Formalising the Categorical Semantics of the Simply-typed Lambda Calculus

The main goal of this project is to formalise the syntax and semantics of the simply typed lambda calculus, giving its typing rules and describing the $\beta\eta$-equality of terms as inductively defined relations based on syntax. The typing relation is then interpreted as morphisms in a cartesian closed category (CCC), and the main theorem of this project - the soundness theorem - states that if two terms of the same type are deemed equal then they are interpreted as the same morphism. This theorem is one direction of a correspondence between the syntactic $\beta\eta$-equality and semantic equality. This correspondence is essentially because the equality rules describe the universal properties of the corresponding categorical constructions (e.g. the β-rule for product types corresponds to the commutativity of the product diagram, while the η-rule describes the uniqueness of the product morphism.)

The mathematical developments in this project are derived from a lecture note on CCCs and the lambda calculus from the Cambridge Computer Science Tripos Part III. However, I diverged in the representation of the lambda calculus, where I use a locally nameless representation using De Bruijn indices based on this paper by Arthur Chargueraud. This representation is much better suited for formal reasoning than the usual named approach, which requires us to reason about equivalence classes of terms up to renaming. This makes it much harder to perform induction and recursion as the equivalence class obfuscates the inductive structure. I explain the representation in the following section.

1 Locally Nameless Representation

I started by defining (in the `src/syntax.lean` file) an inductive datatype `type` to represent the syntax of simple types. There are the unit type, pair types and function types, mirroring the terminal, binary product and exponential objects of a CCC, as well as ground types which are meant to be type constants, semantically denoting some particular object in the category.

$$A, B \quad ::= \quad |G| \mid \text{unit} \mid A \,\Pi\, B \mid A \supset B \mid$$

Next, the syntax of terms are defined. In the locally nameless representation, we distinguish between free and bound variables. For the bound variables, instead of using variable names, they are instead represented as a natural number representing the distance from the variable to the Λ abstraction that binds it, in terms of the number of abstractions in between. As a result, the abstraction no longer needs to explicitly give a name. For example, the lambda term usually represented by $\lambda x.x(\lambda y.xy)$ is represented as $\lambda\lceil 0\rceil(\lambda\lceil 1\rceil\lceil 0\rceil)$. Free variables are not bound, so they get to keep their names. Hence, in the below grammar, x ranges over a set of names while n over the natural numbers. Additionally, the lambda abstraction term is annotated by the type of the variable it binds, which allows us later to prove that each term has a unique type.

In addition to the term constructors corresponding to the categorical constructions, we also have constants, which is attributed with some fixed type. Here, c ranges over a set of constants.

$$t, t_1, t_2 \quad ::= \quad \lfloor x\rfloor \mid \lceil n\rceil \mid |\mathrm{c}| \mid \langle\!\langle\rangle\!\rangle \mid \langle\!\langle t_1, t_2\rangle\!\rangle \mid \mathsf{fst}\ t_1 \mid \mathsf{snd}\ t_2 \mid \Lambda A.t \mid t_1 \cdot t_2$$

Now that we distinguish free variables and bound variables, not all terms formed by this grammar are well-formed. Specifically, a term is well-formed only if each bound variable refers to an existing lambda abstraction in the term. Such a term is called *locally closed*.

We define the `locally_closed` relation (or `lc` for short) inductively, which also checks that the free variables are in a given set Γ. This is because when we unpack a lambda abstraction, we have to take into account that any bound variable previously bound by the abstraction now occurs free, so the body is not guaranteed to be locally closed. Instead, we have to convert any bound variables that refer to this lambda into some free variable, and check the converted body term instead. This process is called *opening*. The question now is which free variables should we do this for? Immediately, the following two options come to mind:

$$\frac{\forall x \notin fv(t) \cup \Gamma, \mathrm{lc}\ \Gamma\ open(x, t)}{\mathrm{lc}\ \Gamma(\Lambda A.t)} \qquad\qquad \frac{\exists x \notin fv(t) \cup \Gamma, \mathrm{lc}\ \Gamma\ open(x, t)}{\mathrm{lc}\ \Gamma(\Lambda A.t)}$$

Figure 1: Potential rules for local closure of lambda abstraction

However, both options are suboptimal. The first version with the universal quantifier is really easy to use, i.e. if you have already a proof of lc $\Gamma(\Lambda A.t)$, then you can deconstruct it and obtain the local closure of the body opened with any variable. Unfortunately, it is difficult to prove the universal assertion as it requires you to reason about an arbitrary variable. The second version is precisely the opposite. Instead, we need to take the middle ground where we assert that x is not in some finite set L of free variables:

$$\frac{\exists L, \forall x \notin L, \text{lc}\ \Gamma\ open(x, t)}{\text{lc}\ \Gamma(\Lambda A.t)}$$

Figure 2: Better rule for local closure of lambda abstraction

This third version of the rule based on cofinite quantification is easy to use just like the first version, as it may still be applied to infinitely many free variables. However it is not as difficult to prove, as one has a wider choice of which set L to use. Unfortunately, it is not possible to encode this rule in Lean 3, as it does not have native support for nested inductive types (here, the existential quantifier is the inductive type σ).

