A Review of Algebraic-Style Reasoning for Type Theory

ENDE JIN, University of Waterloo, Canada

We take a technical review of the algebraic formulation of meta-theoretical reasoning of type theory. We focus on the application side and emphasize on the motivation. We will mainly aim at proving *canonicity* and then mention its relationship with *computational adequacy* and *observational equivalence*, and finally talk about *Normalization by Evaluation*. After that we will review MLTT and its canonicity proof. We will stick with the slogan "*type theory should eat itself* (Chapman 2009)" by formulating most of our reasoning inside the *Quotient Inductive-Inductive Type* (QIIT) framework. Meanwhile, we want our reasoning to be compositional and systematic if possible.

1 MOTIVATION

Algebraic Theory can be roughly considered as a theory using equations to indicate semantics. In our context, these are equations between syntactic terms of a programming language, contrary to how we usually formulate operational semantic as a "unidirectional" relation between terms and might involve non-syntactic term [1]. We take a technical review of the algebraic formulation of meta-theoretical reasoning of type theory. We focus on the application side and emphasize on the motivation instead of the rich mathematical structure. We mainly aims at *canonicity* of our programming language.

The very first question would be – why take this alternative approach? It is well known that *logical equivalence* based on *operational semantic* using logical relation [8, 36, 38] is good enough to handle a wide range of language features for contextual equivalence. What's more, to verify our proposed equational theory to be a **programming language**, we still at the end need to stick with operational semantic/a runnable interpreter.

The reason we choose the algebraic approach is mainly because (A). The equational theory itself acts as a specification for both *program dynamism* and *observational equivalence*. (B). Equational theory is closely related to the universal algebra and categorical logic and thus it is possible for us to resort to category theory. This might bring a compositional and systematic perspective for meta-theory reasoning. (C). This style is inherently intrinsically "well-typed" and resulting a "type theory inside type theory" paradigm thus more friendly to the type theorists and mechanized proving.

The main focus of this article will be familiarizing the technical details, and thus there won't be many fancy type theories involved.

The main motivation of this article is to let type-theory-oriented proof-engineerers realize that, with the language of Quotient Inductive Inductive Type, some metatheorem for MLTT and dependent type is perceivable without the loss of mathematical rigour; and this language is possible to be used in other places as well. There are places involving categorical nonsense in this article but we still hope people without categorical prerequisite can understand the fake Agda code and grasp the main idea.

The whole article is formulated in the following way. Section 2 will cover how the most basic example, STLC , including the syntax and dynamism, can be formulated in an equational style. The formalization is carried out in QIIT framework. We also introduce the formalized meta-theorem that we aim at, mainly surrounding canonicity, and its relation with computational adequacy and observational equality. In Section 3, we will review the Normalization by Evaluation (NbE) technique in our QIIT framework. In Section 4, we will review the canonicity proof for Martin Lof

[1] for example, when modeling heaps, a non-syntactic state-mapping (indicating the heap/store) will be carried around

Author's address: Ende Jin, ende.jin@uwaterloo.ca, University of Waterloo, Canada.

Type Theory. In Section 5, we will review *categorical gluing* as a generalization (and analysis) of logical relation, and reflect on our given formalization.

Each section might have some prerequisites and we will indicate them.

2 STLC USING EXPLICIT SUBSTITUTION

In this section we will use Simply Typed λ Calculus (STLC) as an example to introduce some background knowledge.

Here we will introduce the lambda calculus with explicit substitution (and debruijn indices) and adopt the equational logic from [31], a style also called *substitution calculus*. Because (1). This style can easily scale to dependent type [42]. (2). It has clear categorical semantic. (3). When written in explicit substitution in *intrinsic type system*[2], the substitution lemma is automatically requested from the beginning, as part of the equational rule [3] (4). This style can then be easily adopted in mechanized proving. [26, 46]

Note 2.1. Explicit substitution can be considered as "substitution at object language level". Contrary to how usually when we define (operational) semantic we will define a substitution operation at meta-level, explicit substitution is part of the dynamic of the STLC we are formalizing. Explicit substitution is considered orthogonal with using debruijn indices.

Another highlighted point is that "same terms" in different context cannot be considered to be "equal". Trivial as it is, when **not** using intrinsic style and consider terms **not** attached with its forming context, this detail will sometimes be ignored (even in a mechanized context).

What's more, we will use Quotient Inductive Type(QIT)[4] to formalize our language: STLC syntax, and the semantic, in a type-theoretic flavour[5]. In this way, we only need to refer to [10] for the foundation of the induction principle. We will introduce and prove canonicity theorem after that.

2.1 Formalizing STLC using QIT

From this section forward, we require some familiarity with proof assistants close to Agda. Suggesting [28]

We will use (fake) Agda style[6] to introduce our STLC syntax. The general idea here is to define the syntax of STLC using *Quotient Inductive Type* (QIT). Quotient is a standard mathematical concept where we "consider" two elements are the "same". For example, we take quotient on integers $\mathbb{Z}$ by considering even number as one and odd numbers as another, denoted as $\mathbb{Z}/2\mathbb{Z} = \{\{n \text{ is even}\}, \{n \text{ is odd}\}\}$. Then after quotient, there are only two elements, and by convention we use *equivalence classes* as the elements of the result.

QIT is applying quotient to inductive type[7] – in QIT we can impose ***identity constructor*** between different elements to *consider them same*.

Example 2.2 (parity for Natural). Instead of taking modulo on integers, we use natural number as a simpler illustration.

[2] *Intrinsic Type System* is a type system where only well-typed pieces of code can be constructed. *Extrinsic Type System* allows presenting untyped code and we use typing rule on untyped code afterwards.

[3] This gives a systematic way to derive substitution lemma when using intrinsic style

[4] We will later use Quotient Inductive Inductive Type(QIIT) when formulating MLTT. I will later explain what is an Inductive Inductive Type

[5] Also as one step towards formalizing in type-theoretic proof assistants

[6] Because our syntax and semantic is very close to Agda, but we don't explicitly write out all the implicit parameters, and we will ignore most of the universe issue in Agda when dealing with STLC

[7] Just like QIIT is applying quotient to inductive-inductive type [34]

```
data N : Set where
  z : N
  s : N → N
  parity : {x : N} → x ≡ s s x
```

We should be able to prove that this QIT only has two elements inside.

Now we can define our syntax using QIT, the fullly commented (fake) Agda version is here, the space is not enough in the paper to include all the detail explanations and comments.

```
data Ty: Set where
  ⇒ : Ty → Ty → Ty       -- function type
  B : Ty                 -- boolean type
data Con: Set where
  ·   : Con              -- empty context
  _▹_ : Con → Ty → Con   -- context extension
data Tms : Con → Con → Set where
data Tm : Con → Ty → Set where
  _∘_ : Tms θ Δ → Tms Γ θ → Tms Γ Δ      -- subst composition
  id  : Tms Γ Γ                          -- identity subst
  ε   : Tms Γ ·
  _,_ : (σ : Tms Γ Δ) → Tm Γ A → Tms Γ (Δ ▹ A)
  π₁ : Tms Γ (Δ, A) → Tms Γ Δ            -- subst projection
  _[_] : Tm θ A → (σ : Tms Γ θ) → Tm Γ A
  π₂ : Tms Γ (Δ, A) → Tm Γ A
  -- we will sometime omit the first argument of π₁, π₂

  -- following are all equational rules
  [][] : t[σ][ν] ≡ t[σ ∘ ν]
  ∘∘ : (σ ∘ ν) ∘ δ ≡ σ ∘ (ν ∘ δ)
  id∘ : id ∘ σ ≡ σ
  ∘id : σ ∘ id ≡ σ
  εη : {σ : Tms Γ ·} → σ ≡ ε
  π₁β : π₁ (σ, t) ≡ σ
  πη : (π₁ σ, π₂ σ) ≡ σ
  π₂β : π2 (σ, t) ≡ t (by π₁β)


-- we define a debruijn shifting function
_↑_ : Tms Γ Δ → Ty → Tms (Γ, A) (Δ, A)
σ ↑ A = (σ ∘ π₁, π₂)

data Tm where
  lam : Tm (Γ, A) B → Tm Γ (⇒ A B)
  app : Tm Γ (⇒ A B) → Tm (Γ, A) B
  ⇒β : app (lam t) ≡ t                   -- beta rule
  ⇒η : lam (app t) ≡ t                   -- eta rule
  lam[] : (lam t)[σ] ≡ lam (t[σ ↑ A])    -- subst law

  tt : Tm Γ B
  tt[] : tt[σ] ≡ tt
  ff : Tm Γ B
  ff[] : ff[σ] ≡ ff
  ifb : Tm Γ B → Tm Γ A → Tm Γ A → Tm Γ A
```

```
ifb[] : (ifb c a b)[σ] ≡ ifb c[σ] a[σ] b[σ]
ifbβ₁ : ifb tt a b ≡ a
ifbβ₁ : ifb ff a b ≡ b
```

Note 2.3 (How to read the inference rules?). In this intrinsic setting, every data type in QIT(QIIT) level is actually a judgement. Here we have four data type, indicating four kinds of judgements for well-formed context, type, term and substitutions. QIIT brings the ability to express new judgement in our meta-theory.

If we have `Γ : Con, T : Ty`, then they mean `Γ` is a well-formed context and `T` is a well-formed type. If we have `t : Tm Γ T`, then we have $\Gamma \vdash t : T$. The unconventional judgement – well-typed substitution `σ : Tms Γ θ` is usually informally represented as $\Gamma \vdash \sigma : \Delta$, because an explicit substitution can be considered as a "tuple" of terms.

When formulated using inductive type, the inference rule is expressed by implication[8]. For easier understanding, we spell out the informal presentation here, for `ifb`, `_∘_` and `_[_]`.

$$\frac{\Gamma \vdash c : \mathbb{B} \quad \Gamma \vdash a : T \quad \Gamma \vdash b : T}{\Gamma \vdash ifb\ c\ a\ b\ : T}\text{[ifb-rule]} \qquad \frac{\Gamma, X \vdash t : Y}{\Gamma \vdash (lam\ t) : X \Rightarrow Y}\text{[Abstraction]}$$

$$\frac{\Gamma \vdash \sigma : \Delta \quad \Delta \vdash \tau : \Theta}{\Gamma \vdash \sigma \circ \tau : \Theta}\text{[}\circ\text{-rule]} \qquad \frac{\Gamma \vdash t : (X \Rightarrow Y)}{\Gamma, X \vdash (app\ t) : Y}\text{[Application]}$$

$$\frac{\Gamma \vdash t : T \quad \Delta \vdash \sigma : \Gamma}{\Delta \vdash t[\sigma] : T}\text{[Subst-rule]}$$

Each data type in QIT can have equality information, thus we can express judgemental equal for each type judgement. Equality is usually used between terms and substitutions. For example, the rule `[][]` is postulating that consecutive substitution is equivalent to substitution by the composition. The rule `∘∘`is also necessary to consider the whole substitution formulation as a category. The rule `⇒β` encode the beta-reduction in the conventional operational semantic.

