Abstract

Contents

1 Introduction

2 Agda proof assistant

Agda is a dependently typed programming language based on intuitionistic type theory. By encoding mathematical propositions as types and their proofs as programs, we can ensure that our reasoning is correct and consistent. Agda's type system also provides powerful tools for automatically checking the correctness of proofs[1].

2.1 Propositions as types

Propositions as types associates logical propositions with types in a programming language. It is based on the idea that a proof of a proposition is analogous to a program that satisfies the type associated with the proposition.

In this context, the introduction and elimination rules for logical connectives can be seen as operations that construct and deconstruct values of the corresponding types. For example, the introduction rule for conjunction says that if we have proofs of two propositions, we can construct a proof of their conjunction by pairing the two proofs together. This can be seen as a function that takes two values of the corresponding types and returns a pair value.

On the other hand, the elimination rule for conjunction says that if we have a proof of a conjunction, we can extract proofs of its two conjuncts by projecting the pair onto each component. This can be seen as a function that takes a pair value and returns two values of the corresponding types.

This is similar to the concept of product types in programming languages, where a product type is a type that represents a pair of values. The introduction form of a product type is a pair, and the elimination forms are projection functions that extract the individual components of the tuple. This chart summarizes the correspondence between proposition and types and between proofs and programs.

Prop	Type
$\top$	unit
$\bot$	void
$\phi_1 \wedge \phi_2$	$\tau_1 \times \tau_2$
$\phi_1 \supset \phi_2$	$\tau_1 \rightarrow \tau_2$
$\phi_1 \vee \phi_2$	$\tau_1 + \tau_2$

Table 1: Propositions as types

This allows us to reason about logical propositions in terms of programming language types, and to use the tools and techniques of programming languages like `Agda` to reason about logical proofs.

2.2 Simply typed functions and datatypes

A data declaration is used to introduce datatypes, including their name, type, and constructors along with their types. An example of this is the declaration of the boolean type:

```
data Bool : Set where
    true  : Bool
    false : Bool
```

This states that `Bool` is a data type with `true` and `false` as constructors. Functions over this datatype `Bool` can be defined using pattern matching, similar to Haskell. For instance we can define a function `not` for `Bool` as

```
data not : Bool \to Bool
    not true  = false
    not false = true
```

We start by defining the type of not as a function from `Bool` to `Bool` and then we define the function by using pattern matching on the arguments. Agda checks that the pattern covers all cases and will not accept a function with missing patterns.

The natural numbers can be defined as the datatype:

```
data N : Set where
    zero : N
    suc  : N \to N
```

A natural number is either zero or a successor of another natural number. This is called an *inductively defined type*. We can define addition on the natural numbers with a recursive function.

```
data _+_ : N \to N \to N
    zero    + m = m
    (suc n) + m = suc (n + m)
```

If a name contains underscores (_) in the definition, the underscores represent where the arguments go. So in this case we get an infix operator and we write m + n instead of + m n, which would have been the case if the name was just +. We can set the precedence of an infix operator with an `infix` declaration:

```
infix 25 _+_
```

Datatypes can also be parameterized by other types. The type of lists with elements of an arbitrary type is defined as:

```
infix 20 _::_
data List (A : Set) : Set where
    []   : List A
    _::_ : A \to List A \to List A
```

2.3 Dependent types

A dependent type is a type that depends on elements of another type. An example of a dependent type is a dependent function, where the result type depends on the value of the argument. In Agda, this is denoted by (x : A) $\to$ B, representing functions that take an argument x of type A and produce a result of type B. A special case is when x itself is a type. For instance, we can define the identity function

```
id : (A : Set) \to A \to A
id A x = x
```

This function takes a type argument A and an element x of type A, and returns x. In Agda it is possible to use implicit arguments. To declare an argument as implicit we use curly braces instead of parenthesis when declaring the type argument. In particular, {A : Set} $\to$ B means the same thing as (A : Set) $\to$ B, but we don't need to provide the type explicitly, the type checker will try to infer it for us. We can now redefine the identity function above as

```
id : {A : Set} \to A \to A
id x = x
```

and now we no longer need to supply the type when the function is applied.

3 Propositional calculus in Agda

Propositional calculus is a formal system that consists of a set of propositional constants, symbols, inference rules, and axioms. The symbols in propositional calculus represent logical connectives and parentheses, and are used to construct well-formed formulas that follow the syntax of the system. The inference rules of propositional calculus specify how these symbols can be used to derive additional statements from the initial assumptions, which are given by the axioms of the system.

The semantics of propositional calculus define how the expressions in the system correspond to truth values, typically "true" or "false".

3.1 Formulas

Definition 3.1 (Language)**.** *The language $\mathcal{L}$ of propositional calculus consists of*

- *proposition symbols:* $p_0, p_1, \ldots, p_n$*,*
- *logical connectives:* $\wedge, \vee, \neg, \top, \bot$*,*
- *auxiliary symbols: (,).*

Note that we have omitted the common logical connectives $\to$ and $\leftrightarrow$. This is becuase we can define them using other connectives,

$$\phi \to \psi \stackrel{\text{def}}{=} \neg\phi \lor \psi,$$
$$\phi \leftrightarrow \psi \stackrel{\text{def}}{=} (\neg\phi \lor \psi) \land (\neg\psi \lor \phi),$$

making them reduntant. It is possible to choose an even smaller set of connectives [2], but we choose this as it is convenient.

Definition 3.2 (Well formed formula)**.** *The set of well formed formulas is inductively defined as*

- *any propositional constant $p_0, p_1, \dots, p_n$ is a well formed formula,*
- *$\top$ and $\bot$ are well formed formulas,*
- *if p is a well formed formula, then so is*

$$\neg p,$$

- *if p_i and p_j are well formed formulas, then so are*

$$p_i \land p_j \quad \textit{and} \quad p_i \lor p_j.$$

The formula $\top$ should be thought of as the proposition that is always true, and the formula $\bot$ interpreted as the proposition that is always false.

We represent the concept of a well formed formula in `Agda` as a data type.

```
data Formula : Type where
  _∧_ : Formula → Formula → Formula
  _∨_ : Formula → Formula → Formula
  ¬_  : Formula → Formula
  const : ℕ → Formula
  ⊥   : Formula
  ⊤   : Formula
```

3.2 Context

Definition 3.3 (Context)**.** *A set of sentences in the language $\mathcal{L}$. The set is defined inductively as*

- *the empty set is a context*
- *if Γ is a context, then $\Gamma \cup \{\phi\}$ is also a context, where ϕ a formula.*

In Agda we can define a data type for context.

```
data ctxt : Type where
  ∅   : ctxt
  _·_ : ctxt → Formula → ctxt
```

We also need a way to determine if a given formula is in a given context.

Definition 3.4 (Lookup)**.** *For all contexts Γ and all formulas ϕ and ψ*

- $\phi \in \Gamma \cup \{\phi\}$,
- *if* $\phi \in \Gamma$, *then* $\phi \in \Gamma \cup \{\psi\}$.

We represent this as a data type in Agda

```
data _∈_ : Formula → ctxt → Type where
  Z : ∀ {Γ φ} → φ ∈ Γ · φ
  S : ∀ {Γ φ ψ} → φ ∈ Γ → φ ∈ Γ · ψ
```

3.3 Inference rules

For the inference rules we introduce a data type for provability

```
data _⊢_ : ctxt → Formula → Type where

                        ⋮
```

where we will define our inference rules on the following form:

```
rulename : {... : ctxt} {... : Formula}
        -> premise.1
        -> premise.2
              :
        -> premise.n
        -> conclusion
```

3.3.1 Law of excluded middle

Definition 3.5. *The law of excluded middle states that for every proposition, either the proposition or its negation is true.*

$$\frac{}{\Gamma \vdash \phi \vee \neg\phi}\ \text{LEM}$$

The law of excluded middle in Agda:

```
LEM : {Γ : ctxt} {φ : Formula}
    → Γ ⊢ φ ∨ ¬ φ
```

3.3.2 Logical connectives

Rules for the logical connectives come in pairs of introduction and elimination rules, a part from $\top$, which has only an introduction rule, and $\bot$, which has only an elimination rule. These rules are based on [2].

