Formalising an Intuitionistic Type Theory in Lean

COMP30014 Research Project
William Price, Student ID: 917093
Supervisors: Daniel Murfet, Toby Murray.

June 26, 2020

Abstract

Proofs in formal systems are inherently laborious, and it is difficult to present complete proofs within formal systems without completely sacrificing the presentation. We implement an intuitionistic type theory in Lean, an interactice proof assistant, and demonstrate its utility in guiding, automating and organising formal proofs.

Contents

1 The Intuitionistic Type Theory

1.1 Lean Syntax

Lean is an interactive theorem prover [1] based on the Calculus of Inductive Constructions[6] (CIC). In Lean, everything has a type, and we denote this relationship, `a : A`, meaning `a` has type `A`. There is a type of types called `Type`, a member of which is `Prop`, so can write `Prop : Type`. Any member of `Type` can be defined to have members. In particular `Prop` has members, which are propositions.

1.2 ITT vs. LITT

We implement the intuitionistic type theory defined in Lambek and Scott, Introduction to Higher Order Categorical Logic [2], which we abbreviate **ITT**, with the exception that we do not include a type of natural numbers by default. We also made significant use of Daniel Murfet's notes on **ITT** in developing this type theory.

We abbreviate our Lean implentation of **ITT** by **LITT**. Where the **ITT** construction describes an extendable type theory which can have more types, terms, deduction rules, and non-trivial identifications of types and terms, we implement only the underlying *pure* type theory in **LITT**.

There are many subtleties about the definitions related to use of variables in terms, which cannot be ignored in Lean, so our implementation diverges from Lambek and Scott in key places. We present both definitions of each concept in parallel in order to contextualise the decisions we make.

1.2.1 Meta Theories

A specification or discussion of a formal theory must always take place within some meta-theory. For example, it is not possible to make a statement like "If p and q are propositions, then $p \Rightarrow q$ is a proposition", without any a priori semantics behind concepts like "If", "and", "then" - and a structural understanding of such statements. Once you have a formal system, it is then possible to specify and discuss other formal systems within it, and possibly even vice-versa. So how do you get started? Essentially you have to rely on intuitively meaningful and simple decision procedures, such as the rule above concerning proposition constructions. This is the notion of "effective computability" and is the subject of the Church Turing thesis. For this project, our meta-theory is the formal system, Lean, so these decision procedures are well-defined.

It is worth paying attention to the strength of the meta-theory used to specify a formal system, and what features we may be asking of the meta-theory without realising. For example, one usually expects to produce as many fresh "variables" as one needs when manipulating terms. Is this a reasonable expectation? Maybe, maybe not, but it is worth spotlighting such manners. We benefit from studying one formal system within another by making such subtleties explicit. In order to not overshadow the main intentions with definitional subtleties, we present both the intuitive, **ITT** definitions, and the concrete **LITT** definitions of each concept (though some are unique to **LITT**).

Our type theory manifests in Lean via 4 inductively defined types, the first of which is the type of types. Lean itself already has a type `Type`, so we use the lowercase, `type`, for the type of types in our theory. The relationship here is `type : Type`, i.e. "the type of `type` is `Type`".

Since Lean has an extensible parser, for each definition, we define unicode-based notation to make the syntax more readable. Examples will follow where needed.

Definition 1.1. (**ITT**) A type theory has a class of *types*, including basic types $\Omega, \mathbb{1}$, closed under the following constructions

- If A is a type then $\mathscr{P}A$ is a type
- If A, B are types then $A \times B$ is a type

These types will be inhabited by *terms/variables*, and we introduce this relationship $a : A$, meaning a is a term/variable of type A. For each type A, there is an unlimited supply of *variables*, $x : A$.

Ω is thought of as the type of propositions, and $\mathbb{1}$ will have a unique inhabitant (up to provable equivalence).

Definition 1.2. (**LITT**)

```
/- The inductive definition of a `type`-/
inductive type : Type
| Unit | Omega | Prod (A B : type)| Pow (A : type)
```

Working within lean, we refine the a priori notion of a "class" of types to mean a `Type` called `type`, which is inductively defined via four constructors. The first two take no arguments (they are basic members of `type`), and the other two construct new types from old ones. We also define related notation.

```
notation `Ω` := type.Omega
notation `𝟙` := type.Unit
infix `×`:max := type.Prod
notation `𝒫`A :max := type.Pow A

#check type.Prod type.Omega (type.Pow type.Unit) -- type
#check Ω × (𝒫 𝟙) -- type
```

It's important to note that there is no semantic content in either definition. Nothing yet constrains the Unit type to only have one member, nor the Omega type to be logical, nor the powerset or product constructors to "make" the new type from old ones. It's only once we declare what constitutes a "proof" statement involving terms of these types that they adopt a consistent meaning. The only feature we get at this point is definitional inequality between types with different derivations, meanining there are no non-trivial identifications between types such as `Pow Unit` and `Omega`.

Definition 1.3. (**LITT**) A type theory has a class of *terms*[1], and to each is associated a unique *type*. We denote this relationship, $a : A$, meaning a is of type A. The class of terms is closed under the following constructions (which can also be performed on variables). Let A, B be types and $(a : A)$, $(b : B)$, $(\alpha : \mathscr{P}A)$, $(p, q, \varphi : \Omega)$.

