Δₜ Δₚ A
  wkrpv (leftRen x x₁) pvzero = x
  wkrpv (leftRen x x₁) (pvnext s) = wkrpv x₁ s
  wkrp : {A : For Δₜ} → Ren Δₚ Δₚ' → Pf Δₜ Δₚ A → Pf Δₜ Δₚ' A
  wkrp s (var pv) = var (wkrpv s pv)
  wkrp s (app pf pf₁) = app (wkrp s pf) (wkrp s pf₁)
  wkrp s (lam {A = A} pf) = lam (wkrp (bothRen s) pf)
  wkrp s (p∀∀e pf) = p∀∀e (wkrp s pf)
  wkrp s (p∀∀i pf) = p∀∀i (wkrp (Ren▹tp s) pf)


  --# But we need something stronger than just renamings
  -- introducing: Proof substitutions
  -- They are basicly a list of proofs for the formulæ contained in
  -- the goal context.
  -- It is not defined between all contexts, only those with the same term
context
  data Subp : {Δₜ : Cont} → Conp Δₜ → Conp Δₜ → Prop₁ where
    εₚ : Subp Δₚ ◇p
    _,ₚ_ : {A : For Δₜ} → (σ : Subp Δₚ Δₚ') → Pf Δₜ Δₚ A → Subp Δₚ (Δₚ' ▹p⁰ A)

  --# We write down the access functions from the algebra, in restricted versions
  πₚ¹ : ∀{Γₜ}{Γₚ Δₚ : Conp Γₜ} {A : For Γₜ} → Subp Δₚ (Γₚ ▹p⁰ A) → Subp Δₚ Γₚ
  πₚ¹ (σₚ ,ₚ pf) = σₚ
  πₚ² : ∀{Γₜ}{Γₚ Δₚ : Conp Γₜ} {A : For Γₜ} → Subp Δₚ (Γₚ ▹p⁰ A) → Pf Γₜ Δₚ A
  πₚ² (σₚ ,ₚ pf) = pf

  --# The action of Cont's morphisms on Subp
  _[_]σₚ : Subp {Δₜ} Δₚ Δₚ' → (σ : Subt Γₜ Δₜ) → Subp {Γₜ} (Δₚ [ σ ]c) (Δₚ' [ σ ]c)
  εₚ [ σₜ ]σₚ = εₚ
  (σₚ ,ₚ pf) [ σₜ ]σₚ = (σₚ [ σₜ ]σₚ) ,ₚ (pf [ σₜ ]pₜ)

  --# They are indeed stronger than renamings
  Ren→Sub : Ren Δₚ Δₚ' → Subp {Δₜ} Δₚ' Δₚ
  Ren→Sub zeroRen = εₚ
  Ren→Sub (leftRen x s) = Ren→Sub s ,ₚ var x


  -- From a substition into n variables, we get a substitution into n+1 variables
which don't use the last one
  wkₚσₚ : {Δₜ : Cont} {Δₚ Γₚ : Conp Δₜ}{A : For Δₜ} → Subp {Δₜ} Δₚ Γₚ → Subp {Δₜ} (Δₚ
▹p⁰ A) Γₚ
  wkₚσₚ εₚ = εₚ
  wkₚσₚ (σₚ ,ₚ pf) = (wkₚσₚ σₚ) ,ₚ wkrₚ (rightRen reflRen) pf

  -- From a substitution into n variables, we construct a substitution from n+1
variables to n+1 variables which maps it to itself
  -- i.e. 0 -> 0 and for all i ->(old) σ(i) we get i+1 -> σ(i)+1
  lfₚσₚ : {Δₜ : Cont}{Δₚ Γₚ : Conp Δₜ}{A : For Δₜ} → Subp {Δₜ} Δₚ Γₚ → Subp {Δₜ} (Δₚ
▹p⁰ A) (Γₚ ▹p⁰ A)
  lfₚσₚ σ = (wkₚσₚ σ) ,ₚ (var pvzero)










  wkₜσₚ : Subp {Δₜ} Δₚ' Δₚ → Subp {Δₜ ▹t⁰} (Δₚ' ▹tp) (Δₚ ▹tp)
  wkₜσₚ εₚ = εₚ
  wkₜσₚ {Δₜ = Δₜ} (_,ₚ_ {A = A} σₚ pf) = (wkₜσₚ σₚ) ,ₚ substP (λ Ξₚ → Pf (Δₜ ▹t⁰) Ξₚ
(A [ wkₜσₜ idₜ ]f)) refl (_[_]pₜ {Γₜ = Δₜ ▹t⁰} pf (wkₜσₜ idₜ))