Finally, function introduction and elimination rule is a bit non-conventional in our formulation. We also spelled out their conventional formulation here. *app* rule might be confusing in the first sight – where is the argument that should be doing application? The point is that, if we want to really *use* an argument, we are actually doing *explicit substitution* instead :
`(app u)[(id, v)] : Tm · Y` for `u : Tm · (X ⇒ Y), v : Tm · X`.

A formalized but more conventional formulation of our syntax and inference rules can be found in [42].

Note 2.4. QIIT is not directly supported in Agda yet. The quickest way to get QIIT is to postulate the constructors for QIIT, which is safe since the postulated propositions are consistent (The model is exactly QIIT itself). Then, using Agda's Rewriting facility, we still have a logically sound proof assistant[20] and we retrieve strictness[9] and (automatic-Agda-recognized) judgemental equality directly.

QIT can also be implemented by Higher Inductive Type (HIT) using Cubical Agda[45] and impose strictness using `isSet` . Because HIT in Agda is properly provided, thus using HIT to encode QIT, we can get the *recursor* and *dependent eliminator* for free, which is not the case for the former approach. However we think the former approach is more accessible and easy to use for the interested audience.

It is unintuitive to see why QIIT has computational content (i.e. runnable), but it does[13].

[8]This intrinsic style can also be found [28]

[9]Strictness here refers to UIP: there is a unique proof of a identity proposition. Seemingly surprising, dependent type system adopted by Coq and Agda cannot prove it, which opens the possibility of HoTT.[37]

Note 2.5 (Logical Framework). A logical framework[5] is a kind of type theory that provides a means **to define (or present) a logic as a signature in a higher-order type theory** in such a way that provability of a formula in the original logic reduces to a type inhabitation problem in the framework type theory.

Thus our QIIT + (fake) Agda can be seen as a logical framework, to formalize the syntax(signature) of STLC . We later will show how to formulate Martin Lof Type Theory(MLTT) in this framework.

2.2 Formalized Semantic for STLC using Algebra for QIT

Since we have formalized the syntax as data in QIT form, we now can specify the semantic. The semantic can be indicated by using *algebra*(or model) of QIT [10], which can be informally understood as the *recursor* for the QIT. Instead of specifying the formal semantic of the *algebra* of a QIT, we write down the algebra in our QIT example, which is much more accessible.

```
module Standard where
  Conₘ       = Set
  Tyₘ        = Set
  -- Note Tmsₘ : Conₘ → Conₘ → Set
  Tmsₘ Γ Δ   = Γ → Δ
  Tmsₘ Γ A   = Γ → A

  ·ₘ : Tyₘ   = ⊤       -- Agda's unit type
  A ▹ₘ B     = A × B   -- Agda's product type
  t₁ ∘ₘ t₂   = t₁ ∘ t₂ -- using the composition on Agda functions
  idₘ        = λ x → x
  εₘ         = λ _ → ⊤
  (t₁ ,ₘ t₂) = λ γ → (t₁ γ , t₂ γ)
  π₁ₘ t      = λ γ → (t γ).fst
  t[σ]ₘ      = t ∘ σ
  π₂ₘ t      = λ γ → (t γ).snd
  [][]ₘ : t[σ]ₘ[ν]ₘ ≡ t[σ ∘ₘ ν]ₘ
  []ₘ        = refl
  -- Other Natruality Condition (interpretation for equality)
  --    are omitted
  -- because all of them will be proved by refl
  ⇒ₘ : Ty → Ty → Ty
  ⇒ₘ A B = A → B   -- Agda's function type
  lamₘ t = λ γ  → λ a → t (γ , a)
  appₘ t = λ γₐ → t (γₐ.fst) (γₐ.snd)

  B : Ty  -- boolean type
  Bₘ     = Agda.Bool
  ttₘ    = λ _ → true
  ffₘ    = λ _ → false
  ifbₘ c a b = λ γ → if (c γ) (a γ) (b γ)
```

An algebra for a QIT written here as an Agda module is indicating the interpretation for each syntax piece, e.g. $\texttt{Con}_m$ for interpreting `Con`, and the interpretation of `Con` will affect the interpretation of other syntax like $\cdot, \triangleright$ because their type signature will change accordingly. The equality inside QIT needs to be interpreted as the equality between the corresponding interpretation, and we have to prove our model[10] respect this equality.[11].

[10] We use *model* and *algebra* interchangeably

[11] In real Agda, we only need to fill in refl and then Agda will normalize both sides to see they are equal

First, note that what we do here is using Agda's facility to model the Cartesian Closed Category (CCC) + Boolean structure, for example, both substitution and terms are interpreted as function space in Agda, just like they are interpreted as hom-sets in CCC.

Second, notice that this is actually specifying a recursive function[12] mapping out of QIT. Concretely, this function will be defined mutual recursively:

```
recCon : Con → Conₘ
recTy  : Ty  → Tyₘ
recTms : {Γ Δ} → Tms Γ Δ → Tmsₘ (recCon Γ) (recCon Δ)
recTm  : {Γ Δ} → Tm  Γ A → Tmsₘ (recCon Γ) (recTy A)
-- We only write out some ...
recCon · = ·ₘ
recCon (A ▹ B)   = (recCon A) ▹ₘ (recTy B)
recTms (t₁ ∘ t₂) = (recTms t₁) ∘ₘ (recTms t₂)
-- ... and you should be able to know what the rest looks like
```

Actually there are four functions, and each one acts mutual-recursively with others. If we set the model (interpretation) to be generic, then these four functions can be called *recursors*. This explains what we mean "when we informally define semantic by Induction on the syntactic derivation". Now with the above model, we can prove the "relative-consistency" theorem – i.e. judgemental equality between true and false is underivable and thus our system is "relative-consistent" to our meta-logic (our fake Agda).

```
consistent : tt ≡ ff → ⊥
consistent p with (g p)
... | ()
  where
    g : tt ≡ ff → true ≡ false
    g = ... -- using recTm to map tt, ff to Agda's true and false
```

Note 2.6. Unlike vanilla inductive type, QIIT allows equality between terms of different forms and thus either equality or inequality between two arbitrary QIIT term is not easily decidable[13].

Thus our proof maps QIT (*Syntax of STLC*) into a vanilla inductive type (*Agda's primitive boolean type*). For proof assistant, equality and inequality between terms of vanilla inductive type is easily decidable, and thus the proof assistant can accept our proof. However, this requires the *recursor* `recTm` to be supported.

2.3 Interlude: Canonicity, Computational Adeqaucy and Observational Equivalence

Canonicity means that *every closed term of type boolean is judgementally equal to either true or false*. Of course, canonicity can be stronger than that, especially when there are more base types and "observable" type. For example, when we add natural number into our system, we (should) also require *every closed term of natural number is judgementally equal to either zero or some direct (multiple) successor of zero*. Another example is that [23] requires similar results for closed term of identity type and modalities. In our setting, canonicity is easilly expressible

```
canonicity : (t : Tm · B) → (t ≡ tt) + (t ≡ ff)
    -- we use + to indicate disjunction
```

Unlike *canonicity* that can be stated in a *reduction-free* system, *computational adequacy* and *observational equivalence* can only be stated when there is operational semantic. Meanwhile, like

[12] A better wording is *morphism* because this function will respect the *structure* of the type, i.e. the equality(quotient).
[13] Reference Required

canonicity, they (can) only talk about one fixed "observable" type, which we set to be the only base type – boolean type.

[43] defines that a denotational model is **Computationally Adequate** if a closed term of boolean type (operational semantically) evaluates to true iff it denotes true. Use conventional notation, $\forall M : (Tm \cdot \mathbb{B}), [\![M]\!] = true \iff M \Downarrow tt$

[24] states that two terms are **Observationally Equivalent** if, these two terms plugin in arbitrary context will evaluate to the same boolean value. Here we want to show that our judgemental equality coincides with observational equivalence.

But if we try to state these two properties formally in our (fake) Agda, we will encounter some problems:

```
eval : Tm · T → Tm · T
_≈_  (a b : RTm Γ T) : Set
a ≈ b = (ctx : (RTm Γ T ↝ RTm · B))
        → eval (plug ctx a) ≡ eval (plug ctx b) "literally"
```

We can see here that, observational equivalence requires two terms to be "*literally equal*" than just "*judgemental equal*". However, we cannot express this *literal equality* once we quotient our terms with judgemental equality. *We are manipulating equivalence class instead of concrete representative.* This perspective makes `eval` actually an identity function and not sensible at all. Thus it fails to be a legitimate formalization and we will later come back to fix this issue.

Note 2.7. Our equation-oriented system is said to be *reduction-free* because we don't have operational semantic as our basis.

On the other hand, there are programming languages solely based on operational semantic(thus with a notion of reduction) in an untyped setting. Then the type system is considered to be a proof system that derives unstuck expressions, thus called *Type-Assignment System* [6] . Concretely, define a proposition $\Gamma \models v : T$ to be "$\forall v', v \rightsquigarrow v' \Rightarrow \text{unstuck}(v') \vee \text{value}(v')$". Then a theorem "$\Gamma, x : A \models t : B \Rightarrow \Gamma \models \lambda x : A.t : A \to B$" is exactly the inference rule λ-intro, tracking the meaning of *type-safety* inside the type judgement. This interaction between operational semantic and type-assignment system is just like the relation between models and proof system in logic.

As you can see this style is highly extrinsic because every (smaller) piece of the syntax has its own meaning compare to our QIIT-formulated style, where each term is dependent on its defining context and cannot be decoupled.

2.4 Logical Relation for Canonicity using Dependent Algebra for QIT

This section requires some familiarity of logical relation. Suggesting [9, 38].

Similar as we use *dependent elimination* to prove proposition about inductive type, our aim here is to the present the proof of logical relation using dependent elimination for QIIT. Just like we use algebra to define recursive function, we can define dependent elimination using *dependent algebra* [10], but we will not describe the formal definition of dependent algebra.

Note 2.8. Here we recall how dependent eliminator can be *partially* derived from the recursor for vanilla Inductive Type, which might be helpful for understanding the relationship between dependent algebra and algebra in QIIT.

Consider for natural number, we will have a recursor that act like a pattern-matching $rec : R \to (R \to R) \to \mathbb{N} \to R$, with two computation rules : $rec\ r_0\ f\ (n+1) = f(rec\ r_0\ n)$ and $rec\ r_0\ f\ 0 = r_0$. Apparently, the r_0 and f are two clauses of the pattern matching.

But in proof assistants, we most likely need to reason, and we need induction, thus we will usually have following scheme: $ind : P(0) \to (\forall n : \mathbb{N}.P(n) \to P(n+1)) \to \forall n : \mathbb{N}.P(n)$[14] . To use that, for example, if we want to prove $\forall n, n \equiv n + 0$, we just simply set $P(x) = x \equiv x + 0$, and then the two arguments for ind is exactly base case and inductive step.

By first sight, ind seems much stronger than rec, but actually we can use rec to encode ind to some extent, thanks to the sigma type: we simply set $R = \sum n.P(n)$, then we will have $rec\sum : \sum n.P(n) \to (\sum n.P(n) \to \sum n.P(n)) \to \mathbb{N}. \to \sum n.P(n)$. Here we wrap this predicate using sigma type, and thus we decouple the dependency. Our recursor is not only calculating, but also ***passing a proof*** of $P(n)$ aside. Of course it is unsatisfactory, because we want $rec \sum\ r_0\ f\ 0$ to return $P(0)$, but now it is returning $\sum n.P(n)$. Thus we need to require that the first element of the sigma type "*aligns*" in a post-hoc manner.