Basic Terms

- $* : \mathbb{1}$ ("star")
- $\top : \Omega$ ("true/top")
- $\bot : \Omega$ ("false/bottom")

Propositional connectives

- $p \wedge q : \Omega$ ("conjunction")
- $p \vee q : \Omega$ ("disjunction")
- $p \Rightarrow q : \Omega$ ("implication")
- $a \in \alpha : \Omega$ ("elementhood")

Quantifiers

Let x be a variable of type A.

- $(\forall x \in A)\ \varphi : \Omega$ ("Universal Quantification")
- $(\exists x \in A)\ \varphi : \Omega$ ("Existential Quantification")
- $\{x \in A \mid \varphi\} : \mathscr{P}A$ ("Set Comprehension")

Pairs

- $\langle a, b \rangle : A \times B$

It is worth understanding the quantifier constructions in greater detail, as there is potential for confusion in the use of the $\in$-elementhood symbol.

Let A be a type, x a variable of type A and $\varphi : \Omega$. Note that φ may or not contain x as a free variable (see 1.3). Then the following are also terms,

- $(\forall x \in\ A)\ \varphi : \Omega$ ("Universal Quantification")
- $(\exists x \in\ A)\ \varphi : \Omega$ ("Existential Quantification")
- $\{x \in A \mid \varphi\} : \mathscr{P}A$ ("Set Comprehension")

We have highlighted in red the purely-syntactic content introduced by each construction. In particular, the $\in$ in $\forall x \in A$ should not be confused with the elementhood propositional connective. It is purely a syntactical feature to which indicates the type of the variable x being quantified over.

An important subtlety is that in declaring that the variable x has type A, we are assuming any appearance of a variable with name x is actually the same variable with the same type, A. More correctly, each type has (by meta-theoretical assumption) an unlimited supply of variables of each type, and to each such variable is associated a unique type. In practice we would allow the use of a variable name with different types, just not within the same term (or arrangement of terms to be safe). However, we will need to be more careful once Lean is our metatheory.

[1] One should actually define *preterms* as we define *terms* and then account for α-equivalence (see 1.3). We skip this discussion (noting we are defining "too many" terms) as our Lean interpretation avoids the need for α-equivalence altogether.

1.3 Variables

Definition 1.4. (**ITT**) To each term α we associate a finite set $FV(\alpha)$ called the *free variables* of α. This set is defined recursively as follows.

- $FV(*) = \emptyset$
- $FV(\top) = \emptyset$
- $FV(\bot) = \emptyset$
- $FV(p \wedge q) = FV(p) \cup FV(q)$
- $FV(p \vee q) = FV(p) \cup FV(q)$
- $FV(p \Rightarrow q) = FV(p) \cup FV(q)$
- $FV(a \in \alpha) = FV(a) \cup FV(\alpha)$
- $FV((\forall x \in A)\varphi) = FV(\varphi) \setminus \{x\}$
- $FV((\exists x \in A)\varphi) = FV(\varphi) \setminus \{x\}$
- $FV(\{x \in A \mid \varphi\}) = FV(\varphi) \setminus \{x\}$

Within the term ϕ, the free occurences of the variable x are said to be *captured* by a quantifier in the expressions $(\forall x \in A)\varphi$, $(\exists x \in A)\varphi$, $\{x \in A \mid \varphi\}$. A term with no variables is called *closed.*

Definition 1.5. (**ITT**) For any term φ and variables x, y of some type, by $\varphi[y/x]$, we mean the resulting term when all free occurences of x are replaced with y.

We define an equivalence relation $=_\alpha$ as the smallest equivalence relation on terms, closed under all term-formation rules, and for any variables $x, y : A$, $\varphi : \Omega$,

- $(\forall x \in A)\varphi =_\alpha (\forall y \in A)\varphi[y/x]$
- $(\exists x \in A)\varphi =_\alpha (\forall y \in A)\varphi[y/x]$
- $\{x \in A \mid \varphi\} =_\alpha \{x \in A \mid \varphi[y/x]\}$

as long as no free occurence of x would become captured by a quantifier in $\varphi[y/x]$.

It is necessary to form α-equivalence classes of the terms (they would be called preterms) we defined in **ITT**, in order to identify (what we intend to be) semantically equivalent terms. Alternatively, we could leave α-equivalent terms distinct, and add a proof-formation rule that allows conversion between such terms. In practice (in **LITT**) we avoid the need for α-equivalence entirely by using De Bruijn indices.