  --#
  _[_]p : {A : For Δₜ} → Pf Δₜ Δₚ A → (σ : Subp {Δₜ} Δₚ' Δₚ) → Pf Δₜ Δₚ' A
  var pvzero [ σ ,ₚ pf ]p = pf
  var (pvnext pv) [ σ ,ₚ pf ]p = var pv [ σ ]p
  app pf pf₁ [ σ ]p = app (pf [ σ ]p) (pf₁ [ σ ]p)
  lam pf [ σ ]p = lam (pf [ wkₚσₚ σ ,ₚ var pvzero ]p)
  p∀∀e pf [ σ ]p = p∀∀e (pf [ σ ]p)
  p∀∀i pf [ σ ]p = p∀∀i (pf [ wkₜσₚ σ ]p)







  --# We can now define identity and composition on proof substitutions
  idₚ : Subp {Δₜ} Δₚ Δₚ
  idₚ {Δₚ = ◇p} = εₚ
  idₚ {Δₚ = Δₚ ▹p⁰ x} = lfₚσₚ (idₚ {Δₚ = Δₚ})
  _∘ₚ_ : {Γₚ Δₚ Ξₚ : Conp Δₜ} → Subp {Δₜ} Δₚ Ξₚ → Subp {Δₜ} Γₚ Δₚ → Subp {Δₜ} Γₚ Ξₚ
  εₚ ∘ₚ β = εₚ
  (α ,ₚ pf) ∘ₚ β = (α ∘ₚ β) ,ₚ (pf [ β ]p)
  --# We can now merge the two notions of contexts, substitutions, and everything
  record Con : Set₁ where
    constructor con
    field
      t : Cont
      p : Conp t

  --#
  variable
    Γ Δ Ξ : Con

  --#
  record Sub (Γ : Con) (Δ : Con) : Set₁ where
    constructor sub
    field
      t : Subt (Con.t Γ) (Con.t Δ)
      p : Subp {Con.t Γ} (Con.p Γ) ((Con.p Δ) [ t ]c)

  --# We need this to apply term-substitution theorems to global substitutions
  sub= : {Γ Δ : Con}{σₜ σₜ' : Subt (Con.t Γ) (Con.t Δ)} →
    σₜ ≡ σₜ' →
    {σₚ : Subp {Con.t Γ} (Con.p Γ) ((Con.p Δ) [ σₜ ]c)}
    {σₚ' : Subp {Con.t Γ} (Con.p Γ) ((Con.p Δ) [ σₜ' ]c)} →
    sub σₜ σₚ ≡ sub σₜ' σₚ'
  sub= refl = refl

  --# (Con,Sub) is a category with an initial object
  id : Sub Γ Γ
  id {Γ} = sub idₜ (substP (Subp _) (≡sym []c-id) idₚ)
  _∘_ : Sub Δ Ξ → Sub Γ Δ → Sub Γ Ξ
  sub αₜ αₚ ∘ sub βₜ βₚ = sub (αₜ ∘ₜ βₜ) (substP (Subp _) (≡sym []c-∘) (αₚ [ βₜ ]σₚ)
∘ₚ βₚ)


  --# We have our two context extension operators
  _▹t : Con → Con
  Γ ▹t = con ((Con.t Γ) ▹t⁰) (Con.p Γ ▹tp)
  _▹p_ : (Γ : Con) → For (Con.t Γ) → Con
  Γ ▹p A = con (Con.t Γ) (Con.p Γ ▹p⁰ A)