But anyway, now we have an idea on how to represent proof using algebra in our QIIT, which fortunately, suffices to prove logical relation. But still there is a problem : given `t : Tm · B` we have $[\![\texttt{t : Tm · B}]\!] = (\texttt{t}, ??) : \sum \texttt{t: Tm · B}.P(\texttt{t})$. We know we want this P to indicate `t` judgementally equal to either true or false, and our denotation will act as identity the first element but the second element ?? will need to be a proof of P. But what exactly this P should look like type theoretically?

We try to get inspiration from the classical logical relation argument [9]. Recall in that setting, we will have $\cdot \models t : \texttt{B}$ defined as $\forall \gamma \in [\![\cdot]\!].t[\gamma] \in [\![\texttt{B}]\!]$. If we define $[\![\texttt{B}]\!] = \{tt, ff\}$, then we are done because we are working on the $\beta\eta$-quotiented terms. Thus in our model $[\![\texttt{B}]\!]$ should be this exact subset of `Tm · B`. In dependent type setting, usually we use a predicate to indicate a subset, and thus we will have $[\![\texttt{B}]\!] : \texttt{Tm · B} \to \texttt{Set}$,in which case, $x \in [\![\texttt{B}]\!]$ is encoded by $[\![\texttt{B}]\!](x)$ is inhabited(provable). This predicate style coincides exactly with the earlier P! Thus this P will indicate a subset of the closed terms, and $P(t)$ indicate t belongs to that subset. This algebra makes sense in Curry-Howard-Correspondence, because now we are passing proof as data all around and tracking the elements *satisfy our purpose.*

However, things become tricker because we are in a well-typed setting, and everything in our algebra needs to attach a proof with it. With the above intuition, we will now introduce the canonicity proof as another algebra. Note the intuition of the notation we use : for a syntactical piece `t`, we will have $\texttt{t}_\texttt{p}$ to indicate the proof we wish to attach, and $\texttt{t}_\texttt{2}$ is the data together with the proof guard it. We omitted a lot of details here, and the full version can be found here

2.5 Canonicity Model, Concretely

```
Conₚ : Con → Set -- we want context to be proof-guarded
Conₚ Γ = Tms · Γ → Set

Tyₚ : Ty → Set -- we also want type to be proof-guarded
Tyₚ T = Tm · T → Set
```

These two predicates over `Con, Ty` stands for subsets of `Tms · Γ` and `Tm · Γ`

Now we use a shorthand $\texttt{Con}_\texttt{2}$ to indicate $\sum$ `Γ : Con,` $\texttt{Con}_\texttt{p}$ `Γ`, and similarly, $\texttt{Ty}_\texttt{2}$ to indicate $\sum$ `T: Ty,` $\texttt{Ty}_\texttt{p}$ `T`. Then it comes to the tricky part, how do we define motives for `Tms Γ Δ`?

```
Tmsₚ : (Γ₂ : Con₂) → (Δ₂ : Con₂) → Tms Γ Δ → Set
Tmsₚ Γ₂ Δ₂ σ =
  (γ : Tms · Γ) → (γₚ : Γₚ γ) → Δₚ (σ ∘ γ)

Tmₚ : (Γ₂ : Con₂) → (T₂ : Ty₂) → Tm Γ T → Set
```

[14] Actually with computational rule as well! We don't mention that because we sometimes want *proof-irrelevance.*

```
Tmₚ Γ₂ T₂ t =
  (γ : Tms · Γ) → (γₚ : Γₚ γ) → Tₚ (t[γ])
```

A predicate guarding substitution claims that composed with a "canonical" substitution will lead to a "canonical" substitution. We have a similar claim about the terms. Note that we allow to take out Γ_p and Γ from Γ_2 – because they are just short hands.

Note 2.9. Now you can notice the importance of the shorthand. Without `Con₂, Ty₂` as short hand, we will have to write down this horrible lengthy definition:

```
Tmsₚ : (Γ : Con) → Γₚ : Conₚ Γ → (Δ : Con) → Δₚ : Conₚ Δ
       → Tms Γ Δ → Set
Tmsₚ Γ Γₚ Δ Δₚ σ =
    (γ : Tms · Γ) → (γₚ : Γₚ γ) → Δₚ (σ ∘ γ)
```

Note that there is an obvious pattern here: every guarding predicate $(\cdot)_p$ will have all its parameters in the form of $(\cdot)_2$ and then with an extra parameter about the aiming "syntactic" piece. The main reason we construct predicate like this is because we want to have a model for a bunch of $(\cdot)_2$, for example, for a syntactic derivation `π₁ : Tms Γ (Δ, A) → Tms Γ Δ` we want a corresponding lemma `π₁₂ : Tms₂ Γ₂ (Δ₂ ▹₂ A₂) → Tms₂ Γ₂ Δ₂` in the model , thus it requires a guarding predicate `π₁ₚ : (σ₂ : Tms₂ Γ₂ (Δ₂ ▹₂ A₂)) → Tmsₚ Γ₂ Δ₂ (π₁ σ)`. This is exactly compatibility lemma in the conventional formulation.

Note 2.10. This actually corresponds to the Sterling's Gluing Notation [39] where each syntax piece `T` is `T◐` , the guarding predicate `Tₚ` is `T◑`, the final `T₂` is `T●` and actually just a sigma type, and also called as the "glued element".

Now we continue to specify some details about the constructions:

```
-- ·  : Con
·ₚ : Conₚ ·
  ≡ Tms · · → Set
·ₚ = λ_ → ⊤

-- _▹_ : Con → Ty → Con
_▹ₚ_ : (Γ₂ : Con₂) → (T₂ : Ty₂) → Conₚ (Γ ▹ T)
     ≡ (Γ₂ : Con₂) → (T₂ : Ty₂) → (Tms · (Γ ▹ T)) → Set
Γ₂ ▹ₚ T₂ (γ : (Tms · (Γ ▹ T))) = Γₚ (π₁ γ) × Tₚ (π₂ γ)

--  thus we will have
▹₂ : Con₂ → Ty₂ → Con₂
Γ₂ ▹₂ T₂ = (Γ ▹ T, Γ₂ ▹ₚ T₂)
-- we will not re-emphasize what each subscripted ₚ, ₂ means

-- id  : Tms Γ Γ
idₚ : TmsP Γ₂ Γ₂ id
    ≡ (γ : Tms · Γ) → (γₚ : Γₚ γ) → Γₚ (id ∘ γ)
idₚ = λ γ γₚ. γₚ
```

Are you also tired of the repeating `(γ : Tms · Γ) → (γₚ : Γₚ γ) → ...` ? We can simply again make a short hand : `γ₂ ∈ Γ₂` means `(γ, γ') : ∑ γ : Tms · Γ, Γₚ γ` Interestingly, we can see that `γ₂ ∈ Γ₂  iff  "γ₂" : Tms₂ ·₂ Γ₂`

```
-- _,_ : (σ : Tms Γ Δ) → Tm Γ A → Tms Γ (Δ ▹ A)
_,ₚ_ : (σ₂ : Tms₂ Γ₂ Δ₂) → (t₂ : Tm₂ Γ₂ A₂) → TmsP Γ₂ (Δ₂ ▹₂ A₂) (σ,t)
σ₂ ,ₚ t₂ = λ γ₂. (σ₂.1 γ γₚ, t₂.1 γ γₚ)
```

```
-- π₁ : Tms Γ (Δ, A) → Tms Γ Δ
π₁ₚ : (σ₂ : Tms₂ Γ₂ (Δ₂ ▹₂ A₂)) → TmsP Γ₂ Δ₂ (π₁ σ)
π₁ₚ σ₂ =
  λγ₂.
    let σ₂.1 γ γₚ : (Δ₂ ▹ₚ A₂) ( σ₂.0 ∘ γ )
                  = Δ₂ (π₁ (σ ∘ γ)) × A₂ (π₂ (σ ∘ γ))
    in (σ₂.1 γ γₚ).0 --  because π₁ (σ ∘ γ) = π₁σ ∘ γ

⇒ₚ : (A₂ : Ty₂) → (B₂ : Ty₂) → Tyₚ (A ⇒ B)
    = (A₂ : Ty₂) → (B₂ : Ty₂) → Tm · (A ⇒ B) → Set
A₂ ⇒ₚ B₂ = λ t. ∀ (a₂ : Tm₂ ·₂ A₂) → TmP ·₂ B₂ ((app t)[(id, a)])

lam  : Tm (Γ ▹ A) B → Tm Γ (⇒ A B)
lam₂ : (t₂ : Tm₂ (Γ₂ ▹₂ A₂) B₂) → TmP Γ₂ (⇒₂ A₂ B₂) (lam t)
lamₚ t₂ =
  λγ₂.λ a₂ .λ _ _ . tₚ (γ, a) (γₚ, aₚ) : Bₚ (t[(γ, a)])

appₚ : (t₂ : Tm₂ Γ₂ (⇒₂ A₂ B₂)) → TmP (Γ₂ ▹₂ A₂) B₂ (app t)
appₚ t₂ =
  λγ₂. t₂ (π₁ γ) ((γₚ).0) (π₂ γ, (γₚ).1) _ _
```

Of course, as a model, we need to verify the equational rule/naturality conditions. Because we make sure the syntax piece is unchanged, we only need to verify the part for the guarding predicate. The verification is done by unfolding the definition and in a real formalization, agda should be able to check the result for you.

```
⇒β₂ : app₂ (lam₂ t₂) = t₂
⇒η₂ : lam₂ (app₂ t₂) = t₂
```

Now for the only base type – boolean type. The guarding predicate is saying every closed element of boolean type is judgementally equal to either true or false. We omit the detailed construction because it is trivial.

```
Bₚ : Tyₚ B
  ≡ Tm · B → Set
Bₚ v = (v ≡ tt) + (v ≡ ff)
-- tt : Tm Γ B
ttₚ : Tmₚ Γ₂ B₂ tt
-- ff : Tm Γ B
ffₚ : Tmₚ Γ₂ B₂ tt
```

Then we are done with the canonicity model construction. To prove the canonicity, we simply look at the glued element for an arbitrary closed boolean `t : Tm · B`

```
(recTm t) : Tm₂ ·₂ B₂ ≡ ∑ t. Tmₚ Γ₂ B t
(recTm t).1
  : Tmₚ ·₂ B₂ t
  ≡ γ :Tms · · → γₚ : ·ₚ γ  → Bₚ t[γ]
  ≡ γ :Tms · · → γₚ : ⊤ → (t[γ] ≡ tt) + (t[γ] ≡ ff)

canonicity : (t : Tm · B) → (t ≡ tt) + (t ≡ ff)
canonicity t = (recTm t).1 id () : (t ≡ tt) + (t ≡ ff)
```

2.6 A Real Big-Step Interpreter

Currently we cannot return a good representative for closed terms of function type. Moreover, it is actually hard for us to formalize the specification because we lose the ability to talk about literal equal between representative.