2 The Typing Relation & its Semantics

The typing relation is of the form $\Gamma \Vdash t :: A$, where Γ is a list assigning free variables to their type, t is a term and A is a type. It is defined inductively with one inference rule corresponding to each syntactic constructor. The exception to this is that bound variables have no corresponding rule (i.e. they can never be assigned a type, which makes sense as a single bound variable alone is not locally closed), and that free variables have two rules. The reason for the latter becomes apparent when we discuss the semantics. As with the definition

Unlike for classical logic, where the inference relation $\vdash$ is meta-theoretic and we only semantically interpret formulas, here we interpret the typing relation itself (or more precisely, the derivation/proof of the relation). We interpret into an arbitrary cartesian closed category, which is a category with a terminal object, binary products, and exponentials. Before we can interpret the typing relation however, we must interpret the types and context Γ.

Types are interpreted as objects in the CCC, in a predictable manner: the product type $A \,\Pi\, B$ is interpreted to the categorical product between A and B's interpretation in a recursive manner, the function type is interpreted as the exponential, and the unit type as the terminal object. For ground types, we must provide a ground interpretation function G which maps each ground type to some object in the category. The interpretation of a type A is therefore denoted $G[\![A]\!]$. This is called `eval_type` in the code.

The context is also interpreted recursively. The empty list [] is interpreted as the terminal object, while $\Gamma, x :: A$ is interpreted as the product between the interpretation of Γ and A (i.e. the product takes the role of cons). Similarly, this is denoted $G[\![\Gamma]\!]$. In the code, this is called `eval_env`.

With this in mind, we interpret a derivation of the typing relation $\Gamma \Vdash t :: A$ as a morphism from $G[\![\Gamma]\!]$ to $G[\![A]\!]$. However, this time, we need both a ground interpretation function G to interpret ground types, and a constant interpretation function C to interpret constant terms to morphisms from the terminal object to the interpretation of its type. This is encapsulated as a structure, which we call a model $M = \langle G, C \rangle$. Hence if h has type $\Gamma \Vdash t :: A$, then $M[\![h]\!]$ has type $M.G[\![\Gamma]\!] \longrightarrow M.G[\![A]\!]$. This interpretation is called `eval_has_type` in the code.

In defining this interpretation, I encountered some difficulties with Lean's equation compiler which caused issues down the line. The type of `eval_has_type` depends on `eval_type` and `eval_env`. This caused a bug where not unfolding these two functions before providing a term would cause it to not type-check properly. The solution was to avoid the use of the equation compiler by using the recursors of `type` and `env` respectively to define `eval_type` and `eval_env`. Without this solution, it caused issues down the line when I was trying to prove the soundness theorem, because having to unfold these functions in tactic model made the underlying definition more clunky and complicated.

Additionally, I learned that using mathlib definitions from `category_theory.limits.has_limits` is going make the definition noncomputable as many definitions there depend on invoking the axiom of choice to obtain a concrete limiting construction from a proof of existence.

Because the proof of the typing relation is to be interpreted, we cannot make it a `Prop`, but must be a `Type`. This is because `Prop` is proof-irrelevant, so the structure of the proofs are ignored.

3 Proving Various Lemmas by Induction

Almost all the lemmas and theorems in this project proceed by induction. I will not discuss each individual lemma/theorem here but I will discuss the things I learned from attempting their proofs and may occasionally allude to parts of a specific lemma's proof.

Some of these proofs require a `cases` tactic to be applied in each inductive case right after the `induction`, in order to decompose the structure of hypotheses. I learned to use the semicolon instead of a comma, which applies the next tactic to all goals in scope. One issue with doing this is that doing this nested structural decomposition makes the Lean tactic state lose track of the case names. Wrapping the `induction ...; cases ...` code with `with_cases` allows the case names to be recovered. I also learned to use `pretty_cases` which outputs a template for an inductive proof, saving me lots of time.

In many of my proofs, I induce on the structure of relations such as the $\_ \Vdash \_ :: \_$ relation. The basic `induction` tactic is ill-suited for such tasks. For example, in the proof of `type_unicity` and `deriv_unicity`, the induction tactic could not figure out that the same Γ appears in both the hypothesis that the term is locally closed and in the typing relation I was inducing on, so they were assigned different Γs in the inductive cases. In one of the inductive cases, this missing connection turned out to be crucial. Using the `induction'` tactic, I was able to restore the fact that they have the same Γ, allowing the theorems to be proven. As a bonus, the `induction'` tactic generates better hypotheses names than the basic one.

4 The main Proof

Before we can define the soundness theorem, we need a notion of equality between two terms. This is given as the $\beta\eta$-equality relation, also defined inductively. The soundness theorem simply states then that if two terms can be typed under the same context and type, and are $\beta\eta$-equal to each other, then their typing derivations interpret to same morphism.

The proof proceeds by induction, carefully observing that the *beta* rules essentially express the commutativity of their corresponding categorical construction's diagram, while the *eta* rules express the uniqueness of the construction's induced morphism. Together, they express the universal properties of the categorical constructions - so in the proof, we aim to prove the goal by invoking the universal properties.

In some parts of the proof, I run into goals which I couldn't prove because the hypotheses are too rigid. However, having the cofinite quantification mentioned in section 1 would have provided the necessary flexibility to make the proof much easier. Specifically, this seems to arise in the case for the `cong_lam` rule.