  --# We define the access function from the algebra, but defined for fully-
featured substitutions
  -- For term substitutions
  πₜ¹* : {Γ Δ : Con} → Sub Δ (Γ ▹t) → Sub Δ Γ
  πₜ¹* (sub (σₜ ,ₜ t) σₚ) = sub σₜ (substP (Subp _) ▹tp,ₜ σₚ)
  πₜ²* : {Γ Δ : Con} → Sub Δ (Γ ▹t) → Tm (Con.t Δ)
  πₜ²* (sub (σₜ ,ₜ t) σₚ) = t
  _,ₜ*_ : {Γ Δ : Con} → Sub Δ Γ → Tm (Con.t Δ) → Sub Δ (Γ ▹t)
  (sub σₜ σₚ) ,ₜ* t = sub (σₜ ,ₜ t) (substP (Subp _) (≡sym ▹tp,ₜ) σₚ)
  -- And the equations
  πₜ²∘,ₜ* : {Γ Δ : Con} → {σ : Sub Δ Γ} → {t : Tm (Con.t Δ)} → πₜ²* (σ ,ₜ* t) ≡ t
  πₜ²∘,ₜ* = refl
  πₜ¹∘,ₜ* : {Γ Δ : Con} → {σ : Sub Δ Γ} → {t : Tm (Con.t Δ)} → πₜ¹* (σ ,ₜ* t) ≡ σ
  πₜ¹∘,ₜ* {Γ}{Δ}{σ}{t} = sub= refl
  ,ₜ∘πₜ* : {Γ Δ : Con} → {σ : Sub Δ (Γ ▹t)} → (πₜ¹* σ) ,ₜ* (πₜ²* σ) ≡ σ
  ,ₜ∘πₜ* {Γ} {Δ} {sub (σₜ ,ₜ t) σₚ} = sub= refl
  ,ₜ∘* : {Γ Δ Ξ : Con}{σ : Sub Γ Ξ}{δ : Sub Δ Γ}{t : Tm (Con.t Γ)} → (σ ,ₜ* t) ∘ δ
≡ (σ ∘ δ) ,ₜ* (t [ Sub.t δ ]t)
  ,ₜ∘* {Γ} {Δ} {Ξ} {sub σₜ σₚ} {sub δₜ δₚ} {t} = sub= refl
  -- And for proof substitutions
  πₚ¹* : {Γ Δ : Con} {A : For (Con.t Γ)} → Sub Δ (Γ ⊳p A) → Sub Δ Γ
  πₚ¹* (sub σₜ σap) = sub σₜ (πₚ¹ σap)
  πₚ²* : {Γ Δ : Con} {F : For (Con.t Γ)} (σ : Sub Δ (Γ ⊳p F)) → Pf (Con.t Δ)
(Con.p Δ) (F [ Sub.t (πₚ¹* σ) ]f)
  πₚ²* (sub σₜ (σₚ ,ₚ pf)) = pf
  _,ₚ*_ : {Γ Δ : Con} {F : For (Con.t Γ)} (σ : Sub Δ Γ) → Pf (Con.t Δ) (Con.p Δ)
(F [ Sub.t σ ]f) → Sub Δ (Γ ⊳p F)
  sub σₜ σₚ ,ₚ* pf = sub σₜ (σₚ ,ₚ pf)
  -- And the equations
  ,ₚ∘πₚ : {Γ Δ : Con} → {F : For (Con.t Γ)} → {σ : Sub Δ (Γ ⊳p F)} → (πₚ¹* σ) ,ₚ*
(πₚ²* σ) ≡ σ
  ,ₚ∘πₚ {σ = sub σₜ (σₚ ,ₚ p)} = refl
  ,ₚ∘ : {Γ Δ Ξ : Con}{σ : Sub Γ Ξ}{δ : Sub Δ Γ}{F : For (Con.t Ξ)}{prf : Pf
(Con.t Γ) (Con.p Γ) (F [ Sub.t σ ]f)}
      → (σ ,ₚ* prf) ∘ δ ≡ (σ ∘ δ) ,ₚ* (substP (λ F → Pf (Con.t Δ) (Con.p Δ) F)
(≡sym []f-∘) ((prf [ Sub.t δ ]ₚₜ) [ Sub.p δ ]p))
  ,ₚ∘ {Γ}{Δ}{Ξ}{σ = sub σₜ σₚ} {sub δₜ δₚ} {F = A} {prf} = sub= refl