The easiest way to retrieve that ability is to make an unquotient copy of our syntax[15], raw terms and substitutions `RTm RTms`, defined by vanilla inductive type. Then we will have an injection `inj : RTm → Tm` that acts "almost like identity" .

Note 2.11. Of course, we can't have a reverse function `Tm → RTm` that almost act like identity. This is because `lam (app t)` and `t` are equal in `Tm` but not equal in `RawTm` and we cannot have a function in our meta-logic that maps equal things to unequal things. This is also the reason how we can prove there is a bijection between Example 2.2 and boolean type. It should be checked by the eliminator of the QIT that the function(morphism) is well-defined with respect to the quotiented equality, but when we use the first approach in Note 2.4 we will have to write down the eliminator by ourselves. Thus Agda can only automatically check this *well-defined-ness* when the second approach is used.

Then we can still carry out a logical relation proof using dependent elimination, however this time, we don't need to check the naturality condition anymore. We will also use `inj` to indicate the $\beta\eta$-equality between raw terms. Interestingly, most of the logical relation proof is actually unchanged, we will only spelled out the following motives/definitions that are different from the original. `whnf` stands for *weak head normal form*, in our context, it includes all the lambda and tt, ff.

```
RConP : Con → Set
RConP Γ = (t : Tms · Γ) → Set

RTyP : Ty → Set
RTyP T = (t : Tm · T)   → Set

RTmsP : (Γ₂ : RCon₂) → (Δ₂ : RCon₂)
      → RTms Γ Δ → Set
RTmsP (Γ, Γₚ) (Δ, Δₚ) σ =
  ((γ : RTms · Γ) → (γₚ : Γₚ (inj γ)) → Δₚ (inj (σ ∘ γ)))

RTmP : (Γ₂ : RCon₂) → (T₂ : RTy₂)
     → RTm Γ T → Set
RTmP (Γ, Γₚ) (T, Tₚ) t =
  ((γ : RTms · Γ) → (γₚ : Γₚ (inj γ)) → Tₚ (inj t[γ]))

Bₚ : RTyP B = Tm · B → Set
Bₚ v = (∑ r, (t ≡ inj r) × whnf r)

⇒ₚ : (A₂ : RTy₂) → (B₂ : RTy₂) → RTyP (A ⇒ B)
   = (A₂ : RTy₂) → (B₂ : RTy₂) → Tm · (A ⇒ B) → Set
A₂ ⇒ₚ B₂ =
   λ t.  ((a₂ : RTm₂ ·₂ A₂)
           → RTmP ·₂ B₂ ((app t)[(id, a)]))
             × (∑ r, (t ≡ inj r) × whnf r)
```

[15]Copy all the constructors except for those identity constructors. Still dependent on the original `Con, Ty`

Then the proof is basically unchanged compared to Canonicity proof, only some places we need to plug in `inj` to cast from raw terms to quotiented terms. We also don't need to verify equations because it is not QIT. Then we can have a correct big-step interpreter

```
eval : (t : RTm · T) → ∑ r : RTm · T. (inj r ≡ inj t) × whnf r
```

by using this logical relation proof : look at the interpretation of closed terms of boolean type and function type, `RTmP · B₂` and `RTmP · (X₂ ⇒₂ Y₂)`, unfold these two we can find `(∑ r, (t ≡ inj r) × whnf r)` for both parts, which is essential how our big-step interpreter is defined upon.

Now that we have a big-step interpreter, we can prove Computational Adequacy and that judgemental equality implies Observational Equivalence.

2.6.1 Computational Adequacy. The denotational model of our raw terms is defined by the composition of injection and original model for QIT.

```
_ₘ : Tm Γ T → Tmₘ Γₘ Tₘ      -- recursor to the standard model
denote : RTm Γ T → Tmₘ Γₘ Tₘ
denote t = (inj t)ₘ          -- same, but for raw term

comp_adq : {t : RTm · B} → denote t ≡ true ↔ eval t ≡ tt

-- right to left is trivial, because
eval t ≡ tt  : RTm · B
⇒ inj t ≡ tt : Tm · B -- by definition of eval
⇒ denote t ≡ (inj t)ₘ ≡ true : Agda.Bool


denote t ≡ (inj t)ₘ ≡ true : Agda.Bool
⇒ inj t ≡ tt   : Tm · B -- inj t is either tt or ff by canonicity
⇒ eval t ≡ tt  : RTm · B
```

2.6.2 Judgemental Equality Implies Observational Equivalence. Now we have unquotiented term and big step evaluator, we can state observational equivalence, and the main theorem we want to prove.

```
_≈_ (a b : RTm Γ T) : Set
a ≈ b = (ctx : (RTm Γ T ↝ RTm · B)) → eval (plug ctx a) ≡ eval (plug ctx b)

jeq→obeq : inj a ≡ inj b → a ≈ b
```

We omit the definition of context but essentially it is just program with one hole and define inductively. `RTm Γ T ↝ RTm · B` means the hole is of type `RTm Γ T`, and once `plug`ing in the term, the whole program is closed and of type boolean.

`jeq→obeq` holds because of the following lemma, proved by induction on context

```
context_congruence :
  (ctx : (RTm Γ T ↝ RTm · B)) (a b : RTm Γ T)
  → inj a ≡ inj b → inj (plug ctx a) ≡ inj (plug ctx b)
```

This lemma is ultimately saying that each context will map equivalent terms to equivalent terms, i.e. context can be seen as a morphism `Tm Γ T → Tm · B` (respecting judgemental equality). This lemma holds because our judgemental equality is "rich" enough.

Using `context_congruence`, we know

```
inj a ≡ inj b
⇒ inj (eval (plug ctx a)) ≡ inj (plug ctx a)
   ≡ inj (plug ctx b) ≡ inj (eval (plug ctx b))
```

```
⇒ eval (plug ctx a) ≡ eval (plug ctx b)
-- because (plug ctx a) is closed boolean term
```

Note 2.12. Quoted from [43] : "*under the assumption of correctness for a compositional denotational semantics computational adequacy is equivalent to the requirement that denotational equality entails observational equality*". This suffices as another proof idea. However, to prove our denotational semantic is compositional in the current setting is an unnecessary detour.

2.7 Example : State Monad

Here, we adopt an uninteresting example: Haskell's State Monad (the simple version) in our syntax and show the model, to illustrate the adaptability of this framework.

```
-- new type State Monad
data Ty where
  St : Ty → Ty → Ty

data Tm where
  return : Tm Γ T →  Tm Γ (St A T)
  bind   : Tm Γ (St A T) → Tm (Γ ▷ T) (St A V) → Tm Γ (St A V)
  set    : Tm Γ T →  Tm Γ (St T T)
  get    : Tm Γ (St T T)
  run₁   : Tm Γ (St A T) → Tm Γ A → Tm Γ A
  run₂   : Tm Γ (St A T) → Tm Γ A → Tm Γ T
  return[] : (return t)[γ] ≡ return (t[γ])
  bind[] :   (bind t f)[γ] ≡ bind t[γ] (f[γ ↑ T])
  set[]  :   (set t)[γ] ≡ set t[γ]
  run₁[] :   (run₁ t st)[γ] ≡ run₁ t[γ] st[γ]
  run₂[] :   (run₂ t st)[γ] ≡ run₂ t[γ] st[γ]
  run₁βret:  run₁ (return t) st ≡ st
  run₂βret:  run₂ (return t) st ≡ t
  run₁βbind: run₁ (bind t f) st ≡ run₁ f[(id, (run₂ t st))] (run₁ t st)
  run₂βbind: run₂ (bind t f) st ≡ run₂ f[(id, (run₂ t st))] (run₁ t st)
  run₁βget:  run₁ get st ≡ st
  run₂βget:  run₂ get st ≡ st
  run₁βset:  run₁ (set s) st ≡ s
  run₂βset:  run₂ (set s) st ≡ s
```

The standard model is simple – it is exactly the implementation of State Monad for Haskell!

```
Stₘ Aₘ Tₘ   = Aₘ → Tₘ × Aₘ
bindₘ gₘ fₘ =
  λγ. λa. let (t, a') = gₘ γ a        in
          let (v, a'') = fₘ (γ, t) a' in
            (v, a'')
setₘ tₘ  = λγ. λ_ . (tₘγ, tₘγ)
getₘ = λγ. λt . (t, t)
```

The canonicity model is as expected. We only spell out the type – since after unfold the type, the construction is obvious.

```
Stₚ : (A₂ : Ty₂) → (T₂ : Ty₂) → Tyₚ (St A T)
Stₚ A₂ T₂ t = (st₂ : Tm₂ ·₂ A₂) → Aₚ (run₁ t st) × Tₚ (run₂ t st)

returnₚ : (t₂ : Tm₂ Γ₂ T₂) →  Tmₚ Γ₂ (St₂ A₂ T₂) (return t)
returnₚ t₂ γ₂ : (st₂ : Tm₂ ·₂ A₂)
```

```
        → Aₚ (run₁ (return t)[γ] st) × Tₚ (run₂ (return t)[γ] st)

bindₚ   : (t₂ : Tm₂ Γ₂ (St₂ A₂ T₂)) → (f₂ : Tm₂ (Γ₂ ▹₂ T₂) (St₂ A₂ V₂))
            → Tmₚ Γ₂ (St₂ A₂ V₂) (bind t f)
bindₚ t₂ f₂ γ₂ : (st₂ : Tm₂ ·₂ A₂)
        → Aₚ (run₁ (bind t f)[γ] st) × Vₚ (run₂ (bind t f)[γ] st)

setₚ    : (s₂ : Tm₂ Γ₂ T₂) →  Tmₚ Γ₂ (St₂ T₂ T₂) (set s)
setₚ s₂ γ₂ : (st₂ : Tm₂ ·₂ T₂)
        → Tₚ (run₁ (set s)[γ] st) × Tₚ (run₂ (set s)[γ] st)

getₚ    : Tmₚ Γ₂ (St₂ T₂ T₂) get
getₚ γ₂ : (st₂ : Tm₂ ·₂ T₂)
        → Tₚ (run₁ (get[γ]) st) × Tₚ (run₂ (get[γ]) st)
```

. To construct the interpreter, we only need to include all the constructor for `St` type as weak-head-normal-form.

Note 2.13. The equational rules/dynamics can be derived during proving canonicity. Once we come up with the motives for the predicates of `St`, all the β-rules (and actually substitution rule) are already there when unfolding the motives, even though it is definitely possible to directly come up with them. More interestingly, the canonicity argument here doesn't require any η-rule.

Note 2.14. This example is not so interesting because it can be simulated using function and tuples. A more interesting idea is to remove *run*, and directly let user to program on a monad (like how in Haskell *main* is a `IO ()`). Then the canonicity motive needs to be changed, and canonical form for `St` is no longer trivial. Example 2.7 of [16] suggests using bind with get and set as one canonical form.

Note 2.15 (Free CCC as syntax, Initiality Conjecture). The reader might notice that the semantic (standard model) has a lot of duplicate parts – for example, `Con, Ty` is interpreted in the same way. This actually hints the possibility of a better and more "mathematical" syntax – Free CCC.