1.4 De Bruijn Indices

The usual named-variable approach to quantification excels in readability, but lacks canonicity. Consider the α-equivalent terms $\varphi_1 = \{x_1 \in A \mid (\exists y_1 \in A)(\forall z_1 \in A) \ldots x_1 \ldots z_1 \ldots y_1\}$, and $\varphi_2 = \{x_2 \in A \mid (\exists y_2 \in A)(\forall z_2 \in A) \ldots x_2 \ldots z_2 \ldots y_2\}$ denoting some quantified terms where $x_1, y_1, z_1, x_2, y_2, z_2$ are free variables that become captured by the shown quantifiers. If we pay attention to the structure of these expressions, we notice we can canonically name each variable by the index counting the number of quantifiers structurally "between" each variable and its quantifier (given only one occurence of that variable). Essentially, each variable "points" to its intended quantifier, and we remove the naming at the quantifer itself in order to account for multiple instances of the same variable at different *depths*. Our resulting term looks like this, and the indices used here are called De Bruijn indices [4].

$$\{A \mid (\exists A)(\forall A)\ldots 2\ldots 0\ldots 1\}$$

Note now that the syntax of the inner terms are not yet typed - the index chooses which quantifier will capture it's variable, and the quantifier decides the type of the variable. With the goal of eliminating the need for the α-equivalence via De Bruijn indices, this motivates the type-less terms we introduce in LITT.

Definition 1.6. (**LITT**)

```
inductive term : Type
| star : term
| top  : term
| bot  : term
| and  : term → term → term
| or   : term → term → term
| imp  : term → term → term
| elem : term → term → term
| pair : term → term → term
| var  : ℕ → term
| comp : type → term → term
| all  : type → term → term
| ex   : type → term → term
```

Here we define the `Type` of terms. Note that the constructors here are largely independent of `type`, which is odd, since it is natural to define terms along with the type that they inhabit, as we did in 1.3. The lack of this constraint allows the creation of ill-formed terms such as `and star star` (but not things like `and star ex` - since that does not type-check).

We would like to say something akin to the meta-theoretic `star : Unit`. The problem here is that `Unit : type` is a member of the type `type : Type`, but is not a type itself, so we cannot construct members of it.

There are two solutions here. We could parametrise `term` by `type`, such that `term : type → Type`, and allowing each constructor to constrain the `type` of the input terms, for example `pair (A B): term A → term B → term (A × B)` and `star : term Unit`. However, the former solution would require the `term.var` constructor to decide on a `type`, which can't be known until it is quantified over - so we run into the same issue. In order to simplify and separate the *grammar* of terms and *well-formedness* of terms, we allow untyped construction of terms and where necessary, require a proof of its well-formedness.

Accompanying these constructors are notational shorthands.[2] Since some of these notational operators place input terms within quantifiers, we need to *lift* the free-variables (uncaptured de-

[2]Some of these symbols are reserved in Lean, and in practice we actually use slight variations, but we present the most ideal notation here.

bruijn indices) in the term to avoid unintended capture. We provide only the type signature of `lift` here (a more in depth discussion of lifting can be found in [4]).

```
notation `⋆` := term.star
notation `⊤` := term.top
notation `⊥` := term.bot
infix ` ∧ ` :50 := term.and
infix ` ∨ ` :50 := term.or
infix ` ⟹ `:50 := term.imp

def iff (p q: term) := (p ⟹ q) ∧ (q ⟹ p)
infix ` ⇔ `:50 := iff

infix ∈ := term.elem
notation `{' ` A ` | ` φ ` }` := term.comp A φ

notation `⟨` a `,` b `⟩` := term.pair a b

notation `∀` := term.all
notation `∃` := term.ex

/- Coerce natural numbers to variables -/
instance nat_coe_var : has_coe ℕ term := ⟨term.var⟩

-- Now we can write (term.var 0) as just ↑0
#reduce ∃ Ω ↑0 -- term.all type.Omega (term.var 0)
#reduce ↑1 ∨ ∃ Ω ↑2 -- term.or (term.var 1) (term.ex type.Omega (term.var 2))

-- `lift d k φ` increases all variable indices in `φ' at least `k' by `d'
def lift (d : ℕ) : ℕ → term → term := sorry
notation `^` := lift 1 0

-- Leibniz equality
def eq (A:type) (a₁ a₂ : term) : term
  := ∀ (𝒫 A) $ ((^ a₁) ∈ ↑0) ⇔ ((^ a₂) ∈ ↑0)
notation a ` ≃[`:max A `] `:0 b := eq A a b

-- example
#reduce ↑2 ≃[𝟙] ↑0  -- ∀ (𝒫 𝟙) ( (↑3 ∈ ↑0) ⇔ (↑1 ∈ ↑0) )

-- `subst n b φ` replaces the `n`th free variable in `φ` by `b`
def subst : ℕ → term → term → term
  := sorry -- definition ommitted
notation  `[` φ ` / `  b `]` := subst 0 b φ

-- examples
constants p q r : term
#reduce p ∧ (q ∨ r) -- term.and p (term.or q r)
```

```
#reduce ∃ 𝟙 ([↑0 / p] ∧ ∀ 𝟙 (↑1 ≃[𝟙] ↑0))
  -- term.ex type.Unit (term.and p (term.all (type.Pow type.Unit) (iff
 (term.elem (term.var 2) (term.var 0)) (term.elem (term.var 1) (term.var
 0)))))
```