  -- and FINALLY, we compile everything into an implementation of the FFOL record

  ffol : FFOL {lsuc lzero} {lsuc lzero} {lsuc lzero} {lsuc lzero}
  ffol = record
    { Con = Con
    ; Sub = Sub
    ; _∘_ = _∘_
    ; ∘-ass = sub= ∘ₜ-ass
    ; id = id
    ; idl = sub= idlₜ
    ; idr = sub= idrₜ
    ; ◇ = con ◇t ◇p
    ; ε = sub εₜ εₚ
    ; ε-u = sub= εₜ-u
    ; Tm = λ Γ → Tm (Con.t Γ)
    ; _[_]t = λ t σ → t [ Sub.t σ ]t
    ; []t-id = []t-id
    ; []t-∘ = λ {Γ}{Δ}{Ξ}{α}{β}{t} → []t-∘ {α = Sub.t α} {β = Sub.t β} {t = t}
    ; _⊳ₜ = _⊳t
    ; πₜ¹ = πₜ¹*
    ; πₜ² = πₜ²*
    ; _,ₜ_ = _,ₜ*_
    ; πₜ²∘,ₜ = refl
    ; πₜ¹∘,ₜ = λ {Γ}{Δ}{σ}{t} → πₜ¹∘,ₜ* {Γ}{Δ}{σ}{t}
    ; ,ₜ∘πₜ = ,ₜ∘πₜ*
    ; ,ₜ∘ = λ {Γ}{Δ}{Ξ}{σ}{δ}{t} → ,ₜ∘* {Γ}{Δ}{Ξ}{σ}{δ}{t}
    ; For = λ Γ → For (Con.t Γ)
    ; _[_]f = λ A σ → A [ Sub.t σ ]f
    ; []f-id = []f-id
    ; []f-∘ = []f-∘
    ; R = R
    ; R[] = refl
    ; _⊢_ = λ Γ A → Pf (Con.t Γ) (Con.p Γ) A
    ; _[_]p = λ pf σ → (pf [ Sub.t σ ]ₚₜ) [ Sub.p σ ]p
    ; _⊳p_ = _⊳p_
    ; πₚ¹ = πₚ¹*
    ; πₚ² = πₚ²*
    ; _,ₚ_ = _,ₚ*_
    ; ,ₚ∘πₚ = ,ₚ∘πₚ
    ; πₚ¹∘,ₚ = refl
    ; ,ₚ∘ = λ {Γ}{Δ}{Ξ}{σ}{δ}{F}{prf} → ,ₚ∘ {Γ}{Δ}{Ξ}{σ}{δ}{F}{prf}
    ; _⇒_ = _⇒_
    ; []f-⇒ = refl
    ; ∀∀ = ∀∀
    ; []f-∀∀ = []f-∀∀
    ; lam = λ {Γ}{F}{G} pf → substP (λ H → Pf (Con.t Γ) (Con.p Γ) (F ⇒ H)) []f-id
(lam pf)
    ; app = app
    ; ∀i = p∀∀i
    ; ∀e = λ {Γ} {F} pf {t} → p∀∀e pf
    }


  -- We define normal and neutral forms
  data Ne : (Γₜ : Conₜ) → (Γₚ : Conₚ Γₜ) → For Γₜ → Prop₁
  data Nf : (Γₜ : Conₜ) → (Γₚ : Conₚ Γₜ) → For Γₜ → Prop₁
  data Ne where
    var : {A : For Γₜ} → PfVar Γₜ Γₚ A → Ne Γₜ Γₚ A
    app : {A B : For Γₜ} → Ne Γₜ Γₚ (A ⇒ B) → Nf Γₜ Γₚ A → Ne Γₜ Γₚ B
    p∀∀e : {A : For (Γₜ ▹t⁰)} → {t : Tm Γₜ} → Ne Γₜ Γₚ (∀∀ A) → Ne Γₜ Γₚ (A [ idₜ ,ₜ
t ]f)
  data Nf where
    R : {t u : Tm Γₜ} → Ne Γₜ Γₚ (R t u) → Nf Γₜ Γₚ (R t u)
    lam : {A B : For Γₜ} → Nf Γₜ (Γₚ ▹p⁰ A) B → Nf Γₜ Γₚ (A ⇒ B)
    p∀∀i : {A : For (Γₜ ▹t⁰)} → Nf (Γₜ ▹t⁰) (Γₚ ▹tp) A → Nf Γₜ Γₚ (∀∀ A)


  Pf* : (Γₜ : Conₜ) → Conₚ Γₜ → Conₚ Γₜ → Prop₁
  Pf* Γₜ Γₚ ◇p = ⊤
  Pf* Γₜ Γₚ (Γₚ' ▹p⁰ A) = (Pf* Γₜ Γₚ Γₚ') ∧ (Pf Γₜ Γₚ A)

  Sub→Pf* : {Γₜ : Conₜ} {Γₚ Γₚ' : Conₚ Γₜ} →  Subₚ {Γₜ} Γₚ Γₚ' → Pf* Γₜ Γₚ Γₚ'
  Sub→Pf* εₚ = tt
  Sub→Pf* (σₚ ,ₚ pf) = ⟨ (Sub→Pf* σₚ) , pf ⟩
  Pf*-id : {Γₜ : Conₜ} {Γₚ : Conₚ Γₜ} → Pf* Γₜ Γₚ Γₚ
  Pf*-id = Sub→Pf* idₚ