We will not talk about rigorously how free functor is adjoint to the forgetful functor and free CCC is generated from a graph[29]. Intuitively speaking, a free CCC *looks like* the term model[16], i.e. the model constructed by quotiented out the equation between terms and substitutions in the theory of STLC . We can also express all the STLC inside this free CCC because "STLC is the internal language of CCC"[2, 25]. That is the reason, alternatively, we can choose to **define** free CCC as the syntax of STLC [40].

A more interesting result is that, since the term model is considered as an model of the original theory, which is inside the category of model and homomorphism. But each model will have to satisfy the theory, and the term model seems the "most relaxed" model that satisfies the theory. So it is natural to ask, is the term model the initial object of the category of algebra(model) and homomorphism?(i.e. Is there always a unique morphism from the term algebra to arbitrary algebra?) This kind of conjecture is usually phrased as *initiality conjecture*. For simple type theory it is already proved[14].

3 NORMALIZATION BY EVALUATION (NBE) FOR STLC

This section is a review on [12, 42]. I didn't manage to work out the details, but I will sketch out the algorithm and the proof.

[16] Also called Lindenbaum-Tarski algebra[1]

The motivations for Normalization by Evaluation(NbE) are various. (A). Our Real Big-Step Interpreter is interpreting $\beta\eta$-equal lambda terms into different raw term (concrete presentation), but it is more aesthetic to have equal representation back ; (B). We want to evaluate expressions with free variable, acting like partial evaluation and optimization; (C). We want to replay the Strong Normalizing Theorem in our *reduction-free system*; (D). We want to prove that our current judgemental equality is decidable, which will later be useful in dependent types[17].

3.1 Rough Specification of NbE

Thus the specification for NbE includes: 1. An NbE function `nf` needs to choose a **unique representative** from each equivalence class; 2. This unique representative cannot be further *simplified*. We call these unique representative ***normal forms***. They are of type `Nf : Con → Ty → Set` with `inj: Nf → Tm`, and thus are also unquotiented to stand for non-reducidble concrete raw term[18]. Thus we know the signature `nf : Tm Γ T → Nf Γ T`, with the following commitment:

- *Soundness* `t ≡ t' → nf t ≡ nf t'`
- *Completeness* `t ≡ inj (nf t)`

Here, soundness means that our `nf` will choose a unqiue unquotiented term to return; completeness means we choose a valid representation for each equivalence class. These two together implies $\Gamma \vdash u \equiv v : T \iff \mathtt{nf}(u) \equiv \mathtt{nf}(v)$, which satisify our Specification 1. Interestingly, in QIIT framework, we get soundness for free because morphism out of QIIT needs to respect equality/quotient. The only challenge is completeness.

Now we need to sort out the concrete definition of the inductive type `Nf : Con → Ty → Set` to satisfy the Specification 2.

Note 3.1. Apparently, there is some controversy on what represents soundness and completeness here[3]. We are following the convention of [26], which is opposite to the convention of [6]. The reason we follow [26] is simply because we think `nf t ≡ nf t'` is something on the semantic side.

3.2 Normal Form and Neutral Form

The reason we ant normal form is to expose *concrete irreducible representation for each equivalence class of* ***open terms*** . First they must include something like `tt,ff`, i.e. *constructors*. But they also need to include *blocked computation*, i.e. `ifb x ff tt` – then we notice that the first idea is not concrete enough because we can have constructors wrapping blocked computation, i.e. `λ z. ifb x ff tt`.

Thus to characterize normal form, we need another notion of *neutral form*, which loosely corresponds to those blocked computation. Then, we define normal form based on neutral form. Here we list out some of the definitions.

```
data Nf : Con → Ty → Set where
data Ne : Con → Ty → Set where
data Nfs : Con → Con → Set where
data Nes : Con → Con → Set where
  nenf : Ne Γ A → Nf Γ A
  lam : Nf (Γ ▷ X) Y → Nf Γ (X ⇒ Y)
  if   : Ne Γ B → Nf Γ A → Nf Γ A → Ne Γ A
```

[17] Decidable judgemental equality is one key problem to the decidablity of type checking for dependent type because in dependent type, equality of terms appears in a lot of places in inference rules

[18] They are unquotiented due to the same reason when we are developing the big-step interpreter – we need literal equality/non $\beta\eta$-equality

```
_,_  : Nfs Γ Δ → Nf Γ T → Nfs Γ (Δ ▹ T)
_,_  : Nes Γ Δ → Ne Γ T → Nes Γ (Δ ▹ T)
                                  **   ***   **
```

Now we know the concrete specification, we still don't know how to define this `nf`. Let's step back a little bit and think about the concrete implementation.

3.3 What it should look like algorithmically?

Say we are writing this NbE function in Haskell – which will do as much reduction as possible by just traversing and evaluating each node of the AST tree (if possible) recursively . Doing so will **reflect** the term of the object language into Haskell's data. One most straight-forward idea is to evaluate an open term t a function $\tilde{t}$ in Haskell, which requires the "value" of free variable as input. As we believe there is no redex anymore, we need to **reify** Haskell's function $\tilde{t}$ back into an AST data t'. One way to do that is at the beginning make that $\tilde{t}$ to accept AST (or just neutral form) and also outputing AST (or just normal form).

This is actually the big picture of algorithmically doing NbE, two steps: evaluation and reifiction.

Evaluation is just "eval" an AST in our metalogic, because AST are open terms, so the "evaluated form" should be a function in metalogic, requiring as input the free variable and outputing the final result
If we use haskell as our metalogic, then "eval" is of type: `openAST → ("Value" → "Value")`.

Reification can try to *read-back* the "evaluated form" to arrive at the corresponding AST.
You might think this part is weird : the "evaluated form" in our haskell is a *function*, how can we reify a **function/piece of code in haskell** into our AST data structure inside Haskell? The way to solve that is to allow the input of free variables themselves as `"Value"`, and `"Value"` should also include AST/normal forms themselves (so that the function can directly output a normal form as AST data)

Now, let's try to translate the above intuition into our QIIT framework. We first work on evaluation, which is a function out of QIT syntax, and thus an **algebra** of our syntax. According to our above sketch, "the evaluated form" needs to handle and transform AST or normal forms. What exactly does it look like? Say what should be the interpretation of base type `B`? The answer the category theorists gave is presheaf.

3.4 Translation: Presheaf Over Renaming Category

A presheaf is just a contravariant functor $C^{op} \to \mathrm{Set}$ parametrized by some base category C. The category of presheaf contains presheaves as objects , and natural transformations as morphisms. The intuition of presheaf model is that, consider the CCC as the semantic of our STLC syntax[19], you will notice that, the (open and closed) term of boolean type is actually inside the hom-set $Hom(?, [\![\mathtt{B}]\!])$ (something close to $\{t : ? \vdash t : \mathtt{B}\}$)[20] instead of anything related to the objects in category. However, it is intuitive to want these elements to **really** be in the objects $[\![\mathtt{B}]\!]$ instead of at the hom-set. So the first idea is to have an algebra that interprets `B` into a set of elements of type `B`. But this is not going to work in an intrinsic setting like us because that will include boolean terms **in all kinds of contexts**, so we have to choose *presheaf model* – basically a "set of terms" parametric over the typing context. For example, boolean type, is interpreted as the "set" of normal form of boolean type, **parametric on the context**.

[19] Or Consider free CCC as our STLC syntax
[20] Just like how most category theory should work

Another good news is that most[21] presheaf categories are directly CCC, this ease the details for us, as we directly know what should be the product and exponential object. But here we choose *renaming category* as base category. *Renaming category* take the original `Con` as objects, but only variable permutations as morphisms. We will later illustrate the reason we use presheaf over renaming as the interpretation category. Before we sketch out the presheaf model for our QIT syntax, we first formalize presheaf over renaming, thanks to [26].

```
data Var : Con → Ty → Set where
  vz : Var (Γ ▷ A) A
  vs : Var Γ A → Var (Γ ▷ B) A

data Ren : Con → Con → Set where
  ε : Ren Γ ·
  _,_ : Ren Γ Δ → Var Γ A → Ren Γ (Δ ▷ A)

record PShR : Set where
  field F₀ : Con → Set
        F₁ : Ren Γ Γ' → F₀ Γ' → F₀ Γ

-- natural transformation!
record _↠_ : (F : PShR) → (G : PShR) → Set where
  field FG₀ : {Γ : Con} → F₀ Γ → G₀ Γ
        FG₁ : {h : Ren Γ Γ'}  → FG₀ ∘ (F₁ h) ≡ (G₁ h) ∘ FG₀

-- we assume we already prove it as a CCC category
```

Using this notion, we also introduce YR as a helpful notion that will map stuff into presheaf category:

```
YRNfs : Con → PShR
YRNfs Δ = record {F₀ = λ Γ . Nfs Γ Δ ; F₁ = ..}
YRTms : Con → PShR
YRTms Δ = record {F₀ = λ Γ . Tms Γ Δ ; F₁ = ..}
YRNf  : Ty → PShR
YRNf  T = record {F₀ = λ Γ . Nf Γ T ; F₁ = ..}
YRTm, YRNe   ,..  : Ty → PShR
YRNes, YRRen ,..  : Con → PShR
```

Basically for each kind of representation of terms, if we attach YR[22] in the front, then its argument flipped and is also a presheaf. The earlier defined `inj` is also a natural transformation: `inj Γ : YRNf Γ ↠ YRTm,  inj Γ : YRNe Γ ↠ YRTm, ...`.

Now we can define the presheaf model, just by directly refering to [4]:

```
ConM : Set
ConM = PShR
TyM  = PShR
TmsM : ConM → ConM → Set
TmsM ⟦Γ⟧ ⟦Δ⟧ = ⟦Γ⟧ ↠ ⟦Δ⟧
TmM ⟦Γ⟧ ⟦A⟧ = ⟦Γ⟧ ↠ ⟦A⟧

·M : ConM
·M = record {F₀ = λ_ . ⊤ ; F₁ = λ_ _. true}
```

[21] Presheaf over small category will be CCC [4].

[22] This Y stands for yoneda, this R stands for renaming.

```
-- directly use product object in presheaf category
_▹M_ : ConM → TyM → ConM
⟦Δ⟧ ▹M ⟦A⟧ = record {F₀ = λ Γ . ⟦Δ⟧Γ × ⟦A⟧Γ ;
  F₁ (h : Ren Γ Γ') ((a , b) : ⟦Δ⟧Γ' × ⟦A⟧Γ') =
        (⟦Δ⟧₁ h a, ⟦A⟧₁ h b)}

-- We use CCC similarly on other parts ...
-- ... now look at functions
--    we use exponential of presheaf,
⇒M : TyM → TyM → TyM
⟦A⟧ ⇒M ⟦B⟧ : TyM = Con → Set
⟦A⟧ ⇒M ⟦B⟧ = record { F₀ Γ = (YRRen Γ × ⟦A⟧) ↠ (⟦B⟧) ; F₁ = .. }
```

Note 3.2 (Definable Function). [42] uses a pullback (indicating definable function) instead of exponential in the presheaf category to model the lambdas. I currently cannot work out the detailed reason of doing that, and why that pull back can still act like an exponential in the category.