Definition 1.7. We inductively define a family of meta-theoretic propositions, that is, members of Prop[3]. The parameters that appear in braces on the left side of each `:`, are dependent types to the type of each constructor.[4]

```
def context : list type

inductive WF : context → type → term → Prop
| star {Γ}          : WF Γ 𝟙 ⋆
| top  {Γ}          : WF Γ Ω ⊤
| bot  {Γ}          : WF Γ Ω ⊥
| and  {Γ p q}      : WF Γ Ω p → WF Γ Ω q → WF Γ Ω (p ∧ q)
| or   {Γ p q}      : WF Γ Ω p → WF Γ Ω q → WF Γ Ω (p ∨ q)
| imp  {Γ p q}      : WF Γ Ω p → WF Γ Ω q → WF Γ Ω (p ⟹ q)
| elem {Γ A a α}    : WF Γ A a → WF Γ (𝒫 A) α → WF Γ Ω (a ∈ α)
| pair {Γ A B a b} : WF Γ A a → WF Γ B b → WF Γ (A × B) ⟨a,b⟩
| var  {Γ A n}      : (Γ.nth n = some A) → WF Γ A (var n)
| comp {Γ A φ}      : WF (A::Γ) Ω φ → WF Γ (𝒫 A) {A | φ}
| all  {Γ A φ}      : WF (A::Γ) Ω φ → WF Γ Ω (∀ A φ)
| ex   {Γ A φ}      : WF (A::Γ) Ω φ → WF Γ Ω (∃ A φ)
```

Given Γ `: context`, `A : type`, `a : term`, we have just defined `WF Γ A a` to be the proposition that `a` is a well-formed term of type `A` in context Γ. Each constructor produces a member of some `WF Γ A a`, that is, a *proof* of `WF Γ A a`. For example, given any context, we can summon a proof that ⋆ is well-formed of type 𝟙 in that context just by declaring `WF.star`. Well-formedness proofs of terms made from propositional connectives (and pairs) require proofs that each term is well-formed in the same context.

What is a context?

Since De Bruijn index-variables are not typed, we must use provide a mapping of free-variable-indices to types, in order to prevent conflicting types among variables when quantifying. Since such a mapping is just a function `[n] → type` for some natural number `n`, we can just a use a list of types, where the De Bruijn index `n` maps to the nth type in the list. To introduce a proof that a variable term of some `A : type` is well-formed via `WF.var : (Γ.nth n = some A) → WF Γ A (var n)`, we must provide a proof that the n'th type in the context is infact `A`. To introduce a proof that a quantified statement like `∀ A φ` is well-formed of type Ω, we must provide a proof that φ is well-formed to `WF.all`, where `A` is appended to the start of the context - so any variable captured

[3]Prop is the type of Propositions in Lean - it's the meta-theoretic equivalent of Ω. Members `P : Prop` are propositions, and members `p : P` are proofs of `P`

[4]These are implicit arguments to the constructor, and are inferred other arguments provided to the constructor.

by this quantifier has type `A`.[5] Similar constructions apply to the existential and set-comprehension constructions.

Definition 1.8. Given a finite set Γ of free-variables of any types, an **ITT** has a relation, Γ-entailment, on pairs of propositional terms $\varphi, \psi : \Omega$, such that $FV(\varphi), FV(\psi) \subseteq \Gamma$. We write such a relation $\varphi \vdash_\Gamma \psi$, and abbreviate $\varphi \vdash_\emptyset \psi$ to $\varphi \vdash \psi$, $\top \vdash_\Gamma \psi$ to $\vdash_\Gamma \psi$ and $\top \vdash \psi$ to $\vdash \psi$.

Such a Γ-entailment must contain certain axiomatic relations, as well as certain deduction rules which each determine that entailment can be deduced from one or more other entailments. We do not state them all here, as they will just be repeated in the next definition.

A *proof* of some Γ-entailment, $\varphi \vdash_\Gamma \psi$ is a deduction tree beginning with one or more axiomatic relations.

Definition 1.9. (**LITT**)

We define Γ-entailment $\varphi \vdash_\Gamma \psi$ as the inductively defined relation `entails : context → term → term → Prop`.

```
inductive entails : context → term → term → Prop
| axm         {Γ} {p}       : WF Γ Ω p → entails Γ p p
| vac         {Γ} {p}       : WF Γ Ω p → entails Γ p ⊤
| abs         {Γ} {p}       : WF Γ Ω p → entails Γ ⊥ p
| and_intro   {Γ} {p q r}   : entails Γ p q → entails Γ p r → entails Γ p (q ∧ r)
| and_left    {Γ} (p q r)   : entails Γ p (q ∧ r) → entails Γ p q
| and_right   {Γ} (p q r)   : entails Γ p (q ∧ r) → entails Γ p r
| or_intro    {Γ} {p q r}   : entails Γ p r → entails Γ q r → entails Γ (p ∨ q) r
| or_left     {Γ} (p q r)   : entails Γ (p ∨ q) r → entails Γ p r
| or_right    {Γ} (p q r)   : entails Γ (p ∨ q) r → entails Γ q r
| imp_to_and  {Γ} {p q r}   : entails Γ p (q ⟹ r) → entails Γ (p ∧ q) r
| and_to_imp  {Γ} {p q r}   : entails Γ (p ∧ q) r → entails Γ p (q ⟹ r)
| weakening   {Γ} {p q Δ}    : entails Γ p q → entails (Γ ++ Δ) p q
| cut         {Γ} (p c q)   : entails Γ p c → entails Γ c q → entails Γ p q
| all_elim    {Γ} {p φ A}    : entails Γ p (∀ A φ) → entails (A::Γ) (^p) φ
| all_intro   {Γ} {p φ} (A) : entails (A::Γ) (^p) φ → entails Γ p (∀ A φ)
| ex_elim     {Γ} {p φ A}    : entails Γ p (∃ A φ) → entails (A::Γ) (^p) φ
| ex_intro    {Γ} {p φ} (A) : entails (A::Γ) (^p) φ → entails Γ p (∃ A φ)

/- equality of powerset terms is determined by elementhood  (dollar signs make
    application right-associative for less brackets)-/
| extensionality {A} : entails [] ⊤
                        $  ∀ (𝒫 A) $ ∀ (𝒫 A) $ ∀ A
                        $ ((↑0 ∈ ↑2) ⇔ (↑0 ∈ ↑1)) ⟹ (↑1 ≃[𝒫 A] ↑0)

/- Provably equivalent propositions are equal -/
| prop_ext : entails [] ⊤ $ ∀[Ω,Ω] $ (↑1 ⇔ ↑0) ⟹ (↑1 ≃[Ω] ↑0)

/- ⋆ is unique up to provable equivalence -/
| star_unique : entails [] ⊤ $ ∀ 𝟙 (↑0 ≃[𝟙] ⋆)
```