  Pf*▹p : {Γₜ : Conₜ}{Γₚ Γₚ' : Conₚ Γₜ}{A : For Γₜ} → Pf* Γₜ Γₚ Γₚ' → Pf* Γₜ (Γₚ ▹p⁰
A) Γₚ'
  Pf*▹p {Γₚ' = ◇p} s = tt
  Pf*▹p {Γₚ' = Γₚ' ▹p⁰ x} s = ⟨ (Pf*▹p (proj₁ s)) , (wkᵣₚ (rightRen reflRen) (proj₂
s)) ⟩
  Pf*▹tp : {Γₜ : Conₜ}{Γₚ Γₚ' : Conₚ Γₜ} → Pf* Γₜ Γₚ Γₚ' → Pf* (Γₜ ▹t⁰) (Γₚ ▹tp) (Γₚ'
▹tp)
  Pf*▹tp {Γₚ' = ◇p} s = tt
  Pf*▹tp {Γₚ' = Γₚ' ▹p⁰ A} s = ⟨ Pf*▹tp (proj₁ s) , (proj₂ s) [ wkₜσₜ idₜ ]pₜ ⟩

  Pf*Pf : {Γₜ : Conₜ} {Γₚ Γₚ' : Conₚ Γₜ} {A : For Γₜ} → Pf* Γₜ Γₚ Γₚ' → Pf Γₜ Γₚ' A →
Pf Γₜ Γₚ A
  Pf*Pf s (var pvzero) = proj₂ s
  Pf*Pf s (var (pvnext pv)) = Pf*Pf (proj₁ s) (var pv)
  Pf*Pf s (app p p') = app (Pf*Pf s p) (Pf*Pf s p')
  Pf*Pf s (lam p) = lam (Pf*Pf (⟨ (Pf*▹p s) , (var pvzero) ⟩) p)
  Pf*Pf s (p∀∀e p) = p∀∀e (Pf*Pf s p)
  Pf*Pf s (p∀∀i p) = p∀∀i (Pf*Pf (Pf*▹tp s) p)

  Pf*-∘ : {Γₜ : Conₜ} {Γₚ Δₚ Ξₚ : Conₚ Γₜ} → Pf* Γₜ Δₚ Ξₚ → Pf* Γₜ Γₚ Δₚ → Pf* Γₜ Γₚ
Ξₚ
  Pf*-∘ {Ξₚ = ◇p} α β = tt
  Pf*-∘ {Ξₚ = Ξₚ ▹p⁰ A} α β = ⟨ Pf*-∘ (proj₁ α) β , Pf*Pf β (proj₂ α) ⟩

  module InitialMorphism (M : FFOL {lsuc lzero} {lsuc lzero} {lsuc lzero} {lsuc
lzero} {lsuc lzero}) where
    {-# TERMINATING #-}

    mCon : Con → (FFOL.Con M)
    mCont ◇t = FFOL.◇ M
    mCont (Γt ▷t⁰) = FFOL._▷t M (mCont Γt)
    mTmT : {Γt : Cont} → Tm Γt → (FFOL.Tm M (mCont Γt))
    -- Zero is (πt² id)
    mTmT {Γt ▷t⁰} (var tvzero) = FFOL.πt² M (FFOL.id M)
    -- N+1 is wk[tm N]
    mTmT {Γt ▷t⁰} (var (tvnext tv)) = (FFOL._[_]t M (mTmT (var tv)) (FFOL.πt¹ M
(FFOL.id M)))

    mForT : {Γt : Cont} → (For Γt) → (FFOL.For M (mCont Γt))
    mForT (R t u) = FFOL.R M (mTmT t) (mTmT u)
    mForT (A ⇒ B) = FFOL._⇒_ M (mForT A) (mForT B)
    mForT {Γ} (∀∀ A) = FFOL.∀∀ M (mForT A)

    mSubt : {Δt : Cont}{Γt : Cont} → Subt Δt Γt → (FFOL.Sub M (mCont Δt) (mCont Γt))
    mSubt εt = FFOL.ε M
    mSubt (σt ,t t) = FFOL._,t_ M (mSubt σt) (mTmT t)




    mConp : {Γt : Cont} → Conp Γt → (FFOL.Con M)
    mForP : {Γt : Cont} {Γp : Conp Γt} → (For Γt) → (FFOL.For M (mConp Γp))
    mConp {Γt} ◇p = mCont Γt
    mConp {Γt} (Γp ▷p⁰ A) = FFOL._▷p_ M (mConp Γp) (mForP {Γp = Γp} A)
    mForP {Γt} {Γp = ◇p} A = mForT {Γt} A
    mForP {Γp = Γp ▷p⁰ B} A = FFOL._[_]f M (mForP {Γp = Γp} A) (FFOL.πp¹ M (FFOL.id
M))