The introductory rule for conjunction states that if there is a derivation of ϕ in the context Γ, and a derivation of ψ in the context Γ, then we can conclude that there is a derivation of $\phi \wedge \psi$ in Γ.

$$\frac{\Gamma \vdash \phi \qquad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi} \wedge\text{-I}$$

The conjunction introduction rule in `Agda`:

```
∧-I : {Γ : ctxt} {φ ψ : Formula}
    → Γ ⊢ φ
    → Γ ⊢ ψ
    → Γ ⊢ φ ∧ ψ
```

The accompanying elimination rules says that if there is some derivation concluding in $\phi \wedge \psi$ in Γ, then we can conclude that there is a derivation of ϕ and a derivation of ψ in Γ.

$$\frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi} \wedge\text{-E}_1 \qquad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \psi} \wedge\text{-E}_2$$

The conjunction elimination rules in `Agda`:

```
∧-E₁ : {Γ : ctxt} {φ ψ : Formula}
     → Γ ⊢ φ ∧ ψ
     → Γ ⊢ φ


∧-E₂ : {Γ : ctxt} {φ ψ : Formula}
     → Γ ⊢ φ ∧ ψ
     → Γ ⊢ ψ
```

For disjunction we have two introductory rules. If Γ proves some formula ψ, then Γ proves $\phi \vee \psi$. In the same way Γ proves $\phi \vee \psi$ if Γ proves ϕ.

$$\frac{\Gamma \vdash \psi}{\Gamma \vdash \phi \vee \psi} \vee\text{-I}_1 \qquad \frac{\Gamma \vdash \phi}{\Gamma \vdash \phi \vee \psi} \vee\text{-I}_2$$

The disjunction introduction rules in `Agda`:

```
∨-I₁ : {Γ : ctxt} {φ ψ : Formula}
     → Γ ⊢ ψ
     → Γ ⊢ φ ∨ ψ
```

$$\begin{aligned}\vee\text{-I}_2 : &\ \{\Gamma : \mathsf{ctxt}\}\ \{\phi\ \psi : \mathsf{Formula}\} \\ &\to \Gamma \vdash \phi \\ &\to \Gamma \vdash \phi \vee \psi\end{aligned}$$

The eliminination rule for disjunction is a bit more complicated. If Γ proves $\phi \vee \psi$, then we can conclude Γ proves γ if the extended contexts Γ, ϕ and Γ, ψ both prove γ.

$$\frac{\Gamma \vdash \phi \vee \psi \qquad \Gamma, \phi \vdash \gamma \qquad \Gamma, \psi \vdash \gamma}{\Gamma \vdash \gamma}\ \vee\text{-E}$$

The disjunction elimination rule in `Agda`:

$$\begin{aligned}\vee\text{-E} : &\ \{\Gamma : \mathsf{ctxt}\}\ \{\phi\ \psi\ \gamma : \mathsf{Formula}\} \\ &\to \Gamma \vdash \phi \vee \psi \\ &\to \Gamma : \phi \vdash \gamma \\ &\to \Gamma : \psi \vdash \gamma \\ &\to \Gamma \vdash \gamma\end{aligned}$$

Definition 3.6. *A context Γ is inconsistent if $\Gamma \vdash \bot$. A context that is not inconsistent, i.e. $\Gamma \nvdash \bot$, is called consistent.*

This definition and the law of excluded middle together motivates the introduction and elimination rules for negation.

$$\frac{\Gamma, \phi \vdash \bot}{\Gamma \vdash \neg\phi}\ \neg\text{-I} \qquad\qquad \frac{\Gamma \vdash \phi \qquad \Gamma \vdash \neg\phi}{\Gamma \vdash \bot}\ \neg\text{-E}$$

The negation rules in `Agda`:

$$\begin{aligned}\neg\text{-I} : &\ \{\Gamma : \mathsf{ctxt}\}\ \{\phi : \mathsf{Formula}\} \\ &\to \Gamma : \phi \vdash \bot \\ &\to \Gamma \vdash \neg\ \phi\end{aligned}$$

$$\begin{aligned}\neg\text{-E} : &\ \{\Gamma : \mathsf{ctxt}\}\ \{\phi : \mathsf{Formula}\} \\ &\to \Gamma \vdash \phi \\ &\to \Gamma \vdash \neg\ \phi \\ &\to \Gamma \vdash \bot\end{aligned}$$

Since $\top$ is always trivially true it can be introduced with no premise.

$$\frac{}{\Gamma \vdash \top}\ \top\ \text{I}$$

The $\top$ introduction rule in `Agda`:

$$\begin{aligned}\top\text{-I} : &\ \{\Gamma : \mathsf{ctxt}\} \\ &\to \Gamma \vdash \top\end{aligned}$$

If a context is inconsistent one can derive anything from it, which leads to the elimination rule for $\bot$.

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash \phi}\ \bot\text{-E}$$

The $\bot$ elimination rule in `Agda`:

```
⊥-E : {Γ : ctxt} {φ : Formula}
    → Γ ⊢ ⊥
    → Γ ⊢ φ
```

3.3.3 Structural rules

Weakening is a structural rule that states that we can extend the hypothesis with additional members,

$$\frac{\Gamma \vdash \phi}{\Gamma, \psi \vdash \phi}\ \text{WEAKENING}$$

The weakening rule in `Agda`:

```
weakening : {Γ : ctxt} {φ ψ : Formula}
          → Γ ⊢ ψ
          → Γ : φ ⊢ ψ
```

$$\frac{\Gamma, \phi, \psi \vdash \gamma}{\Gamma, \psi, \phi \vdash \gamma}\ \text{EXCHANGE}$$

```
exchange : {Γ : ctxt} {φ ψ γ : Formula}
         → (Γ : φ) : ψ ⊢ γ
         → (Γ : ψ) : φ ⊢ γ
```

3.4 Properties of a propositional calculus

We prove some properties of propositional calculus

3.4.1 Commutativity

First we prove commutativity for conjunction, that is, we want to show that if $\Gamma \vdash \phi \land \psi$ then $\Gamma \vdash \psi \land \phi$. We can to this by natural deduction, using the rules defined earlier

$$\dfrac{\dfrac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \psi}\ \wedge\text{-E}_2 \qquad \dfrac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi}\ \wedge\text{-E}_1}{\Gamma \vdash \psi \wedge \phi}\ \wedge\text{-I}$$

and we write the proof in `Agda` as

$\wedge$-comm : $\forall$ {ϕ ψ : Formula} $\rightarrow$ Γ : $\phi \wedge \psi \vdash \psi \wedge \phi$
$\wedge$-comm = $\wedge$-I ($\wedge$-E$_2$ (axiom Z)) ($\wedge$-E$_1$ (axiom Z))

Next we show that disjunction is commutative, first by natural deduction

$$\dfrac{\Gamma \vdash \phi \vee \psi \qquad \dfrac{\dfrac{\phi \in \Gamma, \phi}{\Gamma, \phi \vdash \phi}\ \text{AXIOM}}{\Gamma, \phi \vdash \psi \vee \phi}\ \vee\text{-I}_1 \qquad \dfrac{\dfrac{\psi \in \Gamma, \psi}{\Gamma, \psi \vdash \psi}\ \text{AXIOM}}{\Gamma, \psi \vdash \psi \vee \phi}\ \vee\text{-I}_2}{\Gamma \vdash \psi \vee \phi}\ \vee\text{-E}$$

and then by `Agda` proof.