[5] This also lifts all other variables in the context, to correspond to deeper De-Bruijn indices, since we have passed through a quantifier.

```
/- Any term of a product type has a pair representation -/
| pair_rep {A B} : entails [] ⊤
                    $ ∀ (A × B) $ ∃[A,B] $ ↑2 ≃[A × B] ⟨↑1,↑0⟩

/- Terms of product type are distinguished pairwise -/
| pair_distinct {A B} : entails [] ⊤
                         $ ∀ A $ ∀ B $ ∀ A $ ∀ B
                         $ (⟨↑3,↑2⟩ ≃[A × B] ⟨↑1,↑0⟩)
                           ⟹ ((↑3 ≃[A] ↑1) ∧ (↑2 ≃[B] ↑0))

/- Entailments on free variables hold for any well-formed substitution -/
| sub        {Γ} (B b p q) : WF Γ B b
                            → entails (B::Γ) p q
                            → entails Γ [p / b] [q / b]

/- Set comprehension terms are populated precisely by all terms satisfying the condition
   Note: we must lift { A | φ } so it doesn't have ↑0 free -/
| comp       {Γ} (A φ)     : WF (A::Γ) Ω φ → entails Γ ⊤ (∀ A
                            $ (↑0 ∈ (^ { A | φ })) ⇔ φ))
```

Things to note:

- The role of the finite set of free variables Γ, from which both associated terms must source their free-variables from, is now fulfilled by a `context`, which we defined earlier to establish well-formedness on terms.
- Of the 24 constructors, only 5 request a proof of well-formedness of any of the terms appearing in the resulting entailment. These 5 are exactly those through which a new term can be introduced, after which well-formedness is preserved between proofs. The relevant theorem here can be stated and proved precisely in Lean, as we demonstrate in 2.3.

We introduce notation for `entails`. The double turnstile we use is usually distinguished from the single turnstile to represent truth in a model - but our usage is to avoid confusion with Lean's own usage for presenting goals (see 2.1).

```
prefix `⊨`:1 := entails [] ⊤
infix ` ⊨ `:50 := entails []

-- Allows the parser to directly extract the context list
notation φ` ⊨[` Γ:(foldr `,` (h t, list.cons h t) list.nil) `] ` ψ := entails Γ φ ψ
notation `⊨[` Γ:(foldr `,` (h t, list.cons h t) list.nil) `] ` ψ := entails Γ ⊤ ψ

variables p q φ ψ : term

#reduce   ⊨ (p ∨ ¬p)   -- entails [] ⊤ (or p (imp p ⊥))
#reduce q ⊨ (p ∨ ¬p)   -- entails [] q (or p (imp p ⊥))
#reduce   ⊨[Ω,𝟙] p     -- entails [Ω, 𝟙] ⊤ p
#reduce q ⊨[Ω,𝟙] p     -- entails [Ω, 𝟙] q p
```

1.5 Semantics

Through definition 1.9, we have introduced a means of making assertions about well-formed terms in the type theory. For example, given any term `φ : term`, as long as we can produce an `h : WF [] Ω φ` (a proof of its well-formedness in the empty context), we can construct a proof of entailment proposition `Prop`, `entails [] φ φ` (the ITT entailment $\varphi \vdash \varphi$), like this:

```
lemma phi_ent_phi : entails [] φ φ := entails.axm h
```

Similarly, we can also construct a proof of `entails [] φ ⊤`, like this:

```
lemma phi_ent_top : entails [] φ ⊤ := entails.vac h
```

Using both of these, we can show `entails [] φ (φ ∧ ⊤)`.

```
lemma phi_ent_phi_and_top : entails [] φ (φ ∧ ⊤)
  := entails.and_intro phi_ent_phi phi_ent_top
```

Although it's called `entails` - what makes `entails Γ φ ψ` *mean* entailment of ψ from φ in context Γ, and what it the logical content of the associated terms? For now, it's just some parametrised proposition - which can be proven for some term parameters via the constructors. One perspective is to first trust that the term constructions mean what we think they mean, and that well-formed terms of type Ω are in fact propositions. This interpretation then justifies viewing the entailment constructors as valid deduction rules about entailment, including `entails.and_intro` used in the above example. If we inductively suppose the intended meaning of `entails [] φ φ` and `entails [] φ ⊤`, then proofs of these constitute a proof `entails [] φ (φ ∧ ⊤)`.