    mTmP : {Γt : Cont}{Γp : Conp Γt} → Tm Γt → (FFOL.Tm M (mConp Γp))
    mTmP {Γt}{Γp = ◇p} t = mTmT {Γt} t
    mTmP {Γp = Γp ▷p⁰ x} t = FFOL._[_]t M (mTmP {Γp = Γp} t) (FFOL.πp¹ M (FFOL.id
M))

    mCon : Con → (FFOL.Con M)
    mCon Γ = mConp {Con.t Γ} (Con.p Γ)

    mFor : {Γ : Con} → (For (Con.t Γ)) → (FFOL.For M (mCon Γ))
    mFor {Γ} A = mForP {Con.t Γ} {Con.p Γ} A

    mTm : {Γ : Con} → Tm (Con.t Γ) → (FFOL.Tm M (mCon Γ))
    mTm {Γ} t = mTmP {Con.t Γ} {Con.p Γ} t

    e▷tT : {Γt : Cont} → mCont (Γt ▷t⁰) ≡ FFOL._▷t M (mCont Γt)
    e▷tT = refl
    e▷tP : {Γt : Cont}{Γp : Conp Γt} → mConp {Γt ▷t⁰} (Γp [ wktσt idt ]c) ≡ FFOL._▷t M
(mConp Γp)
    e▷tP {Γt = Γt} {Γp = ◇p} = e▷tT {Γt = Γt}
    e▷tP {Γp = Γp ▷p⁰ A} = {!!}
    e▷t : {Γ : Con} → mCon (con (Con.t Γ ▷t⁰) (Con.p Γ [ wktσt idt ]c)) ≡ FFOL._▷t M
(mCon Γ)
    e▷t {Γ} = e▷tP {Γt = Con.t Γ} {Γp = Con.p Γ}

    mForT⇒ : {Γt : Cont}{A B : For Γt} → mForT {Γt} (A ⇒ B) ≡ FFOL._⇒_ M (mForT
{Γt} A) (mForT {Γt} B)
    mForT⇒ = refl
    mForP⇒ : {Γt : Cont}{Γp : Conp Γt}{A B : For Γt} → mForP {Γt} {Γp} (A ⇒ B) ≡
FFOL._⇒_ M (mForP {Γt} {Γp} A) (mForP {Γt} {Γp} B)
    mForP⇒ {Γt} {Γp = ◇p}{A}{B} = mForT⇒ {Γt}{A}{B}
    mForP⇒ {Γp = Γp ▷p⁰ C}{A}{B} = ≡tran (cong (λ X → (M FFOL.[ X ]f) _) (mForP⇒
{Γp = Γp})) (FFOL.[]f-⇒ M {F = mForP {Γp = Γp} A} {G = mForP {Γp = Γp} B} {σ =
(FFOL.πp¹ M (FFOL.id M))})
    mFor⇒ : {Γ : Con}{A B : For (Con.t Γ)} → mFor {Γ} (A ⇒ B) ≡ FFOL._⇒_ M (mFor
{Γ} A) (mFor {Γ} B)
    mFor⇒ {Γ} = mForP⇒ {Con.t Γ} {Con.p Γ}

    mForT∀∀ : {Γₜ : Contₜ}{A : For (Γₜ ▷t⁰)} → mForT {Γₜ} (∀∀ A) ≡ FFOL.∀∀ M (mForT
{Γₜ ▷t⁰} A)
    mForT∀∀ = refl
    mForP∀∀ : {Γₜ : Contₜ}{Γₚ : Conp Γₜ}{A : For (Γₜ ▷t⁰)} → mForP {Γₜ} {Γₚ} (∀∀ A) ≡
FFOL.∀∀ M (subst (FFOL.For M) (e▷tP {Γₜ} {Γₚ}) (mForP {Γₜ ▷t⁰} {Γₚ ▷tp} A))
    mForP∀∀ = {!!}
    -- mFor∀∀ : {Γ : Con}{A : For ((Con.t Γ) ▷t⁰)} → mFor {Γ} (∀∀ A) ≡ FFOL.∀∀ M
(mFor {Γ ▷t} A)