$\vee$-comm : {ϕ ψ : Formula} $\rightarrow$ Γ : $\phi \vee \psi \vdash \psi \vee \phi$
$\vee$-comm = $\vee$-E (axiom Z) ($\vee$-I$_1$ (axiom Z)) ($\vee$-I$_2$ (axiom Z))

3.4.2 Associativity

```
∧-assoc1 : ∀ {φ ψ γ : Formula} → Γ : φ ∧ (ψ ∧ γ) ⊢ (φ ∧ ψ) ∧ γ
∧-assoc1 = ∧-I (∧-I (∧-E₁ (axiom Z))
                    (∧-E₁ (∧-E₂ (axiom Z))))
               (∧-E₂ (∧-E₂ (axiom Z)))
```

```
∧-assoc2 : ∀ {φ ψ γ : Formula} → Γ : (φ ∧ ψ) ∧ γ ⊢ φ ∧ (ψ ∧ γ)
∧-assoc2 = ∧-I (∧-E₁ (∧-E₁ (axiom Z)))
               (∧-I (∧-E₂ (∧-E₁ (axiom Z)))
                    (∧-E₂ (axiom Z)))
```

```
∨-assoc1 : ∀ {φ ψ γ : Formula} → Γ : φ ∨ (ψ ∨ γ) ⊢ (φ ∨ ψ) ∨ γ
∨-assoc1 = ∨-E (axiom Z)
               (∨-I₂ (∨-I₂ (axiom Z)))
               (∨-E (axiom Z)
                    (∨-I₂ (∨-I₁ (axiom Z)))
                    (∨-I₁ (axiom Z)))
```

```
∨-assoc2 : ∀ {φ ψ γ : Formula} → Γ : (φ ∨ ψ) ∨ γ ⊢ φ ∨ (ψ ∨ γ)
∨-assoc2 = ∨-E (axiom Z)
```

```
(∨-E (axiom Z)
     (∨-I₂ (axiom Z))
     (∨-I₁ (∨-I₂ (axiom Z))))
(∨-I₁ (∨-I₁ (axiom Z)))
```

3.4.3 Distributivity

Denote $\Gamma' = \Gamma \cup \{\phi \wedge (\psi \vee \gamma)\}$

$$\dfrac{\dfrac{\Gamma' \vdash \phi \wedge (\psi \vee \gamma)}{\Gamma' \vdash \psi \vee \gamma}\;\wedge\text{-E}_2 \qquad \begin{matrix}\mathcal{D}_1\\ \Gamma', \psi \vdash (\phi \wedge \psi) \vee (\phi \wedge \gamma)\end{matrix} \qquad \begin{matrix}\mathcal{D}_2\\ \Gamma', \gamma \vdash (\phi \wedge \psi) \vee (\phi \wedge \gamma)\end{matrix}}{\Gamma' \vdash (\phi \wedge \psi) \vee (\phi \wedge \gamma)}\;\vee\text{-E}$$

$\mathcal{D}_1$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\phi \wedge (\psi \vee \gamma) \in \Gamma'}{\Gamma' \vdash \phi \wedge (\psi \vee \gamma)}\;\text{AXIOM}}{\Gamma' \vdash \phi}\;\wedge\text{-E}_1}{\Gamma', \psi \vdash \phi}\;\text{WEAKENING} \qquad \dfrac{\psi \in \Gamma', \psi}{\Gamma', \psi \vdash \psi}\;\text{AXIOM}}{\Gamma', \psi \vdash \phi \wedge \psi}\;\wedge\text{-I}}{\Gamma' \psi \vdash (\phi \wedge \psi) \vee (\phi \wedge \gamma)}\;\vee\text{-I}_1$$

$\mathcal{D}_2$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\phi \wedge (\psi \vee \gamma) \in \Gamma'}{\Gamma' \vdash \phi \wedge (\psi \vee \gamma)}\;\text{AXIOM}}{\Gamma' \vdash \phi}\;\wedge\text{-E}_1}{\Gamma', \gamma \vdash \phi}\;\text{WEAKENING} \qquad \dfrac{\gamma \in \Gamma', \gamma}{\Gamma', \gamma \vdash \gamma}\;\text{AXIOM}}{\Gamma', \gamma \vdash \phi \wedge \gamma}\;\wedge\text{-I}}{\Gamma' \gamma \vdash (\phi \wedge \psi) \vee (\phi \wedge \gamma)}\;\vee\text{-I}_1$$

```
∧-dist1 : ∀ {φ ψ γ : Formula} → Γ : φ ∧ (ψ ∨ γ) ⊢ (φ ∧ ψ) ∨ (φ ∧ γ)
∧-dist1 = ∨-E (∧-E₂ (axiom Z))
              (∨-I₂ (∧-I (weakening (∧-E₁ (axiom Z))) (axiom Z)))
              (∨-I₁ (∧-I (weakening (∧-E₁ (axiom Z))) (axiom Z)))
```

$\Gamma \vdash (\phi \wedge \psi) \vee (\phi \wedge \gamma) \to \Gamma \vdash \phi \wedge (\psi \vee \gamma)$:
Denote $\Gamma' = \Gamma \cup (\phi \wedge \psi) \vee (\phi \wedge \gamma)$

$$\dfrac{\begin{matrix}\mathcal{D}_1\\ \Gamma' \vdash \phi\end{matrix} \qquad \begin{matrix}\mathcal{D}_2\\ \Gamma' \vdash \psi \vee \gamma\end{matrix}}{\Gamma' \vdash \phi \wedge (\psi \vee \gamma)}\;\wedge\text{-I}$$

$\mathcal{D}_1$

$$\dfrac{\dfrac{(\phi \wedge \psi) \vee (\phi \wedge \gamma) \in \Gamma'}{\Gamma' \vdash (\phi \wedge \psi) \vee (\phi \wedge \gamma)}\,\text{AXIOM} \qquad \dfrac{\dfrac{\phi \wedge \psi \in \Gamma', \phi \wedge \psi}{\Gamma', \phi \wedge \psi \vdash \phi \wedge \psi}\,\text{AXIOM}}{\Gamma', \phi \wedge \psi \vdash \phi}\,\wedge\text{-E}_1 \qquad \dfrac{\dfrac{\phi \wedge \gamma \in \Gamma', \phi \wedge \gamma}{\Gamma', \phi \wedge \gamma \vdash \phi \wedge \gamma}\,\text{AXIOM}}{\Gamma', \phi \wedge \gamma \vdash \phi}\,\wedge\text{-E}_1}{\Gamma' \vdash \phi}\,\vee\text{-E}$$

$\mathcal{D}_2$

$$\dfrac{\dfrac{(\phi \wedge \psi) \vee (\phi \wedge \gamma) \in \Gamma'}{\Gamma' \vdash (\phi \wedge \psi) \vee (\phi \wedge \gamma)}\,\text{AXIOM} \qquad \dfrac{\dfrac{\dfrac{\phi \wedge \psi \in \Gamma', \phi \wedge \psi}{\Gamma', \phi \wedge \psi \vdash \phi \wedge \psi}\,\text{AXIOM}}{\Gamma', \phi \wedge \psi \vdash \psi}\,\wedge\text{-E}_2}{\Gamma', \phi \wedge \psi \vdash \psi \vee \gamma}\,\vee\text{-I}_2 \qquad \dfrac{\dfrac{\dfrac{\phi \wedge \gamma \in \Gamma', \phi \wedge \gamma}{\Gamma', \phi \wedge \gamma \vdash \phi \wedge \gamma}\,\text{AXIOM}}{\Gamma', \phi \wedge \gamma \vdash \gamma}\,\wedge\text{-E}_2}{\Gamma', \phi \wedge \gamma \vdash \psi \vee \gamma}\,\vee\text{-I}_1}{\Gamma' \vdash \psi \vee \gamma}\,\vee\text{-E}$$

$$\begin{aligned}
&\wedge\text{-dist2} : \forall\ \{\phi\ \psi\ \gamma : \text{Formula}\} \to \Gamma : (\phi \wedge \psi) \vee (\phi \wedge \gamma) \vdash \phi \wedge (\psi \vee \gamma)\\
&\wedge\text{-dist2} = \wedge\text{-I}\ (\vee\text{-E}\ (\text{axiom Z})\\
&\qquad\qquad\qquad\qquad (\wedge\text{-E}_1\ (\text{axiom Z}))\\
&\qquad\qquad\qquad\qquad (\wedge\text{-E}_1\ (\text{axiom Z})))\\
&\qquad\qquad\qquad (\vee\text{-E}\ (\text{axiom Z})\\
&\qquad\qquad\qquad\qquad (\vee\text{-I}_2\ (\wedge\text{-E}_2\ (\text{axiom Z})))\\
&\qquad\qquad\qquad\qquad (\vee\text{-I}_1\ (\wedge\text{-E}_2\ (\text{axiom Z}))))
\end{aligned}$$