Alternatively, we can trust the intended meaning of `entails Γ φ ψ` (and Ω), and then interrogate each term construction and derive their meaning from the way they can appear in provable entailments. For example, we can derive the meaning of $\wedge$ (which is `term.and`) via the following constructors.

```
entails.and_intro {Γ} {p q r} : entails Γ p q → entails Γ p r → entails Γ p (q ∧ r)
entails.and_left  {Γ} (p q r) : entails Γ p (q ∧ r) → entails Γ p q
entails.and_right {Γ} (p q r) : entails Γ p (q ∧ r) → entails Γ p r
```

This injection of semantics into terms and entailment happens simultaenously - neither is prior.

2 Proofs in LITT

Formal proofs of entailment in **ITT** are inherently laborious. The axioms and rules of deduction are presented minimally and serve only the purpose of forcing an interpretable meaning onto terms, and many "obvious" consequences of this interpretation require decent sized proofs supporting them. Upon attempting to perform a proof of any theorem with substantial "logical content", it is impossible to avoid frequent detours in proving various associated lemmas. Such lemmas are frequently "obvious", and it becomes extremely tempting to leave them out altogether, in order to not distract from the central proof.

Since a key point of studying formal systems is demonstrating their ability to

2.1 Tactic Proofs

Typically, a proof is constructed backwards, that is, starting at the conclusion, recognising if the conclusion is an axiom, and otherwise applying some deduction rule in reverse to produce one or more new "sub-conclusions"/"goals" to prove. Repeating this process (intelligently), if we eliminate all of the goals as axioms, we can terminate and claim a proof.

This describes exactly the process of performing an interactive *tactic proof* in Lean. At each point in the proof, Lean presents us with the current goal to prove, and we can invoke any constructor or lemma whose conclusion can be unified with the goal. As an example, we prove `lemma phi_ent_phi_and_top : entails [] φ (φ ∧ ⊤)` again.

```
1 lemma phi_ent_phi_and_top (h : WF [] Ω φ): φ ⊨ (φ ∧ ⊤) :=
2 begin
3   apply entails.and_intro,
4   apply entails.axm,
5   exact h,
6   apply entails.vac,
7   exact h
8 end
```

The tactic environment appears between `begin` and `end`. At the beginning of line 3, we are presented with this goal state in a separate windows.

```
  1 goal
  φ : term,
  h : WF [] Ω φ
  ⊢ φ ⊨ φ ∧ ⊤
```

It outlines the two hypotheses we have access to (that φ is a term and `h` is a proof of its well-formedness) and the goal we need to prove, $\varphi \vDash \varphi \wedge \top$. After applying `entails.and_intro : entails Γ p q → entails Γ p r → entails Γ p (q ∧ r)`, the goal is unified with the conclusion and is replaced with two goals, which are exactly the (unified) hypotheses needed to apply `entails.intro`.

```
  2 goals
  φ : term,
  h : WF [] Ω φ
  ⊢ φ ⊨ φ

  φ : term,
  h : WF [] Ω φ
  ⊢ φ ⊨ ⊤
```

The tactic proof then applies the relevant constructors as needed to solve the resulting goals, and the proof `h` of the hypotheses `WF [] Ω φ` is applied as needed.

This environment for theorem proving allows us to control the non-linearity of much larger proofs, since the task of keeping track of auxilliary goals to prove is handled entirely by Lean, and we can just deconstruct and prove each goal as they are presented. Another powerful programming pattern we can utilise is the organisation of "helper-functions" (lemmas) into libraries. Whenever we are presented a goal which is "obvious", we can extract it out as a `lemma` and invoke it directly (sometimes automatically) in the tactic proof. Such techniques allow us to construct proofs of entailments which are both focused and clean in their presentation (sidelining trivial sub-goals), while still being completely formally verified.

2.2 Automated Well-Formedness Proofs

We demonstrate the power of interactive theorem proving by largely automating the process of proving well-formedness, which arises frequently when proving entailment. Suppose we want to prove, from some `A : type`, that the conclusion term of the extensionality axiom is a closed well-formed term. We demonstrated half of the proof of this lemma here,

```
lemma WF.extensionality {A : type} : WF [] Ω $ ∀' (𝒫 A) $ ∀' (𝒫 A) $ (∀' A $
   (↑0 ∈ ↑2) ⇔ (↑0 ∈ ↑1)) ⟹ (↑1 ≃[𝒫 A] ↑0)
  :=
  begin
    apply WF.all, apply WF.all, apply WF.imp,
      { apply WF.all, apply WF.and,
        { apply WF.imp,
          { apply WF.elem,
            -- refl solves equalities which are definitionally equal
            -- in this case `⊢ [A, 𝒫 A, 𝒫 A].nth 0 = some A`
            apply WF.var, refl,
            apply WF.var, refl },
          { apply WF.elem,
            apply WF.var, refl,
            apply WF.var, refl }
        },
        { apply WF.imp,
          { apply WF.elem,
            apply WF.var, refl,
            apply WF.var, refl },
          { apply WF.elem,
            apply WF.var, refl,
            apply WF.var, refl }
        },
      },
    sorry -- Goal is : `⊢ WF [𝒫 A, 𝒫 A] Ω (↑1 ≃[𝒫 A] ↑0)`
  end
```