    --mForL : {Γ : Con}{A : For (Con.t Γ ▷t⁰)}{t : Tm (Con.t Γ)} → FFOL._[_]f M
(mFor {Γ = {!Γ ▷t!}} A) (FFOL._,ₜ_ M (FFOL.id M) (mTm {Γ = Γ} t)) ≡ mFor {Γ = Γ}
(A [ idₜ ,ₜ t ]f)

    m⊢ : {Γ : Con} {A : For (Con.t Γ)} → Pf (Con.t Γ) (Con.p Γ) A → FFOL._⊢_ M
(mCon Γ) (mFor {Γ = Γ} A)
    m⊢ (var pvzero) = FFOL.πₚ² M (FFOL.id M)
    m⊢ (var (pvnext pv)) = FFOL._[_]p M (m⊢ (var pv)) (FFOL.πₚ¹ M (FFOL.id M))
    m⊢ {Γ} {B} (app {A = A} pf pf') = FFOL.app M (substP (FFOL._⊢_ M _) (mFor⇒
{Γ}{A}{B}) (m⊢ pf)) (m⊢ pf')
    m⊢ {Γ} {A ⇒ B} (lam pf) = substP (FFOL._⊢_ M _) (≡sym (mFor⇒ {Γ}{A}{B}))
(FFOL.lam M (m⊢ pf))
    m⊢ {Γ} (p∀∀e {A = A} {t = t} pf) = substP (FFOL._⊢_ M _) {!!} (FFOL.∀e M {F
= mFor {{!!}} A} (substP (FFOL._⊢_ M _) {!!} (m⊢ pf)) {t = mTm {Γ} t})
    m⊢ {Γ} (p∀∀i {A = A} pf) = substP (FFOL._⊢_ M _) (≡sym mForP∀∀) (FFOL.∀i M
(substP (λ Ξ → FFOL._⊢_ M Ξ (mFor A)) e▷t (m⊢ pf)))


    mSubp : {Γₜ : Contₜ}{Δₚ Γₚ : Conp Γₜ} → Subp {Γₜ} Δₚ Γₚ → (FFOL.Sub M (mConp Δₚ)
(mConp Γₚ))
    mSubp {Γₜ} {Γₚ = ◇p} σₚ = {!FFOL.ε M!}
    mSubp {Γₚ = Γₚ ▷p⁰ A} σₚ = FFOL._,ₚ_ M (mSubp (πₚ¹ σₚ)) {!m⊢ (πₚ² σₚ)!}


    mSub : {Δ : Con}{Γ : Con} → Sub Δ Γ → (FFOL.Sub M (mCon Δ) (mCon Γ))
    mSub {Δ}{Γ} σ = FFOL._∘_ M (subst (FFOL.Sub M (mContₜ (Con.t Δ))) {!!} (mSubt
(Sub.t σ))) ({!mSubp (Sub.p σ)!})


    e▷ₚ : {Γ : Con}{A : For (Con.t Γ)} → mCon (Γ ▷p A) ≡ FFOL._▷ₚ_ M (mCon Γ)
(mFor {Γ} A)
    e[]f : {Γ Δ : Con}{A : For (Con.t Γ)}{σ : Sub Δ Γ} → mFor {Δ} (A [ Sub.t σ
]f) ≡ FFOL._[_]f M (mFor {Γ} A) (mSub σ)


    {-
    e▷ₜ {con Γₜ ◇p} = refl
    e▷ₜ {con Γₜ (Γₚ ▷p⁰ A)} = ≡tran²
      (cong₂' (FFOL._▷ₚ_ M) (e▷ₜ {con Γₜ Γₚ}) (cong (subst (FFOL.For M) (e▷ₜ {Γ =
con Γₜ Γₚ})) (e[]f {A = A}{σ = πₜ¹* id})))
      (substP (λ X → (M FFOL.▷ₚ (M FFOL.▷ₜ) (mCon (con Γₜ Γₚ))) X ≡ (M FFOL.▷ₜ) ((M
FFOL.▷ₚ mCon (con Γₜ Γₚ)) (mFor A))
        (≡tran
          (coeshift {!!})
          (cong (λ X → subst (FFOL.For M) _ (FFOL._[_]f M (mFor A) (mSub (sub
(wkₜσₜ idₜ) X)))) (≡sym (coecoe-coe {eq1 = {!!}} {x = idₚ {Δₚ = Γₚ}}))))
        {!!})
      (cong (M FFOL.▷ₜ) (≡sym (e▷ₚ {con Γₜ Γₚ})))
    -- substP (λ X → FFOL._▷ₚ_ M X (mFor {Γ = ?} (A [ wkₜσₜ idₜ ]f)) ≡ (FFOL._▷ₜ M
(mCon (con Γₜ (Γₚ ▷p⁰ A))))) (≡sym (e▷ₜ {Γ = con Γₜ Γₚ})) ?
    -}
    e[]f = {!!}
    e▹ₚ = {!!}