$\Gamma \vdash \phi \vee (\psi \wedge \gamma) \to \Gamma \vdash (\phi \vee \psi) \wedge (\phi \vee \gamma)$:

$$\dfrac{\Gamma \vdash \phi \vee (\psi \wedge \gamma) \qquad T_{\vee dist1} \qquad T'_{\vee dist1}}{\Gamma \vdash (\phi \vee \psi) \wedge (\phi \vee \gamma)}\,\vee\text{-E}$$

$T_{\vee dist1}$

$$\dfrac{\dfrac{[\Gamma \vdash \phi]}{\Gamma \vdash \phi \vee \psi}\,\vee\text{-I}_2 \qquad \dfrac{[\Gamma \vdash \phi]}{\Gamma \vdash (\phi \vee \gamma)}\,\vee\text{-I}_2}{\Gamma \vdash (\phi \vee \psi) \wedge (\phi \vee \gamma)}\,\wedge\text{-I}$$

$T'_{\vee dist1}$

$$\dfrac{\dfrac{\dfrac{[\Gamma \vdash \psi \wedge \gamma]}{\Gamma \vdash \psi}\,\wedge\text{-E}_1}{\Gamma \vdash (\phi \vee \psi)}\,\vee\text{-I}_1 \qquad \dfrac{\dfrac{[\Gamma \vdash \psi \wedge \gamma]}{\Gamma \vdash \gamma}\,\wedge\text{-E}_2}{\Gamma \vdash (\phi \vee \gamma)}\,\vee\text{-I}_1}{\Gamma \vdash (\phi \vee \psi) \wedge (\phi \vee \gamma)}\,\wedge\text{-I}$$

```
∨-dist1 : ∀ {ϕ ψ γ : Formula} → Γ : ϕ ∨ (ψ ∧ γ) ⊢ (ϕ ∨ ψ) ∧ (ϕ ∨ γ)
∨-dist1 = ∨-E (axiom Z)
              (∧-I (∨-I₂ (axiom Z))
                   (∨-I₂ (axiom Z)))
              (∧-I (∨-I₁ (∧-E₁ (axiom Z)))
                   (∨-I₁ (∧-E₂ (axiom Z))))
```

$\Gamma \vdash (\phi \vee \psi) \wedge (\phi \vee \gamma) \to \Gamma \vdash \phi \vee (\psi \wedge \gamma)$:

$$\dfrac{\dfrac{\Gamma \vdash (\phi \vee \psi) \wedge (\phi \vee \gamma)}{\Gamma \vdash \phi \vee \psi}\,\wedge\text{-E}_1 \qquad \dfrac{[\Gamma \vdash \phi]}{\Gamma \vdash \phi \vee (\psi \wedge \gamma)}\,\vee\text{-I}_2 \qquad T_{\vee dist2}}{\Gamma \vdash \phi \vee (\psi \wedge \gamma)}\,\vee\text{-E}$$

$T_{\vee dist2}$

$$\dfrac{\dfrac{\Gamma \vdash (\phi \vee \psi) \wedge (\phi \vee \gamma)}{\Gamma \vdash \phi \vee \gamma}\,\wedge\text{-E}_2 \qquad \dfrac{[\Gamma \vdash \phi]}{\Gamma \vdash \phi \vee (\psi \wedge \gamma)}\,\vee\text{-I}_2 \qquad T_{\vee dist2}}{\Gamma \vdash \phi \vee (\psi \wedge \gamma)}\,\vee\text{-E}$$

$T'_{\vee dist2}$

$$\dfrac{\dfrac{[\Gamma \vdash \psi] \qquad [\Gamma \vdash \gamma]}{\Gamma \vdash \psi \wedge \gamma}\,\wedge\text{-I}}{\Gamma \vdash \phi \vee (\psi \wedge \gamma)}\,\vee\text{-I}_1$$

```
∨-dist2 : ∀ {ϕ ψ γ : Formula} → Γ : (ϕ ∨ ψ) ∧ (ϕ ∨ γ) ⊢ ϕ ∨ (ψ ∧ γ)
∨-dist2 = ∨-E (∧-E₁ (axiom Z))
              (∨-I₂ (axiom Z))
              (∨-E (∧-E₂ (weakening (axiom Z)))
                    (∨-I₂ (axiom Z))
                    (∨-I₁ (∧-I (weakening (axiom Z)) (axiom Z))))
```

4 Lindenbaum-Tarski algebra in Cubical Agda

4.1 Definition

Definition 4.1. *Let $\mathcal{L}$ be the propositional language described in Section 3 and let S be the set of all the sentences of $\mathcal{L}$. Define the relation $\sim$ such that for $\phi, \psi \in S$,*

$$\phi \sim \psi \qquad \textit{iff} \qquad \Gamma, \phi \vdash \psi \textit{ and } \Gamma, \psi \vdash \phi$$

This is represented in `Agda` as a product type.

```
_∼_ : Formula → Formula → Type
ϕ ∼ ψ = Γ : ϕ ⊢ ψ × Γ : ψ ⊢ ϕ
```

Before we prove that this is an equivalence relation we will prove a lemma.

Lemma 4.1. *Given $\Gamma, \phi \vdash \psi$ and $\Gamma, \psi \vdash \gamma$, it follows that $\Gamma, \phi \vdash \gamma$*

Proof. This is done through natural deduction

$$\dfrac{\dfrac{\Gamma, \phi \vdash \psi}{\Gamma, \phi \vdash \psi \vee \gamma}\ \vee\text{-}\mathrm{I}_2 \qquad \dfrac{\dfrac{\Gamma, \psi \vdash \gamma}{\Gamma, \psi, \phi \vdash \gamma}\ \text{WEAKENING}}{\Gamma, \phi, \psi \vdash \gamma}\ \text{EXCHANGE} \qquad \dfrac{\gamma \in \Gamma, \phi, \gamma}{\Gamma, \phi, \gamma \vdash \gamma}\ \text{AXIOM}}{\Gamma, \phi \vdash \gamma}\ \vee\text{-E}$$

□

And the corresponding `Agda` proof of the lemma:

```
⊢trans : ∀ {φ ψ γ : Formula} → Γ : φ ⊢ γ → Γ : γ ⊢ ψ → Γ : φ ⊢ ψ
⊢trans A B = ∨-E (∨-I₂ A) (exchange (weakening B)) (axiom Z)
```

Theorem 4.1. *The relation $\sim$ is an equivalence relation.*

Proof. Reflexivity follows from the axiom rule and definition 3.4.

$$\dfrac{\phi \in \Gamma, \phi}{\Gamma, \phi \vdash \phi}\ \text{AXIOM}$$

This gives the `Agda` proof:

```
∼-refl : ∀ (φ : Formula) → φ ∼ φ
∼-refl _ = axiom Z , (axiom Z)
```

It should be clear that $\Gamma, \phi \vdash \psi$ and $\Gamma, \psi \vdash \phi$ is just a pair of proofs, hence it does not matter in which order we give them. This means that the relation is also symmetric.

```
∼-sym : ∀ {φ ψ : Formula} → φ ∼ ψ → ψ ∼ φ
∼-sym (A , B) = B , A
```

For transitivity we need to prove that, given $\phi \sim \gamma$ and $\gamma \sim \psi$, it holds that $\phi \sim \psi$. By definition 4.1 we have proof of

(i) $\Gamma, \phi \vdash \gamma$,

(ii) $\Gamma, \gamma \vdash \phi$,

(iii) $\Gamma, \gamma \vdash \psi$, and

(iv) $\Gamma, \psi \vdash \gamma$.