This becomes intractable to have to manually construct such a proof for every term of substantial structure. The key observation is that there is a canonical choice of well-formedness constructor to apply at each step of these proofs. This makes it appropriate to use one of Lean's many *tactics*, called `apply_rules`, which takes a list of rules and repeatedly calls `apply` with the appropriate rule until it can no longer make progress.[6]

By tagging all of the constructors with a `WF_rules` identifier, we can reduce the previous proof to the following.

```
lemma WF.extensionality {A : type} : WF [] Ω $ ∀′ (𝒫 A) $ ∀′ (𝒫 A) $ (∀′ A $
    (↑0 ∈ ↑2) ⇔ (↑0 ∈ ↑1)) ⟹ (↑1 ≃[𝒫 A] ↑0)
  :=
  begin
    apply_rules WF_rules, all_goals {refl}
  end
```

This tactic is incredibly useful, and allows us to greatly reduce the size of proofs without sacrificing proof-completeness. A key example is the following inductive proof.

2.3 Entailments Preserve Well-Formedness

We prove that any proven entailment can contain only well-formed terms. Of the 24 inductive cases to be proven, 12 of them are solved automatically by lines 5 and 6.

```
 1  lemma WF.proof_terms {Γ} {p q} : entails Γ p q → WF Γ Ω p ∧ WF Γ Ω q :=
 2    begin
 3      intro ent,
 4      induction ent,
 5      any_goals {split;apply_rules WF_rules;refl},
 6      any_goals {split; simp * at *;apply_rules WF_rules;refl},
 7      case entails.and_left   : _ _ _ _ _ ih {exact ⟨ih.1, WF.and_left ih.2⟩},
 8      case entails.and_right  : _ _ _ _ _ ih {exact ⟨ih.1, WF.and_right ih.2⟩},
 9      case entails.or_left    : _ _ _ _ _ ih {split, any_goals {simp * at *},
10                                                exact WF.or_left ih.1},
11      case entails.or_right   : _ _ _ _ _ ih {split, any_goals {simp * at *},
12                                                exact WF.or_right ih.1},
13      case entails.imp_to_and : _ _ _ _ _ ih {split, apply WF.and,
14                                                exact ih.1, exact WF.imp_left ih.2,
15                                                exact WF.imp_right ih.2},
16      case entails.and_to_imp : _ _ _ _ _ ih {split, exact WF.and_left ih.1,
17                                                apply WF.imp,
18                                                exact WF.and_right ih.1, exact ih.2},
19      case entails.weakening  : _ _ _ _ _ ih {split; apply WF.add_context, tidy},
20      case entails.all_elim   : _ _ _ _ _ ih {exact ⟨WF.lift_once ih.1,
21                                                        WF.all_elim ih.2⟩},
22      case entails.all_intro  : _ _ _ _ _ ih {exact ⟨WF.drop ih.1, WF.all ih.2⟩},
23      case entails.ex_elim    : _ _ _ _ _ ih {exact ⟨WF.lift_once ih.1, WF.ex_elim ih.2⟩},
24      case entails.ex_intro   : _ _ _ _ _ ih {exact ⟨WF.drop ih.1, WF.ex ih.2⟩},
25      case entails.sub        : Γ B b p q wfb ent ih {
26        suffices : ∀ p, WF (B :: Γ) Ω p → WF Γ Ω ([p // b]),
27          from ⟨this p ih.1, this q ih.2⟩,
28        intros _ wfp, exact WF.subst wfb wfp
29      },
```

[6]It also looks in the goal state for proofs for hypotheses which are precisely the goal to be proven.

```
end
```

3 LITT is a category

The Intuitionistic Type Theory we have presented has an associated category, whose construction is analagous to forming the category of sets from ZF-set theory. In the following theorem we delay the definitions of *term-sets*, *graphs* and their *composition* until the **LITT** presentation.

Theorem 3.1. *Given an **ITT**, $\mathcal{L}$, there is an associated category $T(\mathcal{L})$, whose objects are equivalence classes of term-sets and morphisms are equivalence classes of graphs between term-sets. In both cases, equivalence is defined by provable equality of terms, that is, α and α' are equivalent iff. $\vdash \alpha = \alpha'$ (similarly for graphs).*

It is possible to form equivalence classes of the relevant terms in Lean[7], but we will not present this here to simplify the presentation. Instead we work directly with closed terms, and present statements of the lemmas of equivalence.