3.5 Reification and Reflection

Once we evaluate the AST into presheaf model, we need to reify the presheaf model into normal form. Recall that we can *roughly think* the presheaf model for each type/context is a "variant set" of terms/substitutions of/into that type/context (in normal form), i.e. $[\![T]\!](\Gamma) \approx \{t : \Gamma \vdash t : T\}$. Thus reification will map this *functor* $[\![T]\!]$ ("variant set" of terms/substitutions) into corresponding *functor* $[\![T]\!]$ (terms/substitutions) in normal forms.

```
↓Con : (Δ : Con) → ⟦ Δ ⟧  ↠ YRNfs Δ
↓Ty  : (A : Ty)  → ⟦ A ⟧  ↠ YRNf  A
```

`↓Con` purely relies on `↓Ty`, and `↓Ty B` is trivial, the challenge mainly lies in defining

```
↓Ty (X ⇒ Y) Γ : ⟦(X ⇒ Y)⟧ Γ → YRNf (X ⇒ Y) Γ
                ≡ (F : ((YRRens Γ) × ⟦X⟧) ↠ ⟦Y⟧) → YRNf (X ⇒ Y) Γ
```

Recall earlier: we will try to input a *literal fresh* variable to reify a Haskell function body. Look at our current motive: if we want to input a fresh variable into `F`, we needs to first apply this natural transformation with the weakened context `(Γ ▹ X)` so that we have a fresh variable for type `X`. But we need another fresh variable in the form of `⟦X⟧ (Γ ▹ X)`. This hints the necessity of the *reflection* – it *reflects* a neutral value into the presheaf model, symmetric to ho the reification take in presheaf model.

```
↑Con : (Δ : Con) → YRNes Δ ↠ ⟦ Δ ⟧
↑Ty  : (A : Ty)  → YRNe  A ↠ ⟦ A ⟧
```

Then we can define reflection and reification for function type, here we only spell out the behaviour of natural transformation on object Γ.

```
↓Ty (X ⇒ Y) Γ F =
  let F' = F₀ (Γ ▹ X) : YRRen Γ (Γ ▹ X) × ⟦X⟧ (Γ ▹ X) → ⟦Y⟧ (Γ ▹ X)
  let t₁ = F' (π₁, ↑Ty X (Γ ▹ X) π₂) : ⟦Y⟧ (Γ ▹ X)
  let t₂ = (↓Ty Y (Γ ▹ X) t₁)        : YRNf Y (Γ ▹ X)
  in (lam t₂)                        : YRNf (X ⇒ Y) Γ
```

Note 3.3 (Presheaf over Renaming and Kripke Model). The reason we use presheaf over **renaming**, as you now can notice, is illustrated in the reification for the model $[\![$`lam t`$]\!]$: When we reify the exponential into a lambda – we require to take out a **fresh** variable from the context and apply to the presheaf model. The fresh variable is in the weakened context and thus our modeling of function will have to work all the weakened context, and this is where the part *renaming category* comes.

With the help of presheaf model and reification, we can spell out the translation of our intuition of `nf`

3.6 Complete Specification and Proof Sketch of NbE

Now that, we have all the pieces in mind, we can write out the concrete algorithmic description of nbe function

```
nf : Tm Γ T → Nf Γ T
nf t =  ((↓Ty T Γ) ∘ ⟦ t ⟧ Γ  ∘ (↑Con Γ Γ)) (i')
where  ↑Con Γ Γ   : YRNes Γ Γ → ⟦ Γ ⟧Γ
       ⟦ t ⟧ Γ    : ⟦ Γ ⟧Γ → ⟦ T ⟧Γ
       ↓Ty T Γ    : ⟦ T ⟧Γ → YRNf  T Γ
```

Now to prove completeness `inj (nf t) ≡ t`, we need to prove two invariants[42].

- *Reify-Reflect Yoga:* `inj Γ ∘ ↓Con Γ ∘ ↑Con Γ ≡ inj Γ`
  ```
  where ↑Con Γ : YRNes Γ ↠ ⟦ Γ ⟧
        ↓Con Γ : ⟦ Γ ⟧  ↠ YRNfs Γ
        inj Γ  : YRNfs Γ ↠ YRTms Γ
        inj Γ  : YRNes Γ ↠ YRTms Γ
  ```
- *Logical Relation:* `inj T ∘ ↓Ty T ∘ ⟦ t ⟧ ≡ t[-] ∘ inj Γ ∘ ↓Con Γ`
  ```
  where t : Tm Γ T
        ⟦t⟧ : ⟦ Γ ⟧ ↠ ⟦ T ⟧
        t[-] : YRTms Γ ↠ YRTm T  -- a term waiting for substitution
        inj Γ : YRNf T ↠ YRTm T
        inj T : YRNf T ↠ YRTm T
  ```

Note 3.4. Reify-Reflect Yoga is simply stating neutral value won't be changed after reifying and reflecting. The logical relation is claiming syntax and semantic(presheaf model) is connected by the reification.

These two invariants can help us prove the completeness [12, 42]

```
inj T Γ (nf t) ≡ ((inj T Γ) ∘ (↓Ty T Γ) ∘ ⟦ t ⟧ Γ ∘ (↑Con Γ Γ)) (id')
≡ (inj T ∘ (↓Ty T) ∘ ⟦ t ⟧ ∘ (↑Con Γ)) Γ (id')
          -- (they are all natural transformations)
≡ ((t[-] ∘ inj Γ  ∘ ↓Con Γ) ∘ (↑Con Γ)) Γ (id')   -- by second invariance)
≡ (t[-]  ∘ inj Γ) Γ (id')                         -- by first  invariance)
≡ t[-]Γ (inj Γ Γ id') ≡ t[-] Γ id                 -- by definition
≡ t[id] ≡ t
```

Among the two invariants, Reflect-Reify Yoga is claimed to be proven by induction on the type construction [12, 42], and that makes the logical relation the key part of the proof.

Note 3.5 (Kripke Model). Sometimes the presheaf model over renaming category is also called a generalization of kripke model. The reason is that, in other formulation of correct of NbE, the logical relation is called kripke logical relation – a statement holds in all weakened contexts. We directly cite from Sec 2.6 of [6]:

$\Gamma \vdash r : S \to T \circledR f \iff \forall \Gamma' \leq \Gamma.\Gamma' \vdash s : S \circledR a \Rightarrow \Gamma' rs : T \circledR f(a)$

You can see the morphism between renaming is a direct generalization of the weakening relation $\leq$ (accessibility/possible-world relation) in the Kripke Model formulation.

4 CANONICITY FOR MLTT

In STLC , the judgemental equality is only acting as a specification of the contextual equivalence we want. In Martin Lof Type Theory (MLTT), it is no longer the case because a lot of typing rules

requires the presence of judgemental equality. MLTT also requires type dependent on term and thus brings in a lot complexity including the notion of type substitution. However, the proof of canonicity is a direct generalization from that of STLC – we still use glued elements. One obvious distinction is the presence of universe level in a predicative setting.

Note 4.1 (Universe). For the readers that are unfamiliar with universe – in dependent type theory, we will have a type U – usually written as **Set** in proof assistant – that includes all kinds of small type, i.e. `Bool : U`, thus it is naturally to think of having some device of $U : U$, which is of course leading to paradox[30][23]. Thus an easy way out is to has *levels* assigned to each U, i.e. `Bool : U₀ : U₁`, and also cumulative `T : Uᵢ ⇒ T : Uᵢ₊₁`. This is called predicative cumulative universe. Agda is adopting this style.

One simple inspection is that an identity function `t₀ : (T : U₀) → T → T : U₁` will have universe level increased, and thus it is not possible to instantiate the type parameter of `t₀` with its own type. But recall in System F, there is no such thing as level, and $\cdot \vdash t_1 : \forall \alpha.\alpha \to \alpha$ is still a valid type judgement, here t_1 can be applied to itself. Thus we call System F to have *impredicativity*. There is of course dependent type theory with impredicative universe.

When operation universe level, we will use `lsuc l, l ∪ j` to indicate $l + 1$ and the $min(l, j)$.

However, in our review we will only specify the motives and the important parts and omit most details (for clarity). Interested readers can find a complete version in [21, 27].

4.0.1 Syntax for MLTT. Very similar to STLC , but now it is Quotient **Inductive-Inductive** Type, abbreviated as QIIT. Inductive-Inductive Type, can be consider as stronger form of mutual inductive type where we have `A : Set, B : A → Set` , define together. Concretely, for MLTT, the index of `Ty`, `Con` is defined together with `Ty`.

```
Variable
  α β : Level.

data Con : (α : Level) → Set (lsuc α) where
data Ty  : Con α (β : Level) → Set (α ∪ lsuc β) where
-- type-term dependency is here, needs QIIT to express
data Tms : Con α → Con β → Set (α ∪ β) where
data Tm : (Γ : Con α) → Ty β Γ → Set (α ∪ β) where
  -- T : Ty Γ is indicating T is a well-formed type in context Γ
  -▹- : (Γ : Con α) → Ty β Γ → Con (α ∪ β)  -- context extension
  -[-] : Ty α θ → Tms Γ θ → Ty α Γ  -- type substitution
  -,- : (σ : Tms Γ Δ) (A : Ty α Δ) → Tm Γ A[σ] → Tms Γ (Δ ▹ A)
  -[-] : Tm θ A → (σ : Tms Γ θ) → Tm Γ A[σ]
  π₂ : (σ : Tms Γ (Δ ▹ A)) → Tm Γ A[π₁ σ]
  -- several operations now require type substitution now

-- debruijn shifting function
_↑_ : (σ : Tms Γ Δ) → Ty α Δ → Tms (Γ, A[σ]) (Δ, A)
σ ↑ A = (σ ∘ π₁, π₂)

-- Cumulative Predicative Universe
data Ty where
  U : (α : Level) → Ty α Γ
  c : Ty α Γ → Tm Γ (U (lsuc α))
  El : Tm Γ (U (lsuc α)) → Ty α Γ
```

[23] Something similar to Russell's Paradox

```
  U[] : U[σ] ≡ U
  El[] : (El A)[σ] ≡ El(tr (U[]) A[σ])
  Elc : El (c A) ≡ A
  cEl : c (El a) ≡ a

-- dependent function type and bottom
data Ty where
  ∏ : (A : Ty α Γ) → Ty β (Γ ▹ A) → Ty (α ∪ β) Γ
  ∏[] : (∏ A B)[σ] ≡ ∏ (A[σ]) (B[σ ↑ A])
  ⊥    : Ty Γ
  Bool : Ty Γ
```

Note 4.2 (Logical Framework for MLTT). Now we can see the power of QIIT + (fake) Agda as a logical framework, since it can represent the syntax of MLTT.

Of course, there are other logical frameworks for formalizing MLTT. For example, generalized algebraic theory[24] [19], Uemura's [44] and Sterling's [40].