Now we can apply Lemma 4.1 on (i) and (iii) to get $\Gamma, \phi \vdash \psi$, and again to (iv) and (ii) to get $\Gamma, \psi \vdash \phi$. The proof in `Agda` is straight forward.

```
∼-trans : ∀ {φ ψ γ : Formula} → φ ∼ γ → γ ∼ ψ → φ ∼ ψ
∼-trans x y = ⊢trans (proj₁ x) (proj₁ y) , ⊢trans (proj₂ y) (proj₂ x)
```

□

Definition 4.2 (Lindenbaum-Tarski algebra)**.** *The Lindenbaum-Tarski algebra is the quotient algebra obtained by factoring the algebra of formulas by the equivalence relation $\sim$.*

We use a function from the `Cubical Agda` library to define the Lindenbaum-Tarski algebra.

```
LindenbaumTarski : Type
LindenbaumTarski = Formula / _∼_
```

4.2 Binary operations and propositional constants

4.3 Proof that the Lindenbaum Tarski algebra is Boolean

4.4 Soundness

References

[1] Ana Bove and Peter Dybjer. Dependent types at work, 2008. URL: `https://www.cse.chalmers.se/~peterd/papers/DependentTypesAtWork.pdf`.

[2] Dick van Dalen. *Logic and structure*. Springer, fifth edition, 2013.

A Derivations

A.1 Derivation of $\Gamma, \phi \wedge (\psi \wedge \gamma) \vdash (\phi \wedge \psi) \wedge \gamma$

$$\dfrac{\mathcal{D}_1 \qquad \mathcal{D}_2}{\Gamma, \phi \wedge (\psi \wedge \gamma) \vdash (\phi \wedge \psi) \wedge \gamma}\ \wedge\text{-I}$$

$\mathcal{D}_1$

$$\dfrac{\dfrac{\dfrac{\dfrac{\phi \wedge (\psi \wedge \gamma) \in \Gamma, \phi \wedge (\psi \wedge \gamma)}{\Gamma, \phi \wedge (\psi \wedge \gamma) \vdash \phi \wedge (\psi \wedge \gamma)}\ \text{AXIOM}}{\Gamma, \phi \wedge (\psi \wedge \gamma) \vdash \psi \wedge \gamma}\ \wedge\text{-E}_2}{\Gamma, \phi \wedge (\psi \wedge \gamma) \vdash \psi}\ \wedge\text{-E}_1 \qquad \dfrac{\dfrac{\phi \wedge (\psi \wedge \gamma) \in \Gamma'}{\Gamma, \phi \wedge (\psi \wedge \gamma) \vdash \phi \wedge (\psi \wedge \gamma)}\ \text{AXIOM}}{\Gamma, \phi \wedge (\psi \wedge \gamma) \vdash \phi}\ \wedge\text{-E}_1}{\Gamma, \phi \wedge (\psi \wedge \gamma) \vdash \phi \wedge \psi}\ \wedge\text{-I}$$

$\mathcal{D}_2$

$$\dfrac{\dfrac{\dfrac{\phi \wedge (\psi \wedge \gamma) \in \Gamma'}{\Gamma, \phi \wedge (\psi \wedge \gamma) \vdash \phi \wedge (\psi \wedge \gamma)}\ \text{AXIOM}}{\Gamma, \phi \wedge (\psi \wedge \gamma) \vdash \psi \wedge \gamma}\ \wedge\text{-E}_2}{\Gamma, \phi \wedge (\psi \wedge \gamma) \vdash \gamma}\ \wedge\text{-E}_2$$

A.2 Derivation of $\Gamma, \phi \lor (\psi \lor \gamma) \vdash (\phi \lor \psi) \lor \gamma$

$$\dfrac{\dfrac{\phi \lor (\psi \lor \gamma) \in \Gamma, \phi \lor (\psi \lor \gamma)}{\Gamma, \phi \lor (\psi \lor \gamma) \vdash \phi \lor (\psi \lor \gamma)}\,\textsc{axiom} \qquad \dfrac{\dfrac{\dfrac{\phi \in \Gamma, \phi \lor (\psi \lor \gamma), \phi}{\Gamma, \phi \lor (\psi \lor \gamma), \phi \vdash \phi}\,\textsc{axiom}}{\Gamma, \phi \lor (\psi \lor \gamma), \phi \vdash \phi \lor \psi}\,\lor\text{-I}_2}{\Gamma, \phi \lor (\psi \lor \gamma), \phi \vdash (\phi \lor \psi) \lor \gamma}\,\lor\text{-I}_2 \qquad \begin{array}{c}\mathcal{D}\\ \Gamma, \phi \lor (\psi \lor \gamma), (\psi \lor \gamma) \vdash \psi \lor \gamma\end{array}}{\Gamma, \phi \lor (\psi \lor \gamma) \vdash (\phi \lor \psi) \lor \gamma}\,\lor\text{-E}$$

$\mathcal{D}$

$$\dfrac{\dfrac{\psi \lor \gamma \in \Gamma, \phi \lor (\psi \lor \gamma), \psi \lor \gamma}{\Gamma, \phi \lor (\psi \lor \gamma), \psi \lor \gamma \vdash \psi \lor \gamma}\,\textsc{axiom} \qquad \begin{array}{c}\mathcal{D}'\\ \Gamma, \phi \lor (\psi \lor \gamma), \psi \lor \gamma, \psi \vdash (\phi \lor \psi) \lor \gamma\end{array} \qquad \dfrac{\dfrac{\gamma \in \Gamma', \gamma}{\Gamma, \phi \lor (\psi \lor \gamma), \psi \lor \gamma, \gamma \vdash \gamma}\,\textsc{axiom}}{\Gamma, \phi \lor (\psi \lor \gamma), \psi \lor \gamma, \gamma \vdash (\phi \lor \psi) \lor \gamma}\,\lor\text{-I}_1}{\Gamma, \phi \lor (\psi \lor \gamma), \psi \lor \gamma \vdash (\phi \lor \psi) \lor \gamma}\,\lor\text{-E}$$

$\mathcal{D}'$

$$\dfrac{\dfrac{\dfrac{\psi \in \Gamma, \phi \lor (\psi \lor \gamma), \psi \lor \gamma, \psi}{\Gamma, \phi \lor (\psi \lor \gamma), \psi \lor \gamma, \psi \vdash \psi}\,\textsc{axiom}}{\Gamma, \phi \lor (\psi \lor \gamma), \psi \lor \gamma, \psi \vdash \phi \lor \psi}\,\lor\text{-I}_1}{\Gamma, \phi \lor (\psi \lor \gamma), \psi \lor \gamma, \psi \vdash (\phi \lor \psi) \lor \gamma}\,\lor\text{-I}_2$$