    {-




    e∘ : {Γ Δ Ξ : Con}{δ : Sub Δ Ξ}{σ : Sub Γ Δ} → mSub (δ ∘ σ) ≡ FFOL._∘_ M
(mSub δ) (mSub σ)
    e∘ = {!!}
    eid : {Γ : Con} → mSub (id {Γ}) ≡ FFOL.id M {mCon Γ}
    eid = {!!}
    e◇ : mCon ◇ ≡ FFOL.◇ M
    e◇ = {!!}
    eε : {Γ : Con} → mSub (sub (εₜ {Con.t Γ}) (εₚ {Con.t Γ} {Con.p Γ})) ≡ subst
(FFOL.Sub M (mCon Γ)) (≡sym e◇) (FFOL.ε M {mCon Γ})
    eε = {!!}
    e[]t : {Γ Δ : Con}{t : Tm (Con.t Γ)}{σ : Sub Δ Γ} → mTm (t [ Sub.t σ ]t) ≡
FFOL._[_]t M (mTm t) (mSub σ)
    e[]t = {!!}
    eπₜ¹ : {Γ Δ : Con}{σ : Sub Δ (Γ ▹t)} → mSub (πₜ¹* σ) ≡ FFOL.πₜ¹ M (subst
(FFOL.Sub M (mCon Δ)) e▹ₜ (mSub σ))
    eπₜ¹ = {!!}
    eπₜ² : {Γ Δ : Con}{σ : Sub Δ (Γ ▹t)} → mTm (πₜ²* σ) ≡ FFOL.πₜ² M (subst
(FFOL.Sub M (mCon Δ)) e▹ₜ (mSub σ))
    eπₜ² = {!!}
    e,ₜ : {Γ Δ : Con}{σ : Sub Δ Γ}{t : Tm (Con.t Δ)} → mSub (σ ,ₜ* t) ≡ subst
(FFOL.Sub M (mCon Δ)) (≡sym e▹ₜ) (FFOL._,ₜ_ M (mSub σ) (mTm t))
    e,ₜ = {!!}
    -- Proofs are in prop, so no equation needed
    --[]p : {Γ Δ : Con}{A : For Γ}{pf : FFOL._⊢_ S Γ A}{σ : FFOL.Sub S Δ Γ} → m⊢
(FFOL._[_]p S pf σ) ≡ FFOL._[_]p M (m⊢ pf) (mSub σ)
    e▹ₚ = {!!}
    eπₚ¹ : {Γ Δ : Con}{A : For (Con.t Γ)}{σ : Sub Δ (Γ ▹p A)} → mSub (πₚ¹* σ) ≡
FFOL.πₚ¹ M (subst (FFOL.Sub M (mCon Δ)) e▹ₚ (mSub σ))
    eπₚ¹ = {!!}
    --πₚ² : {Γ Δ : Con}{A : For Γ}{σ : Sub Δ (Γ ▹p A)} → m⊢ (πₚ²* σ) ≡ FFOL.πₚ¹ M
(subst (FFOL.Sub M (mCon Δ)) e▹ₚ (mSub σ))
    e,ₚ : {Γ Δ : Con}{A : For (Con.t Γ)}{σ : Sub Δ Γ}{pf : Pf (Con.t Δ) (Con.p Δ)
(A [ Sub.t σ ]f)}
      → mSub (σ ,ₚ* pf) ≡ subst (FFOL.Sub M (mCon Δ)) (≡sym e▹ₚ) (FFOL._,ₚ_ M
(mSub σ) (substP (FFOL._⊢_ M (mCon Δ)) e[]f (m⊢ pf)))
    e,ₚ = {!!}
    eR : {Γ : Con}{t u : Tm (Con.t Γ)} → mFor (R t u) ≡ FFOL.R M (mTm t) (mTm u)
    eR = {!!}
    e⇒ : {Γ : Con}{A B : For (Con.t Γ)} → mFor (A ⇒ B) ≡ FFOL._⇒_ M (mFor A)
(mFor B)
    e⇒ = {!!}
    e∀∀ : {Γ : Con}{A : For ((Con.t Γ) ▹t⁰)} → mFor (∀∀ A) ≡ FFOL.∀∀ M (subst
(FFOL.For M) e▹ₜ (mFor A))
    e∀∀ = {!!}
    -}
    m : Mapping ffol M
    m = record { mCon = mCon ; mSub = mSub ; mTm = λ {Γ} t → mTm {Γ} t ; mFor = λ
{Γ} A → mFor {Γ} A ; m⊢ = m⊢ }

  --mor : (M : FFOL) → Morphism ffol M
  --mor M = record {InitialMorphism M}
```

```
\end{code}
```