4.0.2 Semantic for MLTT. Then like before, we construct the standard model for demonstrating relative consistency.

```
Variable
    i j : Level

-- Consistency Model, or Standard Model
record StandardModelᴹ : Set where
  Conᴹ : (n: Level) → Setₙ₊₁
  Conᴹ = λ(n:Level). Setₙ
  Tyᴹ  : (j : Level) → Conᵢᴹ → Set (i ∪ lsuc j)
  Tyᴹ j (Γ : Setᵢ) = Γ → Setⱼ
  Tmsᴹ : Conᵢᴹ → Conᵢᴹ → Setᵢ
         ≡ Setᵢ → Setᵢ → Setᵢ
  Tmsᴹ Γ Δ = Γ → Δ
  Tmᴹ  : (Γ : Conᵢᴹ) → Tyᴹ i Γ → Setᵢ
         ≡ (Γ : Setᵢ) → (Γ → Setᵢ) → Setᵢ
  Tmᴹ  Γ A = (env : Γ) → (A env)
  ·ᴹ   : (Conᵢᴹ ≡ Setᵢ)      = ⊤
  ▹ᴹ   : ((Γ: Conᵢᴹ) → Tyᴹ j Γᴹ → Con (i ∪ j)ᴹ)
         ≡ ((Γ: Setᵢ) → (Γᴹ → Setⱼ) → Set (i ∪ j))
     = λ Γ (h :  Γᴹ → Setⱼ). ∑ (γ:Γ). h γ  -- corresponds to term definition
  -[-]ᴹ = λ f g. f ∘ g -- composition!
  -,- = λ σ g γ. (σ(γ), g γ)
  Uᴹ = λ (i : Level) (_ : Γ). Setᵢ
  cᴹ = λ (f : Γ → Setᵢ) (γ : Γ). f γ
  El : Tmᴹ Γᴹ Uᵢᴹ → Tyᴹ i Γᴹ
       ≡ ((γ : Γᴹ) → (Uᵢ γ)) → Γᴹ → Setᵢ
       ≡ (Γᴹ → Setᵢ) → Γᴹ → Setᵢ
  El = id
  U[] = refl
  El[] = refl

  ∏ᴹ : (Aᴹ : Tyᴹ i Γᴹ) → Tyᴹ i (Γᴹ ▹ᴹ Aᴹ) → Tyᴹ i Γᴹ
  ∏ᴹ = λ (A : Γ → Setᵢ) (B : (∑ γ:Γ, A γ) → Setᵢ) (γ : Γ).
```

[24] This is untyped but more expressive than QIIT

```
        (a : A γ) → B (γ, a)
  ⊥ᴹ = λ γ. Empty
  Boolᴹ = λ γ. Agda.Bool
  -- define that "data Empty where end."
```

This time, we can also use this model to prove ¬ Tm · ⊥, i.e. there is no derivation of a closed term of type ⊥.

Note 4.3. If you wonder how to assign universe level, my suggestion is to first construct syntax and standard model without universe level, and then assign universe level to standard model so that things make sense, then extract the syntax from it.

4.0.3 Logical Relation for Canonicity. Still the only difference locate at universe level. The way I approach it is to come up with syntax and model **ignoring** universe level first, then assign universe level accordingly. Still a lot of details are omitted.

```
Variable
  i j k l : Level

record CanonicityModelᴹ : Set where
Conₚ : (i : Level) → Con i → Set (lsuc i)
Conₚ i Γ = Tms · Γ → Set i

-- like before, we use Con₂ to indicate the glued data
-- Con₂ i = ∑ (Γ : Con i), Conₚ i Γ,
-- we also write for Γ₂ : Con₂ i and T₂ : Ty₂ j Γ₂
--  "γ₂ ∈ Γ₂" as the short hand of "γ₂ : ∑ γ : Tms · Γ, Γₚ γ"

Tyₚ : Con₂ i → Ty j Γ → Set (i ∪ (lsuc j))
Tyₚ Γ₂ T = γ₂ ∈ Γ₂ → Tm · T[γ] → Set j
Tmsₚ : (Γ₂ : Con₂ i) → (Δ₂ : Con₂ j) → Tms Γ Δ → Set (i ∪ j)
Tmsₚ Γ₂ Δ₂ σ = γ₂ ∈ Γ₂ → Δₚ (σ ∘ γ)
Tmₚ  Γ₂ T₂ t = γ₂ ∈ Γ₂ → Tₚ γ₂ (t[γ])
·ₚ : Conₚ i ·
·ₚ = λ _ → ⊤
Γ₂ ▹ₚ T₂ : Conₚ (i ∪ j) (Γ ▹ T)
Γ₂ ▹ₚ T₂ = λ (γt : Tms · (Γ ▹ T)) → ∑ (γₚ : Γₚ (π₁ γt)), Tₚ (π₁ γ, γₚ) (π₂ γt)
σ₂ ∘ₚ τ₂ = λ γ₂.  σₚ (τ ∘ γ, τₚ γ₂)
T₂[σ₂]ₚ = λ γ₂ t. Tₚ (σ₂ ∘ γ₂) t
t₂[σ₂]ₚ θ₂ = tₚ (σ₂ ∘₂ θ₂)
⊥ₚ : Tyₚ i Γ₂ ⊥
   ≡ γ₂ ∈ Γ₂ → Tm · ⊥[γ] → Set i
⊥ₚ = λ γ₂ t. False
Boolₚ = λ γ₂ t. {t ≡ tt} + {t ≡ ff}
ttₚ : Tmₚ Γ₂ Bool₂ tt
    ≡ γ₂ ∈ Γ₂ → Boolₚ γ₂ (tt[γ])
ttₚ = λ γ₂ . -- holds trivially, similar for ffₚ
∏ₚ : (A₂ : Ty₂ i Γ₂) → (B₂ : Ty₂ j (Γ₂ ▹₂ A₂)) → Tyₚ (i ∪ j) Γ₂ (∏ A B)
∏ₚ A₂ B₂ = λ γ₂ t.
  a₂ ∈ A₂[γ₂]₂ → Bₚ (γ₂ ,₂ a₂) ((app t) [(γ, a)])

lamₚ : (t₂ : Tm₂ (Γ₂ ▹₂ A₂) B₂) → Tmₚ Γ₂ (∏₂ A₂ B₂) (lam t)
lamₚ t₂ : γ₂ ∈ Γ₂ → a₂ ∈ A₂[γ₂]₂ → Bₚ (γ₂ ,₂ a₂) t[(γ, a)]
lamₚ t₂ = λ γ₂ a₂. tₚ (γ₂ ,₂ a₂)
```

```
Uₚ : (i : Level) → Tyₚ (lsuc i) Γ₂ (U i)
Uₚ i = λ γ₂ t. (Tm · (El t)) → Set i -- exactly Tyₚ

cₚ : (T₂ : Ty₂ i Γ₂) → Tmₚ Γ₂ (U₂ i) (c T)
cₚ T₂ : γ₂ ∈ Γ₂ → (Tm · T[γ]) → Set i
cₚ T₂ = λ γ₂ t. Tₚ γ₂ t  --  ≡ Tₚ by eta

Elₚ : (t₂ : Tm₂ Γ₂ (U₂ i)) → Tyₚ i Γ₂ (El t)
Elₚ t₂ : γ₂ ∈ Γ₂ → Tm · (El t)[γ] → Set i
Elₚ t₂ = λ γ₂ x. tₚ γ₂ x  -- ≡ tₚ by eta

-- verify naturality condition (those equation)
El₂ (c₂ T₂).1 ≡ Elₚ (c₂ T₂) ≡ cₚ T₂ ≡ Tₚ
```

Note 4.4 (Proof-Relevant Logical Relation). [21, 26] emphasizes the logical relation we carry out here is *proof-relevant*, because it has *structure* rather than *a mere proposition*, or put in other words, we are constructing a (fake) Agda model that is basically a piece of data and mapped from QIIT syntax. However, I am not knowledgeable enough to tell what is a *proof-irrelevant* logical relation in this context, and we refer the interested audience to [21].

5 CATEGORICAL GLUING AND LOGICAL RELATION

We have noticed that logical relation on proving canoncity and completeness for NbE. I think the interesting question here is **why** this complicated logical relation can fit into a model for our QIT syntax.

This is where categorical gluing comes in : category theorists try to use categorical gluing to give a complete and general characterization of logical relation. However, this usually requires sophisticated categorical background to have good enough intuition to do meta-theory reasoning using category. Here, I will instead just point out the connection between categorical gluing and our current formulation of logical relation.

Using comma category, we can describe the (partial) definition of categorical gluing in elementary category language.

Note 5.1 (Comma Category, Canonicity Model for STLC). For functors $f : \mathcal{C} \to \mathcal{E}, g : \mathcal{D} \to \mathcal{E}$, the comma category $f \downarrow g$ is a category

- objects are triples (c, d, α) where $c \in \mathcal{C}, d \in \mathcal{D}$, and $\alpha : f(c) \to g(d)$ is a morphism in $\mathcal{E}$,
- morphisms from (c_1, d_1, α_1) to (c_2, d_2, α_2) are pairs (β, γ), where $\beta : c_1 \to c_2$ and $\gamma : d_1 \to d_2$ are morphisms in $\mathcal{C}$ and $\mathcal{D}$, respectively, such that $\alpha_2 \circ f(\beta) = g(\gamma) \circ \alpha_1$.

Now we consider the syntax of STLC as free CCC $\mathcal{C}$, where types and contexts are objects and morphisms are terms and substitutions. Then we will construct the canonicity model as a subcategory of the comma category $(id_{Set} \downarrow Hom(1, -))$ where $id_{Set} : \mathbf{Set} \to \mathbf{Set}$ and $Hom(1, -) : \mathcal{C} \to \mathbf{Set}$.

More specifically, the canonicity model, which has to be an algebra of the syntax, will maintain the structure of CCC; and we further refine each object in the canonicity model to be an inclusion function in **Set** instead of arbitrary function in the original comma category; morphisms are still the commutative diagrams.

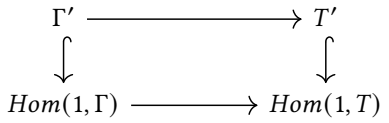
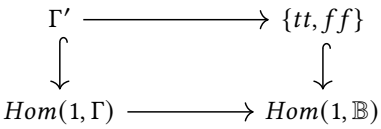


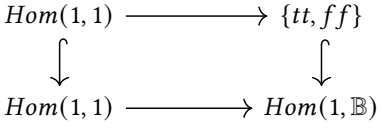
Fig. 2. the interpretation for Γ will be some certain subsets $\Gamma' \subseteq Hom(1, \Gamma)$, and the interpretation for the term/morphism `t : Tm Γ T` in CCC will be the whole commutative diagram.

Then, each object/interpretation for type (e.g. `Bool`) in the canonicity model will actually be a subset of the closed term (e.g. $S = \{tt, ff\} \subset Hom(1, \texttt{Bool}) = \texttt{Tm · Bool}$. The objects/interpretation for contexts are also subsets, as a big cartesian product of those subsets for the types.

Moreover, the terms/morphisms in the free CCC are mapped to a commutative diagram in the comma category – here we take the mapping of `tt : Tm Γ Bool` as example: $\Gamma' \subset Hom(1, \Gamma)$ as the mapping for the context, $tt : Hom(1, \mathbb{B}), tt' : Hom(\Gamma, \mathbb{B})$, trace the diagram, we have $\forall \gamma \in \Gamma', tt \circ \epsilon = tt'[-]$, which is exactly what we have to prove for `ttₚ : Tmₚ Γ₂ B₂ tt`.