B LindenbaumTarski.agda

```
{-# OPTIONS --cubical #-}
module LindenbaumTarski where

open import Cubical.HITs.SetQuotients.Base
open import Cubical.HITs.SetQuotients.Properties

open import Cubical.Foundations.Prelude hiding (_∧_; _∨_)

open import Cubical.Relation.Binary.Base

open import Cubical.Data.Nat.Base
open import Cubical.Data.Prod.Base

open import Cubical.Algebra.DistLattice.Base

-- Definition: Formula

data Formula : Type where
  _∧_ : Formula → Formula → Formula
  _∨_ : Formula → Formula → Formula
  ¬_  : Formula → Formula
  const : ℕ → Formula
  ⊥   : Formula
  ⊤   : Formula

infix 35 _∧_
infix 30 _∨_
infixl 36 ¬_
infix 20 _⊢_
infix 23 _∶_

-- Definition: Context

data ctxt : Type where
  ∅   : ctxt
  _∶_ : ctxt → Formula → ctxt

-- Definition: Lookup

data _∈_ : Formula → ctxt → Type where
  Z : ∀ {Γ ϕ} → ϕ ∈ Γ ∶ ϕ
  S : ∀ {Γ ϕ ψ} → ϕ ∈ Γ → ϕ ∈ Γ ∶ ψ
```

```
-- {-# NO_POSITIVITY_CHECK #-}

-- Definition: Provability

data _⊢_ : ctxt → Formula → Type where

∧-I : {Γ : ctxt} {ϕ ψ : Formula}
    → Γ ⊢ ϕ
    → Γ ⊢ ψ
    → Γ ⊢ ϕ ∧ ψ

∧-E₁ : {Γ : ctxt} {ϕ ψ : Formula}
     → Γ ⊢ ϕ ∧ ψ
     → Γ ⊢ ϕ

∧-E₂ : {Γ : ctxt} {ϕ ψ : Formula}
     → Γ ⊢ ϕ ∧ ψ
     → Γ ⊢ ψ

∨-I₁ : {Γ : ctxt} {ϕ ψ : Formula}
     → Γ ⊢ ψ
     → Γ ⊢ ϕ ∨ ψ

∨-I₂ : {Γ : ctxt} {ϕ ψ : Formula}
     → Γ ⊢ ϕ
     → Γ ⊢ ϕ ∨ ψ

∨-E : {Γ : ctxt} {ϕ ψ γ : Formula}
    → Γ ⊢ ϕ ∨ ψ
    → Γ : ϕ ⊢ γ
    → Γ : ψ ⊢ γ
    → Γ ⊢ γ

--  ∨-E : {Γ : ctxt} {ϕ ψ γ : Formula}
--        → Γ ⊢ ϕ ∨ ψ
--        → (Γ ⊢ ϕ → Γ ⊢ γ)
--        → (Γ ⊢ ψ → Γ ⊢ γ)
--        → Γ ⊢ γ

¬-I : {Γ : ctxt} {ϕ : Formula}
    → Γ : ϕ ⊢ ⊥
    → Γ ⊢ ¬ ϕ
```

```
--  ¬-I : {Γ : ctxt} {ϕ : Formula}
--        → (Γ ⊢ ϕ → Γ ⊢ ⊥)
--        → Γ ⊢ ¬ ϕ

¬-E : {Γ : ctxt} {ϕ : Formula}
    → Γ ⊢ ϕ
    → Γ ⊢ ¬ ϕ
    → Γ ⊢ ⊥

⊥-E : {Γ : ctxt} {ϕ : Formula}
    → Γ ⊢ ⊥
    → Γ ⊢ ϕ

⊤-I : {Γ : ctxt}
    → Γ ⊢ ⊤

axiom : {Γ : ctxt} {ϕ : Formula}
      → ϕ ∈ Γ
      → Γ ⊢ ϕ

LEM : {Γ : ctxt} {ϕ : Formula}
    → Γ ⊢ ϕ ∨ ¬ ϕ

weakening : {Γ : ctxt} {ϕ ψ : Formula}
          → Γ ⊢ ψ
          → Γ : ϕ ⊢ ψ

exchange : {Γ : ctxt} {ϕ ψ γ : Formula}
         → (Γ : ϕ) : ψ ⊢ γ
         → (Γ : ψ) : ϕ ⊢ γ

--  contraction : {Γ : ctxt} {ϕ ψ : Formula} → ((Γ : ϕ) : ϕ) ⊢ ψ → (Γ : ϕ) ⊢ ψ

module _ {Γ : ctxt} where

  infixl 25 ¬/_

  ----------------------------------------
  -- Properties of propositional calculus
  ----------------------------------------

  -- Commutativity on ∧
```

```
∧-comm : ∀ {φ ψ : Formula} → Γ : φ ∧ ψ ⊢ ψ ∧ φ
∧-comm = ∧-I (∧-E₂ (axiom Z)) (∧-E₁ (axiom Z))

--  ∧-comm : ∀ {φ ψ : Formula} → Γ ⊢ φ ∧ ψ → Γ ⊢ ψ ∧ φ
--  ∧-comm x = ∧-I (∧-E₂ x) (∧-E₁ x)

-- Commutativity on ∨

∨-comm : {φ ψ : Formula} → Γ : φ ∨ ψ ⊢ ψ ∨ φ
∨-comm = ∨-E (axiom Z) (∨-I₁ (axiom Z)) (∨-I₂ (axiom Z))

--  ∨-comm : {φ ψ : Formula} → Γ ⊢ φ ∨ ψ → Γ ⊢ ψ ∨ φ
--  ∨-comm x = ∨-E x ∨-I₁ ∨-I₂

-- Associativity on ∧

∧-assoc1 : ∀ {φ ψ γ : Formula} → Γ : φ ∧ (ψ ∧ γ) ⊢ (φ ∧ ψ) ∧ γ
∧-assoc1 = ∧-I (∧-I (∧-E₁ (axiom Z))
                    (∧-E₁ (∧-E₂ (axiom Z))))
                (∧-E₂ (∧-E₂ (axiom Z)))

--  ∧-assoc1 : ∀ {φ ψ γ : Formula} → Γ ⊢ φ ∧ (ψ ∧ γ) → Γ ⊢ (φ ∧ ψ) ∧ γ
--  ∧-assoc1 x = ∧-I (∧-I (∧-E₁ x) (∧-E₁ (∧-E₂ x)))
--                   (∧-E₂ (∧-E₂ x))

∧-assoc2 : ∀ {φ ψ γ : Formula} → Γ : (φ ∧ ψ) ∧ γ ⊢ φ ∧ (ψ ∧ γ)
∧-assoc2 = ∧-I (∧-E₁ (∧-E₁ (axiom Z)))
                (∧-I (∧-E₂ (∧-E₁ (axiom Z)))
                    (∧-E₂ (axiom Z)))

--  ∧-assoc2 : ∀ {φ ψ γ : Formula} → Γ ⊢ (φ ∧ ψ) ∧ γ → Γ ⊢ φ ∧ (ψ ∧ γ)
--  ∧-assoc2 x = ∧-I (∧-E₁ (∧-E₁ x))
--                   (∧-I (∧-E₂ (∧-E₁ x)) (∧-E₂ x))

-- Associativity on ∨

∨-assoc1 : ∀ {φ ψ γ : Formula} → Γ : φ ∨ (ψ ∨ γ) ⊢ (φ ∨ ψ) ∨ γ
∨-assoc1 = ∨-E (axiom Z)
                (∨-I₂ (∨-I₂ (axiom Z)))
                (∨-E (axiom Z)
```

```
                   (∨-I₂ (∨-I₁ (axiom Z)))
                   (∨-I₁ (axiom Z)))

--  ∨-assoc1 : ∀ {φ ψ γ : Formula} → Γ ⊢ φ ∨ (ψ ∨ γ) → Γ ⊢ (φ ∨ ψ) ∨ γ
--  ∨-assoc1 x = ∨-E x (λ y → ∨-I₂ (∨-I₂ y))
--                      λ y → ∨-E y (λ z → ∨-I₂ (∨-I₁ z)) ∨-I₁

∨-assoc2 : ∀ {φ ψ γ : Formula} → Γ : (φ ∨ ψ) ∨ γ ⊢ φ ∨ (ψ ∨ γ)
∨-assoc2 = ∨-E (axiom Z)
               (∨-E (axiom Z)
                   (∨-I₂ (axiom Z))
                   (∨-I₁ (∨-I₂ (axiom Z))))
               (∨-I₁ (∨-I₁ (axiom Z)))

--  ∨-assoc2 : ∀ {φ ψ γ : Formula} → Γ ⊢ (φ ∨ ψ) ∨ γ → Γ ⊢ φ ∨ (ψ ∨ γ)
--  ∨-assoc2 x = ∨-E x (λ y → ∨-E y ∨-I₂ λ z → ∨-I₁ (∨-I₂ z)) λ y → ∨-I₁ (∨-I₁ y)

-- Distributivity over ∧

∧-dist1 : ∀ {φ ψ γ : Formula} → Γ : φ ∧ (ψ ∨ γ) ⊢ (φ ∧ ψ) ∨ (φ ∧ γ)
∧-dist1 = ∨-E (∧-E₂ (axiom Z))
              (∨-I₂ (∧-I (weakening (∧-E₁ (axiom Z))) (axiom Z)))
              (∨-I₁ (∧-I (weakening (∧-E₁ (axiom Z))) (axiom Z)))

--  ∧-dist1 : ∀ {φ ψ γ : Formula} → Γ ⊢ φ ∧ (ψ ∨ γ) → Γ ⊢ (φ ∧ ψ) ∨ (φ ∧ γ)
--  ∧-dist1 x = ∨-E (∧-E₂ x) (λ y → ∨-I₂ (∧-I (∧-E₁ x) y))