Definition 3.2. (**LITT**) The following definitions define the relevant *term-sets graphs* and *composition*.

```
/- Closed terms are well-formed in the empty context -/
def closed : type → term → Prop := WF []

/-! ### tset -/

  /- Closed terms of type 𝒫 A.
    An α : tset A is basically a set of A's, i.e. "term-set" -/
def tset (A: type) : Type := {α : term // WF [] (𝒫 A) α} -- subtype
    construction

/-! ### graph -/

/- F is a tset representing the graph of a function from α to β -/
-- Note: `⊆[-]` and `α × β` have derived meanings
def is_graph {A B: type} (α : tset A) (β : tset B) (F : tset (A × B)) : Prop :=
  (⊨ (F ⊆[𝒫 A] (α × β))) -- F is a subset of the product (of terms, not just
    types)
    ∧
  (⊨ (∀ A ((↑0 ∈ α) ⟹ (∃!' B $ ⟨↑1,↑0⟩ ∈ F))) -- F is functional

/- The Type of graphs - defined via subtpe of tset (A × B) satisfying `
    is_graph`-/
def graph {A B} (α : tset A) (β : tset B) : Type
  := {F : tset (A × B) // is_graph α β F}
```

[7] via the quotient construction

```
-- the identity graph
def diagonal {A} (α : tset A) : graph α α :=
  ( graph.mk
    ( tset.mk (A × A) ({ A × A | ∃ A (↑1 ≃[A 𝕏 A] ⟨↑0,↑0⟩)})
      (by apply_rules [WF_rules, WF.closed_add_context];refl)
    )
  )
  (by sorry)



/-! ### composition -/


variables {A B C D : type}
variable {α : tset A}
variable {β : tset B}
variable {η : tset C}
variable {δ : tset D}

/- The underlying term of the composition of two graphs -/
def composition_term (F : graph α β) (G : graph β η) : term :=
  { A × C | -- all d : A × C such that
        ∃[A,C] -- ∃ a c,
        (
          (↑2 ≃[A × C] ⟨↑1,↑0⟩) -- d = ⟨a,c⟩
         ∧
          (∃ B ((⟨↑2,↑0⟩ ∈ F) ∧ (⟨↑0, ↑1⟩ ∈ G))) -- ∃ b, ⟨a,b⟩ ∈ F ∧ ⟨b,c⟩ ∈ G
        )
  }

/- The composition construction produces a closed term -/
def WF.composition (F : graph α β) (G : graph β η) : closed (𝒫 (A × C))
    (composition_term F G)
  := by WF_prover;refl

/- The graph which is the composition of two graphs
Note we define F ∘ G as what would usually be G ∘ F (this is just the Lean
    convention) -/
def composition (F : graph α β) (G : graph β η) : graph α η :=
  (graph.mk (tset.mk (A × C) (composition_term F G) (WF.composition F G)))
  (by sorry) -- Proof of `is_graph`
```

We can now state the relevant theorems which demonstrate that term-sets and graphs form the relevant category $T(\mathcal{L})$. We employ this use of sorry in Lean as a placeholder for missing proofs.

Definition 3.3. (**LITT**)

```
/- (F ∘ G) ∘ H ≃ F ∘ (G ∘ H) -/
theorem associativity (F : graph α β) (G : graph β η) (H : graph η δ) :
⊨ (composition (composition F G) H ≃[𝒫 (A × D)] composition F (composition
G H))
:= sorry

/- F ∘ Δ_β ≃ F -/
theorem comp_id (F : graph α β) : ⊨ composition F (diagonal β) ≃ F
:= sorry

/- Δ_α ∘ F ≃ F -/
theorem id_comp (F : graph α β) : ⊨ composition (diagonal α) F ≃ F
:= sorry
```

4 Conclusion

We have successfully translated Lambek and Scott's **ITT**[2], into a well-defined formal system **LITT** in Lean. In this implementation we formally defined the inductive types, `type, term` of the type theory, as well as the inductive propositions `WF, entails`, which reason about well-formedness and entailments of terms, and simultaenously force the intended semantic interpretation on `types` and `terms`. We have also circumvented the need for α-equivalence by use of De Bruijn indices [4].

Lean proves to be a viable option for a more careful analysis of formal systems, such as **ITT**. Being able to organise "obvious" lemmas as libraries of functions allows us to control the otherwise-intractable scale of formal proofs, while presenting them in a clean focused manner. Throughout we have seen the value in extending the parser to user defined notation - which greatly improves the readability of terms in proofs. Lean's tactic mode allows inherently non-linear proofs to be carried out in an interactive, goal-based procedure, allowing the programmer/mathematician to prove one goal at a time without having to manually keep track of goals.

Now that we have stated the relevant theorems needed to show that this type theory has a natural associated category, the next step is to develop the foundational libraries of "obvious" lemmas needed to prove these theorems - and then prove them.

References

[1] Moura, Leonardo de, Soonho Kong, Jeremy Avigad, Floris van Doorn and Jakob von Raumer. "The Lean Theorem Prover." (2015).

[2] J. Lambek and P.J. Scott, *Introduction to higher-order categorical logic*, (Vol. 7). Cambridge University Press, (1988).

[3] B. Price, *TL*, 2020. Github Repository. `https://github.com/billy-price/TL`

[4] B.C. Pierce, *Types and Programming Languages*, MIT Press, (2002).

[5] D. Murfet, *Topos theory and categorical logic seminar - Lecture 9 - Higher-order logic and topoi (Part 2)*, (2018).

[6] M. Carneiro, *The Type Theory of Lean*, 2019