(a) Canonicity Model for `ttₚ : Tmₚ Γ₂ B₂ t`



(b) Canonicity Proof from the Canonicity Model, thanks to [40]

The final canonicity proof is also interesting, and the diagram is directly copy and paste from [40]. After we constructed the canonicity model. Now we take an arbitrary closed term `t : Tm · B`, and look at its correspondents in the canonicity model[25], and trace the diagram: it means take arbitrary $\gamma \in Hom(1, 1), t[\gamma] \in \{tt, ff\} \subset Hom(1, \mathbb{B})$, and once we take $\gamma = id$ we are done with the canonicity proof.

From this point of the view, the canonicity model is actually mapping each part of the CCC into a set model where each type is mapped to the set of reduced, canonical closed terms[26]. Meanwhile we need to maintain the CCC structure. In our Agda formalization, the set of closed terms is handled by `Con₂, Ty₂` and the corresponding commutativity of CCC (or compatibility lemma) is handled by `Tm₂, Tms₂`.[27]

Note 5.2. However, this is really elementary understanding of categorical gluing. Moreover, it is only for STLC and not yet MLTT. What's more, currently checking that the equations (commutative diagram) still holds in the model of Logical Relation in STLC is a unavoidable burden.

Several fancier ways to present the proof are in [27, 40], where they promise to reduce the effort to recheck all those equations.

6 EPILOGUE

At this stage, we hope everyone with only some knowledge of Coq, Agda, Software Foundations[35] and PLFA[28] can understand most (and maybe all) of the fake Agda code we provide, and come

[25] We don't know what ?? should be but it doesn't matter

[26] which is already hinted in Sec 3.4

[27] There is more to talk about: e.g. the morphism from the glued model to the syntax is usually called section[40], [27] use family/fibration to represent the guarding predicate (e.g. `Tₚ, T₂`)

to realize that QIIT is not even simplifying the mechanization process but also **provide a clearer, easier and more concise understanding** of the structure of the dependent type theory, freed from the "syntax bureaucracy".

After all, the syntax is just a QIIT data, using equations instead of relations indicating dynamics. The semantic is just a delicately designed QIIT model respecting all the equational information, with all kinds of goals (meta-theorems) we want to prove.

This style scales well from STLC to MLTT, but what about those general purposed programming languages or System F and System Fω?

6.1 The Unfulfilled Promises

Unfortunately, the author aimed a lot more topics than the current layout but the time doesn't permit. Let's take a look at these potential topics.

6.1.1 Computational Effect. The most interesting topic must be about computational effect, as it will make the equipped type theory closer to the real world programming language. We have an incomplete state monad example in Sec 2.7.

What's more, modern formulation of computational effect include using universal algebra and Lawvere theory[16]. This directly fits into our picture of the equational theory and thus worth investigating. In this view, monads will act as standard model (for relative-consistency proof).

Moreover, among all the effects, the general recursion is the most intriguing and important one[11, 15, 17, 18, 22, 31]. It is also worth pondering if general recursion can be formulated directly by imposing a fix-point equation[28]. Currently, the two investigated systems are both strong normalizing, and canonicity theorem can be formulated easily – but once non-termination is involved, the motives for canonicity theorem won't hold and we will need the notion of *type-safety* back.

6.1.2 Categorical Gluing. [27, 40, 42] has clearly described the relationship between categorical gluing and logical relation, where categorical gluing can be considered as a generalization of logical relation. In our review, we described clearly the structure of our logical relation and it is actually an algebra for QIIT and its connection to the categorical gluing. However, we fail to think of this from a categorical aspect. The whole point of category theory is to remove boilerplate reasoning, but here every time we come up with a new model, we need to re-verify all the equational property for that model [29]. Syntheic Tait Computability [40] promised a way to avoid these *"avalanche of naturality condition"*; a proof of similar style for canonicity of STLC has been demonstrated in [42].

6.1.3 Impredicativity. The challenge of impredicativity mainly comes from the fact that, Agda is predicative, thus it is not easy for us to formalize and prove corresponding theorem for System F in this framework[33, 41]. Possible references include [6].

6.1.4 Modality. The interesting thing about modality is purely because it has a non-trivial interaction with substitution and it has a wild range of application, including substructural type theory (resource management)[7], contextual modal type (well-typed metaprogramming) [32] also a formulation of general recursion[15].

REFERENCES

[1] [n. d.]. https://ncatlab.org/nlab/show/Lindenbaum-Tarski+algebra

[28] Apparently not trivially because we cannot even construct standard model for that formulation

[29] though if properly formalized, Agda should be able to verify these for us

[2] [n. d.]. https://cs.stackexchange.com/questions/54872/how-precise-is-the-statement-stlc-is-the-internal-language-of-cccs

[3] [n. d.]. https://twitter.com/ltchen_tw/status/1514536058981339138

[4] [n. d.]. https://ncatlab.org/nlab/show/closed+monoidal+structure+on+presheaves

[5] 2019. https://en.wikipedia.org/w/index.php?title=Logical_framework&oldid=913088895 Page Version ID: 913088895.

[6] Andreas Abel. 2013. Normalization by evaluation: Dependent types and impredicativity. *Habilitation. Ludwig-Maximilians-Universität München* (2013).

[7] Andreas Abel and Jean-Philippe Bernardy. 2020. A unified view of modalities in type systems. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–28.

[8] Amal Ahmed. 2006. Step-indexed syntactic logical relations for recursive and quantified types. In *European Symposium on Programming*. Springer, 69–83.

[9] Amal Ahmed. 2013. Logical relations. *Oregon Programming Languages Summer School (OPLSS)* (2013).

[10] Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. 2018. Quotient inductive-inductive types. In *International Conference on Foundations of Software Science and Computation Structures*. Springer, Cham, 293–310.

[11] Thorsten Altenkirch, Nils Anders Danielsson, and Nicolai Kraus. 2017. Partiality, revisited. In *International Conference on Foundations of Software Science and Computation Structures*. Springer, 534–549.

[12] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. 1995. Categorical reconstruction of a reduction free normalization proof. In *International Conference on Category Theory and Computer Science*. Springer, 182–199.

[13] Thorsten Altenkirch and Ambrus Kaposi. [n. d.]. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Vol. 2016. Association for Computing Machinery (ACM).

[14] Nathanael Arkor and Marcelo Fiore. 2020. Algebraic models of simple type theories: A polynomial approach. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*. 88–101.

[15] Robert Atkey and Conor McBride. 2013. Productive coprogramming with guarded recursion. *ACM SIGPLAN Notices* 48, 9 (2013), 197–208.

[16] Andrej Bauer. 2018. *What is algebraic about algebraic effects and handlers?*

[17] Venanzio Capretta. 2005. General recursion via coinductive types. *Logical Methods in Computer Science* 1 (2005).

[18] Venanzio Capretta, Thorsten Altenkirch, and Tarmo Uustalu. 2005. Partiality is an effect. In *Slides for a talk given by Uustalu at the 22nd meeting of IFIP Working Group*, Vol. 2.

[19] John Cartmell. 1986. Generalised algebraic theories and contextual categories. *Annals of pure and applied logic* 32 (1986), 209–243.

[20] Jesper Cockx. 2020. Type theory unchained: Extending Agda with user-defined rewrite rules. In *25th International Conference on Types for Proofs and Programs (TYPES 2019)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[21] Thierry Coquand. 2018. Canonicity and normalisation for dependent type theory. *arXiv preprint arXiv:1810.09367* (2018).

[22] Daniel P Friedman and Amr Sabry. 2000. Recursion is a Computational E ect. *Unpublished. Available at http://www. cs. uoregon. edu/~ sabry/papers* (2000).

[23] Daniel Gratzer, GA Kavvos, Andreas Nuyts, and Lars Birkedal. 2020. Multimodal dependent type theory. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*. 492–506.

[24] Robert Harper. 2016. *Practical foundations for programming languages*. Cambridge University Press.

[25] Bart Jacobs. 1992. Simply typed and untyped lambda calculus revisited. *Applications of Categories in Computer Science, LMS Lecture Note Series* 177 (1992), 119–142.

[26] Ambrus Kaposi. 2017. *Type theory in a type theory with quotient inductive types.* Ph. D. Dissertation. University of Nottingham.

[27] Ambrus Kaposi, Simon Huber, and Christian Sattler. 2019. Gluing for type theory. In *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[28] Wen Kokke, Jeremy G Siek, and Philip Wadler. 2020. Programming language foundations in Agda. *Science of Computer Programming* 194 (2020), 102440.

[29] Joachim Lambek and Philip J Scott. 1988. *Introduction to higher-order categorical logic*. Vol. 7. Cambridge University Press.

[30] luqui. 2010. What is the manner of inconsistency of Girard's paradox in Martin Lof type theory. https://mathoverflow.net/q/18089/479677

[31] Eugenio Moggi. 1991. Notions of computation and monads. *Information and computation* 93, 1 (1991), 55–92.

[32] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Transactions on Computational Logic (TOCL)* 9, 3 (2008), 1–49.

[33] Max New. 2015. Logical Reations for an Impredicative System in a Predicative MetaTheory. https://cstheory.stackexchange.com/q/32859/40632
[34] Fredrik Nordvall Forsberg and Anton Setzer. 2010. Inductive-inductive definitions. In *International Workshop on Computer Science Logic*. Springer, 454–468.
[35] Benjamin C Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2010. Software foundations. *Webpage: http://www. cis. upenn. edu/bcpierce/sf/current/index. html* (2010).
[36] Andrew M Pitts. 1998. Existential types: Logical relations and operational equivalence. In *International Colloquium on Automata, Languages, and Programming*. Springer, 309–326.
[37] The Univalent Foundations Program. 2013. Homotopy type theory: univalent foundations of mathematics. *arXiv preprint arXiv:1308.0729* (2013).
[38] Lau Skorstengaard. 2019. An introduction to logical relations. *arXiv preprint arXiv:1907.11133* (2019).
[39] JONATHAN STERLING. [n. d.]. ALGEBRAICTYPETHEORY& THEGLUINGCONSTRUCTION. ([n. d.]).
[40] Jonathan Sterling. 2021. *First Steps in Synthetic Tait Computability*. Ph. D. Dissertation. University of Minnesota.
[41] Jonathan Sterling and Robert Harper. 2021. Logical Relations as Types: Proof-Relevant Parametricity for Program Modules. *J. ACM* 68, 6 (Oct. 2021). https://doi.org/10.1145/3474834 arXiv:2010.08599 [cs.PL]
[42] Jonathan Sterling and Bas Spitters. 2018. Normalization by gluing for free {\lambda}-theories. *arXiv preprint arXiv:1809.08646* (2018).
[43] Thomas Streicher. 2006. *Domain-theoretic foundations of functional programming*. World Scientific Publishing Company.
[44] Taichi Uemura. 2019. A general framework for the semantics of type theory. *arXiv preprint arXiv:1904.04097* (2019).
[45] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2021. Cubical Agda: A dependently typed programming language with univalence and higher inductive types. *Journal of Functional Programming* 31 (2021).
[46] Philip Wadler. 2018. Programming language foundations in agda. In *Brazilian Symposium on Formal Methods*. Springer, 56–73.