--                            λ y → ∨-I₁ (∧-I (∧-E₁ x) y)

∧-dist2 : ∀ {φ ψ γ : Formula} → Γ : (φ ∧ ψ) ∨ (φ ∧ γ) ⊢ φ ∧ (ψ ∨ γ)
∧-dist2 = ∧-I (∨-E (axiom Z)
                  (∧-E₁ (axiom Z))
                  (∧-E₁ (axiom Z)))
              (∨-E (axiom Z)
                  (∨-I₂ (∧-E₂ (axiom Z)))
                  (∨-I₁ (∧-E₂ (axiom Z))))

--  ∧-dist2 : ∀ {φ ψ γ : Formula} → Γ ⊢ (φ ∧ ψ) ∨ (φ ∧ γ) → Γ ⊢ φ ∧ (ψ ∨ γ)
--  ∧-dist2 x = ∧-I (∨-E x ∧-E₁ ∧-E₁)
--                   (∨-E x (λ y → ∨-I₂ (∧-E₂ y))
--                            λ y → ∨-I₁ (∧-E₂ y))

-- Distributivity over ∨
```

```
∨-dist1 : ∀ {φ ψ γ : Formula} → Γ : φ ∨ (ψ ∧ γ) ⊢ (φ ∨ ψ) ∧ (φ ∨ γ)
∨-dist1 = ∨-E (axiom Z)
              (∧-I (∨-I₂ (axiom Z))
                   (∨-I₂ (axiom Z)))
              (∧-I (∨-I₁ (∧-E₁ (axiom Z)))
                   (∨-I₁ (∧-E₂ (axiom Z))))

--  ∨-dist1 : ∀ {φ ψ γ : Formula} → Γ ⊢ φ ∨ (ψ ∧ γ) → Γ ⊢ (φ ∨ ψ) ∧ (φ ∨ γ)
--  ∨-dist1 x = ∨-E x (λ y → ∧-I (∨-I₂ y) (∨-I₂ y))
--                     λ y → ∧-I (∨-I₁ (∧-E₁ y)) (∨-I₁ (∧-E₂ y))

∨-dist2 : ∀ {φ ψ γ : Formula} → Γ : (φ ∨ ψ) ∧ (φ ∨ γ) ⊢ φ ∨ (ψ ∧ γ)
∨-dist2 = ∨-E (∧-E₁ (axiom Z))
              (∨-I₂ (axiom Z))
              (∨-E (∧-E₂ (weakening (axiom Z)))
                   (∨-I₂ (axiom Z))
                   (∨-I₁ (∧-I (weakening (axiom Z)) (axiom Z))))

--  ∨-dist2 : ∀ {φ ψ γ : Formula} → Γ ⊢ (φ ∨ ψ) ∧ (φ ∨ γ) → Γ ⊢ φ ∨ (ψ ∧ γ)
--  ∨-dist2 x = ∨-E (∧-E₁ x) ∨-I₂ λ y → ∨-E (∧-E₂ x) ∨-I₂  λ z → ∨-I₁ (∧-I y z)

------------------------------------------------------
-- Defining relation where two formulas are related
-- if they are provably equivalent. Then proving that
-- the relation is an equivalence relation by proving
-- it is reflexive, symmetric and transitive.
------------------------------------------------------

_∼_ : Formula → Formula → Type
φ ∼ ψ = Γ : φ ⊢ ψ × Γ : ψ ⊢ φ

--  _∼_ : Formula → Formula → Type
--  φ ∼ ψ = (Γ ⊢ φ → Γ ⊢ ψ) × (Γ ⊢ ψ → Γ ⊢ φ)

∼-refl : ∀ (φ : Formula) → φ ∼ φ
∼-refl _ = axiom Z , (axiom Z)

--  ∼-refl : ∀ (φ : Formula) → φ ∼ φ
--  ∼-refl _ = (λ x → x) , (λ x → x)

∼-sym : ∀ {φ ψ : Formula} → φ ∼ ψ → ψ ∼ φ
∼-sym (A , B) = B , A
```

```
⊢trans : ∀ {φ ψ γ : Formula} → Γ : φ ⊢ γ → Γ : γ ⊢ ψ → Γ : φ ⊢ ψ
⊢trans A B = ∨-E (∨-I₂ A) (exchange (weakening B)) (axiom Z)

--  ⊢trans : ∀ {φ ψ γ : Formula} → (Γ ⊢ φ → Γ ⊢ ψ) → (Γ ⊢ ψ → Γ ⊢ γ) → (Γ ⊢ φ -
--  ⊢trans x y z = y (x z)

∼-trans : ∀ {φ ψ γ : Formula} → φ ∼ γ → γ ∼ ψ → φ ∼ ψ
∼-trans x y = ⊢trans (proj₁ x) (proj₁ y) , ⊢trans (proj₂ y) (proj₂ x)

-----------------------------------------------------------
-- Lindenbaum-Tarski algebra is defined as the quotioent
-- algebra obtained by factoring the algebra of formulas
-- by the defined equivalence relation.
-----------------------------------------------------------

LindenbaumTarski : Type
LindenbaumTarski = Formula / _∼_

---------------------------------------------------
-- The equivalence relation ∼ respects operations
---------------------------------------------------

∼-respects-∧ : ∀ (a a' b b' : Formula) → a ∼ a' → b ∼ b' → (a ∧ b) ∼ (a' ∧ b')
∼-respects-∧ a a' b b' x y = ∧-I (⊢trans (∧-E₁ (axiom Z)) (proj₁ x))
                                 (⊢trans (∧-E₂ (axiom Z)) (proj₁ y)) ,
                             ∧-I (⊢trans (∧-E₁ (axiom Z)) (proj₂ x))
                                 (⊢trans (∧-E₂ (axiom Z)) (proj₂ y))

--  ∼-respects-∧ : ∀ (a a' b b' : Formula) → a ∼ a' → b ∼ b' → (a ∧ b) ∼ (a' ∧ b'
--  ∼-respects-∧ a a' b b' x y = (λ z → ∧-I (proj₁ x (∧-E₁ z)) (proj₁ y (∧-E₂ z))) ,
--                               (λ z → ∧-I (proj₂ x (∧-E₁ z)) (proj₂ y (∧-E₂ z)))

∼-respects-∨ : ∀ (a a' b b' : Formula) → a ∼ a' → b ∼ b' → (a ∨ b) ∼ (a' ∨ b')
∼-respects-∨ a a' b b' x y = ∨-E (axiom Z)
                                 (exchange (weakening (∨-I₂ (proj₁ x))))
                                 (exchange (weakening (∨-I₁ (proj₁ y)))) ,
                             ∨-E (axiom Z)
                                 (exchange (weakening (∨-I₂ (proj₂ x))))
                                 (exchange (weakening (∨-I₁ (proj₂ y))))

--  ∼-respects-∨ : ∀ (a a' b b' : Formula) → a ∼ a' → b ∼ b' → (a ∨ b) ∼ (a' ∨ b'
--  ∼-respects-∨ a a' b b' A B = (λ x → ∨-E x (λ x → ∨-I₂ (proj₁ A x)) λ x → ∨-I₁ (
```

```
∼-respects-¬ : ∀ (a a' : Formula) → a ∼ a' → (¬ a) ∼ (¬ a')
∼-respects-¬ a a' x = ¬-I (¬-E (exchange (weakening (proj₂ x))) (weakening (axiom Z))) ,
                      ¬-I (¬-E (exchange (weakening (proj₁ x))) (weakening (axiom Z)))


--  ∼-respects-¬ : ∀ (a a' : Formula) → a ∼ a' → (¬ a) ∼ (¬ a')
--  ∼-respects-¬ a a' A = (λ x → ¬-I (λ y → ¬-E (proj₂ A y) x)), (λ x → ¬-I (λ y –


-------------------------------------------------------------------
-- Definition: Binary operations and propositional constants in LT
-------------------------------------------------------------------


_⋀_ : LindenbaumTarski → LindenbaumTarski → LindenbaumTarski
A ⋀ B = setQuotBinOp ∼-refl ∼-refl _∧_ ∼-respects-∧ A B


_⋁_ : LindenbaumTarski → LindenbaumTarski → LindenbaumTarski
A ⋁ B = setQuotBinOp ∼-refl ∼-refl _∨_ ∼-respects-∨ A B


¬/_ : LindenbaumTarski → LindenbaumTarski
¬/ A = setQuotUnaryOp ¬_ ∼-respects-¬ A


⊤/ : LindenbaumTarski
⊤/ = [ ⊤ ]


⊥/ : LindenbaumTarski
⊥/ = [ ⊥ ]


  -- Commutativity on ⋀
  ⋀-comm : ∀ (A B : LindenbaumTarski) → A ⋀ B ≡ B ⋀ A
  ⋀-comm = elimProp2 (λ _ _ → squash/ _ _) λ φ ψ → eq/ _ _ (∼-sym (∧-comm , ∧-comm))

  -- Commutativity on ⋁
  ⋁-comm : ∀ (A B : LindenbaumTarski) → A ⋁ B ≡ B ⋁ A
  ⋁-comm = elimProp2 (λ _ _ → squash/ _ _) λ φ ψ → eq/ _ _ (∼-sym (∨-comm , ∨-comm))

  -- Associativity on ⋀
  ⋀-assoc : ∀ (A B C : LindenbaumTarski) → A ⋀ (B ⋀ C) ≡ (A ⋀ B) ⋀ C
  ⋀-assoc = elimProp3 (λ _ _ _ → squash/ _ _) λ _ _ _ → eq/ _ _ (∧-assoc1 , ∧-assoc2)

  --Associativity on ⋁
  ⋁-assoc : ∀ (A B C : LindenbaumTarski) → A ⋁ (B ⋁ C) ≡ (A ⋁ B) ⋁ C
  ⋁-assoc = elimProp3 (λ _ _ _ → squash/ _ _) λ _ _ _ → eq/ _ _ (∨-assoc1 , ∨-assoc2)
```

```
  -- Distributivity over ⋀
  ⋀-dist : ∀ (A B C : LindenbaumTarski) → A ⋀ (B ⋁ C) ≡ (A ⋀ B) ⋁ (A ⋀ C)
  ⋀-dist = elimProp3 (λ _ _ _ → squash/ _ _) λ _ _ _ → eq/ _ _ (∧-dist1 , ∧-dist2)

  --Distributivity over ⋁
  ⋁-dist : ∀ (A B C : LindenbaumTarski) → A ⋁ (B ⋀ C) ≡ (A ⋁ B) ⋀ (A ⋁ C)
  ⋁-dist = elimProp3 (λ _ _ _ → squash/ _ _) λ _ _ _ → eq/ _ _ (∨-dist1 , ∨-dist2)


  -- Definition: Superweakening
  superweakening : ∀ (Γ : ctxt) → Γ ⊢ ⊤
  superweakening ∅ = ⊤-I
  superweakening (Δ ∶ x) = weakening (superweakening Δ)


  -- Absorbtion law ⋁
  ⋁-abs : ∀ (A B : LindenbaumTarski) → (A ⋀ B) ⋁ B ≡ B
  ⋁-abs = elimProp2 (λ _ _ → squash/ _ _) λ _ _ → eq/ _ _ (∨-E (axiom Z) (∧-E₂ (axiom Z)) (axiom Z) ,
--  ⋁-abs : ∀ (A B : LindenbaumTarski) → (A ⋀ B) ⋁ B ≡ B
--  ⋁-abs = elimProp2 (λ _ _ → squash/ _ _) λ _ _ → eq/ _ _ ((λ x → ∨-E x ∧-E₂ λ a →

  -- Absorbtion law ⋀
  ⋀-abs : ∀ (A B : LindenbaumTarski) → A ⋀ (A ⋁ B) ≡ A
  ⋀-abs = elimProp2 (λ _ _ → squash/ _ _) λ _ _ → eq/ _ _ (∧-E₁ (axiom Z) , ∧-I (axiom Z) (∨-I₂ (axiom
--  ⋀-abs : ∀ (A B : LindenbaumTarski) → A ⋀ (A ⋁ B) ≡ A
--  ⋀-abs = elimProp2 (λ _ _ → squash/ _ _) λ _ _ → eq/ _ _ (∧-E₁ , λ x → ∧-I x (∨-I₂

  -- Identity law ⋁
  ⋁-id : ∀ (A : LindenbaumTarski) → A ⋁ ⊥/ ≡ A
  ⋁-id = elimProp (λ _ → squash/ _ _) λ _ → eq/ _ _ (∨-E (axiom Z) (axiom Z) (⊥-E (axiom Z)) , ∨-I₂ (a
--  ⋁-id : ∀ (A : LindenbaumTarski) → A ⋁ ⊥/ ≡ A
--  ⋁-id = elimProp (λ _ → squash/ _ _) λ _ → eq/ _ _ ((λ x → ∨-E x (λ y → y) ⊥-E)

  -- Identity law ⋀
  ⋀-id : ∀ (A : LindenbaumTarski) → A ⋀ ⊤/ ≡ A
  ⋀-id = elimProp (λ _ → squash/ _ _) λ _ → eq/ _ _ (∧-E₁ (axiom Z) , ∧-I (axiom Z) (superweakening _))
--  ⋀-id : ∀ (A : LindenbaumTarski) → A ⋀ ⊤/ ≡ A
--  ⋀-id = elimProp (λ _ → squash/ _ _) λ _ → eq/ _ _ (∧-E₁ , λ x → ∧-I x (superweake


  -------------------------------------------------------------
  -- If ϕ is provable in Γ then [ϕ] should be the same as ⊤/.
  -- We can view this as a form of soundness.
  -------------------------------------------------------------
  sound : ∀ {ϕ : Formula} → Γ ⊢ ϕ → [ ϕ ] ≡ ⊤/
  sound x = eq/ _ _ (superweakening _ , weakening x)
--  sound : ∀ {ϕ : Formula} → Γ ⊢ ϕ → [ ϕ ] ≡ ⊤/
--  sound x = eq/ _ _ ((λ _ → superweakening _) , λ _ → x )
```

```
--------------------------------------------------------------
-- By proving the Lindenbaum-Tarski algebra can be viewed as
-- a distributive lattice, we prove that it is also boolean.
--------------------------------------------------------------
isSet-LT : ∀ (A B : LindenbaumTarski) → isProp(A ≡ B)
isSet-LT A B = squash/ _ _

--  LT-isDistLattice : IsDistLattice ⊥/ ⊤/ _⋁_ _⋀_
--  LT-isDistLattice = makeIsDistLattice∧lOver∨l isSet-LT ⋁-assoc ⋁-id ⋁-comm ⋀-as

LindenbaumTarski-DistLattice : DistLattice _
LindenbaumTarski-DistLattice = makeDistLattice∧lOver∨l ⊥/ ⊤/ _⋁_ _⋀_ isSet-LT ⋁-assoc ⋁-id ⋁-comm


{-
open DistLattice         -- DistLattice (not in scope) -> DistLatticeStr? IsDistLatti

LindenbaumTarski-Boolean : (x y : fst LindenbaumTarski-DistLattice) -> x ∨l y ≡ 1l
LindenbaumTarski-Boolean x y = {!!}
-}


-- Complemented
⋀-inv : ∀ (A : LindenbaumTarski) → A ⋀ ¬/ A ≡ ⊥/
⋀-inv = elimProp (λ _ → squash/ _ _) λ _ → eq/ _ _ (¬-E (∧-E₁ (axiom Z)) (∧-E₂ (axiom Z)) , ⊥-E (axi
--  ⋀-inv : ∀ (A : LindenbaumTarski) → A ⋀ ¬/ A ≡ ⊥/
--  ⋀-inv = elimProp (λ _ → squash/ _ _) λ _ → eq/ _ _ ((λ x → ¬-E (∧-E₁ x) (∧-E₂ x))

⋁-inv : ∀ (A : LindenbaumTarski) → A ⋁ ¬/ A ≡ ⊤/
⋁-inv = elimProp (λ _ → squash/ _ _) λ _ → eq/ _ _ (superweakening _ , LEM)
--  ⋁-inv : ∀ (A : LindenbaumTarski) → A ⋁ ¬/ A ≡ ⊤/
--  ⋁-inv = elimProp (λ _ → squash/ _ _) λ _ → eq/ _ _ ((λ x → superweakening _) , λ
```